Randomized Data Structures: New Perspectives and Hidden Surprises

by

William Kuszmaul

B.S., Stanford University (2018)
S.M., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2023

© 2023 William Kuszmaul. All rights reserved.
The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: William Kuszmaul
Department of Electrical Engineering and Computer Science
August 30, 2023

Certified by: Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Randomized Data Structures: New Perspectives and Hidden Surprises

by

William Kuszmaul

Submitted to the Department of Electrical Engineering and Computer Science
on August 30, 2023, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis revisits some of the oldest and most basic questions in the theory of randomized data structures—questions such as: *How efficient is a linear probing hash table? How fast can you maintain a sorted array of numbers? How big does a pointer have to be?* With the help of new techniques, along with a willingness to look beyond conventional wisdom, we are able to achieve much stronger bounds for each of these questions than were previously thought to be possible.

Our results also come with a powerful set of tools that span a wide range of problems and settings. Perhaps the most surprising of these tools is a new paradigm for designing efficient dynamic data structures, in which, by '*tying our hands behind our back*' (i.e., by artificially restricting ourselves to a special class of privacy-preserving data structures), we are able to circumvent decades-old barriers in time/space efficiency. This technique appears three (completely separate) times in the thesis.

Combined, our results overturn a 60-year-old myth on linear-probing hash tables; refute a 30-year-old conjecture and solve a 40-year-old open problem on dynamic sorting; resolve a 20-year-old open problem on dynamic load balancing; settle some of the most basic and fundamental questions from the theory of space-efficient data structures; and answer a 20-year-old question on memory allocation that was left as the central open problem in the first paper on history independence.

Part I. Linear Probing Revisited: Overturning the Oldest Myth in Data Structures. First introduced in 1954, the *linear-probing hash table* is among the most influential data structures in computer science. Thanks to its unrivaled data locality, it remains one of the fastest and most widely used hash tables today.

But linear probing also comes with a major drawback: as elements are inserted into the hash table, they have a tendency to cluster together into long runs. This *clustering effect*—which was first discovered by Donald Knuth in 1963—causes the time per insertion to explode as the hash table fills up.

For nearly six decades, Knuth's clustering result has shaped how researchers and practitioners think about linear probing. Many courses and textbooks even teach Knuth's exact formula: that for a hash table at $(1 - 1/x)$ full, the expected number

of probes per insertion is $\approx (x^2 + 1)/2$. This simple formula has caused generations of hash-table designers to avoid linear probing for space-sensitive applications.

In Part I we consider a simple question: what if we could somehow reduce clustering? What we discover is much more surprising: the classical linear-probing hash table *already* exhibits far less clustering than the classical analysis suggests.

We prove that, as insertions and deletions are performed over time, the structure of the hash table stabilizes in a way that reduces clustering. If certain design decisions are made correctly, then the *amortized* expected time per operation drops all the way to $\tilde{O}(x)$. Taking these ideas further, we also introduce a new version of linear probing, called graveyard hashing, that avoids clustering entirely. At $1 - 1/x$ full, graveyard hashing achieves $O(x)$ expected time for every operation.

Part II. Dynamic Sorting Revisited: The Power of History Independence. The *dynamic sorting problem* (a.k.a. the *list-labeling problem* or the *packed-memory array problem*) considers the following basic question: can you store n elements in *sorted order* in an array of size $O(n)$, while supporting insertions/deletions cheaply.

A priori, the answer might seem to be no—a single insertion/deletion could force many elements in the array to be moved. However, a famous result by Itai et al. in 1981 shows that this intuition is false: the dynamic sorting problem can actually be solved in $O(\log^2 n)$ amortized time per insertion/deletion.

Despite a great deal of additional work, by both theoreticians and practitioners, *the $O(\log^2 n)$ bound remained the state of the art for more than 40 years.* In the 1990s, it was conjectured that the bound should be optimal—and over the course of three decades, researchers obtained lower bounds applying to larger and larger classes of data structures. However, an all-encompassing lower bound has remained elusive.

We present a randomized data structure that performs dynamic sorting in $O(\log^{1.5} n)$ expected time per insertion/deletion. This represents the first asymptotic improvement to dynamic sorting since it was first studied in 1981, and disproves a 30-year-old conjecture as to the optimality of the $O(\log^2 n)$ bound.

Part III. Balls and Bins: When Greedy Allocation Fails and How to Fix It. Next, we turn our attention to a technical phenomenon known as *the power of two choices*, which in recent decades has been applied not just to data structures, but more broadly to problems in scheduling, distributed computing, etc.

The power-of-two-choices result considers the following basic setting. Consider a set of m balls that are placed into n bins, one after another. Each ball x must choose between two random bins $h_1(x)$ and $h_2(x)$. The goal is to end in a state where the bins have almost exactly equal loads of $\approx m/n$.

In one of its most general forms, what the classical *power-of-two-choices* result states is that: if each ball *greedily* selects the emptier of its two bin options, then the outcome will be remarkably balanced. Each bin will contain almost exactly m/n balls, up to $\pm O(\log m)$, with high probability in m.

The power-of-two-choices result has had tremendous impact on how both theoreticians and practitioners think about load-balancing. But the result comes with a significant weakness: it is restricted to the *arrival-only setting*.

For more than two decades, it remained an open question whether the same result holds in the *dynamic setting*, where balls arrive and depart over time with up to m balls present at once. It was widely believed that the same result *should* hold, but it was only known for special cases—where $m = n$ or where departures are stochastic.

In Part III, we give a surprising resolution to this problem: the reason that the dynamic case has been so hard to analyze is because *the result is false*. Greedily picking the less empty of two random bins actually does poorly, allowing for some bins to become significantly overloaded over time. But an alternative approach *does do well*: the key is to be strategically (and randomly) *non-greedy* in one's decisions. By using randomization not just to select the two bins, but also to choose between them, one can once again achieve strong bounds.

We present variations of these results that can be applied both to scheduling and to data-structural settings. Indeed, the data-structural variation serves as a key technical ingredient for several of the space-efficient data structures later in the thesis.

Part IV. Hashing it Out: Some Barriers Are Fundamental and Others Are Not. In Part IV we revisit three widely studied problems from the field of hashing. Each of these problems comes with a well-known barrier that has prevented progress. But it has remained open whether these barriers are fundamental.

Our first result resolves a 60-year-old question on the optimal space efficiency of hash tables. For more than two decades, the state of the art has been a hash table that, when compared to the information-theoretic optimum, uses $O(\log \log n)$ extra bits of space per key. In recent decades, the $O(\log \log n)$ bound has been shown to be optimal for several restricted versions of the problem, but it has remained unknown whether the bound is optimal in general. We show that, perhaps surprisingly, it is not: we reduce the extra bits per key to

$$O(\log^{(k)} n) = O\left(\underbrace{\log \log \cdots \log}_{k} n\right)$$

for any $k = O(1)$. More generally, for $k = \omega(1)$, the same bound can be achieved with an insertion time of $\Theta(k)$. It was subsequently shown by Li, Liang, Yu, and Zhou that our space/time tradeoff is the optimal one for *any dynamic data structure.*

Our second result resolves an open question on hash-table failure probabilities: how small of a failure probability can a hash table, storing $\Theta(\log n)$-bit keys, offer? Past work on this question has encountered a surprising bottleneck: If we assume access to fully random hash functions, then it is possible to achieve very strong probabilistic guarantees, but if the same hash tables are implemented using the known families of hash functions, it is unknown how to achieve a failure probability better than $1/2^{\operatorname{polylog} n}$. To get around these obstacles, we show how to construct a random-

ized data structure that has the same guarantees as a hash table, but that *avoids the direct use of hash functions*. This allows us to achieve failure probability $1/n^{n^{1-\varepsilon}}$, which represents a nearly exponential improvement over the previous state of the art.

Our third result is a tight lower bound for the problem of monotone minimal perfect hashing. Simply stated, this problem captures the task of constructing a *static* data structure that can be used to query, for some set $S \subseteq [U]$ of n elements, the *rank* $|\{y \in S \mid y \leq x\}|$ of each element $x \in S$. Belazzougui, Boldi, Pagh, and Vigna showed in 2009 that this problem can be solved with space $O(n \log \log \log u)$ bits. It has remained open whether this somewhat unusual bound can be improved. We show that, somewhat surprisingly, the bound is optimal, not just for time-efficient data structures but for any information-theoretic solution to the problem.

Part V. How Many Bits Does It Take to Write Down a Pointer? One type of data structure that is typically *not* viewed as space efficient, is any data structure that makes heavy use of pointers. Examples include chained hash tables, binary search trees, stable dictionaries, internal-memory stashes, etc. In these data structures, the pointers between elements consume a significant fraction of the total space.

We introduce a new data-structural object called the *tiny pointer* that, using randomization, allows us to compress $\log n$-bit pointers in many applications to be $o(\log n)$ bits. Interestingly, the key to constructing optimal tiny pointers is to carefully make use of our balls-and-bins techniques from Part III.

We apply tiny pointers to five classical problems in data structures, obtaining significant improvements in dynamic data retrieval, space-efficient search trees, stable dictionaries, key-value stores with variable-size values, and external-memory stashes.

Our pointer-compression techniques have also been applied to computer hardware design. Here, tiny pointers are used to obtain a new, more effective, design for hardware TLBs—this result won Distinguished Paper at ASPLOS'23.

Part VI. A Strong Theory of Strong History Independence. An important technical theme in several parts of this thesis is the surprising role of *history-independence* as a mechanism for bypassing classical data-structural bottlenecks. At the same time, basic questions about history independence have remained open for decades—questions involving memory allocation/reallocation, space-efficient hashing, and worker-task assignment.

Part VI gives nearly optimal solutions to each of these problems. Our results on memory reallocation resolve the central open question posed by Naor and Teague in their 2001 paper on history independence. The result gives a new state of the art not just for history-independent solutions, but even for *non*-history-independent ones.

Our algorithms come with nearly matching lower bounds—these are the first super-constant lower bounds to be achieved for any of the aforementioned problems.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Acknowledgments

I was first introduced to research as a high-school freshman, when Pavel Etingof and David Jordan offered me, along with Surya Bhupatiraju and Jason Li, a research project as part of the MIT PRIMES program. It is impossible for me to overstate the significance that MIT PRIMES had in my development as a researcher. It was where, under the guidance of Darij Grinberg and Sergei Bernstein, I discovered my love for combinatorics. And far more importantly, it was where I learned how to identify a good research problem, and how to persist until I could solve it. I am deeply grateful to Pavel Etingof, David Jordan, Darij Grinberg, and Sergei Bernstein for their mentoring.

Later, during my undergraduate years, I continued to be blessed with extraordinary mentors. I am incredibly grateful to Joe Gallian, who mentored me during the summer after freshman year, and to Moses Charikar, Ofir Geri, and Michael P. Kim, who met with me weekly during my Junior and Senior years and taught me how to be an algorithms researcher.

Several of my undergraduate instructors had profound effects on my interests, and ultimately on this thesis. These include John Ousterhout, Gregory Valient, Virginia V. Williams, and Julie Zelenski. I am especially indebted to Virginia Williams, whose incredible course on graph algorithms convinced me that I wanted to be an algorithmist in my freshman year, to Keith Schwartz, whose brilliant course on data structures permanently changed the trajectory of my research, and to Yufei Zhao who generously met with me on-one-one to teach me the probabilistic method during my study abroad at Oxford.

During the summer before my PhD, I began my first collaboration with Michael A. Bender and Martín Farach-Colon, both of whom would go on to become among my most frequent collaborators. I am also grateful to my many other collaborators, including Kunal Agrawal, Sepehr Assadi, Nikhil Bansal, Aaron Berger, Abhishek Bhattacharjee, Moses Charikar, Rezaul A Chowdhury, Alex Conway, Rathish Das, Daniel DeLayo, Martín Farach-Colton, Krishnan Gosakan, Jayneel Gandhi, Ofir Geri, Jaehyun Han, Rob Johnson, Timothy G. Kaler, Sudarsun Kannan, Michael P. Kim, Hanna Komlós, Tsvi Kopelowitz, Bradley C. Kuszmaul, John Kuszmaul, Charles E. Leiserson, Andrea Lincoln, Mingmou Liu, Quanquan C Liu, Jayson Lynch, Tao B. Schardl, Clifford Stein, Ibrahim N. Mubarek, Nirjhar Mukherjee, Shyam Narayanan, Enoch Peserico, Prashant Pandey, Seth Pettie, Adam Polak, Ely Porat, Donald E. Porter, Michele Scquizzato, Karthik Sriram, Guido Tagliabini, Jonathan Tidor, Nicole Wein, Evan West, Alek Westover, David P. Woodruff, Zoe Xi, Helen Xu, and Lin F. Yang; and to the many fantastic high-school, undergraduate, and masters students that I got work with during my PhD, including Kevin Chang, Michael Ma, Alek Westover, Zoe Xi, Qi Qi, Emily Liu, and Wanlin Li.

I spent the final four years of my PhD living in F Entry of MacGregor House, where I had the pleasure of serving as the residential advisor for an amazing group of undergraduates. I'm grateful to Terry and Larry Sass, to Charlie McBurney, and to the amazing residents of F Entry for making my stay there so special.

The most important moment of my PhD was the winter evening when I met my (now) partner Rose Silver at Toscanini's Ice Cream. Rose has been my greatest cheerleader, advisor, and friend for the past three and a half years.

I am unendingly grateful to my family: My parents Dana Henry-Kuszmaul and Bradley Kuszmaul; my siblings Anne Sullivan, Elizabeth Kuszmaul, John Kuszmaul, and Margaret Kuszmaul; my grandmother Alena Hadju; my siblings-in-law Stephanie Frankian and Ty Sullivan; and my teeny-tiny niece Suzie Sullivan.

Most significantly, I am grateful to my advisor Charles E. Leiserson for offering me advice and mentoring at every point in my PhD, and for always letting me try things my way even when they were a bad idea. I could not have asked for a better advisor-advisee relationship. I am also grateful to the other members of my thesis committee, Michael A. Bender and Virginia V. Williams.

Additionally, I would like to thank the John and Fannie Hertz Foundation for their support during my PhD, and for giving me (and Charles) financial freedom in my pursuits. Additionally, I would like to think the NSF GRFP program for their fellowship support and to acknowledge the support that I received from NSF grants CCF 1314547 and CCF 1533644; as well as grants from the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator[1].

[1]Parts of this thesis were accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

To my family, Bradley, Dana, Anne, Elizabeth, John, Margaret, and Alena

To my partner Rose

Contents

1 Introduction **17**

1.1 Randomized Data Structures: Past and Present 17

1.2 Contributions and Outline 18

1.3 Bibliographics 30

I Linear Probing Revisited: Overturning the Oldest Myth in Data Structures **36**

2 Introduction **37**

2.1 Notation and Conventions 46

3 The Surprisingly Strong Anti-Clustering Effects of Tombstones **49**

3.1 Technical Overview 50

3.1.1 Understanding the Classic Bounds: A Tale of Standard Deviations 51

3.1.2 Analyzing Primary Anti-Clustering with Small Rebuild Windows 52

3.1.3 Stronger Primary Anti-Clustering with Larger Rebuild Windows 57

3.2 Some Basic Balls-and-Bins Lemmas 59

3.3 Bounds on Insertion Surplus 63

3.3.1 Proof of Proposition 9 65

3.3.2 Proof of Proposition 10 67

3.4 Relating Insertion Surplus to Crossing Numbers 70

3.5 Relating Crossing Numbers to Running Times 78

4 Graveyard Hashing, an Ideal Linear-Probing Hash Table **85**

II Dynamic Sorting Revisited: The Power of History Independence **90**

5 Introduction **91**

5.1 Preliminaries and Technical Background 94
5.1.1 The Classical Solution and its History-Independent Analogue . 95
5.2 Related work 97

6 Breaking the $O(\log^2 n)$ Barrier 101
6.1 Technical Overview 101
6.2 Zeno's Random Walk 103
6.3 The Zeno Embedding: a Data Structure for $m \geq (1+\varepsilon)n$ 105
6.3.1 The Static Zeno Embedding 106
6.3.2 Dynamizing the Zeno Embedding 109
6.3.3 Achieving a Bound on Arbitrary Insertions/Deletions. 112
6.4 Upper Bound For Sparse Arrays 115

7 A Lower Bound for History-Independent Solutions 119
7.1 Proof of Proposition 61 121
7.2 Proof of Proposition 63 124

III Balls and Bins: When Greedy Allocation Fails and How to Fix It 128

8 Introduction 129
8.1 Chapter 9. The Scheduling Perspective 131
8.2 Chapter 10. The Data-Structural Perspective 133
8.3 Preliminaries 134
8.4 Other Related Work 135

9 The Scheduling Perspective 137
9.1 A Lower Bound for Greedy with Deletions 138
9.1.1 A Simpler $\Omega(m^{1/4})$ Bound 138
9.1.2 The Stronger $\Omega(m^{1/2})$ Lower Bound 140
9.2 ModulatedGreedy: Handling poly(m) Insertions/Deletions 144
9.2.1 The Algorithm 145
9.2.2 Analysis 146
9.2.3 Tightness of the Bound 149
9.3 Generalizations of ModulatedGreedy 150
9.3.1 The Algorithm and Overview 150
9.3.2 Algorithm Analysis 152

9.3.3 Extensions 156

Appendices **159**

9.A Proof of Lemma 74 159

9.B Proof of Lemma 100 160

10 The Data-Structural Perspective **163**

10.1 An Impossibility Result For The Deletions with Reinsertions 164

10.1.1 The Marble-Splitting Game 164

10.1.2 Proof of Theorem 97 167

10.2 An Upper Bound for the Moderately-Loaded Regime 171

10.3 A Strong Backyarding Lemma 171

10.3.1 The Iceberg 3-Choice Strategy 173

IV Hashing it Out: Some Barriers Are Fundamental and Others Are Not **175**

11 Introduction **177**

12 The Optimal Space-Time Tradeoff Curve for Hash Tables **179**

12.1 Introduction 180

12.2 Overview of Results and Techniques 181

12.2.1 Hash Tables and Balls-To-Slots Schemes 182

12.2.2 Transforming a k-Kick Tree into a k-Kick Hash Table 184

12.2.3 An Application to Optimal Dynamic Filters 186

12.2.4 Preliminaries 187

12.3 The Probe-Complexity Problem 189

12.3.1 A Balls-to-Slots Scheme with Small Average Probe Complexity 190

12.3.2 A Lower Bound on Switching Cost vs. Probe Complexity . . . 195

12.4 Encoding Metadata in an Augmented Open-Addressed Hash Table . . 209

12.4.1 Preliminaries: The Lookup-Table Technique 210

12.4.2 Storing a Mini-Array of Variable-Size Values 211

12.4.3 Query Routers 212

12.5 An Optimal Augmented Open-Addressed Hash Table 214

12.5.1 Turning the k-Kick Tree into a Hash Table 215

12.5.2 Supporting Dynamic Resizing 218

12.5.3 Succinctness Through Quotienting 222

12.6 Large Keys, Small Keys, and Filters 226
12.6.1 Supporting Large Keys/Values 226
12.6.2 Optimizing for Very Small Keys 231
12.6.3 Constructing Optimal Filters 237

13 A Hash Table Without Hash Functions 241
13.1 Introduction 242
13.2 Preliminaries and Conventions (for Non-Succinct Dictionaries) 245
13.3 A Warmup Data Structure: The Rotated Trie. 246
13.4 The Amplified Rotated Trie 249
13.5 The Budget Rotated Trie 252
13.6 Achieving Succinctness 256
13.6.1 Reduction to the Many-Sets Problem 257
13.6.2 Proof of Theorem 148 260

Appendices 267
13.A Universe Reduction Using $O(\log n)$ Random Bits 268

14 Tight Bounds for Monotone Minimal Perfect Hashing 271
14.1 Introduction 272
14.2 Preliminaries 275
14.2.1 Problem Definition and Model of Computation 275
14.2.2 Fractional Colorings 275
14.3 A Lower Bound for MMPHF via Fractional Colorings 276
14.3.1 Conflict Graph and its Fractional Chromatic Number 277
14.3.2 Extending the MMPHF Lower Bound to Small Universes . . . 279
14.4 Fractional Chromatic Number of Conflict Graphs 280
14.4.1 A High-Level Overview of the Proof 281
14.4.2 The Hard Input Distribution in Lemma 166 282
14.4.3 Analysis of the Hard Distribution (and Proof of Lemma 166) . 285

Appendices 297
14.A Proofs of Standard Results in Fractional Coloring 298
14.B Covering The Full Range of the Universe Size 300

V How Many Bits Does It Take to Write Down a Pointer? 302

15 Introduction 303

16 From Balls and Bins to Tiny Pointers **307**
16.1 Preliminaries 308
16.2 Upper Bound for Fixed-Size Pointers 310
16.3 Upper Bounds for Variable-Sized Pointers 314
16.4 Lower Bounds 321

17 Five Applications to Data Structures **325**
17.1 Some General-Purpose Techniques for Using Tiny Pointers 329
17.2 Overcoming the $\Omega(\log\log n)$-Bit Lower Bound for Data Retrieval . . . 332
17.3 Succinct Binary Search Trees 335
17.4 Space-Efficient Stable Dictionaries 336
17.5 Space-Efficient Dictionaries with Variable-Size Values 337
17.6 An Optimal Internal-Memory Stash 340

VI A Strong Theory of Strong History Independence **344**

18 Introduction **345**
18.1 Conventions 349

19 Strong Upper Bounds for Stateless Allocation **351**
19.1 Achieving $O(1 + \log \varepsilon^{-1})$ Expected Overhead 354
19.1.1 Achieving $O(1 + \log \varepsilon^{-1})$ Overhead with Power-of-Two Sizes . 354
19.1.2 Supporting Dynamic Resizing 356
19.1.3 Allocating Items with Sizes in $[1, 2)$ 360
19.1.4 Allocating Objects of Arbitrary Sizes 362
19.2 Achieving polylog n Worst-Case Overhead for Worker-Task Assignment 364
19.2.1 Achieving Switching Cost $O(\log w \log(wt))$ 365
19.2.2 Derandomizing the Construction 372

20 Strong Lower Bounds for Stateless Allocation **375**
20.1 Proof of Theorem 223 375

21 Efficient Data-Structural Implementations **385**
21.1 An Efficient Allocator for Small Objects 386
21.2 An Efficient Allocator for Large Objects 392
21.3 Putting the Pieces Together 395

Chapter 1

Introduction

1.1 Randomized Data Structures: Past and Present

In 1952, IBM completed the design for the IBM 701, their first commercial scientific computing machine. Almost immediately, it became clear that the efficiency of the machine would depend not just on its hardware but on the algorithms and data structures implemented within it. In the following two years, the concept of a hash table was invented not once, but twice (*independently*!), by researchers at IBM [226]. First, by Hans Peter Luhn in 1953, who invented the chained hash table, and then again by Gene Amdahl, Elaine Boehme, and Arthur Samuel, who invented the linear-probing hash table while developing an assembly program for the IBM 701 [226]. The almost instantaneous invention of hash tables is remarkable when compared to the evolution of other ideas in algorithms and data structures. Hans Peter Luhn's implementation of chained hashing appears to also be the earliest recorded computational use of a *linked list* [226]. The modern notion of a binary search tree wouldn't be described until 1959 [164, 226], and the balanced binary search tree would remain undiscovered until 1962 [41, 226]. But hash tables were discovered immediately—efficient computing would have been almost impossible without them.

A few years later, in 1963, a young mathematician named Donald Knuth wrote a paper that changed the course of computing. Knuth used what (even by today's standards) was a relatively sophisticated probabilistic analysis [224] in order to show that linear-probing hash tables behave very differently from how engineers in the 1950s had theorized [309].[1] The phenomenon that Knuth had discovered, now known as *primary clustering*, continues to be taught to undergraduates today [131, 165, 201, 230, 241, 253, 257, 323, 324, 332, 343, 353]. (And, in fact, showing how to *eliminate* primary clustering will be the first part of this thesis.)

Knuth took an almost bashful tone in his note, explaining why he (a reputable mathematician) was writing about a simple problem having to do with storing data. Nonetheless, the note marked a central point in Knuth's career—it was while writing

[1]Knuth's result [224] was never formally published and was subsequently rediscovered by Konheim and Weiss in 1966 [228].

it that he decided to pivot to the study of algorithms as a field [225]. Today he is widely regarded as the father of modern algorithms.

In the half-century since Knuth's paper, randomization has become an essential tool in any algorithm designer's toolbox. Nonetheless, several of the most basic and fundamental questions concerning randomized algorithms, and specifically randomized data structures, have remained either unanswered or (as we shall see) answered with incorrect folklore. This is the problem that we seek to rectify in this thesis.

This Thesis: New Perspectives and Hidden Surprises. In this thesis, we use modern perspectives on randomization to revisit core questions in the field of data structures—questions such as: *How efficient is a linear probing hash table? How fast can you maintain a sorted array of numbers? How big does a pointer have to be?*

In many cases, these are problems that were thought to have hit fundamental barriers decades ago. We will see, however, that even for these problems, there are often still hidden surprises waiting to be discovered. To unlock these surprises, we will need two things: a willingness to disregard classical conventional wisdom, and a new suite of algorithmic/probabilistic techniques.

As we shall see, our results come with strong technical strands that tie problems and solutions together across a wide range of settings. Perhaps the most surprising of these is a new paradigm for designing efficient dynamic data structures, in which by 'tying our hands behind our back' (i.e., by artificially restricting ourselves to a special class of privacy-preserving data structures), we are repeatedly able to circumvent decades-old barriers in time/space efficiency. This technique shows up independently in three different parts of the thesis.

1.2 Contributions and Outline

The thesis consists of six parts. Each part revisits a core problem (or, in some cases, a collection of closely related problems) in data structures. What we will see in each case is that, by really understanding the randomized processes that underlie these problems, and by adopting new perspectives, we can establish results that go far beyond what was previously known.

Combined, our results overturn a 60-year-old myth on linear-probing hash tables (Part I); refute a 30-year-old conjecture and solve a 40-year-old open problem on dynamic sorting (Part II); resolve a 20-year-old open question on dynamic load balancing (Part III); settle some of the most basic and fundamental questions from the theory of space-efficient data structures (Parts IV and V); and answer a 20-year-old question on memory allocation that was left as the central open problem in the first paper on history independence (Part VI).

Although our focus is primarily theoretical, many of the techniques and ideas from this thesis are also beginning to appear in practical settings, including in my work on high-performance hash tables [11] and on improved designs of hardware TLBs [10]

(Distinguished Paper at ASPLOS'23).

The thesis contains only a portion of the work from my PhD, but many of the themes and techniques in the thesis also appear in my other work [10–33]. Examples include my work on randomized buffer flushing [15, 18, 20, 22, 27, 28, 30], network contention resolution [26], parallel scheduling [14, 21], compiler instrumentation [29], etc. The papers that I have written during my PhD have won several awards, including Distinguished Paper at ASPLOS'23, Best Student Paper at ESA'22, Best Paper Finalist at SPAA'22, Best Paper at FUN'20, and Best Paper Finalist at APOCS'20. I have also had the fantastic experience of advising many brilliant students, including four high-school students whose projects won national awards.[2]

Part I. Linear Probing Revisited: Overturning the Oldest Myth in Data Structures

First introduced in 1954 [226], the linear-probing hash table is among the oldest data structures in computer science, and thanks to its unrivaled data locality, linear probing continues to be one of the fastest hash tables in practice. It is widely believed and taught, however, that linear probing should never be used at high load factors[3]; this is because of an effect known as *primary clustering* which causes insertions at a load factor of $1 - 1/x$ to take expected time $\Theta(x^2)$ (rather than the intuitive running time of $\Theta(x)$). The dangers of primary clustering, first discovered by Knuth in 1963 [224], have now been taught to generations of computer scientists, and have influenced the design of some of the most widely used hash tables in production [38, 173].

Chapter 3. The Surprisingly Strong Anti-Clustering Effect of Tombstones. In Chapter 3, we show that primary clustering is not the foregone conclusion that it is reputed to be. We demonstrate that seemingly small design decisions in how deletions are implemented have dramatic effects on the asymptotic performance of insertions: if these design decisions are made correctly, then even if a hash table operates continuously at a load factor of $1 - \Theta(1/x)$, the expected amortized cost per insertion/deletion is $\tilde{O}(x)$.

The key insight that makes our result possible is that the *tombstones* left behind by deletions actually cause an *anti-clustering* effect that combats primary clustering. What's remarkable is just how strong the anti-clustering effect ends up being. Even if insertions and deletions are one-for-one, the anti-clustering effect of the deletions ends up *overpowering* the clustering effect of insertions. In fact, the clustering effects of insertions are *almost completely eliminated*, yielding an amortized expected $\tilde{O}(x)$ time bound per operation.

[2]These include Kevin Chang who won the Regeneron STS Semifinalist Award; Michael Ma who won the \$25,000 Regeneron STS Finalist Award; Alek Westover who won the \$70,000 Regeneron STS 7-th place award; and Zoe Xi who won the \$25,000 Regeneron STS Finalist award, along with the ESA'22 Best-Student-Paper award.

[3]*Load factor* is a term used to refer to how full a hash table is: a load factor of α means that an α-fraction of slots in the hash table are occupied.

What the $\tilde{O}$ notation means is that the true upper bound is of the form $O(xf(x))$ for some $f(x) \leq \text{polylog}(x)$. Perhaps surprisingly, we show that this low-order term is real—even for a basic *hovering workload*, which alternates between insertions/deletions, the amortized time complexity per operation is $\omega(x)$. This raises a natural question: can linear probing be modified to achieve the ideal bound of $O(x)$?

Chapter 4. Graveyard Hashing, an Ideal Linear-Probing Hash Table. In Chapter 4, we present a new version of linear probing, which we call *graveyard hashing*, that completely eliminates primary clustering on any sequence of operations: if, when an operation is performed, the current load factor is $1 - 1/x$ for some x, then the expected cost of the operation is $O(x)$. This bound holds even for insertion-only workloads, and even in hash tables that are frequently dynamically resized in order to keep their load factors high.

Graveyard hashing draws on the basic lesson from the previous chapter—that *tombstones* are an unexpected force for good. Taking this lesson one step further, graveyard hashing *artificially injects tombstones into the hash table, without waiting for them to be caused by deletions.* Doing this in the right way, we can get the full anti-clustering effects of tombstones on every operation, no matter the workload.

Our results overturn one of the most widely-taught conventional wisdoms in the field of data structures, and in doing so, yield the first asymptotic improvement to linear probing since its insertion time was first analyzed in 1963. The results also act as a roadmap for practitioners, giving insight into which engineering decisions affect the asymptotics of linear probing.

Part II. Dynamic Sorting Revisited: The Power of History Independence

The *dynamic sorting problem* (a.k.a. the *list-labeling problem* or the *packed-memory array problem*) considers the following basic question: can you store n elements in *sorted order* in an array of size $O(n)$, while supporting insertions/deletions cheaply?

A priori, the answer might seem to be no—a single insertion/deletion in the tree could force many elements in the array to be moved. However, a famous result due to Itai et al. in 1981 [214] shows that this intuition is false: the dynamic sorting problem can be solved in $O(\log^2 n)$ amortized time per insertion.

Despite a great deal of additional work, by both theoreticians and practitioners, *the $O(\log^2 n)$ bound remained the state of the art for more than 40 years.* In the 1990s, it was conjectured [150–152] that the bound should be optimal—and over the course of three decades, researchers obtained lower bounds applying to larger and larger classes of data structures [120, 150–152]. However, an all-encompassing lower bound remained elusive.

Chapter 6. Breaking the $O(\log^2 n)$ Barrier. In Chapter 6, we present a randomized data structure that performs dynamic sorting in $O(\log^{1.5} n)$ expected time per insertion/deletion. This represents the first asymptotic improvement to dynamic sorting since it was first studied in 1981, and disproves a 30-year-old conjecture as to the optimality of the $O(\log^2 n)$ bound [150–152].

Our solution draws on a connection to a seemingly unrelated area of security/privacy research. By designing our data structure to achieve a type of privacy known as *history independence*, we are able to guard against workloads that might try to force worst-case behavior on the data structure.

This leads to an intriguing lesson. Past work on history independence has focused on bounding the *cost* of history independence for various data-structural problems—that is, how much slower must the data structure become if we wish to achieve history independence? Our results suggest a different perspective—that history independence should actually be viewed as an algorithmic tool that can be used to arrive at faster/better data structures. This perspective will prove to be remarkably useful, not just in the context of dynamic sorting, but also for several other well-studied problems.

Chapter 7. A Matching Lower Bound for History-Independent Solutions. *A priori*, an $O(\log^{1.5} n)$ bound may seem a bit unnatural. Nonetheless, in Chapter 4, we prove that this bound is tight for any history-independent data structure. Thus, if there is to exist a data structure that does better, it will need to make use of fundamentally different techniques than the ones developed here.

Part III. Balls and Bins: When Greedy Allocation Fails and How to Fix It

In Part III, we turn our attention to a technical phenomenon known as *the power of two choices*. In the past two decades, this phenomenon has been widely applied not just to data structures, but more broadly to problems in scheduling, distributed computing, etc.

The power-of-two-choices result considers the following basic setting. Consider a set of m balls that are placed into n bins, one after another. Each ball x must choose between two random bins $h_1(x)$ and $h_2(x)$. The goal is to end in a state where the bins have almost exactly equal loads of $\approx m/n$.

In one of its most general forms [98], what the classical *power-of-two-choices* result states is that: if each ball *greedily* selects the emptier of its two bin options, then the outcome will be remarkably balanced. Each bin will contain almost exactly m/n balls, up to $\pm O(\log m)$, with high probability in m.

The power-of-two-choices result has had tremendous impact on how both theoreticians and practitioners think about load-balancing. But the result comes with a significant weakness: it is restricted to the *arrival-only setting*.

For more than two decades, it remained an open question whether the same result holds in the *dynamic setting* [98, 128], where balls arrive/depart over time, and m is an upper bound on the number of tasks present at any given moment. It was widely believed that the same result *should* hold in this setting, that is, that each bin should contain at most $m/n + O(\log m)$ balls, with high probability in m. But it was only known how to prove this in certain special cases—where either $m = n$ [98], or where arrivals/departures are stochastic.

Before continuing, it is worth taking a moment to understand why we should care about the dynamic setting. From the perspective of *scheduling* applications, we should think of the balls as *tasks* and the bins as *machines* to which they are assigned. The dynamic setting represents the case where each task has arrival and departure times, and where the only constraint on these times is that the maximum load of the system should never exceed m. What's remarkable is that, for this very simple scheduling problem, it has remained an open question of whether strong load-balancing guarantees can be achieved.

On the other hand, the dynamic setting also matters from the perspective of *data-structural* applications. Here the balls represent elements, and the bins represent components of the data structure that are constrained by space. We should think of each element x as having two hash functions $h_1(x)$ and $h_2(x)$ that determine where it can go. This actually leads to a slightly different model than we had in the scheduling case: since elements can be inserted, deleted, and later *reinserted*, any load-balancing scheme must be able to handle the reuse of randomness that occurs during reinsertions. To distinguish between these, we will refer to the dynamic setting without reinsertions as the *scheduling perspective* and to the dynamic setting with reinsertions as the *data-structural perspective.*

Chapter 9. Dynamic Balls and Bins: The Scheduling Perspective. In Chapter 9, we consider the dynamic balls-and-bins problem from the scheduling perspective (i.e., without reinsertions after deletion). Here we uncover a surprising phenomenon: The classical power-of-two-choices result actually *fails* in this setting. With m balls and n bins, there exists an oblivious sequence of insertions and deletions after which, with probability $\Omega(1)$, some bin contains $\Omega(\sqrt{m}/\operatorname{poly}(n))$ more balls than it is supposed to. When $n = O(1)$, this means that some bin contains an overload of $\Omega(\sqrt{m})$ balls, *the same overload bound that is achieved if we simply throw each ball into a random bin without any power-of-two-choices!*

A critical insight, however, is that this lower bound is specific to the greedy strategy—i.e., the strategy in which each ball greedily selects the emptier of two random bins. We show that an alternative *non-greedy* strategy can actually achieve strong bounds: every bin is guaranteed to have an overload of at most $O(\log m)$ balls, with high probability in m.

Our non-greedy strategy is itself a *randomized strategy.* Now, not only is randomness used to select the two bins ($h_1(x)$ and $h_2(x)$), but it is also used to choose between them.

Interestingly, the theme of *history independence as an algorithmic tool* (which we saw in Part II), reappears here. The actual assignment of balls to bins is *not* history independent, but the histogram of how many balls are in each bin *is.* This history independence is what allows us to avoid the types of feedback loops that caused the greedy strategy to perform so poorly.

Chapter 10. Dynamic Balls and Bins: The Data-Structural Perspective. In Chapter 10, we consider the dynamic balls-and-bins problem from the data-structural perspective (i.e., with reinsertions after deletions). Here we uncover a surprising separation: not only does the classical power-of-two-choices result fail, but so do almost all of its possible generalizations. We consider any *ID-oblivious* strategy—that is, any strategy that makes its decisions purely based on the bin choices that each ball has, rather than on the history of that specific ball's insertions/deletions/reinsertions. We show that, for any such strategy, there exists a worst-case workload on 4 bins that causes an overload of $\text{poly}(m)$ balls at some point within the first $\text{poly}(m)$ operations. Interestingly, the worst-case workload that we construct is *universal*—the same workload is simultaneously worst-case for all ID-oblivious strategies.

Here, again, there is reason for optimism, however. Our lower bound applies only to the very heavily loaded case (the number of balls is much larger than the number of bins). However, in most data-structural settings we actually care about a different parameter regime in which $m = nk$ for some $k \in [\omega(\log \log n), O(\log n)]$. Interestingly, this is also the regime that we historically have known the least about—it has remained an open question whether it is possible to achieve a bound even of the form $(1 + o(1))k$ on the maximum load.

We answer this question in the affirmative with a 3-choice scheme that guarantees, with high probability in n, that every bin contains at most $k + O(\sqrt{k \log k}) + O(\log \log n)$ balls. This scheme, which we refer to as Iceberg, also reappears later in the thesis as a valuable tool for designing space-efficient data structures.

Part IV. Let's Hash it Out: Some Barriers Are Fundamental And Others Are Not

In Part IV, we revisit three well-studied theoretical problems concerning hash functions and hash tables. Each of these problems comes with a well-known barrier that has prevented progress. For each problem we seek to answer a simple question: Is this barrier fundamental?

Chapter 12. The Optimal Space-Time Tradeoff Curve for Hash Tables. For nearly six decades, the central question in theoretical work on hash tables has been to determine the optimal achievable tradeoff curve between time and space. State-of-the-art theoretical hash tables offer the following guarantee: If keys/values are $k = \Theta(\log n)$ bits each, then it is possible to achieve constant-time operations while using space

$$\log \binom{2^k}{n} + O(n \log \log n)$$

bits. The first term is the *information-theoretic* space complexity of a hash table. The second represents the *wasted bits*, compared to the information-theoretic optimum.

The bound of $O(n \log \log n)$ wasted bits has stood as the state of the art for more than two decades [313]. There are several reasons to believe that this bound should be optimal. Indeed, the same bound *is optimal* for closely related problems involving resizable filters [301] and dynamic retrieval [146, 273]. And, perhaps more importantly, for the hash-table problem itself, the bound is known to be optimal for any hash table satisfying a certain natural property known as *stability* [75, 146]. It has remained an open question, however, whether the bound is optimal for all hash tables.

Chapter 12 shows that there is actually no fundamental barrier at $O(n \log \log n)$ wasted bits. In fact, for any $k \in [\log^* n]$, it is possible to achieve $O(k)$-time insertions/deletions, $O(1)$-time queries, and

$$O(n \log^{(k)} n) = O\left(n \underbrace{\log \log \cdots \log}_{k} n\right)$$

wasted bits (all with high probability in n). This means that, each time we increase insertion/deletion time by an *additive constant*, we reduce the wasted bits *per key* exponentially.

In recent follow-up work, Li, Liang, Yu, and Zhou [243] proved our space/time tradeoff is *optimal across all dynamic data structures*. Combined, our data structure along with their lower bound close off one of the longest-standing research directions in the field of data structures.

We emphasize that our results hold not just for fixed-capacity hash tables, but also for hash tables that are dynamically resized (this is a fundamental departure from what is possible for filters [301]); and for hash tables that store very large keys/values, each of which can be up to $n^{o(1)}$ bits (this breaks with the conventional wisdom that larger keys/values should lead to more wasted bits per key [52, 313]). For very small keys/values, we are able to tighten our bounds to $o(1)$ wasted bits per key, even when $k = O(1)$. Building on this, we obtain a constant-time dynamic filter that uses $n \lceil \log \varepsilon^{-1} \rceil + n \log e + o(n)$ bits of space for a wide choice of false-positive rates ε, resolving a long-standing open problem on the design of dynamic filters [52, 82, 93, 122, 245, 249, 294].

Chapter 13. A Hash Table Without Hash Functions. Chapter 13 considers the basic question of how strong of a probabilistic guarantee can a hash table storing $(1+\Theta(1)) \log n$-bit key/value pairs offer? Past work on this question has encountered a surprising bottleneck: If we assume access to fully random hash functions, then it is possible to achieve very strong probabilistic guarantees [75, 198, 199], but if the same hash tables are implemented using the known families of hash functions, it is unknown how to achieve a failure probability better than $1/2^{\operatorname{polylog} n}$.

Thus, somewhat surprisingly, the barrier for this problem would appear to be one about *hash functions* rather than *hash tables*. The techniques that we have for simulating fully random hash functions introduce additional failure probabilities that become the bottleneck.

We show that, once again, the barrier is not fundamental. Our solution is to construct a data structure that has the same guarantees as a hash table, and that is still randomized, but that *avoids the use of hash functions*. The resulting data structure achieves a failure probability of $1/n^{n^{1-\varepsilon}}$ for an arbitrary positive constant ε. This bound is close to the best that one could hope for: If the failure probability could be improved to $1/n^{\Omega(n)}$ (i.e., the ε could be removed), then this would imply the (non-constructive) existence of a deterministic hash table, which is widely believed to be impossible [341].

Our failure probability of $1/n^{n^{1-\varepsilon}}$ represents a nearly exponential improvement over the previous state of the art of $1/2^{\operatorname{polylog} n}$, and resolves an open problem posed by Goodrich, Hirschberg, Mitzenmacher, and Thaler [198, 199].

Chapter 14. Tight Bounds For Minimal Monotone Perfect Hashing. The *monotone minimal perfect hash function* (MMPHF) problem is the following indexing problem. Given a set $S = \{s_1, \ldots, s_n\}$ of n distinct keys from a universe U of size u, create a data structure $\mathbf{D}$ that answers the following query:

$$\textsc{Rank}(q) = \begin{cases} \text{rank of } q \text{ in } S & q \in S \\ \text{arbitrary answer} & \text{otherwise.} \end{cases}$$

The name of the problem comes from interpreting the data structure $\mathbf{D}$ as a hash function: given a sorted array $A = [a_1, \ldots, a_n]$, $\mathbf{D}$ is a function mapping each a_i to its position i. Such a hash function is *minimal*, meaning that it maps n items to n distinct positions, and *monotone*, meaning that whenever $a_i < a_j$ we have $\mathbf{D}(a_i) < \mathbf{D}(a_j)$, and vice versa.

It may seem strange at first glance that $\mathbf{D}$ is permitted to return arbitrary answers on negative queries. A key insight, however, is that this relaxation allows for asymptotic improvements in space efficiency: whereas the set $\mathcal{S}$ would require $\Omega(n \log(u/n))$ bits to encode, Belazzougui, Boldi, Pagh, and Vigna [64] show that it is possible to construct an MMPHF using as few as $O(n \log\log\log u)$ bits, while supporting $O(\log\log u)$-time queries.

Since its introduction in 2009, MMPHF has found a variety of practical applications (e.g., in security [110], key-value stores [244] and information retrieval [280]). A high-performance implementation can be found in the Sux4J library [63, 108]. MMPHF has also been widely used in the theory community for the design of space-efficient combinatorial pattern-matching algorithms (see, e.g., [62, 65–68, 127, 191, 203]).

Despite the widespread use of MMPHF, it has remained an open question [64, 109, 154] to determine the optimal bounds for solving this problem. Even disregarding applications (and the running time to answer queries), the information-theoretic question has been posed as a problem of independent combinatorial interest [154].

Whereas in the previous two chapters, the apparent barriers were based on limitations of known techniques, here the $O(n \log\log\log u)$ barrier would seem to be much more arbitrary. Although it is unknown how to do better, there is also no clear reason why this somewhat unusual bound should be tight.

Nonetheless, in Chapter 14, we show that there actually *is* a fundamental barrier at $O(n \log\log\log u)$ bits. In fact, our lower bound of $\Omega(n \log\log\log u)$ is information-theoretic, meaning that it holds for any data structure regardless of its time efficiency. At a technical level, our lower bound is achieved by constructing a graph whose fractional chromatic number reveals the optimal answer to the question at hand. We believe that this fractional-chromatic-number approach will likely prove to be generally useful for establishing lower bounds on related data-structural problems in the future.

Part V. How Many Bits Does It Take to Write Down a Pointer?

One type of data structure that is typically *not* viewed as space efficient, is any data structure that makes heavy use of pointers. Examples include chained hash tables, binary search trees, stable dictionaries, internal-memory stashes, etc. In each of these data structures, the pointers between elements consume a significant fraction of the total space used by the data structure.

Chapter 16: From Balls and Bins to Tiny Pointers. In Chapter 16, we introduce a new data-structural object that we call the tiny pointer. In many applications, traditional $\log n$-bit pointers can be replaced with $o(\log n)$-bit tiny pointers at the cost of only a constant-factor time overhead. We develop a comprehensive theory of tiny pointers, and give optimal constructions for both fixed-size tiny pointers (i.e., settings in which all of the tiny pointers must be the same size) and variable-size tiny pointers (i.e., settings in which the average tiny-pointer size must be small, but some tiny pointers can be larger). If a tiny pointer references an element in an array filled to load factor $1 - 1/k$, then the optimal tiny-pointer size is $\Theta(\log\log\log n + \log k)$ bits in the fixed-size case, and $\Theta(\log k)$ expected bits in the variable-size case.

Interestingly, what makes tiny pointers possible are the balls-to-bins techniques that we developed in Part III. Indeed, the main algorithmic primitive on which our constructions are based is the ICEBERG scheme developed in Chapter 6.

Chapter 17: Five Applications to Data Structures. Using tiny pointers, we revisit five classic data-structure problems. We show that:

- A data structure storing n v-bit values for n keys with constant-time modifications/queries can be implemented to take space $nv + O(n \log^{(r)} n)$ bits, for any constant $r > 0$, as long as the user stores a tiny pointer of expected size $O(1)$ with each key—here, $\log^{(r)} n$ is the r-th iterated logarithm.
- Any comparison-based binary search tree can be made succinct with constant-

factor time overhead, and can even be made to be within $O(n)$ bits of optimal if we allow for $O(\log^* n)$-time modifications—this holds even for rotation-based trees such as the splay tree and the red-black tree.
- Any fixed-capacity key-value dictionary can be made stable (i.e., items do not move once inserted) with constant-time overhead and $1 + o(1)$ space overhead.
- Any key-value dictionary that requires uniform-size values can be made to support arbitrary-size values with constant-time overhead and with an additional space consumption of $\log^{(r)} n + O(\log j)$ bits per j-bit value for an arbitrary constant $r > 0$ of our choice.
- Given an external-memory array A of size $(1 + \varepsilon)n$ in which we must store a dynamic set of up to n key-value pairs, it is possible to maintain an internal-memory stash of size $O(n \log \varepsilon^{-1})$ bits so that the location of any key-value pair in A can be computed in constant time (and with no IOs).

These are all well-studied and classic problems, and in each case, tiny pointers allow us to take a natural space-inefficient solution that uses pointers and make it space-efficient for free.

In several of the applications above, we are actually making use of not just tiny pointers, but also the super-space efficient hash tables from Chapter 10. These tools turn out to be remarkably symbiotic, allowing us to bring the same (extremely strong) space-efficiency bounds that we achieved in Chapter 12 to other problems that would seem to *a priori* to be fundamentally different.

We remark that our pointer-compression techniques have also found practical applications in the design of computer hardware. Here, tiny pointers act not just as a theoretical technique, but as an avenue for obtaining real-world performance improvements. Indeed, concurrently with our theoretical work on the topic, we engaged in a multi-year collaboration between systems researchers, computer architects, and theoreticians, where we used tiny pointers to redesign a piece of hardware known as the translation look-aside buffer (TLB), in order to increase its coverage while mitigating a problem known as fragmentation. The results of this collaboration appeared at ASPLOS'23 where they won the Distinguished Paper Award [10].

Part VI. A Strong Theory of Strong History Independence

In several parts of the thesis, *history independence* has appeared as a technical hero. By using it as an algorithmic/analytical technique, we were able to break through long-standing barriers for both dynamic sorting (Part II) and dynamic power-of-two choices (Part III). However, history-independent data structures have also been studied extensively as a subfield on their own. And several of the most basic questions that one can ask—concerning the closely related problems of memory allocation [279], hashing [104, 124, 279], and worker-task assignment [334, 335]—have remained open. In this chapter, we give nearly tight answers to these questions.

Somewhat unintentionally, the results in this chapter constitute yet another example where history independence allows us to obtain a new state of the art for a (non-history-independent) problem—indeed, our history-independent algorithm for the variable-size memory (re)-allocation is the first algorithm (history independent or not) to beat the folklore bound for the problem [81].

Before continuing, we remark that two types of history independence have been studied in the literature, *strong* and *weak* history independence. All of the algorithmic results in this part are for strong history independence (although in some cases they resolve questions that were open for weak history independence too). As background, a data structure ALG is said to be ***strongly history independent*** (or ***uniquely representable***) if its state is *fully determined by its current set of elements.* That is, given the current set S of elements that the data structure stores, and given the random bits R that the data structure uses, one can reconstruct the data structure $\text{ALG}(S, R)$. Critically, such a data structure *cannot* depend on the order in which elements were inserted, or on the history of what other elements were present in the past.

Chapter 19. Strong Upper Bounds for Stateless Allocation. One of the most basic data-structural questions is memory allocation: given a set S of items x whose sizes $\pi(x)$ add up to $(1-\varepsilon)n+1$, one wishes to partition an array A of size n amongst the items, so that each item $x \in S$ gets a disjoint sub-array of size at least $\pi(x)$.

An allocation algorithm is said to incur an expected ***overhead*** at most L if, whenever two input sets S_1 and $S_2 = S_1 \cup \{x\}$ differ by the insertion of an element x, the resulting allocations ϕ_1 and ϕ_2 satisfy the following guarantee: the elements $\Delta = \{y \in S_1 \mid \phi_1(y) \neq \phi_2(y)\}$ have cumulative expected size $\mathbb{E}[\sum_{y \in \Delta} \text{size}(y)] \leq L \cdot \text{size}(x)$.

The problem of constructing a *strongly history-independent* allocation algorithm that achieves small overhead has remained one of the central open questions in the field since it was first posed by Naor and Teague [279] in their seminal 2001 paper on history independence. For more than two decades, the only positive result has been for the case of $\varepsilon = 1/2$, where Naor and Teague constructed a *weakly* history-independent result with overhead $O(\log n)$ [279].

We show that in the same parameter regime ($\varepsilon = 1/2$), it is possible to construct an *strongly* history-independent algorithm that achieves an expected overhead of $O(1)$. Moreover, for arbitrary ε, we show how to achieve an expected overhead of $O(1 + \log \varepsilon^{-1})$, so long as each element $x \in S$ has size at most $\pi(x) \leq n \operatorname{poly} \varepsilon$.

Our bound of $O(1+\log \varepsilon^{-1})$ represents a new state-of-the-art even for *non-history-independent* algorithms. Indeed, in this setting, the best previous bound was an overhead of $O(\varepsilon^{-1})$ [81]. Once again, we have an example where history independence allows us to bypass seemingly natural barriers.

In fact, the bound of $O(1 + \log \varepsilon^{-1})$ is so strong that it represents a new strongly-history-independent state of the art even in the *fixed-size* case, where every element $x \in S$ has size $\pi(x) = 1$. Here, we are considering the most basic possible allocation problem: place a set of up to $(1-\varepsilon)n+1$ elements into an array of size n. Past strongly history-independent solutions [104, 124, 279], have all had expected overheads $\Omega(\varepsilon^{-1})$. Our bound of $O(1 + \log \varepsilon^{-1})$ is an exponential improvement.

The fixed-size case is of special interest to the distributed-computing community, where the setting of $\varepsilon = n^{-1}$ (i.e., the array is fully saturated) corresponds to the so-called distributed memoryless worker-task assignment problem [334, 335]. Here, there is a special focus on achieving *worst-case* guarantees. Intuitively, this would seem to be much harder (how can we have strong history independence without randomization?), and indeed no past solution has beaten the trivial bound of $O(n)$ overhead.

Our final result of the chapter is a solution to the distributed memoryless worker-task assignment problem that achieves a *worst-case* overhead of $O(\log^2 n)$ on sets $S \subseteq [\text{poly}(n)]$. Our solution is achieved via the probabilistic method, so that the *analysis* makes use of randomization, but the final *algorithm* is deterministic. To the best of our knowledge, this is the first example of a deterministic data structure solving a (nontrivial) dynamic problem while offering strong history independence.

Chapter 20. Strong Lower Bounds for Stateless Allocation. In Chapter 20, we prove that our results from the previous chapter are tight up to doubly logarithmic factors: we give an $\Omega(\log \varepsilon^{-1} / \log \log \varepsilon^{-1})$ lower bound on the expected overhead incurred by any strongly history-independent allocation algorithm.

Perhaps surprisingly, this lower bound applies even in the *fixed-size case*, where elements $x \in S$ all have sizes $\pi(x) = 1$. Prior to our work, the state-of-the-art lower bounds were of the form $\Omega(1)$ [334, 335] (and even this required quite sophisticated arguments [335]!). Moreover, these lower bounds held only for *worst-case* overhead, in the setting where $\varepsilon^{-1} = n$, and where the elements came from a universe of size $\geq \text{tow}(n)$. In contrast, our lower bound of $\Omega(\log \varepsilon^{-1} / \log \log \varepsilon^{-1})$ applies to *expected* overhead, supports any value of ε, and can be implemented using a universe of size $O(n)$.

Combined, our upper and lower bounds give nearly tight answers for both the fixed-size and variable-size allocation problems. Once more, the bounds reveal an unexpected plot point—that the optimal bounds for the two problems are essentially the same. There is no cost to having variable-size objects.

Chapter 21. Efficient Data-Structural Implementations. Finally, in Chapter 21, we show how to adapt our results in the previous chapters to obtain nearly optimal solutions to a closely related problem: that of constructing an optimal strongly-history-independent hash-table problem.

A hash table is, in essence, a time-efficient memory-allocation scheme that also supports fast queries. Thus the schemes in Chapter 19 immediately imply approaches for constructing strongly history-independent hash tables. However, even though these approaches achieve low overhead, they would *a priori* seem to be difficult to implement time efficiently.

The main contribution of Chapter 21 is that, using techniques from the theory of succinct data structures, our allocation algorithms can be made time efficient as well. This leads to near-optimal constructions for strongly history-independent hash-table constructions and closes (up to doubly logarithmic factors) a line of work initiated by Naor and Teague [279] and Blelloch and Golovin [104].

1.3 Bibliographics

This thesis comprises some of my favorite papers and collaborations from my PhD. Part I is based on joint work with Bender and Kuszmaul [1] at FOCS'21. Part II is based on joint work with Bender, Conway, Farach-Colton, Komlós, and Wein [2] at FOCS'22. Part III is based on joint work with Bansal [3] at FOCS'22 and on joint work with Bender, Conway, Farach-Colton, and Tagliavini [7] at SODA'23. Part IV is based on joint work with Bender, Farach-Colton, Kuszmaul, and Liu [4] at STOC'22, as well as a solo-authored paper [5] at FOCS'22, and joint work with Assadi and Farach-Colton [6] at SODA'23. Part V is based on joint work with Bender, Conway, Farach-Colton, and Tagliavini [7] at SODA'23. And Part VI is based on joint work with Berger, Polak, Tidor, and Wein [9] at ICALP'22 as well as on an upcoming solo-authored paper [8] at FOCS'23.

The results in this thesis are part of a larger body of work from my PhD [10–33]. Although it was not possible to include all of my work in this thesis, I have included a select bibliography consisting of the papers that contributed the most to the themes, techniques, and perspectives in this thesis.

Papers Contained in this Thesis

(Ordered by appearance)

[1] Michael A Bender, Bradley C Kuszmaul, and William Kuszmaul. Linear probing revisited: Tombstones mark the demise of primary clustering. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1171–1182. 2021.

[2] Michael A Bender, Alex Conway, Martín Farach-Colton, Hanna Komlós, William Kuszmaul, and Nicole Wein. Online list labeling: Breaking the $\log^2 n$ barrier. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 980–990. 2022.

[3] Nikhil Bansal and William Kuszmaul. Balanced allocations: The heavily loaded case with deletions. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 801–812. 2022.

[4] Michael A Bender, Martín Farach-Colton, John Kuszmaul, William Kuszmaul, and Mingmou Liu. On the optimal time/space tradeoff for hash tables. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1284–1297. 2022.

[5] William Kuszmaul. A hash table without hash functions, and how to get the most out of your random bits. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 991–1001. 2022.

[6] Sepehr Assadi, Martín Farach-Colton, and William Kuszmaul. Tight bounds for monotone minimal perfect hashing. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 456–476. 2023.

[7] Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. Tiny pointers. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 477–508. 2023.

[8] William Kuszmaul. Strongly History-Independent Storage Allocation: New Upper and Lower Bounds. In *2023 IEEE 64rd Annual Symposium on Foundations of Computer Science (FOCS)*, to appear. 2023.

[9] Aaron Berger, William Kuszmaul, Adam Polak, Jonathan Tidor, and Nicole Wein. Memoryless worker-task assignment with polylogarithmic switching cost. In *49th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 229, page 19. 2022.

Select Other Papers From my PhD

(Ordered by publication date)

[10] Krishnan Gosakan, Jaehyun Han, William Kuszmaul, Ibrahim N. Mubarek, Nirjhar Mukherjee, Karthik Sriram, Guido Tagliabini, Evan West, Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martín Farach-Colton, Jayneel Gandhi, Rob Johnson, Sudarsun Kannan, and Donald E. Porter Mosaic Pages: Big TLB Reach with Small Pages. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 3, pages 433–448. 2023. ***(Distinguished Paper)***

[11] Prashant Pandey, Michael A. Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, Guido Tagliavini, and Rob Johnson. IcebergHT: High-Performance PMEM Hash Tables Through Stability and Low Associativity. In *Proceedings of the ACM on Management of Data (SIGMOD)*, volume 1(1), pages 1–26. 2023.

[12] Michael A. Bender, Daniel DeLayo, Bradley C. Kuszmaul, William Kuszmaul, Evan West. Analyzing Every Cache, Everywhere, All the Time. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 129–139. 2023.

[13] William Kuszmaul and Zoe Xi. Approximating Dynamic Time Warping Distance Between Run-Length Encoded Strings. In *Proceedings of 30th Annual European Symposium on Algorithms (ESA)*. 2022. ***(Best Student Paper)***

[14] Kunal Agrawal, Michael A. Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. Parallel Paging with Optimal Makespan. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 205–216. 2022. ***(Best-Paper Finalist)***

[15] William Kuszmaul and Shyam Narayanan. Optimal Time-Backlog Tradeoffs for the Variable-Processor Cup Game. In *49th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 229, page 85. 2022.

[16] Michael A. Bender, Martín Farach-Colton, and William Kuszmaul. What Does Dynamic Optimality Mean in External Memory? In *Proceedings of the 13th Innovations in Theoretical Computer Science (ITCS) Conference.* 2022.

[17] Michael A. Bender, Tsvi Kopelowitz, William Kuszmaul, Ely Porat, Clifford Stein. Incremental Edge Orientation in Forests. In *Proceedings of 29th Annual European Symposium on Algorithms (ESA)*, volume 204, page 12. 2021.

[18] William Kuszmaul. How Asymmetry Helps Buffer Management: Achieving Optimal Tail Size in Multi-Processor Cup Games. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1248–1261. 2021.

[19] William Kuszmaul and Charles E. Leiserson. Floors and Ceilings in Divide-and-Conquer Recurrences. In *Proceedings of SIAM Symposium on Simplicity in Algorithms (SOSA)*, pages 133-141. 2021.

[20] Michael A. Bender and William Kuszmaul. Randomized Cup Game Algorithms Against Strong Adversaries. In *Proceedings of the 2021 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2059–2077. 2021.

[21] Kunal Agrawal, Michael A. Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. Tight Bounds for Parallel Paging and Green Paging. In *Proceedings of the 2021 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3022–3041. 2021.

[22] William Kuszmaul and Alek Westover. The Variable-Processor Cup Game. In *Proceedings of the 12th Innovations in Theoretical Computer Science (ITCS) Conference.* 2021.

[23] Krishnan Gosakan, Jaehyun Han, William Kuszmaul, Ibrahim N. Mubarek, Nirjhar Mukherjee, Karthik Sriram, Guido Tagliabini, Evan West, Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martín Farach-Colton, Jayneel Gandhi, Rob Johnson, Sudarsun Kannan, and Donald E. Porter. Paging and the Address-Translation Problem. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 105–117. 2021.

[24] William Kuszmaul. Train Tracks with Gaps. In *Proceedings of the 10th International Conference on Fun with Algorithms (FUN)*. 2020. Extended Version in Special Issue of *Theoretical Computer Science.* 2022. ***(Best Paper)***

[25] Michael A Bender, Rezaul A Chowdhury, Rathish Das, Rob Johnson, William Kuszmaul, Andrea Lincoln, Quanquan C Liu, Jayson Lynch, and Helen Xu. Closing the Gap Between Cache-oblivious and Cache-adaptive Analysis. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 63–73. 2020.

[26] Michael A. Bender, Tsvi Kopelowitz, William Kuszmaul, and Seth Pettie. Contention Resolution without Collision Detection. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1284–1297. 2020.

[27] Michael A. Bender, Rathish Das, Martín Farach-Colton, Rob Johnson, and William Kuszmaul. Flushing Without Cascades. In *Proceedings of the 2020 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 650–659. 2020.

[28] William Kuszmaul. Achieving Optimal Backlog in the Vanilla Multi-Processor Cup Game. In *Proceedings of the 2020 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1558–1577. 2020.

[29] Tim Kaler, William Kuszmaul, Tao B. Schardl, and Daniele Vettorel. Cilkmem: Algorithms for Analyzing the Memory High-Water Mark of Fork-Join Parallel Programs. In *Proceedings of Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pp. 162-176. 2020. ***(Best Paper Finalist)***

[30] Michael A. Bender, Martín Farach-Colton, and William Kuszmaul. Achieving Optimal Backlog in Multi-Processor Cup Games. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1148–1157 2019.

[31] Vladimir Braverman, Moses Charikar, William Kuszmaul, David P. Woodruff, and Lin F. Yang. The One-Way Communication Complexity of Dynamic Time Warping Distance. In *Proceedings of the 35th International Symposium on Computational Geometry (SoCG)*. 2019. Extended Version in Invited Issue of *Journal of Computational Geometry*, 11(2), 62-93. 2021.

[32] William Kuszmaul. Dynamic Time Warping in Strongly Subquadratic Time: Algorithms for the Low-Distance Regime and Approximate Evaluation. In *46th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 132, page 80. 2019.

[33] William Kuszmaul. Efficiently Approximating Edit Distance Between Pseudorandom Strings. In *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1165–1180. 2019.

Part I

Linear Probing Revisited: Overturning the Oldest Myth in Data Structures

Chapter 2

Introduction

The linear probing hash table [131, 165, 201, 230, 241, 253, 257, 323, 324, 332, 343, 353] is among the most fundamental data structures to computer science. The hash table takes the form of an array of some size n, where each ***slot*** of the array either contains an element or is empty (i.e., is a ***free slot***). To insert a new element u, the data structure computes a hash $h(u) \in [n]$, and places u into the first free slot out of the sequence $h(u),\ h(u)+1,\ h(u)+2,\ \ldots$ (modulo n). Likewise, a query for u simply scans through the slots, terminating when it finds either u or a free slot.

There are two ways to implement deletions: immediate compactions and tombstones. An immediate compaction rearranges the neighborhood of elements around a deletion to make it as though the element was never there [217,226]. Tombstones, the approach we will default to, replaces the deleted element with a ***tombstone*** [226,309]. Tombstones interact asymmetrically with queries and insertions: queries treat a tombstone as being a value that does not match the query, whereas insertions treat the tombstone as a free slot. In order to prevent slowdown from an accumulation of tombstones, the table is occasionally rebuilt in order to clear them out.

The appeal of linear probing. Linear probing was discovered in 1954 by IBM researchers Gene Amdahl, Elaine McGraw, and Arthur Samuel, who developed the hash table design while writing an assembly program for the IBM 701 (see discussion in [224, 226]). The discovery came just a year after fellow IBM researcher Hans Peter Luhn introduced chained hashing (also while working on the IBM 701) [226].[1] Linear probing shared the relative simplicity of chaining, while offering improved space utilization by avoiding the use of pointers.

The key property that makes linear probing appealing, however, is its data locality: each operation only needs to access one localized region of memory. In 1954, this meant that queries could often be completed at the cost of accessing only a single drum track [224, 309]. In modern systems, it means that queries can often be completed in

[1]Interestingly, Luhn's discussion of chaining may be the first known use of linked lists [226, 227]. Linked lists are often misattributed [360] as having been invented by Newell, Shaw, and Simon [286, 287] during the RAND Corporation's development of the IPL-2 programming language in 1956 (see discussion in [227]).

just a single cache miss [316].[2] The result is that, over the decades, even as computer architectures have changed and as the study of hash tables has evolved into one of the richest and most studied areas of algorithms, linear probing has persisted as one of the best-performing hash tables in practice [316].

The drawback: primary clustering. Unfortunately, the data locality of linear probing comes with a major drawback known as ***primary clustering*** [224, 228]. Consider the setting in which one fills a linear-probing hash table up to a ***load factor*** of $1 - 1/x$ (i.e., there are $(1 - 1/x)n$ elements) and then performs one more insertion. Intuitively, one might expect the insertion to take time $\Theta(x)$, since one in every x slots are free. As Knuth [224] discovered in 1963[3], however, the insertion actually runs much slower, taking expected time $\Theta(x^2)$.

The reason for these slow insertions is that elements in the hash table have a tendency to cluster together into long runs; this is known as primary clustering. Primary clustering is often described as a "winner keeps winning" phenomenon, in which the longer a run gets, the more likely it is to accrue additional elements; see e.g., [131,165,230,241,271,330,343,361]. As Knuth discusses in [226], however, winner-keeps-winning is not the main cause of primary clustering.[4] The true culprit is the globbing together of runs: a single insertion can connect together two already long runs into a new run that is substantially longer.

Interestingly, primary clustering is an asymmetric phenomenon, affecting insertions but not queries. Knuth showed that if a query is performed on a random element in the table, then the expected time taken is $\Theta(x)$ [224]. This can be made true for all queries (including negative queries) by implementing a simple additional optimization: rather than placing elements at the end of a run on each insertion, order the elements within each run by their hashes. This technique, known as ***ordered linear probing*** [45, 125], allows a query to terminate as soon as it reaches an element whose hash is sufficiently large.[5] Whereas insertions must traverse the entire run, queries need not.

Knuth's result predates the widespread use of asymptotic notation in algorithm analysis. Thus Knuth derives not only the asymptotic formula of $\Theta(x^2)$ for expected

[2]Moreover, even when multiple cache misses occur, since those misses are on adjacent cache lines, hardware prefetching can mitigate the cost of the subsequent misses. Both the Masstree [254] and the Abseil B-tree [39] treat the effective cache-line size as 256 bytes even though the hardware cache lines are 64 bytes.

[3]Knuth's result [224] was never formally published, and was subsequently rediscovered by Konheim and Weiss in 1966 [228].

[4]This can be easily seen by the following thought experiment. Consider the length ℓ of the run at a given position in the hash table, and then consider a sequence of $(1-1/x)n$ insertions where we model each insertion as having probability $(1+\ell)/n$ of increasing the length of the run by 1 (here ℓ is the length of the run at the time of the insertion). The expected length ℓ of the run at the end of the insertions won't be $\Theta(x^2)$. In fact, rather than being an asymptotic function of x, it will simply be $\Theta(1)$.

[5]Contemporary works sometimes also refer to this as ***Robin hood hashing***, but as noted in [125], Robin hood hashing is actually a generalization of ordered linear probing to other open-addressing schemes such as double hashing, uniform probing, etc.

insertion time, but also a formula of

$$\frac{1}{2}\left(1 + x^2\right), \tag{2.1}$$

which is exact up to low order terms. There has also been a great deal of follow-up work determining detailed tail bounds and other generalizations of Knuth's result; see e.g., [180, 215, 228, 262, 345–347].

The practical and cultural effects of primary clustering. Primary clustering is taught extensively in both theoretical and practical courses [60, 143, 145, 147, 171, 179, 202, 221, 264, 274, 322, 336]. Many textbooks not only teach primary clustering [131, 165, 201, 230, 241, 253, 257, 323, 324, 332, 343, 353], but also teach the full formula (2.1) for insertion time [165, 230, 241, 253, 323, 324, 332, 343, 353].[6] (For a more detailed summary of how courses and textbooks teach linear probing and primary clustering, see Figure 2.)

Because (2.1) is exact, it is viewed as representing the full picture of how linear probing behaves at high load factors. One consequence is that there has been little empirical work on analyzing the asymptotics of real-world linear probing at high load factors. (And for good reason, what would be the point of verifying what is already known?) As Sedgewick observed in 1990 [324], it is not even clear that the classic formula has been empirically verified at high load factors.

A common recommendation [60, 131, 143, 165, 179, 202, 221, 336, 353, 359] is that, in order to avoid primary clustering, one should use ***quadratic probing*** [211, 255, 362] instead. Whereas linear probing places each element x in the first available position out of the sequence $h(x), h(x) + 1, h(x) + 2, h(x) + 3, \ldots$, quadratic probing uses the first available position out of a quadratic sequence such as $h(x), h(x) + 1, h(x) + 4, h(x) + 9, \ldots, h(x) + k^2, \ldots$ or $h(x), h(x) + 1, h(x) + 3, \ldots, h(x) + k(k-1)/2, \ldots$. By using a more spread out sequence of probes, quadratic probing seems to eliminate primary clustering in practice. In doing so, quadratic probing also compromises the most attractive trait of linear probing, its data locality. This tradeoff is typically viewed as unfortunate but necessary.

The dangers of primary clustering (and the advice of using quadratic probing as a solution) have been taught to generations of computer scientists over roughly six decades. The folklore advice has shaped some of the most widely used hash tables in production, including the high-performance hash tables authored by both Google [39] and Facebook [117]. The consequence is that primary clustering—along with the design compromises made to avoid it—has a first-order impact on the performance of hash tables used by millions of users every day.

Our contribution: How to get rid of primary clustering. In this part of the thesis, we demonstrate that primary clustering is not the fixed and universal

[6] Some books also go through worked examples or tables of (2.1) to give intuition for how badly linear probing scales, e.g., [241, 324, 332, 343].

Textbooks' Stances on Linear Probing

Source	Teaches primary clustering	Teaches Knuth's formulae	Recommends (QP=quadratic probing) (DH=double hashing)
Cormen, Leiserson, Rivest, and Stein [131]	yes	no	QP or DH
Dasgupta, Papadimitriou, and Vazirani [141]	no	no	not applicable
Drozdek and Simon [165]	yes	yes	QP or DH
Goodrich and Tamassia [201]	yes	no	$\leq$ 50% load factor
Kleinberg and Tardos [223]	no	no	not applicable
Kruse [230]	yes	yes	chaining
Lewis and Denenberg [241]	yes	yes	DH with ordered probing
Main and Savitch [253]	yes	yes	DH
McMillan [257]	yes	no	chaining
Sedgewick [323, 324]	yes	yes	chaining or DH
Standish [332]	yes	yes	not prescriptive
Tremblay and Sorenson [343]		yes	DH
Weiss [353]	yes	yes	QP
Wengrow [354]	no	no	$\leq$ 70% load factor
Wikipedia [359]	yes	yes	QP or DH
Wirth [367, 368]	yes	partly	search trees

Some Course Notes' Stances on Linear Probing

Source	Teaches primary clustering	Teaches Knuth's formulae	Recommends
CMU Systems [221]	yes	no	QP or DH
CalPoly Fundamentals of CS [179]	yes	no	QP or DH
Columbia Data Structs in Java [60]	yes	no	QP or DH
Cornell Prog. and Data Structs [202]	partly	partly	QP
Harvard Intro. to CS [336]	yes	no	QP or DH
MIT Advanced Data Structs [145]	yes	no	low load factor
MIT Intro. to Algorithms [147]	yes	no	DH
Stanford Data Structs [322]	yes	yes	chaining or low load factor
UIUC Algorithms [171]	yes	no	binary probing
UMD Data Structs [274]	yes	yes	DH
UT Software Design [264]	yes	no	$\leq 2/3$ load factor
UW Data Structs and Algorithms [143]	yes	yes	QP or DH

phenomenon that it is reputed to be. When implementing linear probing, there is a small set of design decisions that are typically treated as implementation-level engineering choices. We show that these decisions actually have a remarkable effect on performance: even if a workload operates continuously at a load factor of $1-\Theta(1/x)$, if the design decisions are made correctly, then the expected amortized cost per insertion can be decreased all the way to $\tilde{O}(x)$. Our results come with actionable lessons for practitioners, indicating that implementation-level decisions can have unintuitive asymptotic consequences on performance.[7]

We also present a new variant of linear probing, which we call ***graveyard hashing***, that completely eliminates primary clustering on any sequence of operations: if, when an operation is performed, the current load factor is $1 - 1/x$ for some x, then the expected cost of the operation is $O(x)$.

Thus we achieve the data locality of traditional linear probing without any of the disadvantages of primary clustering. One corollary is that, in the external-memory model with data blocks of size B, graveyard hashing offers the following remarkably strong guarantee: at any load factor $1 - 1/x$ satisfying $x = o(B)$, graveyard hashing achieves $1 + o(1)$ expected block transfers per operation. In contrast, past external-memory hash tables have only been able to offer a $1 + o(1)$ guarantee when the block size B is at least $\Omega(x^2)$ [216].

What the classical analysis misses. Classically, the analysis of linear probing considers the costs of insertions in an insertion-only workload. Of course, the fact that the final insertion takes expected time $\Theta(x^2)$ doesn't mean that all of the insertions do; most of the insertions are performed at much lower load factors, and the average cost is only $\Theta(x)$.

The more pressing concern is what happens for workloads that operate continuously at high load factors, for example, the workload in which a user first fills the table to a load factor of $1-1/x$, and then alternates between insertions and deletions indefinitely. Now almost all of the insertions are performed at a high load factor. Conventional wisdom has it that these insertions must therefore all incur the wrath of primary clustering.

This conventional wisdom misses an important point, however, which is that the tombstones created by deletions actually substantially change the combinatorial structure of the hash table. Whereas insertions add elements at the ends of runs, deletions tend to place tombstones in the middles of runs. If implemented correctly, then the anti-clustering effects of deletions actually outpace the clustering effects of insertions.

We call this new phenomenon ***primary anti-clustering***. The effect is so powerful that, as we shall see, it is even worthwhile to simulate deletions in insertion-only workloads by prophylactically adding tombstones.

Our results flip the narrative surrounding deletions in hash tables: whereas past work on analyzing tombstones [55, 217] has focused on showing that tombstones do not *degrade* performance in various open-addressing-based hash tables, we argue that

[7]The right thing to do may even be the *opposite* of what the literature recommends [55, 217].

tombstones actually *help* performance. By harnessing the power of tombstones in the right way, we can rewrite the asymptotic landscape of linear probing.

Chapter 3. The Surprisingly Strong Anti-Clustering Effects of Tombstones

We begin by giving nearly tight bounds on the amortized performance of ordered linear probing.

There are two design decisions that affect performance: (1) the use of tombstones (which we assume by default) and (2) the frequency with which the hash table is rebuilt and the tombstones are cleared out. Thus, in addition to the size parameter n and the load-factor parameter x, our analysis uses a ***rebuild window size*** parameter R, which is the number of insertions that occur between rebuilds.

The parameter R is classically set to be $n/(2x)$, since this means that the number of tombstones cannot affect the asymptotic load factor. Each rebuild can be implemented in time $\Theta(n)$, so the rebuilds only contribute amortized $\Theta(x)$ time per insertion, which is a low-order term.

A subquadratic analysis of linear-probing insertions. Our first result considers the classical setting $R = n/(2x)$ and analyzes a ***hovering workload***, i.e., an alternating sequence of inserts/deletes at a load factor of $1 - 1/x$.

We prove that the expected amortized cost of each insertion is $\tilde{O}(x^{1.5})$. This is tight, which we establish with a lower bound of $\Omega(x^{1.5}\sqrt{\log\log x})$. A surprising takeaway is that the answer is both $\tilde{O}(x^{1.5})$ and $\omega(x^{1.5})$.

This first result is already substantially faster than the classical $\Theta(x^2)$ bound, but it still has several weaknesses. The first (and most obvious) weakness is that we are still not achieving the ideal bound of $\tilde{O}(x)$. The second weakness is that, although the result applies to hovering workloads, it doesn't generalize to *arbitrary* workloads, as can be seen with the following pathological example: consider a workload in which every rebuild window consists of $R-1$ insertions followed by R deletions followed by 1 insertion. The first $R - 1$ insertions in each rebuild window cannot benefit at all from tombstones and thus necessarily incur $\Theta(x^2)$ expected time each.

It turns out that both of these weaknesses can be removed if we simply use a larger rebuild window size R. Intuitively, the larger the R, the more time there is for tombstones to accumulate and the better the insertions perform. On the other hand, tombstone accumulation is *precisely* the reason that R is classically set to be small, since it breaks the classical analysis and potentially tanks the performance of queries.

We show that the sweet spot is to set $R = n/\operatorname{polylog}(x)$. Here, the expected amortized cost per insertion drops all the way to $\tilde{O}(x)$, while queries continue to take expected time $O(x)$. Once again, and somewhat surprisingly, the low-order factors are an artifact of reality rather than merely the analysis: no matter the value of R used, either the average insertion cost or the average query cost must be $\omega(x)$.

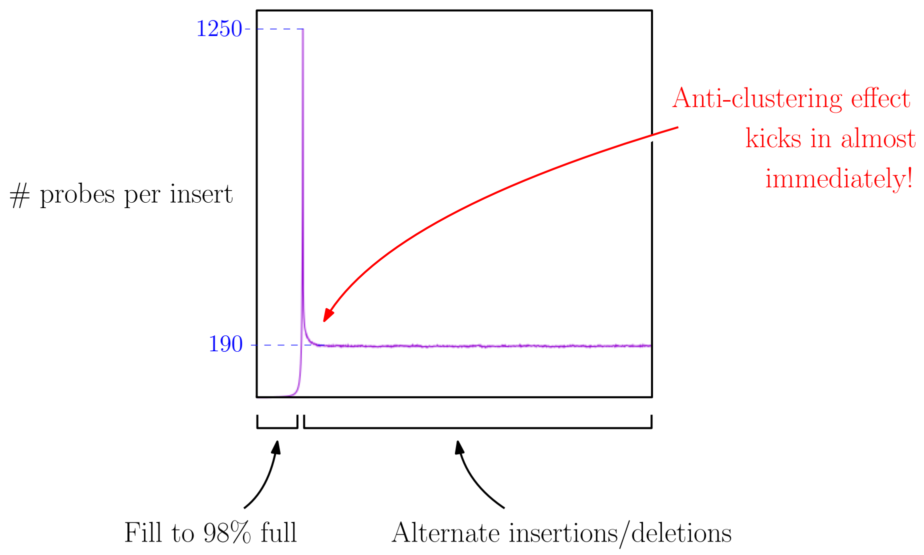


Figure 2-1: A graph of insertion time in a linear-probing hash table (without rebuilds) as (1) the hash table is filled from empty to 98% full, and then (2) an alternating sequence of random insertions/deletions is performed. Note that, if we were to include rebuilds, then after each rebuild, the insertion time would once again begin at the top of the spike.

The bound of $\tilde{\Theta}(x)$ holds not only for hovering workloads, but also for *any workload* that maintains a load factor of at most $1 - 1/x$. Note that here we are analyzing a table in which the capacity n is fixed, and the load factor is permitted to vary over time. It's interesting to think about how the pathological case described above is avoided here. Because the rebuild window is so large, the only way there can be a long series of insertions without deletions is if most of them are performed at low load factors, meaning that they are not slow after all.

It is illustrative to see what our $\tilde{\Theta}(x)$ result looks like in an actual linear-probing hash table. Figure 2-1 graphs the average time per insertion as (1) the hash table is filled from empty to 98% full, and then (2) an alternating sequence of random insertions/deletions is performed. As the hash table is filled up, the insertion time grows exactly as the classical analysis predicts. But, as soon as the hovering workload begins, the insertion time drops back down, bringing the amortized cost to $\tilde{O}(x)$. What's interesting about the hovering workload is that the insertions and deletions are one-for-one, so although the deletions have an anti-clustering effect, the insertions also have a clustering effect. Intuitively, one might expect the two effects to cancel, so that the insertion time would plateau at $\Theta(x^2)$. What our results reveal is that this is not the case: the combinatorial interactions between insertions and deletions actually leads to a strong *net* anti-clustering effect.

A surprising lesson: linear probing is already faster than we thought. The core lesson of our results is that linear probing is far less affected by primary clustering than the classical analysis would seem to suggest. Although the classic $\Theta(x^2)$ bound is mathematically correct, it does not accurately represent the amortized cost of insertions at high load factors. This suggests that the conclusions that are taught in courses and textbooks, namely that linear probing scales poorly to high load factors, and that alternatives with less data locality such as quadratic probing should be used in its place, stem in part from an incomplete understanding of linear probing and warrant revisiting.

The second lesson is that small implementation decisions can substantially change performance. From a software engineering perspective, our results suggest two simple optimizations (the use of tombstones and the use of large rebuild windows) that should be considered in any implementation of linear probing.

Interestingly, tombstones (and even relatively large rebuild windows) are already present in some hash tables. Thus, one interpretation of primary anti-clustering is as a phenomenon that, to some degree, already occurs around us, but that until now has gone apparently unnoticed.

Chapter 4. Graveyard hashing, an Ideal Linear-Probing Hash Table

Our final result is a new version of linear probing, which we call ***graveyard hashing***, that fully eliminates primary clustering on any sequence of operations. The key insight is that, by artificially inserting extra tombstones (that are not created by deletions), we can ensure that every insertion has good expected behavior.

Insertions, deletions, and queries are performed in exactly the same way as for standard ordered linear probing. The difference is in how we implement rebuilds. In addition to cleaning out tombstones, rebuilds are now also responsible for inserting $\Theta(n/x)$ *new tombstones* evenly spread across the key space.

Graveyard hashing adapts dynamically to the current load factor of the table, performing a rebuild every time that x changes by a constant factor. The running time of each operation is a function of whatever the load factor is at the time of the operation. If a query/insert/delete occurs at a load factor of $1 - 1/x$, then it takes expected time $O(x)$ (even in insertion-only workloads). Graveyard hashing can also be implemented to resize dynamically, so that it is always at some target load factor of $1 - \Theta(1/x)$.

We remark that there is an important sense in which our results on ordered linear probing are orthogonal to our results on graveyard hashing. One set of results revisits the question of how classical versions of linear probing behave, while the other answers the question as to whether it is possible to design new versions of linear probing that do even better.

Coming full circle: improved external-memory hashing. As we mentioned at the outset, one of the big advantages of linear probing is its data locality (e.g., good cache or I/O performance).

Data locality is formalized via the external-memory model, first introduced by Aggarwal and Vitter in 1988 [42]: a two-level memory hierarchy is comprised of a small, fast ***internal memory*** and a large, slow ***external memory***; blocks can only be read and modified when they are in internal memory, and the act of copying a block from external memory into internal memory is referred to as a ***block transfer***. The model has two parameters, the number B of records that fit into each block and the number M of records that fit in internal memory. Performance is measured by the number of block transfers that a given algorithm or data structure incurs. This model can be used to capture an algorithm's I/O performance (internal memory is RAM, external memory is disk, and block transfers are I/Os) or cache performance (internal memory is cache, external memory is RAM, and block transfers are cache misses).

Ideally, a hash table incurs only amortized expected $1 + o(1)$ block transfers per operation, even when supporting a high load factor $1 - 1/x$.[8] We call the problem of achieving these guarantees the ***space-efficient external-memory hashing problem***. Standard linear probing is a solution when either x is a (small) constant, or the block size B is very large ($B = \omega(x^2)$), but otherwise, due to primary clustering, it is not [302].

For $B \neq \omega(x^2)$, the state of the art for space-efficient external-memory hashing is due to Jensen and Pagh [216]. They give an elegant construction showing that, if the block size B is $\Theta(x^2)$, then it is possible to achieve amortized expected $1 + O(1/x)$ block transfers per operation, while maintaining a load factor of $1 - O(1/x)$. (In contrast, standard linear probing requires $B = \Omega(x^3)$ to achieve the same $1 + O(1/x)$ result.) However, if $B = o(x^2)$, then no solutions to the problem are known.

Graveyard hashing enables linear probing to be used directly as a solution to the space-efficient external-memory hashing problem, matching Jensen and Pagh's bound for $B = \Theta(x^2)$, and offering an analogous guarantee for arbitrary block sizes $B > \omega(x)$. If $B = \Theta(xk)$ for some $k > 1$, then the amortized expected cost of each operation is $1 + O(1/k)$ block transfers. This means that, even if the block size B is only *slightly* larger than the load factor parameter x, we still get $1 + o(1)$ block transfers per operation.

Additionally, graveyard hashing is cache oblivious [189, 302], meaning that the block size B need not be known by the data structure. Consequently, if a system has a multi-level hierarchy of caches, each of which may have a different set of parameters B and M, then the guarantee above applies to every level of cache hierarchy.

[8]Many hash tables that are otherwise very appealing perform poorly on this front. For example, if one uses cuckoo hashing, then negative queries require ≥ 2 block transfers, and insertions at high load factor require $\omega(1)$ block transfers [161, 182].

2.1 Notation and Conventions

We say that an event occurs with probability $1 - 1/\operatorname{poly}(j)$ for some parameter j if, for any positive constant c, the event occurs with probability $1 - O(1/j^c)$.[9] Throughout, we use standard interval notation, where $[m]$ means $\{1, 2, \ldots, m\}$, $[i, j]$ means $\{i, i+1, \ldots, j\}$, $(i, j]$ means $\{i+1, i+2, \ldots, j\}$, etc.

When discussing an ordered linear probing hash table, we use n to denote the number of ***slots*** (i.e., ***positions***). We use $1 - 1/x$ to refer to the ***load factor*** (i.e., the fraction of slots that are taken by elements), and R to refer to the ***rebuild-window size*** (i.e., the number of insertions that must occur before a rebuild is performed). We use S to denote the sequence of operations being performed, and we refer S as the ***workload***.

The operations on the hash table make use of a ***hash function*** $h : U \to [n]$, where U is the universe of possible ***keys*** (also known as ***records*** or ***elements***). We shall assume that h is uniform and fully independent, but as we discuss in later sections, our results also hold for natural families of hash functions such as tabulation hashing (and, for the analysis of graveyard hash tables, also 5-independent hashing). We can also refer to the ***hash*** $h(u)$ of either a tombstone u (i.e., the hash of the element whose deletion created u) or of an operation u (i.e., the hash of the key that the operation is inserting/deleting/querying).

Each slot in the hash table can either contain a key, be empty (i.e., a free slot), or contain a tombstone. Any maximal contiguous sequence of non-empty slots forms a ***run***. With ordered linear probing, the keys/tombstones in each run are always stored in order of their hash.

Our analysis will often discuss ***sub-intervals*** $I = [i, j] \subseteq [n]$ of the slots in the hash table. We say that an element u ***hashes to*** I if $h(u) \in I$ (but this does not necessarily mean that u resides in one of the slots I). We say that an interval I is ***saturated*** if it is a subset of a run.

Formally, operations in an ordered linear probing hash table are implemented as follows. A query for a key u examines positions $h(u), h(u) + 1, \ldots$ until it either finds u (in which case the query returns true), finds an element with hash greater than $h(u)$ (in which case the query returns false), or finds a free slot (in which case the query also returns false). A deletion of a key u simply performs a query to find the key, and then replaces it with a tombstone. Finally, an insertion of a key u examines positions $h(u), h(u) + 1, \ldots$ until it finds the position j where u belongs in the run; it then inserts u into that position, and shifts the elements in positions $j, j+1, j+2, \ldots$ each to the right by one until finding either a tombstone or a free slot. We say that the insertion ***makes use*** of that tombstone/free slot. Finally, rebuilds are performed every R insertions, and a rebuild simply restructures the table to remove all tombstones. Throughout, n is fixed and does not get changed during rebuilds, although when we describe graveyard hashing (Chapter 4), we also give a version that dynamically resizes n.

[9] The constants used to define the event may depend on c.

There are two ways to handle overflow off the end of the hash table. One option is to wrap around, meaning that we treat 1 as being the position that comes after n; the other is to extend the table by $o(n)$ (i.e., it ends in slot $n + o(n)$), so that operations never fall off the end of the table. Both solutions are compatible with all of our results. For concreteness, we assume that the wrap-around solution is used, but to simplify discussion, we treat the slots that we are analyzing as being sufficiently towards the middle of the table that we can use the $<$-operator to compare slots.

Chapter 3

The Surprisingly Strong Anti-Clustering Effects of Tombstones

In this chapter, we demonstrate the anti-clustering effects of tombstones. We will prove two main theorems. The first considers a hovering workload, that alternates between insertions and deletions. We obtain nearly tight bounds on the expected amortized time per operation as a function of the rebuild window size R. In the classical setting of $R = \Theta(n/x)$, we find that the amortized insertion time becomes $\tilde{O}(x^{1.5}) \cap \Omega(x^{1.5}\sqrt{\log \log x})$.

Theorem 1. *Consider an ordered linear probing hash table that uses tombstones for deletions, and that performs rebuilds every R insertions. Suppose that the table is initialized to have capacity n and load factor $1 - 1/x$, where $R = \Omega(n/x)$ and $R \leq n$. Finally, consider a sequence S of operations that alternates between insertions and deletions (and contains arbitrarily many queries).*

Then the expected amortized time I spent per insertion satisfies

$$I \leq \tilde{O}\left(x\sqrt{\frac{n}{R}}\right) \tag{3.1}$$

and, if all insertions/deletions in each rebuild window are on distinct keys, then

$$I \geq \Omega\left(x\sqrt{\frac{n}{R}\log \log x}\right). \tag{3.2}$$

Moreover, the expected time Q of a given query/deletion satisfies

$$Q \leq O(x) + \tilde{O}\left(x\sqrt{\frac{R}{n}}\right) \tag{3.3}$$

and, if all operations in each rebuild window are on distinct keys, then for any negative *query at the end of a rebuild window, we have*

$$Q \geq \Omega\left(x + x\sqrt{\frac{R}{n}\log\log x}\right). \tag{3.4}$$

One consequence of Theorem 1 is that, for any choice of R, there is a workload that forces an amortized expected time of $\omega(x)$. At the same time, we can get *remarkably close* to $O(x)$: if we set $R = n/\operatorname{polylog}(x)$, then the average insertion time becomes $\tilde{O}(x)$, while the query time remains $O(x)$.

In this optimal parameter regime (i.e., $R = n/\operatorname{polylog}(x)$), we can extend our results to apply not just to hovering workloads, but to any sequence of insertions/deletions in which the load factor never exceeds $1 - 1/x$. This leads to the second theorem of the chapter:

Theorem 2. *Let c be a sufficiently large positive constant. Consider an ordered linear probing hash table that uses tombstones for deletions, and that performs rebuilds every $R = n/\log^c x$ insertions. Finally, consider a sequence of operations S that never brings the load factor above $1 - 1/x$.*

Then the expected amortized cost of each insertion is $\tilde{O}(x)$ and the expected cost of each query/deletion is $O(x)$.

Combined, Theorems 1 and 2 tell a remarkable story: tombstones, which historically have been viewed as a minor implementation detail, completely change the asymptotic behavior of the linear-probing hash table. Of course, what we will see in Chapter 4 is that we can go even further—by strategically adding additional tombstones into the hash table, we will be able to achieve $O(x)$ expected time per insertion for any workload (including insertion-only ones).

In the rest of the chapter, we prove Theorems 1 and 2. The proofs will require a great deal of technical machinery. Thus, we begin with a technical overview of the entire analysis in Section 3.1

3.1 Technical Overview

In this section, we give a technical overview of our analysis of ordered linear probing. We begin by describing the intuition behind Knuth's classic $\Theta(x^2)$ bound. We then turn our attention to sequences of operations that contain deletions, and show that the tombstones left behind by those deletions have a *primary-anti-clustering effect*, that is, they have a tendency to speed up future insertions. One of the interesting components of the analysis is that we perform a series of problem transformations, taking us from the question of how to analyze ordered linear probing to a seemingly very different question involving the combinatorics of monotone paths on a grid. By applying geometric arguments to the latter, we end up being able to achieve nearly tight bounds for the former.

3.1.1 Understanding the Classic Bounds: A Tale of Standard Deviations

Suppose we fill an ordered linear-probing hash table from empty up to a load factor of $1-1/x$. Knuth [224] famously showed that the final insertion in this procedure takes expected time $\Theta(x^2)$. As discussed in the introduction, the fact that the insertion takes time $\omega(x)$ can be attributed to primary clustering.

But why does the running time end up being $\Theta(x^2)$ specifically? This turns out to be a result of how standard deviations work. Consider an interval I of x^2 slots in the hash table. The expected number of items that hash into I is $(1-1/x)\,x^2 = |I| - x$. On the other hand, the standard deviation for the number of such items is $\Theta(x)$. It follows that, with probability $\Omega(1)$, the number of items that hash into I is $\Omega(x)$ greater than $|I|$. If we then consider the interval I' consisting of the x^2 slots that follow I, then this interval I' must handle not only the items that hash into it, but also the overflow elements from I. The result is that, with probability $\Omega(1)$, the interval I' is fully saturated and forms a run of length x^2.

The above argument stops working if we consider intervals of size $\omega(x^2)$, because the standard deviation on the number of items that hash into I stops being large enough to overflow the interval. The result is that runs of length $\Theta(x^2)$ are relatively common, but that longer runs are not. This is why the expected running time of the insertion performed at load factor $1-1/x$ is $\Theta(x^2)$.

Now suppose that, after reaching a load factor of $1-1/x$, we perform a query in our ordered linear-probing hash table. Unlike an insertion, which takes expected time $\Theta(x^2)$, the query takes expected time $\Theta(x)$. We can again see this by looking at standard deviations.

If the query hashes to some position j and takes time t, then there must be at least t elements u that have hashes $h(u) \le j$ but that reside in positions j or larger. Hence, there is some sub-interval $I = [j_0, j]$ of the hash table (ending in position j) that has overflowed by at least $t-1$ elements, that is, the number of elements that hash to I is at least $|I| + t - 1$. As before, the interval that matters most ends up being the one of size $\Theta(x^2)$, and the amount by which it overflows in expectation is proportional to the standard deviation $\Theta(x)$ of the number of items that hash into the interval.

Thus, the running times of both insertions and queries are consequences of the same two facts: that (a) for any interval I of size x^2, there is probability $\Omega(1)$ that the interval overflows by $\Theta(x)$ elements; and that (b) a given interval of size $\omega(x^2)$ most likely doesn't overflow at all. The only difference is that the running time of an insertion is proportional to the *size* of the interval that overflows (i.e., $\Theta(x^2)$), but the running time of a query is proportional to the *amount* by which the interval overflows (i.e., $\Theta(x)$).

3.1.2 Analyzing Primary Anti-Clustering with Small Rebuild Windows

In this subsection, we consider a hovering workload, that is, a sequence of operations that alternates between insertions and deletions on a table with load factor $1 - 1/x$, and we set the size of the rebuild window to be $R = n/(2x)$ (i.e., the value that it is classically set to). Our task is to consider a sequence of $R = n/(2x)$ insertion/deletion pairs between two rebuilds, and to analyze the amortized running times.

Analyzing displacement instead of running time. Define the ***peak*** p_u of an insertion u to be either the hash of the tombstone that the insertion uses (if the insertion makes use of a tombstone) or the position of the free slot that the insertion uses (if the insertion makes use of a free slot). Define the ***displacement*** d_u of an insertion to be $d_u = p_u - h(u)$.

One of the subtleties of how displacement is defined is that, if an insertion u uses a tombstone v, then the displacement measures the difference between $h(u)$ and the *hash* $h(v)$, rather than the difference between $h(u)$ and the *position* of v. This ends up being important for how displacement is used in the analysis[1], but it also means that the displacement of an insertion can potentially be substantially smaller than the running time. For example, if the insertion hashes to position 7 and makes use of a tombstone with hash 13 that resides in position 54, then the displacement is only $13 - 7 = 6$ but the running time is proportional to $54 - 7 = 47$.

Although we skip the proof for now (see Lemma 33), it turns out that one can bound the expected difference between displacement and running time by $O(x)$. Thus, even though displacement is not always the same as running time, any bound on average displacement also results in a bound on average running time.

Relating displacement to crossing number. Rather than analyzing the displacement of each *individual* insertion, we bound the average displacement over all R insertions in the rebuild window by relating the displacements to another set of quantities that we call the ***crossing numbers*** $\{c_j\}_{j \in [n]}$. The crossing number c_j counts the number of insertions u in the rebuild window that have a hash $h(u) < j$ but that have a peak $p_u \geq j$ (we consider even the insertions that are subsequently deleted). Each insertion u increments d_u different crossing numbers c_j. Thus

$$\sum_u d_u = \sum_{j \in [n]} c_j.$$

Because we are analyzing the insertions in a rebuild window of size R, the summation

[1]The reason for this is actually very simple. Whenever a deletion v is performed, the value of $h(v)$ is fixed (it depends only on the hash function) but the position of v is not (it depends on the other elements in the table, and will change over time). Thus, it is cleaner to measure the deletion's effect on future insertions in terms of $h(v)$ (the thing that is fixed) rather than v's position (the thing that is not).

on the left side has $R = \Theta(n/x)$ terms, while the summation on the right side has n terms. Thus, if we consider a random insertion u and a random position j, then

$$\mathbb{E}[d_u] = \Theta(x\mathbb{E}[c_j]).$$

If our goal is to establish that the average insertion takes time $o(x^2)$, then it suffices instead to show that the average crossing number c_j is $o(x)$.

Notice that the ratio between $\mathbb{E}[d_u]$ and $\mathbb{E}[c_j]$ is a function of the rebuild window size R; this will come into play later when we consider larger rebuild windows.

Capturing the dependencies between past insertions/deletions. What makes the analysis of a given crossing number c_j interesting is the way in which insertions and deletions interact over time. If an insertion u has hash $h(u) < j$, and there is a tombstone v with hash $h(v) \in [h(u), j)$, then u can make use of the tombstone and avoid contributing to c_j. But, in order to determine whether a given tombstone v is present during the insertion u, we must know whether any past insertions have already used v. That, in turn, depends on which tombstones were present during past insertions, resulting in a chain of dependencies between operations over time.

A key insight is that the interactions between insertions and deletions over time can be reinterpreted as an elegant combinatorial problem about paths on a two-dimensional grid. We now give the transformation.

The geometry of crossing numbers. For the sake of analysis, define Z to be the state that our table would be in if we performed only the insertions in the rebuild window and not the deletions.

Since $R = n/(2x)$, the load factor of Z is at most $1 - 1/x + R/n = 1 - 1/(2x)$, which by the classic analysis of linear probing means that the expected distance from any position to the next (and previous) free slot is $O(x^2)$.

Now consider some crossing number c_j. Let j' be the position of the closest free slot to the left of j in Z, that is, the largest $j' < j$ such that position j' is a free slot in Z. Then the only insertions/deletions that we need to consider when analyzing c_j are those that hash into the interval $I = [j' + 1, j)$.

We know from our analysis of Z that $\mathbb{E}[|I|] = O(x^2)$. Although $|I|$ is a random variable with mean $\Theta(x^2)$, to simplify our discussion in this section, we shall treat $|I|$ as simply deterministically equaling x^2. We also treat I as containing no free slots (even at the beginning of the time window being considered). In particular, we know from the classical analysis of linear probing that any interval I of size x^2 has probability $\Omega(1)$ of containing no free slots, so there is no point in trying to make use of potential free slots in I for our analysis.

We can visualize the insertions and deletions that hash into I by plotting them in a two-dimensional grid, as in Figure 3-1a. The vertical axis represents time flowing up from 1 to $2R$, and the horizontal axis represents the hash locations in the interval I. We draw a blue dot in position (i, t) if the t-th operation is an insertion with hash

$i \in I$, and we draw a red dot in position (i, t) if the t-th operation is a deletion with hash $i \in I$. (Note that most operations do not hash to I and thus do not result in any dot.)

In order for a given insertion u to be able to make use of a given deletion v's tombstone, it must be that (a) $h(u) \leq h(v)$;[2] that (b) u occurs temporally after v; and that (c) v's tombstone is not used by any other insertion temporally before u. The first two criteria (a) and (b) tell us that for a given insertion (i.e., blue dot) in the grid, the insertion can only make use of tombstones from deletions (i.e., red dots) that are below it and to its right.

Define the set of ***monotone paths*** through the grid to be the set of paths that go from the bottom left to the top right of the grid, and that never travel downward or leftward. Define the ***blue-red deviation*** of such a path to be the number of blue dots below the path minus the number of red dots below the path (see Figure 3-1a for an example).

What do these monotone paths have to do with the crossing number c_j? Monotone paths with large blue-red deviations serve as witnesses for the crossing number c_j also being large. Suppose that there is a monotone path γ with blue-red deviation $r > 0$. Since we assume that I is initially saturated, each of the insertions (i.e., blue dots) below γ must either make use of the tombstone for a deletion (i.e., red dot) that is also below γ or contribute 1 to c_j. Since there are r more blue dots than red dots below γ, it follows that $c_j \geq r$.

In fact, this relationship goes in both directions (although the other direction requires a bit more work; see Lemma 17). If the crossing number c_j takes some value r, then there must also exist some monotone path with blue-red deviation at least r. The result is that, if we wish to prove either upper or lower bounds on $\mathbb{E}[c_j]$, it suffices to instead prove bounds on the largest blue-red deviation of any monotone path through the grid.

Formalizing the blue-red deviation problem. Let us take a moment to digest the combinatorial problem that we have reached, since on the face of things it is quite different from the problem that we started at.

The expected number of blue dots (and also of red dots) in our grid is $R \cdot |I|/n = \Theta(x)$. If we break the grid into $\sqrt{x}$ rows and $\sqrt{x}$ columns (see Figure 3-1b for an example), then each cell of the broken-down grid expects to contain $\Theta(1)$ blue and red dots. To simplify our discussion here, think of each cell as independently containing a Poisson random variable $\text{Pois}(1)$ number of blue dots and a Poisson random variable $\text{Pois}(1)$ number of red dots.[3]

Furthermore, rather than considering all monotone paths through the grid, we can

[2]Recall that ordered linear probing only ever moves elements to the right over time, meaning that a given insertion u will only use a tombstone if that tombstone has hash at least $h(u)$.

[3]This allows for us to ignore two minor issues in our discussion here: (1) the fact that a single key can potentially be inserted, deleted, and reinserted, resulting in blue and red dots whose horizontal coordinates are deterministically equal; and (2) the fact that the numbers of blue/red dots in each cell are actually very slightly negatively correlated.

restrict ourselves exclusively to the paths that stay on the row and column lines that we have drawn (for an example, see Figure 3-1b). With high probability in x, this restriction changes the maximum blue-red deviation of any path by at most $\tilde{O}(\sqrt{x})$.

In summary, we have a $\sqrt{x} \times \sqrt{x}$ grid where each cell of the grid contains a Poisson random variable Pois(1) number of blue points (resp. red points). Whereas our original grid was much taller than it was wide (its height was $2R$ and its width was x^2), our new grid is a $\sqrt{x} \times \sqrt{x}$ square. There are $\binom{2\sqrt{x}}{\sqrt{x}} = \exp(\Omega(x))$ monotone paths γ through the grid, and we wish to prove bounds on the maximum blue-red deviation D achieved by any such path.

Gaining intuition: how blue-red deviations behave. To gain intuition, let us start by considering the trivial path γ that contains the entire grid beneath it. The expected number of blue dots (as well as red dots) beneath γ is $\Theta(x)$, and the standard deviation on the number of blue dots (as well as red dots) is thus $\Theta(\sqrt{x})$. With probability $\Omega(1)$, there are $\Omega(\sqrt{x})$ more blue dots in the grid than red dots, which results in a blue-red deviation of $\Omega(\sqrt{x})$ for γ.

Of course, that's just the blue-red deviation of a *single fixed path*. What should we expect the *maximum* blue-red deviation D over all paths to be? On one hand, there are exponentially many paths that we must consider, but on the other hand, the blue-red deviations of the paths are closely correlated to one another. The result, it turns out, is that D ends up being an $x^{o(1)}$ factor larger than $\sqrt{x}$.

We will show that, with probability $1 - o(1)$, the maximum blue-red deviation D is between $\Omega(\sqrt{x \log\log x})$ and $\tilde{O}(\sqrt{x})$. If we backtrack to our original problem (i.e., we relate the blue-red deviations to the crossing numbers, the crossing numbers to the displacements, and the displacements to the running times), we get that the amortized cost of insertions is between $\Omega(x^{1.5}\sqrt{\log\log x})$ and $\tilde{O}(x^{1.5})$.

An upper bound of $\tilde{O}(\sqrt{x})$ on the maximum blue-red deviation D. The first step in bounding D is to prove a general result about decompositions of monotone paths. Let $k = 4\sqrt{x}$ be the perimeter of the grid. We claim that for any monotone path γ through the grid, it is always possible to decompose the area under γ into disjoint rectangles $R_1, R_2, \ldots$ such that the sum of the perimeters of the rectangles is at most $k \log k$.

Such a decomposition can be constructed recursively as follows: (1) find the point q halfway along the path, and drop a rectangle from q to the bottom-right-most point in the grid; (2) then recursively construct a rectangular decomposition for the portion of the path prior to q, and recursively construct a rectangular decomposition for the portion of the path after q. (See Figure 3-1c for an example.)

The recursive decomposition is designed so that, in the i-th level of recursion, each recursive subproblem takes place on a grid with perimeter exactly $k/2^i$. Since there are at most 2^i subproblems in each level of recursion, each of which contributes a rectangle with perimeter at most $k/2^i$, the sum of all the rectangle perimeters over all levels of recursion is at most $k \log k$.

The next step in the analysis is to consider the maximum amount that any given rectangle can contribute to the blue-red deviation of a path. If a given rectangle has area a, then the expected number of blue/red dots in the rectangle is $\Theta(a)$, and with high probability in x, the rectangle as a whole has blue-red deviation $O(\sqrt{a}\log x)$. This, in turn, means that if a rectangle has perimeter p, then with high probability in x, it has blue-red deviation at most $O(p \log x)$. Finally, since there are only $O(x^2)$ possible rectangles in the entire grid, and this property holds for each of them with probability $1 - 1/\operatorname{poly}(x)$, the property also holds simultaneously for all of them with probability $1 - 1/\operatorname{poly}(x)$.

Putting the pieces together, we know that every path γ has a rectangular decomposition such that the sum of the rectangle perimeters is $O(\sqrt{x}\log x)$. We further know that, if a rectangle in the decomposition has perimeter p, then it contributes at most $O(p \log x)$ to the blue-red deviation of γ. It follows that the total blue-red deviation of any path γ is at most $O(\sqrt{x}\log^2 x)$.

A lower bound of $\Omega(\sqrt{x \log\log x})$ on the maximum blue-red deviation D. Now we turn our attention to proving a lower bound on D. We wish to find a monotone path γ whose blue-red deviation is $\Omega(x^{1.5}\sqrt{\log\log x})$.

Call a $j \times j$ square within the grid ***high-value*** if it has blue-red deviation at least $\Omega(j\sqrt{\log\log x})$. The definition is designed so that every square has probability at least $1/\sqrt{\log x}$ of being high-value.

We construct a monotone path γ recursively as follows. First break the grid into quadrants, and check whether the top left quadrant is a high-value square. If so, then return the trivial path that contains the entire grid below it. Otherwise, recursively construct a path through the bottom-left quadrant, recursively construct a path through the top-right quadrant, and set γ to be the concatenation of the two paths.[4] (For an example, see Figure 3-1d.)

There are two types of base cases in the recursion. The first type is when a subproblem finds a high-value square; we call this a ***successful base case***. The second type is when a subproblem terminates because it is on a 1×1 grid; we call this a ***failed base case***.

If a successful base case takes place on a sub-grid of width w, then the high-value square that it discovers contributes $\Omega(w\sqrt{\log\log x})$ to the total blue-red deviation of γ. In order to establish a lower bound of $\Omega(\sqrt{x \log\log x})$ on the blue-red deviation, it therefore suffices to show that the sum of the widths of the successful base cases is $\Omega(\sqrt{x})$.[5]

By construction, the sum of the widths of both the successful base cases and the failed base cases is exactly $\sqrt{x}$. Moreover, each failed base case has width exactly 1.

[4]When we recurse, we do not change the threshold $\Omega(j\sqrt{\log\log x})$ that dictates whether a given $j \times j$ square is special; that is, x acts as a global variable setting this threshold.

[5]There is also a large portion of the area underneath γ that is not contained in any of the high-value squares of the subproblems. Technically, we must also ensure that the blue/red dots in this unaccounted-for area do not substantially change the blue-red deviation, but this follows from a straightforward Chernoff bound.

Thus our task reduces to bounding the number of failed base cases by $o(\sqrt{x})$ with probability $1 - o(1)$.

In order for a given failed base case to occur, there is a recursion path of $\Theta(\log x)$ sub-problems that must all fail to find a high-value square. Each of these failures occurs with probability at most $1-1/\sqrt{\log x}$, so the probability of all of them occurring is

$$\left(1 - 1/\sqrt{\log x}\right)^{\Theta(\log x)} = o(1).$$

Since the probability of any given failed base case occurring is $o(1)$, the expected number of failed base cases that occur is $o(\sqrt{x})$. By Markov's inequality, the number of failed base cases is $o(\sqrt{x})$ with probability $1 - o(1)$, as desired.

3.1.3 Stronger Primary Anti-Clustering with Larger Rebuild Windows

In this section, we consider what happens if a larger rebuild window size $R = n/\operatorname{polylog}(x)$ is used. As discussed in the introduction, this allows for us to improve our amortized insertion time from $\tilde{\Theta}(x^{1.5})$ to $\tilde{\Theta}(x)$, while still achieving average query time $\Theta(x)$.

Remarkably, these bounds hold not just for hovering workloads, but also for arbitrary workloads that stay below a load factor of $1 - 1/x$. To simplify discussion in this section, however, we continue to focus on the hovering case.

There are two main technical challenges that our analysis must overcome. The first challenge is obvious: we must quantify the degree to which tombstones left behind by deletions improve the performance of subsequent insertions. The second challenge is a bit more subtle: in order to support large rebuild-window sizes R, our analysis must be robust to the fact that tombstones can accumulate over time, increasing the effective load factor of the hash table. This latter challenge is further exacerbated by the fact that the choice of which tombstones are in the table at any given moment is a function not only of the sequence of operations being performed, but also of the randomness in the hash table. This means that, even if the cumulative load factor from the elements and tombstones can be bounded (e.g., by $1 - \Theta(1/x)$), we still cannot analyze the tombstones as though they were normal elements; a consequence of this is that we cannot even apply the classic analysis to deduce an $O(x^2)$-time bound for insertions or an $O(x)$-time bound for queries.

Using crossing numbers to rescue the queries. Let us consider the time that it takes to query an element whose hash is j. The time is proportional to the number of elements and tombstones that have hashes smaller than j but that reside in positions j or larger.[6] Call the slots containing these elements ***j-crossed***. Right after a rebuild is performed, the number s of j-crossed slots has expected value $O(x)$. Over the course

[6]Technically, we must also consider elements that hash to exactly j but the expected number of such elements is $O(1)$.

of the time window between consecutive rebuilds, however, the quantity s gradually increases.

Fortunately, the amount by which s increases is *precisely* the crossing number c_j. Indeed, c_j gets incremented exactly whenever a formerly non-j-crossed slot becomes j-crossed.

Thus, if we can bound c_j to be small then we hit two birds with one stone: we are able to bound the running times of both queries and insertions.

But how do we rescue the crossing numbers? Large rebuild windows also break the analysis of crossing numbers, however. In the original analysis, we argued that there is most likely some position $j' < j$ satisfying $j - j' = O(x^2)$ such that position j' remains an empty slot throughout the entire rebuild window. This meant that, when considering c_j, we only had to analyze insertions and deletions that hash into the interval $I = [j' + 1, j)$ of size $\Theta(x^2)$.

We can no longer argue that such a j' necessarily exists, however, since the accumulation of tombstones over time might eliminate all of the free slots near position j. Thus we must extend our analysis to consider intervals I of size $\omega(x^2)$.

Fortunately, if we consider any interval I of size at least $x^2 \operatorname{polylog} x$, then we can argue that the interval most likely *initially* contains $\Omega(|I|/x)$ free slots. Define the ***insertion surplus*** of an interval I to be the maximum blue-red deviation of any monotone path through the grid representing I, *minus* the number of free slots initially in I. We prove that the crossing number c_j is exactly equal to the maximum insertion surplus of any interval I of the form $[j' + 1, j)$. Since large intervals I have a $\Theta(1/x)$-fraction of their slots initially empty, it is very unlikely that they end up determining the crossing number c_j. The result is that we can again focus primarily on intervals of size $O(x^2)$, and perform the analysis of c_j as before.

Putting the pieces together for $R = n/\operatorname{polylog}(x)$. We now analyze the case of $R = n/\operatorname{polylog}(x)$. As before, we use the displacement d_u for an insertion as a proxy for insertion time (although bounding the difference between the two requires a more nuanced argument than before, see Lemmas 25 and 26). We can then relate the displacements to the crossing numbers by

$$\sum_u d_u = \sum_{j \in [n]} c_j.$$

Now, however, both sums consist of between $n/\operatorname{polylog} x$ and n terms. This means that for a random insertion u and a random $j \in [n]$,

$$\mathbb{E}[d_u] \leq \mathbb{E}[c_j \operatorname{polylog}(x)].$$

As before, we can transform the problem of bounding c_j into the problem of bounding the maximum blue-red deviation of any monotone path in a certain grid. The expected number of blue/red dots in the grid is now $x^2/\operatorname{polylog}(x)$ (rather than $\Theta(x)$). Thus our bound on blue-red deviation comes out as $\tilde{O}(x/\operatorname{polylog}(x)) = O(x)$ (rather than $\tilde{O}(\sqrt{x})$). This means that $\mathbb{E}[c_j] = O(x)$, which implies that the expected time taken by any query is also $O(x)$ and that the average time taken by each insertion is $\tilde{O}(x)$.

We can now also see why $n/\operatorname{polylog}(x)$ is the right rebuild window size to use. In particular, if we make R smaller than $n/\operatorname{poly}(x)$, then insertion times suffer, but if we let R get too close to n (or exceed n) then the crossing numbers c_j become $\omega(x)$ (thanks to our lower bound construction on blue-red deviations), and thus queries take time $\omega(x)$. Thus it is impossible to select a value for R that achieves expected time $O(x)$ for all operations, and if we want $O(x)$-time queries, we must make $R = o(n)$.

Analyzing arbitrary workloads. Finally, we generalize these results to *any* sequence of operations that stays below a load factor of $1 - 1/x$. We argue that, within any rebuild window, it is possible to re-organize the operations in such a way that (a) none of the crossing numbers decrease, and (b) the operations consist of a series of insertions, followed by a series of alternating insertions/deletions, followed by a series of deletions (see Proposition 24). The alternating insertions/deletions can be analyzed as above, and because R is large, the crossing-number cost of the initial insertions can be amortized away. The fact that the re-organization of operations does not decrease any crossing numbers ends up being easy to prove using our characterization of crossing numbers in terms of insertion surpluses of intervals.

3.2 Some Basic Balls-and-Bins Lemmas

We begin by proving several basic lemmas having to do with balls and bins. What makes these lemmas different from standard balls-and-bins bounds is that they consider all prefixes of a sequence of bins, bounding the probability of any prefix behaving abnormally. In order to demonstrate how these lemmas relate to linear probing, we will also use them to reprove the classic bounds on the performance of insertions and queries for ordered linear probing.

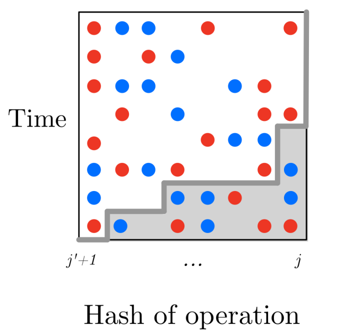
Throughout the rest of the section, consider the setting in which we place $m = \Theta(n)$ balls randomly into n bins. Let $\mu = n/m$ be the expected number of balls in each bin.

Lemma 3. *Let $x > 1$ and $k \geq 1$. With probability $1 - 2^{-\Omega(k)}$, for every $i \geq x^2 k$, the number of balls in the first i bins is between $(1 - 1/x)i\mu$ and $(1 + 1/x)i\mu$.*

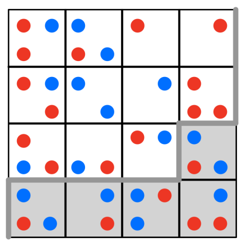
Proof. Let $\ell = x^2 k$. We wish to show that, with probability at least $1 - 2^{-\Omega(k)}$, there is no $i \geq \ell$ such that the first i bins contain either fewer than $(1 - 1/x)i\mu$ balls or more than $(1 + 1/x)i\mu$ balls. We will focus on the more-than-$(1 + 1/x)i\mu$ case, since the fewer-than-$(1 - 1/x)i\mu$ case follows by a symmetric argument.

Let X_i be the indicator random variable for the event that the first i bins contain at least $(1 + 1/x)i\mu$ balls. By a Chernoff bound,

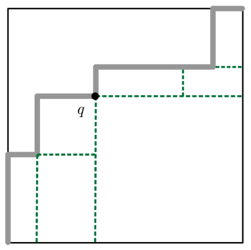
$$\Pr[X_\ell] \leq 2^{-\Omega(k)}.$$



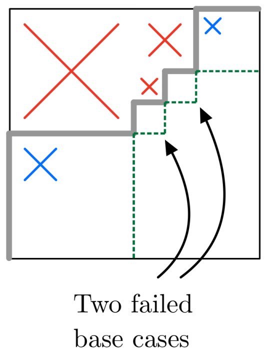
(a) A sample graphic of insertions/deletions in $I = [j'+1, j)$ over time. Blue dots are insertions and red dots are deletions. (To simplify, we plot each point in one of eight y-coordinates, but in reality, each y-coordinate would be distinct.) A monotone path, with blue-red deviation $6-4=2$, is also given.



(b) To simplify the problem, we draw a $\sqrt{x} \times \sqrt{x}$ grid, and consider only paths that go along the drawn grid lines. Here $x = 16$.



(c) An example of a recursively constructed rectangular decomposition for a path. The point q used in the top level of recursion is also labeled.



(d) An example of the construction for a path with high blue-red deviation. For each recursive subproblem whose top-left quadrant is not high-value, we place a red X through the quadrant; for each recursive subproblem whose top-left quadrant is high-value, we place a blue X through the quadrant; the only subproblems not to have an X drawn in them are the failed base cases, each of which takes place on a 1×1 grid. There are four base cases, two of which are failed.

Figure 3-1

It is tempting to apply a similar Chernoff bound to every $i \geq \ell$, and then to take a union bound; but this would yield a bound of $\Pr[X_i \text{ for some } i \geq \ell] \leq \ell 2^{-\Omega(k)}$ rather than the desired bound of $\Pr[X_i \text{ for some } i \geq \ell] \leq 2^{-\Omega(k)}$. Thus a slightly more delicate approach is needed.

To complete the proof, we prove directly that

$$\Pr[X_i \text{ for some } i \geq \ell] \leq O(\Pr[X_\ell]), \tag{3.5}$$

which we have already shown to be $2^{-\Omega(k)}$.

Suppose that X_i holds for some $i \geq \ell$ and let j be the largest such i. The number B of balls in the first j bins must satisfy $B \geq (1 + 1/x)j\mu$. Conditioning on j and on B, each of the B balls independently has probability ℓ/j of being in one of the first ℓ bins. Thus (still conditioning on j and B), the number of balls in the first ℓ bins is a binomial random variable with expected value at least $(1 + 1/x)\ell\mu$. With probability $\Omega(1)$ (and using the fact that $(1 + 1/x)\ell\mu = \Omega(1)$), this binomial random variable takes a value greater than or equal to its mean $(1 + 1/x)\ell\mu$, which implies that X_ℓ occurs. Having established (3.5), the proof is complete. ∎

Corollary 4. *Consider the largest i such that the first i bins contain at least $(1 + 1/x)i\mu$ balls. Then $\mathbb{E}[i] \leq O(x^2)$.*

Lemma 5. *Let $x > 1$ and $k \geq 1$. With probability $1 - 2^{-\Omega(k)}$, there does not exist any i such that the first i bins contain at least $(1 + 1/x)i\mu + kx$ balls.*

Proof. Let Y_i be the indicator random variable for the event that the first i bins contain at least $(1 + 1/x)i\mu + kx$ balls. Let $r = x^2k$. By Lemma 3,

$$\Pr[Y_i \text{ for some } i \geq r] \leq 2^{-\Omega(k)}.$$

Thus it suffices to argue that

$$\Pr[Y_i \text{ for some } i \leq r] \leq 2^{-\Omega(k)}.$$

As in the previous proof, taking a Chernoff bound and then summing over i will not give the result that we are aiming for. Thus a more refined approach is again needed.

Suppose that Y_i occurs for some $i \leq r$, and let j be the largest such i. Let B satisfying $B \geq j\mu + kx$ be the number of balls in the first j bins, and let 2^q be the largest power of two satisfying $2^q \leq j$. Each of the B balls in the first j bins has probability $2^q/j$ of being in the first 2^q bins. Conditioning on a given value of B, we therefore have that the number of balls that land in the first 2^q bins is a binomial random variable with expected value at least $2^q\mu + kx/2$. With probability $\Omega(1)$, this random variable takes a value at least as large as its mean. That is, if we condition on Y_i occurring for some $i \leq r$, then with constant probability there is some power of two $2^q \leq r$ such that at least $2^q\mu + kx/2$ balls land in the first 2^q bins.

For $q \in \{0, 1, \ldots, \log r\}$, let Z_q denote the indicator random variable for the event that at least $2^q\mu + kx/2$ balls land in the first 2^q bins. So far, we have shown that

$$\Pr[Y_i \text{ for some } i \le r] = O(\Pr[Z_q \text{ for some } q \le \log r]).$$

By a Chernoff bound, if we consider a given Z_q, and we define s such that $2^{q+s} = kx$, then

$$\Pr[Z_q] \le \begin{cases} 2^{-\Omega(skx)} & \text{if } s > 0, \\ 2^{-\Omega\left(k^2x^2/2^q\right)} & \text{if } s \le 0. \end{cases}$$

Recalling that $r = x^2k$, it follows that

$$\Pr[Z_q \text{ for some } q \le \log r] \le \sum_{s>0} 2^{-\Omega(skx)} + \sum_{q=\log(kx)}^{\log(kx^2)} 2^{-\Omega\left(k^2x^2/2^q\right)}.$$

The first sum is a geometric series summing to $2^{-\Omega(kx)}$. The second sum is dominated by its final term which is $2^{-\Omega(k)}$. Thus the lemma is proven. ∎

To demonstrate how these results relate to linear probing, we now use them to recreate the classic upper bounds on insertion performance (Proposition 6) and query performance (Proposition 7) for ordered linear probing.

Proposition 6. *Starting with an empty linear-probing hash table of n slots, suppose that we perform $(1 - 1/x)n$ insertions. The running time T of the final insertion satisfies*

$$\Pr[T \ge kx^2] \le 2^{-\Omega(k)}.$$

Proof. If the final insertion hashes to some position p, and takes time $T \ge kx^2$, then the insertion must have been inserted into a run of elements going from position $p - i$ to position $p + T - 1 \ge P + kx^2 - 1$ for some i. Consequently, the number of elements u that satisfy $h(u) \in [p - i, p + kx^2 - 1]$ must be at least $i + kx^2$, even though the expected number of such elements is $(1 - 1/x)(i + kx^2)$. By treating the positions $p + kx^2 - 1, p + kx^2 - 2, \ldots$ as bins, we can apply Lemma 3 to deduce that the probability of such an i existing is at most $2^{-\Omega(k)}$. ∎

Proposition 7. *Consider an ordered-linear-probing hash table with n slots that contains $(1 - 1/x)n$ elements and no tombstones, and suppose we perform a query. The running time T of the query satisfies*

$$\Pr[T \ge kx] \le 2^{-\Omega(k)}.$$

Proof. Notice that we do not have to distinguish between positive and negative queries, since even for a negative query we only need to perform a linear scan until we find a key with a larger hash than the one we are querying.

If the key that we are querying hashes to some position p, and takes time $T \geq kx$, then all of the elements in positions $p, \ldots, p + T - 2 \geq p + kx - 2$ must have hashes at most p. Consider the largest i such that all of positions $p - i, \ldots, p + kx - 2$ contain elements. All of the elements in positions $p - i, \ldots, p + kx - 2$ must have hashes between $p - i$ and p. Thus the total number of elements that hash to positions $[p-i, p]$ must be at least $i + kx - 1$, even though the expected number of elements that hash to those positions is $(1 - 1/x)(i + 1)$. By treating the positions $p, p-1, p-2, \ldots$ as bins, we can apply Lemma 5 to deduce that the probability of any such i existing is at most $2^{-\Omega(k)}$. ∎

The proof of Proposition 7 comes with a corollary that will be useful to reference later.

Corollary 8. *Consider an ordered-linear-probing hash table with n slots that contains $(1 - 1/x)n$ elements and no tombstones. Consider a position i, and let T be the number of elements u that reside in positions i or greater, but that have hashes $h(u) < i$. Then*

$$\Pr[T \geq kx] \leq 2^{-\Omega(k)}.$$

3.3 Bounds on Insertion Surplus

In this section, we introduce two core technical propositions that will be used in subsequent sections for our analysis of ordered linear probing.

Consider a sequence S of operations which alternate between insertions and deletions. (Think of S as the operations between two rebuilds.) Let n be the number of slots in the hash table. Let P be a sub-interval of $[n]$ and let S_P denote the subset $\{u \in S \mid h(u) \in P\}$.

We say that a subset $S' \subseteq S_P$ is ***downward-closed*** (with respect to S_P) if it satisfies the following property: for every insertion or deletion $u \in S'$, every $v \in S_P$ that occurs temporally before u and satisfies $h(v) \geq h(u)$ is also in S'. Define the ***insertion surplus*** of a downward-closed set S' to be the number of insertions in S' minus the number of deletions in S', if there are more insertions than deletions in S', and 0 otherwise.

The purpose of this section is to prove upper and lower bounds on the maximum insertion surplus of any downward-closed subset $S' \subseteq S_P$. We will parameterize these bounds by $\mu := \mathbb{E}[|S_P|]/2$, which is the expected number of insertions (and also the expected number of deletions) in S_P.

Notice that S_P itself is downward-closed. If we assume that all of the operations in S_P are on distinct keys, then the expected insertion surplus of S_P will be $\Omega(\sqrt{\mu})$ (since the number of insertions and the number of deletions in S_P both have standard deviation $\Theta(\sqrt{\mu})$). A natural question is whether there exists any downward-closed subset $S' \subseteq S_P$ with a significantly larger insertion surplus.

This section proves two propositions.

Proposition 9. *Suppose that $|P| \geq \sqrt{\mu}$. With probability $1 - 1/\operatorname{poly}(\mu)$, every downward-closed subset $S' \subseteq S_P$ has insertion surplus $\tilde{O}(\sqrt{\mu})$.*

Proposition 10. *Suppose that the insertions/deletions in S_P are all on different keys. Suppose that $|S| \leq n$, that n is sufficiently large as a function of μ and $|P|$, and that $|P| \geq \sqrt{\mu}$. Then with probability $1 - o(1)$, there exists a downward-closed subset $S' \subseteq S_P$ with insertion surplus $\Omega(\sqrt{\mu \log \log \mu})$.*

Whereas Proposition 9 tells us that no downward-closed subset S' has significantly larger insertion surplus then the expected insertion surplus of S_P, Proposition 10 tells us that there most likely is some downward-closed subset S' whose insertion surplus is (at least slightly) asymptotically larger than the expected insertion surplus of S_P. Note that the $o(1)$ term in the probability bound given by Proposition 10 is in terms of μ.

Reinterpreting insertion surplus in terms of paths on a grid. Before proving the propositions, we first reinterpret the propositions in terms of paths on a grid. Consider a grid G with height $|S|$ and width $|P|$. Define r so that $r + 1$ is the beginning of interval P. Color the cell (i, j) in the grid G blue if the i-th operation in S is an insertion whose hash is $r + j$; and color the cell (i, j) red if the i-th operation in S is a deletion whose hash is $r + j$. (Many cells will be neither red nor blue.) The blue and red cells in the grid G correspond to the insertions and deletions in S_P.

Now consider the set of ***monotone paths through*** G, that is, the set of paths that begin in the bottom left corner, and then walk along grid lines to the top right corner, only ever moving up or to the right. Define the ***blue-red differential*** of a path to be the number of blue cells that reside below the path minus the number of red cells that reside below the path (we say that the path ***covers*** these cells).

A subset $S' \subseteq S$ is downward-closed if and only if there is a monotone path γ through G such that the blue and red cells covered by γ in G are precisely the insertions and deletions in S'. Thus, rather than considering downward-closed subsets S' of S_P, we can consider monotone paths γ through G, and rather than considering the insertion surplus of each subset S', we can consider the blue-red differential of each monotone path γ. Although this distinction may at first seem superficial, we shall see later that the geometry of monotone paths makes them amenable to clean combinatorial analysis.

Considering a more coarse-grained grid G'. One aspect of G that makes it potentially unwieldy is that it will likely be sparse, meaning that the vast majority of cells are neither red nor blue. Thus, in our proofs, it will also be useful to define a more coarse-grained grid G' that is laid on top of G. The grid G' has width and height $\sqrt{\mu}$, meaning that each cell in G' corresponds to a sub-grid of G with width $|P|/\sqrt{\mu}$ and height $|S|/\sqrt{\mu}$. To avoid confusion, we will refer to the blue/red cells in G as blue/red dots in G'.

Note that, since G' is a $\sqrt{\mu} \times \sqrt{\mu}$ grid, we are implicitly assuming that $\sqrt{\mu}$ is a positive integer; this assumption is w.l.o.g.. To simplify our discussion (so that we can treat all of the cells of G' as having uniform widths and heights), we will further assume that $|P|$ and $|S|$ are divisible by $\sqrt{\mu}$. These assumptions can easily be removed by rounding all of the quantities to powers of four, and performing the analysis using the rounded quantities.[7]

The grid G' is parameterized so that the expected number of blue dots (resp. red dots) in each cell is exactly 1. As terminology, we say that for each cell in G', the insertions and deletions that ***pertain*** that cell are the ones that have blue/red dots in that cell; and the keys that ***pertain*** to the cell are the keys that have at least one insertion/deletion pertaining to the cell. The expected number of distinct keys that pertain to a given cell in G' is at most 1. Moreover, by a Chernoff bound, and with probability $1 - 1/\operatorname{poly}(\mu)$, each cell has at most $O(\log \mu)$ keys that pertain to it.

3.3.1 Proof of Proposition 9

We wish to show that, with probability $1-1/\operatorname{poly}(\mu)$, every monotone path γ through G has blue-red differential at most $\tilde{O}(\sqrt{\mu})$. The next lemma establishes that, rather than considering monotone paths in G, it suffices to instead consider monotone paths in G'.

Lemma 11. *With probability $1-1/\operatorname{poly}(\mu)$, the following holds. For every monotone path α with blue-red differential a in G, there is a monotone path β with blue-red differential b in G' such that $a - b = O(\sqrt{\mu}\log \mu)$.*

Proof. Define β to be the same as α, except that the path is rounded to the grid lines of G' as follows: for any cell in G' that the path α goes through the interior of, we round the path so that β does not cover any of the points from that cell.

Every cell in G' that is entirely covered by α is also entirely covered by β; similarly, every cell in G' that is entirely not covered by α is also entirely not covered by β. Thus the only difference between α and β is that there are $O(\sqrt{\mu})$ cells in G' that are partially covered by α but that are not covered by β.

For each cell in G', define the ***risk potential*** of that cell to be the maximum blue-red differential of any monotone path in G from the bottom left corner of that cell to the top right corner of that cell. To complete the proof, it suffices to show that every cell in G' has risk potential at most $O(\log \mu)$. Notice, however, that the risk potential of each cell is at most as large as the number of distinct keys that pertain to the cell. Thus the risk potentials of the cells are all $O(\log \mu)$ with probability $1 - 1/\operatorname{poly}(\mu)$. ∎

To complete the proof of Proposition 9, it suffices to show that, with probability $1 - 1/\operatorname{poly}(\mu)$, every monotone path through G' has blue-red differential at most $\tilde{O}(\sqrt{\mu})$.

[7]Importantly, both propositions assume $|P| \geq \sqrt{\mu}$, so if we round both $|P|$ and μ to powers of 4, then the rounded value for $|P|$ will be a multiple of the rounded value for μ (and μ will be a square number).

The rest of the proof is completed in two pieces. The first piece is to show that for every monotone path γ, it is possible to decompose the area under the path into rectangles where the sum of the perimeters of the rectangles is $O(\mu \log \mu)$. The second piece is to show that, for each rectangle in G', the blue-red differential of that rectangle is at most the perimeter of the rectangle times $O(\log \mu)$. Combining these two facts, we get a bound on the blue-red differential of any monotone path γ.

Lemma 12. *Consider any monotone path γ through an $a \times b$ grid, where $\ell = a + b$ is a power of two. The area under the path can be decomposed into disjoint rectangles such that the sum of the perimeters of the rectangles is $O(\ell \log \ell)$.*

Proof. We construct the rectangular decomposition through the following recursive process. Break the path γ into two pieces of length $\ell/2$ which we call γ_1 and γ_2, and let q be the point at which the two pieces meet. Let R be the rectangle whose top left corner is q, and whose bottom right corner is the bottom right point of the grid. Then we define our rectangular decomposition to consist of the rectangle R, along with a recursively constructed rectangular decomposition for the path γ_1, and a recursively constructed rectangular decomposition for the path γ_2. The two recursive decompositions take place in the grids containing γ_1 and γ_2, and the base case of the recursion is when we get to path that is either entirely vertical or entirely horizontal (meaning that there is no area to decompose into rectangles).

By design, the i-th level of recursion takes place on a grid with perimeter $O(\ell/2^i)$. It follows that the rectangles added in the i-th level of recursion each have perimeters $O(\ell/2^i)$. On the other hand, the i-th level of recursion has at most 2^i recursive subproblems, so the total perimeter of the rectangles added in those subproblems is at most $O(\ell)$. There are at most $O(\log \ell)$ levels of recursion, so the sum of the perimeters of all of the rectangles in the decomposition is at most $O(\ell \log \ell)$. ■

Lemma 13. *Consider any rectangle X in grid G'. If X has perimeter k, then with probability $1 - 1/\operatorname{poly}(\mu)$, the number of blue dots minus the number of red dots in X is $O(k \log \mu)$.*

Proof. Let T be the time window that X covers on its vertical axis. The total number of insertions that occur in T is the same as the number of deletions that occur in T, up to ± 1. Call an insertion in T ***serious*** if the key being inserted has not previously been deleted in the same time window, and call a deletion in T ***serious*** if the key being deleted is not subsequently reinserted in the same time window. Notice that the number of blue dots from non-serious insertions in X is the same as the number of red dots from non-serious deletions in X, so we can ignore both. Since the number of non-serious insertions equals the number of non-serious deletions, the number of serious insertions in T is the same (up to ± 1) as the number of serious deletions in T.

Since X has perimeter k, it can contain at most k^2 cells in G'. Thus the expected number of distinct keys that pertain to the cells in X is $O(k^2)$. It follows that the expected number of serious insertions (and similarly, the expected number of serious deletions) of keys that pertain to X is $O(k^2)$. In order for the number of blue dots minus the number of red dots in X to exceed $D = \Omega(k \log \mu)$, we would need that either (a) the number of serious insertions pertaining to X is at least $D/2$ greater than its mean; or (b) the number of serious deletions pertaining to X is at least $D/2$ smaller than its mean. By a Chernoff bound, the probability of either (a) or (b) occurring is that most $1/\operatorname{poly}(\mu)$.[8] ∎

Proof of Proposition 9. By Lemma 11, it suffices to show with probability $1 - 1/\operatorname{poly}(\mu)$ that every monotone path γ through G' has blue-red differential $\tilde{O}(\sqrt{\mu})$. By Lemma 12, every such path has a rectangular decomposition where the sum of the perimeters of the rectangles is $\tilde{O}(\sqrt{\mu})$. By Lemma 13, with probability $1 - 1/\operatorname{poly}(\mu)$, the contribution of each of the rectangles to the blue-red differential of γ is that most $O(\log \mu)$ times the perimeter of the rectangle. Thus, with probability $1 - 1/\operatorname{poly}(\mu)$, every monotone path γ through G' has blue-red differential $O(\sqrt{\mu} \log^2 \mu)$. ∎

3.3.2 Proof of Proposition 10

We wish to construct a monotone path through G' that has blue-red deviation $\Omega(\sqrt{\mu \log \log \mu})$.

Let $a_{i,j}$ be the number of insertions that pertain to cell (i, j) and let $b_{i,j}$ the number of deletions that pertain to cell (i, j) in G'. Each $a_{i,j}$ and each $b_{i,j}$ is a binomial random variable with mean 1. However, the random variables are not completely independent (they are slightly negatively correlated), which makes them a bit unwieldy to work with. To handle this, the following lemma Poissonizes the random variables in order to show that they are $o(1)$-close to being independent.

Lemma 14. *Let A be the random variable $\langle a_{i,j}, b_{i,j} \mid i, j \in [\sqrt{\mu}]\rangle$ and let $\overline{A}$ be a random variable $\langle \overline{a}_{i,j}, \overline{b}_{i,j} \mid i, j \in [\sqrt{\mu}]\rangle$, where each $\overline{a}_{i,j}$ and each $\overline{b}_{i,j}$ is an independent Poisson random variable with mean 1. Then it is possible to couple the random variables A and $\overline{A}$ such that they are equal with probability $1 - o(1)$.*

Proof. Recall that, by assumption in Proposition 23, all of the operations in S are on different keys. We can think of each insertion/deletion as being performed on a random hash in $[n]$ (rather than being performed on any specific key). As part of our construction of $\overline{A}$, we will end up modifying S (i.e., adding and removing some operations) to get a new operation sequence $\overline{S}$. When adding new operations to S, we will need not bother associating the new operations with actual keys, and will instead think of the new operations is simply each being associated with a random hash.

We now describe our construction of $\overline{A}$. Let $T_1, T_2, \ldots, T_{\sqrt{\mu}}$ be the time windows that correspond to the rows of G'. Let $y_1, y_2, \ldots, y_{\sqrt{\mu}}, z_1, z_2, \ldots, z_{\sqrt{\mu}}$ be independent Poisson random variables with mean $|S|/(2\sqrt{\mu})$. Define $\overline{S}$ to be S, except that the operations in each time window T_i are modified so that the number of insertions is y_i and the number of deletions is z_i; note that this may require us to either add or

[8]Note that all of the serious insertions (resp. serious deletions) are on distinct keys, meaning that their hashes are independent, hence the Chernoff bound.

remove operations to the time window. Then define $\overline{A}$ in exactly the same way as A, except using $\overline{S}$ in place of S. That is, we let $\overline{a}_{i,j}$ be the number of insertions in $\overline{S}$ that pertain to cell (i, j) in G' and we let $\overline{b}_{i,j}$ the number of deletions in $\overline{S}$ that pertain to cell (i, j) in G'.

To understand the distribution of $\overline{A}$, we make use of an important fact about Poisson random variables: if J balls are placed at random into K bins, and J is a Poisson random variable with mean ϕ, then for each bin the number of balls in the bin is an independent Poisson random variable with mean ϕ/K (see Chapter 5.4 of [269]). This implies that the $\overline{a}_{i,j}$'s and $\overline{b}_{i,j}$'s are independent Poisson random variables each of which has mean 1.

To complete the proof, we must establish that $\Pr[A \neq \overline{A}] = o(1)$. For each time window T_i, the expected number of operations that are in one of S or $\overline{S}$ but not the other is

$$O(\sqrt{|T_i|}) = O\left(\sqrt{|S|/\sqrt{\mu}}\right) = O\left(\sqrt{n/\sqrt{\mu}}\right) = O(\sqrt{n}).$$

In total over all $\sqrt{\mu}$ time windows T_i, the expected number of operations that are in one of S or $\overline{S}$ but not the other is therefore $O(\sqrt{n\mu})$. The probability that any of these operations are on keys that hash to P is

$$O\left(\frac{|P|}{n} \cdot \sqrt{n\mu}\right) = O(|P|\sqrt{\mu/n}),$$

which by the assumptions of Proposition 23 is $o(1)$. ∎

Throughout the rest of the proof, we will treat the $a_{i,j}$s and $b_{i,j}$s as independent Poisson random variables each of which has mean 1. Our proof will make use of the fact that for any Poisson random variable J with mean $\phi \geq 1$,

$$\Pr[J > \phi + \sqrt{\phi}t] \geq \exp(-\Omega(t^2)). \tag{3.6}$$

For a derivation of (3.6), see Theorem 1.2 of [307]. Set $D = \sqrt{\log\log\mu}/c$, where c is a sufficiently large constant. We will be making use of (3.6) in the case where $t = 2D$, that is,

$$\begin{aligned}\Pr[J > \phi + 2\sqrt{\phi}D] &\geq \exp(-\Omega((\sqrt{\log\log\mu}/c)^2)) \\ &= \exp(-\Omega(\log\log\mu/c^2)) \\ &\geq 1/\sqrt{\log\mu}. \end{aligned} \tag{3.7}$$

We now construct a monotone path γ through the grid G', and show that γ has blue-red deviation $\Omega(\sqrt{\mu\log\log\mu})$ with probability $1 - o(1)$. The construction of γ is recursive, with different levels of recursion operating on squares grids of different sizes. To avoid ambiguity, we will use k to refer to the height (or width) of the grid in the current recursive sub-problem, and we will use $\sqrt{\mu}$ to refer to the height (or width) of the grid in the top-level sub-problem.

The construction of γ in a $k \times k$ subproblem is performed as follows. Break the $k \times k$ grid into four $k/2 \times k/2$ quadrants. If the top left quadrant has blue-red deviation at least kD (that is, it contains at least kD more blue dots than red dots) then we say that the subproblem ***successfully terminates***, and we set γ to be the path that consists of k vertical steps followed by k horizontal steps. Otherwise, we construct γ by concatenating together a path recursively constructed through the bottom left quadrant and a path recursively constructed through the top right quadrant. If a recursive subproblem is on a 1×1 grid, then we return the path consisting of a vertical step followed by a horizontal step, and we call the subproblem a ***failed leaf***.

Lemma 15. *Each subproblem with $k > 1$ has probability at least $\Omega(1/\sqrt{\log \mu})$ of successfully terminating.*

Proof. The number of blue dots and the number of red dots in the top left quadrant of the subproblem are both Poisson random variables with means $k^2/4$. It follows by (3.7) that the number of blue dots has probability at least $1/\sqrt{\log \mu}$ of exceeding its mean by kD. Since the number of red dots has probability at least $\Omega(1)$ of being less than or equal to its mean, it follows that the blue-red deviation of the quadrant is at least kD with probability at least $\Omega(1/\sqrt{\log \mu})$. ∎

Lemma 16. *With probability $1 - o(1)$, the number of failed leaves is less than $\sqrt{\mu}/2$.*

Proof. We will prove that the expected number of failed leaves is $o(\sqrt{\mu})$, after which the lemma follows by Markov's inequality.

There are $\sqrt{\mu}$ potential failed leaves in the recursion tree, so it suffices to show that each of them has $o(1)$ probability of occurring. In order for a given failed leaf to occur, the recursion path of depth of $\log \sqrt{\mu}$ that must occur. By Lemma 15, each of the subproblems in the recursion path (except for the leaf) independently has at least a $\Omega(1/\sqrt{\log \mu})$ probability of successfully terminating. Thus each potential failed leaf in the recursion tree has probability at most

$$\left(1 - \Omega(1/\sqrt{\log \mu})\right)^{\log \sqrt{\mu} - 1} \leq o(1)$$

of occurring. ∎

We can now analyze the blue-red deviation of γ to prove Proposition 10.

Proof of Proposition 10. By Lemma 16, we may assume that the number of failed leaves is less than $\sqrt{\mu}/2$.

Say that the ***width*** of a subproblem is width of the grid in which it takes place. The sum of the widths of the leaf subproblems is $\sqrt{\mu}$. Each failed leaf has width 1, so if the number of failed leaves is less than $\sqrt{\mu}/2$, then the sum of the widths of the leaves that successfully terminate must be at least $\sqrt{\mu}/2$.

For each leaf subproblem with width k that successfully terminates, its top left quadrant contributes $\Omega(k\sqrt{\log \log \mu})$ to the blue-red deviation of γ. Summing over the leaf subproblems that successfully terminate, the top left quadrants of all of them contribute at least $\Omega(\sqrt{\mu \log \log \mu})$ to the blue-red deviation.

Additionally, we must consider the effect of the blue and red dots below γ that are not contained in any of these aforementioned top-left quadrants. Once the path γ is determined, the number X of such blue dots and the number Y of such red dots are independent Poisson random variables satisfying $\mathbb{E}[X] = \mathbb{E}[Y] \leq \mu$. By a Chernoff bound, we have that with probability $1 - o(1)$,

$$Y - X \leq \sqrt{\mu \log \log \log \mu}.$$

Thus, with probability $1 - o(1)$, the blue-red deviation of γ is at least

$$\Omega(\sqrt{\mu \log \log \mu}) - \sqrt{\mu \log \log \log \mu} = \Omega(\sqrt{\mu \log \log \mu}).$$

∎

3.4 Relating Insertion Surplus to Crossing Numbers

In this section we use our insertion-surplus bounds from Section 3.3 to obtain bounds on a different set of quantities $\{c_j\}_{j \in [n]}$ that we call the crossing numbers; later, in Section 3.5, our bounds on crossing numbers will allow for us to analyze the amortized costs of insertions/deletions/queries in ordered linear probing.

Consider an ordered linear probing hash table with n slots, and suppose that the hash table is initialized to have load factor $1 - 1/x$ or smaller. Consider a sequence of insertion and deletion operations S such that the load factor never exceeds $1 - 1/x$. (Note that, unlike in Section 3.3, the lemmas in this section will not all require that S alternates between insertions and deletions.)

Based on the initial state of the hash table and on the sequence S of operations, define the ***crossing numbers*** $c_1, c_2, \ldots, c_n$ so that c_i is the number of times that an insertion with a hash smaller than i either (a) uses a tombstone left by a key that had hash at least i; or (b) uses a free slot in a position greater than or equal to i.

The purpose of this section is to prove nearly tight bounds on $\mathbb{E}[c_i]$. Subsequent sections will then show how to use these bounds in order to analyze the performance of linear probing.

We will need the following additional definitions. Define the ***insertion surplus*** of a subinterval $P \subseteq [n]$ to be the maximum insertion surplus of any downward-closed subset of $\{u \in S \mid h(u) \in P\}$, minus the number of free slots that are initially in the range P. Define the ***peak*** p_u of an insertion u to be the hash of the tombstone that the insertion uses (if it uses a tombstone) or the position of the free slot that the insertion uses (if it uses a free slot).

The following lemmas characterize the crossing numbers in terms of the insertion surpluses of intervals.

Lemma 17. *For each $s \in [n]$, there exists an interval $P = [r, s-1]$ whose insertion* *surplus is at least c_s.*

Proof. Call an insertion u with hash smaller than s ***special*** if it satisfies the following recursive property: either (a) $p_u \geq s$; or (b) there is another special insertion v that occurs temporally after u such that $p_u \geq h(v)$. Call a deletion v ***special*** if $h(v) < s$ and there exists a special insertion u that occurs temporally after v and satisfies $h(u) \leq h(v)$.

Let r be the smallest hash of any special insertion/deletion, and define $P = [r, s-1]$. We will prove that the insertion surplus of P is at least c_s. Towards this end, define A to be the number of special insertions, define B to be the number of special deletions, and define C to be the number of free slots initially in P. The set of special operations is downward-closed by design, and its insertion surplus is $A - B - C$. Thus we wish to show that

$$A - B - C \geq c_s.$$

In order for an insertion u to contribute to the crossing number c_s, the insertion must have peak $p_u \geq s$ and thus must also be special. To complete the proof, we will show that there are at least $B + C$ special insertions with peaks $p_u < s$ (and thus at most $A - B - C$ special insertions have $p_u \geq s$). That is, we will show that every tombstone created by a special deletion and every free slot initially in P is used by some special insertion.

Consider a tombstone that is created by a special deletion v. Since v is special, there must exist a special insertion u that occurs after v and satisfies $h(u) \leq h(v)$. Let u be the last such insertion. We must have that $p_u > h(v)$, since otherwise, we could arrive at a contradiction as follows: in order so that u could be special, there would have to be a subsequent special insertion z with $h(z) \leq p_u$; but this would imply that $h(z) \leq h(v)$, which would contradict the fact that u is the final special insertion satisfying $h(u) \leq h(v)$. Since $p_u > h(v)$, it must be that the tombstone created by the deletion v has already been used by the time that insertion u is performed. The insertion w that used the tombstone must have occurred before the insertion u and must have had peak $p_w = h(v) \geq h(u)$. This means that w is itself a special insertion. Thus the tombstone created by v is used by a special insertion, as desired.

Now consider a free slot j that is initially present in P. By the definition of P, there must exist a special insertion u that satisfies $h(u) \leq j$. Let u be the last such insertion. We must have that $p_u > j$, since otherwise, we could arrive at a contradiction as follows: in order so that u could be special, there would have to be a subsequent special insertion z with $h(z) \leq p_u$; but this would imply that $h(z) \leq j$, which would contradict the fact that u is the final special insertion satisfying $h(u) \leq j$. Since $p_u > j$, it must be that the free slot j has already been used by the time that insertion u is performed. The insertion w that used slot j must have occurred before the insertion u and must have had peak $p_w = j \geq h(u)$. This means that w is itself a special insertion. Thus the free slot is used by a special insertion, as desired. ∎

The converse of the previous lemma is also true.

Lemma 18. *If there exists an interval $P = [r, s-1]$ with insertion surplus q, then $c_s \geq q$.*

Proof. Let S' be the downward-closed subset of $\{u \in S \mid h(u) \in P\}$ with the largest insertion surplus. Every insertion in S' must either (a) use a tombstone created by a deletion in S', (b) use a free slot initially present in P, or (c) have peak at least s. It follows that if A is the number of insertions in S', B is the number of deletions in S', and C is the number of free slots initially in P, then we must have that

$$c_s \geq A - B - C.$$

Since the quantity on the right side is exactly the insertion surplus of P, the proof is complete. ∎

The previous lemmas tells us that, in order to understand the crossing numbers c_s, it suffices to understand the insertion surplus of each interval P. This insertion surplus, in turn, depends on two quantities: the maximum insertion surplus of any downward-closed subset of $\{u \in S \mid h(u) \in P\}$; and the number of free slots *initially* in P. We have already achieved a good understanding of the first quantity in the previous section. The next two lemmas analyze the second quantity.

Lemma 19. *Suppose that the hash table initially has load factor $1 - 1/x$. Consider any interval $P \subseteq [n]$ of size at least $c\,x^2 \log x$, where c is taken to be a sufficiently large constant. With probability $1 - 1/\operatorname{poly}|P|$, the interval P initially contains at least $\Omega(|P|/x)$ free slots.*

Proof. The expected number of elements that hash into P initially is $(1 - 1/x)|P|$, which since $|P| \geq c\,x^2 \log x$, is at most $|P| - \sqrt{|P| c \log |P|}$. It follows by a Chernoff bound that, with probability $1 - 1/\operatorname{poly}(|P|)$, the number of elements that initially hash into P is at most $|P| - \frac{1}{2}\sqrt{|P| c \log |P|}$. On the other hand, by Corollary 8, and with probability $1 - 1/\operatorname{poly}(|P|)$, the number of elements that reside in P but hash to a position prior to P is at most $O(x \log P) \leq \frac{1}{4}\sqrt{|P| c \log |P|}$. The total number of elements that reside in P is therefore at most $|P| - \frac{1}{4}\sqrt{|P| c \log |P|}$, which completes the proof. ∎

Lemma 20. *Suppose that the hash table initially has load factor $1 - 1/x$. Consider any interval $P = [a, b] \subseteq [n]$ of size x^2/c, where c is taken to be a sufficiently large constant. With probability $\Omega(1)$, there are initially no free slots in P.*

Proof. Consider the state of the hash table initially, and let r be the length of the run of non-free slots beginning at position a. Knuth in [224] established that $\mathbb{E}[r] = \Theta(x^2)$. On the other hand, Proposition 6 tells us that $\Pr[r > kx^2] \leq 1/\operatorname{poly}(k)$ for all k. The only way that these two facts can be consistent is if $r = \Omega(x^2)$ with probability $\Omega(1)$. Thus the lemma is established. ∎

We are now in a position to upper bound the crossing number c_j.

Proposition 21. *Suppose that the hash table initially has load factor $1 - 1/x$, suppose that $|S| = \Omega(n/x)$ and $|S| \le n$, and suppose that S alternates between insertions and deletions.*

There exists a positive constant d such that for any $j \in [n]$ and any

$$r \ge \sqrt{\frac{|S|}{n} x^2 \log^d x},$$

we have $c_j < r$ with probability $1 - 1/\operatorname{poly}(r)$.

Proof. Define

$$\mu_i = |S| \cdot i/n$$

to be the expected number of operations in S that hash into the $[j-i, j-1]$. Define

$$\lambda_i = \sqrt{\mu_i} \operatorname{polylog} \mu_i$$

where the polylogarithmic factor is selected so that Proposition 9 offers the following guarantee: with probability $1 - 1/\operatorname{poly}(\mu_i)$, every downward-closed subset of $\{u \in S \mid h(u) \in [j-i, j-1]\}$ has insertion surplus less than λ_i. Let

$$K = x^2 \log^c x$$

for some sufficiently large positive constant c. This results in

$$\begin{aligned}\lambda_K &= \sqrt{\frac{|S|}{n} x^2 \log^c x} \operatorname{polylog}\left(\frac{|S|}{n} x^2 \log^c x\right) \\ &= \sqrt{\frac{|S|}{n} x^2} \operatorname{polylog} x.\end{aligned}$$

Thus, if we select the constant d in the proposition statement appropriately, then the requirement that $r \ge \sqrt{\frac{|S|}{n} x^2 \log^d x}$ implies that $r \ge \lambda_K$, and thus that $r = \lambda_R$ for some $R \ge K$. To prove the proposition, it suffices to establish that for every $R \ge K$, we have

$$\Pr[c_j \ge \lambda_R] \le 1/\operatorname{poly}(\lambda_R). \tag{3.8}$$

Note that in the parameter regime $R \ge K$, we have that $\operatorname{poly}(\lambda_R) = \operatorname{poly}(\mu_R) = \operatorname{poly}(R)$ (here we are using that $\Omega(n/x) \le |S| \le O(n)$), so we will treat the three as interchangeable throughout the rest of the proof.

Define $P_i = [j-i, j-1]$ and define $s(P_i)$ to be the insertion surplus of P_i. By Lemma 17,

$$\begin{aligned}\Pr[c_j \ge \lambda_R] &\le \Pr[s(P_i) \ge \lambda_R \text{ for some } i] \\ &\le \Pr[s(P_i) \ge \lambda_R \text{ for some } i < R] + \sum_{i \ge R} \Pr[s(P_i) > 0].\end{aligned}$$

To prove (3.8), we begin by bounding $\Pr[s(P_i) \geq \lambda_R$ for some $i < R]$. If $s(P_i) \geq \lambda_R$ for some $i < R$, then there must be a downward-closed subset S' of $\{u \in S \mid h(u) \in [j - (R-1), j-1]\}$ such that the insertion surplus of S' is at least λ_R. But by Proposition 9 and by the definition of λ_{R-1}, we know that with probability $1 - 1/\operatorname{poly}(\mu_{R-1})$ (and thus also in λ_{R-1}), every such S' has insertion surplus at most $\lambda_{R-1} < \lambda_R$. Thus the probability that $s(P_i) \geq \lambda_R$ for any $i < R$ is at most $1/\operatorname{poly}(\lambda_R)$.

To complete the proof, it remains to show that

$$\sum_{i \geq R} \Pr[s(P_i) > 0] \leq \frac{1}{\operatorname{poly}(\lambda_R)}.$$

We will establish a stronger statement, namely that for every $i \geq K$,

$$\Pr[s(P_i) > 0] \leq \frac{1}{\operatorname{poly}(i)}.$$

Since $i > K$, we can apply Lemma 19 to deduce that, with probability $1 - 1/\operatorname{poly}(i)$, the interval P_i initially contains at least $\Omega(i/x)$ free slots. We further have that, by Proposition 9, and with probability $1 - 1/\operatorname{poly}(\mu_i) = 1 - 1/\operatorname{poly}(i)$, every downward-closed subset of $\{u \in S \mid h(u) \in [j-i, j-1]\}$ has insertion surplus less than λ_i. It follows that $s(P_i)$ is that most

$$\max(0, \lambda_i - \Omega(i/x)).$$

In order to establish that $s(P_i)$ is zero, it suffices to show that

$$\lambda_i = o(i/x).$$

This is simply a matter of calculation:

$$\begin{aligned}
\lambda_i &= \sqrt{\mu_i}\operatorname{polylog}\mu_i && \text{(by definition of } \lambda_i\text{)}\\
&= \sqrt{\frac{|S|i}{n}}\operatorname{polylog}\left(\frac{|S|i}{n}\right) && \text{(by definition of } \mu_i\text{)}\\
&\le \sqrt{i}\operatorname{polylog} i && \text{(since } |S| \le n\text{)}\\
&\le O\left(\frac{i\operatorname{polylog} i}{\sqrt{K\lg^c(i/K)}}\right) && \text{(since } i > K\text{)}\\
&= O\left(\frac{i\operatorname{polylog} i}{\sqrt{x^2\log^c x\lg^c(i/K)}}\right) && \text{(since } K = x^2\lg^c x\text{)}\\
&= O\left(\frac{i\operatorname{polylog} i}{\sqrt{x^2\log^c K\lg^c(i/K)}}\right) && \text{(since } K = x^2\lg^c x\text{)}\\
&= O\left(\frac{i\operatorname{polylog} i}{\sqrt{x^2\log^c i}}\right) && \text{(since } \lg a\lg b \ge \Omega(\lg(ab))\text{)}\\
&= \frac{i/x\operatorname{polylog} i}{\sqrt{\log^c i}}\\
&= o(i/x) && \text{(since } c \text{ is a sufficiently large constant).}
\end{aligned}$$

∎

Corollary 22. *Suppose that the hash table initially has load factor $1-1/x$, suppose that $|S| = \Omega(n/x)$ and $|S| \le n$, and suppose that S alternates between insertions and deletions.*

For each $j \in [n]$,

$$\mathbb{E}[c_j] \le \sqrt{\frac{|S|}{n}x^2}\operatorname{polylog} x.$$

We can also obtain a nearly matching lower bound for $\mathbb{E}[c_j]$.

Proposition 23. *Suppose that the hash table initially has load factor $1-1/x$, suppose that $|S| = \Omega(n/x)$ and $|S| \le n$, and suppose that S alternates between insertions and deletions.*

Further suppose that each operation in S applies to a different key. Then for each $j \in [n]$,

$$\mathbb{E}[c_j] = \Omega\left(\sqrt{\frac{|S|}{n}x^2\log\log x}\right).$$

Proof. By Lemma 18, it suffices to show that there is some interval $P_i = [j-i, j-1]$

with insertion surplus

$$\Omega\left(\sqrt{\frac{|S|}{n}x^2 \log\log x}\right).$$

By Lemma 20, there exists a positive constant c such that, with probability $\Omega(1)$, the interval P_{cx^2} in the hash table initially (i.e., at the beginning of the operations S) contains no free slots. Furthermore, Proposition 10 tells us that with probability $1-o(1)$, there exists a downward-closed subset of $\{u \in S \mid h(u) \in P_{cx^2}\}$ with insertion surplus at least

$$\begin{aligned}&\Omega\left(\sqrt{\frac{|S|}{n}x^2 \log\log\left(\frac{|S|}{n}x^2\right)}\right)\\ &= \Omega\left(\sqrt{\frac{|S|}{n}x^2 \log\log x}\right).\end{aligned}$$

With probability $\Omega(1)$, both of the preceding events occur simultaneously. Thus the proposition is proven. ∎

The previous two propositions both focus on the case in which the workload S alternates between insertions and deletions. We conclude this section by considering the case where S is allowed to perform an arbitrary sequence of insertions and deletions, subject only to the constraint that the load factor never exceeds $1 - 1/x$.

Proposition 24. *Suppose that the hash table begins at a load factor of at most $1 - 1/x$, and that the sequence of operations S keeps the load factor at or below $1 - 1/x$. Finally, suppose that $|S| \le n/\operatorname{polylog}(x)$. Then for each $j \in [n]$,*

$$\mathbb{E}[c_j] = O(x).$$

Proof. We begin by constructing an alternative sequence of insertions/deletions $\overline{S}$ such that the crossing numbers with respect to $\overline{S}$ are guaranteed to be at least as large as the crossing numbers with respect to S. We will then complete the proof by analyzing the crossing numbers of $\overline{S}$.

Suppose that the hash table initially contains $(1 - 1/x)n - w$ elements for some w. We construct $\overline{S}$ through two steps:

- Call an insertion ***novel*** if the key being inserted has not been inserted in the past and was not originally present in the hash table. The first step is to take each of the first w novel insertions, and to move them to the front of the operation sequence.[9]

[9]We can assume without loss of generality that there are at least w such novel insertions, since if there are not, we can artificially add additional insertions to the end of S and then perform the rest of the proof without modification.

- Call a triple of three consecutive operations ***unbalanced*** if the triple consists of two deletions followed by an insertion. At least one of the two deletions in such a triple must act on a different key than the insertion acts on. We can ***balance*** the triple by changing the order of operations so that the aforementioned deletion occurs last. The second step in constructing $\overline{S}$ is to repeatedly find and balance any unbalanced triples until there are no such triples left.

Observe that the sequence $\overline{S}$ is a valid sequence of operations, since the transformation from S to $\overline{S}$ never swaps the order of any two operations that act on the same key.

We claim that the crossing numbers with respect to $\overline{S}$ are at least as large as the crossing numbers with respect to S. The transformation from S to $\overline{S}$ moves certain insertions to occur earlier than they would have otherwise, and certain deletions to occur later than they would have otherwise. Importantly, these types of moves cannot decrease the insertion surplus of any interval P in the hash table. By Lemmas 17 and 18, the crossing numbers are completely determined by the insertion surpluses of the intervals $P \subseteq [n]$. Since the insertion surpluses for $\overline{S}$ are at least as large as those for S, it follows that the crossing numbers for $\overline{S}$ are also at least as large as those for S.

Our next claim is that $\overline{S}$ never causes the load factor to exceed $1-1/x$. This can be seen by analyzing each of the two steps of the construction separately. The first step modifies only the window of time in which the first w novel insertions are performed; no rearrangement of the operations in this window of time can possibly cause the load factor to exceed $1 - 1/x$. The second step repeatedly performs balancing operations on unbalanced triples; such a balancing operation does not change the maximum load factor that is achieved during the triple, however, since that load factor is achieved prior to the first operation of the triple. Combining the analyses of the two steps, we see that the load factor never exceeds $1 - 1/x$.

Now let us reason about the structure of $\overline{S}$. By design, $\overline{S}$ begins with w insertions, bringing the load factor up to exactly $1 - 1/x$. Since the load factor never exceeds $1 - 1/x$, and since there are never two deletions in a row followed by an insertion, it must be that the remaining insertions in $\overline{S}$ are each preceded by exactly one deletion. In other words, $\overline{S}$ must consist of three parts $\overline{S}_1, \overline{S}_2, \overline{S}_3$ where $\overline{S}_1$ consists only of insertions, $\overline{S}_2$ alternates between deletions and insertions, and $\overline{S}_3$ consists only of deletions.

Since $\overline{S}_3$ consists only of deletions, it does not contribute anything to the crossing numbers. By Corollary 22, the expected contribution of the operations in $\overline{S}_2$ to each crossing number c_j is at most $O(x)$.

It remains to bound the contribution of $\overline{S}_1$ to the crossing numbers. If an insertion takes time t, then it can contribute at most t to the sum $\sum_j c_j$. Knuth showed in [224] that the total time needed to fill an empty table up to a load factor of $1-1/x$ is $O(nx)$ in expectation. Thus the expected contribution of $\overline{S}_1$ to $\sum_j c_j$ is $O(nx)$, completing the proof. ∎

3.5 Relating Crossing Numbers to Running Times

In this section, we give nearly tight bounds on the performance of ordered linear probing. Notably, we find that, if the size R of each rebuild window is chosen correctly, then the amortized time per insertion is guaranteed to be $\tilde{O}(x)$. The key technical component to the section will be a series of arguments transforming our bounds on crossing numbers (in Section 3.4) into bounds on running times.

Consider an ordered linear probing hash table that uses tombstones for deletions. Recall that there are three parameters: the number n of slots in the table, the number R of insertions in each time window between rebuilds, and the maximum load factor $1 - 1/x$ that the hash table ever achieves. Based on these parameters, we wish to analyze the average running time of the operations being performed on the hash table.

We will be focusing exclusively on the regime in which $R = \Omega(n/x)$ and $R \leq n$. Since each rebuild can be implemented in linear time $O(n)$, the average time spent performing rebuilds per operation is $O(n/R) = O(x)$ (which for our purposes will be negligible). Thus the focus of our analysis will be on analyzing the costs of the operations that occur between consecutive rebuilds.

Before diving into the details, we remark that there are two main technical challenges that our analysis must overcome. The first challenge is obvious: we must quantify the degree to which tombstones left behind by deletions improve the performance of subsequent insertions. The second challenge is a bit more subtle: in order to support large rebuild-window sizes R, our analysis must be robust to the fact that tombstones can accumulate over time, increasing the effective load factor of the hash table. This latter challenge is exacerbated by the fact that the choice of which tombstones are in the table at any given moment is a function not only of the sequence of operations being performed, but also of the randomness in the hash table. This means that, even if the cumulative load factor from the elements and tombstones can be bounded (e.g., by $1 - 1/(2x)$), we still cannot apply the classic analysis at that load factor in order to bound the expected time of queries.

One of the interesting features of our analysis is that we completely circumvent the issue of how fast tombstones accumulate over time. Rather than focusing on what the effective load factor of the hash table is at each moment in time, the analysis instead analyzes the state of the hash table at the beginning of the rebuild window, and then analyzes the dynamics of how the local structure of the hash table changes over time.

In the following lemmas, we will focus on a single window W of time between two rebuilds. We begin by defining three quantities that we will be able to express the running times of operations in terms of.

For a given position $i \in [n]$, define the ***positional offset*** o_i to be the quantity $j - i$ where $j \geq i$ is the largest position such that, at the end of the time window W, all of the positions $[i, j-1]$ contain elements and tombstones whose hashes are smaller than i. Note that, although the positional offset is defined at the end of the time window W, if we were to define the same quantity at any other point during the time window, it would be at most o_i (that is, the positional offset only increases over time).

For a given position $i \in [n]$, define the ***spillover*** s_i to be the largest $k \geq 0$ such that if we consider all keys that are either initially present or inserted at some point during W, at least $4k$ of them have hashes in the range $[i, i+k)$.

For each insertion u, define the ***displacement*** d_u of the insertion to be $p_u - h(u)$, where p_u is the peak of the insertion as defined in Section 3.4.

Lemma 25. *If an insertion u hashes to a position i, then the insertion takes time at most*

$$O(o_i + s_i + d_u + 1).$$

Similarly, if a query/deletion u hashes to a position i, then the operation takes time at most

$$O(o_i + s_i + 1).$$

Proof. Consider an insertion u that hashes to a position i. If u uses a free slot in some position $i + r$, then the time to perform the insertion is $r = d_u + 1$. Suppose, on the other hand, that u uses a tombstone with some hash $i + t$, and the tombstone is in some position $i + r$. Then the running time of the insertion is r, and we wish to show that

$$r = O(o_i + s_i + t + 1). \tag{3.9}$$

If $r - o_i = O(t)$, then (3.9) trivially holds and we are done. Otherwise, we may assume that $r - o_i \geq 4(t+1)$. By the definition of the positional offset o_i, all of the elements/tombstones in positions $[i + o_i, i + r]$ must have hashes in the range $[i, i+t]$. Combining this with the fact that $r - o_i \geq 4(t+1)$, it follows that the spillover s_i satisfies $s_i \geq \lfloor (r - o_i)/4 \rfloor$, hence (3.9).

Next consider a query/deletion u that hashes to a position i. The operation takes time at most $O(o_i + T)$ where T is the total number of elements with hash i that are either present at the beginning of W or inserted at some point during W. By the definition of s_i, we have that $T \leq 4s_i + O(1)$. Thus the operation takes time $O(o_i + s_i + 1)$.

Our next lemma relates o_i, s_i, and d_u to the crossing numbers c_i defined in the previous section.

Lemma 26. *For each $i \in [n]$, the positional offset o_i satisfies $o_i \geq c_i$ and*

$$\mathbb{E}[o_i] = O(x) + \mathbb{E}[c_i], \tag{3.10}$$

and the spillover s_i satisfies

$$\mathbb{E}[s_i] = O(1). \tag{3.11}$$

Finally, if we consider a random insertion u in the time window W, then

$$\mathbb{E}[d_u] = \frac{n}{R}\mathbb{E}[c_i]. \tag{3.12}$$

Proof. Although the positional offset o_i is only defined at the end of the time window W, let us slightly abuse notation and think about how the quantity evolves over time (that is, what would happen if we defined the quantity at each point in time in the time window). By Corollary 8, the initial positional offset (at the beginning of W) has expected value $O(x)$. During the time window, the positional offset increases by one each time that an insertion whose hash is less than i has a peak that is at least i. The number of such insertions is precisely c_i. Since these insertions are the only operations that can change the positional offset, it follows that $o_i \geq c_i$ and $\mathbb{E}[o_i] = O(x) + \mathbb{E}[c_i]$.

Next we consider the spillover s_i. Let V be the set of all elements that are present at some point during W. Then $|V| \leq 2n$, and the expected number of elements in V that hash to a given position j is at most 2. It follows by Lemma 3 that $\Pr[s_i > k] \leq \exp(-\Omega(k))$ for all k. This implies (3.11).

Finally we establish (3.12). Observe that, if an insertion u has displacement d_u, then the insertion contributes to exactly d_u crossing numbers c_i. It follows that

$$\sum_u d_u = \sum_{i \in [n]} c_i.$$

If we select a random insertion u out of the R insertions that occur in W, then

$$R \cdot \mathbb{E}[d_u] = \sum_{i \in [n]} c_i.$$

Since the c_i's all of the same expected values, it follows that for a given i,

$$R \cdot \mathbb{E}[d_u] = n \cdot \mathbb{E}[c_i].$$

This implies (3.12). ∎

We are now prepared to prove the main results of the section. We begin by considering a hovering workload, that is a workload in which queries can be performed at arbitrary times, but insertions and deletions must alternate.

Theorem 1. *Consider an ordered linear probing hash table that uses tombstones for deletions, and that performs rebuilds every R insertions. Suppose that the table is initialized to have capacity n and load factor $1 - 1/x$, where $R = \Omega(n/x)$ and $R \leq n$. Finally, consider a sequence S of operations that alternates between insertions and deletions (and contains arbitrarily many queries).*

Then the expected amortized time I spent per insertion satisfies

$$I \leq \tilde{O}\left(x\sqrt{\frac{n}{R}}\right) \tag{3.1}$$

and, if all insertions/deletions in each rebuild window are on distinct keys, then

$$I \geq \Omega\left(x\sqrt{\frac{n}{R}\log\log x}\right). \tag{3.2}$$

Moreover, the expected time Q of a given query/deletion satisfies

$$Q \leq O(x) + \tilde{O}\left(x\sqrt{\frac{R}{n}}\right) \tag{3.3}$$

and, if all operations in each rebuild window are on distinct keys, then for any negative query at the end of a rebuild window, we have

$$Q \geq \Omega\left(x + x\sqrt{\frac{R}{n}\log\log x}\right). \tag{3.4}$$

Proof. Consider a random insertion u with some hash i. By Lemma 25, we have that u takes time at most $O(o_i + s_i + d_u + 1)$. By Lemma 26, this has expectation at most

$$O\left(x + \mathbb{E}[c_i] + \frac{n}{R}\mathbb{E}[c_i] + 1\right) = O\left(x + \frac{n}{R}\mathbb{E}[c_i]\right).$$

By Corollary 22, this is that most[10]

$$\tilde{O}\left(x + \frac{n}{R}\sqrt{\frac{R}{n}x^2}\right) = \tilde{O}\left(x\sqrt{\frac{n}{R}}\right).$$

On the other hand, the insertion time is necessarily at least d_u, which by Lemma 26, has expectation

$$\frac{n}{R-1}\mathbb{E}[c_j]$$

for each $j \in [n]$. If we assume that every insertion/deletion in the rebuild window is on a different key, then we can further apply Proposition 23 to conclude that the insertion time has expected value at least

$$\Omega\left(\frac{n}{R}\sqrt{\frac{R}{n}x^2\log\log x}\right) = \Omega\left(x\sqrt{\frac{n}{R}\log\log x}\right).$$

Now instead consider a query/deletion u that hashes to some position i. By Lemma 25, we have that u takes time at most $O(o_i + s_i + 1)$. By Lemma 26, the

[10]There is one technicality that we must be slightly careful about here: the hash $i = h(u)$ is independent of where every key $v \neq u$ hashes to, but is (trivially) not independent of where key u hashes to. However, since u is only one key, it can easily be factored out of the analysis so that we can treat i as being a random slot (independent of the hash function h).

expected time that u takes is therefore at most

$$O\left(x + \mathbb{E}[c_i] + 1\right).$$

By Corollary 22, this is that most

$$\tilde{O}\left(x + \sqrt{\frac{R}{n}x^2}\right) = \tilde{O}\left(x + x\sqrt{\frac{R}{n}}\right).$$

If we assume that the query is a negative query, performed at the end of a rebuild window whose insert/delete operations are all on different keys, then the query time is necessarily at least o_i, which by Lemma 26 is at least c_i. It follows by Proposition 23 that the expected query time is at least

$$\Omega\left(x\sqrt{\frac{R}{n}\log\log x}\right).$$

The expected query time is also $\Omega(x)$ trivially, by the standard analysis of linear probing [224].

Theorem 1 has several important corollaries. Our first corollary considers the setting in which rebuilds are performed every $\Theta(n/x)$ insertions.

Corollary 27. *Suppose $R = \Theta(n/x)$. Then*

$$I \le \tilde{O}(x^{1.5}),$$

and, if all insertions/deletions in each rebuild window are on distinct keys, then

$$I \ge \Omega(x^{1.5}\sqrt{\log\log x}).$$

Moreover, $Q = \Theta(x)$.

Our next corollary considers the setting in which rebuilds are performed every $n/\operatorname{polylog}(x)$ insertions. In this case, the hash table achieves nearly optimal scaling.

Corollary 28. *If $R = n/\log^c x$ for a sufficiently large positive constant c, then*

$$I = \tilde{\Theta}(x).$$

Moreover, $Q = \Theta(x)$.

Our final corollary considers the question of whether it is possible to select a value of R that allows for both I and Q to be $O(x)$. The corollary establishes that no such R exists.

Corollary 29. *For every choice of R, there exists S such that either $I = \omega(x)$ or $Q = \omega(x)$.*

Proof. If $n/R = \omega(1)$, then (3.2) tells us that there exists a sequence of operations for which I is $\omega(x)$. On the other hand, if $n/R = o(\lg\lg x)$, then (3.4) tells us that there exists a sequence of operations for which Q is $\omega(x)$. ∎

Up until now, we have been focusing on a hovering workload. Our final result considers an arbitrary workload of operations, where the only constraint is that the load factor never exceeds $1 - 1/x$. Notice that if R is small (i.e., $R = \Theta(n/x)$), then ordered linear probing can potentially perform very poorly in the setting, since an entire rebuild window can potentially consist of only insertions, none of which are able to make use of tombstones, but all of which are performed at a load factor of $1-\Theta(1/x)$. Our next theorem establishes, however, that if R is selected appropriately, then the amortized performance of ordered linear probing is near the optimal $O(x)$ that one could hope to achieve.

Theorem 2. *Let c be a sufficiently large positive constant. Consider an ordered linear probing hash table that uses tombstones for deletions, and that performs rebuilds every $R = n/\log^c x$ insertions. Finally, consider a sequence of operations S that never brings the load factor above $1 - 1/x$.*

Then the expected amortized cost of each insertion is $\tilde{O}(x)$ and the expected cost of each query/deletion is $O(x)$.

Proof. This follows from the same proof as Theorem 1, except using Proposition 24 instead of Corollary 22. ∎

Remark 30. *The proofs of Theorems 1 and 2 assume a fully random hash function, but it turns out this assumption is not needed. In particular, one can instead use tabulation hashing, and modify the proofs in the preceding sections as follows: analogues of Lemmas 3 and 5 follow directly from Theorem 8 of [305], and then all of the Chernoff bounds throughout our proofs can be re-created using Theorem 1 of [305]. Note that each application of Theorem 8 and Theorem 1 of [305] introduces a $1/\operatorname{poly}(n)$ failure probability, but this is easily absorbed into the analysis.*

Chapter 4

Graveyard Hashing, an Ideal Linear-Probing Hash Table

In this chapter, we describe and analyze a new variant of linear probing, which we call ***graveyard hashing***. Graveyard hashing takes advantage of the key insight from the previous chapter, which is that tombstones have the ability to significantly improve insertion performance.

Description of graveyard hashing. Graveyard hashing uses different rebuild window sizes, depending on the load factor. If a rebuild is performed at a load factor of $1 - 1/x$, then the next rebuild will be performed $n/(4x)$ operations later.[1]

Whenever the hash table is rebuilt, Graveyard hashing first removes all of the tombstones that are currently present. It then spreads *new tombstones* uniformly across the table. If the current load factor is $1 - 1/x$, then $n/(2x)$ tombstones are created, with one tombstone assigned to each of the hashes $\{2ix \mid i \in [n/(2x)]\}$. The purpose of these tombstones is to help all of the up to $n/(4x)$ insertions that occur between the current rebuild and the next rebuild.

The insertion of tombstones during rebuilds is the *only difference* between graveyard hashing and standard ordered linear probing. Thus insertions, queries, and deletions are implemented exactly as in a traditional ordered linear probing hash table.

If desired, one can implement graveyard hashing so that each rebuild also dynamically resizes the table, ensuring that the load factor is always $1 - \Theta(1/x)$ for some fixed parameter x. Note that, when resizing the table, the elements of the table will need to be assigned to new hashes, and thus will need to be permuted. In the RAM model, this can easily be done in linear time (and in place) using an in-place radix sort. In the external-memory model, resizing can be implemented in $O(n/B)$ block transfers (where B is the external memory block size) using Larson's block-transfer efficient scheme for performing partial expansions/contractions on a

[1]Note that graveyard hashing counts both insertions and deletions as part of the length of a rebuild window.

hash table [233] (this technique has also been used in past work on external-memory hashing, see [216, 302]).

Analysis of graveyard hashing. To perform the analysis, we will need one last balls-and-bins lemma:

Lemma 31. *Suppose that μn balls are placed into n bins at random. Let $x > 1$, $k \geq 1$, and $j \in [n]$. With probability $1 - 2^{-\Omega(k)}$, for every interval $I \subseteq [n]$ that contains j, the number of balls in the bins I is at most $(1 + 1/x)|I|\mu + xk$.*

Proof. Suppose there is some interval I satisfying $i \in I$ such that the number of balls in the interval I is greater than $(1+1/x)|I|\mu + xk$. If $I = [j_0, j_1]$ for some j_0, j_1, then we can break I into two sub-intervals $I_1 = [j_0, j]$ and $I_2 = (j, j_1]$. Since I contains at least $(1 + 1/x)|I|\mu + xk$ balls, at least one of the two subintervals $I_q \in \{I_1, I_2\}$ must contain at least

$$(1 + 1/x)|I_q|\mu + xk/2$$

balls. However, by Lemma 5, the probability of any such subinterval I_q existing is at most $2^{-\Omega(k)}$. ∎

Corollary 32. *Suppose that μn balls are placed into n bins at random. Let $x > 1$, $k \geq 1$, and $j \in [n]$. With probability $1-2^{-\Omega(k)}$, for every interval $I \subseteq [n]$ that contains j and has size $|I| \geq x^2k$, the number of balls in the bins I is at most $(1 + 1/x)|I|\mu$.*

Proof. This follows by applying Lemma 31 with $x' = x/2$ and $k' = kx/2$. ∎

We now turn our attention to analyzing graveyard hashing. As in the previous sections, it will be easier to analyze the displacement of an insertion rather than directly analyzing the running time of each insertion. Recall that the displacement d_u of an insertion u is $i - h(u)$ where i is the hash of the tombstone that u uses, if u uses a tombstone, and i the position of the free slot that u uses, if u uses a free slot.

The next lemma bounds the difference between displacement and running time. The fact that the rebuild windows for graveyard hashing are so small (only $n/(4x)$ operations) ends up allowing for an especially simple argument.

Lemma 33. *Consider the insertion of an element u. Let d denote the displacement of the insertion, and t denote the running time. Then, for any $r \geq 1$,*

$$\Pr[t - d - 1 \geq rx] \leq \exp(-\Omega(r)).$$

Proof. We can assume without loss of generality that u makes use of some tombstone v (rather than a free slot), since otherwise the lemma is trivial. The displacement of u is therefore given by $d = h(v) - h(u)$ and the running time of u is given by $t = k - h(u) + 1$, where k is the position in which v resides. Thus

$$\Pr[t - d - 1 \geq rx] = \Pr[k - h(v) \geq rx],$$

which means that we want to show that

$$\Pr[k - h(v) \geq rx] \leq \exp(-\Omega(r)).$$

For each element/tombstone v' in the run containing position $h(u)$, define the ***placement-error*** $e_{v'}$ to be $k' - h(v')$, where k' is the position in which v' resides (at the moment of time prior to the insertion u).

We wish to show that $e_v < rx$, but we must be careful about the fact that v is partially a function of the randomness of the hash table. In order to bound e_v, we assume that v is selected adversarially, and instead bound the quantity

$$\beta = \max\{e_{v'} \mid v' \text{ is in the same run as } u, \text{ and } h(v') \geq h(u)\}.$$

We cannot afford to simply union bound over the different options for v' here; instead we must make use of the fact that the values of $e_{v'}$ are closely correlated for different keys v' in the same run.

Let v' be the element for which $\beta = e_{v'}$. Let p be the position of the left-most element in u's run. All of the elements/tombstones that reside in positions $p, \ldots, h(v') + e_{v'}$ must have hashes in $[p, h(v')]$. The number of elements/tombstones (at the time prior to u's insertion) that hash to the interval $I = [p, h(v')]$ must therefore be at least $|I| + e_{v'}$. In contrast, the expected number of elements/tombstones that hash into the interval I is that most $|I|$. Thus there are at least $e_{v'}$ more element/tombstones that hash into I then are expected. By Lemma 31, it follows that $\Pr[e_{v'} \geq rx] \leq \exp(-\Omega(r))$. ∎

Graveyard hashing is designed so that there are always copious tombstones for insertions to make use of. Note, in particular, that each rebuild window begins with $n/(2x)$ tombstones but only contains at most $n/(4x)$ insertions. This allows for the following bound on displacement.

Lemma 34. *Consider an insertion u. The displacement d of u satisfies*

$$\Pr[d \geq rx] \leq \exp(-\Omega(r))$$

for all $r > 1$.

Proof. Call a tombstone ***primitive*** if it was inserted during the rebuild prior to the current rebuild window. Let T be the set of primitive tombstones present when u is inserted. Let

$$j_0 = \max\{h(v) \mid v \in T, h(v) < h(u)\}$$

and

$$j_1 = \min\{h(v) \mid v \in T, h(v) \geq h(u)\}.$$

The displacement d is at most $j_1 - h(u) \leq j_1 - j_0$. To complete the proof, we will bound the probability that $j_1 - j_0 \geq rx$.

At the beginning of the rebuild window, there were $\frac{j_1 - j_0}{2x} - 1$ primitive tombstones with hashes in the range $I = (j_0, j_1)$; denote the set of these tombstones by L. By the time u is inserted, all of the tombstones L have been used by insertions (this is by the definition of j_0 and j_1). Since there is still a primitive tombstone with hash j_0, the insertions that used up L must have all had hashes at least $j_0 + 1$. Thus, during the current rebuild window, there have been at least $|L| = \frac{j_1 - j_0}{2x} - 1$ insertions that hashed into the interval $I = (j_0, j_1)$.

Recall, however, that each rebuild window consists of only $n/(4x)$ operations. The expected number of insertions that hash into I is therefore a most $\frac{j_1 - j_0}{4x}$.

In summary, the only way have $j_1 - j_0 \geq rx$ is if (1) there is an interval I containing $h(u)$ that satisfies $|I| \geq rx - 2$, and (2) the number q of insertions (during the current rebuild window) that hash into I is a constant factor larger than the expected number $\mathbb{E}[q]$ of such insertions. To bound the probability of such an I existing, we partition the slots of the hash table into "bins" of size x, and treat keys inserted during the rebuild window as balls that each hash to a bin. In order for I to exist, there must be a contiguous subsequence of $\Theta(r)$ bins such that the interval of hashes covered by the bins contains $h(u)$, and such that the bins contain a constant factor more balls than expected. By Corollary 32, the probability of such a subsequence of bins existing is that most $\exp(-\Omega(r))$. ∎

Combining the previous lemmas, we can analyze the running time of graveyard hashing.

Theorem 35. *Consider a graveyard hash table. For each insertion/query/deletion, if the operation is performed at some load factor of $1 - 1/x$ then the operation takes expected time $O(x)$, and incurs amortized time $O(x)$ for rebuilds.*

Proof. Since graveyard hashing uses small rebuild windows (i.e., of size $n/(4x)$), we can analyze queries by ignoring the deletions in the rebuild window, and applying the classic $O(x)$ bound for query time in an insertion-only table (Proposition 7). Deletions of keys u take the same amount of time that a query for that key would have, so deletions also take expected time $O(x)$. To analyze insertions, we can apply Lemmas 33 and 34, which together bound the expected time by $O(x)$.

Finally, we analyze the amortized cost of rebuilds per operation. If a rebuild window starts at a load factor of $1 - 1/x$, then the next rebuild is performed after $\Theta(n/x)$ operations, and all of those operations are performed at load factors $1 - \Theta(1/x)$. The rebuild can be performed in time $\Theta(n)$ and thus the amortized cost per operation is $\Theta(x)$. ∎

Remark 36. *The proof of Theorem 35 assumes a fully random hash function, but this assumption is not necessary. The theorem continues to hold if we use either tabulation hashing or 5-independent hashing. In particular, one can use Equation (23) from [305] as a substitute for Lemma 31, and then re-create all of the proofs above without modification.*

We conclude the section by analyzing graveyard hashing in the external-memory model [42].

Theorem 37. *Consider the external memory model with $B = rx$ for some $r \geq 1$ and $M = \Omega(B)$. Graveyard hashing can be implemented to offer the following guarantee on any sequence of operations. The load factor of the table is maintained to be $1 - 1/\Theta(x)$ at all times, and each operation incurs*

$$1 + O(1/r)$$

block transfers in expectation. Furthermore, the amortized block-transfer cost (per operation) of rebuilds is $O(1/r)$.

Proof. By Theorem 35, the expected time taken by a given operation is $O(x)$. It follows that the expected number of block boundaries that are crossed by the operation is $O(x/B) = O(1/r)$. Thus the expected number of block transfers incurred is $O(1/r)$.

Next we analyze the cost of rebuilds. Each rebuild window contains $\Theta(n/x)$ operations at a load factor of $1 - \Theta(1/x)$, and, as discussed earlier, the rebuild at the end of the window can be implemented with $O(n/B)$ block transfers. This implies the desired bound of $O(1/r)$ on amortized rebuild cost. ∎

Corollary 38. *If $x = o(B)$, then the amortized expected number of block transfers per operation is $1 + o(1)$.*

Part II

Dynamic Sorting Revisited: The Power of History Independence

Chapter 5

Introduction

The online ***dynamic sorting problem***, which in the theoretical literature is more widely referred to as the ***list-labeling problem***, is one of the most basic and well-studied algorithmic primitives in data structures, with an extensive literature spanning upper bounds [47, 58, 73, 79, 88–90, 193, 213, 214, 219, 364–366], lower bounds [120, 150–152, 319, 372], variants [46, 47, 70, 90, 148, 149, 193, 314], and open-problem surveys [192, 319]. The problem has been independently re-discovered in many different contexts [46, 193, 314, 363], and it has found extensive applications to areas such as ordered maintenance [72, 73, 88, 149], cache-oblivious data structures [79, 80, 84, 89, 114], dense file maintenance [363–366], applied graph algorithms [238, 240, 303, 355–357], etc. (For a detailed discussion of related work and applications, see Section 5.2.)

The list-labeling problem was originally formulated [214] as follows. An algorithm must store a set of n elements (where n changes over time) in sorted order in an array of $m \geq n$ slots. Elements are inserted and deleted over time, with each insertion specifying the new element's ***rank*** $r \in \{1, 2, \ldots, n+1\}$ among the other elements that are present (e.g., inserting at rank 1 means that the inserted element is the new smallest element). To keep the elements in sorted order in the array, the algorithm must sometimes move elements around. The ***cost*** of an algorithm is the number of elements moved during the insertions/deletions.[1]

The list-labeling problem is well understood in the regime where $m \gg n$. In the ***pseudo-exponential regime***, when $\frac{m}{n} = 2^{n^{\Omega(1)}}$, it is possible to achieve $O(1)$ amortized cost per operation [58]. In the ***polynomial regime***, when $\frac{m}{n} = n^{\Theta(1)}$, the amortized cost becomes $O(\log n)$ [46, 193, 229]. These bounds are known to be tight for both deterministic and randomized algorithms [57, 58, 121].

It has remained an open problem, however, what happens in the ***linear regime***, where $m = (1 + \varepsilon)n$ for some $\varepsilon = \Theta(1)$. In 1981, Itai, Konheim, and Rodeh [214] showed how to achieve amortized cost $O(\log^2 n)$, and posed as an open question

[1]To accommodate the many ways in which list labeling is used, some works describe the problem in a more abstract (but equivalent) way: the list-labeling algorithm must dynamically assign each element x a label $\ell(x) \in \{1, 2, \ldots, m\}$ such that $x \prec y \iff \ell(x) < \ell(y)$, and the goal is to minimize the number of elements that are *relabeled* per insertion/deletion—hence the name of the problem.

whether any algorithm could do better. Despite a great deal of subsequent work on alternative solutions (including deterministic, randomized, and deamortized algorithms) for the same problem [70, 73, 88, 90, 213, 219, 364–366], the bound of $O(\log^2 n)$ has remained unimproved for four decades.

Starting in 1990, there has been a long line of work towards establishing a matching $\Omega(\log^2 n)$ lower bound [120,121,150–152]. It is known that any deterministic algorithm requires $\Omega(\log^2 n)$ amortized cost per insertion [120]. And the same lower bound holds for ***smooth*** algorithms, where the relabelings are restricted to evenly rebalance elements across a contiguous subarray [152]. This second lower bound is surprisingly strong: it applies even to randomized algorithms and even to the offline problem, where the entire sequence of operations is known *a priori*. However, the best general lower bound remains $\Omega(\log n)$ [121].

These lower bounds tell us that, if an algorithm is to beat the $O(\log^2 n)$ bound, then the algorithm must be both randomized and non-smooth. Whether or not any such algorithm is possible has remained the central open question [120, 121, 150–152, 214] in this research area (see also discussion of the problem in open-problem surveys and textbooks [192, 259, 319]). Several sources [150–152] have conjectured that $\Theta(\log^2 n)$ cost is optimal in general.

Chapter 6. Breaking the $\log^2 n$ Barrier

We present a randomized list-labeling algorithm that achieves expected cost $O(\log^{3/2} n)$ per insertion/deletion in the linear regime (Corollary 53). In breaking through the $\log^2 n$ barrier, we establish that there is a fundamental gap between deterministic and randomized algorithms for online list labeling. Our result is the first asymptotic improvement in the linear regime in the 40-year history of the problem.

The original $O(\log^2 n)$ upper bound by Itai et al. [214] also extends to the ***dense regime*** of $\varepsilon = o(1)$, where the bound on amortized cost becomes $O(\varepsilon^{-1} \log^2 n)$ [47, 103, 372]. Extending our algorithm to the same regime, we achieve expected cost $O(\varepsilon^{-1} \log^{3/2} n)$ (Theorem 40).

Applying our result to the insertion-only setting, the array can be filled from empty to full (i.e., $n = m$) in total expected time $O(n \log^{2.5} n)$ (Corollary 54). This improves over the previous state of the art of $O(n \log^3 n)$, which was known to be optimal for deterministic algorithms [120]—again we have a separation between what can be achieved with deterministic and randomized algorithms.

A surprising aspect of our results is how they contrast with the polynomial regime $m = n^{1+\Theta(1)}$, where randomized and deterministic algorithms are asymptotically equivalent [57, 58, 121]. Our final upper-bound result considers a continuum between these regimes, where $m = \omega(n) \cap n^{o(1)}$. In this ***sparse regime*** there is a folklore bound [46, 193, 229] of $O\left(\frac{\log^2 n}{\log(m/n)}\right)$, which continuously deforms between $O(\log^2 n)$ for the linear regime and $O(\log n)$ for the polynomial regime. Using our techniques

(Theorem 57), we achieve expected cost

$$O\left(\frac{\log^{3/2} n}{\sqrt{\log(m/n)}}\right).$$

Thus we achieve asymptotic improvements for list labeling for all $m = n^{1+o(1)}$.

An unexpected tool: history independence. One research area that our algorithms build directly upon is the study of history-independent data structures: a data structure is said to be ***history independent*** [263,279] if its current state reveals nothing about the history of the past operations beyond the current set of elements that are present.

History independence is typically viewed as a security guarantee, with the intent being to minimize the risk incurred by a security breach. Research on history-independent data structures [104,118,195,196,208,263,278,279] (as well as on history-independent list labeling [70] specifically) has focused on history independence as an *end goal*, with the question being whether history independence can be achieved without any increase in running time.

We find that, in the context of list labeling, history independence is actually a valuable *algorithmic tool* for building faster randomized data structures. History independence allows for us to have a data structure with vulnerabilities (i.e., certain spots where an insertion would be expensive) while (1) keeping those vulnerabilities hidden from the adversary; and (2) preventing the adversary from having any control over where those vulnerabilities appear. This simple paradigm plays an important role in allowing our randomized data structures to bypass the $\log^2 n$ barrier.

Chapter 7. A Matching Lower Bound for History-Independent Solutions

Finally, we show that our bounds in the dense regime are asymptotically optimal for any history-independent data structure: there exists a positive constant c such that, for all $1/n^{1/3} \le \varepsilon \le 1/c$, the expected insertion/deletion cost when $m = (1+\varepsilon)n$ is necessarily at least $\Omega(\varepsilon^{-1}\log^{3/2} n)$ for any history-independent data structure (Theorem 60).

This means that, if there exists a randomized data structure that achieves better bounds, then the data structure must fundamentally be *adaptive* in how it responds to the history of the operations being performed. Of course, by being adaptive, such a data structure would also implicitly surrender the structural anonymity that history independence offers, revealing information about where the "hotspots" are within the data structure. We conjecture that, in general, our bounds are optimal—proving or disproving this conjecture remains an enticing open problem.

The rest of the introduction is split into two parts. First, we present preliminaries and technical background in Section 5.1. Then, in Section 5.2, we give an expanded overview of past work.

5.1 Preliminaries and Technical Background

In this section, we (1) formally define the list-labeling problem, (2) define history independence, (3) outline the classical $O(\log^2 n)$ solution [214], and (4) outline a more recent history-independent variation on that solution [70].

The list-labeling problem. A list-labeling data structure stores a dynamically changing set of size $n \leq m$ in an array of m slots. It supports two operations:

- INSERT(r), $r \in \{1, 2, \ldots, n+1\}$: This operation adds an element whose rank is r. This increments n and also increments the ranks of each of the elements whose ranks were formerly in $\{r, \ldots, n\}$.
- DELETE(r), $r \in \{1, 2, \ldots, n\}$: This operation removes the element whose rank is r. This decrements n and also decrements the ranks of each of the elements whose ranks were formally in $\{r+1, \ldots, n\}$.

The list-labeling algorithm must maintain the invariant that the elements appear in sorted order (by rank) within the array. The ***cost*** of an insertion/deletion is the number of elements that are moved within the array during the insertion/deletion (including the element being inserted/deleted). In the case where $n = \Omega(m)$, we will further guarantee (for our upper bounds) that the maximum gap between any two consecutive elements in the array is at most $O(1)$ positions—this extra guarantee is often required for applications of list labeling in which algorithms perform range queries within the array, e.g., [79, 80, 303, 356, 357].

We will typically use an additional parameter ε such that either $n \leq (1-\varepsilon)m$ or $m \geq (1+\varepsilon)n$ (the specific convention that we follow will differ from section to section to optimize for simplifying the algebraic manipulation in each section).

From the perspective of the list-labeling data structure, the elements that it stores are black boxes—the only information that the data structure knows about its elements is their sorted order. This allows for list labeling to be used in applications where the elements are from arbitrary universes.

Finally, it is important to emphasize that the insertions/deletions are performed by an ***oblivious adversary***, who does not get to see the random decisions made by the list-labeling data structure. If the adversary were to be adaptive, then, trivially, no randomized list-labeling data structure could incur expected cost any better than the worst-case cost of the best deterministic list-labeling data structure.

History independence. A data structure is said to be ***history independent*** [70, 104, 118, 195, 196, 208, 263, 278, 279] if, given access to the current state of the data structure, the only information that an adversary can deduce is the current set of elements; that is, the adversary gains no information about the history of operations performed. In the list-labeling data structure the current set of elements is specified only by their relative ranks, so the only information that an adversary can deduce is the *number* of elements.

History independence plays an important supporting role in our results. Indeed, although history independence does not on its own improve the asymptotics of list labeling, it does create a natural abstraction for how to separate the behavior of the data structure that we are designing from the actions of the user.

There are several basic mathematical properties of history independence that will be useful in both our upper and lower bounds. Define the ***array configuration*** of a list-labeling data structure to be the boolean vector in $\{0,1\}^m$ indicating which n positions of the array contain elements. We have the following properties of a history-independent data structure for list labeling:

Property 39.

(a) *Whenever the array contains n elements, its array configuration A satisfies $A \sim \mathcal{C}_{n,m}$, where $\mathcal{C}_{n,m}$ is some probability distribution over array configurations.*
(b) *Whenever an insertion is performed at rank $r \in \{1, 2, \ldots, n+1\}$ in an array with n elements, the array configurations A_0 and A_1 before and after the insertion satisfy $(A_0, A_1) \sim \mathcal{I}_{n,m,r}$, where $\mathcal{I}_{n,m,r}$ is a joint distribution between $\mathcal{C}_{n,m}$ and $\mathcal{C}_{n+1,m}$.*[2]
(c) *Whenever a deletion is performed at rank $r \in \{1, 2, \ldots, n+1\}$ in an array with $n+1$ elements, the array configurations A_1 and A_0 before and after the deletion satisfy $(A_0, A_1) \sim \mathcal{D}_{n,m,r}$, where $\mathcal{D}_{n,m,r}$ is a joint distribution between $\mathcal{C}_{n,m}$ and $\mathcal{C}_{n+1,m}$.*

These properties imply that the (probability distribution on the) behavior of the algorithm on any given operation is fully determined by n, m, the operation (insertion or deletion), and the rank r of the element being inserted/deleted. In our upper bounds, we will further have that $\mathcal{D}_{n,m,r} = \mathcal{I}_{n,m,r}$; we call any list-labeling data structure with this property ***insertion/deletion symmetric***.

5.1.1 The Classical Solution and its History-Independent Analogue

List labeling with weight-balanced trees. The original solution to list labeling [214], due to Itai et al. [214] in 1981, can be described in terms of weight-balanced trees [193, 288, 289]. For brevity, we will describe the solution here for the linear regime, where $m = (1 + \Theta(1))n$, but the same solution directly generalizes to all regimes from dense ($n = (1 - \varepsilon)m$) to polynomial ($m = n^{1+\Theta(1)}$).

[2]A probability distribution $\mathcal{X}$ is a joint distribution between distributions $\mathcal{A}$ and $\mathcal{B}$ if $(A, B) \sim \mathcal{X} \implies A \sim \mathcal{A}$, $B \sim \mathcal{B}$.

Consider an array of size m, and impose a tree structure on it, where the root node represents the entire array, the nodes in the i-th level of the tree represent disjoint sub-arrays of size $m/2^{i-1}$, and the leaf nodes represent sub-arrays of size $\Theta(\log n)$. We keep the tree tightly weight balanced, meaning that, for any pair of sibling nodes x and y, their densities are always within a $1 \pm O(1/\log n)$ factor of each other. In particular, whenever an insertion or deletion breaks this invariant for some pair of siblings x and y, we take the elements in the sub-array $x \cup y$ and rearrange them to be distributed evenly across that sub-array.[3]

This tight weight balancing ensures that *all* of the nodes in the tree have densities that are within a factor of $(1 + O(1/\log n))^{O(\log n)} = O(1)$ of each other. By selecting the constants in the algorithm appropriately, one can ensure that every leaf has more slots than it has elements, which guarantees the correctness of the data structure. On the other hand, in order to maintain such tight weight balancing, one must rebuild nodes a factor of $O(\log n)$ more often than in a standard weight-balanced binary search tree [193, 288, 289], leading to an amortized cost of $O(\log^2 n)$.

Intuitively, the above data structure would seem to be the asymptotically optimal approach to maintaining tightly-balanced densities within an array—the known lower bounds for list labeling [120, 150–152] confirm that this is the case for both deterministic and smooth data structures. The results, however, reveal that it is not the case for randomized data structures. Randomization fundamentally reduces the cost to maintain a tightly weight-balanced tree.

History-independent list labeling. To understand how history independence can be achieved in the context of list labeling, it is helpful to first understand it in the context of balanced binary search trees. The classic example of a balanced binary tree with a history-independent topology is the ***randomized binary search tree*** [49,325] (or, similarly, the treap [49, 325]), which maintains as an invariant that, at any given moment, the structure of the tree is random (i.e., that within each subtree, the root of that subtree is a random element). This can be achieved with reservoir sampling [49, 70, 242, 325, 348]—in particular, whenever a new item is added to a subtree of (former) size r, the element becomes the new root with probability $1/(r+1)$ (in which case the subtree is rebuilt from scratch). This simple approach yields an expected time of $O(\log n)$ per operation.

As shown by Bender et al. [70], the same basic approach can be used to achieve history-independent list labeling. Now, the tree is random across all *tightly balanced trees*—that is, within each subtree T containing elements $x_1 < x_2 < \cdots < x_k$, the root is a random element x_i of those satisfying $|i - k/2| \in O(k/\log n)$. As before, this structure can be maintained using reservoir sampling. However, the restriction that the tree must be tightly balanced increases the frequency with which subtrees are rebuilt, so that the expected cost per operation becomes $O(\log^2 n)$, just as for the standard solution to list labeling.

[3]This approach is both deterministic and smooth, and thus consistent with the assumptions made by lower bounds [57, 150–152].

5.2 Related work

Formulations and reformulations. The list-labeling problem has been independently formulated several times and under various names. It was first studied by Itai, Konheim and Rodeh [214] as a sparse table scheme for implementing priority queues. Willard [363] considered the *file-maintenance problem*, where records are inserted and deleted in a sequentially ordered file. Dietz [149] formulated the similar *order-maintenance problem* of maintaining order in a linked list with efficient insertions. Andersson [46] and Andersson and Lai [47] studied a version of the problem in the context of balanced binary search trees, which Galperin and Rivest [193] independently studied under the name *scapegoat trees*. Raman [314] posited an analogous problem related to building locality preserving dictionaries.

This problem has mainly been studied in four regimes for the size m of the label array: dense ($m = (1 + o(1))n$), linear ($m = (1 + \Theta(1))n$), polynomial ($m = n^{1+\Theta(1)}$), and superpolynomial ($m = n^{\omega(1)}$).

Upper and lower bounds in the linear regime. In the linear regime, Itai, Konheim and Rodeh [214], first proved that items can be inserted with $O(\log^2 n)$ amortized cost. Various subsequent works have made improvements or simplifications to the algorithms achieving this cost, but the upper bound has remained unchanged. Willard [364–366] deamortized this result to a $O(\log^2 n)$ worst-case cost. Bender, Cole, Demaine, Farach-Colton and Zito [73], Bender, Fineman, Gilbert, Kopelowitz and Montes [88] and Katriel [219] provided simplified algorithms for this result for the order-maintenance problem. Itai and Katriel [213] additionally simplified the algorithm for the amortized upper bound.

The list-labeling problem where $m = (1 + \varepsilon)n$, and where the gap between any two inserted items is $O(1)$ is often called the *packed-memory array problem*, for which bounds of $O(\varepsilon^{-1} \log^2 n)$ are known [78, 79, 89]. Bender and Hu [90] provided an *adaptive* packed-memory array algorithm, that is, it matches the $O(\log^2 n)$ worst case insertion cost in the linear regime while achieving cost of $O(\log n)$ on certain common classes of instances. Bender, Berry, Johnson, Kroeger, McCauley, Phillips, Simon, Singh and Zage [70] presented a history-independent packed-memory array which again matches the existing upper bound in the linear regime.

Dietz and Zhang [152] proved a lower bound on insertion costs of $\Omega(\log^2 n)$ amortized per insertion in the linear regime for the natural class of *smooth* algorithms, where the relabelings are restricted to evenly rebalance elements across a contiguous subarray. Bulánek, Koucký and Saks. [120] showed a $\Omega(\log^2 n)$ lower bound for deterministic algorithms in the linear regime, and thus proved that the best known upper bounds were tight for deterministic algorithms. The best general lower bound is $\Omega(\log n)$ in the linear regime [121].

Other upper bounds. In the dense setting, Andersson and Lai [47], Zhang [372], and Bird and Sadnicki [103] showed an $O(n \log^3 n)$ upper bound for filling an array from empty to full for $m = n$. For arrays of polynomial size, it was known as a folklore algorithm that an amortized $O(\log n)$ insertion cost can be achieved by modifying the techniques in [214]. Kopelowitz [229] extended this to a worst case upper bound. This bound was also matched in the balanced search tree setting [46, 193]. In the superpolynomial array regime, Babka, Bulánek, Cunát, Koucký and Saks [58] showed an algorithm with amortized $O(\log n / \log \log m)$ cost when $m = \Omega(2^{\log^k n})$, which implies constant amortized cost in the pseudo-exponential regime of $m = 2^{n^{\Omega(1)}}$. Devanny, Fineman, Goodrich and Kopelowitz [148] studied the *online house numbering problem*, which is similar to the list-labeling problem, except with the objective to minimize the maximum number of times an element is relabeled.

Other lower bounds. Dietz and Zhang [152] proved a lower bound of $\Omega(\log n)$ per insertion in the polynomial regime for *smooth* algorithms. Bulánek, Koucký and Saks [120] showed an $\Omega(n \log^3 n)$ lower bound for n insertions into an initially empty array of size $m = n + n^{1-\varepsilon}$. Dietz, Seiferas and Zhang [151] proved a lower bound of $\Omega(\log n)$ in the polynomial regime for general deterministic algorithms, with a simplification by Babka, Bulánek, Cunát, Koucký, and Saks [57]. Bulanek, Koucký and Saks [121] also proved that the $\Omega(\log n)$ lower bound for the polynomial regime extends to randomized algorithms. In the superpolynomial regime, Babka, Bulánek, Cunát, Koucký and Saks [58] showed a lower bound of $\Omega\left(\frac{\log n}{\log \log m - \log \log n}\right)$ for m from n^{1+C} to 2^n, which reduces to a bound of $\Omega(\log n)$ for $m = n^{1+C}$.

Theoretical Applications. Applications of list labeling include the diverse motivating problems under which it was first studied, such as priority queue implementation, ordered file maintenance, etc. Hofri and Konheim [210] studied a similar array structure for use in a *control density array*, a sparse table that supports search, insert and deletion by keys. Fagerberg, Hammer and Meyer [174] used upper bounds from [214] for their rebalancing scheme, which maintains optimal height in a balanced B-tree.

Bender, Demaine and Farach-Colton [79] used the packed-memory array in their *cache-oblivious B-tree* algorithm, so our result directly implies an improvement in that scheme. Specifically, insertions into their B-tree take $O(\log_B N + (\log^2 N)/B)$ I/Os, and using our list-labeling algorithm, this is improved to $O(\log_B N + (\log^{3/2} N)/B)$ I/Os. Brodal, Fagerberg and Jacob [114] and Bender, Duan, Iacono and Wu [80] independently simplified the cache-oblivious B-tree algorithm. Bender, Fineman, Gilbert and Kuszmaul [89] presented *concurrent* cache-oblivious B-trees for the distributed setting. Bender, Farach-Colton and Kuszmaul [84] described *cache-oblivious string B-trees* for improved performance on variable length keys, compressed keys, and range queries. All of these cache-oblivious algorithms use packed-memory arrays.

In their results on the *controller problem* for managing global resource consumption in a distributed network, Emek and Korman [169] reduced the list-labeling problem to prove their lower bounds. Bender, Cole, Demaine, Farach-Colton and Zito [73] also applied list labeling lower bounds to the problem of maintaining a dynamic ordered set which supports traversals in the cache-oblivious and sequential-access models. Kopelowitz [229] studied the *predecessor search on dynamic subsets of an ordered dynamic list problem*, which combines the order-maintenance problem with the *predecessor problem* of maintaining dynamic sets which support predecessor queries. Nekrich used techniques for linear list labeling from [214] in data structures supporting various problems related to querying points in planar space, such as orthogonal range reporting [282, 283], the stabbing-max problem [285], and the related problem of searching a dynamic catalog on a tree [284]. Mortensen [272] similarly considered applications to the orthogonal range and dynamic line segment intersection reporting problems.

Practical Applications. Additionally, a variety of practical applications use the packed-memory array as an algorithmic component. Durand, Raffin and Faure [168] proposed using a packed-memory array to maintain sorted order during particle movement simulations for efficient searching. Khayyat, Lucia, Singh, Ouzzani, Papotti, Quiané-Ruiz, Tang and Kalnis [222] applied it to handle dynamic database updates in their inequality join algorithms. Toss, Pahins, Raffin and Comba [342] presented a *packed-memory quadtree*, which supports large streaming spatiotemporal datasets. De Leo and Boncz [239] presented the *rewired memory array*, an implementation of a packed-memory array which improves on its practical performance. Several works [238, 240, 303, 355–357] implemented *parallel* packed-memory arrays for the purpose of storing dynamic graphs with fast updates and range queries. Assessing whether our results can be used to obtain practical speedups for these applications remains an interesting direction for future work.

Related work on history independence. History independence has been studied for data structures in both internal and external memory models [70, 104, 118, 195, 196, 208, 263, 278, 279]. Even prior to the formalization of history independence [263, 279] in the late 1990s, there were several notable early works on hashing and search trees that implicitly achieved history-independent topologies [45, 48, 49, 310–312, 331, 338]. The notion of history independence studied in is sometimes referred to as *weak history independence*—although if we extend our problem statement in order to give items distinct ids, then our solutions become *strongly history independent* as well. For a survey of history independence, see recent work [200] by Goodrich, Kornaropoulos, Mitzenmacher and Tamassia.

History independence is typically treated as a security property: the goal is to minimize the amount of information that is leaked if an adversary sees the internals of the data structure. To the best of our knowledge, the results in Chapter 6 are the first to use techniques from history independence in order to achieve *faster* algorithms than were previously possible. We will see in subsequent parts of the thesis that this is not a fluke—indeed, the same design pattern will allow for us to make problem on several other seemingly unrelated and well-studied problems.

Chapter 6

Breaking the $O(\log^2 n)$ Barrier

In this chapter, we present a randomized list-labeling solution that, in the regime of $m = O(n)$, achieves $O(\log^{1.5} n)$ expected cost per insertion/deletion. We begin with Section 6.1, which gives an intuitive overview of our technical approach. A key technical idea will be to control the local density of the array via a random process that we call a Zeno random walk—we describe and analyze this random walk in Section 6.2. Section 6.3 then gives our (history-independent) list-labeling data structure and uses the bounds on Zeno random walks to analyze it. This leads to the following theorem:

Theorem 40. *Let $\varepsilon \in (0, 1)$, and suppose $m \geq (1 + \varepsilon)n$, where m is a static value while n changes dynamically. The shifted Zeno embedding on an array of size m with n elements incurs expected cost $O(\varepsilon^{-1} \log^{3/2} n)$ per insertion/deletion.*

Finally, Section 6.4 gives a black-box reduction for transforming dense list-labeling solutions into sparse list-labeling solutions. Thus our results in the dense/linear regime also imply new results in the sparse regime.

6.1 Technical Overview

In this section, we present an intuitive overview of our upper bound and proof techniques. Comprehensive technical details can be found in Sections 6.2 and 6.3. For simplicity, we shall assume in this section that $m = 2n$.

Intuitively, our starting point is the history-independent list labeling solution by Bender, et al. [70]. As described in Section 5.1, in [70], the root of any subtree of size k is a random element of the middle $O(k/\log n)$ elements of the subtree. We call this middle set of elements the *candidate set*.

A natural idea for decreasing the cost of this algorithm is to increase the size of the candidate set to δk for some $\delta = \omega(1/\log n)$. This way, the root would be resampled less often, resulting in fewer total rebalances. However, there is a problem with this approach: the subarrays representing the nodes in the i-th level of the tree have densities bounded between $\frac{1}{2}(1-\delta)^i$ and $\frac{1}{2}(1+\delta)^i$, but this means that nodes in the $\Theta(\log n)$-th level can overflow with a density of $\frac{1}{2}(1+\delta)^{\Theta(\log n)} = \omega(1)$. Thus, having $\delta = \omega(1/\log n)$ violates the correctness of the algorithm.

Notice, however, that *most* nodes in the i-th level of the tree avoid a density of the form $\frac{1}{2}(1+\delta)^i$. Indeed, if we were to perform a random walk down the tree, then the node that we encountered on our i-th step would likely have a density bounded above by $\frac{1}{2}(1+\delta)^{O(\sqrt{i})}$. This means that, if we only wanted *most nodes* to behave well, then we could set δ close to $\frac{1}{\sqrt{\log n}}$.

In order to obtain the benefits of $\delta \approx 1/\sqrt{\log n}$ while maintaining the correctness of $\delta \approx 1/\log n$, we smoothly adjust the candidate set size for each subtree as a function of the subtree's density. We show that almost all subtrees are sparse enough to support a "large" candidate set ($\delta \approx 1/\sqrt{\log n}$), while only a small fraction of subtrees require "smaller" candidate sets (with δ closer to $1/\log n$). This means that most parts of the array support fast insertions/deletions, while only a small portion of the array is slow to insert/delete to.

While we have made progress by ensuring that most of the array can support fast updates, this is not sufficient to prove the final bound. Specifically, if the adversary *knows* which parts of the array are slow to update, they could simply focus all of their insertions/deletions on these slow parts of the array, causing the total cost to be large. Instead, we would like to *hide* the slow parts of the array from the adversary. More precisely, we are concerned about two distinct problems: the adversary could *create* dense regions through their insertion sequence (e.g., by concentrating insertions in one location), or, the adversary could *detect* dense regions created by the algorithm (e.g., through prior knowledge of the algorithm's distribution of states.)

History independence comes into play in guarding against these problems. By definition, the first problem cannot happen with a history-independent algorithm, since the configuration of the array does not depend on the adversary's specific sequence of insertions. For the second problem, we add an additional layer of randomness called a *random shift*. At the start of the algorithm, we insert random number $k \in [m]$ of dummy elements at the front of the array, and $m - k$ at the end. This converts a potentially adversarial insertion at rank j to a uniformly random insertion of rank between j and $j + m$. Together with history independence, the random shift ensures that the adversary cannot target specific regions of the array.

To analyze our algorithm, we introduce the notion of a *Zeno random walk*, which is a special type of bounded random walk where the step size decreases as the distance to a boundary decreases. The Zeno walk captures the way in which the densities of subproblems evolve if we perform a random walk down our tree. Our analysis of this random walk (Proposition 43) allows us to bound the cost of a random insertion (Lemma 51). Finally, we extend this analysis for a *random* insertion to an *arbitrary* insertion using the ideas outlined above of history independence and a random shift, achieving an expected $O(\log^{3/2} n)$ cost for any insertion/deletion.

6.2 Zeno's Random Walk

This section describes and analyzes a simple but somewhat unusual type of random walk that we will refer to as a ***Zeno walk***—this random walk will play an important algorithmic role in later sections.

Let $\delta \in (0, 1/2]$. A Zeno walk $Z_0, Z_1, Z_2, \ldots$ starts at $Z_0 = 0$ and deterministically satisfies $Z_i \in (-1, 1)$ for all i. We define $\alpha_i = 1 - |Z_i|$ to be the distance between Z_i and the nearest boundary 1 or -1. We determine Z_{i+1} from Z_i as follows:

- An adaptive adversary selects a quantity $\delta_i \leq \delta$, possibly as a function of $Z_0, Z_1, \ldots, Z_i$.
- Z_{i+1} is then set to be one of $Z_i + \alpha_i\delta_i$ or $Z_i - \alpha_i\delta_i$, each with equal probability.

What makes the Zeno walk unusual is that, the closer it gets to -1 or 1, the smaller its steps become (since the i-th step has its size multiplied by α_i). The result is that (as in Zeno's paradox), the walk can get arbitrarily close to ± 1 but can never reach ± 1.

We will be interested in Zeno walks $Z_1, \ldots, Z_\ell$ where the relationship between δ and the length ℓ of the walk is $\delta = O(1/\sqrt{\ell})$. To gain some intuition here, consider the case where $\delta_i = \delta = 1/\sqrt{\ell}$ for all i, and let us compare the Zeno walk $Z_1, \ldots, Z_\ell$ to a standard unbiased random walk $X_1, \ldots, X_\ell$ that changes by $\pm 1/\sqrt{\ell}$ on each step. After ℓ steps, the random walk $X_1, \ldots, X_\ell$ deviates from the origin by $O(1)$ in *expectation* (but could deviate by much more) and has the property that each step is *deterministically* the same size. The Zeno walk does the complement of this: it deviates from the origin by at most 1 *deterministically*, but to do this it decreases the size of the i-th step by a factor of $1/\alpha_i$. The key property that we will prove (Proposition 42) is that, although the multiplier $1/\alpha_i$ can potentially be large, the *expected value* satisfies $O(1/\alpha_i) = O(1)$ for $i \in [\ell]$. With this intuition in mind, we can now begin the analysis.

Define $Y_i := \ln(1/(1 - Z_i))$. Rather than analyze the Z_i's directly, we will instead analyze the Y_i's. We will see that the sequence $Y_1, Y_2, \ldots$ behaves similarly to the standard random walk $X_1, X_2, \ldots$ that we described in the previous paragraph (except that (1) Y_i is slightly biased and (2) Y_i can never go below $\ln 0.5$). To make this more precise, the next lemma shows that the random walk $Y_1, Y_2, \ldots$ takes steps of size at most $O(\delta)$ and has bias at most $O(\delta^2)$ per step.

Lemma 41. *For $i \geq 0$, we have that*

$$|Y_{i+1} - Y_i| = O(\delta) \tag{6.1}$$

deterministically, and that

$$\left|\mathbb{E}[Y_{i+1} - Y_i \mid Y_1, \ldots, Y_i, \delta_i]\right| = O(\delta^2). \tag{6.2}$$

Proof. Define

$$\gamma_i = \frac{\alpha_i \delta_i}{1 - Z_i}.$$

Note that, if $Z_i \geq 0$, then $\gamma_i = \delta_i$, and otherwise $\gamma_i < \delta_i$. Since $Z_{i+1} = Z_i \pm (1 - Z_i)\gamma_i$, we have that

$$\begin{aligned}
Y_{i+1} &= \ln\left(\frac{1}{1 - Z_i \pm (1 - Z_i)\gamma_i}\right) \\
&= \ln\left(\frac{1}{1 - Z_i} \cdot \frac{1}{1 \pm \gamma_i}\right) \\
&= \ln\left(\frac{1}{1 - Z_i}\right) + \ln\left(\frac{1}{1 \pm \gamma_i}\right) \\
&= Y_i + \ln\left(\frac{1}{1 \pm \gamma_i}\right).
\end{aligned}$$

By a Taylor approximation, we know that $\ln\left(\frac{1}{1 \pm \gamma_i}\right)$ is within $O(\gamma_i^2)$ of $\pm\gamma_i$. That is, Y_{i+1} can be computed from Y_i by first adding $\pm\gamma_i$ at random to Y_i, and then adding/subtracting an additional $O(\gamma_i^2)$. We therefore have that

$$|Y_{i+1} - Y_i| \leq \gamma_i + O(\gamma_i^2) \leq \delta_i + O(\delta_i^2) \leq O(\delta)$$

and that

$$\left|\mathbb{E}[Y_{i+1} - Y_i \mid Y_1, \ldots, Y_i, \gamma_i]\right| \leq O(\gamma_i^2) \leq O(\delta_i^2) \leq O(\delta^2).$$

Using Lemma 41, we can now bound $\mathbb{E}[1/\alpha_\ell]$ for the $\ell = O(1/\delta^2)$-th step of a Zeno walk:

Proposition 42. *For $\ell = O(1/\delta^2)$, we have $\mathbb{E}[1/\alpha_\ell] = O(1)$.*

Proof. By symmetry, it suffices to show that

$$\mathbb{E}[1/\alpha_\ell \cdot \mathbb{I}_{Z_\ell \geq 0}] = O(1),$$

where $\mathbb{I}_{Z_\ell \geq 0}$ is 0-1 indicator random variable for the event $Z_\ell \geq 0$. Note that

$$\begin{aligned}
\mathbb{E}[1/\alpha_\ell \cdot \mathbb{I}_{Z_\ell \geq 0}] &= E[1/(1 - Z_\ell) \cdot \mathbb{I}_{Z_\ell \geq 0}] \\
&\leq E[1/(1 - Z_\ell)],
\end{aligned}$$

so we can complete the proof by showing that

$$\mathbb{E}[1/(1 - Z_\ell)] = O(1). \tag{6.3}$$

Let c be a sufficiently large positive constant and define the sequence $X_1, X_2, \ldots$, where

$$X_i = Y_i - i \cdot c\delta^2.$$

This means that $X_{i+1} - X_i = Y_{i+1} - Y_i - c\delta^2$, so we can think of the X_i's as being a modification of the Y_i's that eliminates any upward bias that the Y_i's might have (recall by Lemma 41 that the Y_i's have bias at most $O(\delta^2)$).

Formally, one can apply Lemma 41 to deduce that the X_i's are a supermartingale with bounded differences of $O(\delta)$. That is, by (6.2) we have $\mathbb{E}[X_{i+1} \mid X_1, \ldots, X_i] \le X_i$ (so the X_i's form a supermartingale) and by (6.1) we have $|X_{i+1} - X_i| \le O(\delta)$ (so the martingale has bounded differences of $O(\delta)$).

We can apply Azuma's inequality for supermartingales with bounded differences to deduce the following tail bound. For $k \ge 1$, we have

$$\Pr[X_i \ge \delta k\sqrt{i}] \le e^{-\Omega(k^2)}.$$

Unrolling the definition of X_i, we get that

$$\Pr[\ln(1/(1-Z_i)) \ge \delta k\sqrt{i} + ic\delta^2] \le e^{-\Omega(k^2)}.$$

Plugging in $i = \ell = O(1/\delta^2)$, we conclude that

$$\Pr[\ln(1/(1-Z_\ell)) \ge \Omega(k)] \le e^{-\Omega(k^2)}.$$

This further simplifies to

$$\Pr\left[1/(1-Z_\ell) \ge e^{\Omega(k)}\right] \le e^{-\Omega(k^2)},$$

which implies (6.3), and completes the proof. ∎

We conclude the section by generalizing Zeno walks to take place in an arbitrary interval $(\lambda - \varepsilon, \lambda + \varepsilon)$. This works exactly as before, except that now the Zeno walk begins at $Z_0 = \lambda$; it deterministically stays in the interval $(\lambda - \varepsilon, \lambda + \varepsilon)$; it sets $\alpha_i = \varepsilon - |Z_i - \lambda|$ to be the distance from Z_i to the nearest boundary $\lambda - \varepsilon$ or $\lambda + \varepsilon$; and then $Z_{i+1} = Z_i \pm \alpha_i \delta_i$ where $\delta_i \le \delta$ is selected by an adversary. Equivalently, a sequence $\{Z_i\}$ is a Zeno walk in the interval $(\lambda - \varepsilon, \lambda + \varepsilon)$ if $\{(Z_i - \lambda)/\varepsilon\}$ is a Zeno walk in $(-1, 1)$ (and the two Zeno walks have the same parameter δ as each other). Thus we get the following generalization of Proposition 42.

Proposition 43. *Consider a Zeno walk in $(\lambda - \varepsilon, \lambda + \varepsilon)$. For $\ell = O(1/\delta^2)$, we have $\mathbb{E}[1/\alpha_\ell] \le O(\varepsilon^{-1})$.*

6.3 The Zeno Embedding: a Data Structure for $m \ge (1+\varepsilon)n$

In this section, we give a list-labeling solution for $m \ge (1+\varepsilon)n$ that achieves expected cost $O(\varepsilon^{-1}\log^{3/2} n)$ per insertion and deletion. We will treat $m \in \mathbb{N}$ and $\varepsilon \in (0,1)$ as being fixed, and we will allow the number n of elements to vary subject to the constraint that $m \ge (1+\varepsilon)n$. We will also assume without loss of generality that n

is at least a sufficiently large positive constant.

We construct and analyze the data structure in three phases. First, we describe a certain type of static construction, which we call the ***Zeno embedding***, for how to embed n elements into m slots. Then we show how to dynamize the Zeno embedding in order to efficiently implement *random* insertions/deletions. Finally, we present one last modification to the Zeno embedding in order to implement arbitrary insertions/deletions efficiently.

6.3.1 The Static Zeno Embedding

The Zeno embedding treats the array as having a simple recursive structure: the ***level-0 subproblem*** consists of the entire array; and the ***level-i subproblems*** each consist of either $\lfloor m/2^i \rfloor$ or $\lceil m/2^i \rceil$ contiguous slots in the array.

Each level-i subproblem S is either a ***base case*** (meaning it does not have child subproblems) or has two recursive children. If S has $q \in \{\lfloor m/2^i \rfloor, \lceil m/2^i \rceil\}$ slots, then the children of S have $\lfloor q/2 \rfloor$ and $\lceil q/2 \rceil$ slots, respectively. Here we are taking advantage of the basic mathematical fact that

$$\{\lfloor \lfloor m/2^i \rfloor /2 \rfloor, \lfloor \lceil m/2^i \rceil /2 \rfloor, \lceil \lfloor m/2^i \rfloor /2 \rceil, \lceil \lceil m/2^i \rceil /2 \rceil\} \subseteq \{\lfloor m/2^{i+1} \rfloor, \lceil m/2^{i+1} \rceil\}.$$

For each level-i subproblem S, define $|S|$ to be the number of elements stored in that subproblem, and define the ***density*** μ_S of the subproblem to be

$$\mu_S = \frac{|S|}{n/2^i}.$$

Note that in the definition of μ_S, the denominator is the average number of elements per level-i subproblem, which means that μ_S can be greater than 1. In fact, we will guarantee deterministically that $\mu_S \in [1 - \varepsilon/2, 1 + \varepsilon/2]$. The upper bound will ensure correctness (i.e., that no subproblem overflows), and the lower bound will ensure that every pair of consecutive elements are within $O(1)$ slots of each other.

We can now describe how to implement a given level-i subproblem S. Define

$$\alpha_S = \varepsilon/2 - |1 - \mu_s|$$

to be the distance between μ_S and the nearest boundary $\{1 - \varepsilon/2, 1 + \varepsilon/2\}$. Let $x_1, \ldots, x_{|S|}$ denote the elements of S in sorted order. Define the ***pivot candidate set*** for S to be

$$C_S = \left\{ x_i \;\middle|\; \frac{|S|}{2} - \frac{n}{2^i} \cdot \frac{\alpha_S}{\sqrt{\log n}} \le i \le \frac{|S|}{2} + \frac{n}{2^i} \cdot \frac{\alpha_S}{\sqrt{\log n}} \right\}.$$

Roughly speaking, C_S consists of the elements representing the middle $\Theta(\alpha_S/\sqrt{\log n})$-fraction of the subproblem.

If $|C_S| \le 4$, we declare S to be a ***base case***, and we spread the elements of S

evenly across its slots. Otherwise, we define the ***pivot*** p_S for S to be an element of C_S chosen uniformly at random. The elements $x_i \le p_S$ are recursively placed in S's left child, and the elements $x_i > p_S$ are recursively placed in S's right child.

Later on, when we discuss the *dynamic* Zeno embedding, we will see several ways that one can implement the random choice of p_S. For concreteness, we will mention one natural approach here: define $h_0, h_1, h_2, \ldots, h_{O(\log n)}$ to be an independent sequence of hash functions[1] where each h_i maps each element to a uniformly random real number in $[0, 1]$, and set

$$p_S = \operatorname{argmin}_{x \in C_S} h_i(x).$$

The key property of the Zeno embedding is that if we perform a random walk down the recursive tree, then the densities μ_S that we encounter form an $O(\log n)$-step Zeno walk in the interval $[1 - \varepsilon/2, 1 + \varepsilon/2]$:

Lemma 44. *Fix any outcomes for the hash functions $h_0, h_1, h_2, \ldots$. Consider a random walk $S_0, S_1, S_2, \ldots, S_\ell$ down the recursion tree, where each S_{i+1} is a random child of S_i, and S_ℓ is a base-case subproblem. Then the sequence $\{\mu_{S_i}\}_{i=1}^{\ell}$ is a Zeno walk on $[1 - \varepsilon/2, 1 + \varepsilon/2]$ with $\delta = O(1/\sqrt{\log n})$.*

Proof. Recall that a Zeno walk on $[1-\varepsilon/2, 1+\varepsilon/2]$ is any walk $Z_0, Z_1, \ldots$ that starts at 1 and takes the following form: each step $Z_{i+1} - Z_i$ is randomly $\pm \alpha_i \delta_i$ for some $\delta_i \le \delta$ (that may be chosen by an adversary) and where $\alpha_i = \varepsilon/2 - |1 - Z_i|$. Or, equivalently, each step $Z_{i+1} - Z_i$ is randomly $\pm \beta_i$ for some $\beta_i \le \delta\left(\varepsilon/2 - |1 - Z_i|\right)$.

Consider a non-base-case subproblem S_i, and let A and B be the child subproblems of S_i. By construction,

$$\big||A| - |B|\big| = O\left(\frac{n}{2^i} \cdot \frac{\alpha_S}{\sqrt{\log n}}\right).$$

Since $|A| + |B| = |S_i|$, we have that $\mu_A + \mu_B = 2\mu_{S_i}$ and

$$|\mu_A - \mu_B| = \frac{\big||A| - |B|\big|}{n/2^{i+1}} = O\left(\frac{\alpha_{S_i}}{\sqrt{\log n}}\right).$$

Thus, since S_{i+1} is randomly one of A or B, we have that $\mu_{S_{i+1}}$ is randomly one of

$$\mu_{S_i} + \beta_i \text{ or } \mu_{S_i} - \beta_i,$$

where

$$\beta_i = |\mu_A - \mu_B|/2 = O\left(\frac{\alpha_{S_i}}{\sqrt{\log n}}\right) = O\left(\delta\left(\varepsilon/2 - |1 - \mu_s|\right)\right).$$

Thus the sequence $\{\mu_s\}$ is a Zeno walk on $[1 - \varepsilon/2, 1 + \varepsilon/2]$ with $\delta = O(1/\sqrt{\log n})$.

[1]Technically, our data structure does not necessarily have access to the internal values of elements, so it cannot compute a hash $h_i(x)$ of any given element. However, we can simulate a hash function h_i by assigning each element x a random value $h_i(x)$ when the element is inserted.

For clarity, we remark that the definition of the Zeno walk includes an adaptive adversary who chooses $\delta_i < \delta$. The adversary for the Zeno walk in this lemma simply chooses a pivot uniformly at random from the pivot candidate set, which determines δ_i. ∎

The reason that Lemma 44 is important is that it allows for us to bound the quantities α_S^{-1}. Indeed, we use Proposition 43 to prove the following inequality.

Lemma 45. *Let $\mathcal{S}_i$ be the set of level-i subproblems. Then*

$$\frac{1}{2^i}\sum_{S\in\mathcal{S}_i}\alpha_S^{-1} = O(\varepsilon^{-1}).$$

Proof. Fix any outcomes for the hash functions $h_0, h_1, h_2, \ldots$. Consider a random walk $S_0, S_1, S_2, \ldots, S_\ell$ down the recursion tree, where each S_{i+1} is a random child of S_i, and S_ℓ is a base-case subproblem. Lemma 44 tells us that $\{\mu_{S_i}\}_{i=1}^{\ell}$ is a Zeno walk on $[1-\varepsilon/2, 1+\varepsilon/2]$ with $\delta = O(1/\sqrt{\log n})$ (and, moreover, α_{S_i} corresponds to α_i in the Zeno walk).

For $i \in [0, \log m]$, define $\overline{\alpha}_i$ to be α_{S_i} if S_i exists and 0 otherwise (i.e., if $i > \ell$). Proposition 43 tells us that, for each $i \in [0, \log m]$,

$$\mathbb{E}[1/\overline{\alpha}_i] = O(\varepsilon^{-1}). \tag{6.4}$$

On the other hand, each level-i subproblem has probability exactly $1/2^i$ of being S_i. Thus

$$\mathbb{E}[1/\overline{\alpha}_i] = \frac{1}{2^i}\sum_{S\in\mathcal{S}_i}\alpha_S^{-1}. \tag{6.5}$$

Combined, (6.4) and (6.5) imply the lemma. ∎

It is interesting to note that, whereas Lemma 44 is a statement about random walks, Lemma 45 is a *deterministic* bound on the α_S^{-1}s, even though it uses a probabilistic argument to derive the bound.

Lastly, we also need to explicitly show that no subproblem ever overflows:

Lemma 46. *Each level-i subproblem S satisfies $|S| \le \lfloor m/2^i \rfloor$.*

This lemma is a technicality that is essentially immediate from the fact that each subproblem S has density $\mu_S \le 1+\varepsilon/2$. The only difficulty in the proof comes from the necessity to carefully handle floors/ceilings. Nonetheless, for completeness, we include the proof below.

Proof. By construction, each level-i subproblem S has

$$|C_S| \le 2\alpha_S \cdot \frac{n}{2^i} \le \varepsilon \cdot \frac{n}{2^i}.$$

Thus, if $|C_S| > 4$ (i.e., S is a non-base-case subproblem), we must have $\varepsilon n/2^i \geq 4$. Since every base-case subproblem is the child of a non-base-case subproblem, we have that for base-case subproblems $\varepsilon n/2^{i-1} \geq 4$. This means that every subproblem S is in a level i satisfying

$$\frac{\varepsilon n}{2^i} \geq 2. \tag{6.6}$$

We wish to show that $\lfloor \frac{m}{2^i} \rfloor - |S| \geq 0$. We know that

$$\left\lfloor \frac{m}{2^i} \right\rfloor - |S| = \left\lfloor \frac{m}{2^i} \right\rfloor - \mu_S \frac{n}{2^i} \geq \frac{m}{2^i} - \mu_S \frac{n}{2^i} - 1 \geq \frac{(1+\varepsilon)n - \mu_S n}{2^i} - 1.$$

Since $\mu_S \leq 1 + \varepsilon/2$, it follows that

$$\left\lfloor \frac{m}{2^i} \right\rfloor - |S| \geq \frac{\varepsilon n/2}{2^i} - 1.$$

By (6.6), we can conclude that $\lfloor \frac{m}{2^i} \rfloor - |S| \geq 0$, as desired. ∎

6.3.2 Dynamizing the Zeno Embedding

We now describe a dynamic version of the Zeno embedding; we will treat m and ε as fixed, and allow n to vary subject to the constraint that $n \geq (1+\varepsilon)m$.

We note that, in this section we will focus on analyzing *random* insertions/deletions, that is, an insertion/deletion that is performed at a random rank (in an array with arbitrary contents). Our solution will be history independent, and we will see in the next subsection that this allows the random-rank assumption to be removed.

Implementing insertions and deletions. To implement an insertion/deletion in the Zeno embedding, we simply update the embedding to account for the element being added/removed. More concretely, we can implement an insertion/deletion of an element x as follows. We will describe the process recursively, focusing on how to insert/delete x into a given level-i recursive subproblem S. The insertion/deletion of x may change the values of μ_S, α_S, C_S, and p_S. Note that the values of C_s and p_S can change regardless of whether the insertion/deletion of x takes place in the candidate set. If it changes the pivot p_S, or if S is a base-case, then we implement the insertion/deletion by rebuilding the entire subproblem from scratch, incurring a cost of $O(n/2^i)$. Otherwise, we recursively insert/delete x into either the left child (if $x \leq p_S$) or the right child (if $x > p_S$). Once the insertion/deletion is complete, the Zeno embedding will be the same as if it were constructed from scratch on the current set of elements.

As described in the static Zeno embedding, there are multiple ways to implement randomly choosing a pivot. One way is to use the hash functions h_i described in the previous subsection. This means that a level-i subproblem S being inserted/deleted into gets rebuilt if $\text{argmin}_i\{h_i(x) \mid x \in C_S\}$ is changed by the insertion/deletion. We note that, in this construction, the hash functions are fixed at the very beginning and are never resampled (even when subproblems are rebuilt).

Another way to implement the random choice of pivot is to use reservoir sampling [49, 70, 242, 325, 348]. This means that, when a subproblem is first built (or rebuilt), it picks a random $x \in C_S$ to be the pivot; whenever an element x is added to C_S, it has probability $1/|C_S \cup \{x\}|$ of becoming the pivot; and whenever an element x is removed from C_S, if x was the pivot, then a random element in $C_S \setminus \{x\}$ is chosen as the new pivot. Like the hashing method, reservoir sampling maintains as an invariant that each candidate in C_S is equally likely to be the pivot.

Each of the two methods (hashing and reservoir sampling) have their own benefits: reservoir sampling can be used to immediately obtain an algorithm in the RAM-model that has the same asymptotic running time as its list-labeling cost, while hashing, on the other hand, ensures that the embedding is deterministic after fixing the hash functions. In our formal arguments, we use the hash function method, but this can easily be replaced with reservoir sampling.

Analyzing a random insertion/deletion. To begin analyzing the dynamic Zeno embedding, we observe that, by construction, the dynamic Zeno embedding is insertion/deletion symmetric and history independent.

Observation 47. *The dynamic Zeno embedding is insertion/deletion symmetric and history independent.*

Due to the insertion/deletion symmetry, the expected cost of a random insertion on an array with n elements is the same as the expected cost of a random deletion on an array with $n + 1$ elements. Thus we need only analyze the expected cost of a random deletion.

We will analyze the probability that the deletion of an element x causes the rebuild of a subproblem. More precisely, we say that a subproblem S is ***rebuilt*** if the pivot of S changes, while the pivots of all of the ancestors of S do not change.

Next, we will prove that, if we delete an element x, and S is the level-i subproblem that contains x, then the probability that S is rebuilt is $O(|C_S|^{-1})$.

Lemma 48. *If an element x is deleted from a subproblem S, then S is rebuilt with probability*

$$O\left(|C_S|^{-1}\right).$$

Proof. If S is a base-case subproblem, either before or after the deletion, then $|C_S| = O(1)$, and the lemma is trivial. Now, suppose S is not a base-case subproblem.

Let C_S denote the pivot candidate set prior to the deletion of x, and let $\overline{C}_S$ denote the pivot candidate set after the deletion. Each time that we add/remove an element to/from C_S, the probability that $p_S = \operatorname{argmin}_{x \in C_S} h_i(x)$ changes is $\Theta(1/|C_S|)$. It therefore suffices to show that C_S and $\overline{C}_S$ have a symmetric difference of at most $O(1)$ elements.

We can think of the transformation of C_S into $\overline{C}_S$ as taking place in three steps. First we update

$$\alpha_S = \varepsilon/2 - \left|1 - \frac{|S|}{n/2^i}\right|$$

to become

$$\alpha_S = \varepsilon/2 - \left|1 - \frac{|S|-1}{n/2^i}\right|.$$

This changes α_S by at most $\pm\frac{1}{n/2^i}$, which changes the set

$$C_S = \left\{x_i \;\middle|\; \frac{|S|}{2} - \frac{n}{2^i} \cdot \frac{\alpha_S}{\sqrt{\log n}} \le i \le \frac{|S|}{2} + \frac{n}{2^i} \cdot \frac{\alpha_S}{\sqrt{\log n}}\right\} \tag{6.7}$$

by at most $O(1)$ elements. Second, we replace $|S|$ in (6.7) with $|S| - 1$. This again changes the set C_S by at most $O(1)$ elements. Third, we remove the element x; if $x = x_j$ for some j, then the removal of x has the effect of decrementing the index of each x_i with $i \ge j$. This again changes C_S by at most $O(1)$ elements.

Combined, the three steps complete the transformation of C_S into $\overline{C}_S$, meaning that $\overline{C}_S$ and C_S have a symmetric difference of $O(1)$ elements, as desired. ∎

Lemma 48 immediately implies a bound on the expected cost incurred from rebuilding S.

Lemma 49. *If an element x is deleted from a level-i subproblem S, the expected cost incurred from possibly rebuilding S is*

$$O\left(\frac{n/2^i}{|C_S|}\right).$$

Proof. A rebuild of S costs $\Theta(n/2^i)$. Thus the lemma follows from Lemma 48. ∎

Observe that, by design,

$$\frac{n/2^i}{|C_S|} = O(\alpha_S^{-1}\sqrt{\log n}).$$

This is where Lemma 45 comes into play: it tells us that even though $\frac{n/2^i}{|C_S|}$ may be large for some subproblems S, it cannot be consistently large across all subproblems. Using this, we can analyze the expected cost to delete a random element.

Lemma 50. *The expected cost to delete a random element x from the Zeno embedding is $O(\varepsilon^{-1}\log^{3/2} n)$.*

Proof. Let $\mathcal{S}_i$ denote the set of level-i subproblems (prior to the deletion). Each

$S \in \mathcal{S}_i$ contains $\Theta(n/2^i)$ elements, so

$$\Pr[x \in S] = \Theta\left(\frac{1}{2^i}\right).$$

If $x \in S$, then we have by Lemma 49 that S incurs expected rebuild cost

$$O\left(\frac{n/2^i}{|C_S|}\right) = O(\alpha_S^{-1}\sqrt{\log n}).$$

The expected cost from rebuilds in the i-th level of recursion is therefore at most

$$O\left(\sum_{S \in \mathcal{S}_i} \frac{1}{2^i} \cdot \alpha_S^{-1}\sqrt{\log n}\right),$$

which by Lemma 45 is at most

$$O\left(\varepsilon^{-1}\sqrt{\log n}\right).$$

Summing over the $O(\log n)$ levels of recursion, the total expected cost of the deletion is $O(\varepsilon^{-1}\log^{3/2} n)$. ∎

Due to the previously described symmetry between insertions and deletions, the same lemma is true for insertions.

Lemma 51. *The expected cost to insert an element x with a random rank in $\{1, 2, \ldots, n+1\}$ into the Zeno embedding is $O(\varepsilon^{-1}\log^{3/2} n)$.*

6.3.3 Achieving a Bound on Arbitrary Insertions/Deletions.

So far, we have only analyzed random insertions/deletions. At first glance, this may seem like an insignificant accomplishment. (Indeed, it is already known that random insertions/deletions can be supported in $O(\varepsilon^{-1})$ amortized time per operation [87].)

What makes the Zeno embedding special is that it is history independent. We will now show how to reduce the list-labeling problem (with *arbitrary* insertions/deletions) to the problem of constructing an insertion/deletion-symmetric history-independent embedding that supports efficient *random* insertions/deletions.

Within any history-independent data structure, the expected cost to perform a deletion at rank r on an array of size m containing n elements can be expressed by a ***cost function*** $T(m, n, r)$ only dependent on m, n and r. Moreover, if the data structure is insertion/deletion symmetric, then the same cost function T expresses the expected cost for an insertion; specifically, the expected cost to perform an insertion at rank r on an array of size m containing n elements is $T(m, n-1, r)$.

To reduce from the arbitrary insertion/deletion case to the random insertion/deletion case, we will show that given any (insertion/deletion-symmetric) history-independent algorithm $\mathcal{A}$ with cost function $T(m,n,r)$, we can construct a history-independent algorithm $\mathcal{B}$ with cost function $T'(m,n,r)$ such that for each individual rank r, the cost $T'(m,n,r)$ is upper bounded by the *average* of the costs $T(m,n,r)$ across all ranks (up to constant factors).

Lemma 52. *Suppose there is an insertion/deletion-symmetric history-independent algorithm $\mathcal{A}$ whose cost is determined by a function $T(m,n,r)$. Then we can construct a new insertion/deletion-symmetric history-independent algorithm $\mathcal{B}$ with cost function $T'(m,n,r)$ satisfying*

$$T'(m,n,r) = O\left(\frac{1}{m+1}\sum_{j=1}^{2m} T(2m, m+n, j)\right)$$

for all r.

Proof. Fix a history-independent algorithm $\mathcal{A}$. We will construct a history-independent algorithm $\mathcal{B}$. We will describe the behavior of the algorithm $\mathcal{B}$ on an array of size m with an arbitrary sequence $\mathcal{S}$ of insertions/deletions.

To do so, we will construct from $\mathcal{S}$ an input to $\mathcal{A}$. The input to $\mathcal{A}$ is an array of size $2m$ with the following insertion/deletion sequence. First we insert m dummy elements as follows. Let q be a uniformly random integer in $[0,m]$. Insert q dummy elements that are treated as taking infinitely small values (i.e., $-\infty$), and insert $m-q$ dummy elements that are treated as taking infinitely large values (i.e., ∞). Now, execute the sequence $\mathcal{S}$.

Now, define $\mathcal{B}$ as the algorithm that behaves identically to $\mathcal{A}$ on $\mathcal{A}$'s subarray $[q, q+m]$ (that is, $\mathcal{A}$'s subarray from the q^{th} slot to the $q+m^{th}$ slot), ignoring the dummy elements. That is, for all i, after the i^{th} insertion from $\mathcal{S}$, the subarray $[q, q+m]$ of $\mathcal{A}$'s array with the dummy elements removed, is identical to $\mathcal{B}$'s array.

We note that $\mathcal{B}$ is well defined in the sense that all elements of $\mathcal{S}$ always appear in $\mathcal{A}$'s subarray $[q, q+m]$. This is simply due to the existence of the dummy elements in $\mathcal{A}$'s array.

Now let us bound the expected cost $T'(m,n,r)$ for $\mathcal{B}$ to perform a deletion at rank r. This corresponds to a deletion at rank $r+q$ in $\mathcal{A}$, which has cost $T(2m, m+n, r+q)$. Notice, however, that $r+q$ is a random element in $\{r, r+1, \ldots, r+m\}$. Thus,

$$T'(m,n,r) = \frac{1}{m+1}\sum_{j=r}^{r+m} T(2m, m+n, j),$$

which in turn is at most

$$O\left(\frac{1}{m+1}\sum_{j=1}^{2m} T(2m, m+n, j)\right).$$

∎

In the case where $\mathcal{A}$ is the Zeno embedding, we refer to $\mathcal{B}$ as the ***shifted Zeno embedding***. Now, we are ready to put everything together and prove our main theorem, that the shifted Zeno embedding incurs expected cost $O(\varepsilon^{-1}\log^{3/2} n)$ per insertion/deletion.

Theorem 40. *Let $\varepsilon \in (0,1)$, and suppose $m \geq (1+\varepsilon)n$, where m is a static value while n changes dynamically. The shifted Zeno embedding on an array of size m with n elements incurs expected cost $O(\varepsilon^{-1}\log^{3/2} n)$ per insertion/deletion.*

Proof. Let $T(m,n,r)$ be the cost function associated with the Zeno embedding, and let $T'(m,n,r)$ be the cost function associated with the shifted Zeno embedding. From Lemma 52, we know that

$$T'(m,n,r) = O\left(\frac{1}{m+1}\sum_{j=1}^{2m} T(2m, m+n, j)\right). \tag{6.8}$$

The right side of Equation 6.3.3 is within a constant factor of the average value of $T(2m, m+n, j)$ over all ranks j. Thus, it is within a constant factor of the expected value of $T(2m, m+n, j)$ where j is chosen uniformly at random over all ranks, which we know from Lemmas 50 and 51, is $O(\varepsilon^{-1}\log^{3/2} n)$. Thus, $T'(m,n,r) = O(\varepsilon^{-1}\log^{3/2} n)$, as desired. ∎

The following corollary follows immediately by applying Theorem 40 to an $n(1+\varepsilon)$ sized subarray of a linearly sized array for any $\varepsilon < 1$.

Corollary 53. *There exists a list-labeling algorithm for an array of size $m = n(1+\Theta(1))$ with expected cost $O(\log^{3/2} n)$ per insertion/deletion.*

We can also use the theorem to bound the total cost to insert into every slot in an array.

Corollary 54. *There exists a list-labeling algorithm to fill an array of size m from empty to full with expected total cost $O(m\log^{2.5} m)$.*

Proof. We will apply a shifted Zeno embedding in $\Theta(\log m)$ phases, using an ε_i, defined below, for phase i and rebuilding the array between phases. The first phase consists of the first $m/2$ insertions, and each phase inserts half as many elements as the preceding phase. This continues until $n > m - \log m$, at which point the final phase consists of inserting the remaining at most $\log m$ elements.

More precisely, let $k = \lceil \log m - \log\log m \rceil$, and define

$$n_i = \frac{m(2^i - 1)}{2^i} \quad \text{for } i = 0, 1, \ldots, k,$$

and $n_{k+1} = m$.

Items are inserted by ranks, specified by $r_1, \ldots, r_m$, so that for example, since the first insertion is into an empty array, $r_1 = 1$. Phase P_i is defined by the insertions r_j with $j \in (n_{i-1}, n_i]$. We define $\varepsilon_i = (2^i - 1)^{-1}$ for $i > 1$, and $\varepsilon_1 = \frac{m-1}{m}$.

Let $C(P_i)$ denote the expected total cost of the insertions in phase P_i. For $i > 1$ and for all $j \in (n_{i-1}, n_i]$,

$$(1+\varepsilon_i)j \le (1+\varepsilon_i)n_i = \left(1 + \frac{1}{2^i - 1}\right)\left(\frac{m(2^i-1)}{2^i}\right) = m.$$

Similarly, in phase P_1, we have

$$(1+\varepsilon_i)j \le \left(1 + \frac{m-1}{m}\right) \cdot \frac{m}{2} \le m.$$

Therefore, we can apply Theorem 40 to say that for all i, an insertion during phase P_i incurs expected cost

$$O(\varepsilon_i^{-1} \log^{3/2} n) = O(2^i \log^{3/2} m),$$

and

$$C(P_i) = O\left(2^i \cdot \frac{m}{2^i} \cdot \log^{3/2} m\right) = O(m \log^{3/2} m).$$

Summing over the first $k = O(\log m)$ phases, this gives expected total insertion cost $O(m \log^{2.5} m)$.

By construction, the final phase has at most $\log m$ insertions, and thus has total expected cost $O(m \log m)$. Finally, since the total number of elements in the array is bounded by m, the rebuilds between phases incur total cost $O(m \log m)$, completing the proof.

We conclude the section with a remark.

Remark 55. *Many applications of list labeling require that, if $m = \Theta(n)$, then the number of empty slots between any two consecutive elements is at most $O(1)$. The Zeno embedding satisfies this property by design, since each subproblem has density at least $1-\varepsilon/2$. The shifted Zeno embedding therefore also satisfies the same property.*

6.4 Upper Bound For Sparse Arrays

Define the ***τ-sparse list-labeling problem*** to be the list-labeling problem in the regime of $n \le m/\tau$. Previously in this chapter, we studied the setting where $\tau = O(1)$. In this section, we extend our upper bounds to apply to the sparse regime where $m = \tau n$ for some $16 \le \tau \le n^{o(1)}$. We do this via a simple general-purpose reduction from the sparse setting to the linear setting.

We will prove the following proposition:

Proposition 56. *Let T be a non-negative convex function satisfying $T(\Theta(i)) = \Theta(T(i))$ for all i and satisfying $T(0) = 0$. Let $16 \le \tau \le n^{o(1)}$. If there exists a 2-sparse list-labeling solution whose expected amortized cost is upper bounded by $T(\log n)$, then there exists a τ-sparse list-labeling solution whose expected amortized* *cost is upper bounded by*

$$O\left(T\left(\frac{\log n}{\log \tau}\right) \cdot \log \tau\right).$$

Combining Proposition 56 and Corollary 53, we obtain the following upper bound for the sparse regime:

Theorem 57. *For $16 \leq \tau \leq n^{o(1)}$, there exists a solution to the τ-sparse list-labeling problem with expected amortized cost upper bounded by*

$$O\left(\frac{\log^{3/2} n}{\sqrt{\log \tau}}\right).$$

To prove Proposition 56, we introduce an intermediate problem that we call the ***bucketed list-labeling problem***. In this problem, there are m buckets and up to $N = \Omega(m)$ elements at a time, with elements being inserted and deleted as in the classical list-labeling problem. Elements must be assigned to buckets so that, if two elements a and b are assigned to buckets $u \neq v$, then $a < b \iff u < v$. The cost of adding/removing an element to/from a bucket is 0 when that element is inserted/deleted, but the cost of rearranging items is equal to the *sum* of the sizes of the buckets containing those items. (So even moving one item from a bucket u to a bucket v costs $|u| + |v|$). Finally, each bucket has a maximum capacity of $8N/m$ elements.

Our next lemma reduces bucketed list labeling to 2-sparse list labeling.

Lemma 58. *Let T be a non-negative convex function. If there exists a 2-sparse list-labeling solution whose expected amortized cost is upper bounded by $T(\log n)$, then there exists a bucketed list-labeling solution whose expected amortized cost is upper bounded by*

$$O\left(T\left(\log m\right)\right).$$

Proof. An important component of our bucketed list-labeling solution is to partition the elements into up to $m/2$ disjoint ***blocks***, where each block contains up to $8N/m$ consecutive elements. We maintain these blocks using hysteresis: every time that a block's size falls below $2N/m$ (due to deletions), we merge it with an adjacent block (unless there is only one block in the system); and every time that a block's size exceeds $8N/m$ (due to insertions or merges), we split that block into two blocks of equal size. Note that a block's size can never exceed $10N/m$ because a block of size $\leq 8N/m$ can be merged with a block of size $< 2N/m$, and there is no way to create a larger block. Thus, after a split, the size of each resulting block is between $4N/m$ and $5N/m$. Starting from an empty array, during a sequence of k insertions/deletions, the number of block splits/merges will be at most $O(km/N)$.

To construct a bucketed list-labeling solution, we treat the m buckets as slots in an array of size m, and we treat the up-to-$m/2$ blocks as elements that reside in that array. This allows for us to treat the bucketed list-labeling problem as a 2-sparse list-labeling problem: block splits corresponded to element insertions in the 2-sparse list-labeling problem; and block merges correspond to element deletions in the 2-sparse list-labeling problem.

If an operation incurs cost S in the 2-sparse list-labeling problem, then it incurs cost $O(S \cdot N/m)$ in the bucketed list-labeling problem (since each element in the former problem corresponds to a block of $O(N/m)$ elements in the latter problem). On the other hand, starting from an empty array, if k insertions/deletions are performed in the bucketed list-labeling problem, the number of insertions/deletions in the 2-sparse list-labeling problem will only be $O(km/N)$. Combining these with the assumption that the 2-sparse list-labeling problem incurs cost $T(\log n)$, we have that the total cost of the bucketed list-labeling problem is $O(T(\log(m/2)) \cdot N/m \cdot k \cdot m/N) = O(kT(\log m))$, thus the amortized cost of the bucketed list-labeling problem is $O(T(\log m))$. ∎

Next we reduce sparse list labeling to bucketed list labeling.

Lemma 59. *Let T be a non-negative convex function satisfying $T(\Theta(i)) = \Theta(T(i))$ for all i and satisfying $T(0) = 0$. Let $16 \le \tau \le n^{o(1)}$. If there exists a bucketed list-labeling solution whose expected amortized cost is upper bounded by $T(\log m)$, then there exists a τ-sparse list-labeling solution whose expected amortized cost is upper bounded by*

$$O\left(T\left(\frac{\log n}{\log \tau}\right) \cdot \log \tau\right).$$

Proof. We may assume without loss of generality that τ is a natural number. We prove the result by induction on τ. The base case of $16 \le \tau \le O(1)$ is trivial, since we can break the array into $\Theta(n)$ chunks of size $\Theta(1)$ and treat each chunk as a bucket in the bucketed list-labeling problem.

Now suppose that $\omega(1) \le \tau \le n^{o(1)}$. Let c be a large positive constant (to be selected later), and partition the array into $n^{c/\log \tau}$ chunks of size $m' = \lfloor m/n^{c/\log \tau} \rfloor$ slots each (possibly orphaning $O(n^{c/\log \tau})$ slots due to rounding errors). Treat each of the $n^{c/\log \tau}$ chunks as a bucket, and assign elements to chunks using bucketed list labeling. By assumption, bucketed list labeling has expected amortized cost $T(\log m)$ where m is the number of buckets, and plugging in the value $n^{c/\log \tau}$ for m we get that the expected amortized cost of the bucketed list labeling instance is:

$$T(\log n^{c/\log \tau}) = T\left(\frac{c \log n}{\log \tau}\right)$$

per operation. Since $T(\Theta(i)) = \Theta(T(i))$, we can further bound the above cost to be at most

$$c' \cdot T\left(\frac{\log n}{\log \tau}\right), \tag{6.9}$$

where c' is a constant determined by c.

By design, each chunk contains at most $n' = 8n/n^{c/\log \tau}$ elements, so

$$\frac{m'}{n'} \geq \frac{\lfloor m/n^{c/\log \tau} \rfloor}{8n/n^{c/\log \tau}} \geq \frac{m}{16n} \geq \tau/16.$$

Thus we can recursively implement each chunk as an instance of $\frac{\tau}{16}$-sparse list labeling. By our inductive hypothesis for $\tau' = \frac{\tau}{16}$, we have that for every sufficiently large positive constant Q, the expected amortized cost of performing an insertion/deletion in a given chunk is at most

$$\begin{aligned}
&Q \cdot T\left(\frac{\log n'}{\log \tau'}\right) \cdot \log \tau' \\
&= Q \cdot T\left(\frac{(1 - c/\log \tau + 3/\log n)\log n}{(1 - 4/\log \tau)\log \tau}\right) \cdot (1 - 4/\log \tau)\log \tau \\
&\leq Q \cdot T\left(\frac{(1 - 1/\log \tau)\log n}{\log \tau}\right) \cdot \log \tau \qquad \text{(since } c \geq 8\text{)} \\
&\leq Q \cdot (1 - 1/\log \tau) \cdot T\left(\frac{\log n}{\log \tau}\right) \cdot \log \tau \quad \text{(since } T \text{ is convex and } T(0) = 0\text{)} \\
&\leq Q \cdot T\left(\frac{\log n}{\log \tau}\right) \cdot \log \tau - Q \cdot T\left(\frac{\log n}{\log \tau}\right).
\end{aligned}$$

Combining this with (6.9), the total expected amortized cost of an insertion/deletion is at most

$$c' \cdot T\left(\frac{\log n}{\log \tau}\right) + Q \cdot T\left(\frac{\log n}{\log \tau}\right) \cdot \log \tau - Q \cdot T\left(\frac{\log n}{\log \tau}\right).$$

Choosing Q to be at least c', this is at most

$$Q \cdot T\left(\frac{\log n}{\log \tau}\right) \cdot \log \tau,$$

which completes the proof by induction. ∎

Lemmas 58 and 59 directly imply Proposition 56, completing the section.

Chapter 7

A Lower Bound for History-Independent Solutions

The shifted Zeno embedding (Theorem 40) has the property that it is ***history independent***, meaning that the state of the data structure does not reveal any information about the history of insertions/deletions. In this chapter, we prove that the $\varepsilon^{-1}\log^{3/2} n$ bound achieved by the shifted Zeno embedding is, in fact, optimal for history-independent data structures.

The main result will be the following lower bound:

Theorem 60. *Consider any history-independent list-labeling data structure. Let m be the size of the array and let $n = (1-\varepsilon)m$, where ε is at most some small positive constant and is at least $m^{-1/3}$. The expected cost to insert an element with a random rank in $\{1, 2, \ldots, n+1\}$ and then delete the element with rank $n+1$ is $\Omega(\varepsilon^{-1}\log^{3/2} n)$.*

Throughout the chapter, let c be a large constant, and assume that m is sufficiently large as a function of c. Let $m^{-1/3} \le \varepsilon \le 1/c$, and set $n = (1-\varepsilon)m$. We shall consider sequences of insertions/deletions, where each insertion is into an array of n elements and each deletion is from an array of $n+1$ elements.

To aid in the proof of Theorem 60, let us take a moment to establish several definitions and conventions. Let $J = \{2, 4, 8, \ldots, 2^{\lfloor \log m \rfloor - 2}\}$. For each $j \in [m]$, define a j-***block*** to be a block of j consecutive slots in the array, allowing for wrap-around (so there are m possible j-blocks).

Define the ***density*** of a j-block to be k/j, where k is the number of elements in the j-block. Call a j-block ***live*** if it has density at least $1 - c\varepsilon$, and dead otherwise. Note that this definition of density is slightly different from that used for recursive subproblems in the upper-bound section (Section 6.3) in that we define the density to be between 0 and 1—this difference will make the algebraic manipulation cleaner in several places.

For each $j \in J$, define the ***imbalance*** of a j-block to be $|\mu_1 - \mu_2|$, where μ_1 is the density of the first $j/2$ slots in the block, and μ_2 is the density of the final $j/2$ slots in the block. Define the ***adjusted imbalance*** $\Delta(x)$ of a j-block x to be the block's imbalance if the block is live, and 0 if the block is dead. Finally, define the ***boundary set*** $B(x)$ to be the set of up to three elements in positions $\{1, j/2, j\}$ of x.

For a given array configuration A, define $\Delta_j(A)$ to be the average adjusted imbalance across all j-blocks. Finally, define $\Delta_j = \mathbb{E}_{A\sim\mathcal{C}_{n,m}}[\Delta_j(A)]$.

We will split the proof of Theorem 60 into two key components. Section 7.1 proves the following combinatorial bound, which holds deterministically for any array configuration.

Proposition 61. *For any array-configuration A with $n = (1 - \varepsilon)m$ elements,*

$$\sum_{j\in J}(\Delta_j(A))^2 = O(\varepsilon^2).$$

Section 7.1 also uses Cauchy-Schwarz to arrive at the following corollary.

Corollary 62. *For any array-configuration A with $n = (1 - \varepsilon)m$ elements,*

$$\frac{1}{|J|}\sum_{j\in J}\Delta_j(A) = O(\varepsilon/\sqrt{\log n}).$$

Section 7.2 then gives a lower bound in terms of the Δ_i's on the expected cost that any history-independent data structure must incur.

Proposition 63. *Suppose $n = (1 - \varepsilon)m$, where $m^{-1/3} \le \varepsilon \le 1/c$ and c is some sufficiently large positive constant. Suppose we perform an insertion at a random rank $r \in \{0, \dots, n + 1\}$ and then delete the element with rank $n + 1$. The expected total cost of the insertion/deletion is at least*

$$\Omega\left(\sum_{j\in J}\frac{1}{\Delta_j + 1/j}\right).$$

Note that the expected cost in Proposition 63 is with respect to the randomness introduced by both the random rank r and the randomness in the history-independent data structure.

Intuitively, the above results tell us that any optimal history-independent data structure must behave a lot like the Zeno embedding. Indeed, Corollary 62 tells us that, no matter how we configure our array A, it is impossible to achieve imbalances that are consistently $\omega(\varepsilon/\sqrt{\log n})$—so, if our goal is to maximize the imbalances in our array, we can't hope to do any better than the Zeno embedding already does. Proposition 63 then tells us that small imbalances are necessarily expensive to maintain (and, in fact, the asymptotic relationship between cost and imbalance is the same as the one achieved by the Zeno embedding). Combining the propositions, we can prove the theorem as follows.

Proof of Theorem 60. By Corollary 62, we have

$$\frac{1}{|J|}\sum_{j\in J}\Delta_j = \mathbb{E}_{A\sim\mathcal{C}_{n,m}}\left[\frac{1}{|J|}\sum_{j\in J}\Delta_j(A)\right] \le O\left(\varepsilon/\sqrt{\log n}\right). \tag{7.1}$$

By Proposition 63, the the expected cost of the insertion/deletion is at least

$$\Omega\left(\sum_{j\in J}\frac{1}{\Delta_j + 1/j}\right). \tag{7.2}$$

If $\Delta_j \le 1/\sqrt{n}$ for any $j \ge \sqrt{n}$, then (7.2) becomes $\Omega(\sqrt{n}) \ge \Omega(\varepsilon^{-1}\log^{3/2} n)$ (since $\varepsilon \ge \Omega(n^{-1/3})$), and we are done. On the other hand, if $\Delta_j \ge 1/\sqrt{n}$ for all $j \ge \sqrt{n}$, then (7.2) is at least

$$\Omega\left(\sum_{j\in J, j\ge\sqrt{n}}\frac{1}{\Delta_j}\right). \tag{7.3}$$

By (7.1), we know that at least half of the Δ_j's in the above sum satisfy $\Delta_j = O(\varepsilon/\sqrt{\log n})$. Thus the expected cost comes out to at least

$$\Omega(\varepsilon^{-1}\log^{3/2} n).$$

∎

7.1 Proof of Proposition 61

Although Proposition 61 is a deterministic statement, we will prove it with a probabilistic argument.

For $j \in J$, define the ***children*** of a j-block to be the $j/2$-blocks consisting of the first and last $j/2$ slots of the block, respectively. Also, let $j^* = 2^{\lfloor \log m\rfloor - 2}$ be the largest element of J.

For a j-block x with density μ, define the ***potential*** $\phi(x)$ to be

$$\phi(x) = \begin{cases} 0 & \text{if } x \text{ is dead} \\ (\mu - (1 - c\varepsilon))^2 & \text{otherwise.} \end{cases}$$

For a random j-block x, one should think of $\phi(x)$ as measuring something similar to (but not quite equal to) the variance of μ. The key differences between what ϕ and variance measure is that (1) ϕ evaluates directly to 0 on any j-block x that is dead (i.e., has density less than $1 - c\varepsilon$), and (2) ϕ examines the square of the distance between μ and the death-threshold $1 - c\varepsilon$, rather than the square of the distance from μ to $\mathbb{E}[\mu] = 1 - \varepsilon$.

Note that $0 \le \phi(x) \le O(\varepsilon^2)$ deterministically. On the other hand, we will now see how to relate the expected potential $\phi(x)$ of a random 1-block to the quantity $\sum_{j \in J}(\Delta_j(A))^2$.

Lemma 64. *Let x be a random 1-block. Then*

$$\mathbb{E}[\phi(x)] = \Omega\left(\sum_{j \in J}(\Delta_j(A))^2\right).$$

Proof. Let x_0 be a random j^*-block, and for $i \in [\log j^*]$, let x_i be a random child of x_{i-1}. This means that each x_i is itself a random $2^{\log j^* - i}$-block, and that $x := x_{\log j^*}$ is a random 1-block. Define μ_i to be the density of x_i.

We will argue that

$$\mathbb{E}[\phi(x_i) - \phi(x_{i-1})] = \Omega\left((\Delta_{j^*/2^i}(A))^2\right). \tag{7.4}$$

This would imply that

$$\mathbb{E}[\phi(x)] = \mathbb{E}[\phi(x_0)] + \sum_i \mathbb{E}[\phi(x_i) - \phi(x_{i-1})] = \Omega\left(\sum_{j \in J}(\Delta_j(A))^2\right),$$

as desired.

For the rest of the proof, consider some ϕ_i and set $j = j^*/2^i$. We claim that with probability at least $1 - 1/c \geq 0.9$, x_{i-1} is live. Indeed, in expectation at most an ε fraction of the slots in x_{i-1} are free, so by Markov's inequality the probability that more than a $c\varepsilon$ fraction of the slots in x_{i-1} are free is at most $1/c \leq 0.1$.

If we condition that x_{i-1} is live, then its imbalance Δ satisfies $\mathbb{E}[\Delta] = \Theta(\Delta_{j^*/2^{i-1}}(A))$. Furthermore, if x_{i-1} is live, then we have that $\phi(x_i)$ is randomly one of

$$(\sqrt{\phi(x_{i-1})} + \Delta/2)^2$$

or

$$(\max\{0, \sqrt{\phi(x_{i-1})} - \Delta/2\})^2.$$

Note that the average of these is

$$0.5(\sqrt{\phi(x_{i-1})} + \Delta/2)^2 + 0.5(\max\{0, \sqrt{\phi(x_{i-1})} - \Delta/2\})^2.$$

If $0 < \sqrt{\phi(x_{i-1})} - \Delta/2$, this average is

$$\phi(x_{i-1}) + \Delta^2/4.$$

On the other hand, if $0 \geq \sqrt{\phi(x_{i-1})} - \Delta/2$, this average is

$$\begin{aligned} &0.5 \cdot \phi(x_{i-1}) + \Delta\sqrt{\phi(x_{i-1})}/2 + \Delta^2/8 \\ &\geq 1.5 \cdot \phi(x_{i-1}) + \Delta^2/8. \end{aligned}$$

Thus, in either case, this average is

$$\geq \phi(x_{i-1}) + \Delta^2/8.$$

It follows that

$$\begin{aligned}\mathbb{E}[\phi(x_i) \mid x_{i-1} \text{ live}] &\geq \mathbb{E}[\phi(x_{i-1}) \mid x_{i-1} \text{ live}] + \cdot\mathbb{E}[\Delta^2 \mid x_{i-1} \text{ live}]/8 \\ &\geq \mathbb{E}[\phi(x_{i-1}) \mid x_{i-1} \text{ live}] + \cdot\mathbb{E}[\Delta \mid x_{i-1} \text{ live}]^2/8 \\ &\geq \mathbb{E}[\phi(x_{i-1}) \mid x_{i-1} \text{ live}] + \Omega(\Delta_{j^*/2^{i-1}}(A)^2).\end{aligned}$$

On the other hand,

$$\begin{aligned}\mathbb{E}[\phi(x_i) \mid x_{i-1} \text{ not live}] &\geq 0 \\ &= \mathbb{E}[\phi(x_{i-1}) \mid x_{i-1} \text{ not live}].\end{aligned}$$

So we can conclude that

$$\begin{aligned}\mathbb{E}[\phi(x_i)] &\geq \mathbb{E}[\phi(x_{i-1})] + \Pr[x_{i-1} \text{ live}] \cdot \Omega(\Delta_{j^*/2^i}(A)^2) \\ &\geq \mathbb{E}[\phi(x_{i-1})] + 0.9 \cdot \Omega(\Delta_{j^*/2^{i-1}}(A)^2),\end{aligned}$$

hence (7.4). ∎

We can now prove Proposition 61.

Proof of Proposition 61. Let x be a random 1-block. Then by Lemma 64,

$$\mathbb{E}[\phi(x)] = \Omega\left(\sum_{j\in J} \Delta_j^2(A)\right).$$

On the other hand, $\phi(x) = O(\varepsilon^2)$ deterministically. Combined, these imply

$$\sum_{j\in J} \Delta_j(A)^2 = O(\varepsilon^2).$$

Proof of Corollary 62. Cauchy-Schwarz implies

$$\sum_{j\in J} \Delta_j(A)^2 \geq \left(\sum_{j\in J} \Delta_j(A)\right)^2 /|J| = \left(\sum_{j\in J} \Delta_j(A)\right)^2 /O(\log n).$$

Thus we have

$$\sum_{j\in J} \Delta_j(A) \leq O(\sqrt{\log n})\sqrt{\sum_{j\in J} \Delta_j(A)^2} \leq O\left(\varepsilon\sqrt{\log n}\right),$$

where the final inequality uses Proposition 61. Dividing by $|J| = \Theta(\log n)$, we have

$$\frac{1}{|J|}\sum_{j \in J} \Delta_j(A) = O(\varepsilon/\sqrt{\log n}).$$

∎

7.2 Proof of Proposition 63

In this section, we prove Proposition 63. All of the lemmas in this section assume an array of size m that initially contains n elements, where $n = (1-\varepsilon)m$.

We begin by establishing that, if we consider an element with random rank $t \in [n/2]$, and we examine the j-block beginning at that element, then there are several basic properties that hold with probability at least 0.9.

Lemma 65. *Let $j \in J$ and consider a random $t \in [n/2]$. Define x to be the j-block whose first position contains the current rank-t element. With probability at least* 0.9*, the following all hold:*

- *x is live;*
- *$|B(x)| = 3$;*
- *$\Delta(x) < c\Delta_j$,*

Proof. It suffices to show that each individual property holds with probability at least 0.97. Observe that x is chosen at random from one of $n/2 \geq m/3$ j-blocks. It therefore suffices to show that, if we define x' to be a uniformly random j-block, then each property holds with probability at least 0.99 for x'.

Note that x' contains at most εj free slots in expectation, so by Markov's inequality the probability that x' contains $\geq c\varepsilon j$ free slots is at most $1/c \leq 0.01$. Thus x' is live with probability at least 0.99.

We claim that $\mathbb{E}[3 - |B(x')|] = 3\varepsilon$. This is because the probability that a given slot is occupied is $1-\varepsilon$, so $\mathbb{E}[|B(x')|] = 3(1-\varepsilon) = 3 - 3\varepsilon$. Thus, $\mathbb{E}[3 - |B(x')|] = 3\varepsilon \leq 3/c$. Thus, by Markov's inequality we have $\Pr[3 - |B(x')| \geq 1] = 3/c \leq 0.01$. Thus $|B(x')| = 3$ with probability at least 0.99.

Finally, observe that $\mathbb{E}[\Delta(x')] = \Delta_j$, so by Markov's inequality we have $\Pr[\Delta(x') \geq c\Delta_j] \leq 1/c \leq 0.01$. Thus $\Delta(x') < c\Delta_j$ with probability at least 0.99. ∎

Call an insertion/deletion ***critical*** to a j-block x if: x is live when the operation is performed; and the operation leads to at least one of the elements in $B(x)$ being rearranged. The next lemma argues that, if we perform enough insertions/deletions inside a random j-block, then at least one of them will likely be critical.

Lemma 66. *Let $j \in J$, let $s \in [j/6, j/3]$, and consider a random $t \in [n/4-s, n/2-s]$. Define x to be the j-block whose first position contains the element with rank t. Suppose we perform $\lfloor cj\Delta_j \rfloor + 1$ insertion/deletion pairs, where each insertion adds a new element with rank $t+s$ and each deletion removes the highest-ranked element (i.e., the element with rank $n+1$). With probability at least 0.6, at least one of the insertions/deletions is critical to x.*

Proof. We know that, with probability at least 0.6, the properties in Lemma 65 hold for x both before the insertions/deletions are performed and after the insertions/deletions are performed. Suppose for contradiction that none of the insertions/deletions are critical to x.

Thus, we know that none of the operations rearrange any of the elements in $B(x)$. We additionally claim that none of the elements in $B(x)$ are deleted. This follows from the fact that we always delete the element of rank $n+1$, while the highest-ranked element in x is has rank less than $n+1$ for the following reason. The first element of x is at rank at most $n/2$, and x contains at most $j^* = 2^{\lfloor \log m \rfloor - 2} \leq \frac{n}{4(1-\varepsilon)}$ elements. So the last element of x has rank at most $\frac{n}{2} + \frac{n}{4(1-\varepsilon)}$, which is less than $n+1$ since $\varepsilon < 1/2$.

Additionally, we claim that all of the insertions go into the first $j/2$ slots of x. This is because otherwise there would be at most $s \leq j/3$ elements in the first $j/2$ slots, which means that there would be least $j/6$ empty slots, which contradicts the fact that x is live.

Thus, during the course of the insertions/deletions, the first $j/2$ slots in x gain $\lfloor cj\Delta_j \rfloor + 1$ elements, while the second $j/2$ slots of x remain stable in their number of elements. We know that $\Delta(x) < c\Delta_j$ both before and after the insertions/deletions are performed. So, over the course of the insertions/deletions, the density of the first $j/2$ slots in x changes by less than $2c\Delta_j$. This means that the number of elements in the first $j/2$ slots of x changes by at most $j/2 \cdot 2c\Delta_j = cj\Delta_j$, which contradictions the fact that the first $j/2$ slots in x gain $\lfloor cj\Delta_j \rfloor + 1$ elements. ∎

Using symmetry, we can reinterpret the previous lemma as a statement about a single insertion/deletion pair.

Lemma 67. *Let $j \in J$, let $s \in [j/6, j/3]$, and consider a random $t \in [n/4-s, n/2-s]$. Define x to be the j-block whose first position contains the element with rank t. Suppose we perform a single insertion/deletion pair, where the insertion adds a new element with rank $t+s$ and the deletion removes the current highest-ranked element (i.e., the element with rank $n+1$). With probability $\Omega\left(\frac{1}{\lfloor j\Delta_j \rfloor + 1}\right)$, at least one of the insertion/deletion is critical to x.*

Proof. By history independence, the probability distribution of array configurations is only dependent upon n, m, and r, and these quantities are the same after each insertion/deletion pair in Lemma 66. Thus, Lemma 66 immediately extends to each individual insertion/deletion pair. ∎

The previous lemma analyzes for a specific block x the probability that a specific insertion/deletion pair is critical to x. Notice, however, that a given insertion/deletion pair can be critical to many j-blocks simultaneously. Indeed, by applying Lemma 66 simultaneously for multiple different values of s, we can deduce a lower bound on the expected number of elements that are rearranged at distance $\Theta(j)$ (in rank) from the element currently being inserted.

Lemma 68. *Let $j \in J$. Suppose we perform an insertion at a random rank $r \in [n/4, n/2]$ and then we delete the highest-ranked element (i.e., the element with rank $n+1$). Let q_j be the number of elements that are rearranged by the insertion/deletion, and that have ranks r' satisfying $|r - r'| = \Theta(j)$ after the insertion. Then*

$$\mathbb{E}[q_j] = \Omega\left(\frac{1}{\Delta_j + 1/j}\right).$$

Proof. For $s \in [j/6, j/3]$, define x_s to be the j-block beginning with the element whose rank is $r - s$. Note that the sets $B(x_s)$ are disjoint across $s \in [j/6, j/3]$. Let B_s be the number of elements in $B(x_s)$ that are rearranged by the insertion/deletion; and let E_s be the event that both $B_s \geq 1$ and that x_s is live.

If x_s is live then the elements of B_s have ranks r' satisfying $|r - r'| = \Theta(j)$. Since the B_s's are disjoint, it follows that

$$\mathbb{E}[q_j] \geq \sum_{s \in [j/6, j/3]} \Pr[E_s].$$

For each $s \in [j/6, j/3]$, we have by Lemma 67 that

$$\Pr[E_s] = \Omega\left(\frac{1}{\lfloor j\Delta_j \rfloor + 1}\right).$$

Thus

$$\mathbb{E}[q_j] = \Omega\left(\frac{j}{\lfloor j\Delta_j \rfloor + 1}\right) = \Omega\left(\frac{1}{\Delta_j + 1/j}\right),$$

as desired. ∎

Finally, we can deduce a lower bound on the total number of elements that are rearranged by a random insertion/deletion.

Lemma 69. *Suppose we perform an insertion at a random rank $r \in [n/4, n/2]$ and then we delete the highest-ranked element. The expected total cost of the insertion/deletion is at least*

$$\Omega\left(\sum_{j \in J} \frac{1}{\Delta_j + 1/j}\right).$$

Proof. Let q be the number of elements that are rearranged by the insertion/deletion.

For each $j \in J$, define q_j as in Lemma 68. Each element that is rearranged by the insertion/deletion has a rank r' satisfying $|r' - r| = \Theta(j)$ for at most a constant number of $j \in J$. That is, each rearrangement is counted by at most $O(1)$ of the q_j's. Thus

$$q = \Omega\left(\sum_j q_j\right).$$

By Lemma 68, it follows that

$$\mathbb{E}[q] = \Omega\left(\sum_{j \in J} \frac{1}{\Delta_j + 1/j}\right).$$

▮

Lemma 69 considers an insertion with a random rank $r \in [n/4, n/2]$, but this trivially implies the same claim for a random rank $r \in \{0, \ldots, n\}$ (i.e., Proposition 63). Thus the section is complete.

Part III

Balls and Bins: When Greedy Allocation Fails and How to Fix It

Chapter 8

Introduction

Randomized balls-into-bins processes [268,358] serve as a useful abstraction for studying load-balancing problems, with applications such as scheduling, distributed systems, and data structures. The goal is to assign balls (e.g., tasks) to bins (e.g., machines) such that the balls are balanced as evenly as possible across the bins, where each individual ball may have only a few available random options for bins that it can be placed in.

It is well known that, if n balls are placed into n bins using the classical SINGLECHOICE rule, where each ball is placed independently in a uniformly random bin, then the maximum load is $\Theta(\log n/\log\log n)$ with probability $1 - 1/\operatorname{poly}(n)$.

The power of 2-choices. In a seminal 1994 paper, Azar, Broder, Karlin and Upfal [56] showed that under a seemingly minor modification, where for each ball *two* bins are chosen independently and uniformly at random, and the ball is placed *greedily* in the least loaded of the two bins, the maximum load reduces to $\log\log n + O(1)$ with high probability in n. In the decades since, this *power of 2-choices* paradigm has been extremely influential, with both theoretical (e.g., [111, 178, 187, 206, 298]) and empirical (e.g., [115, 144, 290, 292, 371]) applications, and with a large literature on generalizations; see e.g., [268, 358] for some excellent surveys.

The heavily-loaded case. Azar et al.'s result [56] prompted researchers to consider the ***heavily-loaded case***, where $m \gg n$ balls are inserted into n bins. The early techniques that were developed for the lightly-loaded setting (i.e., layered induction [56], witness trees [128,350], and differential-equation approaches [266,267]) struggled to deliver strong bounds in the heavily-loaded setting, and for several years the best known bound stood at $m/n + \log\log n + O(m/n)$ [128,350]. If we define the ***overload*** to be the amount by which the maximum load exceeds m/n, then this bound allows for an overload as large as $\log\log n + O(m/n)$—such a bound is useful if $m \approx n$, but when $m \gg n\log n$, the bound becomes worse even than the standard bound offered by SINGLECHOICE (i.e., an overload of $O(\sqrt{(m/n)\log n})$).

In a breakthrough result, Berenbrink, Czumaj, Steger and Vöcking [98] showed how to use Markov-chain techniques to obtain a much stronger bound of $\log \log n + O(1)$ on the overload, with probability $1 - 1/\operatorname{poly}(n)$. Thus, somewhat remarkably, the gap between the maximum and average loads in the heavily-loaded case *is the same* as in the lightly-loaded case, with high probability in n.

When $m \gg n$, the $O(\log \log n)$ overload bound does not, in general, extend to hold with probability $1 - 1/\operatorname{poly}(m)$ (i.e., w.h.p. in the number of *balls*). However, the known techniques can be used to achieve a quite strong (and, when $n = O(1)$, optimal) bound of $O(\log m)$ on the overload in this case.

The dynamic setting. In typical load-balancing and data-structures applications, however, the items can be both inserted and deleted dynamically over time. This can be captured by allowing for balls to also be inserted/deleted over time.

Whereas in the insertion-only setting, m is set to be the total number of insertions, in the dynamic setting, m is set to be an *upper bound* on the number of balls that are present at any given moment (and the sequence of insertions/deletions may be infinite). The objective is to minimize the *overload*, which is now defined as the amount by which the maximum load exceeds m/n at any given moment.[1]

Azar et al. [56] considered the insertion/deletion model with $m = n$ and with *random deletions*: that is, n balls are inserted initially, and then there is an infinite sequence of alternating insertions/deletions, where each deletion removes a *random* ball. They showed that, at any given moment, the GREEDY strategy achieves a maximum load of $\log \log n + O(1)$, with high probability in n.

Subsequent work has considered the more general setting where the insertions/deletions are determined by an ***oblivious-adversary*** (i.e., an adversary that does not know the random choices of the algorithm), and where the only constraint on the adversary is that the number of balls in the system can never exceed m. Using the witness tree technique, first introduced by [129], Cole et al. [128] analyzed the reinsertion/deletion model with $m = n$, and established that the GREEDY strategy guarantees a maximum load of $O(\log \log n)$ with high probability in n. Later, Vöcking [350] improved this to $\log \log n + O(1)$, which remarkably, matches the bound in the non-dynamic (insertion-only) case up to an additive $O(1)$ term.

What about the dynamic heavily-loaded case? For more than two decades, it has remained an open question what the optimal bounds are in the *heavily-loaded case* if we wish to support both insertions and deletions performed by an oblivious adversary. Besides obvious theoretical interest, the question also arises naturally in practice, both in scheduling and data-structural applications.

[1]It is tempting to define the overload to be the amount by which the maximum load exceeds $m(t)/n$, where $m(t)$ is the number of balls present at time t. However, the following (folklore) example demonstrates the flaw with such a definition: Suppose we insert m balls (using an arbitrary insertion strategy), and then we delete a random $m/2$ of those balls. Since the $m/2$ deletions are random, even if the system was perfectly balanced after the initial m insertions, the bin loads will typically be $m/2n \pm \sqrt{m/2n}$, and the maximum load will be $m(t)/n + \tilde{\Theta}(\sqrt{m/n})$, which is no better than the bound trivially achieved by SINGLECHOICE.

Past works by Cole et al. [128] and then by Vöcking [350,351] showed that Greedy has overload $\log\log n + O(m/n)$ with high probability in n.[2] But this bound is already worse for $m \gg n \log n$ than the $O(\sqrt{(m/n)\log n})$ overload bound for SingleChoice (which also holds in the dynamic setting).

However, it is widely believed that Greedy should also achieve similar bounds in the dynamic heavily-loaded case as in the non-dynamic heavily-loaded case (i.e., an overload of $O(\log\log n)$ and $O(\log m)$, w.h.p. in n and m, respectively). The current limitation would seem to be a technical one: the witness-tree techniques that allow for us to analyze dynamic games with oblivious adversaries [128, 351] are incompatible with the techniques (i.e., Markov-chain [98] and potential-function [248, 308, 340] arguments) that achieve strong bounds in the heavily-loaded case.

As we shall, see, the above intuition is actually wrong. The reason that researchers have been unable to analyze Greedy allocation is that the bounds that one might hope for fail. Nonetheless, with the help of new non-greedy strategies, it will be possible to salvage the situation.

Two perspectives: scheduling vs. data structures. It will be helpful to split the dynamic case into two models. The first model is the ***insertion/deletion*** model in which each insertion involves a new ball with independent random bin choices. This model is natural from the perspective of scheduling problems, where insertions represent the addition of new jobs to the system. The second model is the ***reinsertion/deletion*** model in which a ball can be *reinserted* after being deleted, and has the same two random bin choices each time it is reinserted. This model is natural from the perspective of data-structural problems, where the balls represent elements of a data structure, and the same element can be inserted/deleted/reinserted many times.

These two models may seem quite similar at first glance, and indeed in past work, the results for the two models have always been the same. One of the surprising takeaways of our results, however, will be that, in the heavily-loaded regime, the two models actually lead to remarkably different conclusions and require very different techniques from one another.

8.1 Chapter 9. The Scheduling Perspective

We begin by considering the insertion/deletion model, that is, an oblivious adversary performs an arbitrary sequence of insertions/deletions subject only to the constraint that no more than m balls are present at a time.

A lower bound for Greedy. We show that the Greedy strategy *does not* offer strong bounds in the dynamic heavily-loaded setting. In particular, already for $n = 4$ bins, there exists an oblivious sequence of insertions/deletions after which there is a

[2]Credit should also be given to Woelfel [369] for fixing an error in the original arguments.

maximum load of

$$m/n + \Omega(\sqrt{m})$$

with probability $\Omega(1)$. In other words, the GREEDY strategy is no better than SINGLECHOICE in this setting!

Our result represents a remarkable departure from the lightly-loaded $m = n$ case, where GREEDY achieves an optimal bound of $O(\log \log n)$ (even in the *re*insertion/deletion model). The result also offers an explanation for why all previous attempts [128, 351] to analyze GREEDY for large m have yielded only relatively weak bounds.

The high-level intuition behind our lower bound is as follows. Using GREEDY, if some bin i contains far fewer balls than the other bins, then there will be a contiguous time window during which all of the insertions are maximally biased towards bin i. But this means that, later on, the adversary can perform a sequence of deletions in which the balls being *deleted* exhibit a strong bias towards being from bin i. In other words, the biases that GREEDY exhibits during insertions can be thrown back at it by future deletions.

The MODULATEDGREEDY algorithm. Of course, the above phenomenon is not isolated to the GREEDY strategy. Any strategy that exhibits biases between bins is at risk of having those biases thrown back at it via future deletions. This raises a natural question: is it possible for *any 2-choice allocation strategy* to beat the bounds trivially achieved in the single-choice model?

Our second result is a new algorithm called MODULATEDGREEDY, in the insertion/deletion model, that at any time, with high probability in m, achieves a maximum load of

$$m/n + O(\log m).$$

This bound is optimal for any strategy that achieves high-probability bounds in m .

Given the choice between two bins i and j, the MODULATEDGREEDY algorithm chooses between the bins probabilistically, based on how their loads compare. It carefully modulates its biases between bins so that the adversary is unable to find any non-trivial correlations between how balls are inserted.

A returning hero: history independence. The analysis of MODULATEDGREEDY draws a surprising connection to the techniques that we developed in Part II for the dynamic sorting problem. The analysis *couples* MODULATEDGREEDY to a history-independent process that we call the stone game. The history-independence of the stone game makes the game trivial to analyze, no matter what sequence of insertions/deletions are performed. The fact that MODULATEDGREEDY can be coupled to the game then allows for us to (with almost no work!) recover tight bounds for MODULATEDGREEDY.

In fact, what is really going on here is that, although the assignment of balls to bins in MODULATEDGREEDY is not history independent, the actual histogram of how many balls are in each bin is (with high probability). This is in stark contrast to what happens for Greedy, where the state of the system is a complicated function of the past insertion/deletion history.

It is worth emphasizing that the role of history independence here is not just analytical (although it certainly makes the analysis much simpler). The introduction of history independence into ModulatedGreedy's algorithmic structure is precisely what allows for us to bypass the lower bound that holds for Greedy. In other words, history independence is again serving as an *algorithmic tool* for obtaining better algorithms.

Generalizations. Our analysis of ModulatedGreedy extends to support a number of generalizations and applications. This includes a tight bound of $m/n + O(\beta^{-1}\log m)$ for the $(1+\beta)$-*choice* version of the game [308], where a $(1-\beta)$-fraction of the balls are inserted using SingleChoice and only a β-fraction of the balls get two choices; a bound of $m/n + \operatorname{polylog} m$ for the dynamic balls-and-bins game on an undirected well-connected regular graphs [59, 220]; and a bound of $m/n + O(\log M)$ for the setting in which m is permitted to increase over time, subject only to the constraint that $m \leq M$. In all of these settings, the previous states of the art were restricted to the insertion-only model.

8.2 Chapter 10. The Data-Structural Perspective

Next, we turn our attention to the reinsertion/deletion model. That is, the adversary can perform an arbitrary sequence of insertions, deletions, and reinsertions (as long as the ball being reinserted is not currently present) subject only to the constraint that no more than m balls are present at a time.

A (much) stronger lower bound. We begin by establishing a strong impossibility result. Consider any 2-choice bin-allocation strategy that is *oblivious* to the specific *identities* of balls (i.e., when a ball is inserted, all that the strategy gets to see is the pair i, j of bins that the ball is assigned to). We show that, against any such strategy, it is possible for an oblivious adversary on $n = 4$ bins to force a maximum load of $m/4 + \operatorname{poly}(m)$ at some point in the first $\operatorname{poly}(m)$ insertions/deletions, with high probability in m.

This result reveals a fundamental (and perhaps unexpected) gap between the insertion/deletion model and the reinsertion/deletion model. In particular, in the lightly-loaded setting with deletions where $m \leq n$, both models yield the same $O(\log\log n)$ bounds even for infinite sequences of reinsertions/deletions [128, 351]. But, in the heavily-loaded setting, the cyclic dependencies that are introduced by reinsertions (i.e., a ball x being reinserted is being placed into a system whose state has *already* been affected by x's bin choices in the past) end up being lethal to any ID-oblivious allocation strategy.

Iceberg hashing: a strategy for the medium-loaded case. In most data-structural settings, it is the *very* heavy case that we are interested in, but rather the parameter regime where the number m of balls satisfies $m = hn$ for some $h \in \omega(n) \cap o(n \log n)$. Here, SingleChoice offers a bound of $h + \Theta(\sqrt{h}\sqrt{\log n})$ on the maximum load (with high probability in n). However, since $\log n \gg h$, this bound is actually $\omega(h)$! We can get a slightly stronger bound of $O(h + \log\log n)$ using Greedy with two choices [350, 351, 369], but this bound is still giving up a constant factor on the h term. It has remained an open question whether, using $O(1)$ choices, one can achieve a bound of $(1 + o(1))h + O(\log\log n)$ on the maximum load.

We answer this question in the affirmative with a new 3-choice scheme. Our scheme, which we call Iceberg, achieves a maximum load of

$$h + O(\sqrt{h \log h}) + \log\log n + O(1)$$

with high probability in n, in the reinsertion/deletion model.

Unlike our results for the non-reinsertion case, which are probably tight, it remains an open question whether the bounds achieved by Iceberg hashing are optimal. Nonetheless, as we shall see in Part V, the bounds achieved by Iceberg hashing are already strong enough for many data-structural applications, enabling a new type of data-structural abstraction that we call the *tiny pointer*.

The rest of the chapter proceeds in two sections. Section 8.3 presents preliminaries and definitions. Then, Section 8.4 gives a more in-depth discussion of related work.

8.3 Preliminaries

In the ***dynamic 2-choice allocation problem***, an oblivious adversary performs a sequence of ball insertions and deletions subject to the constraint that the number of balls in the system can never exceed m. Whenever a ball x is inserted, a uniformly random pair $h(x) = (h_1(x), h_2(x)) \in [n] \times [n]$ of distinct bins is selected, and the ***insertion strategy*** must choose which of the bins $h_1(x)$ or $h_2(x)$ the ball will be placed in. The pair $h(x)$ is sometimes referred to as the ***hash*** of the ball x.

There are two models that we will consider for insertions and deletions. In the ***insertion/deletion model***, each insertion Insert(x) places a new ball x into the system that has never been present before. In the ***reinsertion/deletion model***, each insertion Insert(x) places a ball x into the system that is not *currently* present, but that may have been present in the past (each time x is inserted, its bin pair $h(x)$ stays the same). In both models, the Delete(x) operation selects a ball x that is currently present and removes it.

We are interested in bounding the maximum ***load*** (i.e., the number of balls) of any bin. Our algorithms will offer guarantees with high probability (w.h.p.) in m, meaning that the failure probability is $1/\operatorname{poly}(m)$ for a polynomial of our choice. Two basic insertion strategies that we will discuss frequently are Greedy, which always selects the least full of the bins $h_1(x), h_2(x)$, and SINGLECHOICE, which always selects bin $h_1(x)$.

In our lower bound for the reinsertion/deletion model (Section 10.1), we will study the class of ***ID-oblivious*** insertion strategies—such a strategy makes each insertion decision based on the hash $h(x)$ of the ball being inserted, rather than based on the specific identity x of the ball. Formally, an ID-oblivious strategy is one that can be implemented with operations INSERT$(h_1(x), h_2(x))$ (indicating the pair of bins for the ball being inserted) and DELETE(r) (indicating a deletion of the r-th-most-recently-inserted ball of those present).

Finally, although $h(x) = (h_1(x), h_2(x))$ is a uniformly random pair of *distinct* bins, any strategy in the insertion/deletion model can choose to view $h(x)$ as a pair of *independent bins* by artificially resetting $h_2(x) = h_1(x)$ with probability $1/n$. The strategies that we design will assume (without loss of generality) that they are given a uniformly random pair of (not necessarily distinct) bins for each insertion.

8.4 Other Related Work

Beyond research on the heavily-loaded and dynamic settings, there has been a large body of work on other ways to extend the 2-choice allocation framework—because the literature on this subject is so extensive, we give only a brief overview here. These extensions have included work on restricted classes of insertion strategies (e.g., $(1+\beta)$-choice strategies [308], thinning strategies [177, 246–248], strategies with limited information [247], etc.), on balls with nonuniform sizes [99, 308, 339, 340], on parallel settings in which balls arrive in batches [97,100,101,237,333], on settings in which bins correspond to vertices on a graph [59, 220], on settings where balls can be relocated after insertion [37, 85], etc. Another notable extension is Vöcking's asymmetric d-choice paradigm [351] which, in the lightly-loaded setting, chooses between d bins on each insertion to achieve a maximum load of $O((\log\log n)/d)$.

Another line of work, related to the current work on the dynamic setting, is on queuing models [112, 113, 172, 251, 252, 267, 275, 352], where insertions and deletions are *stochastic*. Many of these focus on the so-called supermarket model, introduced by [267, 352], in which customers (i.e., balls) arrive in a Poisson stream of rate λn, $\lambda < 1$, and are processed within each queue (i.e., bin) in FIFO order, where each customer requires processing time that is exponentially distributed with mean 1. In the case where λ is allowed to go to 1 (see, e.g., [113, 172]), the number of balls in the system can become $\omega(n)$ (this is analogous to the heavy case in standard balls and bins). However, because insertions/deletions are assumed to be stochastic, the analyses (and the flavors of the results) take a very different form than those here (where deletions are performed by an oblivious adversary, and the number of balls in the system is deterministically bounded by a parameter m).

Finally, there are a number of works [77, 86, 91, 140, 231, 232, 246, 265] that study load-balancing problems in which slightly non-greedy behavior can out-perform more greedy approaches (either because the less-greedy approach relies less on stale information [140, 246, 265], or because the less-greedy approach benefits from randomization [77, 86, 91, 231, 232]). Our work reveals that this same theme appears somewhat unexpectedly even in the classical setting of power-of-two-choice with deletions (but, of course, for different reasons).

Chapter 9

The Scheduling Perspective

In this chapter, we consider the insertion/deletion model. We show that the GREEDY strategy does not, in general, achieve strong bounds, but that an alternative strategy (which we call MODULATEDGREEDY) does.

A lower bound for greedy with deletions. Recall that the trivial SINGLECHOICE strategy achieves an overload of $O(\sqrt{(m/n)\log n})$ (w.h.p. in m). A natural question is whether GREEDY does any better. In Section 9.1, we show that, at least in some parameter regimes, it does not:

Theorem 70. *Consider the insertion/deletion model on $n = 4$ bins, with the restriction that at most m balls can be present at any time, and suppose that insertions are implemented using* GREEDY. *There exists an oblivious sequence of* $\text{poly}(m)$ *insertions/deletions such that, after the sequence is complete, we have with probability* $\Omega(1)$ *that some bin contains $m/4 + \Omega(\sqrt{m})$ balls.*

For ease of exposition, and to keep the main ideas as clear as possible, we focus our lower bound on $n = 4$ bins. For general n and m, the same construction naturally extends to give an $m/n + \Omega(\sqrt{m}/\text{poly}(n))$ lower bound on the maximum load.

A tight upper bound via MODULATEDGREEDY. Next we introduce the MODULATEDGREEDY strategy, a two-choice strategy that significantly outperforms GREEDY in the dynamic setting. To describe the main ideas as clearly as possible, break the presentation of the algorithm into two parts. In Section 9.2 we consider a simpler version of MODULATEDGREEDY that guarantees the $m/n + O(\log m)$ bound for insertion/deletion sequences of $\text{poly}(m)$ length:

Theorem 71. *Let $m \geq n$. Consider the insertion/deletion model with n bins and an upper bound of at most m balls present at a time. Consider a sequence of* $\text{poly}(m)$ *insertions/deletions, where insertions are implemented using* MODULATEDGREEDY. *With high probability in m,* MODULATEDGREEDY *does not halt during any of the insertions/deletions, and no bin ever has load more than $m/n + O(\log m)$.*

Then, in Section 9.3, we extend the algorithm to support unbounded request sequences and to allow m to increase over time. Finally, in the same section, we show that the same techniques can be applied to the $(1+\beta)$-choice and the graphical 2-choice processes in order to obtain dynamic results in those settings.

9.1 A Lower Bound for Greedy with Deletions

In this section, we prove Theorem 70. To motivate our techniques, we being in Subsection 9.1.1 by proving a simpler (but already surprising) lower bound of $m/4 + \Omega(m^{1/4})$. Then, in Section 9.1.2, we build on these ideas to prove Theorem 70.

9.1.1 A Simpler $\Omega(m^{1/4})$ Bound

We begin by describing a construction with the property that, if we ever reach a state where one of the bins (say, bin 1) contains significantly fewer balls (say, k fewer balls) than the other bins, then we can subsequently reach a state in which (with probability $\Omega(1)$) some bin contains at least $m/4+\Omega(\sqrt{k})$ balls. As we shall see later in the subsection, this can be used to directly obtain the $m/4 + \Omega(m^{1/4})$ bound.

Proposition 72 (Gap to overload)**.** *Consider the* Greedy *algorithm on 4 bins, on instances where at most m balls can be present at a time. Suppose we begin in a state that contains at most $m-k$ balls, and where bin 1 contains $k+1$ fewer balls than each of bins 2, 3, 4. Then there is an oblivious sequence of $O(m)$ insertions/deletions such that, after the sequence is complete, we have the following property with probability $\Omega(1)$: some bin contains $m/4+\Omega(\sqrt{k})$ balls.*

Proof. Let X_0 denote the initial state of the game. Consider the sequence with the following three steps.

1. Insert k balls $x_1, x_2, \ldots, x_k$ to get to a state X_1.
2. Then insert $m-j$ balls $y_1, y_2, \ldots, y_{m-j}$, where j is the number of balls in state X_1—this brings us to a state X_2 with m balls in total.
3. Finally, delete the balls $x_1, x_2, \ldots, x_k$, and insert new balls $z_1, z_2, \ldots, z_k$ to reach a state X_3.

We claim that, for at least one of the two states X_2 and X_3, we have with probability $\Omega(1)$ that some bin contains $m/4+\Omega(\sqrt{k})$ balls. (This means that, if we terminate the sequence of operations randomly at one of X_2 or X_3, then after the sequence is complete, we have with probability $\Omega(1)$ that some bin contains $m/4+\Omega(\sqrt{k})$ balls.)

During the insertions of $x_1, x_2, \ldots, x_k$, we are always in a state where bin 1 contains fewer balls than bins 2, 3, 4. Thus, each insertion x_i will go into bin 1 if and only if $1 \in \{h_1(x_i), h_2(x_i)\}$ (this is where we are exploiting that the Greedy algorithm is too aggressive). The number A of balls $x_1, x_2, \ldots, x_k$ that are placed in bin 1 is therefore given by

$$A = |\{i \mid 1 \in \{h_1(x_i), h_2(x_i)\}\}|.$$

Let $\mu = \mathbb{E}[A]$. As A is a binomial random variable with mean $\mu = \Theta(k)$, with probability $\Omega(1)$ we have

$$A \geq \mu + \Omega(\sqrt{k}).$$

Now consider the number B of balls $z_1, z_2, \ldots, z_k$ that are placed into bin 1. We deterministically have that

$$B \leq |\{i \mid 1 \in \{h_1(z_i), h_2(z_i)\}\}|. \tag{9.1}$$

Since the right side of (9.1) is a binomial random variable with mean μ, we have with probability $\Omega(1)$ that

$$B \leq \mu.$$

Moreover, since A and B are independent, the above bounds on A and B hold simultaneously with probability $\Omega(1)$.

Finally, let us consider the number of balls in bins $2, 3, 4$ once we reach state X_3. Assume that state X_2 has maximum load $m/4 + o(\sqrt{k})$, otherwise we are already done. Then, since X_2 contains m balls in total, bins $2, 3, 4$ must contain a total of at least $3m/4 - o(\sqrt{k})$ balls. By step 3 of the input sequence above, it follows that, in state X_3, the total number of balls in bins $2, 3, 4$ is at least

$$3m/4 - o(\sqrt{k}) - (k - A) + (k - B).$$

Conditioning on the event above, and plugging in our bounds for A and B, we see that (with probability $\Omega(1)$) this is at least $3m/4 + \Omega(\sqrt{k})$. Thus, at least one of bins $2, 3, 4$ must contain $m/4 + \Omega(\sqrt{k})$ balls, as desired. ∎

The lower bound. Using Proposition 72, the claimed lower bound follows quite easily. Consider the following input sequence, starting from an empty system. (1) Insert m balls into the system; (2) delete each ball independently and randomly with probability $1/2$; and (3) apply the sequence in Proposition 72 with $k = \sqrt{m}$.

As the deletions are random in step (2), the precondition for Proposition 72 (i.e., the least loaded bin contain at least $k = \sqrt{m}$ fewer balls than every other bins) holds with probability $\Omega(1)$. So by Proposition 72, we can achieve $m/4 + \Omega(\sqrt{k}) = m/4 + \Omega(m^{1/4})$ balls in some bin, with probability $\Omega(1)$.

General n. For n bins, where n is arbitrary, the same approach gives a lower bound of

$$m/n + \Omega(m^{1/4}/\sqrt{n^3 \log n}). \tag{9.2}$$

with probability $\Omega(1)$.

In particular, Proposition 72 can be directly modified, in this setting, to achieve an overload of $\Omega(k^{1/2}/n)$: instead of using k balls in each of steps 1 and 3, use $kn/100$ balls; then by the same argument as in the lemma, we have $A - B = \Omega(k^{1/2})$ with constant probability; this means that bin 1 is under-loaded by at least $\Omega(k^{1/2})$, and thus that some other bin is over-loaded by at least $\Omega(k^{1/2}/n)$.

To achieve (9.2) using the modified Proposition 72, we just need to cause the smallest load to be $k = \Theta(\sqrt{m/(n \log n)})$ smaller than the other loads—this can again be achieved again by performing m insertions and then deleting each ball independently with probability 1/2. After the m insertions, every bin will have essentially the same load ($\pm O(\log n)$ w.h.p. in n). Conditioning on the loads, the number of balls deleted from each bin is a Gaussian with standard deviation $\sigma = \Theta(\sqrt{m/n})$ (and the Gaussians are independent between bins). By standard estimates on order statistics, the difference in loads between the least loaded and the second least loaded bins is roughly the difference between the $1/n$-th and $2/n$-th percentile of the distribution, see e.g., [317], which in expectation is $\Theta(\sigma/\sqrt{\log n})$ for the Gaussian $N(0, \sigma^2)$—hence an imbalance of $k = \Theta(\sqrt{m/(n \log n)})$.

9.1.2 The Stronger $\Omega(m^{1/2})$ Lower Bound

We now show how to achieve the stronger bound of $m/4 + \Omega(\sqrt{m})$ balls in some bin. Given Proposition 72, to prove Theorem 70 it suffices to show how to achieve a gap of $k = \Omega(m)$ between bin 1 and bins $2, 3, 4$. This is accomplished in the following proposition.

Proposition 73. *Consider the* Greedy *algorithm on 4 bins, with the restriction that at most m balls can be present at a time. There exists an oblivious sequence of* poly(m) *insertions/deletions such that, after the sequence is complete, we have the following property with probability $\Omega(1)$: Bin 1 contains $\Omega(m)$ fewer balls than each of bins $2, 3, 4$.*

The rest of the section is focused on the proof of Proposition 73.

Let $0 < \varepsilon_1, \varepsilon_2, \varepsilon_3 < 1$ be constants, where ε_2 is sufficiently small as a function of ε_1, and let ε_3 is sufficiently small as a function of ε_2. Sometimes we will write $\varepsilon_1, \varepsilon_2, \varepsilon_3$ inside the $O(\cdot)$ notation, to make the dependence on them explicit, while hiding fixed constants that do not depend on $\varepsilon_1, \varepsilon_2, \varepsilon_3$.

Some basic gadgets

We begin with a basic technical lemma establishing that Greedy has a tendency of eliminating imbalances over time. For brevity (and since the proof follows from standard arguments), we defer the proof of Lemma 74 to Appendix 9.A.

Lemma 74. *Consider the* Greedy *algorithm on 4 bins, and fix an arbitrary initial state in which the bins have loads within $\varepsilon_2 m$ of each other. If $\varepsilon_1 m$ insertions are performed, then after the sequence is complete, all of the bins have loads within $O(\log m)$ of each other with high probability in m. Furthermore, with high probability in m, there is a point in time prior to the final insertion at which all of the bins have equal loads.*

Using Lemma 74, we now construct a simple strategy for forcing Greedy to add a ball to a uniformly random bin.

Lemma 75 (Uniform ball placement gadget)**.** *Consider the* Greedy *algorithm on 4 bins, and fix an arbitrary initial state in which the bins have loads within $\varepsilon_2 m$ of each other. Suppose we insert balls $x_1, \ldots, x_{\varepsilon_1 m}$, and then we delete balls $x_1, \ldots, x_{\varepsilon_1 m - 1}$ (all except the last insertion). With high probability in m, this is equivalent to placing the ball $x_{\varepsilon_1 m}$ uniformly at random into one of the bins $1, 2, 3, 4$.*

Proof. We have by Lemma 74 that, with high probability in m, there is some insertion x_i, $i \in [\varepsilon_1 m - 1]$, after which the bins have equal loads. It follows that, from the perspectives of insertions $x_{i+1}, \ldots, x_{\varepsilon_1 m}$, the four bins are symmetric. Thus the last insertion $x_{\varepsilon_1 m}$ is equally likely to be placed into each of the bins, which establishes the lemma. ∎

Lemma 75 allows for us to place a ball into a random bin, but we can only do this $O(m)$ times before there are too many balls ($> m$) in the system. But for the purposes of Proposition 73, we will need to do this $\Omega(m^2)$ times. Our next lemma provides a mechanism for reducing the number of balls that are present while having only a small effect on the relative loads of the bins.

Lemma 76 (Almost equal load reduction gadget)**.** *Consider the* Greedy *algorithm on 4 bins, and fix an arbitrary initial state in which the bins $1, 2, 3, 4$ have loads $\ell_1, \ell_2, \ell_3, \ell_4$ within $\varepsilon_2 m$ of each other. We can construct an oblivous sequence of $O(\varepsilon_1 m)$ insertions/deletions such that, after this sequence, the total number of balls in the system is at most $\varepsilon_1 m$; and such that, with high probability in m, the new bin loads ℓ'_i for $i \in [4]$ satisfy*

$$\ell'_i = \ell_i - r + Y^{(i)}, \tag{9.3}$$

where $r \in \mathbb{N}$, $|Y^{(i)}| \leq O(\log m)$, and $\mathbb{E}[Y^{(i)}] = 0$.

Proof. Let us begin by describing a sequence of $O(\varepsilon_1 m)$ insertions/deletions after which (1) the total number of balls in the system is at most $\varepsilon_1 m$; and (2) the new loads ℓ'_i of the bins satisfy (w.h.p. in m)

$$\ell'_i = r - \ell_i + Y^{(i)}, \tag{9.4}$$

where $r \in \mathbb{N}$, $|Y^{(i)}| \leq O(\log m)$, and $\mathbb{E}[Y^{(i)}] = 0$. (Note that (9.4) is the same as (9.3) but with r and ℓ_i flipped).

The lemma will then follow by applying the above construction twice. That is, first we obtain $\ell'_1, \ell'_2, \ell'_3, \ell_4$ satisfying (9.4), and then apply it again to obtain $\ell''_1, \ell''_2, \ell''_3, \ell''_4$ satisfying

$$\ell_i{}'' = r' - \ell'_i + Y'^{(i)}, \tag{9.5}$$

where $r' \in \mathbb{N}$, $|Y'^{(i)}| \leq O(\log m)$, and $\mathbb{E}[Y'^{(i)}] = 0$. Chaining together (9.4) and (9.5), we get relationship between $\ell_1, \ell_2, \ell_3, \ell_3$ and $\ell''_1, \ell''_2, \ell''_3, \ell''_4$ as desired by (9.3).

Our construction for achieving (9.4) is very simple: we perform $\varepsilon_1 m$ insertions $x_1, x_2, \ldots, x_{\varepsilon_1 m}$, and then we delete all of the *other elements* besides $x_1, x_2, \ldots, x_{\varepsilon_1 m}$. Let ℓ_i be the load of bin i before these insertions/deletions, let q_i be the load of bin i

after the insertions are completed (but the deletions have not yet begun), and let ℓ'_i be the load of bin i after the deletions have completed.

By Lemma 76, the quantities q_1, q_2, q_3, q_4 are within $O(\log m)$ of each other (w.h.p. in m). Moreover, w.h.p. in m, there is some point during the insertions at which all of the bins have equal loads—if we condition on this, then we have $\mathbb{E}[q_1] = \mathbb{E}[q_2] = \mathbb{E}[q_3] = \mathbb{E}[q_4]$ by symmetry. Defining $Y^{(i)} = q_i - \mathbb{E}[q_i]$, we have $|Y^{(i)}| \le O(\log m)$, and $\mathbb{E}[Y^{(i)}] = 0$.

As $\ell'_i = q_i - \ell_i$, we have $\ell'_i = \mathbb{E}[q_i] + Y^{(i)} - \ell_i$. Setting $r = \mathbb{E}[q_i]$, it follows that (9.4) holds w.h.p. in m. ∎

Applying the gadgets

We say that an application of Lemma 75 or of Lemma 76 *fails* if either: the precondition of $\ell_1, \ell_2, \ell_2, \ell_4$ being within $\varepsilon_2 m$ of each other fails (this is a *precondition failure)*; or the high-probability guarantee offered by the lemma fails (this is a *probabilistic failure*).

We now describe the sequence of insertions/deletions that we use to achieve Proposition 73. We perform $\varepsilon_3 m$ phases, where phase $a \in [\varepsilon_3 m]$ proceeds as follows:

- Apply Lemma 75 m times, one after another. For $b \in [m]$, use $Z_{m\cdot(a-1)+b}$ to denote the bin that the b-th application of the lemma adds a ball to. If the lemma fails, then for the sake of analysis, we redefine $Z_{m\cdot(a-1)+b}$ to be uniformly random in [4]. This ensures that, regardless of whether the lemma fails, the Z_i's are independently and uniformly random in [4].

- Apply Lemma 76 once to reduce the loads almost equally. Let $Y_a^{(1)}, Y_a^{(2)}, Y_a^{(3)}, Y_a^{(4)}$ denote the outcomes of $Y^{(1)}, Y^{(2)}, Y^{(3)}, Y^{(4)}$ in that application of the lemma. If the lemma fails, then for the sake of analysis, we redefine $Y_a^{(1)}, Y_a^{(2)}, Y_a^{(3)}, Y_a^{(4)}$ to be 0.

To analyze the sequence of insertions/deletions, we first argue that the $Y_i^{(s)}$s have a negligible effect on the loads of the bins at any given moment.

Lemma 77. *Let $s \in [4]$ and $k \in [\varepsilon_3 m]$. Then w.h.p. in m, it holds that for each k, $|\sum_{a=1}^{k} Y_a^{(s)}| \le \tilde{O}(\sqrt{m})$, where $\tilde{O}(\cdot)$ hides polylogarithmic factors in m.*

Proof. The sequence of partial sums $P_r = \sum_{a=1}^{r} Y_a^{(s)}$ for $r = 0, \ldots, k$ forms a martingale satisfying $|P_r - P_{r-1}| = O(\log m)$ deterministically for each $r \in [k]$. The lemma follows from Azuma's inequality. ∎

Next we consider the effect of the $\varepsilon_3 m^2$ insertions Z_i over the εm phases, and show that with probability at least $1 - \varepsilon_2$, there is no point in time at which the Z_i's cause an imbalance of more $\varepsilon_2 m/2$.

For $k \in [\varepsilon_3 m^2]$ and $s \in [4]$, let

$$S(k, s) = |\{i \in [k] \mid Z_i = s\}|$$

denote the number insertions in bin s during the first k applications of Lemma 75.

Lemma 78. *Let $s \in [4]$ and $\varepsilon_2 = (2\varepsilon_3)^{1/3}$. With probability at least $1 - \varepsilon_2$, it holds (simultaneously) for all $k \in [\varepsilon_3 m^2]$ that*

$$|S(k, s) - k/4| \leq \varepsilon_2 m/2.$$

Proof. As Z_i is equal to s independently with probability $1/4$, the sequence $S(k, s) - k/4$ for $k = 0, 1, \dots, \varepsilon_3 m^2$ forms a martingale with increments $\{-1/4, 3/4\}$ (and hence variance at most 1). By the maximal inequality for martingales, for any $\lambda > 0$,

$$\Pr\left[\max_{k\in[\varepsilon_3 m^2]} |S(k, s)| > \lambda\right] \leq 2\frac{\mathrm{Var}[S(\varepsilon_3 m^2, s)]}{\lambda^2} \leq 2\frac{\varepsilon_3 m^2}{\lambda^2}.$$

Setting $\lambda = m(2\varepsilon_3/\varepsilon_2)^{1/2}$ so that the right hand side above is ε_2, and choosing $\varepsilon_2^3 \leq 2\varepsilon_3$ so that $\lambda \geq \varepsilon_2 m$ gives the claimed result. ∎

Combining Lemmas 77 and 78, we can bound the probability of any failures occurring during our construction.

Lemma 79. *With probability at least $1 - \varepsilon_2 - 1/\operatorname{poly}(m)$, no failures (either precondition failures or probabilistic failures) occur during the construction.*

Proof. Probabilistic failures occur with probability only $1/\operatorname{poly}(m)$ per application of Lemma 75 or Lemma 76. Across the $O(m^2)$ applications of the lemmas, the probability of a probabilistic failure ever occurring is at most $1/\operatorname{poly}(m)$. For the rest of the proof, we condition on no probabilistic failures occurring.

We now bound the probability of any precondition failure. Before any particular application of Lemma 75 or Lemma 76 (during the input sequence of insertions/deletions), for bin $s \in [4]$, the amount by which its load differs from the mean can be expressed as

$$\left|\sum_{i=1}^{k_1} Y_i^{(s)} + S(k_2, s) - k_2/4\right|$$

for some k_1, k_2. By Lemmas 77 and 78, the probability that this quantity ever exceeds $\varepsilon_2 m$ (and hence any precondition failure occurring) is at most $\varepsilon_2 + 1/\operatorname{poly}(m)$, which completes the proof. ∎

Finally, we argue that with probability at least ε_1, the Z_i's *do cause* an imbalance of $\Omega(m)$ at the end of the construction. In particular, bin 1 contains $\Omega(m)$ fewer balls than bins $2, 3, 4$.

Lemma 80. *With probability at least ε_1, we have that*

$$|S(\varepsilon_3 m^2, 1)\}| < \max_{s\in\{2,3,4\}} |S(\varepsilon_3 m^2, s)| - \Omega(\sqrt{\varepsilon_3} m).$$

Proof. Let X_s denote the number of such balls inserted in bin s. Then X_1 is a binomial random variable with mean $\mu = \Theta(\varepsilon_3 m^2)$. Thus, with probability at least

$2\varepsilon_1$, we have that, $X_1 \le \mu - 10\sqrt{\mu}$. On the other hand, if we condition on some value $\le \mu - 10\sqrt{\mu}$ for X_1, then the variables X_2, X_2, X_4 become binomial random variables with means $\mu' > \mu$. Each X_i has probability at least 0.9 of satisfying $X_i > \mu' - 5\sqrt{\mu'} \ge \mu - 5\sqrt{\mu}$. Thus, if we condition on $X_1 \le \mu - 10\sqrt{\mu}$, then the probability at least 0.7, we have $X_2, X_3, X_4 > \mu - 5\sqrt{\mu}$. Putting these together, the probability that $\max\{X_2, X_3, X_4\} - X_1 > 5\sqrt{\mu}$ is at least

$$\Pr[X_1 \le \mu - 10\sqrt{\mu}] \cdot \Pr[X_2, X_3, X_4 > \mu - 5\sqrt{\mu} \mid X_1 \le \mu - 10\sqrt{\mu}] \ge 2\varepsilon_1 \cdot 0.7 > \varepsilon_1. \qquad \blacksquare$$

We can now complete the proof of Proposition 73.

Proof of Proposition 73. We prove the proposition using the construction described in this section. Note that, by design, there are never more than m balls present at a time, as Lemma 76 brings the number of balls back down to $\varepsilon_1 m$ every $O(\varepsilon_1 m)$ operations.

By Lemma 79, with probability at least $1 - \varepsilon_2 - 1/\operatorname{poly}(n)$, all of the applications of Lemma 75 and Lemma 76 succeed. Conditioned on this, at the end of the construction, the gap of each bin $s \in [4]$ can be expressed as

$$\sum_{i=1}^{\varepsilon_3 m} Y_i^{(s)} + S(\varepsilon_3 m^2, s) - \varepsilon_3 m^2/4.$$

By Lemma 77, we have $\left|\sum_{i=1}^{\varepsilon_3 m} Y_i^{(s)}\right| \le \tilde{O}(\sqrt{m})$ with high probability in m. On the other hand, by Lemma 80,

$$|S(\varepsilon_3 m^2, 1)| < \max_{s \in \{2,3,4\}} |S(\varepsilon_3 m^2, s)| - \Omega(\sqrt{\varepsilon_3} m)$$

with probability at least ε_1. It follows that, with probability at least $\varepsilon_1 - \varepsilon_2 - 1/\operatorname{poly}(n)$, the load of bin 1 at the end of the construction is $\Omega(\sqrt{\varepsilon_3} m)$ smaller than the loads of bins $2, 3, 4$. $\blacksquare$

9.2 ModulatedGreedy: Handling $\operatorname{poly}(m)$ Insertions/Deletions

In this section, we consider the insertion/deletion model, with n bins and up to m balls present at a time, and we describe an insertion strategy, called ModulatedGreedy, that achieves a strong bound on maximum load. Here, we describe the simplest possible version of the strategy, which supports any sequence of $\operatorname{poly}(m)$ insertions/deletions while guaranteeing a maximum load of $m/n + O(\log m)$ with high probability in m. Later, in Section 9.3, we will extend ModulatedGreedy in various ways, such as supporting an infinite sequence of insertions/deletions, allowing m to increase over time, etc.

The main result of the section is the following:

Theorem 71. *Let $m \geq n$. Consider the insertion/deletion model with n bins and an upper bound of at most m balls present at a time. Consider a sequence of* poly(m) *insertions/deletions, where insertions are implemented using* ModulatedGreedy. *With high probability in m,* ModulatedGreedy *does not halt during any of the insertions/deletions, and no bin ever has load more than $m/n + O(\log m)$.*

It's worth taking a moment to understand what aspect of the Greedy strategy led to the lower bound in Section 9.1: the main problem with Greedy is that it is too aggressive. Given the choice between two bins i, j, as Greedy always chooses the less loaded of the two—this creates correlations between balls that can be exploited to construct a bad sequence of insertions/deletions. In contrast, ModulatedGreedy will try to be as *unaggressive* as possible, while still guaranteeing an upper gap of $O(\log m)$. In particular, it carefully modulates its behavior and only exhibits a strong bias between two bins i and j if (1) the two bins i and j have significantly different loads; and (2) the system is nearly saturated (i.e., there are nearly m balls present).

As we shall see, this modulated behavior also allows for a simple (but clever) combinatorial analysis, marking a departure from the (typically quite involved) potential-function and Markov-chain arguments used in past analyses of the heavily-loaded case.

9.2.1 The Algorithm

The ModulatedGreedy algorithm for allocating a bin to a ball is given below. We assume without loss of generality that m is a multiple of n.

Algorithm 1 The ModulatedGreedy insertion strategy. Here, ℓ_k is the number of balls in bin k prior to the insertion, and c is a large positive constant.

```
procedure ModulatedGreedy
    Select two bins $i, j \in [n]$ independently and uniformly at random.
    Set $T = m/n + c \log m - \sum_r \ell_r / n$.
    if $(\max_k \ell_k) - (\min_k \ell_k) \leq T$ then
        Assign the ball to bin $i$ with probability $1/2 + \frac{\ell_j - \ell_i}{2T}$, and otherwise assign it to bin $j$.
    else
        Halt.
```

For $k \in [n]$, let ℓ_k denote the load on bin k prior to the insertion, let $\overline{\ell} = \sum_k \ell_k / n$ be the average bin load, and c be a (sufficiently large) fixed constant. When choosing between two bins i, j, the algorithm exhibits bias

$$(\ell_j - \ell_i)/2T$$

towards bin i, where

$$T = m/n + c \log m - \overline{\ell}.$$

Note that the algorithm is well-defined as long as $|\ell_j - \ell_i| \le T$ for all $i, j \in [n]$. One should think of T as representing the average amount of leftover space that each bin would have if each bin had a total capacity of $m/n + c\log m$ balls. This means that the bias is proportional to the difference $\ell_j - \ell_i$ between the loads of the bins, and is inversely proportional to the average amount T of space left in each bin.

The following lemma gives a closed-form solution for the probability of a given bin k being selected by ModulatedGreedy.

Lemma 81. *Suppose that $|\ell_i - \ell_j| \le T$ for all bins i, j. Consider a bin k, and set $T_k = m/n + c\log m - \ell_k$. Upon an insertion, a bin k is selected with probability $T_k/(nT) = T_k/(\sum_i T_i)$.*

Proof. Let i, j denote the random bin choices for the ball being inserted. The probability that a given bin k is selected is given by

$$\begin{aligned}
&\Pr[i=k, j=k] + \sum_{s\neq k} \Pr[i=k, j=s]\left(\frac{1}{2} + \frac{\ell_s - \ell_k}{2T}\right) + \\
&\qquad\qquad \sum_{s\neq k} \Pr[i=s, j=k]\left(\frac{1}{2} + \frac{\ell_s - \ell_k}{2T}\right) \\
&= \frac{1}{n^2} + \frac{2}{n^2}\sum_{s\neq k}\left(\frac{1}{2} + \frac{\ell_s - \ell_k}{2T}\right) = \frac{2}{n^2}\sum_{s=1}^{n}\left(\frac{1}{2} + \frac{\ell_s - \ell_k}{2T}\right) \\
&= \frac{2}{n}\left(\frac{1}{2} + \frac{\overline{\ell} - \ell_k}{2T}\right) = \frac{T + \overline{\ell} - \ell_k}{nT} = \frac{T_k}{nT}.
\end{aligned}$$

Finally we note that $\sum_{i=1}^n T_i = \sum_{i=1}^n (m/n + c\log m - \ell_i) = m + nc\log m - n\overline{\ell} = nT$.
∎

9.2.2 Analysis

To analyze ModulatedGreedy, we begin by describing a seemingly different process (which we call the stone game) that, by design, yields to a simple combinatorial analysis. We then show that the ModulatedGreedy algorithm and the stone game can be *coupled together* so that bounds on the behavior of the stone game directly imply bounds on the behavior of ModulatedGreedy.

Stone Game. In the (Q, n)-*stone game*, parameterized by Q and n, there are Qn stones which are distributed among two bags; an *inactive bag* and an *active bag.* Initially the active bag is empty, and all the stones are in the inactive bag.

The game supports two types of operations: the Activate() operation moves a *random* stone from the inactive bag to the active bag; and the Deactivate(r) operation examines the stones in the active bag, selects the stone that was added the r-th most recently, and moves it back to the inactive bag. (Activate() can only be called if the inactive bag is non-empty, and Deactivate(r) can only be called if the active bag contains r or more balls). The sequence of operations is generated by an oblivious adversary, independent of the random bits used by the game.

The stones are labeled $x_{k,q}$ for $k \in [n], q \in [Q]$. We call k the *color* of the stone, so that there are Q stones of each color. However, the labels of the stone should be thought of as *hidden*, since the behaviors of $\textsc{Activate}()$ and $\textsc{Deactivate}(r)$ do not depend on the labels of the stones.

We will now prove some lemmas establishing that the stone game is, by design, very well behaved. Our first lemma shows that, even though the adversary gets to perform activations/deactivations, it has no control over which specific stones are in the active bag.

Lemma 82. *At any given moment, if the active/inactive bag contains s stones, then these stones are a uniformly random subset of size s of the stones $\{x_{k,q}\}_{k\in[n],q\in[Q]}$.*

Proof. The point is that the activation/deactivation operations do not depend on the labels of the balls.

Formally, fix any sequence of activations/deactivations and the random choices of the $\textsc{Activate}()$ operations, and let S be set of stones currently in the inactive bag (the argument for the active bag is identical). Then for any run of the game with a random permutation π applied to the Qn labels $\{x_{k,q}\}_{k\in[n],q\in[Q]}$, the set stones in the active bag will be $\pi(S)$. Thus, if the inactive bag contains s stones, every s-element subset of the nQ stones is equally likely. ∎

This implies that as long as the inactive bag contains a reasonably large number of stones (namely, $\Omega(n\log(nQ))$), each color is guaranteed to have roughly equal representation in the bag.

Lemma 83. *Suppose at some given moment, the inactive bag contains $s \geq cn\log(nQ)$ stones, for some large enough constant c. Let s_k be the number of these stones with color k. Then $s_k \in [s/2n, 3s/2n]$ for each $k \in [n]$, with probability at least $1 - 1/(Qn)^{\Omega(c)}$.*

Proof. By Lemma 82, the balls S in the inactive bag are a random subset of size s of the Qn balls $\{x_{k,q}\}$. Let $X_k = \{x_{k,1}, \dots, x_{k,Q}\}$ be the set of all color-k balls. Then $s_k = |X_k \cap S|$, the number s_k of balls of color k in S, has the hypergeometric distribution $H(Qn, Q, s)$.

As the standard tail bounds on sampling without replacement at least as sharp as those given by Chernoff bounds for sampling with replacement [188] (Section 23.5), and as $\mathbb{E}[s_i] = s/n$, we get that

$$\Pr[|s_k - s/n| \geq \varepsilon s/n] \leq 2\exp(-\varepsilon^2 s/3n). \tag{9.6}$$

Setting $\varepsilon = 1/2$, and taking a union bound over the n colors, gives that $s_k \in [s/2n, 3s/2n]$ for each $k \in [n]$ with probability $1 - 2n\exp(-\Omega(c\log Qn))$ which is $1 - 1/(Qn)^{\Omega(c)}$ for large enough c. ∎

Relating the stone game to the balls-and-bins game

One can think of the stones in the stone game as being similar to balls in the balls-and-bins game—the active bag represents the set of balls that are present, the color of a stone dictates which "bin" a given ball is in, and activations/deactivations correspond to insertions/deletions.

However, there are several significant differences between the games. Notably, the whole point of the balls-and-bins game is to ensure that no single bin contains too many balls, but in the stone game, the active bag trivially (and deterministically) has at most Q stones of any given color. Nonetheless, we shall now see how to couple the two games together in such a way that our analysis of the stone game yields a bound for the balls-and-bins game.

Mapping between instances. We first give a mapping between the sequence of insertions/deletions for balls-and-bins game and the input sequence for the stone game. For any sequence $\mathcal{S}$ of insertions/deletions in balls-and-bins game, define $\phi(\mathcal{S})$ to be a corresponding sequence of activations/deactivations, where each INSERT operation is replaced with an ALLOCATE operation, and where each DELETE(x) operation on a ball x is replaced with a DEACTIVATE(r) operation, where $r-1$ is the number of balls in the system that were inserted after x.

The following key lemma shows that the random choices in the two games can be coupled.

Lemma 84 (Coupling). *Let $n \le m$ and let $\Delta = c \log m$, where c is the positive constant used by* MODULATEDGREEDY. *Consider a sequence $\mathcal{S}$ of insertions/deletions in a balls-and-bins game on n bins, where there are never more than m balls present at a time. Let G_1 be a balls-and-bins game with operation-sequence $\mathcal{S}$ and let G_2 be (Q, n)-stone game with $Q = m/n + \Delta$ with operation sequence $\phi(\mathcal{S})$.*

If G_1 is implemented using MODULATEDGREEDY, *then there exists a coupling between G_1 and G_2 with the following property: Up until* MODULATEDGREEDY *halts, the number of balls in a given bin k (in the balls-and-bins game) always equals the number of stones in the active bag with color k (in the stone game).*

Proof. Let $\ell_1, \ell_2, \ldots, \ell_n$ denote the loads of the bins at any given moment. By Lemma 81, we know that, on any given insertion in which MODULATEDGREEDY does not halt, each bin k is selected with probability

$$\frac{T_k}{nT} = \frac{T_k}{\sum_{i=1}^n T_i}. \tag{9.7}$$

Now suppose that, for each color k there are ℓ_k stones with color k in the active bag (and hence $Q - \ell_k$ such stones in the inactive bag) of the stone game. Then on any given activation, the probability of a ball with color k being moved into the active bag is

$$\frac{Q - \ell_k}{nQ - \sum_{i=1}^n \ell_i} = \frac{m/n + \Delta - \ell_k}{m + n\Delta - \sum_i \ell_i} = \frac{T_k}{\sum_{i=1}^n T_i}, \tag{9.8}$$

where the first equality uses that $Q = m/n + \Delta$. The two probabilities (9.7) and (9.8) are precisely equal. Thus, we can couple the games so that the bin selected by the insertion in the balls-and-bins game is the same as the stone color selected by the activation in the stone game.

If we implement the insertions/activations in this way, then the deletions/deactivations also become coupled: whenever a ball is deleted from a bin k, a stone with color k is removed from the active bag (in particular, the ball and stone were assigned to have the same bin/color when they were inserted/activated previously). Thus the proof of the lemma is complete. ∎

Proof of Theorem 71. Finally, we can use the coupling in Lemma 84 to bound the probability of ModulatedGreedy halting and prove Theorem 71.

Proof. (Theorem 71) Observe that, if ModulatedGreedy does not halt, then deterministically there are at most $m/n + O(\log m)$ balls in any given bin. In particular, the condition $\max_k \ell_k - \min_k \ell_k \le T$ implies that $\max_k \ell_k - \overline{\ell} \le T$. Plugging $T = m/n + c\log m - \overline{\ell}$, this gives that $\max_k \ell_k \le m/n + c\log m$.

Thus, it suffices to analyze the probability of halting.

By Lemma 84, up until ModulatedGreedy halts, it can be coupled to a stone game on $nQ = m + nc\log m$ balls, where the number of balls in the active bag never exceeds m. Under this coupling, the number of balls ℓ_k in bin k satisfies $\ell_k = Q - s_k$, where s_k is the number of color-k stones in the inactive bag.

The ModulatedGreedy algorithm halts only if

$$|\ell_i - \ell_j| > T = m/n + c\log m - \overline{\ell} = Q - \overline{\ell} \tag{9.9}$$

for some pair i, j of bins. For the stone game, denoting $s = \sum_k s_k = \sum_k (Q - \ell_k) = n(Q - \overline{\ell})$, and as $|s_i - s_j| = |\ell_i - \ell_j|$, condition (9.9) is equivalent to

$$|s_i - s_j| > s/n.$$

But we know by Lemma 83 that, w.h.p. in m, we have $|s_i - s_j| \le s/n$ at all times during the stone game (since the number of balls in the inactive bag is always at least $nc\log m$). Thus, we have w.h.p. in m that ModulatedGreedy never halts. ∎

9.2.3 Tightness of the Bound

Clearly, the bound of $m/n + O(\log m)$ is not optimal for all parameter regimes, since it is known that Greedy achieves maximum load $O(\log\log n)$ in the regime of $n = m$. We remark, however, that for parameter regimes where m is much larger than n, or when n is fixed, this bound is essentially optimal.

Proposition 85. *Consider m insertions into 4 bins using any sequential 2-choice insertion strategy. With probability at least $1/\operatorname{poly}(m)$, some bin contains at least $m/4 + \Omega(\log m)$ balls.*

Proof. Let us consider the final $\log m$ insertions $x_1, \ldots, x_{\log m}$. Suppose, without loss of generality, that prior to those insertions being performed, bins $1, 2$ contain at least as many total balls as bins $3, 4$. With probability $1/\operatorname{poly}(m)$, all of the insertions $x_1, \ldots, x_{\log m}$ are forced to choose between bins 1 and 2. No matter how they are assigned, this forces at least one of bins $1, 2$ to have load $m/4 + \Omega(\log m)$ balls at the end of the insertions. ∎

By repeatedly applying Proposition 85 $\operatorname{poly}(m)$ times, we can amplify the lower bound to apply with high probability.

Corollary 86. *Let c be a sufficiently large positive constant. Consider 4 bins using any sequential 2-choice insertion strategy, and consider a sequence of m^c batches of operations, where each batch consists of m insertions followed by m deletions. With high probability in m, there is some point in time at which some bin contains at least $m/4 + \Omega(\log m)$ balls.*

9.3 Generalizations of ModulatedGreedy

We now generalize the ModulatedGreedy algorithm from Section 9.2 in several interesting ways:

1. We give guarantees over an infinite time horizon, instead of $\operatorname{poly}(m)$ steps.
2. We allow m (the maximum number of balls present in the system) to increase with time, and only require an a-priori bound M on m.
3. We consider the more general $(1+\beta)$-choice and the graphical 2-choice settings (defined in Section 9.3.3) and extend the previous results for these settings (which were insertion-only) to also handle deletions.

These generalizations require extending both the algorithm and the analysis techniques. We begin in Subsection 9.3.1 by describing the algorithm and giving an overview of the key ideas; we then present the analysis and applications in Subsections 9.3.2 and 9.3.3.

9.3.1 The Algorithm and Overview

The algorithm, which we call GeneralizedModulatedGreedy, is described as Algorithm 2 below. Its key properties are summarized in the following theorem.

Theorem 87. *Consider the insertion/deletion model with n bins, and an arbitrarily long sequence of insertions/deletions, with no more than M balls present at a time. Suppose the parameters n, M, ε are known to the algorithm. Then the GeneralizedModulatedGreedy algorithm satisfies the following guarantees:*

- Bounded Load: *At any given moment, every bin has load at most $m/n + O(\varepsilon^{-1} \log M)$ with high probability in M, where m is the largest number of balls that were ever present so far.*

- Bounded Bias: *For any given insertion, if i and j are the two bins being chosen between, then each bin is selected with a probability in the range* $[1/2-\varepsilon, 1/2+\varepsilon]$.

Algorithm 2 The GENERALIZEDMODULATEDGREEDY algorithm. The algorithm has parameters M (an upperbound on the number of balls that will ever be present) and ε, and makes use of a sufficiently large constant $c > 0$. The algorithm outputs a bin and a color for the ball being inserted.

```
procedure GENERALIZEDMODULATEDGREEDY
    For $k \in [n]$, let $\ell_k$ denote # balls with color $k$. Let $\overline{\ell} = \frac{1}{n}\sum_k \ell_k$.
    Let $m$ be the largest number of balls that have been present in the system at once thus far.
    Let $\Delta = c\varepsilon^{-2}\log M$.
    Set $T = \lceil m/n \rceil + \Delta - \overline{\ell}$.
    Select two bins $i, j \in [n]$ independently and uniformly at random.
    if $(\max_k \ell_k) - (\min_k \ell_k) \le \varepsilon T$ then
        With probability $1/2 + \frac{\ell_j - \ell_i}{2T}$, assign the ball to bin $i$ and assign it color $i$.
        Otherwise, assign the ball to bin $j$ and assign it color $j$.
    else
        Declare the ball to be corrupted.
        Select $\rho \in [n]$ such that, for each $k \in [n]$,
            $$\Pr[\rho = k] = \frac{\lceil m/n \rceil + \Delta - \ell_k}{n \cdot T}.$$
        Assign the ball uniformly at random in $\{i, j\}$ and assign it color $\rho$.
```

Notice that the algorithm assigns a ball both a bin and a color. Typically, the color is the same as the bin to which the ball is assigned, but occasionally a ball will get *corrupted*, in which case the bin and color may differ. Moreover, at any time, the maximum load is bounded with respect to m/n (instead of M/n).

Before giving the detailed analysis, we briefly describe the new ideas we need over those in Section 9.2.

Infinite time horizon. A key feature of the algorithm is that it offers guarantees on an infinite time horizon. To achieve this we explicitly incorporate the coupling with the stone game into the design of the algorithm. In particular, whenever there is an insertion that MODULATEDGREEDY would have been at risk of halting on, GENERALIZEDMODULATEDGREEDY instead declares that ball to be ***corrupted***. The algorithm then "fudges" its bookkeeping: it treats the corrupted ball as being placed into whichever bin is necessary to maintain the coupling with the stone game.

More concretely, we assign each ball both to a bin (where it truly resides) and to a color (which, if the ball is corrupted, may differ from the ball's bin). The algorithm makes all of its decisions based on ball colors (and ignores the actual bins that balls reside in). This allows for the algorithm to maintain a coupling forever between the colors of its balls and the colors of the balls in the stone game.

Increasing m. Another interesting feature is that the algorithm allows for m to grow over time, subject only to the constraint $m \leq M$. To handle this, GENERALIZEDMODULATEDGREEDY bases its allocation decisions on the largest value of m that it has witnessed so far. At first glance, this seems to significantly break the relationship between the balls-and-bins game and the stone game, and indeed Lemma 82 no longer holds—however, as we shall see, the stone game and its analysis can be modified to also handle the incremental growth in m over time.

Bias, $(1 + \beta)$-choice and graphical process. Finally, a third feature of the algorithm is that it introduces a new variable ε that constrains the amount of bias that the algorithm is permitted to exhibit. We will see at the end of the section that this seemingly minor modification allows us to extend the algorithm to the $(1 + \beta)$-choice and the graphical 2-choice process, both of which are generalizations of the classical 2-choice process. Moreover, the guarantees of the resulting algorithms matches the previous known results for the insertion-only case for these settings.

9.3.2 Algorithm Analysis

We now turn to proving Theorem 87. We begin by defining the generalized stone game, which extends the stone game in Section 9.2. Then we show how this game is closely related to the balls and bins game and use this relationship to analyze GENERALIZEDMODULATEDGREEDY.

The generalized stone game

The Δ-GENERALIZED STONE GAME has an inactive bag and an active bag. The inactive bag is initialized to contain $\Delta \cdot n$ stones $x_{k,j}$ for $k \in [n]$ and $q \in [\Delta]$, and the active bag is initialized to be empty. We say that the ball $x_{k,q}$ has *color* $k \in [n]$. The game supports two operations that are performed by an oblivious adversary: ACTIVATE() and DEACTIVATE(r).

The ACTIVATE() operation (described formally in Algorithm 3) takes two steps: First, the operation moves a random stone from the inactive bag to the active bag. Second, if there are fewer than $\Delta \cdot n$ stones in the inactive bag, then it computes the number $Q \cdot n$ of stones currently in the system (active and inactive bags), and it adds n new stones $\{x_{k,Q+1}\}_{k\in[n]}$, one of each color, to the inactive bag. This second step is different from the standard stone game in Section 9.2, and in particular, the total number of stones now can increase over time (in increments of n).

The DEACTIVATE(r) operation works exactly as before—it takes whichever stone was added to the active bag r-th most recently, and moves that stone back to the inactive bag.

Algorithm 3 The Activate method for the generalized stone game. The algorithm has parameter Δ. The moves a random stone from the inactive bag to the active bag, and then (possibly) adds additional stones to the inactive bag.

```
procedure Activate
    Move a random stone from the inactive bag to the active bag.
    if Inactive bag contains fewer than $\Delta \cdot n$ balls then
        Let $Q \cdot n$ be # stones currently in the system
        Add a batch $B_{Q+1} = \{x_{k,Q+1}\}_{k \in [n]}$ of $n$ new balls to the inactive bag.
```

We begin by proving a basic fact about the generalized stone game.

Lemma 88. *Let $c > 0$ be a sufficiently large constant, and let ε, M be parameters. Fix any time in the $(c\varepsilon^{-2} \log M)$-generalized stone game, and for $k \in [n]$, let s_k denote the number of stones with color k in the inactive bag. With probability $M^{-\Omega(c)}$, for each $k \in [n]$, we have that*

$$(1 - \varepsilon/2)\mathbb{E}[s_k] \le s_k \le (1 + \varepsilon/2)\mathbb{E}[s_k].$$

Proof. Let $Q \cdot n$ be the number of stones currently in the system. For each $q \in \{1, 2, \ldots, Q\}$, define $B_q = \{x_{k,q}\}_{k \in [n]}$. The n stones in B_q are all inserted into the system in the same instant and are indistinguishable from one another in terms of how they interact with the sequence of operations being performed. If there are a_k balls from B_k in the inactive set, then the probability that any of them have color i is simply a_k/n.

Thus, if we fix some outcome for the values of the a_k's, then we can write $s_k = \sum_{q=1}^{Q} A_k$, where A_k are independent indicator random variables with $\Pr[A_q = 1] = a_q/n$. Using I to denote the set of balls in the inactive set, the expected value of s_k evaluates to

$$\mathbb{E}[s_k] = \sum_{q=1}^{Q} a_q/n = |I|/n.$$

By design, however, the inactive set always at least $|I| \ge \Delta \cdot n = c\varepsilon^{-2} n \log M$ balls, so that $\mathbb{E}[s_k] \ge \Omega(c\varepsilon^{-2} \log M)$. Applying a Chernoff bound (and as c is a large constant), for each $k \in [n]$, s_k lies between $(1 - \varepsilon/2)\mathbb{E}[s_k]$ and $(1 + \varepsilon/2)\mathbb{E}[s_k]$ with probability $M^{-\Omega(c)}$. ∎

Coupling with GeneralizedModulatedGreedy

Next we establish the connection between the generalized stone game and the GeneralizedModulatedGreedy algorithm.

First, as in Section 9.2, the oblivious sequences of insertion/deletions for the balls-and-bins game maps to an input sequence of the Δ-generalized stone game as follows: each insertion in the balls-and-bins game causes an activation in the stone game, and each deletion Delete(x) in the balls-and-bins game causes a deactivation

DEACTIVATE(r), where $r-1$ is the number of balls present in the balls-and-bins game that were inserted after x.

The following key lemma shows that the random choices in the two games can be coupled.

Lemma 89 (Coupling). *Consider a sequence $\mathcal{S}$ of insertions/deletions in a balls-and-bins game on n bins, with no more than M balls present at a time. Let G_1 be a balls-and-bins game with operation-sequence $\mathcal{S}$, let $\Delta = c\varepsilon^{-2}\log M$, and let G_2 be Δ-generalized stone game with operation sequence $\phi(\mathcal{S})$.*

If G_1 is implemented using the GENERALIZEDMODULATEDGREEDY *algorithm with parameters M, c and ε, then there exists a coupling between G_1 and G_2 such that: (1) the number of balls with a given color $k \in [n]$ in G_1 always equals the number of active-bag stones with color k in G_2; and (2) the total number $n \cdot Q$ of stones in G_2 always satisfies $Q = \lceil m/n \rceil + \Delta$, where m is the largest number of balls ever present at once so far in the balls-and-bins game.*

Proof. Let ℓ_k denote the number of balls with color k at any given moment and let $\overline{\ell} = \sum_k \ell_k/n$. By Lemma 81 (modified so that $T = \lceil m/n \rceil + \Delta - \overline{\ell}$ and $T_k = \lceil \frac{m}{n} \rceil + \Delta - \ell_k$), we know that, on any given insertion in which GENERALIZEDMODULATEDGREEDY does not create a corrupted ball, each color k is selected with probability

$$\frac{T_k}{n \cdot T} = \frac{\lceil \frac{m}{n} \rceil + \Delta - \ell_k}{n \cdot T}. \tag{9.10}$$

On the other hand, on insertions that do create corrupted balls, we have by design that (9.10) is still the probability of color k being selected. Thus, (9.10) is always the probability of any given color k being selected on any given insertion.

Next we turn our attention to the generalized stone game. By design, the number $n \cdot Q$ of stones in the generalized stone game at any given moment satisfies $Q = \lceil m/n \rceil + \Delta$, where m is the largest number of balls that have ever been present at once in the balls-and-bins game. Suppose that, for each color k there are ℓ_k stones with color k in the active set of the stone game. Then on any given activation, the probability of a ball with color k being moved into the active set is

$$\frac{Q - \ell_k}{n \cdot Q - \sum_i \ell_i} = \frac{\lceil \frac{m}{n} \rceil + \Delta - \ell_k}{n \cdot (\lceil \frac{m}{n} \rceil + \Delta - \overline{\ell})} = \frac{\lceil \frac{m}{n} \rceil + \Delta - \ell_k}{n \cdot T}. \tag{9.11}$$

The two probabilities (9.10) and (9.11) are precisely equal. Thus, we can couple the games so that the color selected by the insertion in the balls-and-bins game is the same as the stone color selected by the activation in the stone game.

If we implement the insertions/activations in this way, then the deletions/deactivations also become coupled: whenever a ball is deleted with a color k, a stone with color k is removed from the active bag (in particular, the ball and stone were assigned to have the same color when they were inserted/activated previously). Thus the proof of the lemma is complete. ∎

Combining Lemmas 89 and 88, we can bound the probability that a given ball is corrupted.

Lemma 90 (Corruption probability)**.** *Consider a sequence of insertions/deletions in a balls-and-bins game on n bins with no more than M balls ever present at a time, and suppose that insertions are implemented using the* GeneralizedModulatedGreedy *algorithm with parameters M and ε. For any given insertion, the probability that the ball being inserted is corrupted is at most $1/\operatorname{poly}(M)$.*

Proof. For $k \in [n]$, let ℓ_k denote the number of balls with color k. Let $\overline{\ell} = \sum_k \ell_k/n$ and let $\Delta = c\varepsilon^{-2}\log M$, where c is the constant used by GeneralizedModulatedGreedy. In order for the inserted ball to be corrupted, we would need

$$\left(\max_k \ell_k\right) - \left(\min_k \ell_k\right) > \varepsilon T = \varepsilon(\lceil m/n\rceil + \Delta - \overline{\ell}). \tag{9.12}$$

If we couple the process to a Δ-generalized stone game as in Lemma 88, then we have (1) that the number of balls with each color k in the active bag of the generalized stone game is ℓ_k; and (2) that the total number of stones in the generalized stone game is $n(\lceil m/n\rceil + \Delta)$. It follows by Lemma 93 that, w.h.p. in M,

$$(1-\varepsilon/2)\mathbb{E}[s_k] \le s_k \le (1+\varepsilon/2)\mathbb{E}[s_k],$$

where $s_k = \lceil m/n\rceil + \Delta - \ell_k$ and $\mathbb{E}[s_k] = \lceil m/n\rceil + \Delta - \overline{\ell}$. That is, each s_k deviates by at most $\frac{1}{2}\varepsilon(\lceil m/n\rceil + \Delta - \overline{\ell})$ from its mean. The same holds for each ℓ_k (as $\ell_k + s_k$ is fixed), which implies that (9.12) does not occur. ∎

Finally, we can prove Theorem 87.

Proof of Theorem 87. It suffices to prove the Bounded Load guarantee, since the Bounded Bias guarantee is hardcoded into the GeneralizedModulatedGreedy algorithm by design. In particular, given the bin choices i, j, if the ball is not corrupted then $|\ell_i - \ell_j| \le \varepsilon T$ and it is assigned to bin i with probability $1/2 + (\ell_j - \ell_i)/2T \le 1/2 + \varepsilon/2$. On the other hand if it is corrupted, then it is assigned uniformly.

Let $\Delta = c\varepsilon^{-2}\log M$. Couple the balls-and-bins game to the Δ-generalized stone game as in Lemma 89, and consider the state of both systems at some fixed point in time.

By Lemma 90 (applied with a union bound to each of the balls that are present at any given moment), we have with high probability in M that there are no corrupted balls in the balls-and-bins game. Thus the number of balls in any given bin k (in the balls-and-bins game) is equal to the number of active-bag stones with color k (in the generalized stone game). Moreover, if m is the most balls that were ever present in the balls-and-bins game, the number of stones in the generalized stone game is $\lceil m/n\rceil + \Delta$.

Using ℓ_k to be the number of active-bag stones with color k, and s_k to be the number of inactive-bag stones with color k, by Lemma 93 we have that $s_k > (1 -$

$\varepsilon)\mathbb{E}[s_k] \geq (1-\varepsilon)\Delta$, which gives the desired bound

$$\ell_k = \lceil m/n \rceil + \Delta - s_k \leq \lceil m/n \rceil + \varepsilon\Delta = m/n + O(\varepsilon^{-1} \log M). \quad \blacksquare$$

9.3.3 Extensions

We conclude the section with applications of GeneralizedModulatedGreedy to several more general settings.

$(1+\beta)$-choice process. The $(1+\beta)$-choice setting was proposed by Peres, Talwar, and Wieder [308] as a useful generalization of the 2-choice process, where each insertion selects a random bin with probability $(1-\beta)$, and gets to choose between two random bins i, j with probability β. For any fixed $0 < \beta < 1 - \Theta(1)$, they showed that in the insertion-only case, the Greedy algorithm achieves maximum load $m/n + \Theta(\beta^{-1} \log n)$ with high probability in n; this load becomes $m/n + \Theta(\beta^{-1} \log m)$ if one wishes for a high-probability guarantee in m. They further proved that these bounds are optimal for any $(1+\beta)$-choice insertion strategy.

We can directly use GeneralizedModulatedGreedy to construct an optimal $(1+\beta)$-choice insertion strategy for the insertion/deletion model.

Theorem 91. *Consider a balls-and-bins game with n bins and with no more than m balls present at a time. In the insertion/deletion model, there exists a $(1+\beta)$-choice algorithm that at any given moment, with probability in m, has maximum load*

$$m/n + O(\beta^{-1} \log m).$$

Proof. If we set $\varepsilon = \beta/2$, then GeneralizedModulatedGreedy selects between bins i, j with a probabilities in the range $1/2 \pm \varepsilon$; this is equivalent to selecting a random bin (i.e., a random one of i, j) with probability $1 - 2\varepsilon = 1 - \beta$, and then selecting between bins i, j with a probabilities in the range $[0, 1]$.

Graphical-Allocation. Graphical allocation is another generalization of the 2-choice model, introduced by Kenthapadi and Panigrahy [220]. Here we are given an arbitrary fixed d-regular graph G on n vertices (i.e., bins). To assign a ball to a bin, we select a uniformly random edge $e = (v_1, v_2)$ choose one of bins v_1, v_2. The classic 2-choice process corresponds to the complete graph $G = K_n$.

Bansal and Feldheim [59] showed that, in the insertion-only case, it is possible to guarantee a maximum load of $m/n + O((d/k) \log^4 n \log\log n)$ w.h.p. in n, where k is the edge-connectivity of G. The linear dependence on (d/k) is necessary and the bound becomes $m/n + O((d/k) \log m \log^3 n \log\log n)$ if one requires the bound to be w.h.p. in m.

Their algorithm reduces the problem, in a black-box manner, to that of constructing a $(1+\beta)$-choice strategy on two bins (in particular, where the two "bins" represent sibling sets in a binary hierarchical decomposition of the vertices of G, and the different sibling pairs use different choices for β, see [59]). In the insertion-only case [59], they use the GREEDY $(1+\beta)$-choice strategy—to extend this to handle deletions, we can simply use GENERALIZEDMODULATEDGREEDY instead (as in Theorem 91). Together with the framework developed in [59], this gives the following result.

Theorem 92. *Consider a graphical process where, given a k-edge-connected d-regular graph G on n vertices (i.e., bins), the two bin choices for each ball are given by the endpoints of a uniformly random edge $e = (v_1, v_2)$ of G. Consider any sequence of insertions/deletions where the number of balls in the system never exceeds m. Then it is possible to guarantee a maximum load of $m/n + O((d/k)\log m \log^3 n \log\log n)$ w.h.p. in m, at any given moment.*

Appendices

9.A Proof of Lemma 74

We prove Lemma 74, reformulated here to use a constant c in place of constants $\varepsilon_1, \varepsilon_2$, and to use a variable k in place of $\varepsilon_2 m$:

Lemma 93 (Lemma 74 reformulated)**.** *Let $c > 0$ be a sufficiently large constant. Consider the* Greedy *algorithm on 4 bins, and fix an arbitrary initial state in which the bins have loads within k of each other. If ck insertions are performed, then after the sequence is complete, all of the bins have loads within $O(\log k)$ of each other with high probability in k. Furthermore, with high probability in k, there is some intermediate point in time during which all of the bins have equal loads.*

We break the proof of this lemma into a few simple claims.

Claim 94. *Given an arbitrary initial state with bin loads within k of each other, if $j \geq ck$ insertions are performed, then at end of the sequence, the bin loads will be within $O(\log k)$ of each other, w.h.p. in k.*

Proof. Let $D_{i,j}$ be the difference between the loads of the i-th and j-th bins (where $i \neq j$). It suffices to show that, after the insertions are complete, $D_{i,j} \leq O(\log k)$ with high probability in k.

Notice that whenever $D_{i,j} \neq 0$ and we insert a ball, $D_{i,j}$ has a random increment with $\Omega(1)$ bias towards 0 (it surely decreases by 1 when i, j are the two choices, which has $\Omega(1)$ probability as $n = 4$, and has zero bias otherwise). So starting at $|D_{i,j}| \leq k$, w.h.p. in k that the random walk thus reaches 0 within $O(k) \leq ck$ steps. Moreover, each time that the random walk hits 0, w.h.p. in k it will hit 0 again within $O(\log k)$ steps. Thus, after the ck insertions are performed, we have $|D_{i,j}| = O(\log k)$ w.h.p. in k. ∎

Next we show that, during the insertions, the loads become equal at some point with probability $\Omega(1)$.

Claim 95. *Given any arbitrary initial state the bin loads within k of each other, if $2ck$ insertions are performed, then with probability at least $\Omega(1)$ there is some time at which all the 4 bins have equal loads.*

Proof. This follows by iterated applications of Claim 94. After ck insertions, all the 4 the bins have loads within $T_1 = O(\log k)$ of each other, w.h.p. in k. After cT_1 further insertions, the bins have loads within $T_2 = O(\log T_1)$ of each other, w.h.p. in T_1. After cT_2 further insertions, the bins have loads within $T_3 = O(\log T_2)$ of each other, w.h.p. in T_2. Continuing like this, after $c(k + T_1 + T_2 + \cdots + T_{O(\log^* n)}) = (c + o(1))k$ insertions, we reach a state where all bin loads are within $O(1)$ of each other with probability $\Omega(1)$. Once this occurs, we have with probability $\Omega(1)$ that during the next $O(1)$ insertions after that, there is a point at which the 4 bins have equal loads. ∎

Finally, we amplify Claim 95 in order to achieve a high-probability bound.

Claim 96. *Given an arbitrary initial state with bin loads within k of each other, if ck insertions are performed, then w.hp. in k there is some time when all the bins have equal loads.*

Proof. By Claim 94, w.h.p. in k) the loads are within $T = O(\log k)$ of each other during each of the final $ck/2$ insertions. Break these insertions into $\Omega(k/\log k)$ chunks of size $2cT$. Within each chunk, we have by Claim 95 that the loads equalize (at some point) with probability at least $\Omega(1)$. Thus, the probability that the loads stay unequal during all $\Omega(k/\log k)$ chunks is $\exp(-\Omega(k/\log k))$. ∎

Combined, Claims 94 and 96 imply Lemma 93.

9.B Proof of Lemma 100

For $(i, j) \in Q$, define $A_{i,j}$ (resp. $B_{i,j}$) to be the set of balls in A (resp. B) that hash to the bin pair (i, j). Let $a_{i,j} = |A_{i,j}|$ and $b_{i,j} = |B_{i,j}|$. Let

$$p_{i,j} = \frac{v\left(A_{i,j} \cup B_{i,j}\right)}{|A_{i,j} \cup B_{i,j}|}$$

denote the (random) fraction of balls in $A_{i,j} \cup B_{i,j}$ that are placed into bins $1, 2$.

We remark that there are two sources of randomness in this lemma: the first, which we denote by $\mathcal{R}_1$, is the outcome of the hashes of the balls in A and B (i.e., the random bits that determine $\{a_{i,j}\}$ and $\{b_{i,j}\}$); the second, which we denote by $\mathcal{R}_2$, is the random order in which the balls $A \cup B$ are inserted into the system.

Note that, from the perspective of the ID-oblivious insertion strategy, the balls $A_{i,j}$ are indistinguishable from the balls $B_{i,j}$ (this is due to the randomness from $\mathcal{R}_2$). Thus we have that, for any fixed outcome of $\mathcal{R}_1$,

$$\mathbb{E}\left[v(A_{i,j}) - v(B_{i,j}) \mid \mathcal{R}_1\right] = \mathbb{E}[p_{i,j}(a_{i,j} - b_{i,j}) \mid \mathcal{R}_1].$$

Summing over $(i, j) \in Q$, we have that (again for any fixed outcome of $\mathcal{R}_1$)

$$\mathbb{E}[v(A) - v(B) \mid \mathcal{R}_1] = \sum_{(i,j)\in Q} \mathbb{E}\left[v(A_{i,j}) - v(B_{i,j}) \mid \mathcal{R}_1\right] = \sum_{(i,j)\in Q} \mathbb{E}[p_{i,j}(a_{i,j} - b_{i,j}) \mid \mathcal{R}_1].$$

Considering all outcomes for $\mathcal{R}_1$ that satisfy $\mathcal{E}$, it follows that

$$\mathbb{E}[v(A) - v(B) \mid \mathcal{E}] = \sum_{(i,j)\in Q} \mathbb{E}[p_{i,j}(a_{i,j} - b_{i,j}) \mid \mathcal{E}].$$

Thus, to prove the lemma, it suffices to show that

$$\mathbb{E}\left[\sum_{(i,j)\in Q} p_{i,j}(a_{i,j} - b_{i,j}) \mid \mathcal{E}\right] \geq t - O(\sqrt{k}).$$

Note that $p_{(1,2)} = 1$ and $p_{(3,4)} = 0$ deterministically. Moreover,

$$\mathbb{E}[a_{1,2} - b_{1,2} \mid \mathcal{E}] \geq \mathbb{E}[k/12 + t - O(\sqrt{k}) - b_{1,2}] = t - O(\sqrt{k}) - \mathbb{E}[b_{1,2} - k/12] = t - O(\sqrt{k}).$$

Thus

$$\begin{aligned}
&\mathbb{E}\left[\sum_{(i,j)\in Q} p_{i,j}(a_{i,j} - b_{i,j}) \mid \mathcal{E}\right] \\
&= \mathbb{E}[a_{1,2} - b_{1,2} \mid \mathcal{E}] + \mathbb{E}\left[\sum_{(i,j)\in Q\setminus\{(1,2),(3,4)\}} p_{i,j}(a_{i,j} - b_{i,j}) \mid \mathcal{E}\right] \\
&= t - O(\sqrt{k}) + \mathbb{E}\left[\sum_{(i,j)\in Q\setminus\{(1,2),(3,4)\}} p_{i,j}(a_{i,j} - b_{i,j}) \mid \mathcal{E}\right] \\
&\geq t - O(\sqrt{k}) - \sum_{(i,j)\in Q\setminus\{(1,2),(3,4)\}} \mathbb{E}[|a_{i,j} - b_{i,j}| \mid \mathcal{E}].
\end{aligned}$$

To complete the proof, it suffices to show that for each $(i,j) \in Q \setminus \{(1,2),(3,4)\}$, we have

$$\mathbb{E}[|a_{i,j} - b_{i,j}| \mid \mathcal{E}] \leq O(\sqrt{k}).$$

Let $\alpha_{i,j} = \mathbb{E}[a_{i,j} \mid \mathcal{E}]$ and $\beta_{i,j} = \mathbb{E}[b_{i,j} \mid \mathcal{E}]$. By Chernoff bounds, we know that $\mathbb{E}[|a_{i,j} - \alpha_{i,j}| \mid \mathcal{E}] \leq O(\sqrt{k})$ and $\mathbb{E}[|b_{i,j} - \beta_{i,j}| \mid \mathcal{E}] \leq O(\sqrt{k})$. Thus, it suffices to show that

$$|\alpha_{i,j} - \beta_{i,j}| = O(\sqrt{k}).$$

For each ball $x \in A$ with $h(x) \notin \{(1,2),(3,4)\}$, we have that $h(x)$ is random among the $|Q| - 2 = 10$ pairs in $Q \setminus \{(1,2),(3,4)\}$; and for each ball $x \in B$, we have that $h(x)$ is random among the $|Q| = 12$ pairs in Q. Thus $\alpha_{i,j} = \mathbb{E}[\frac{1}{10}(k - a_{1,2} - a_{3,4}) \mid E]$ and $\beta_{i,j} = k/12$. Finally, as $a_{1,2} + a_{3,4} = k/6 \pm O(\sqrt{k})$ (conditioned on event $\mathcal{E}$ occurring), we get

$$\begin{aligned}
&\alpha_{i,j} - \beta_{i,j} \\
&= \mathbb{E}\left[\frac{1}{10}(k - a_{1,2} - a_{3,4}) \mid \mathcal{E}\right] - k/12 \\
&= \frac{1}{10}(k - k/6) - k/12 \pm O(\sqrt{k}) \\
&= \pm O(\sqrt{k}),
\end{aligned}$$

which completes the proof.

Chapter 10

The Data-Structural Perspective

In this chapter, we consider the reinsertion/deletion model. Here, we show that not only does the GREEDY strategy do poorly, but more generally any ID-oblivious strategy also does poorly. In the parameter regime that most matters to data structures, however, we show that it is still possible to achieve a nontrivial improvement over the known results—the new strategy that we present, called ICEBERG will be useful as an algorithmic tool for data-structural applications later in the thesis.

An impossibility result for deletions with reinsertions. In Section 10.1, we show that no ID-oblivious insertion strategy can guarantee sub-polynomial overload.

Theorem 97. *Consider the reinsertion/deletion model with* 4 *bins, and with a limit of up to* m *balls present at a time. Against any ID-oblivious insertion strategy, it is possible for an oblivious adversary to force a maximum load of* $m/4 + m^{\Omega(1)}$ *at some point in the first* $\operatorname{poly}(m)$ *operations, with high probability in* m*.*

Unlike our lower bound in the previous chapter, this lower bound does not easily extend to $n > 4$. It remains an open question whether one can prove a strong lower bound, for example, in the setting where $m = \operatorname{poly}(n)$ for some large polynomial.

A better algorithm for the moderately-loaded regime In Section 10.2, we present a 3-choice allocation rule called ICEBERG that achieves an overload of $o(m/n) + O(\log\log n)$:

Theorem 98. *Suppose* $1 \le h \le n^{o(1)}$*. Suppose balls are inserted/deleted/reinserted into* n *bins over time (by an oblivious adversary) according to the* ICEBERG *rule, where each ball has three random choices for where it can go, and where there are never more than* hn *balls present at a time. Then, w.h.p. in* n*, at any given moment, the number of balls in the fullest bin is* $h + O(\sqrt{h \log h}) + \log\log n + O(1)$*.*

This theorem is of special interest in the regime where $h = o(\log n)$, since in this regime it was previously unknown how to achieve an overload of $o(\log n)$. The ICEBERG rule will also have numerous applications later in the thesis when we apply balls-and-bins to dynamic data-structural applications.

The name ICEBERG stems from the visual structure of the strategy: almost all balls are inserted via SINGLECHOICE, and these balls all rest at height at most $h + O(\sqrt{h \log h})$ (think of these balls as the large portion of an iceberg that rests below water). The small number of balls that exceed this height are then allocated using GREEDY (think of these as the small part of an iceberg that sticks out above water). Critically, GREEDY is being applied to a set of at most $O(n)$ balls here, which is the one parameter regime where it is known to do very well even with insertions/deletions/reinsertions [350, 351, 369]. The name is in reference to the fact that most icebergs are much larger than they look, because only a small fraction of the iceberg sticks out above water.

10.1 An Impossibility Result For The Deletions with Reinsertions

In this section, we prove an impossibility result for the reinsertion/deletion model, namely, that no ID-oblivious insertion strategy can guarantee sub-polynomial overload.

Theorem 97. *Consider the reinsertion/deletion model with* 4 *bins, and with a limit of up to m balls present at a time. Against any ID-oblivious insertion strategy, it is possible for an oblivious adversary to force a maximum load of $m/4 + m^{\Omega(1)}$ at some point in the first* $\text{poly}(m)$ *operations, with high probability in m.*

The section splits the proof of Theorem 97 into two parts. First, in Subsection 10.1.1, we introduce and analyze the so-called ***marble-splitting game***; then, in Subsection 10.1.2 we show how to perform a sequence of insertions/deletions that simulates an instance of the marble-splitting game and forces some bin to contain load $m/4 + m^{\Omega(1)}$ with non-negligible probability.

10.1.1 The Marble-Splitting Game

In this section we present and analyze a simple game, which we call the ***marble-splitting game***—the game plays an important role in our lower bound for balls-and-bins games with reinsertions.

In the marble-splitting game, there are two players Alice and Bob. The player Alice has two types of moves: she can perform an INSERT operation, which adds a new marble into the game, or she can perform a SPLIT(x, y) operation, which takes two marbles x and y and replaces them with new marbles x' and y'. Alice must decide her moves at the beginning of time (so she is an oblivious adversary).

The second player Bob gets to assign a ***value*** v_x to each marble x, according to the following rule: whenever Alice performs an INSERT, Bob can assign the new marble an arbitrary real-numbered value in the range $[-1, 1]$; and whenever Alice performs a $\textsc{Split}(x, y)$ operation, Bob assigns x' and y' values $v_{x'}$ and $v_{y'}$ satisfying

$$v_{x'} + v_{y'} = v_x + v_y \pm o(R^{-2}), \quad \text{and} \quad v_{x'} - v_{y'} \geq 2/R. \tag{10.1}$$

Equivalently, $v_{x'} = (v_x + v_y)/2 + \Delta \pm o(R^{-2})$ and $v_{y'} = (v_x + v_y)/2 - \Delta \pm o(R^{-2})$ for some $\Delta \geq 1/R$.

Alice's goal is to force some marble (she need not know which one) to have a value greater than 1 at some point within the first $O(R^3)$ steps of the game. Her disadvantage is that she does not know the precise values of marbles. Intuitively, she would like to perform split operations on marbles x and y that satisfy $|v(x) - v(y)| = o(1/R)$. But she might, for example, accidentally split two marbles x and y whose values differ considerably—this would result in x' and y' having values that are *closer together* than x and y had, which is intuitively counterproductive for Alice. We shall see that, nonetheless, Alice can deterministically force a win within $O(R^3)$ steps.

In constructing Alice's strategy, we will find it helpful for accounting purposes to artificially place the following additional constraints on Alice. We think of there as being *bags* $0, 1, 2, \ldots$, each of which is capable of holding arbitrarily many marbles. Whenever a marble is inserted, we place it in bag 1. Whenever a split $\textsc{Split}(x, y)$ operation is performed, we require that the marbles x and y are currently in the *same* bag $i \geq 1$ as each another, and after the split, we place the new marbles x' and y' into bags $i + 1$ and $i - 1$, respectively. This restriction somewhat limits Alice's possible strategies, but, as we shall see, it also simplifies the task of analyzing Alice's "progress" over time.

The key result of this section is the following.

Proposition 99. *Alice can deterministically force some marble to have a value greater than 1 at some point within the first $O(R^3)$ steps of the game. Moreover, the strategy performs only $O(R^2)$ insertions.*

Proof. We begin by describing Alice's strategy. Let c be a large positive constant. She initially performs one insertion into bag 1. She then proceeds in cR phases, where at the beginning of phase $i \in \{1, 2, \ldots, cR\}$, the state of the system is as follows: bag 0 contains some arbitrary number of marbles; bags $1, 2, \ldots, i$ each contain one marble; and bags $i + 1, i + 2, \ldots$ are empty.

The i-th phase consists of $(i + 1)$ sub-phases, where at the beginning of each subphase $j \in \{1, 2, \ldots, i + 1\}$, the state of the system is as follows: bag 0 contains some arbitrary number of marbles; and, with the exception of bag $i - j + 2$, which is empty, all of bags $2, 3, 4, \ldots, i + 1$ contain one marble (so bags $1, 2, 3, 4, \ldots, i - j + 1$ each contain one marble; bag $i - j + 2$ is empty; and bags $i - j + 3, \ldots, i + 1$ each contain one marble).

The $(i + 1)$-th subphase is special in that, all Alice does is perform one more insertion in order to reach the starting state for phase $i+1$ (i.e., all of bags $1, 2, \ldots, i+1$ contain 1 marble).

For $j < i + 1$, the j-th subphase of phase i is implemented as follows. Alice inserts one marble into bag 1. She then performs splits, one after another, on bags

$1, 2, 3, \ldots, i-j+1$. For each $t \in \{1, 2, \ldots, i-j\}$ (i.e., for every split but the final split), after she performs a split on bag t, the state of the system is that: bags $1, 2, \ldots, t-1$ contain one marble each; bag t is empty; bag $t+1$ contains 2 marbles; and bags $t+2, t+3, \ldots$ are as they were at the beginning of the subphase. The final split that Alice performs (i.e., the split in $i-j+1$) has the effect of placing a marble into the previously empty bags $i-j$ and $i-j+2$, and leaving bag $i-j+1$ as the solitary empty bag out of bags $0, 1, 2, \ldots, i+1$. Thus we reach the starting state for the $(j+1)$-th subphase.

Analysis of the strategy. The analysis will need only the following basic facts about Alice's strategy: (1) it performs a total of $O(c^2R^2)$ Insert operations and $\Omega(c^3R^3)$ Split operations; (2) it only places marbles in bags $i \le cR+1$; and (3) at the end of the game, there is at most 1 marble in each bag i for $i > 0$.

Let B_i denote the marbles in bag i at any given moment, and define the potential function

$$\phi = \sum_{i=0}^{\infty} i \cdot \sum_{x \in B_i} v_x.$$

We will prove the proposition by analyzing how ϕ evolves over time.

Each time that an Insert is performed, ϕ may decrease by up to 1, as $v_x \in [-1, 1]$ and the marble is inserted in bag 1. During the entire game, this leads to a decrease of at most $O(c^2R^2)$.

Each time that a Split is performed, two marbles x and y in some bag i are replaced by x' and y' with values given by (10.1). Removing x and y decreases ϕ by $i \cdot (v_x + v_y)$ and inserting x' and y' increases ϕ by

$$(i+1)v_{x'} + (i-1)v_{y'} = i \cdot (v_x + v_y) + (v_{x'} - v_{y'}) \pm o(iR^{-2}) \ge i \cdot (v_x + v_y) + 1/R.$$

The net effect of a split is therefore to increase ϕ by at least $1/R$. As there are $\Omega(c^3R^3)$ split operations across the entire game, this increases ϕ by $\Omega(c^3R^2)$.

Combining the bounds for Insert and Split operations, we have that, at the end of the game,

$$\phi \ge \Omega(c^3R^2) - O(c^2R^2) = \Omega(c^3R^2).$$

But this means that some bin $i \le cR+1$ must satisfy $i \cdot \sum_{x \in B_i} v_x > \Omega(c^2R)$, and thus that $\sum_{x \in B_i} v_x > \Omega(c)$. As $|B_i| \le 1$, this implies that there is a ball x with $v_x > 1$. ∎

Remark. It is worth noting that, in the strategy in Proposition 99, we could have alternatively performed all of the insertions into bag 1 up front (i.e., at the beginning of the game), and then applied the appropriate Split operations without performing any further insertions—each marble would simply remain in bag 1 until it was used for the Split operations involving it. This perspective will be convenient in our application of marble splitting.

10.1.2 Proof of Theorem 97

We will now derive a sequence of insertions/deletions that can be used to establish Theorem 97.

As notation, let $Q = \{(i,j) \mid i,j \in [4], i \neq j\}$, and let h be a fully independent hash function mapping each ball x to a uniformly random pair $h(x) = (h_1(x), h_2(x)) \in Q$. Notice that $|Q| = 12$.

insertion

$$|\{x \in A \mid h(x) = (1,2)\}| = k/12 + t \pm O(\sqrt{k}) \tag{10.2}$$

$$\text{and} \quad |\{x \in A \mid h(x) = (3,4)\}| = k/12 - t \pm O(\sqrt{k}). \tag{10.3}$$

Note that $\mathcal{E}$ only depends on the hash values for balls in A.

We will show that, if we condition on $\mathcal{E}$ occurring, and if t is moderately large (i.e., $c\sqrt{k}$ for some sufficiently large positive constant c), then we can perform a sequence of insertions/deletions that make use of the sets A and B in order to defeat any ID-oblivious insertion strategy. While $\mathcal{E}$ only has a small constant probability of occurring, this can be amplified by repeating the strategy multiple times.

As a final but crucial piece of notation, for any set S of balls present in the system, define the ***value*** $v(S)$ to be the number of balls $x \in S$ that reside in bins $1, 2$. The ultimate structure of our analysis will be to show that, if an ID-oblivious algorithm guarantees a maximum load of $m/4 + m^{o(1)}$ (with high probability), then we can construct a set S for which we can derive the clearly false assertion that $\mathbb{E}[v(S)] > |S|$.

Some basic gadgets

We will now prove a series of lemmas showing how to construct a malicious sequence of insertions/deletions using the sets A and B (and conditioned on $\mathcal{E}$). We begin by observing what happens if we simply insert the elements $A \cup B$ in a random order.

Lemma 100. *Consider a balls-and-bins game with* 4 *bins, starting from an arbitrary state. Suppose balls are allocated to bins using an arbitrary ID-oblivious insertion strategy that has already been shown the sets A, B (i.e., the algorithm can depend on the multisets $\{h(x) \mid x \in A\}$ and $\{h(x) \mid x \in B\}$). Condition on event $\mathcal{E}$, and suppose that we insert the balls $A \cup B$ in a random order. Then, after the insertions are completed, we have*

$$\mathbb{E}[v(A) - v(B)] \geq t - O(\sqrt{k}).$$

The intuition behind Lemma 100 is quite simple. For $(i,j) \in Q$, define $A_{i,j}$ (resp. $B_{i,j}$) to be the set of balls in A (resp. B) that hash to the bin pair (i,j). Due to event $\mathcal{E}$, we have that $\mathbb{E}[|A_{1,2}| - |B_{1,2}|] \geq t - O(\sqrt{k})$, so this immediately gives A an extra $t - O(\sqrt{k})$ balls (in expectation) in bins $1, 2$ that B doesn't get. On the other hand, for each $(i,j) \in Q \setminus \{(1,2),(3,4)\}$, we expect the number of balls from $A_{i,j}$ that are in bins $1, 2$ to be roughly the same as the number of balls from $B_{i,j}$ that are in bins $1, 2$, hence the conclusion of the lemma. Formalizing this argument requires some care as the algorithm can try to distinguish the balls in A from those in B based on the differences between $|A_{i,j}|$ and $|B_{i,j}|$, for $(i, j) \in Q$. Thus we defer the full proof of the lemma to Appendix 9.B.

Our next lemma makes a simple observation about what happens when we remove a set X of balls and replace it with a set X' of balls, in a balls-and-bins game that is at capacity (i.e., contains m balls).

Lemma 101. *Consider a balls-and-bins game with 4 bins, starting with m balls in the system. Let X be a set of r balls that are present. Suppose that we delete the balls X, and then insert new balls X', where $|X'| = r$. Then one of the following events must occur:*

- *there is some point in time at which some bin contains $m/4 + \omega(\sqrt{k})$ balls;*
- *or, $|v(X) - v(X')| = O(\sqrt{k})$.*

Proof. Suppose that they are never more than $m/4 + \Omega(\sqrt{k})$ balls in any given bin. This means that, whenever there are m balls in the system, the number of balls in bins $1, 2$ must be within $O(\sqrt{k})$ of $m/2$.

When we remove balls X, we decrease the number of balls in bins $1, 2$ by $v(X)$. When we insert balls X', we increase the number of balls and bins $1, 2$ by $v(X')$. In total, we must change the load of bins $1, 2$ by $O(\sqrt{k})$, meaning that $|v(X) - v(X')| = O(\sqrt{k})$. ∎

Gadget for splitting. By combining the previous two lemmas in the right way, we can construct a sequence for *splitting* a set X of size $\operatorname{poly}(k)$ into two sets Y and Z such that $v(Y) + v(Z) = (1 \pm o(k^{-1}))v(X)$ and $\mathbb{E}[v(Y) - v(Z)] \geq \Omega(|X|/\sqrt{k})$.

Lemma 102 (Splitting gadget)**.** *Consider a balls-and-bins game with 4 bins, starting from an arbitrary state with m balls, and where balls are allocated to bins using an ID-oblivious insertion strategy that, as in Lemma 100, has already been shown the sets A, B, and that keeps the load of each bin below $m/4 + O(\sqrt{k})$ w.h.p. in m. Finally, condition on event $\mathcal{E}$ with $t = c\sqrt{k}$ for some sufficiently large constant $c > 0$.*

Let X be a set of $q = k^{1.5} \log k$ balls that are currently present in the system. There exists a sequence of $\operatorname{poly}(k)$ insertions/deletions that (without ever placing more than m balls in the system at a time) replaces X with $q/2$-element sets Y, Z satisfying

$$\mathbb{E}[v(Y) - v(Z)] \geq k \log k, \tag{10.4}$$

and satisfying

$$\mathbb{E}[v(Y) + v(Z)] = v(X) \pm O(\sqrt{k}). \tag{10.5}$$

Proof. Roughly speaking, the goal is to transfer the *imbalance* between the sets A and B (in how they allocate balls to bins 1,2 vs. 3,4) to the set X, so that the resulting sets Y and Z have similar *relative* imbalance to what A and B have. Of course, A and B have size k each, while X has size $q = k^{1.5} \log k$, so the imbalance between A and B needs to be amplified in order to get the same relative imbalance between Y and Z. As we shall see, this is where we crucially make use of the ability to delete and *reinsert* $A \cup B$ multiple times.[1]

Let us partition X into sets $X_1, X_2, \ldots, X_{q/k}$ of size k each. For each $i \in [q/2k]$, we will replace X_{2i-1} by a new set Y_i and X_{2i} by a new set Z_i, in such a way that the relative imbalance between Y_i and Z_i is similar to that between A and B. This is accomplished by performing the following sequence of insertions and deletions:

1. Delete the balls $X_{2i-1} \cup X_{2i}$.

2. Insert the balls $A \cup B$ in a random order.

3. Delete the balls of A, and replace them with a set Y_i of k elements.

4. Delete the balls of B, and replace them with a set Z_i of k elements.

By Lemma 100, we have after Step (2) that

$$\mathbb{E}[v(A) - v(B)] \geq t - O(\sqrt{k}).$$

By Lemma 101 (and since the insertion strategy keeps bin loads of $m/4 + O(\sqrt{k})$ with high probability in m), we then have that $\mathbb{E}[v(Y_i)]$ and $\mathbb{E}[v(Z_i)]$ are within $O(\sqrt{k})$ of $\mathbb{E}[v(A)]$ and $\mathbb{E}[v(B)]$, respectively. Thus

$$\mathbb{E}[v(Y_i) - v(Z_i)] \geq t - O(\sqrt{k}) \geq 2\sqrt{k},$$

where the final inequality uses the fact that $t = c\sqrt{k}$ for a sufficiently large positive constant c.

Summing over $i \in \{1, 2, \ldots, q/(2k)\}$, and denoting $Y = \cup_i Y_i$ and $Z = \cup_i Z_i$, we get the claimed bound

$$\mathbb{E}[v(Y) - v(Z)] = \sum_i \mathbb{E}[v(Y_i) - v(Z_i)] \geq k \log k.$$

Next, applying Lemma 101 with $X' = Y \cup Z$, we have that either

$$v(Y) + v(Z) = v(X) \pm O(\sqrt{k}),$$

or that there is some point in time at which a bin has load $m/4 + \omega(\sqrt{k})$. Since the latter event is assumed to occur with probability at most $1/\operatorname{poly}(m)$, this completes the proof of the lemma. ∎

[1] The other place where we make use of reinsertions is that, ultimately, we will apply Lemma 102 multiple times, and we will continue to reuse A and B across those multiple applications.

Connection to marble-splitting

We are now ready to prove Theorem 97. We begin by proving a slightly weaker version of the theorem, namely that no ID-oblivious insertion strategy can offer a high-probability guarantee of achieving overload $m^{o(1)}$.

Proposition 103. *Consider the reinsertion/deletion model with* 4 *bins, and with a limit of up to* m *balls present at a time. Suppose there is an ID-oblivious bin-allocation algorithm that, for the first* $\operatorname{poly}(m)$ *steps, bounds the load of each bin by* $m/4 + f(m)$ *with high probability in* m*. Then* $f(m) = m^{\Omega(1)}$.

Proof. Set $k = m^{\varepsilon}$ for a positive constant ε to be selected later in the proof, and suppose for contradiction that $f(m) = O(\sqrt{k})$.

Let A and B be disjoint sets of k balls each. Let c be a sufficiently large positive constant, and set $t = c\sqrt{k}$. Finally, let $\mathcal{E}$ be the event that (10.2) and (10.3) hold. Note that $\mathcal{E}$ occurs with probability $\Omega(1)$; for the rest of the proof, condition on $\mathcal{E}$.

Let $X_1, X_2, \ldots, X_{ck}$ be disjoint sets of $(k^{1.5}\log k)/2$ balls each. To begin, insert m balls into the system, where those balls include $X_1, X_2, \ldots, X_{ck}$. The sets $X_1, X_2, \ldots, X_{ck}$ will act as *marbles* in a marble-splitting game. There are two types of operations that we will perform in this game: an INSERT operation, which adds one of the sets $X_1, X_2, \ldots, X_{ck}$ as a new marble in the game; and a SPLIT(X, Y) operation, which takes two sets X and Y of size $(k^{1.5}\log k)/2$ balls each, and applies Lemma 102 to replace them with sets X', Y' (also of $(k^{1.5}\log k)/2$ balls each) satisfying

$$\frac{\mathbb{E}[v(X')]}{|X'|} - \frac{\mathbb{E}[v(Y')]}{|Y'|} \geq 2/\sqrt{k}, \qquad \text{(by (10.4))}$$

$$\frac{\mathbb{E}[v(X)]}{|X|} + \frac{\mathbb{E}[v(Y)]}{|Y|} = \frac{\mathbb{E}[v(X')]}{|X'|} + \frac{\mathbb{E}[v(Y')]}{|Y'|} \pm o(1/k). \qquad \text{(by (10.5))}$$

If we define $v_X := \frac{\mathbb{E}[v(X)]}{|X|}$ for each set X of $k^{1.5}/2$ balls, it follows that we are playing a marble-splitting game with $R = \sqrt{k}$, and where marbles correspond to sets of $(k^{1.5}\log k)/2$ balls. By Proposition 99, there is an $O(R^3) = O(k^{1.5})$-step strategy that results in some marble X satisfying $v_X > 1$. This is a contradiction, since v_X must deterministically be in the range $[0, 1]$.

Note that the marble-splitting game requires $O(R^2) = O(k)$ marbles at a time, each of which consists of $O(k^{1.5}\log k)$ balls. Thus, the entire game uses $O(k^{2.5}\log k)$ balls, meaning that we can set $k = m^{1/2.5 - o(1)}$. We can therefore conclude that $f(m)$ must be at least $m^{1/5 - o(1)}$. ∎

Finally, we prove Theorem 97 by applying a basic amplification argument to Proposition 103.

Proof of Theorem 97. By Proposition 103, there exists a parameter $s \in \operatorname{poly}(m)$ such that, within $\operatorname{poly}(m)$ operations, an oblivious adversary can achieve maximum load $m/4 + m^{\Omega(1)}$ with probability $1/s$. By independently repeating this construction

$\Theta(s \log n) = \operatorname{poly}(m)$ times, the probability of achieving a load of $m/4 + m^{\Omega(1)}$ at some point during the sequence becomes

$$1 - (1 - 1/s)^{\Theta(s \log n)} = 1 - 1/\operatorname{poly}(n),$$

as desired. ∎

10.2 An Upper Bound for the Moderately-Loaded Regime

In this section, we present the ICEBERG strategy, a bin-selection rule with 3 hash functions that achieves maximum load

$$h + O(\sqrt{h \log h}) + \log \log n + O(1). \tag{10.6}$$

We begin in Subsection 10.3 by proving a useful technical lemma. Then, in Subsection 10.3.1, we present and analyze ICEBERG.

10.3 A Strong Backyarding Lemma

Consider a dynamic balls-and-bins game with n bins and at most $m = hn$ balls at all times, that are placed with the SINGLECHOICE rule. Whenever a ball is thrown into a bin, if the bin contains $h + \tau$ or more balls, then the ball is labeled as ***τ-exposed*** (and the label persists until the ball is next deleted).

Lemma 104. *Suppose* $1 \leq \tau \leq h$. *At any fixed point in time, the number of τ-exposed balls is* $\operatorname{poly}(h) \cdot ne^{-\tau^2/(3h)}$ *with probability* $1 - \exp(-\Omega(me^{-\tau^2/(3h)}))$.

We remark that Lemma 104 has a somewhat complicated history. Special cases of the lemma, but with weaker probabilistic bounds, have appeared several times in the data structures literature, first in a paper by Demaine et al. [146], and then in subsequent work by Bercea and Even [96],[2] and then again in subsequent work by a group of authors including myself [75] (with probabilistic guarantees that were stronger than those in the prior works but weaker than those here). Finally, the current version of the lemma was presented by the same group of authors in our follow-up work on tiny pointers [76] (which appears in Part V of this thesis), which included the balls-and-bins results that we will be proving in the next section. The lemma as stated is actually much stronger than we will need for our balls-and-bins result, but the strong probabilistic guarantees achieved by the lemma will end up being quite important for our data-structural applications (which use balls-and-bins as an algorithmic subroutine) later in the thesis (see Part V).

[2] As noted by [75], however, the proof in [96] contained a subtle but serious bug.

Our proof of the lemma will make use of a variant of Talagrand's inequality [270, Chapter 12]:

Theorem 105 (Talagrand's inequality)**.** *Let $X_1, \dots, X_n$ be n independent random variables from an arbitrary domain. Let F be a function of $X_1, \dots, X_n$, not identically 0. Suppose that for some $c, r > 0$, F is c-Lipschitz and r-certifiable, defined as follows:*

- *F is c-Lipschitz if changing the outcome of any single X_i changes F by at most c.*
- *F is r-certifiable if, for any s, if $F(X_1, \dots, X_n) \geq s$, then there is a certifying set of at most rs X_i's whose outcomes serve as a witness that $F \geq s$, that is, $F \geq s$ no matter the outcome of the other X_j not in the certifying set.*

Then, for any $0 \leq t \leq \mathbb{E}[F]$,

$$\Pr\left(|F - \mathbb{E}[F]| > t + 60c\sqrt{r\,\mathbb{E}[F]}\right) \leq 4\exp\left(-\frac{t^2}{8c^2 r\,\mathbb{E}[F]}\right).$$

The proof of the lemma proceeds by bounding the expected number of exposed balls, then using Talagrand's inequality to achieve a concentration bound.

In what follows, we refer to the balls which are present at the end as $a_1, \dots, a_k$ and we refer to the remaining balls in the universe as $a_{k+1}, \dots, a_\ell$. We denote by α_i the bin choice for a_i. For $i \in [k]$, we define t_i to be the last time at which a_i is inserted, we define X_i to be the random variable indicating if a_i is an exposed ball at the end of the game, and we define $X = \sum_{i=1}^k X_i$ to be the total number of exposed balls.

Claim 106. *The expected number of exposed balls satisfies $\mathbb{E}[X] = O(me^{-\tau^2/(3h)})$.*

Proof. Recall that $X = \sum_i X_i$ where X_i indicates whether a_i is exposed. By linearity of expectation, it suffices to show that $\mathbb{E}[X_i] = O(e^{-\tau^2/(3h)})$ for each $i \in [k]$.

Fix $i \in [k]$. Consider the final time t_i at which ball a_i is inserted. The ball a_i is exposed if and only if the number of balls in bin α_i is at least $h + \tau$. If we set Y to be the number of balls in bin α_i, and we set $\varepsilon = \tau/h$, then we can bound the probability of $Y \geq h + \tau$ using a Chernoff bound:

$$\Pr(Y \geq h + \tau) = \Pr(Y \geq (1 + \varepsilon)h) \leq e^{-\varepsilon^2 h/3} = e^{-\tau^2/(3h)}.$$

Thus $\Pr(X_i) = e^{-\tau^2/(3h)}$. ■

Claim 107. *The random variable X is $(h+\tau+1)$-Lipschitz and $(h+\tau+1)$-certifiable as a function of $\{\alpha_i\}_{i=1}^{\ell}$.*

Proof. Changing the value of a single α_i to α_i' can only affect the number of exposed balls in bin α_i (which may decrease) and in bin α_i' (which may increase). The number of *unexposed* balls in a bin is deterministically at most $h+\tau$. This means that moving ball a_i out of bin α_i can increase the number of unexposed balls in the bin by at most

$h+\tau$, and thus can decrease the number of exposed balls by at most $h+\tau+1$ (where the $+1$ accounts for the removal of a_i itself). Similarly, moving ball a_i into bin α'_i can decrease the number of unexposed balls in the bin by at most $h+\tau$, and thus can increase the number of exposed balls by at most $h+\tau+1$. This establishes that X is $(h+\tau+1)$-Lipschitz.

To certify that $X \geq s$, let J with $|J| = s$ be a set of values $j \in [k]$ such that a_j is exposed at the end of the game. For each $j \in J$, let R_j be a selection of $h+\tau$ balls i such that ball a_i was present at the last time t_j that a_j was inserted and such that $\alpha_i = \alpha_j$. The set of random variables $\{\alpha_i \mid i \in R_j\} \cup \{\alpha_j\}$ acts as a certificate that a_j is exposed. Thus the set

$$\bigcup_{j \in J} \{\alpha_i \mid i \in R_j\} \cup \{\alpha_j\}$$

acts as a certificate that $X \geq s$. This certificate consists of $s(h+\tau+1)$ random variables, hence X is $(h+\tau+1)$-certifiable. ∎

Proof of Lemma 104. Set $Q = m\exp(-\tau^2/(3h))$. By Claim 106, we know that $\mathbb{E}[X] \leq Q$. By Claim 107, we can apply Talagrand's inequality (Theorem 105) to X with $c = r = h+\tau+1 = O(h)$. Applying Talagrand's inequality with $t = \Theta(c\sqrt{r}Q)$, and using Q as an upper bound on $\mathbb{E}[X]$, we can deduce that

$$X = O(c\sqrt{r}Q)$$

with probability at least

$$1 - \exp(-\Omega(Q)).$$

It follows that $X \leq \text{poly}(h) \cdot O(ne^{-\tau^2/(3h)})$ with probability $1 - \exp(-\Omega(me^{-\tau^2/(3h)}))$. ∎

10.3.1 The Iceberg 3-Choice Strategy

We now present the Iceberg balls insertion strategy, which uses three bin choices for each ball in order to achieve an overload of $o(m/n) + O(\log\log n)$.

Let n be the number of bins, and let $m = hn$ be the maximum number of balls allowed to be present at any given moment. Let g, h_1, h_2 be hash functions mapping balls to uniformly random bins.

We shall have three types of balls: level-one balls, level-two balls, and level-three balls. Each level-one ball x will reside in bin $g(x)$, each level-two ball x will reside in one of bins $h_1(x), h_2(x)$, and each level-three ball x will reside in an arbitrary bin (but, at any given moment, the number of level-three balls will be zero w.h.p.).

Set $\tau = c\sqrt{h\log h}$ for some sufficiently large positive constant c. We shall also keep track of a variable q counting the number of level-two balls present at any given moment.

The procedure for inserting a ball x is as follows. If bin $g(x)$ contains $h + \tau$ level-one balls or fewer, then we place x in bin $g(x)$, and we classify x as a level-one ball. Otherwise, we check whether $q < n$. If $q < n$, then we examine bins $h_1(x), h_2(x)$, and we place x as a level-two ball into whichever bin $h_i(x)$ contains the fewest level-two balls (breaking ties arbitrarily). Finally, if $q \geq n$, then we place x as a level-three ball into an arbitrary bin.

Theorem 98. *Suppose $1 \leq h \leq n^{o(1)}$. Suppose balls are inserted/deleted/reinserted into n bins over time (by an oblivious adversary) according to the* ICEBERG *rule, where each ball has three random choices for where it can go, and where there are never more than hn balls present at a time. Then, w.h.p. in n, at any given moment, the number of balls in the fullest bin is $h + O(\sqrt{h \log h}) + \log \log n + O(1)$.*

Proof. Each bin deterministically contains at most $h + \tau = h + O(\sqrt{h \log h})$ level-one balls. Thus, it suffices to bound the number of level-two and level-three balls in each bin by $\log \log n + O(1)$.

The number q of level-two balls in the entire system is deterministically at most n at any given moment. In other words, the level-two balls are placed according to GREEDY two-choice strategy with up to n balls at a time. This implies [350,351,369] that the number of such balls in any given bin is at most $\log \log n + O(1)$ with high probability in n.

We complete the proof by showing that, w.h.p., The number of level-three bins is zero. By Lemma 104, the number q of level-two balls satisfies $q < n$ (at any given moment) with probability at least $1 - \exp(-n/\operatorname{poly}(h))$, which by the assumption $h \leq n^{o(1)}$ is at least $1 - 1/\operatorname{poly}(n)$. It follows that each individual ball insertion has probability at most $1/\operatorname{poly}(n)$ of being level-three. Taking a union bound over all of the balls in the system, the probability that any of them are level-three is at most $1/\operatorname{poly}(n)$, as desired. ∎

Part IV

Hashing it Out: Some Barriers Are Fundamental and Others Are Not

Chapter 11

Introduction

In this part of the thesis, we revisit three well-studied problems from the areas of hash tables and hashing:

- **Chapter 12:** The optimal tradeoff curve between time- and space-efficiency in a hash table.
- **Chapter 13:** The strongest probabilistic guarantees that a hash table can offer.
- **Chapter 14:** The optimal space efficiency of a monotone minimal perfect hash function.

Past research on these problems has hit what would seem to be natural barriers. However, it has remained unclear whether these barriers are fundamental, or whether they are simply limitations of the currently known techniques. This is the question that we seek to answer in the next three chapters.

What we will see in Chapters 12 and 13 is that, without new algorithmic techniques, we are able to achieve significantly stronger guarantees than were previously known to be possible.

Our hash table in Chapter 12 establishes a very strong tradeoff between time and space—with $O(1)$-time queries and $O(k)$-time insertions/deletions, one can get within

$$O(n \log^{(k)} n) = O\left(n \underbrace{\log \log \cdots \log}_{k} n\right)$$

bits of the information-theoretic space requirement that any hash table must satisfy. This bypasses a barrier of $O(n \log \log n)$ bits that had previously stood as the state of the art for close to two decades [313], and that was known to be optimal for any *stable* hash table [75,146]. The result reveals that, perhaps surprisingly, *unstable* hash tables (i.e., hash tables that strategically rearrange elements on each insertion/deletion) can achieve fundamentally stronger guarantees than their stable counterparts.

Our hash table in Chapter 13 stores $\Theta(\log n)$-bit key/value pairs while guaranteeing *worst-case* $O(1)$-time operations with an extremely high probability of $1 - 1/2^{n^{1-\varepsilon}}$.

The previous states of the art [75, 198, 199] had gotten stuck at $1 - 1/2^{\operatorname{polylog} n}$ and had identified what seemed to be a serious bottleneck: no matter how well designed the hash table itself was, the known techniques for simulating random hash functions were *themselves* dominating the failure probability. What makes our approach in Chapter 13 interesting is that it bypasses this barrier entirely—rather than relying on the traditional hash-function framework, our hash table uses randomization in a more explicit (and unusual) way in order to directly obtain strong probabilistic guarantees.

In Chapter 14, on the other hand, we find that there actually is a fundamental barrier. We show that monotone minimal perfect hash functions necessarily require $\Omega(n \log\log\log u)$ bits to encode, where n is the number of items being hashed and u is the universe size. This matches an upperbound given by Belazzougui, Boldi, Pagh and Vigna [64] in 2009, and establishes that the triple-logarithmic dependence on u is actually information-theoretic, rather than being an artifact of the known techniques.

Because these three problems each represent a different line of research, we will defer more detailed descriptions of the results to the individual chapters. Moreover, in order to accommodate readers who may have targeted interests, we have written each chapter to be completely self-contained.

Chapter 12

The Optimal Space-Time Tradeoff Curve for Hash Tables

12.1 Introduction

Formally, a ***hash table*** [226] (sometimes called a ***dictionary***) is a data structure that stores a set of keys from some key-universe U and that supports three operations on that set: insertions, deletions, and queries. Some hash tables are also capable of storing a ***value*** $v \in V$ associated with each key. In this case, a query on a key k returns both whether key k is present and what the associated value v is, if k is present.

Since hash tables were introduced in 1953, there has been a vast literature on the question of how to design space- and time-efficient hash tables [50,52,75,146,153,156, 156,181,185,224,226,245,296,300,304,313,370]. Whereas early hash tables [224,226] required $\omega(1)$ time per operation in order to support a load factor of $1-o(1)$, modern hash tables [52, 75, 245] offer a much stronger guarantee. Not only are these hash tables constant time (with high probability), and not only do they support a load factor of $1-o(1)$, but they have even converged towards the *information-theoretically optimal* number of bits of space, given by

$$\mathcal{B}(U,V,n) = \log \binom{|U|}{n} + n \log |V|.$$

A hash table that uses $\mathcal{B}(U,V,n) + rn$ bits of space is said to incur r ***wasted bits per key***. When $\log|U| + \log|V| = c \log n$ for some constant $c > 1$, the state of the art for r is $O(\log\log n)$, which was first achieved in 2003 by Raman and Rao [313] with constant expected-time operations, and which after a long line of work [52, 75, 146, 245] has now also been achieved [75] with (high-probability) worst-case constant-time operations.

Besides having remained the state of the art for nearly two decades, there are several more fundamental reasons to believe that $r = O(\log\log n)$ might be optimal. It is known that $\Theta(\log\log n)$ wasted bits per key is optimal for the closely related problems of dynamic value retrieval[1] [146, 157, 273] and fully-dynamic approximate set membership[2] [122,245,301]. And it is known that *stable* hash tables [75,146] (i.e., hash tables in which each key/value pair is assigned a fixed and unchanging position upon arrival) have an optimal value of $r = \Theta(\log\log n)$.

Nonetheless, it is *not known* whether $r = O(\log\log n)$ wasted bit per key is optimal for dynamic constant-time hash tables. More generally, it is an open question what the optimal tradeoff is between time and space (e.g., can slightly super-constant-time

[1]A dynamic data-retrieval data structure is a hash table with the added restriction that queries must be for keys/value pairs that are present. If keys are from a universe of size $\text{poly}(n)$ and v is the size of each value in bits, then static value-retrieval requires $nv + o(n)$ bits [157], and dynamic value-retrieval requires $nv + \Theta(n \log\log n)$ bits [146, 273].

[2]Fully-dynamic approximate set membership data structures, also known as dynamically-resizable filters, are analogous to dynamically-resizable hash tables, but with some ε probability of queries returning false-positives. Whereas an optimal static filter requires $n \log \varepsilon^{-1}$ bits [122], an optimal resizable filter requires $n \log \varepsilon^{-1} + \Omega(n \log\log n)$ space [301], which is known to be optimal for $\varepsilon^{-1} \le \text{polylog}\, n$ [245, 301].

operations yield major space savings?).

The optimal tradeoff curve between time and space. In this chapter, we present a data structure that achieves a much stronger time/space tradeoff. We also give a matching lower bound that holds across a large class of data structures. In fact, subsequent work by Li, Liang, Yu, and Zhou [243] has shown that our tradeoff curve is *optimal across all dynamic data structures.* This closes off one of the longest-standing directions of research in the field of data structures.

For any parameter $k \in [\log^* n]$, we construct a hash table that supports constant-time queries, that supports $O(k)$-time insertions/deletions, and that incurs

$$O(\log^{(k)} n) = O\left(\underbrace{\log \log \cdots \log}_{k} n\right)$$

wasted bits per key, where the guarantees on time and space are worst-case with high probability in n. Our result holds not just for fixed-capacity hash tables, but also for dynamically-resizable hash tables, as well as for hash tables storing very large keys/values (up to $n^{o(1)}$ bits each).

Our result implies a remarkably steep tradeoff: each time that we increase insertion/deletion time by an *additive constant*, we are able to *exponentially reduce* the number of wasted bits per key. In particular, we obtain a hash table that supports $O(1)$-time insertions/deletions/queries with $O(\log^{(c)} n)$ wasted bits per key, for any positive constant c of our choice; and we obtain a hash table that supports $O(\log^* n)$-time insertions/deletions with $O(1)$ wasted bits per key and constant-time queries.

Finally, in the special case where keys/values are small, meaning that each key-value pair consists of $\log n + o(\log n / \log^{(k)} n)$ bits (but keys are still from a universe of size $\omega(n)$), we are able to further tighten our bounds to obtain $o(1)$ wasted bits per key. Building on this, we obtain a dynamic constant-time approximate set-membership data structure (i.e., a filter) that achieves space

$$n \log \varepsilon^{-1} + n \log e + o(n)$$

bits, for a wide choice of false-positive rates ε, resolving a long-standing open problem as to whether $O(1)$ wasted bits per key is achievable by dynamic filters. In fact, not only is $\log e + o(1)$ constant, but it is the provably optimal number of wasted bits per key for any filter that is constructed by storing fingerprints in a hash table [82, 93, 122, 245, 294].

12.2 Overview of Results and Techniques

This section presents an overview of the main results and techniques in the chapter.

12.2.1 Hash Tables and Balls-To-Slots Schemes

An implicit theme in the design and analysis of hash tables is that the problem of constructing a space-efficient hash table is closely related to the problem of placing balls into slots of an array. We now formalize this relationship by defining the class of ***augmented open-addressed hash tables*** (which includes all known succinct constant-time hash tables [52,75,82,93,245,313]), and by formally defining the balls-to-slots problem that any augmented open-addressed hash table must solve (we will call this problem the ***probe-complexity problem***). Later, in Section 12.3, we will give tight upper and lower bounds for the probe-complexity problem, which will then allow for us to construct optimal augmented open-addressed hash tables.

Augmented open addressing. Augmented open-addressed hash tables are hash tables that abide by the following basic framework: elements are stored in an array of some size $m = (1+\varepsilon)n$, and each element $x \in U$ is assigned a probe sequence $h_1(x), h_2(x), h_3(x), \ldots \in [m]$ of array slots where it can be stored; queries are then implemented using a secondary query-routing data structure which, for each key x, stores the index i of the position $h_i(x)$ containing the key.[3] If a hash table is to be succinct, it must simultaneously achieve $\varepsilon = o(1)$ (we call the quantity $1-\varepsilon$ the ***load factor***), while also ensuring that the quantities stored by the query-routing data structure don't take up too many bits (i.e., keys are in positions $h_i(x)$ for relatively small values of i).

The use of a probe sequence to determine where a key can reside is analogous to classical open addressing [226]. An important difference is that the query-routing data structure allows for queries to be performed in constant time, without needing to scan through the positions $h_1(x), h_2(x), \ldots$.

Of course, there is flexibility in terms of what granularity augmented open addressing is used at. For example, in order to support dynamic resizing [75,245,313], a hash table might use augmented open addressing on bins of size $\text{polylog}\, n$, and then use a different set of techniques to determine which bin each key should go into. Nonetheless, all known succinct constant-time hash tables [52,75,82,93,245,313] rely on some form of augmented open-addressing as the highest-granularity abstraction layer in which elements are stored.

The probe complexity problem. We can formalize the balls-to-slots problem that any augmented open-addressed hash table must solve as follows. Each key x is thought of as a "ball" that is associated with some probe sequence $h(x) = \langle h_1(x), h_2(x), \ldots, h_m(x)\rangle$ where without loss of generality the sequence is a permutation of $\langle 1, 2, \ldots, m\rangle$. A balls-to-slots scheme must support an (online) sequence of ball insertions/deletions so that, at any given moment, the up to n balls that are present are each assigned distinct positions in an array of $m = (1+\varepsilon)$ slots. The balls-to-slots scheme is measured by two objectives: the average ***probe complexity*** of the balls,

[3]Additionally, a technique known as "quotienting" is used to shave $\log n$ bits off of each key—this is what bridges the gap between using $\mathcal{B}(U,V,n)+nr$ space instead of $n\log|U|+n\log|V|+nr$ space.

which for a ball x in slot $h_i(x)$ is given by $(1 + \log i)$; and the ***switching cost*** of the balls-to-slots scheme, which is the number of balls that the scheme rearranges on each insertion/deletion (including the ball being inserted/deleted).

The probe complexity of each ball x can be viewed as the minimum number of bits (asymptotically) that must be stored in the query router in order for the position of the ball to be recovered by queries. The switching cost, on the other hand, can be viewed as (a lower bound on) the amount of time that it takes to implement a ball insertion/deletion. Thus lower bounds on the relationship between probe complexity and switching cost in the probe-complexity problem directly translate to lower bounds on the relationship between space and insertion-time in augmented open-addressed hash tables.

Intuitively, there are three challenges to designing an augmented open-addressed hash table: one must construct a balls-to-slots scheme with low probe complexity, low switching cost, and high load factor; one must efficiently implement that balls-to-slots scheme so that insertions/deletions can (ideally) be performed in time proportional to the switching cost; and one must implement a query-routing data structure that maps each key x to the slot $h_i(x)$ where it resides (ideally, this should use space proportional to the probe complexity of x). Thus the problem of determining the optimal tradeoff between average probe complexity and switching cost is central to the problem of designing an optimal augmented open-addressed hash table.

The two approaches to designing balls-to-slots schemes. One way to design a balls-to-slots scheme is to base it on a traditional open-addressed hash table such a linear probing [226], double hashing [226], or Cuckoo hashing [300]. Cuckoo hashing [300] (and its variants [50, 160, 181]) is especially appealing because it bounds probe complexity deterministically. Standard Cuckoo hashing achieves a probe complexity of $O(1)$, but is only able to support a load factor of $1 - \varepsilon < 1/2$. Generalizations of Cuckoo hashing (i.e., d-ary Cuckoo hashing [181] and Cuckoo hashing with d-slot bins [160]) are able to support higher load factors $1-\varepsilon$, but at the cost of incurring a super-constant switching cost of at least $\Omega(\log \varepsilon^{-1})$. Thus Cuckoo hashing cannot be used on its own to obtain a succinct constant-time hash table (although it has been used in past work as an essential building block [52]).

To achieve small probe complexity and switching cost, while also supporting $\varepsilon = o(1)$, past work has used balls-to-slots schemes that are based on standard balls-to-bins techniques. One simple approach is to set $m = (1 + 1/\log n)n$; to partition the array of size m into bins of size $\ell = \text{polylog}\, n$; and finally to hash each key to a random bin $g(x) \in \{0, 1, 2, \ldots, m/\ell - 1\}$ and set $h_i(x) = g(x) \cdot \ell + i$ for all i. With high probability in n, every key will find a free position in the bin that it hashes to, meaning that each key is assigned to one of its first $\ell = \text{polylog}\, n$ choices. This scheme achieves load factor $1 - 1/\log n$, worst-case probe complexity $O(\log \log n)$, and worst-case switching cost 1 (with high probability in n).

It's natural to hope that an even better probe complexity could be achieved by making use of more sophisticated balls-to-bins schemes (e.g., power of two choices [267]). This turns out not to be possible, as one can prove a lower bound of $\Omega(\log \log n)$

average probe complexity for *any* balls-to-slots scheme with switching cost 1 and load factor $1-1/\log n$; in fact, this is a special case of a more general lower bound [146,273] which says that ***stable hash tables*** (i.e., hash tables in which elements are assigned permanent positions when they are inserted) must incur $\Omega(\log\log n)$ wasted bits per key.

The central bottleneck to designing augmented open-addressed hash tables that make use of this simple balls-to-slots scheme has been the issue of achieving constant-time operations while preserving space efficiency [52, 75, 82, 93, 245, 313]. Raman and Rao [313] gave an elegant solution with constant expected time in which the query-routing data structure is itself a collection of small hash tables that store fingerprints of keys. The bottleneck to achieving the same guarantee with worst-case time bounds has been, until recently, the difficulty of constructing efficient query-routing data structures for bins of polylog n elements—this led researchers to develop more sophisticated balls-to-slots schemes that make use of smaller bins [52, 82, 93, 245], which allowed for them to overcome the query-routing bottleneck, but resulted in a worse space utilization. Recently, [75] resolved this issue by showing how to perform query-routing on bins of size polylog n while incurring only $O(\log\log n)$ extra wasted bits per key.

In summary, it is known how to use the balls-to-bins framework for the probe-complexity problem in order to achieve $O(\log\log n)$ wasted bits per key, and this results in an optimal stable hash table. It has not been known whether the ability to move keys around during insertions opens the door to even higher space efficiency.

An optimal solution to the probe-complexity problem. An essential technical insight in this chapter is that one can achieve an extremely small average probe complexity by moving around just a few balls on each insertion. We present a balls-to-slots scheme, called the ***k-kick tree***, that achieves average probe complexity $\log^{(k)} n$ while achieving a worst-case switching cost of $O(k)$. Moreover, this result holds even when the number n of balls equals the number m of slots, so the load factor is 1.

We prove that this tradeoff between switching cost and probe complexity is asymptotically optimal (as long as the load factor $1-\varepsilon$ is at least, say, $1-1/\log\log n$). In particular, if a balls-to-slots scheme achieves average probe complexity $O(\log^{(k)} n)$, it must move an average of $\Omega(k)$ items per insertion/deletion.

Interpreting our lower bound as a statement about augmented open-addressed hash tables, we can conclude that the hash tables constructed in this chapter are optimal in the class of augmented open-addressed hash tables. In fact, subsequent work has shown an even stronger claim [243]: that our hash table is optimal across all dynamic data structures.

12.2.2 Transforming a k-Kick Tree into a k-Kick Hash Table

The k-kick tree serves as the balls-to-slots scheme for all of the hash tables that we construct in this chapter, but existing techniques for constructing the other parts of an augmented open-addressed hash table, (e.g. the query router, how to dynamically resize, etc.) are not themselves space and time efficient enough to fully take advantage of the efficiency of k-kick trees. Next, we summarize the main technical obstacles that we overcome in order to use k-kick trees time and space efficiently in our hash tables.

An improved query router (Section 12.4). We show how to build general-purpose query-routing data structures with strong space and time guarantees. Even if different keys have very different probe complexities from one another, our query-routing data structure uses space within a constant factor of optimal and supports constant-time queries/updates. The building blocks that we use to construct the query-router will likely also be useful in future work on related problems.

Supporting dynamic resizing (Section 12.5.2). Past approaches [75, 245, 313] have performed resizing at a granularity of $1+1/\operatorname{polylog} n$ factors.[4] This has required the data to be partitioned into $\operatorname{polylog} n$ chunks, and for the query-router to store an additional $\Theta(\log\log n)$ bits associated with each key in order to identify its chunk. In other words, dynamic resizing introduces yet another source of $\Theta(\log\log n)$ wasted bits per key. We give a general-purpose technique for avoiding this type of overhead—surprisingly, the technique results in the *same* tradeoff curve that we encounter for probe-complexity: at the cost of $O(k)$ time per insertion/deletion, we can reduce the space overhead of resizing to $O(\log^{(k)} n)$ bits per key.

Handling large keys/values (Section 12.6.1). Now consider the setting where the keys and values are u and v bits long, respectively, for some potentially large u, v satisfying $u + v \leq n^{o(1)}$. Past techniques have encountered several major obstacles in this case, resulting in the wasted space per key growing substantially as the key size u becomes super-logarithmic [52, 245, 313]. The only known succinct hash table that scales gracefully in the regime of $u + v = \omega(\log n)$ is the hash table of Raman and Rao [313], which achieves $O(\log(u + v))$ wasted bits per key with constant expected-time insertions—subsequent work [52,245] on worst-case insertion times has encountered much larger space blowups due to technical difficulties surrounding the use of quotienting and the use of lookup-tables in hash tables with large keys.

Our approach to handling large keys and values is to give a general-purpose reduction from the setting where $u \geq \omega(\log n)$ to the setting where $u = O(\log n)$. In essence, our reduction allows for us to move bits from the key length u to the value-length v.

We then show how to adapt our hash tables to support arbitrarily large values with *no additional space wastage*. Here, we exploit a special property of the k-kick tree, namely that it is capable of supporting a load factor of 1, which ends up allowing for us to construct a dynamically-resized hash table in which there are *no empty slots*.

Combining these techniques, we conclude that the tradeoff curve in this chapter

[4]The specific ways in which resizing has been implemented have differed, with some papers [245, 313] performing resizing at a per-bin level, and others [75] performing it globally.

is agnostic to key/value size: with $O(k)$-time per insertion/deletion, we can achieve $O(\log^{(k)} n)$ wasted bits per key.

Handling small keys/values (Section 12.6.2). In addition to considering large keys, past work [50, 146, 245, 313] has also focused in on the small case, where $u + v = \log n + t$ for some $t = o(\log n)$ (and where the universe $U = [2^u]$ of keys may have an arbitrarily small size satisfying $|U| = \omega(n)$). In this setting, we show that if t is even *slightly* sublogarithmic, that is,

$$t = O(\log n / \log^{(k)} n)$$

for some positive constant k, then it is possible to support constant-time insertions/deletions/queries while achieving $o(1)$ wasted bits per key. Prior to our work, this type of guarantee was only known to be possible in the much smaller regime of $t = \tilde{O}((\log n)^{1/3})$ [52, 313]. What makes our expanded range for t interesting is that, as we shall see shortly, it enables us to design optimal dynamic filters for a large range of false-positive rates (in fact, for all false-positive rates except for those that are nearly polynomially small).

Our small-key result again follows from a general-purpose reduction, which in this case reduces the setting of small keys/values to the setting of larger keys/values. Interestingly, this reduction relies heavily on the ability to efficiently support dynamic-resizing and on the steep tradeoff curve between time/space for standard-sized keys/values.

12.2.3 An Application to Optimal Dynamic Filters

Finally, in Section 12.6.3, we apply our small-keys result to the widely studied problem of maintaining space-efficient approximate-membership data structures, also known as filters. A (dynamic) ***filter*** is a data structure that supports inserts/queries/deletes on a set of keys but that is permitted to return a false positive on a query with some probability ε. Information theoretically, a filter must use at least $\mathcal{F}(n, \varepsilon) = n \log \varepsilon^{-1}$ bits [122].

It remains an open question what the optimal achievable wasted-bits-per-key is, that is, what is the smallest value of r such that it is possible to construct a time-efficient dynamic filter using $\mathcal{F}(n, \varepsilon) + nr$ bits. We remark that, here, n is taken to be a fixed upper bound on the number of keys—if n is permitted to change dynamically, then it is known that the optimal r satisfies $r = \Omega(\log \log n)$ [301].

Filters tend to be used in applications where space efficiency is a central concern; the result is that most applications select ε such that $\log^{-1} \varepsilon$ is very small (for a practical discussion of filters, see, e.g., [83, 163, 175]). This leads to the close relationship between the filter problem and the hash-table problem with small keys.

Perhaps the most famous filter is the so-called Bloom filter [107], which supports $O(\varepsilon^{-1})$-time insertions and achieves $r = O(\log \varepsilon^{-1})$ (the Bloom filter does not support deletions). After a long line of work [82, 93, 107, 122, 245, 294], contemporary filters are able to achieve much stronger bounds than this. Indeed, there are now a number of filters [82, 93, 294] that and that achieve

$$r = o(\log \varepsilon^{-1})$$

wasted bits per key for all ε satisfying

$$\log \varepsilon^{-1} \in [\omega(1), O(\log n)],$$

while supporting constant-time insertions/deletions/queries either in expectation [294] or in the worst case [75, 93] (with high probability).

The central open question in the study of filters is whether it is possible to achieve $r = O(1)$ wasted-bits-per-key for all ε. It is known that $\Omega(1)$ wasted-bits-per-key are *necessary*, at least for some values of ε [249] (namely, $\varepsilon = \Theta(1)$), but it is not known whether $O(1)$ wasted-bits-per-key is *achievable*.

We show that, for any positive constant k, it is possible to achieve a filter that uses space

$$r = \log e + o(1) = O(1) \tag{12.1}$$

wasted bits per key for all inverse-power-of-two ε satisfying

$$\log \varepsilon^{-1} \in [\omega(1), \log n / \log^{(k)} n],$$

while supporting worst-case constant-time insertions/deletions/queries (with high probability). The total space used by the data structure is therefore

$$n \log \varepsilon^{-1} + n \log e + o(n)$$

bits. This resolves the question of whether $r = O(1)$ is achievable in all cases except for when $\log^{-1} \varepsilon$ is very close to $\log n$. Finally, we show that for any value of $\log^{-1} \varepsilon$ (including $\log^{1} \varepsilon = \Theta(\log n)$), the same time/space tradeoff curve that we achieve for hash tables, in which $O(k)$-time insertions/deletions yield $O(\log^{(k)} n)$ wasted bits per key, is achievable for dynamic filters.

We remark that the specific constant $\log e$ that we achieve in Equation (12.1) is information-theoretically optimal for any filter that is constructed by storing fingerprints in a hash table. (This includes all modern dynamic filters [82,93,122,245,294].) Thus, improving upon this constant would require a fundamentally new approach to building constant-time filters. We conjecture that no such improvements are possible (even for non-constant-time dynamic filters)—proving a lower bound for this claim is an appealing direction for future work.

12.2.4 Preliminaries

We conclude the section by formalizing several preliminaries that we will need throughout the chapter.

Notation. We use $[i, j]$ to denote the range $\{i, \ldots, j\}$, we use $[i]$ to denote $[1, i]$, and we use $\log^{(i)} n$ to denote the function given by $\log^{(0)} n = n$ and $\log^{(i)} n = \max(\log \log^{(i)} n, 1)$ for all $i \geq 0$. Note that, as a matter of convention, we do not allow $\log^{(i)} n$ to become sub-constant.

High-probability guarantees. We say that an event occurs with high probability (w.h.p.) in n if it occurs with probability $1 - 1/\operatorname{poly}(n)$. Our hash tables will offer a deterministic guarantee on the running times of queries, a high-probability guarantee on the running time of any given insertion/deletion, and a high-probability guarantee on the space consumption at any given point in time. To simplify discussion, we will allow for our hash tables to have alternative failure modes (e.g., some bin overflows), with the implicit assumption that whenever a low-probability failure event occurs during an insertion/deletion, the hash table is then be rebuilt from scratch using new randomness—this means that failure events cause the hash table to violate time/space guarantees, but not correctness guarantees.

Simulating fully random hash functions. Whereas early work on hash tables [153, 156, 185] was bottlenecked by the known families of hash functions, there are now well-established techniques [50,75,162,291,326] for simulating fully random hash functions in hash tables. Notably, Siegel [326] showed that for some positive $\varepsilon > 0$, there is a family of constant-time hash functions that can be constructed in time $o(n)$ and that is n^{ε}-independent.[5] In the context of hash tables, this can be amplified to simulate $\operatorname{poly}(n)$-independence [50, 75] with the following "sharding" technique: use a hash function h_1 to partition the elements into buckets with sizes in the range $[n^{\delta}, n^{\delta} + n^{2\delta/3}]$; then implement each bucket as its own hash table, where the all of the buckets share access to a single n^{ε}-independent family $\mathcal{H}$ of hash functions—if a given bucket has size $m = \Theta(n^{\delta})$, then $\mathcal{H}$ is $\operatorname{poly}(m)$-independent. Thus we can assume without loss of generality that we have access to $\operatorname{poly}(n)$-independent hash functions.

In Section 12.5.3, in order to perform quotienting, we will also want access to random *permutation* hash functions, that is, hash functions h that are bijective on some universe U of keys. As long as $|U| \leq \operatorname{poly}(n)$, then there are again well-established techniques for simulating full randomness. Naor and Reingold (Corollary 8.1 of [277]), building on seminal work by Luby and Rackoff [250], showed how to construct in time $o(n)$ an n^{ε}-wise $1/n^{\delta}$-dependent family of permutations with constant-time evaluation. Subsequent work showed how to amplify this to n^{ε}-wise $1/\operatorname{poly}(n)$-dependence, which allows for the simulation of n^{ε}-wise independence with probability $1 - 1/\operatorname{poly}(n)$. Finally, using a similar sharding technique as described above (but with h_1 implemented using a single-round Feistel permutation, as in [50]), one can use such a family of hash functions to simulate $\operatorname{poly}(n)$-independence in a hash table (see Section 7 of [75] for an in-depth discussion). Thus, as long as $|U| \leq \operatorname{poly}(n)$, then we can assume with-

[5]Siegel's construction requires that the universe U of keys has at most polynomial size—but it can also be used with a larger universe by first performing dimension reduction to a $\operatorname{poly}(n)$-size universe using a pairwise independent hash function.

out loss of generality that we have access to poly(n)-independent permutation hash functions.

Machine model. Our analyses will be in the standard RAM model. If keys/value pairs are each w bits long, then we shall assume a machine word of size at least w. To analyze space consumption, we will assume that algorithms have the ability to allocate/free memory with $O(\log n)$-bit pointers. We remark, however, that all of our algorithms have highly predictable allocation patterns, and are therefore straightforward to implement using a small number of large memory slabs (e.g., when keys/values are $\Theta(\log n)$ bits, we need only to allocate polylog(n) slabs of memory at a time).

12.3 The Probe-Complexity Problem

Recall that the probe-complexity problem can be defined formally as follows. Let U be a universe of balls, and let $n \in \mathbb{N}$ be the number of slots[6], and let $\varepsilon \in [0, 1)$ be a load-factor parameter. A ***balls-to-slots scheme*** assigns to every ball $x \in U$ a fixed ***probe sequence*** $h(x) = \langle h_1(x), h_2(x), \ldots \rangle$, each $h_i(x) \in [n]$.

We define the ***probe-complexity problem*** as follows. There are n ***slots*** each with capacity 1. An oblivious adversary (who does not know h) selects a sequence of ball insertions/deletions such that at most $(1 - \varepsilon)n + 1$ balls are present at a time. A balls-to-slots scheme must maintain an assignment of balls to slots such that each ball x (that is present) is assigned to slot $h_i(x)$ for some i. If a ball x is in slot r, then we say that x has ***probe complexity*** $\Theta(1 + \log i)$ where $i = \operatorname{argmin}_j\{h_j(x) = r\}$.

To simplify discussion, we shall also give the balls-to-slots scheme n extra ***special slots***. Any ball that is stored in a special slot automatically has probe complexity $\log n$. Whenever a ball insertion occurs, the ball is first placed into a special slot. The balls-to-slots scheme can then move that ball (and other balls) around in order to reduce the average probe complexity of the balls that are present.

The balls-to-slots scheme is measured by two objectives: the average probe complexity of the balls that are present; and the ***switching cost***, which is the number of balls that the balls-to-slots scheme moves around on any given insertion/deletion. When a balls-to-slots scheme is used in an augmented open-addressed hash table, the switching cost is (a lower bound on) the time spent on a given insertion/deletion, and the probe complexity of a key x is (a lower bound on) the number of metadata bits that must be stored to support constant-time queries for x.

In this section, we give an optimal solution to the probe-complexity problem (Subsection 12.3.1), achieving probe-complexity $O(\log^{(k)} n)$ with switching cost $O(k)$. This holds even when $\varepsilon = 1/n$, meaning that there are up to n balls present at a time

[6]Whereas the hash table literature typically uses n to be the number of keys/values, the balls-to-bins literature typically uses n to be the number of slots (or bins). In this section, we follow the balls-to-bins convention, and in the rest of the chapter we follow the hash-table convention.

(and there are up to $n-1$ balls present prior to any given insertion).

We then also prove a matching lower bound in Subsection 12.3.2: any balls-to-slots scheme that supports $\varepsilon \le 1/\log^{(O(1))}(n)$ with expected average probe complexity $O(\log^{(k)} n)$ must incur average switching cost $\Omega(k)$. Note that, whereas our upper bound supports $\varepsilon = 1/n$ (i.e., the slots are completely full), our lower bound allows for ε to be as large as $1/\log^{(O(1))} n$, without changing the answer for what the optimal tradeoff curve between probe complexity and switching cost is.

12.3.1 A Balls-to-Slots Scheme with Small Average Probe Complexity

In this section, we fix $\varepsilon = 1/n$, and we construct a balls-to-slots scheme that achieves switching cost $k+1$ (1 for inserting a ball, and k for moving around balls already in the system) while also achieving expected average probe complexity $\Theta(\log^{(k+1)} n)$. At the end of the section, we also show how to transform the bound on average probe complexity into a high-probability result.

Defining each ball's probe sequence. Define $s_0 = n$ and define $s_i = \Theta((\log^{(i)} n)^6)$ to be a power of two for each $i \in [1, k]$. We shall assume for simplicity that n is divisible by s_i for each $i > 0$, but the same arguments easily extend to arbitrary n.

We shall consider $k+1$ different ways of partitioning the n slots into bins: for $i \in [0, k]$, the ***depth-i partition*** breaks the slots into contiguous bins of size s_i. For each depth-i bin b, with $i > 0$, the ***parent bin*** b' of b is the depth-$(i-1)$ bin that contains b. (And b is a ***child*** of b'.) So the partitions are arranged in a tree, where the depth-i components are children of the depth-$(i-1)$ components, and where the branching factor of the tree decreases roughly exponentially between levels. We call this tree the ***k-kick tree***.

Before defining h, we define an auxiliary function g. Each ball x randomly selects a leaf of the tree (i.e, some depth-k bin b) and defines $g_i(x)$ to be the depth-i ancestor of b. In other words, each $g_i(x)$ is a uniformly random depth-i bin, and the sequence $g_0(x), g_1(x), \ldots, g_k(x)$ forms a root-to-leaf path through the kick tree.

The function $h_i(x)$ first cycles through the slots of $g_k(x)$, then the slots of $g_{k-1}(x)$, then the slots of $g_{k-2}(x)$, etc. Formally, this means that for each depth i and for each $j \in [s_i]$, $h_{(k+1-i)s_i+j}(x)$ is the j-th position in bin $g_i(x)$. Since the bin $g_0(x)$ contains *all* slots in $[n]$, the sequence $\{h_i(x)\}_{i \in [(k+1)n+1,(k+2)n]}$ hits every slot, so we do not need to define h_i for $i > (k+2)n$.

Whenever a ball x is inserted, it ends up at some depth i, and within that depth it is assigned to some position $j \in [s_i]$ of bin $g_i(x)$. The ball's probe complexity is then

$$O(1 + \log(k+1-i) + \log s_i).$$

Since $k+1-i = O(\log^* s_i)$, the probe complexity reduces to

$$O(\log s_i).$$

Throughout the rest of the section, we will think of each ball x's position as being determined by a pair (i, j), where i is a depth and j is a position in $g_i(x)$, rather than being determined directly by the probe sequence h. If a ball x is associated with depth i, we will treat it as having probe complexity $\Theta(s_i)$.

The structure of a ball insertion. Call a depth-i bin ***saturated*** if the bin contains no free slots and if all of the balls in the bin are associated with depths i or greater. Note that, when we are performing an insertion, $g_0(x)$ cannot be saturated, but $g_i(x)$ for $i > 0$ may be.

Whenever a ball x is inserted, we select a depth i such that none of the bins $g_0(x), g_1(x), \ldots, g_i(x)$ are saturated. (We will describe the process for selecting i later.) We assign x to bin $g_i(x)$ with depth i. If there is a free slot in $g_i(x)$, then we use it; otherwise, since $g_i(x)$ is not saturated, the bin must contain a ball x' associated with some depth $i' < i$. We assign x to the slot that x' is in, and we reassign x' to a new slot as follows. If there is a free slot in $g_{i'}(x')$, then we use it; otherwise, since $g_{i'}(x') = g_{i'}(x)$ is not saturated, the bin must contain a ball x'' associated with some depth $i'' < i'$. We assign x' to the slot that x'' is in, and we reassign x'' to a new slot, etc., where x'' may displace some ball x''' at a depth $i''' < i''$, and so on. In effect, we treat the depths as priorities, so that whenever a ball y is moved, it is permitted to displace any other ball y' that is of a lower priority.

Each insertion has switching cost at most $k + 1$, since it places the ball that is being inserted and then rearranges at most one ball in each depth $\{0, 1, \ldots, k-1\}$. Moreover, whenever a ball is moved, the depth that it is in stays the same, and thus the $O(\log s_i)$-bound on the probe complexity for that ball also stays the same. In order to achieve $O(\log^{(k+1)} n)$ average probe complexity (in expectation), it therefore suffices to ensure that, whenever a ball is inserted, the expected probe complexity for the new ball is $O(\log s_k) = O(\log^{(k+1)} n)$.

Choosing which depth to use. The final piece of the algorithm that we must specify is how to choose the depth i that a given ball insertion will use.

The most natural approach is to be greedy: select the largest i such that none of bins $g_0(x), g_1(x), \ldots, g_i(x)$ are saturated. This optimizes the probe complexity of the current insertion but comes with a downside. We are not doing anything to control which bins are saturated, so even though we are selecting i greedily, we cannot argue that i will actually be large for any given insertion (for example, what if $g_1(x)$ is saturated?).

Our solution is to take an *almost* greedy approach. Each ball x is assigned an independent hash $s(x) \in [0, k]$ satisfying

$$\Pr[s(x) < i] = 1/(\log^{(i)} n)^2$$

for each $i \in [1, k]$. The hash $s(x)$ dictates the *maximum possible depth* that ball x is permitted to be in. Each insertion x uses depth $\min(j, s(x))$, where j is the largest value such that none of the bins $g_0(x), g_1(x), \ldots, g_j(x)$ are saturated.

Analyzing a given insertion. We now analyze the expected probe complexity of a given ball.

Lemma 108. *Consider the insertion of some ball x into a k-kick tree with n slots. The expected probe complexity of x is $O(\log^{(k+1)} n)$.*

Proof. Let j be the largest value such that none of the bins $g_0(x), g_1(x), \ldots, g_j(x)$ are saturated. Then the probe complexity of x, after being inserted, is

$$O(\log \max(s_{s(x)}, s_j)) = O(\log s_{s(x)}) + O(\log s_j).$$

We can bound the expected value of the first quantity by

$$\begin{aligned}
\mathbb{E}[\log s_{s(x)}] &= \log s_k + \sum_{i \in [0,k)} \Pr[s(x) = i] \cdot \log s_i \\
&\le O(\log^{(k+1)} n) + \sum_{i \in [0,k)} \Pr[s(x) < i+1] \cdot \log s_i \\
&= O(\log^{(k+1)} n) + \sum_{i \in [0,k)} \frac{1}{(\log^{(i+1)} n)^2} \cdot \log s_i \\
&= O(\log^{(k+1)} n) + \sum_{i \in [0,k)} \frac{1}{(\log^{(i+1)} n)^2} \cdot \log(\log^{(i)} n)^6 \\
&= O\left(\log^{(k+1)} n + \sum_{i \in [0,k)} \frac{1}{(\log^{(i+1)} n)^2} \cdot \log^{(i+1)} n\right) \\
&= O\left(\log^{(k+1)} n + \sum_{i \in [0,k)} \frac{1}{\log^{(i+1)} n}\right) \\
&= O(\log^{(k+1)} n).
\end{aligned}$$

We can bound the expected value of the second quantity by

$$\mathbb{E}[\log s_j] \le \log s_k + \sum_{i \in [0,k)} \Pr[g_{i+1}(x) \text{ saturated}] \cdot \log s_i. \tag{12.2}$$

In order for $g_{i+1}(x)$ to be saturated (prior to x's insertion), there must be s_{i+1} balls

y present that satisfy $g_{i+1}(y) = g_{i+1}(x)$ and $s(y) \geq i+1$. For a given $y \neq x$,

$$\begin{aligned}
&\Pr[g_{i+1}(y) = g_{i+1}(x) \text{ and } s(y) \geq i+1] \\
&= \Pr[g_{i+1}(y) = g_{i+1}(x)] \cdot \Pr[s(y) \geq i+1] \\
&= \frac{1}{n/s_{i+1}} \cdot (1 - \Pr[s(y) < i+1]) \\
&= \frac{1}{n/s_{i+1}} \cdot \left(1 - 1/(\log^{(i+1)} n)^2\right).
\end{aligned}$$

The number Y of such y therefore satisfies

$$\begin{aligned}
\mathbb{E}[Y] &\leq n \cdot \frac{1}{n/s_{i+1}} \cdot \left(1 - 1/(\log^{(i+1)} n)^2\right) \\
&= s_{i+1} \cdot \left(1 - 1/(\log^{(i+1)} n)^2\right) \\
&= (\log^{(i+1)} n)^6 - (\log^{(i+1)} n)^4.
\end{aligned}$$

Since Y is a sum of independent indicator random variables, we can apply a Chernoff bound to deduce that

$$\begin{aligned}
\Pr[Y \geq (\log^{(i+1)} n)^6] &\leq e^{-\Omega(\log^{(i+1)} n)^2} \\
&\leq O\left(\frac{1}{(\log^{(i)} n)^2}\right).
\end{aligned}$$

This is an upper bound on the probability that $g_{(i+1)}(x)$ is saturated. Thus, by (12.2),

$$\begin{aligned}
\mathbb{E}[\log s_j] &\leq \log s_k + \sum_{i \in [0,k)} \frac{1}{(\log^{(i)} n)^2} \cdot \log s_i \\
&= O\left(\log^{(k+1)} n + \sum_{i \in [0,k)} \frac{1}{(\log^{(i)} n)^2} \cdot \log^{(i+1)} n\right) \\
&= O(\log^{(k+1)} n).
\end{aligned}$$

This completes the proof of the lemma. ∎

It's worth taking a moment to understand the bottlenecks in the Lemma 108. For convenience, let us focus on the setting where we are aiming for average probe complexity $O(1)$, so $k = \Theta(\log^* n)$; and further assume that, if a ball is placed in a depth-d bin, then the ball has probe complexity $\Theta(\log s_d)$. Consider some bin b with depth $i > 0$, meaning that the bin has size s_i. If we want to ensure that the probability of b being saturated is $o(1)$, then we must ensure that the expected number of elements in b is $s_i - \omega(\sqrt{s_i})$ (because the standard deviation of the number of elements in the bin is $\Theta(\sqrt{s_i})$). This means that the hash function $s(x)$ must satisfy

$$\Pr[s(x) < i] = \omega(1/\sqrt{s_i}).$$

However, whenever $s(x) < i$, the probe complexity of x is forced to be at least $\log s_{i-1}$. Thus the expected probe complexity of each ball x must be at least

$$\omega\left(\frac{\log s_{i-1}}{\sqrt{s_i}}\right).$$

Since we want average probe complexity $O(1)$, it follows that $\frac{\log s_{i-1}}{\sqrt{s_i}} \le 1$, or equivalently,

$$s_{i-1} \le 2^{\sqrt{s_i}}.$$

This is the inequality that fundamentally limits the rate at which the s_i's can shrink and that forces us to have $\Omega(\log^* n)$ depths in order to achieve average probe complexity $O(1)$. In fact, we'll see in Section 12.3.2 that this relationship between probe complexity and switching cost is fundamental—no balls-to-slots scheme can do better than the k-kick tree does.

An immediate consequence of Lemma 108 is:

Theorem 109. *For any $k \in [\log^* n - 1]$, the k-kick tree is a balls-to-slots scheme with $\varepsilon = 1/n$ that achieves worst-case switching cost $k+1$ and expected average probe complexity $O(\log^{(k+1)} n)$.*

We conclude the section by transforming our bound on expected average probe complexity into a high-probability bound.

Theorem 110. *For any $k \in [\log^* n - 1]$, there is a balls-to-slots scheme with $\varepsilon = 1/n$ that achieves worst-case switching cost $k+1$ and average probe complexity $O(\log^{(k+1)} n)$, with probability $1 - 1/2^{n^{\Omega(1)}}$ at any given moment.*

Proof. Let $\rho : U \to [\log n]$ be a fully independent and uniformly random hash function. Whenever a ball x is inserted, if $\rho(x) = 1$, then place x into a special slot.[7] With probability $1 - 1/2^{n/\log n}$, the number of balls in special slots is $O(n/\log n)$ at any given moment, meaning that they contribute $O(1)$ to the average probe complexity.

We hash the remaining balls x (i.e., balls x satisfying $\rho(x) > 1$) randomly to subarrays of size $\sqrt{n}$. The expected number of balls in a given sub-array is at most $\sqrt{n}(1 - 1/\log n)$, so with probability $1 - 1/2^{n^{\Omega(1)}}$ each of the subarrays receives at most $\sqrt{n} - 1$ balls at any given moment. In the rare case that a sub-array overflows, we can simply put the ball x being inserted into a special slot.

Finally, we implement each subarray using a k-kick tree. By Theorem 109, each insertion incurs switching cost $k+1$ and each subarray independently incurs expected

[7]Alternatively we could place x into whatever slot s is free and then move x to a different free slot whenever that slot s needs to be used by a different ball. This would increase the switching cost by at most 1 per operation and avoid the use of special slots.

average probe complexity $O(\log^{(k+1)} n)$.

If we define $X_1, X_2, \ldots, X_{\sqrt{n}}$ to be the average probe complexities of the subarrays, then $\{X_i\}$ are independent random variables satisfying $X_i \le O(\log n)$ and $\mathbb{E}[X_i] = O(\log^{(k+1)} n)$. By a Chernoff bound, we have with probability $1 - 1/2^{n^{\Omega(1)}}$ that the average probe complexity across the entire balls-to-slots scheme is $O(\log^{(k+1)} n)$, as desired. ▮

12.3.2 A Lower Bound on Switching Cost vs. Probe Complexity

In this section we will construct a sequence of insertions/deletions such that, in order for an online balls-to-slots scheme to achieve small average probe complexity, they must incur a large average switching cost. Since we are constructing a lower bound, we shall refer to the balls-to-slots scheme that we are analyzing as our ***adversary***.

Let U be the universe of balls, let h be the function mapping each ball x to a probe sequence $\{h_i(x)\}$, and let n be the number of slots. For $i \in \mathbb{N}, j \in [n]$, define

$$q(h, i, j) = n \Pr_{x \in U}[h_k(x) = j \text{ for some } k \le i].$$

Intuitively, if there are n random balls present, then $q(h, i, j)$ represents the expected number of balls that are capable of residing in slot j with probe complexity at most $1 + \log i$.

If the $h_i(x)$'s are selected uniformly and independently in $[n]$, then we will have $q(h, i, j) = \Theta(i)$ for each $i \in [n]$. Call h ***nearly uniform*** if $q(h, i, j) < \operatorname{poly}(i)$ for all i, j. As a minor technical convention, we will also allow for $h_i(x)$ to be null, in which case it does not contribute to any $q(h, i, j)$ (and $h_i(x)$ cannot be used by any ball assignment).

We shall begin by proving a lower bound that holds assuming a nearly uniform h. We shall also initially assume that $\varepsilon = 1/n$ and that the average probe complexity being achieved by the balls-to-slots scheme is $O(1)$. We will then remove these assumptions at the end of the section.

Theorem 111. *Suppose the universe U has sufficiently large polynomial size. Consider any balls-to-slots scheme that uses nearly-uniform probe sequences, that achieves expected average probe complexity $O(1)$ (across all balls in the system at any given moment), and that supports $\varepsilon = 1/n$. The expected amortized switching cost per insertion/deletion must be $\Omega(\log^* n)$.*

Throughout the rest of the section, we shall consider an input sequence that begins right after $n - 1$ random balls have just been inserted, and then proceeds to perform $M = \operatorname{poly}(n)$ insertions and the same number of deletions. The insertions and deletions alternate; each insertion inserts a random ball (which with probability $1 - 1/\operatorname{poly}(n)$ has never been inserted in the past); and each deletion deletes a random ball out of those present.

Define $L = \lceil(\log^* n)/2\rceil$. Define $\text{tow}(0) = 1$ and $\text{tow}(i) = 2^{\text{tow}(i-1)}$ for all integer $i > 0$. Say that a ball x is in ***level 0*** if it has been assigned to a slot $h_i(x)$ for some $i \leq \text{tow}(L)$, and say that a ball x is in ***level*** $j \in \{1, 2, \ldots, L\}$ if it has been assigned to a slot $h_i(x)$ for some $i \in (\text{tow}(L + j - 1), \text{tow}(L + j)]$. If a ball is in a special slot, or if it has been assigned to a slot $h_i(x)$ with $i \geq n$, then the ball is said to be in level L.

Say that a move by the adversary has ***impact*** r if it decreases the level of some ball by r. Positive impact means that the ball's level decreased, and negative impact means that the ball's level increased.

Lemma 112. *For $i \in [M]$, define α_i to be the sum of the impacts of the moves that the adversary performs during the i-th insertion, and define β_i to be the sum of the impacts of the moves that the adversary performs during the i-th deletion. Finally, define $\psi = \sum_{i \in [M]}(\alpha_i + \beta_i)$ to be the total impact by the adversary across all insertions/deletions. Then*

$$\mathbb{E}[\psi] = \Theta(ML).$$

Proof. Recall that M is the number of insertions (resp. deletions) performed, and that L is the number of levels. Define a dynamically-changing quantity J to be the sum of the levels of the balls in the system at any given moment.

Each insertion places a ball into a special slot, thereby increasing J by L. On the other hand, we claim that each deletion decreases J by $O(1)$ in expectation. To see this, observe that the deletion decreases J by s where s is the level of the element being deleted. Since the adversary guarantees an expected average probe complexity of $O(1)$, we have that $\mathbb{E}[s] = O(1)$, which means that J decreases by $O(1)$ in expectation.

By the definitions of α_i and β_i, we have that during the i-th insertion (resp. i-th deletion), the adversary's moves decrease J by α_i (resp. β_i). Across all operations, the total effect of the adversary's moves on J is to decrease it by ψ. If J_0 is the value of J prior to the first of the $2M$ operations and J_* is the value of J after the final operation, then

$$\mathbb{E}[J_*] = \mathbb{E}[J_0] + LM - \Theta(M) - \mathbb{E}[\psi] = \mathbb{E}[J_0] + \Theta(LM) - \mathbb{E}[\psi],$$

where the LM term accounts for insertions, the M term accounts for deletions, and the ψ term accounts for moves by the adversary. On the other hand, J_0 and J_* are both deterministically in the range $[0, O(n \log^* n)]$, so we must have

$$\Theta(LM) - \mathbb{E}[\psi] \leq O(n \log^* n).$$

Since M is a large polynomial, it follows that $\mathbb{E}[\psi] = \Theta(LM)$, as desired. ∎

The main technical ingredient to complete the proof of Theorem 111 will be to construct a potential function ϕ with the following properties:

- **Property 1:** Each insertion/deletion increases ϕ by at most $O(1)$ in expectation.
- **Property 2:** If a move by the adversary has impact $r \in \mathbb{Z}$, it decreases ϕ by $r \pm O(1)$.
- **Property 3:** ϕ always satisfies $0 \le \phi \le nL$.

Before we construct ϕ, let us assume the existence of such a ϕ and use it to complete the proof. At any given moment, define ψ to be the sum of the impacts of the moves that the adversary has made so far. We will examine how the quantity $\psi + \phi$ evolves over time.

By Property 3, the quantity $\psi + \phi$ is initially at most nL. By Property 1, each insertion/deletion increases $\psi + \phi$ by $0 + O(1) = O(1)$ in expectation. By Property 2, each move by the adversary increases $\psi + \phi$ by at most $r - (r - O(1)) = O(1)$ (deterministically). Thus, after M insertions/deletions have been performed, if k is the total number of moves that the adversary makes, then

$$\mathbb{E}[\psi + \phi] \le nL + O(M) + O(\mathbb{E}[k]) = O(M) + O(\mathbb{E}[k]).$$

On the other hand, by Property 3, $\mathbb{E}[\psi] \le \mathbb{E}[\psi + \phi]$, so

$$\mathbb{E}[\psi] \le O(M) + O(\mathbb{E}[k]).$$

Lemma 112 tells us that $\mathbb{E}[\psi] = \Theta(ML)$. Thus

$$ML \le O(M) + O(\mathbb{E}[k]),$$

which means that $\mathbb{E}[k] = \Omega(ML) = \Omega(M \log^* n)$, hence Theorem 111. The main challenge is therefore to construct a potential function ϕ with the three desired properties.

Constructing the potential function $\boldsymbol{\phi}$. The basic idea behind ϕ is that it should approximate the amount of impact that the adversary could hope to achieve with a small number of moves. One way to do this would be as follows. We could define $\mathcal{S}$ to be the set of all possible move sequences that the adversary could make; for each $S \in \mathcal{S}$, we could define $I(S)$ to be the total impact of S and $|S|$ to be the number of moves in S; and we could define

$$\phi = \max_{S \in \mathcal{S}} \left(I(S) - c|S|\right)$$

for some large positive constant c. This potential function would exactly capture the adversary's ability to achieve large impact with a small number of moves, but it comes with the drawback that it can behave somewhat erratically with respect to insertions, deletions, and adversary-moves.

A key idea in this section is to construct ϕ in a more intricate way, still upper-bounding the amount of impact that the adversary can achieve cheaply, but while also intentionally designing ϕ to behave nicely. In order to give the technical definition of ϕ, we must first define the notion of an ***i*-stanza**, which intuitively corresponds to a sequence of moves in which the adversary is able to reduce the level of some ball b from $\geq i$ to $\leq i-3$ while preserving for every other ball b' how the level ℓ' of b' compares to the quantities $i-2, i-1, i$.

Define the ***level of a slot*** s to be the level of the ball in the slot, if there is such a ball, and to be L otherwise. For $i \in [L]$, define an ***i*-stanza** to be a sequence of slots $s_1, \ldots, s_j$ such that slots s_1 and s_j have levels at least i; such that slots $s_2, \ldots, s_{j-1}$ have levels at most $i-3$; such that each slot s_k, $k \in [j-1]$, contains a ball x that can be placed into s_{k+1} with a new level of at most $i-3$; and such that $s_2, \ldots, s_{j-1}$ are distinct. Note that, by design, $s_2, \ldots, s_j$ cannot be special slots (since they must be capable of containing a ball with level $\leq i-3$), but s_1 can be.

Importantly, the final slot s_j in a stanza does *not* have to be an empty slot in order for the stanza to be valid. With that said, if the final slot s_j were empty, then the stanza would have a very intuitive interpretation: one could think of the stanza is representing a possible chain of ball moves, where the first ball move (from slot s_1 to slot s_2) decreases the level of some ball from $\geq i$ to $\leq i-3$, where each subsequent ball move (from slot s_k to slot s_{k+1} for some $k > 0$) maintains the level of some ball to be at most $i-3$, and where the final move places a ball into an empty slot.

We say that an i-stanza $s_1, \ldots, s_j$ has ***size*** j and has ***potential*** $1-(j-1)/L$. We refer to s_1 as the ***starting slot*** of the stanza, to $s_2, \ldots, s_{j-1}$ as the ***internal slots*** of the stanza, and to s_j as the ***final slot*** of the stanza. We say that a collection of i-stanzas are ***disjoint*** if each slot with level $\geq i$ is used at most once as a starting slot and at most once as a final slot, and if each slot with level $\leq i-3$ is used at most once as an internal slot. (The only overlap allowed is that the starting slot of one stanza may be the ending slot of another.) The ***potential*** of a disjoint collection of i-stanzas is the sum of the potentials of the individual stanzas.

For $i \in [L]$, define ϕ_i to be the maximum potential of any disjoint collection of i-stanzas. Finally, define the potential function ϕ by

$$\phi = \sum_{i=3}^{L} \phi_i.$$

The intuition behind $\boldsymbol{\phi}$. Before analyzing ϕ, let us give a bit more intuition for why ϕ acts as a natural upper-bound for how much impact the adversary can achieve cheaply (i.e., with only a small number of moves relative to the impact being achieved).

Consider any sequence of moves that the adversary could perform, and define a ***realized stanza*** to be a sequence of slots $s_1, \ldots, s_j$ such that s_j is an empty slot and, for each $k \in [j-1]$, the ball from slot s_k gets moved to the next slot s_{k+1} in the sequence. One can think of a realized stanza as a sequence of moves, where balls $x_1, \ldots, x_{j-1}$ are in slots $s_1, \ldots, s_{j-1}$ and are being moved to positions $s_2, \ldots, s_j$, respectively. Each ball x_k is moved from some initial level b_k to some potentially different level e_k. (As an edge case,since there is no ball initially in s_j, define $b_j = L$, and leave e_j undefined.)

For a given move, from some level b_k to some level e_k, there are three cases for the adversary. If $e_k \in \{b_k - 2, b_k - 1\}$, then we think of the move as having been neither good nor bad for the adversary—the move created $\Theta(1)$ impact, but at the cost of 1 move. If $e_k \leq b_k - 3$, then we think of the move as being good for the adversary, and we say that the adversary has ***stolen*** $b_k - e_k - 2$ levels $b_k, b_k - 1, \ldots, e_k + 3$. Finally, if $e_k \geq b_k$, then we think of the move as being bad for the adversary, and we say that the adversary has ***paid*** for $e_k - b_k + 1$ levels $b_k + 1, \ldots, e_k + 2$. Whenever the adversary pays for a level i, that cancels out the previous time that the adversary stole that level i. In order for the adversary to steal a level i without subsequently paying for it, there must be a sequence $(b_k, e_k), \ldots, (b_{k'}, e_{k'})$ such that $b_k \geq i$, such that $e_k, b_{k+1}, e_{k+1}, b_{k+2}, \ldots, e_{k'-1} \leq i - 3$, and such that $b_{k'} > i$. This sequence of ball moves corresponds exactly to an i-stanza. In other words, each i-stanza represents a possible opportunity for the adversary to steal level i without subsequently paying for it.

In order for a realized stanza to be worthwhile to the adversary, however, the adversary must perform an average of $\omega(1)$ steals per move. This means that, on average, each level of the L levels $i > 0$ must be stolen $\omega(1)$ times for every L moves that are performed. In other words, whenever the adversary steals level i, but then fails to steal level i again for L moves, then that first steal wasn't actually worthwhile. The value of a given steal can be modeled as $1 - q/L$, where q is the number of subsequent moves until the next steal of the same level. This is why we define the potential of an i-stanza in the way that we do: the longer that a i-stanza is, the less worthwhile of an opportunity that it represents for the adversary.

In summary, each i-stanza represents an opportunity for the adversary to steal level i without subsequently paying for it; and the i-stanza's potential upper-bounds how valuable that steal would be to the adversary. An important aspect of how we define ϕ is that we analyze each of the levels i separately, so that the i-stanzas do not have to care about ball moves (s_k, e_k) satisfying $s_k, e_k \geq i + 1$ or satisfying $s_k, e_k \leq i - 3$. As we shall see, this decouples the analyses of the levels from one another in several critical ways.

Analyzing the properties of ϕ. At any given moment, let A_1 denote the set of balls that are present, and, for the sake of analysis, let A_2 denote a set of n random balls that are not present, one of which is the ball that will next be inserted. Define $A = A_1 \cup A_2$. Define $B = [n]$ to be the set of all non-special slots.

Define a bipartite graph $G_i = (A, B)$, where for each $a \in A$ and $b \in B$ we draw an edge (a, b) if ball a is capable of residing in slot b with level at most i. That is, there is an edge from a to b if $b \in \{h_1(a), \ldots, h_{\text{tow}(L+i)}(a)\}$. Note that balls $a \in A$ all deterministically have degrees at most $\text{tow}(L + i)$. For each slot b, let $d_i(b)$ denote the degree of b in G_i, and call b ***high-degree in*** G_i if $d_i(b) \geq (\text{tow}(L + i))^c$ for some sufficiently large constant c. We call all other nodes in G_i (including all $a \in A$) ***low-degree in*** G_i.

We now argue that most nodes in G_i are far away from any high-degree nodes.

Lemma 113. *Let a_1 be a random ball in A_1 and let a_2 be a random ball in A_2. With probability $1 - 1/\operatorname{poly}(L)$, neither a_1 nor a_2 is within distance $O(L)$ of any high-degree vertex in G_i.*

Proof. Since the balls $a \in A$ are independent and randomly selected, the degree $d_i(b)$ is a sum of independent indicator random variables. Moreover, by the near-uniformity of h, we know that each $b \in B$ satisfies

$$\begin{aligned}\mathbb{E}[d_i(b)] &= 2 \cdot n \Pr_{x \in U}[h_k(x) = b \text{ for some } k \leq \operatorname{tow}(L+i)] \\ &= 2 \cdot q(h, \operatorname{tow}(L+i), b) \\ &\leq 2 \cdot \operatorname{poly}(\operatorname{tow}(L+i)) \\ &\leq (\operatorname{tow}(L+i))^c/2.\end{aligned}$$

Applying a Chernoff bound, it follows that for all $D \geq (\operatorname{tow}(L+i))^c$, we have

$$\Pr[d_i(b) \geq D] \leq \frac{1}{2^{\Omega(D)}}.$$

Thus

$$\mathbb{E}[d_i(b) \cdot \mathbb{I}_{d_i(b) \geq (\operatorname{tow}(L+i))^c}] \leq \frac{1}{2^{\Omega((\operatorname{tow}(L+i))^c)}} = \frac{1}{2^{\operatorname{poly}(\operatorname{tow}(L+i))}}.$$

This means that the expected sum S of the degrees of the high-degree slots in G_i satisfies

$$\mathbb{E}[S] \leq n/2^{\operatorname{poly}(\operatorname{tow}(L+i))}.$$

One can also think of S as an upper bound on the number of low-degree nodes in G_i that are adjacent to high-degree nodes in G_i. Every low-degree node in G_i has degree at most $\operatorname{poly}(\operatorname{tow}(L+i))$. It follows that the number λ of nodes in G_i that are within distance $O(L)$ of a high-degree node satisfies

$$\mathbb{E}[\lambda] \leq S \cdot \operatorname{poly}(\operatorname{tow}(L+i))^{O(L)},$$

where the first factor S counts the number of nodes s in G_i that are within distance 1 of a high-degree node, and the second factor counts the number of $O(L)$-long paths starting at a such a node s and then using only low-degree nodes. Using our bound on S, we get that

$$\mathbb{E}[\lambda] \leq \frac{n}{2^{\operatorname{poly}(\operatorname{tow}(L+i))}} \cdot \operatorname{poly}(\operatorname{tow}(L+i))^{O(L)}.$$

The above quantity is dominated by its first factor, so

$$\mathbb{E}[\lambda] \leq \frac{n}{2^{\operatorname{poly}(\operatorname{tow}(L+i))}}.$$

Applying Markov's inequality, we have that with probability $1 - 1/2^{\text{poly}(\text{tow}(L+i))} \geq 1 - 1/\text{poly}(L)$,

$$\lambda \leq \frac{n}{2^{\text{poly}(\text{tow}(L+i))}}.$$

Let a_1 be a random ball in A_1 and a_2 be a random ball in A_2. The probability that either a_1 or a_2 is within distance $O(L)$ of a high-degree vertex in G_i is at most

$$\Pr\left[\lambda > \frac{n}{2^{\text{poly}(\text{tow}(L+i))}}\right] + \frac{\frac{n}{2^{\text{poly}(\text{tow}(L+i))}}}{\Theta(n)} = \frac{1}{\text{poly}(L)} + \frac{1}{2^{\text{poly}(\text{tow}(L+i))}} \leq \frac{1}{\text{poly}(L)}.$$

This completes the proof of the lemma. ∎

The next lemma argues that, if we remove the high-degree nodes from G_i, then most of the remaining nodes are far away from any nodes with levels $\geq i + 3$.

Lemma 114. *Define G_i' to be the graph G_i, but with all high-degree nodes removed. Let X be the set of balls and non-special empty slots that are currently at a level at least $i + 3$. For random balls a_1, a_2 in A_1, A_2, respectively, the probability of either a_1 or a_2 being within distance $O(L)$ of X in G_i' is at most $1/\text{poly}(L)$.*

Proof. Note that X is determined by the balls-to-slots scheme, so we will think of X as being selected by an adversary who has full knowledge of A_1 and A_2 but who has no control over the contents of A_1 and A_2. Since $\varepsilon = 1/n$, the number of slots in X is at most 1 greater than the number of balls in X, so to bound $|X|$, we can focus on the number of balls with levels $\geq i + 3$.

Each ball $x \in X$ has a level of $i + 3$ or greater, so it has probe complexity at least $\log \text{tow}(L + i + 2) = \text{tow}(L + i + 1)$. This means that, at any given moment,

$$\mathbb{E}[|X|] \leq \frac{O(n)}{\text{tow}(L + i + 1)}, \tag{12.3}$$

where the randomness here comes from the fact that the balls-to-slots scheme guarantees an *expected* average probe complexity of $O(1)$ at any given moment. By Markov's inequality,

$$|X| \leq \frac{n\,\text{poly}(L)}{\text{tow}(L + i + 1)}$$

with probability $1 - 1/\text{poly}(L)$. The number of nodes y that are within distance $O(L)$ of X in G_i' is therefore at most

$$\frac{n\,\text{poly}(L)}{\text{tow}(L + i + 1)}\,\text{tow}(L + i)^{O(L)} \ll \frac{n}{\text{poly}(L)}. \tag{12.4}$$

It follows that, for random balls a_1, a_2 in A_1, A_2, respectively, the probability of either a_1 or a_2 being within distance $O(L)$ of X in G_i' is at most $1/\text{poly}(L)$. ∎

We can now argue that each insertion/deletion increases ϕ by $O(1)$ (actually $o(1)$) in expectation. Intuitively, this means that insertions/deletions do not, on average, introduce opportunities for the adversary to cheaply achieve a large amount of impact.

Lemma 115 (Establishing Property 1 for ϕ)**.** *Each insertion/deletion increases ϕ by at most $1/\operatorname{poly}(L)$ in expectation.*

Proof. Consider an insertion of a random ball $a \in A_2$. Let us consider the effect of the insertion on ϕ_{i+3} for some i. Notice that, when a is inserted (i.e., placed into a special slot), ϕ_{i+3} either stays the same or increases by ≤ 1, where the increase comes from the fact that a may be part of some $(i+3)$-stanza that has positive potential and did not exist before. On the other hand, the only way that a can be part of an $(i+3)$-stanza that has positive potential is if, in G_i, a is within distance L of some slot whose level is $\geq i+3$—the probability of this occurring is therefore an upperbound on the expected increase to ϕ_{i+3} due to the insertion.

By Lemma 113, we have with probability $1 - 1/\operatorname{poly}(L)$ that the set of nodes $y \in G_i$ that are within distance $O(L)$ of a is the same as the set of nodes $y \in G'_i$ that are within distance $O(L)$ of a. By Lemma 114, we have that with probability $1 - 1/\operatorname{poly}(L)$, that within the graph G'_i, a is not within distance $O(L)$ of any node with level $\geq i+3$ (besides a itself). Thus, with probability $1 - 1/\operatorname{poly}(L)$, we have that in G_i, a is not within distance $O(L)$ of any node with level $\geq i+3$ (besides a itself). This establishes that, with probability $1 - 1/\operatorname{poly}(L)$, a is not the starting node for any $(i+3)$-stanza that has positive potential; and thus the expected increase to ϕ_i due to the insertion is $O(1/\operatorname{poly}(L))$.

Now consider the deletion of a random ball $a \in A_1$. By the same reasoning as in the preceding paragraph, with probability $1 - 1/\operatorname{poly}(L)$, we have that in the graph G_i, a is not within distance $O(L)$ of any node with level $\geq i+3$ (besides possibly a itself). Thus, once a is deleted, we have with probability $1 - 1/\operatorname{poly}(L)$ that the slot which contained a is not the final slot for any $(i+3)$-stanza that has positive potential. Hence the expected increase to ϕ_i due to the deletion is $1/\operatorname{poly}(L)$.

In either case, the expected increase to

$$\phi = \sum_{i=3}^{L} \phi_i$$

is at most $\sum_{i=3}^{L} 1/\operatorname{poly}(L) = 1/\operatorname{poly}(L)$. Thus the lemma is proven. ∎

Next we analyze the effect that a given move by the adversary has on ϕ.

Lemma 116 (Establishing Property 2 for ϕ)**.** *If a move by the adversary has impact r, it decreases ϕ by $r \pm O(1)$.*

Proof. We can assume without loss of generality that the only moves that the adversary ever makes are either (a) to move a ball from a non-special slot into a special slot or (b) to move a ball from a special slot to a non-special slot. Indeed, any move that takes a ball from a non-special slot to another non-special slot can be replaced by a move of type (a) followed by a move of type (b). Also recall that, when a ball is inserted, it initially resides in a special slot, and the adversary can then move it to a non-special slot if desired.

Since moves of type (a) are the reverse of moves of type (b), it suffices to analyze only moves of type (b), and to show that ϕ decreases by $r \pm O(1)$.

Suppose the adversary moves ball x from a special slot s_1 to a non-special slot s_2, where it has level j. Let Σ denote the state of the system before the move, and Σ' denote the state of the system after the move. Let ϕ be the potential of Σ and ϕ' be the potential of Σ'.

To complete the proof, we will argue that for each $i \in [L]$:

- If $i \leq j$, then $\phi'_i = \phi_i$.
- If $i \in \{j+1, j+2\}$, then $\phi_i - 2 \leq \phi'_i \leq \phi_i$.
- If $i \geq j+3$, then $\phi_i - 1 \leq \phi'_i \leq \phi_i - 1 + 1/L$.

Case 1: The first case is immediate, since changes to the positions/levels of balls with levels $\geq i$ do not affect which sequences of ball moves correspond to valid i-stanzas.

Case 2: Any valid i-stanza in Σ' is also a valid stanza in Σ (hence $\phi'_i \leq \phi_i$), but there may be some i-stanzas in Σ that are not valid in Σ' (specifically, any i-stanza in Σ that makes use of either ball x to start a stanza, or slot s_2 to finish a stanza). For any set of disjoint i-stanzas in Σ, up to two of those i-stanzas might be invalid in Σ' (but no more than two!). Thus $\phi'_i \geq \phi_i - 2$.

Case 3: In the rest of the proof, we focus on the third case, where $i \geq j+3$. Let C be a set of disjoint i-stanzas in Σ that maximizes the sum of the potentials of the i-stanzas. Let $s_1 \circ c_1$ (where s_1 is the slot defined earlier in the proof and c_1 is a sequence of slots) be the i-stanza in C that uses x as its first ball (if such a stanza exists), and let $c_2 \circ s_2$ (where c_2 is a sequence of slots and s_2 is the slot defined earlier in the proof) be the i-stanza in C that uses slot s_2 as its final slot (if such a stanza exists).

We begin by claiming that c_1 and c_2 exist without loss of generality. If c_1 does not exist, then we can modify C by removing any stanza that uses slot s_2, and inserting the stanza $\langle s_1, s_2 \rangle$ instead (this replacement either keeps the total potential of C the same or increases it). So c_1 exists without loss of generality. If c_2 does not exist, then we can modify C by removing any stanza that uses s_1, and inserting the stanza $\langle s_1, s_2 \rangle$ instead (again, this cannot decrease the total potential of C). Thus c_2 also exists without loss of generality. We can further observe that, if $s_1 \circ c_1$ and $c_2 \circ s_2$ happen to be the *same* stanzas as one another, then that stanza is simply $\langle s_1, s_2 \rangle$ (indeed, if that stanza were not $\langle s_1, s_2 \rangle$, then we could replace it with $\langle s_1, s_2 \rangle$ in order to increase the potential of C, which would be a contradiction).

We will now argue that $\phi' \geq \phi - 1$. If C contains the stanza $\langle s_1, s_2 \rangle$, then $C \setminus \{\langle s_1, s_2 \rangle\}$ is a set of disjoint i-stanzas in Σ' with potential exactly $1 - 1/L$ smaller than that of C; thus $\phi' \geq \phi - (1 - 1/L) \geq \phi - 1$. On the other hand, if C does not contain the stanza $\langle s_1, s_2 \rangle$, then the stanzas $s_1 \circ c_1$ and $c_2 \circ s_2$ must be distinct. In this case, we claim that $c_3 = c_2 \circ s_2 \circ c_1$ is a valid i-stanza in Σ'. Indeed, slot s_2 in Σ' contains ball x at level $j \leq i - 3$, so slot s_2 is allowed to be an internal slot in an i-stanza; and since, in Σ, the i-stanzas $s_1 \cdot c_1$ (which begins with the slot containing ball x) and $c_2 \circ s_2$ (which ends in slot s_2) are valid, it follows that, in Σ', the i-stanza $c_3 = c_2 \circ s_2 \circ c_1$ is valid. Since c_3 is a valid i-stanza in Σ', we have that $C' = C \setminus \{s_1 \circ c_1, c_2 \circ s_2\} \cup \{c_3\}$ is a set of disjoint i-stanzas in Σ'. The potential of C' is exactly 1 smaller than that of C. So $\phi' \geq \phi - 1$.

To complete the proof, we must also establish that $\phi \geq \phi' + 1 - 1/L$. Let $\overline{C}'$ be a set of disjoint i-stanzas in Σ' that maximizes the sum of the potentials of the i-stanzas. If there is no stanza in $\overline{C}'$ that makes use of slot s_2, then $\overline{C}' \cup \{\langle s_1, s_2 \rangle\}$ is a valid set of disjoint i-stanzas in Σ, which would mean that $\phi \geq \phi' + 1 - 1/L$. Suppose, on the other hand that there is some i-stanza of the form $c_1 \circ s_2 \circ c_2$ in $\overline{C}'$. Then the stanzas $c_1 \circ s_2$ and $s_1 \circ c_2$ are valid in Σ, and thus $\overline{C}' \setminus \{c_1 \circ s_2 \circ c_2\} \cup \{c_1 \circ s_2, s_1 \circ c_2\}$ is a valid set of disjoint i-stanzas in Σ. This means that $\phi \geq \phi' + 1 \geq \phi' + 1 - 1/L$, completing the proof. ∎

The previous two lemmas establish Properties 1 and 2 for ϕ. Finally, the third property, which states that $0 \leq \phi \leq Ln$ is trivially true, since $\phi_i \in [0, n]$ for all $i \in [L]$. Thus Theorem 111 is proven.

Generalizing to other values of load factor and of probe complexity. So far we have assumed for simplicity that $\varepsilon = 1/n$ and that the balls-to-slots scheme being analyzed achieves expected average probe complexity $O(1)$. We now generalize our lower bound to consider $\varepsilon \geq 1/n$ and probe complexity $\omega(1)$.

Theorem 117. *Consider a universe U of sufficiently large polynomial size. Consider any balls-to-slots scheme that uses nearly uniform probe sequences, that achieves expected average probe complexity $O(\text{tow}(a))$ (across all balls in the system at any given moment), and that supports some $\varepsilon = 1/\log^{(b)} n$ where $b \leq (\log^* n)/4$.*[8] *The expected amortized switching cost per insertion/deletion must be at least*

$$\Omega(\log^* n - a - b).$$

Note that, when $a = O(1)$ and $b = 0$, Theorem 117 becomes Theorem 111, which we have already proven. And as we shall now see, the proof of Theorem 117 requires only a slight modification to the proof of Theorem 111.

Proof. Let us begin by considering $a > 0$ and $b = 0$, so average probe complexity may be $\omega(1)$ but $\varepsilon = 1/n$.

The only substantive modification to the proof is that, if $a - L \geq 0$, then we redefine any balls in levels less than $a - L$ to now be in level $a - L$ (so we eliminate levels $0, 1, \ldots, a - L - 1$). Intuitively, this is because, since the balls-to-slots scheme is

[8]In this notation, if $b = 0$, then $\varepsilon = 1/n$.

allowed to have average probe complexity $O(\text{tow}(a))$, it is without loss of generality the case that every ball is in level $a - L$ or above.

Formally, the reason that we need to restrict to levels $i \geq a - L$ is to preserve Lemma 114. The bound (12.3) on the expected number of balls with probe complexity at least $\text{tow}(L + i + 1)$ now becomes

$$O\left(\frac{n \,\text{tow}(a)}{\text{tow}(L + i + 1)}\right), \tag{12.5}$$

instead of $O(n/\text{tow}(L+i+1))$. In order for the $\text{tow}(a)$ term not to become significant in the proof of Lemma 114, we need $\text{tow}(a) \leq \text{tow}(L + i)$ (that way, in (12.4), the newly introduced $\text{tow}(a)$ term can be absorbed into the $\text{tow}(L + i)^{O(L)}$ term). Since we restrict ourselves to levels i satisfying $i \geq a - L$, Lemma 114 continues to be correct for every valid level i.

We must also modify Lemma 112 to accommodate the fact that each deletion now removes a ball with expected level $\max(0, a - L) + o(1)$ (rather than expected level $O(1)$). This changes our final lower bound on expected average switching cost to $\Omega(L - (a - L)) = \Omega(\log^* n - a)$.

Now suppose we also allow $b > 0$. To handle this, we again modify how we define the levels: we define $\ell_* = L - b - 1$, and we declare any ball or slot (including special slots) that was previously in some level $\ell' > \ell_*$ to now be in level ℓ_*. The intuition for why we do this is that, once we get to level ℓ_*, many of the slots that are in that level or above are actually empty slots, so it makes sense to treat that as the top level.

Formally, the reason that we need to restrict to levels $i \leq \ell_*$ is to again preserve (12.3) in Lemma 114. In particular, (12.3) must count not just the balls that have probe complexity $\text{tow}(L + i + 1)$ but also any (non-special) *empty slots* (since such slots represent maximum-level nodes in G'_i and therefore contribute to $|X|$). There may be up to $O(n/\log^{(b)} n)$ such slots (in expectation), each of which is in the top level; to preserve (12.3), we therefore need that

$$O(n/\log^{(b)} n) \leq \frac{O(n \,\text{tow}(a))}{\text{tow}(L + i + 1)}. \tag{12.6}$$

Recall, however, that we have limited ourselves to levels i satisfying $i \leq l_*$, which implies $i \leq (\log^* n)/2 - b - 1$, and thus that $L + i + 1 \leq \log^* n - b$, and therefore that

$$\log^{(b)} n = \text{tow}(\log^* n - b) \geq \text{tow}(L + i + 1).$$

Hence, as long as i is a valid level, then (12.5) still holds, which preserves the correctness of Lemma 114.

Since we restrict ourselves to $L - b - 1$ levels, we must also modify Lemma 112 to accommodate the fact that each insertion now increases the sum of the levels of the balls J by only $L - b - 1$ (instead of by L). In the case where $\max(0, a - L) \leq L$, this reduces the final lower-bound that we achieve on expected average switching cost to $\Omega(L - b - 1) = \Omega(L)$ (here we are using that $b \leq (\log^* n)/4$), and in the case where

$\max(0, a-L) > L$, this reduces the final lower bound to $\Omega(\log^* n - a - b)$. Both lower bounds are equivalent to $\Omega(\log^* n - a - b)$. ∎

We remark that the restriction $b \leq (\log^* n)/4$ can easily be reduced by defining L to be much smaller than $(\log^* n)/2$. Such values of b are not relevant to hash-table design, however, since any augmented open-addressing hash table with load factor of at least, say, $1 - 1/O(\log \log n)$ must use a balls-to-slots scheme that supports $b \leq 2$.

Non-nearly-uniform probe sequences. Finally, we extend our lower bound to non-nearly-uniform probe sequences. To do this, we formally reduce the non-nearly-uniform case to the nearly-uniform case.

For any assignment A mapping some set of up to n balls to slots, and for any function h determining the probe sequences $h_1(x), h_2(x), \ldots$ for each ball, define $c(A, h)$ to be the total probe complexity needed to implement assignment A using h.

Lemma 118. *Consider any universe U and consider any function h assigning a probe sequence to each ball $x \in U$. Then there exists a nearly uniform h' that has the following guarantee. For any assignment A of $\Theta(n)$ balls to slots, $c(A, h') \leq O(c(A, h) + n)$.*

Proof. For each ball $x \in U$ and each $j \in [n]$, let $s(x, j) = \operatorname{argmin}_k \{h_k(x) = j\}$ and let $s'(x, j) = \operatorname{argmin}_k \{h'_k(x) = j\}$ (we can assume without loss of generality that these quantities exist).

We now describe how to construct h'. Rather than specifying $h'_i(x)$ for all i, x, it suffices to specify $s'(x, j)$ for all x, j. Note that, in order for $s'(x, j)$ to be well defined, the only restriction is that the quantities $s(x, 1), s(x, 2), \ldots, s(x, n)$ must be *distinct* natural numbers.

Define $t : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ to be an injective function satisfying $\log t(a, b) \leq O(1 + \log a + \log b)$ for all $(a, b) \in \mathbb{N} \times \mathbb{N}$ and satisfying $t(a, b) \geq \max(a, b)$ for all $(a, b) \in \mathbb{N}$. Recall that, for a probe sequence function h', if we have n random balls, then $q(h', i, j)$ is the expected number of balls x that are capable of residing in position j using one of the first i probe values $h'_1(x), \ldots, h'_i(x)$. We set

$$s'(x, j) = t(s(x, j), \lceil q(h, s(x, j), j) \rceil). \tag{12.7}$$

We claim that $s'(x, j)$ is well defined. Indeed, if $s'(x, j_1) = s'(x, j_2)$ for some $j_1 \neq j_2$, then we must also have that $s(x, j_1) = s(x, j_2)$, which would be a contradiction. We also observe that h' (constructed using s') has strictly larger probe complexities than does h—if a ball x is in a position j, then its probe complexity using h would be $\Theta(1 + \log s(x, j))$ but its probe complexity using h' would be $\Theta(1 + \log s(x, j) + \log\lceil q(h, s(x, j), j) \rceil)$. It may seem strange that we are defining h' to be worse than h, but as we shall now prove, this allows for us to guarantee that h' is nearly uniform. Once we establish this, then our only remaining task will be to bound how much worse h' is than h, in the worst case.

We now argue that h' is nearly uniform, meaning that $q(h', i, j) \leq \operatorname{poly}(i)$ for all

i. Observe that

$$q(h', i, j) = n \cdot \Pr_{x \in U}[h'_k(x) = j \text{ for some } k \le i]$$
$$\le \frac{n \cdot |\{x \in U \mid s(x,j) \le i \text{ and } q(h, s(x,j), j) \le i\}|}{|U|},$$

since in order to have $h'_k(x) = j$ for some $k \le i$, we must have that $t(s(x,j), \lceil q(h, s(x,j), j) \rceil)) \le i$ and thus that $s(x,j) \le i$ and $q(h, s(x,j), j) \le i$. By expanding out the definition of $q(h, s(x,j), j)$, we get that $q(h', i, j)$ is at most

$$\begin{aligned}
&\frac{n \cdot |\{x \in U \mid s(x,j) \le i \text{ and } \Pr_{y \in U}[h_r(y) = j \text{ for some } r \le s(x,j)] \le i/n\}|}{|U|} \\
&\le \frac{n \cdot |\{x \in U \mid s(x,j) \le i \text{ and } \Pr_{y \in U}[h_{s(x,j)}(y) = j] \le i/n\}|}{|U|} \\
&\le \frac{n \cdot |\{x \in U \mid \exists\, s \in [i] \text{ s.t. } h_s(x) = j \text{ and } \Pr_{y \in U}[h_s(y) = j] \le i/n\}|}{|U|} \\
&\le \frac{n}{|U|} \sum_{s \in [i] \text{ such that } \Pr_{y \in U}[h_s(y)=j] \le i/n} |\{x \in U \mid h_s(x) = j\}| \\
&= \frac{n}{|U|} \sum_{s \in [i] \text{ such that } \Pr_{y \in U}[h_s(y)=j] \le i/n} |U| \Pr_{y \in U}[h_s(y) = j] \\
&\le \frac{n}{|U|} \sum_{s \in [i] \text{ such that } \Pr_{y \in U}[h_s(y)=j] \le i/n} \frac{i|U|}{n} \\
&\le \frac{n}{|U|} \cdot \sum_{s \in [i]} \frac{i|U|}{n} \\
&= i^2.
\end{aligned}$$

This establishes the near-uniformity of h'.

To complete the proof, we must argue that $c(A, h') \le O(c(A, h) + n)$. Consider a ball x that A assigns to some position j. The probe complexity of x using h is $1 + \log s(x,j)$, and the probe complexity of x using h' is $O(1 + \log s(x,j) + \log q(h, s(x,j), j))$. Thus, if A assigns addresses $j_1, j_2, \ldots, j_m$ to balls $x_1, x_2, \ldots, x_m$, respectively, for some $m = \Theta(n)$, then our goal is to show that

$$\sum_{i=1}^{m} \log q(h, s(x_i, j_i), j_i) \le O\left(n + \sum_{i=1}^{m} \log s(x_i, j_i)\right).$$

The cases where $q(h, s(x_i, j_i), j_i) \le \text{poly}(s(x_i, j_i))$ trivially satisfy $\log q(h, s(x_i, j_i), j_i) \le O(\log s(x_i, j_i))$, so it suffices to show

$$\sum_{i=1}^{m} \mathbb{I}_{q(h, s(x_i, j_i), j_i) > \text{poly}(s(x_i, j_i))} \log q(h, s(x_i, j_i), j_i) \le O(n).$$

Each $j \in [n]$ appears as a j_i at most once in the above sum. Thus

$$\sum_{i=1}^{m} \mathbb{I}_{q(h,s(x_i,j_i),j_i)>\operatorname{poly}(s(x_i,j_i))} \log q(h, s(x_i, j_i), j_i) \leq \sum_{s=1}^{\infty} \sum_{j=1}^{n} \mathbb{I}_{q(h,s,j)>\operatorname{poly}(s)} \log q(h, s, j).$$

We can therefore complete the proof by showing that

$$\sum_{s=1}^{\infty} \sum_{j=1}^{n} \mathbb{I}_{q(h,s,j)>\operatorname{poly}(s)} \log q(h, s, j) \leq O(n).$$

Let Q_s be the set of j for which $q(h, s, j) > \operatorname{poly}(s)$. Then,

$$\sum_{s=1}^{\infty} \sum_{j=1}^{n} \mathbb{I}_{q(h,s,j)>\operatorname{poly}(s)} \log q(h, s, j) = \sum_{s=1}^{\infty} \sum_{j \in Q_s} \log q(h, s, j).$$

For any fixed s, we have that

$$\begin{aligned}
\sum_{j=1}^{n} q(h, s, j) &= \sum_{i=1}^{s} \sum_{j=1}^{n} n \Pr_{x \in U}[h_i(x) = j] \\
&= n \sum_{i=1}^{s} \sum_{j=1}^{n} \Pr_{x \in U}[h_i(x) = j] \\
&= n \sum_{i=1}^{s} \Pr_{x \in U}[h_i(x) = j \text{ for some } j \in [n]] \\
&\leq ns.
\end{aligned}$$

Thus $\sum_{j \in Q_s} q(h, s, j)$ is also at most sn and has at most $sn/\operatorname{poly}(s) = n/\operatorname{poly}(s)$ terms. By Jensen's inequality, this implies that

$$\sum_{j \in Q_s} \log q(h, s, j) \leq \frac{n}{\operatorname{poly}(s)} \log \frac{sn}{n/\operatorname{poly}(s)} = \frac{n}{\operatorname{poly}(s)}.$$

Summing over all s,

$$\sum_{s=1}^{\infty} \sum_{j \in Q_s} \log q(h, s, j) \leq \sum_{s=1}^{\infty} \frac{n}{\operatorname{poly}(s)} = O(n).$$

This completes the proof. ∎

By the preceding lemma, the assumption in Theorem 117 that h is nearly uniform is true without loss of generality, since we can substitute any non-nearly-uniform h with a nearly-uniform h' while having an asymptotically negligible effect on the probe complexity of any balls-to-slots assignment. Thus we arrive at the main theorem of the section:

Theorem 119. *Suppose the universe U has sufficiently large polynomial size. Consider any balls-to-slots scheme that achieves expected average probe complexity $O(\text{tow}(a))$ (across all balls in the system at any given moment) and supports some $\varepsilon = 1/\log^{(b)} n$ where $b \leq (\log^* n)/4$. The expected amortized switching cost per insertion/deletion must be at least*

$$\Omega(\log^* n - a - b).$$

Corollary 120. *Suppose the universe U has sufficiently large polynomial size. Consider any balls-to-slots scheme that achieves expected average probe complexity $O(1)$ (across all balls in the system at any given moment) and supports $\varepsilon = 1/n$. The expected amortized switching cost per insertion/deletion must be $\Omega(\log^* n)$.*

To conclude the section, we reinterpret our result as a lower bound on augmented open-addressing.

Corollary 121. *Any augmented open-addressed hash table that stores quotiented $(1 + \Theta(1)) \log n$-bit elements in an array and incurs $O(\log^{(k)} n)$ expected wasted bits per key must have average insertion/deletion time $\Omega(k)$.*

Proof. We can assume without loss of generality that $k \geq 2$. In order for the wasted bits per key to have expected value $O(\log^{(k)} n) \leq O(\log \log n)$, the load factor $1 - \varepsilon$ of the array must satisfy $1 - \varepsilon \geq 1 - \frac{\log \log n}{\log n}$. That is, the balls-to-slots scheme used by the hash table must support $\varepsilon \leq \frac{\log \log n}{\log n}$. In the language of Theorem 119, this means that $b < 2$.

The bound of $O(\log^{(k)} n)$ wasted bits per key also implies that the (expected) average probe complexity of the balls-to-slots scheme is $O(\log^{(k)} n)$. In the language of Theorem 119, this means that $a \geq (\log^* n) - k$.

Applying Theorem 119, we get that the average switching cost of the balls-to-slots scheme is at least $\Omega(\log^* n - a - b) = \Omega(k)$. Thus the average insertion/deletion time of the hash table is $\Omega(k)$. ∎

12.4 Encoding Metadata in an Augmented Open-Addressed Hash Table

So far, we have computed tight bounds on the probe complexity of any balls-to-slots scheme. If the balls-to-slots scheme used by an augmented open-addressing hash table has total probe complexity ℓ, then the hash table must store at least $\Omega(\ell)$ bits of metadata. In this section, we present general machinery for how to implement the metadata of the hash table to use exactly $O(\ell)$ bits, while also allowing for constant-time modifications to the metadata. The key difficulties here are that ℓ may differ for between elements (i.e., it is nonuniform) and that ℓ may be, on average, very small, meaning that we cannot afford a high space overhead per element.

To address these issues, we develop two fundamental building blocks: the first is a data structure that we call the ***mini-array***, which compactly stores a polylog n-element dynamic array of items that are between 1 and $O(\log n)$ bits each so that array entries can be queried and modified in constant time; the second is a data structure that we call the ***local query router***, which compactly stores routing information (i.e., information about where some element x can be found in the hash table) for up to $O((\log n)/\log\log n)$ elements at a time, while supporting queries/updates to the routing information in constant time.

As foreshadowing, and to give some additional context, let us comment on how these building blocks will be used later. Ultimately, our approach to storing metadata in a hash table will be the following: we will hash keys to buckets of some expected size $K = \text{polylog}\, n$; then, within each bucket, we will hash keys to K smaller buckets of expected size $O(1)$; for each of these smaller buckets, we will use a local query router to store the metadata for the elements in that bucket; and for each of the larger buckets, we will use a mini-array to store the K local query routers for its K smaller buckets. In this section, however, our goal is simply to construct mini-arrays and local query routers.

12.4.1 Preliminaries: The Lookup-Table Technique

Several of the data structures in this section will make use of the ***lookup-table technique*** (sometimes also called the Method of Four Russians). This allows for us to implement potentially complicated operations on $(\log n)/2$-bit inputs in constant time.

More formally, call such a function $f(x_1, \ldots, x_j)$ ***lookup-table-compatible*** if: the input tuple $(x_1, \ldots, x_j)$ takes less than $(\log n)/2$ bits; the output takes $O(\log n)$ bits; and f can be evaluated in time $O(n^{1/4})$.

If f is lookup-table-compatible, then, when we initialize a hash table of size n, we can pre-construct a lookup table L of size $\sqrt{n}$ such that $L[x_1, \ldots, x_j] = f(x_1, \ldots, x_j)$ for each of the up to $\sqrt{n}$ input tuples $(x_1, \ldots, x_j)$. The lookup table allows for us to evaluate f in constant time during hash-table operations. The lookup table L consumes at most $\tilde{O}(\sqrt{n})$ bits of space and can be constructed in time at most $O(n^{3/4})$.

We can also rebuild the lookup table (in a deamortized fashion) whenever the parameter n changes by more than a constant factor, so the restriction that each input tuple $(x_1, \ldots, x_j)$ takes less than $(\log n)/2$ bits is always a function of the current n.

Finally, suppose that we have a function f for which the input tuple $(x_1, \ldots, x_j)$ takes $\Theta(\log n)$ bits, rather than $(\log n)/2$ bits. We say that f is ***lookup-table-friendly*** if for some positive constant c, there exist lookup-table-compatible functions $f_1, \ldots, f_c$ such that: the input tuple $(x_1, \ldots, x_j)$ can be decomposed into $(\log n)/2$-bit (or smaller) pieces $P_1, \ldots, P_c$, and $f(x_1, \ldots, x_j)$ can be computed in constant time given $f_1(P_1), \ldots, f_c(P_c)$. To implement f in constant time, we can implement each f_i

using the lookup-table technique. So lookup-table-friendly functions can be evaluated in constant time without loss of generality.

12.4.2 Storing a Mini-Array of Variable-Size Values

Consider the following basic data-structural problem, which we call the ***mini-array problem***. Let c be a sufficiently large positive constant, and let $K = \log^c n$. We wish to store a K-element ***mini-array*** $A[1], \ldots, A[K]$, where each element $A[i]$ has some size $s_i \in [0, O(\log n)]$ bits. We wish to support queries (i.e., tell me $A[i]$) and updates (i.e., set $A[i]$ to a new value) in constant time, and we wish to use space $O(K + \sum_i s_i)$ bits. In our setting, we will have a large collection of mini-arrays, each a part of a larger data structure whose total size is $\Omega(n)$. So we will allow for our solution to use lookup tables that are shared among all of the mini-arrays.

How should we implement a mini-array? The problem is that the sizes s_i of the elements in the array are non-uniform and change over time. So we cannot implement A as a standard array. Instead, we take inspiration from the external-memory model [349], and we implement A as a B-tree [61] T. The basic idea is that, since we can implement (most) operations on $\Theta(\log n)$-bit machine words in constant time using the lookup-table approach, we can think of machine words as representing data blocks in the external-memory model.

The tree T consists of polylog n nodes, each of which is $\Theta(\log n)$ bits (the only exception is the root node, which may contain fewer bits). Because the tree consists of only polylog n nodes, pointers within the tree need only be $\Theta(\log \log n)$ bits each.[9]

Each internal node of T stores $\Theta(\log n / \log \log n)$ pointers to children (although, again, the root may contain fewer), and for each child the node stores two pivots $p_1, p_2 \in [K]$ indicating the range of indices that the child covers. Each leaf of T stores $\Theta(\log n)$ bits of array entries (i.e., $A[i], \ldots, A[j]$ for some i, j such that $\sum_{\ell=i}^{j}(s_\ell + 1) = \Theta(\log n)$). We call these the ***array bits***. Each leaf also stores a $\Theta(\log n)$-bit bitmap indicating where each $A[\ell]$ begins within the array bits.

Nodes are merged and split as in a standard B-tree: there is some positive constant d such that, whenever a node exceeds $d \log n$ bits, the node is split into two nodes, and whenever a node's size falls below $d(\log n)/2$ bits, the node is merged with one of its neighbors (and then the new merged node may also need to be split). The only way that the height of the tree can increase is if the root splits into two nodes a and b (in which case a new root is created with a and b as children), and the only way that

[9]For the applications in this chapter, the amount of memory needed to implement T will always be known (up to constant factors) up front, so we can preallocate the memory in a single contiguous array. Even if the size of T is not known up front, however, it is still possible to implement pointers within the tree using $\Theta(\log \log n)$ bits per pointer. Indeed, we can assign the nodes distinct $\Theta(\log \log n)$-bit identifiers, and then we can maintain a dynamic fusion tree [306] mapping identifiers to true $\Theta(\log n)$-bit pointers—the fusion tree allows us to perform address translation in order to go from an identifier to the corresponding actual node. Note that the fusion tree introduces only a constant-factor space overhead overall, and introduces only on additive constant time overhead for each operation; so we can feel free to ignore the fusion tree, and treat pointers as each using $\Theta(\log \log n)$ bits.

the height of the tree can decrease is if the root has only a single child, in which case the root is eliminated. Every node except the root has the property that it always uses $\Theta(\log n)$ bits, but the root may be smaller (since it has no neighbors that it can merge with). Since the tree has fanout $\Theta((\log n)/\log\log n)$ (for all internal nodes except for the root), and since the tree consists of $O(K) \le \operatorname{polylog} n$ nodes, the depth is $O(1)$.

Using the lookup-table approach, we can implement both queries and updates in constant time. In particular, the tasks of navigating down the tree, finding where a given $A[i]$ resides in some leaf, modifying some A_i in some leaf, and modifying internal nodes are all directly implementable using lookup-table-friendly functions.

This concludes the description of how to implement a mini-array. Each operation is deterministically constant time. And, up to constant factors, the space-usage of the tree is dominated by the leaves, which in aggregate use $O(K + \sum_i s_i)$ space, as desired. The lookup tables used to implement the mini-array take a total of $\tilde{O}(\sqrt{n})$ space, but since these lookup tables can be shared across all instances of mini-arrays, that space is negligible.

12.4.3 Query Routers

We now describe a second data-structural problem, which we call the ***query-router problem***. To understand the query-router problem, it is helpful to understand how we will use local query routers in our hash tables. We will hash $\Theta(n)$ keys to $\Theta(n)$ different local query routers, and each local query router will be responsible for storing the probe-indices corresponding to those keys—that is, if a local query router stores a key x that resides in slots $h_i(x)$ of the hash table, a query searching for key x must be able to recover the value i from the local query router. The way in which the local query router is used results in several interesting properties that we will make exploit in its construction: with high probability in n, each local query router will be storing information for at most $O(\log n/\log\log n)$ keys; additionally, if a local query router wishes to access one of the keys x for which it is storing information, it can do so in constant time (without having to actually store x).

With these properties in mind, we now formally define the query-routing problem: Consider a set S of distinct keys, and a function $f : S \to \mathbb{N}$ that maps keys to distinct values. We wish to support modifications to S and f (i.e., delete s from S, or insert s into S with $f(s) := u$) and f-evaluation queries (i.e., what is $f(s)$ for some specific $s \in S$?) in constant time (with high probability in n). We are guaranteed that $|S|$ never exceeds $O((\log n)/\log\log n)$ and that $f(s)$ always takes $O(\log\log n)$ bits. Setting $r = |S| + \sum_{s\in S} \log f(s)$ to be the sum of the sizes of the $f(s)$'s, we wish to have a data structure of *expected size* $O(r)$ bits, at any given moment, and of *worst-case size* $O(\log n)$ bits, at any given moment with high probability in n. Our data structure also has access to a constant-time oracle for the function $g = f^{-1}$. That is, if $f(s) = u$ for some s, then the oracle function satisfies $g(u) = s$. (If $f(s) \ne u$ for all $s \in S$, then $g(u)$ is not defined, and could return an arbitrary value.) The oracle makes it so that our data structure does not have to store keys—it can recover each key based on the corresponding f-value.

We now describe a data structure, which we call a ***local query router***, that solves the above problem. Although the precise specifications are slightly different, the design for the local query router is very similar to the querying mechanism used in past work on adaptive filters [82] (as well as by other subsequent work on succinct filters [245]).

Before we continue, let us make some simplifications to the requirements of a local query router, and argue that these simplifications are without loss of generality. First, we may assume that the local query router has a lifespan of only $O(\log n / \log \log n)$ operations, since we can rebuild the local query router from scratch once every $O(\log n / \log \log n)$ operations (and in a deamortized fashion). Second, it suffices to construct a local query router with failure probability $1 - 1/n^{\varepsilon}$ on any given insertion/deletion, since we can amplify this failure probability to $1/\operatorname{poly}(n)$ by storing $O(1)$ independent local query routers, and keeping track of which one(s) haven't yet failed—in any sequence of $O(\log n / \log \log n)$ operations, the probability of all $O(1)$ local query routers failing is $1 - 1/\operatorname{poly}(n)$. We call a local query router that makes the above simplifications a ***simplified local query router***.

To construct a simplified local query router, we will need the following basic lemma about binary tries.

Lemma 122. *Let $k = O(\log n / \log \log n)$ and let $r_1, \ldots, r_k$ be random binary strings. Let T be the binary trie storing the smallest unique prefix of each r_i (i.e., if the smallest unique prefix of r_1 is* 01101*, then there is a path corresponding to* 01101 *in the trie). Then T has expected size $O(k)$, and for any constant $c > 1$ there exists a constant $\varepsilon > 0$ such that with probability $1 - 1/n^{\varepsilon}$, T has size that most $\log n / c$ bits.*

Proof. Imagine constructing T by inserting each of $r_1, \ldots, r_k$ into the trie one after another. Inserting a new element into T corresponds to performing a random walk down the tree T to some leaf ℓ, and then appending a path of some length X below that leaf, and then adding two new leaves at the end of that path. Note that the random variable X is independent between insertions and satisfies

$$\Pr[X \geq i] = 1/2^i.$$

Thus, once all of the k insertions are performed, the size of T is simply a sum of independent geometric random variables. By a Chernoff bound for sums of independent geometric random variables, the lemma follows. ∎

We can now construct a simplified local query router. We hash of the elements of S to random binary strings, and we place those binary strings in a trie T. For each leaf of T corresponding to some $s \in S$, we also store the value $f(s)$ at that leaf.

In more detail, we can encode the tree, along with the f-value for each of the leaves as follows. Perform a depth-first traversal through the tree, and write down the sequence of moves that the traversal performs (i.e., moves of the form "go to left child", "go to right child", "go up"); call this portion of the encoding E_1, and observe that $|E_1|$ is $\Theta(|T|)$ bits. Then write down the f-values for the leaves in the same order that they appear in the depth-first traversal of the tree (it is straightforward to encode the value in such a way that it can easily be determined where one value begins and another finishes); call this portion of the encoding E_2, and observe that $|E_2|$ is $\Theta(|S| + \sum_{s \in S} \log f(s))$ bits.

The total number of bits in the encoding is $|E_1|+|E_2| = O(|T|+|S| \sum_{s\in S} \log f(s))$ which, by Lemma 122, has expected value $O(|S|+\sum_{s\in S} \log f(s))$. Lemma 122 further tells us that, for any positive constant c, there exists a positive constant ε such that $|E_1| \le (\log n)/c$ with probability $1 - 1/n^{\varepsilon}$. Since, by assumption, we have that $|E_2| = O(\log n)$, it follows that the total encoding takes $O(\log n)$ bits. Finally, since $|E_1| \le (\log n)/c$, and since E_2 can be broken into $O(1)$ lists of f-values that are $(\log n)/c$ bits each, we can implement insertions/deletions/queries on the encoding in constant time using lookup-table-friendly functions. (Note that, when we insert a new element into the trie, we need to know the random string corresponding to the leaf that the insertion ends up splitting—this is where our data structure makes use of the oracle g, which allows for us to recover the key corresponding to any leaf of the trie in constant time.)

In the preceding paragraphs, we have constructed a constant-time simplified local query router, and since the reduction from a full local query router to a simplified local query router is without loss of generality, we have also completed the construction and analysis for the full local query router.

12.5 An Optimal Augmented Open-Addressed Hash Table

Using the techniques developed in the previous sections, we can now construct a dynamically-resized augmented open-addressed hash table that stores $\Theta(\log n)$-bit key-value pairs, that supports insertions/deletions in time $O(k)$, that supports queries in time $O(1)$, and that achieves $O(\log^{(k)} n)$ wasted bits per key. (The running-time and space guarantees are with high probability in n). By Corollary 121, our data structure achieves the best possible tradeoff curve between time and space that any augmented open-addressed hash table can achieve.

We break the section into three parts:

- Subsection 12.5.1 constructs a fixed-capacity hash table that uses $nw + O(n \log^{(k)} n)$ bits of space to store n w-bit keys-value pairs.

- Subsection 12.5.2 shows how to make the hash table dynamically-resizable.

- And Subsection 12.5.3 reduces the space consumption to be within $O(n \log^{(k)} n)$ bits of the information-theoretic optimum.

12.5.1 Turning the k-Kick Tree into a Hash Table

In this section, we construct a fixed-capacity hash table that uses $nw + O(n \log^{(k)} n)$ bits of space to store n w-bit keys-value pairs.

The layout. Let $K = \operatorname{polylog} n$ be a parameter. We hash keys to $(1+1/K^{1/3})(n/K)$ bins, each of which we refer to as a ***cubby***. With high probability in n, each cubby receives at most K keys at any particular time.

Each cubby maintains a ***storage array*** capable of storing up to K keys/values. Keys are assigned a position in the storage array using the k-kick tree from Theorem 109 for some parameter k. (We will discuss how to do this time-efficiently later.) The parameter k will determine the tradeoff between time and space efficiency in our data structure.

Recall that the k-kick tree associates each key x with a random sequence of hash functions $g_0(x), \ldots, g_k(x)$, where each $g_{i+1}(x)$ is a child bin of $g_i(x)$. Of course, $g_k(x)$ determines all of $g_1, \ldots, g_{k-1}(x)$, and one way to pick $g_k(x)$ is to select a random $g(x) \in [K]$, and set $g_k(x)$ to be the depth-k bin that contains position $g(x)$. We will refer to $g(x)$ as x's ***preferred slot*** (within the cubby).

For each cubby, and for each $i \in [K]$, we maintain a local query router that stores metadata for the keys who have preferred slot $g(x) = i$. For each such key x, the local query router stores the index j such that x is in position $h_j(x)$ of the cubby—if x is stored at depth i by the k-kick tree, then we can store j using $O(\log^{(i+1)} K) = O(\log^{(i+2)} n)$ bits. As a slight abuse of notation, to simplify discussion throughout the rest of the chapter, we shall redefine the probe complexity of x to be exactly $\Theta(\log^{(i+1)} K)$, even though technically the true probe complexity may be smaller.

We store the K local query routers in a mini-array A. The result is that any key x in the data structure can be recovered by (a) hashing to the appropriate cubby; (b) finding the $g(x)$-th local query router in the mini-array; and (c) using that local query router to determine which slot of the storage array the key resides in. Note that the array A is local to each individual cubby.

Implementing insertions/deletions/queries in constant time. We have already seen how to implement queries in constant time using the mini-array A of local query routers. Deletions can also be implemented in constant time by simply removing the key/value pair.

Insertions are slightly more tricky, however. Recall that the balls-to-slots scheme has $k+1$ classes of bin sizes, where the sizes are denoted $s_0, \ldots, s_k$. Note that, in this setting, $s_0 = K = \operatorname{polylog} n$ and $s_i = \operatorname{poly}(\log^{(i+1)} n)$ for each $i \in [k]$.

Let us start by ignoring depth 0 and discuss how to implement depths $1, \ldots, k$. We maintain a second mini-array M storing metadata for each of the K/s_1 depth-1 bins. For each such bin, the metadata that we store is the information of which slots are free in that bin, and for each slot that is not free in that bin, what the depth is for the element in that slot. In aggregate, this information comprises $\operatorname{poly}(\log\log n)$ bits.

Using this metadata, along with the mini-array A, we can use lookup-table-friendly functions to implement the portions of an insertion that occur in depths $1, \ldots, k$ in time $O(k)$ (i.e., we can perform the entire insertion, except possibly the final step in which we must find a free slot to place some depth-0 element in).

The only task that remains is to locate a free slot in depth-0 (i.e., in the entire cubby). For this, we can simply maintain a $\log_{\log n} K = O(1)$-depth tree with uniform fanout $\log n$, in which each internal node stores a $\log n$-bit bitmap indicating which of its children contain at least one free slot, and each leaf stores a $\log n$-bit bitmap indicating which of the $\log n$ slots corresponding to that leaf are free. We refer to this as the ***free-slot tree***. The free-slot tree supports constant-time modifications and queries (where a query finds a free slot).

Proving correctness. We now establish the correctness of our data structure.

Lemma 123. *The above data structure correctly implements insertions/deletions/queries, ensures that insertions/deletions take time $O(k)$ with high probability in n, and ensures that queries take time $O(1)$ deterministically.*

Proof. By a Chernoff bound, each cubby has at most K keys at any specific time, so each insertion has a high probability of hashing to a cubby that has room for it. This means that the k-kick tree can operate correctly without overflowing.

We next verify that each of the local query routers operates correctly: each local query router requires that it store metadata for at most $O(\log / \log\log n)$ keys, and that each key has at most $O(\log\log n)$ bits of metadata. The first requirement follows by a Chernoff bound on the number of keys that hash to a given cubby and have a given value of $g(x)$. (The number of such keys has expected value 1, and is at most $O(\log n / \log\log n)$ with high probability in n.) The second requirement follows from the fact that each key has probe complexity $O(\log K) = O(\log\log n)$ bits in the balls-to-slots scheme.

Next we verify that each mini-array operates correctly: each mini-array requires that its entries are each $O(\log n)$ bits. This is immediate for M, and for A it follows from the fact that each local query router takes $O(\log n)$ bits (with high probability).

Since the requirements for correctness have been met for each mini-array and query-router, all of them will support constant-time operations with high probability in n. It follows that insertions and deletions are correct and take $O(k)$ time (with high probability)[10], and that queries are correct and take $O(1)$ time deterministically. ∎

Finally, we analyze the space consumed by our data structure. We shall assume that the number w of bits taken by each key/value pair satisfies $w = \Theta(\log n)$.

Lemma 124. *With high probability in n, the size of the data structure is $nw + O(n \log^{(k)} n)$ bits.*

[10]In the low-probability event that an insertion fails to be implementable, either because a cubby overflows, or because a query-router overflows, we simply rebuild the entire data structure from scratch.

Proof. We start by bounding the space consumed by storage arrays. There are $(1 + 1/K^{1/3})n/K$ cubbies each of which has a storage array of size Kw bits. This reduces to

$$(1 + 1/K^{1/3})nw = nw + o(n)$$

bits.

Next we bound the expected space consumed by the mini-arrays A and M in a cubby. If a cubby has $j \leq K$ keys and their probe complexities are $a_1, \ldots, a_j$, then the expected space consumed by A and M is $O(K + \sum_i a_i)$ bits (this is expected rather than worst-case because the local query routers may add more bits in the worst case). On the other hand, by Theorem 109,

$$\mathbb{E}\left[K + \sum_i a_i\right] = O(K \log^{(k+2)} n) = O(K \log^{(k)} n).$$

Thus the expected amount of space used by the mini-arrays in a given cubby is $O(K \log^{(k)} n)$. Summing over the cubbies, the total amount of space used by mini-arrays in the data structure is

$$O(n \log^{(k)} n)$$

bits in expectation.

We can turn this into a high-probability bound as follows. Define $r_1, \ldots, r_{(1+1/K^{1/3})n/K}$ so that r_i is the number of bits consumed by the mini-arrays in the i-th cubby. Note that each r_i is deterministically at most $O(K \log n)$, since the data structure is rebuilt whenever either (1) more than K elements simultaneously hash to some cubby, or (2) $\omega(\log n)$ bits are needed for some local query router in some cubby. Moreover, regardless of the outcomes of $\{r_j \mid j \neq i\}$, we have that $\mathbb{E}[r_i] = O(K \log^{(k)} n)$. Thus we can apply a Chernoff bound to deduce that $\sum_i r_i$ is tightly concentrated around its mean, so the total space used by mini-arrays is $O(n \log^{(k)} n)$ bits with high probability in n. ∎

Putting the pieces together, we have the following theorem:

Theorem 125. *Let $w = \Theta(\log n)$ and $k \in [\log^* n]$. One can construct a dictionary that stores up to n w-bit key/value pairs, while supporting insertions/deletions in time $O(k)$, supporting queries in time $O(1)$, and using total space $wn + O(n \log^{(k)} n)$ bits, with high probability in n.*

The preceding theorem has several limitations that we will remove in the coming sections. The first limitation is that our hash table does not yet support dynamic resizing (i.e., it has a fixed capacity). The second limitation is that our hash table stores each key/value pair in its entirety, even though information-theoretically, only $w - \log n + O(1)$ bits are needed per key/value pair. Each of the next two sections will remove one of these constraints.

We conclude the section by proving a simple technical lemma about cubbies that will be useful later. The lemma says that, even though modifying a cubby takes time $O(k)$, we can build a cubby from scratch in linear time $O(K)$.

Lemma 126. *Let S be a set of at most K key/value pairs. We can construct a cubby storing S in time $O(K)$ with high probability in n*

Proof. Recall that keys are stored in one of $k+1$ depths. Inserting a key into depth i takes up to $O(i)$ time, since we may have to relocate one key in each of depths $i-1, \ldots, 0$. To get around this issue, we build the cubby as follows: we first try to place each key into depth k, and if a key cannot be placed in depth k (either because there is no room, or because the key has hash $s(x) < k$, then we do not insert the key); we then try to place the remaining keys into depth $k-1$, and again if a key cannot be placed into depth $k-1$, then we do not insert the key; we continue like this for each of depths $k-2, k-3, \ldots, 0$ one after another.

For each key x, define j_x so that $k - j_x + 1$ is the depth at which x ends up being inserted. The total time to build the cubby is $O(\sum_x j_x)$. Define r_x to be the probe complexity of x. Then $j_x \le O(1) + r_x / \log^{(k)} n$. Thus the total time to build the cubby is

$$O(K) + O(\sum_x r_x / \log^{(k)} n).$$

We know from the analysis in Theorem 109 that $\mathbb{E}[\sum_x r_x] \le O(K \log^{(k)} n)$, so the expected time to build the cubby is $O(K)$.

To turn this into a high-probability bound, we must obtain a high-probability bound on $\sum_x j_x$. For this, we can perform a similar analysis as in Theorem 110. Break the cubby into $\sqrt{K}$ parts. Since there are at most $K = \operatorname{polylog} n$ elements total, the number of elements that hash to any given part is at most $\sqrt{K}(1 + 1/\operatorname{polylog} n)$ with high probability in n. If a part receives more than $\sqrt{K}$ keys, then call the remaining $K/\operatorname{polylog} n$ keys that it receives ***extra keys***. Modify the construction of the cubby so that we first find places for all of the non-extra keys, and then we insert the extra keys—since there are so few extra keys, they add a negligible total amount to the running time. Define $J_1, \ldots, J_{\sqrt{K}}$ so that J_i is the sum of the depths of the up to K non-extra keys that map to the i-th part. By the same analysis as above, we have that $\mathbb{E}[J_i] = O(\sqrt{K})$ for each i, regardless of the outcomes of the outcomes of $\{J_r \mid r \ne i\}$. We also have that $J_i \le O(k\sqrt{K})$ deterministically. Thus we can apply a Chernoff bound to $J = \sum_{i=1}^{\sqrt{K}} J_i$ to determine that $J = O(K)$ with high probability in n. This implies that the total construction time for the cubby is $O(K)$, as desired. ∎

12.5.2 Supporting Dynamic Resizing

In this section, we adapt the hash table from the previous section in order to support dynamic resizing: the amount of space that the hash table consumes will now be a function of the current number n of elements in the table, rather than some maximum capacity n.

To begin, we will focus on supporting n in a fixed range $[N, 2N]$, and we shall assume that the size of a key-value pair is $w = \Theta(\log N)$ bits. At the end of the section, we will generalize to allow for n to vary over a polynomial range (i.e., it is subject only to the constraint that $\log n = \Theta(w)$). (And, in fact, later in Section 12.6.1, we will show how fully generalize for arbitrary values of n.)

The basic layout. Let $K = \operatorname{polylog} N$ and let $k \in [\log^* K]$ be a parameter. Our hash table will consist of N/K ***facilities***, where each facility contains $\Theta(K)$ elements. When an element is inserted, it is hashed to a random facility.

Each facility is composed of many cubbies (implemented as in the previous section) of different sizes. More specifically, at any given moment, we will always maintain a ***distribution invariant***, which guarantees that for each facility there are:

- $\Theta((\log^{(k)} n)^2)$ cubbies of capacity $K/(\log^{(k)} n)^2$;
- and $\Theta\left(\frac{(\log^{(j)} n)^2}{(\log^{(j+1)} n)^2}\right)$ cubbies of size $K/(\log^{(j)} n)^2$, for each $j \in [k-1]$.

We say that an cubby is ***j-tiered*** if its size is $K/(\log^{(j)} n)^2$. The way to think about the distribution of cubby sizes is that, for each $j < k$, the *total size* of the j-tiered cubbies is asymptotically equal to the size of a *single* $j+1$-tiered cubby. That is, for $j < k$, there are at most $O(K/(\log^{(j+1)} n)^2)$ elements in j-tiered cubbies at a time.

At any given moment, one of the 1-tiered cubbies is designated as the ***tail***. The second invariant that we will maintain is that, at any given moment, *all* of the cubbies except for the tail are completely full. We call this the ***saturation invariant***.

We will describe how to efficiently maintain the distribution and saturation invariants shortly, but first we finish describing the layout of a facility. Each facility must always store the following: (a) pointers to all of the cubbies stored in the facility; and (b) metadata allowing for queries to determine which cubby the key they are looking for is in. Since each cubby has size $\operatorname{polylog} n$, the pointers to cubbies take negligible space. The metadata for queries can be stored as follows: we maintain a mini-array D with K entries; we hash each key x to one of the entries of the mini-array, and each entry stores a local query router that maps each key x to the appropriate cubby. Note that, if a key x is in a k-tiered cubby, then we can can indicate which k-tiered cubby it is in using

$$O(\log(\log^{(k)} n)^2) = O(\log^{(k+1)} n)$$

bits; and if a key x is in a j-tiered cubby for some $j < k$, then we can indicate which cubby x is in using

$$O\left(\log(k-j) + \log \frac{(\log^{(j)} n)^2}{(\log^{(j+1)} n)^2}\right) = O\left(\log(k-j) + \log^{(j+1)} n\right)$$

bits. Since $k - j \le \log^* N - j = \Theta(\log^* \log^{(j)} n) = O(\log^{(j+1)} n)$, the number of bits needed to indicate which cubby x is in can be upper-bounded by

$$O(\log^{(j+1)} n).$$

In general, keys that are in lower-tiered cubbies require more bits of metadata than those that are in higher tiers; but since there aren't very many low-tier keys, the total amount of metadata will remain small. (We'll see the full analysis of space consumption later in the section.)

Enforcing the invariants. We enforce the saturation invariant as follows. Whenever a deletion occurs in some non-tail cubby s, we move one of the elements from the tail to that cubby s. Whenever an insertion occurs in the facility, we place the new element into the tail. Whenever the tail fills up, we create a new tail, and whenever the tail empties out, we eliminate that cubby, and declare another one of the 1-tiered cubbies to be the new tail.

For each $j \in [k-1]$, define $t_j = \frac{(\log^{(j)} n)^2}{(\log^{(j+1)} n)^2}$ to be the target number of j-tiered cubbies. This means that t_j is also the number of j-tiered cubbies whose aggregate size equals one $j+1$-tiered cubby. Let $r_j = K/(\log^{(j)} n)^2$ be the size of a j-tiered cubby.

To enforce the distribution invariant, we must accommodate for the fact that new 1-tiered cubbies are being added and removed over time. In general, for each $j \in [k-1]$, whenever the number of j-tiered cubbies falls below $t_j/2$, we take one of the $j+1$-tiered cubbies and rebuild it as t_j cubbies with tier j (this is a ***j-creation rebuild***). And whenever the number of j-tiered cubbies rises above $3t_j$, we take t_j cubbies with tier j and rebuild them as a single $j+1$-tiered cubby (this is a ***j-destruction rebuild***).

We will describe how to deamortize these rebuilds (without compromising time or space efficiency) shortly. For now, let us simply observe that we only need to perform at most one j-creation rebuild for every $\Theta(r_{j+1})$ insertions that occur and we only need to perform at most one j-destruction rebuild for every $\Theta(r_{j+1})$ deletions that occur. By Lemma 126, each j-creation rebuild and each j-destruction rebuild can be performed in time $\Theta(r_{j+1})$, with high probability in n. It follows that, for each j, the amortized time cost of the j-rebuilds is $O(1)$ per insertion/deletion. Since there are k levels, the amortized time cost of all rebuilds is $O(k)$ per operation.

When enforcing the saturation and distribution invariants, there is one technical subtlety that we must be careful about. Whenever we move an element from the tail cubby to another cubby, we should always choose that element at random[11]; and whenever we perform a j-destruction rebuild, we should partition the elements in the $j+1$-tiered cubby randomly across the t_j new j-tiered cubbies being created. Call the random bits used to perform these choices ***the non-hash randomness***. Importantly, our use of non-hash randomness ensures that that for any given key x, the choice of which cubby it is currently in (within the facility that it hashes to) is

[11]Note the it takes constant time to choose a random key in the tail cubby, since the tail cubby consists of only a $O(1/\log n)$-fraction of the keys in the facility, so we can afford to use an extra $\log n$ bits per key in the tail in order to maintain an auxiliary random-choice data structure that lets us select random keys. In fact, at any given moment, we should maintain a random-choice data structure for both the tail and $\Theta(1)$ other cubbies in the same tier; this ensures that when one tail gets eliminated, another is ready to use. The constructions of the random-choice data structures are straightforward to deamortize to take $O(1)$ time per insertion/deletion.

always a function exclusively of the sequence of operations that has been performed and of non-hash randomness, and it is *not* a function of the hash functions used to perform insertions/deletions in cubbies.

Finally, we must describe how to deamortize the j-creation and j-destruction rebuilds for all $j \in [k-1]$. A critical observation here is that all of the facilities have almost exactly the same sizes as each other at any given moment. Indeed, assuming that K is sufficiently large in $\operatorname{polylog} n$, then a Chernoff bound tells us that all of the facilities have the same number of elements as each other up to a factor of $1 \pm 1/\operatorname{polylog} n$. Thus, we can synchronize the rebuilds for the facilities, so that whenever we perform a j-creation or j-destruction rebuild, we are actually performing a rebuild on all of the facilities at once over the course of $\Theta(r_{j+1}N/K)$ operations. When this happens, we perform the rebuild on one facility at a time (so it doesn't matter whether we perform the rebuild space efficiently); when we are performing a rebuild on a facility, some insertions/deletions on that facility may occur concurrently, but with high probability in n those operations will affect a total of $O(1)$ distinct keys, and thus can easily be incorporated into the rebuild. Each j-creation and j-destruction takes total time $\Theta(r_{j+1}N/K)$ across all facilities, and we only have to perform such a j-creation or j-destruction once every $\Theta(r_{j+1}N/K)$ insertion/deletions on the hash table. Thus, for each j, we can spread out the work of performing j-creations/destructions to be $O(1)$ time per insertion/deletion. Summing over $j \in [k-1]$, this amounts to $O(k)$ work per insertion/deletion.

This concludes the discussion of how to correctly enforce the two invariants without affecting space efficiency, and with only $O(k)$ extra time being spent per insertion/deletion.

Analyzing space efficiency. We first analyze the total space consumed by cubbies, and then we analyze the total space consumed by the mini-arrays D in each facility.

Within a given facility, all of the cubbies are completely full except for the tail. The tail cubby takes total space at most $O(K/\log n)$ machine words, which equals $O(K)$ bits; thus the total space consumed by tails adds only $O(1)$ bits per key in the hash table. The cubbies that are full can be analyzed exactly as in Lemma 124, allowing us to conclude that their total space consumption is $nw + O(n \log^{(k)} n)$ bits.

Now let us analyze the space consumption of the mini-array D within a given facility. For each key x we store which cubby it is in. As discussed earlier in the section, if the key is in a j-tiered cubby for some j, then it takes only $\Theta(\log^{(j+1)} n)$ bits to encode which cubby the key is in (note that this is always at most $(\log \log n)$ bits, which means that it can be encoded as an f-value in a local query router). On the other hand, for each $j \in [k-1]$, the fraction of keys that are in a j-tiered cubby is $O(1/(\log^{(j+1)} n)^2)$. Thus the average size of the metadata in D that we are storing for each key x is

$$\begin{aligned}
&O(\log^{(k+1)} n) + \sum_{j<k} O\left(\frac{\log^{(j+1)} n}{(\log^{(j+1)} n)^2}\right) \\
&= O(\log^{(k+1)} n) + O(1) \\
&= O(\log^{(k)} n)
\end{aligned}$$

bits. By the same argument as in Lemma 124 (for bounding the total amount of space used by local query routers in cubbies), we can deduce that, across the entire data structure, the total space used by mini-arrays D is $O(n \log^{(k)} n)$ bits, with high probability in n.

Thus, across the entire data structure, the total space consumption is

$$nw + O(n \log^{(k)} n)$$

bits, as desired.

Supporting large changes in size. So far we have focused exclusively on the case where the average size of each cubby stays within the range $[K, 2K]$.

We can generalize this to support a larger range of sizes with the following approach. Every time that the hash table's size changes by a constant factor, we move all of the elements from the current hash table H_1 into a new hash table H_2 whose capacity is twice as large (resp. small) as that of H_1. This means that each facility (resp. pair of adjacent facilities) in H_1 becomes a pair of adjacent facilities (resp. single facility) in H_2. The transformation from H_1 to H_2 takes $O(nk)$ time, and can be spread across $\Theta(n)$ insertions/deletions to take $O(k)$ time each. The transformation can also be performed space efficiently, by transforming one facility (resp. one pair of facilities) at a time, so that each key/value pair only takes up space in one of the two hash tables.

Putting the pieces together, we arrive at the following theorem:

Theorem 127. *One can construct a dictionary storing w-bit key/value pairs so that if n is the current number of keys and $k \in [\log^* n]$, insertions/deletions take time $O(k)$, queries take time $O(1)$, and, if $\log n = \Theta(w)$, the total space consumption is $wn + O(n \log^{(k)} n)$ bits. The running-time and space guarantees are with high probability in n.*

12.5.3 Succinctness Through Quotienting

In this section, we modify our hash table so that it can store n keys from a polynomial-size universe U in total space

$$\log \binom{|U|}{n} + \Theta\left(n \log^{(k)} n\right) = n \log \binom{|U|}{n} + \Theta\left(n \log^{(k)} n\right)$$

bits, where $\log\binom{|U|}{n}$ is the information-theoretical lower bound on the number of bits needed to store n elements from a universe of size $|U|$. If we are also storing λ-bit values for some $\lambda = O(\log n)$, then our total space consumption becomes

$$n \log \binom{|U|}{n} + n\lambda + \Theta\left(n \log^{(k)}\right)$$

bits.

A recap: the current structure of our hash table. Before we begin, let us briefly recap the structure of our hash table, and give names to the components and hash functions that are used. At any given moment, use N to denote the power-of-two range $[N, 2N]$ that the current table-size n is in.

The layout of our hash table is as follows: for some parameter $K = \operatorname{polylog} N$, there are N/K facilities, each of which contains a metadata mini-array D of size K. Call the metadata array in the i-th facility D_i, and think of the D_is as partitioning a larger array D^* with N entries. Each key x hashes to a random facility i, and then to a random slot in that facility's array D_i—equivalently, this means that each key x hashes to a random slot $g^*(x)$ in the array D^*.

Once a key x selects a facility, the key x is then assigned to one of the cubbies in the i-th facility (and the local query router $D^*[g^*(x)]$ stores which cubby x is assigned to).

Each cubby I, capable of storing $|I|$ keys, has the following layout. The cubby maintains two metadata mini-arrays A_I and M_I and a storage array S_I of size $|I|$; the mini-array A_I stores local query routers, the mini-array M_I stores metadata used to implement insertions/deletions in constant time, and the array S_I stores the actual key/value pairs. Each key x stored in the cubby hashes to some target position $g_I(x) \in [|I|]$, and this target position is used both to determine (a) which query-router in A_I handles key x, and (b) what x's target position is in the k-kick tree used to determine where keys go in S_I.

It turns out that, in this section, it will be important that the hash function g_I reuses the random bits from g^*, so $g_I(x)$ is the lowest-order $\log |I|$ bits of $g^*(x)$. As a convention, when discussing the bits of a number a, we will use $a[i, j]$ to refer to bits ranging from the i-th least significant bit to the j least significant bit, for $i \le j$. Thus $g_I(x) = g^*(x)[1, \log |I|]$. Also, using the same notation, if r is the facility that a key x is in, then $r = g^*(x)[\log K + 1, \log N]$.

Modifying the data structure to use quotienting. We can make our data structure more space efficient by using the ***quotienting*** technique [226]. Let π be a random permutation hash function on the universe U.[12] By applying π to each key that we are storing, we can assume that the keys being stored form a random subset of U.

[12]Recall that in Section 12.2.4, we discussed how to simulate permutation hash functions that are poly(n)-independent.

Since the keys x are random, we can define $g^*(x)$ to simply be the first $\log N$ bits of x, that is, $g^*(x) = x[1, \log N]$. We then make two modifications to our data structure: first, we modify each cubby so that its storage array stores only the final $\log U - \log N$ bits $x[\log N + 1, \log |U|]$ of each key x (as well as the λ-bit value stored with the key); second we add additional metadata (which we will describe in a moment) so that, for each key x, we can recover the first $\log N$ bits of x despite the fact that we are not explicitly storing them.

The metadata that we add is the following. For each cubby I, we add a new mini-array B_I of size $|I|$. If $S_I[i]$ is empty (there is no key stored there), then $B_I[i]$ stores nothing. Otherwise, if x is the key being stored in $S_I[i]$, then $B[i]$ stores the following two quantities: (a) the difference $g_I(x) - i$ and (b) the $\log(K/|I|)$ bits $x[\log |I|+1, \log K]$ of $g^*(x)$. The first quantity can be added to i to reconstruct $g_I(x) = x[1, \log |I|]$. The second quantity can be appended to $g_I(x)$ to obtain $g(x)[1, \log K] = x[1, \log K]$. And finally, the facility number can be used to recover the final $\log N - \log K$ bits of $g^*(x)$ (i.e., if we are in facility r then $x[\log K + 1, \log N] = g^*(x)[\log K + 1, \log N] = r$). This means that we can reconstruct the entire quantity $g^*(x) = x[1, \log N]$, which are the bits of x that we do not explicitly store.

To analyze our new data structure, there are two tasks that we must complete. The first is to analyze the amount of space used by the B_I's. The second is to handle the fact that the hash function $g^*(x)$ is no longer a fully independent hash function, and instead contains small negative correlations between keys (by virtue of being a permutation hash function).

Analyzing space consumption. Let R be the total number of bits used to store the metadata mini-arrays B_I for each cubby I. Then the total space consumed by our data structure is

$$n(\log N - \log U + \lambda) + O(R + n \log^{(k)} n) = n \log \binom{|U|}{n} + n\lambda + O\left(R + n \log^{(k)} n\right).$$

Thus, we wish to show that $R \le O(n \log^{(k)} n)$. To prove this, we will show that the metadata stored by the B_I's takes at most as much total space as the metadata stored in other mini-arrays in the data structure. Since we have already bounded the space consumption of those other mini-arrays by $O(n \log^{(k)} n)$ bits (with high probability), it follows that the same bound applies to the B_I's.

For each key x in some position i, B_I stores the quantity $g_I(x) - i$. Notice, however, that the number of bits needed to store this quantity is simply the probe complexity of x in the cubby. The same number of bits are already stored in A_I in order to route queries.

The second quantity that B_I stores for x is $\log(K/|I|)$ bits of $g^*(x)$. If $|I|$ has size $K/\log^{(j)} n$, then this quantity is $\Theta(\log^{(j+1)} n)$ bits. But that's the same number of bits that are already being stored for x in the metadata mini-array D^*.

Thus the total space consumed by our hash table is

$$n \log \binom{|U|}{n} + n\lambda + O\left(n \log^{(k)} n\right)$$

bits.

Proving correctness in the face of negative dependencies. Our final task is to handle the fact that the hash function g^* is not a fully independent hash function. Recall that we generate g^* by first performing a random permutation π on the universe (so each $x \in U$ is mapped to $\pi(x)$), and then setting $g^*(x)$ to be $\pi(x) \mod N$. The problem is that the $\pi(x)$'s form a random subset of U *without replacement.* This means that the quantities $\pi(x)$ are necessarily unique, which prevents them from being totally independent from one another.

There are three places in our analysis where we must be careful:

- For the k-kick tree analysis from Theorem 109 to apply to each cubby, we need to be able to apply a Chernoff bound to the number of keys x in the cubby that have $g_I(x)$ in any given sub-interval $J \subseteq [|I|]$.

- In order so that each local query router in D^* (and each local query router in each A_I) stores metadata for at most $O(\log n / \log \log n)$ keys, we need to be able to apply a Chernoff bound to the number of keys that hash to a given local query router.

- So that the facilities can all be resized simultaneously, we need each facility to contain at most $(1 + 1/\operatorname{polylog} n)K/n$ keys, at any given moment, with high probability in n. In other words, we need to be able to apply Chernoff bounced the number of keys x that hash to a given facility.

In summary, for any subset $Q \subseteq [N]$, we need to be able to apply a Chernoff bound to the number of keys x satisfying $g^*(x) \in Q$. Or, more generally, for any subset $Q \subseteq U$, we need to be able to apply a Chernoff bound to the number of keys x that satisfy $\pi(x) \in Q$.[13]

Fortunately, the indicator random variables Q_x indicating whether $\pi(x) \in Q$ are negatively correlated. That is, for any subset $R \subseteq [U]$ we have

$$\Pr\left[\prod_{x \in R} Q_x = 1\right] \leq \prod_{x \in R} \Pr[Q_x = 1] = \left(\frac{|Q|}{|U|}\right)^{|R|}.$$

[13]Technically, we also want to be able add conditions to this as follows: we want to be able to apply a Chernoff bound to the number of keys $\pi(x) \in Q$, having already conditioned on which keys $x \in U$ satisfy $\pi(x) \in Q'$ for some $Q' \supset Q$. (This lets us split each cubby into polylog n chunks that are analyzed independently.) These conditioned Chernoff bounds can be achieved in the same way as the condition-free Chernoff bounds.

Chernoff bound are known to hold for indicator random variables satisfying this type of negative correlation (see, e.g., Section 1.3.1 of [212]). Thus, our high-probability analysis from the previous sections continues to hold without modification.

Putting the pieces together, we arrive at the following theorem:

Theorem 128. *Suppose we wish to store key/value pairs where keys are from a universe U, and values are $\lambda \le O(\log |U|)$ bits. Assume a machine-word size of $\Omega(\log |U|)$ bits and let $k \ge 0$.*

One can construct a dictionary that supports insertions/deletions in time $O(k)$, that supports queries in time $O(1)$, and that offers the following guarantee on space: if the current number of keys is n, and $\log |U| = \Theta(\log n)$, then the total space consumption is

$$n \log \binom{|U|}{n} + n\lambda + O\left(n \log^{(k)} n\right)$$

bits. The running-time and space guarantees are with high probability in n.

12.6 Large Keys, Small Keys, and Filters

In this section, we give three extensions of Theorem 128. In Subsection 12.6.1, we consider the setting in which key/value pairs are $w = \omega(\log n)$ bits, and we show that the guarantee of $O(\log^{(k)} n)$ wasted bits per key can be extended to any $w \in n^{o(1)}$. In Subsection 12.6.2, we consider the setting in which key/value pairs are very small, taking a total of $\log n + s$ bits for some $s = o(\log n)$. We show that, if s is even slightly sublogarithmic, then it is possible to reduce the number of wasted bits per key all the way to $o(1)$. Finally, in Subsection 12.6.3, we apply our results to the problem of constructing space-efficient dynamic filters.

Our results on very-large and very-small keys both hinge on novel reductions that transform the the very-large/very-small case into the standard $\Theta(\log n)$-bit case. These reductions may be independently useful in future work.

12.6.1 Supporting Large Keys/Values

So far we have restricted ourselves to the case where keys/values are $O(\log n)$ bits each. We prove the following theorem:

Theorem 129. *Suppose we wish to store key/value pairs where keys are from a universe $U = [2^a]$ and values are b bits. Assume a machine-word size of $\Omega(a + b)$ bits and let $k \ge 0$.*

One can construct a dictionary that supports insertions/deletions in time $O(k)$, that supports queries in time $O(1)$, and that offers the following guarantee on space: if the current number of keys is n, and $a + b \le n^{o(1)}$, then the total space consumption is

$$n \log \binom{|U|}{n} + nb + O\left(n \log^{(k)} n\right)$$

bits. The running-time and space guarantees are with high probability in n.

Historically, the task of supporting universes U of superpolynomial size has proven to be quite difficult. Indeed, the best known guarantees for worst-case constant-time hash tables [52, 245] use

$$\log \binom{|U|}{n} + \Omega(\min(|U|^{\Omega(1)}, n \log n))$$

bits of space,[14] and the best known guarantee for constant expected-time hash tables [313] uses

$$\log \binom{|U|}{n} + \Omega(n \log(a + b))$$

bits of space. In contrast, Theorem 129 says that, as the size of the universe grows, the number of wasted bits per key does not—it remains $O(\log^{(k)} n)$ bits per key even for very large keys/values.

There are several reasons that large universes are difficult to handle. One difficulty is that it is not known how to efficiently perform quotienting on keys from large universes. For any universe U, it is known how to construct an efficient family of n^{ε}-wise $(1/\operatorname{poly}(n))$-dependent permutations [218, 250, 277]—but each member $\pi \in \Pi$ requires $|U|^{\Omega(1)}$ bits to store, so if $|U|$ is super-polynomial, then we cannot store π in polynomial space. This has prevented past work [52, 75, 245] from using quotienting on large universes. Another difficulty [52, 75, 245] with large universes is that we can no longer afford to store lookup tables of size $|U|^{\Omega(1)}$—this is a serious bottleneck for any hash table that uses the Method of Four Russians (including the hash tables in this chapter) as a path to worst-case constant-time operations. Finally, even if both of these issues were eliminated, the known techniques [52, 245, 313] for constructing succinct hash tables would still incur an asymptotic blow-up in wasted-bits-per-key as the universe size grows.

In this section, we present a new approach for handling large universes that lets us get around all of the above issues at once. Define an (a, b)***-dictionary*** to be a dictionary that stores a-bit keys with b-bit values. We prove that there is an efficient (high probability) reduction from the problem of constructing a succinct (a, b)-dictionary to the problem of constructing a succinct $(\Theta(\log n), a + b - \Theta(\log n))$-bit dictionary. This means that we can always assume that keys are $\Theta(\log n)$ bits. Once we have proven this reduction, our only challenge will be to handle large values, which it turns out will be relatively straightforward using the techniques we have already developed.

The reduction is captured in the following theorem.

[14]The result of Liu et al. [245] uses $\log \binom{|U|}{n} + |U|^{\Omega(1)}$ bits of space and the result of Arbitman et al. [52] uses $\log \binom{|U|}{n} + \Omega(n \log n)$ bits of space. The solution that Arbitman et al. use is to just hash elements to a polynomial-size universe—if one is willing to allow for query-correctness to be violated a $1/\operatorname{poly} n$ fraction of the time, then it suffices to store only these hashes, but otherwise one must also store the entire original element.

Theorem 130. *Let $a, b, N, \gamma \in \mathbb{N}$ be parameters, and let $\gamma > 0$ be a sufficiently large constant. Suppose that $a \geq \gamma \log N$, and suppose that the machine-word size w satisfies $w \geq \Omega(a + b)$. Finally let $f_N(n)$ be a non-negative non-decreasing function.*

Suppose we have a dynamically resizable $(\gamma \log N, a + b - \gamma \log N + 1)$-dictionary S that is capable of storing n key-value pairs in space $n(a + b - \log n) + f_N(n)$ bits for any $n \in [N]$, while offering running-time guarantees that are high-probability in N. Then we can construct a dynamically resizable (a, b)-dictionary L that is capable of storing n key-value pairs in space $n(a + b - \log n) + f(n) + O(\sqrt{N})$ bits, for any $n \in [N]$, and that, with high probability in N, takes at most $O(1)$ more time per operation than does S.

Proof. To capture the fact that γ is at least a sufficiently large positive constant, we shall treat γ as an asymptotic variable. We implement L with three data structures:

- The first data structure is the $(\gamma \log N, a + b - \gamma \log N + 1)$-dictionary S;
- The second data structure is a dynamically resizable $(\sqrt{\gamma} \log N, \log N)$-dictionary D_1 that can store r keys/value pairs in $O(r\sqrt{\gamma} \log N + \sqrt{N})$ bits of space for any $r \leq N$, and that supports constant-time operations with high probability in N;
- The third data structure is a $(\sqrt{\gamma} \log N, a + b - \gamma \log N + \sqrt{\gamma} \log N)$-dictionary D_2 that can store r key/value pairs in $\Theta(r\sqrt{\gamma} \log N + \sqrt{N}) + r(a + b - \gamma \log N + \sqrt{\gamma} \log N)$ bits of space for any $r \leq N$, and that supports constant-time operations with high probability in N.

Let us briefly comment on how to implement D_1 and D_2. Notice that D_1 does not need to be succinct (or even compact)—it can be implemented using any standard constant-time dictionary that supports load factor $\Omega(1)$. To implement D_2, we store values with a layer of indirection, meaning that the dictionary allocates separate $(a + b - \gamma \log N + \sqrt{\gamma} \log N)$-bit chunks of memory for each individual value, and stores a $\Theta(\log N)$-bit pointer to that value. The $(\sqrt{\gamma} \log N, O(\log N))$-dictionary that stores the keys and the pointers can again be implemented with any standard constant-time dictionary that supports load factor $\Omega(1)$.[15]

For any key $x \in [2^a]$, define the ***core*** $\phi(x)$ to be the first $\gamma \log N$ bits of x, and define $\psi(x)$ to be the final $|x| - \gamma \log N$ bits of x. So $\phi(x) \circ \psi(x) = x$.

We will use S to store key/value pairs of the form $(\phi(x), \psi(x) \circ y \circ \ell)$ where $\ell \in \{0, 1\}$ is an extra bit of information that we call the ***abundance bit***. If there is exactly one key $x \in L$ with a given core $\phi(x)$, then S stores $(\phi(x), \psi(x) \circ y \circ 0)$. If there is more than one key x with a given core $j = \phi(x)$, then S stores $(\phi(x), \psi(x) \circ y \circ 1)$ for *one* such key/value pair (x, y). The fact that the abundance bit is set to 1 in this latter case indicates that there are also other key/value pairs (x', y') satisfying

[15] We remark that the $\sqrt{N}$ term in the space-usage for D_1 and D_2 stems from the fact that, in order for the probabilistic guarantees of D_1 and D_2 to be high-probability in N, we need D_1 and D_2 to be size at least $\operatorname{poly} N$.

$\phi(x') = \phi(x)$. In this case, we say that the core $\phi(x)$ is ***rabid***, and any key $x' \neq x$ that has core $\phi(x') = \phi(x)$ is also said to be a ***rabid key*** (regardless of whether x' is a member of the dictionary L that we are constructing).

Rabid keys $z \in L$ (and their corresponding values w) are stored as follows. Let $h(\phi(z)) \in [\sqrt{\gamma} \log N]$ be a pairwise-independent hash, and let $g(z) \in [\sqrt{\gamma} \log N]$ also be a pairwise-independent hash. We store the pair $(g(z), h(\phi(z)) \circ \psi(z) \circ w)$ in D_2. If we ever attempt to insert a key $g(z)$ into D_2 that is already there (i.e., we have a collision), then we rebuild the entire data structure with new random bits (this only occurs with probability $1/\operatorname{poly} N$ per insertion). Finally, for each rabid core $\phi(z)$, we store the pair $(h(\phi(z)), q)$ in D_1, where q is a reference counter keeping track of the number of rabid keys in L have that core.

The data structure D_1 has two purposes. The first (and less important) purpose is to maintain the reference counter q so that, on deletions, we know when there are no longer any rabid elements with a given core $\phi(z)$, at which point we can both remove $h(\phi(z))$ from D_1 and we can set the abundance bit for $\phi(z)$ in S to be 0.

The more important purpose of D_1, however, is to detect collisions between $h(\phi(z))$ and $h(\phi(z'))$ for rabid keys z, z' satisfying $\phi(z) \neq \phi(z')$. Whenever we insert some rabid key z, we can tell based on the abundance bit for $\phi(z)$ (prior to the insertion) whether this is the first rabid key in L to have core $\phi(z)$; if it is the first such rabid key, but there is already a pair of the form $(h(\phi(z)), q)$ in D_1, then that means a collision on h has occurred, and we rebuild the entire data structure from scratch. Such collisions occur with probability only $1/\operatorname{poly} N$ per insertion.

What D_1 ensures is that h is injective on the set of rabid cores—that is, if $\phi(z)$ and $\phi(z')$ are two different rabid cores, then $h(\phi(z)) \neq h(\phi(z'))$. It follows that there is a bijection between rabid keys $z \in U$ and pairs $(h(\phi(z)), \psi(z))$. Thus D_2 can be used to perform queries on rabid keys z as follows: check if there is a key-value pair of the form $(g(z), h(\phi(z)) \circ \psi(z) \circ w)$; if there is, then return value w, and otherwise declare that z is not present.

It is straightforward to perform insertion/deletion/queries using S, D_1, D_2. Note that, if a non-rabid key x is deleted from L, but there is a rabid key x' in L that has the same core $\phi(x') = \phi(x)$, then we must move one such rabid key x' out of (D_1, D_2) and into S (so that x' is no longer rabid).

Our final task is to analyze the space consumption of our data structure L. We will use r to denote the number of rabid keys in L, and we will use k to denote the number of non-rabid keys in L (so $n = k + r$). Since γ is a sufficiently large constant, the data structures D_1 and D_2 collectively use

$$O(\sqrt{N}) + O(r\sqrt{\gamma} \log N) + r(a + b - \gamma \log N + \sqrt{\gamma} \log N)$$

bits of space, where the final term accounts for the space consumed by the values of D_2, and the first two terms account for the space consumed by the rest of D_1, D_2. Using the fact that γ is a sufficiently large positive constant, it follows that D_1 and D_2 collectively use

$$O(\sqrt{N}) + r(a + b - \log n)$$

bits. The data structure S, on the other hand, uses

$$k(a+b-\log n)+f(k) \leq k(a+b-\log n)+f(n)$$

bits. The total number of bits used is therefore

$$n(a+b-\log n)+f(n)+O(\sqrt{N}).$$

∎

The result of Theorem 130 is that we can always assume without loss of generality that our keys are size $O(\log n)$ bits. Thus, in order to prove Theorem 129, it suffices to construct an $(O(\log n), b)$-dictionary, where the only constraint on b is $b \leq n^{o(1)}$ (and where the machine-word size w satisfies $w \geq \Omega(\log n + b)$).

Storing large values space efficiently. To store large values, we exploit an interesting feature of the dynamically resizable dictionary that we constructed for the proof of Theorem 128: in each facility, all of the cubbies except for the tail are *completely full.* Thus, for each cubby I (except for the tail), we can allocate bI bits of space V_I to store values—the key stored in the j-th position of I has its value stored in bits $(j-1)b+1, \ldots, jb$ of V_I. Importantly, V_I is fully saturated, so it wastes no space.

To handle the keys/values in the tail, recall that the tail consists of less than an $O(\frac{1}{\log n})$-fraction of the keys in the cubby. Thus, we can individually allocate space for each value in the tail, and we can store a $\Theta(\log n)$-bit pointer to that value. The $\Theta(\log n)$-bit pointers contribute only $O(1)$ amortized bits of space overhead per key in the data structure.

One technical detail that we must be careful about is that, whenever an cubby toggles between being/not-being the tail, we must change how the values are stored in that cubby. This is straightforward to do in a deamortized fashion using the same deamortized-rebuilding techniques as in Section 12.5.2.

In summary, we have the following lemma:

Lemma 131. *Suppose we wish to store key/value pairs where keys are from a universe $U = [2^a]$, values are b bits. Suppose $a = O(\log n)$ and $b \leq n^{o(1)}$. Assume a machine-word size of $\Omega(a+b)$ bits. Finally, let $k \geq 0$.*

One can construct a dictionary that supports insertions/deletions in time $O(k)$, that supports queries in time $O(1)$, and that offers the following guarantee on space: if the current number of keys is n, then the total space consumption is

$$n \log \binom{|U|}{n} + nb + O\left(n \log^{(k)} n\right)$$

bits. The running-time and space guarantees are with high probability in n.

Combined, Theorem 130 and Lemma 131 imply Theorem 129.

12.6.2 Optimizing for Very Small Keys

In this section we consider the case of very small keys, that is keys of size $\log n + o(\log n)$ bits. For most of the section we shall focus exclusively on dictionaries that store keys without values, but at the end of the section we will also generalize to the case where the dictionary also stores very small values.

We begin with the fixed-capacity case. We show that, if keys are of size $\log n + s$ for some $s \le \log n / \log\log \cdots \log n$ (where there are a constant number of logarithms), then it is possible to construct a constant-time dictionary with $o(1)$ wasted bits per key.

Theorem 132. *Let k, n be parameters, where $k \in [\log^* n]$. Let $\phi = \Theta(\log^{(k)} n)$. Let $U = [2^{\log n + s}]$ for some s satisfying $s \in \omega(1) \cap o(\log n)$ and suppose that $s \le O(\log n/\phi)$.*

There exists a fixed-capacity dictionary that stores up to n keys from U at a time, that supports insertions/deletions in time $O(k)$ (with high probability in n), that supports queries in time $O(1)$, and that uses a total of

$$\log \binom{|U|}{n} + o(n)$$

bits of space (with high probability in n).

We will assume without loss of generality that $\phi \le \log\log n$ and that ϕ is rounded to the nearest power of two. We will also assume without $\phi = \omega(1)$, since the fact that $s = o(\log n)$ ensures that the theorem requirements hold for some $\phi = \omega(1)$. And finally, we will assume without loss of generality that $s \le o(\log n/\phi)$, since we otherwise can replace ϕ with $\phi' = \Theta(\log^{(k+1)} n)$ and prove the theorem for the new ϕ'.

We begin by describing the data structure. First, we assume that the keys form a random subset of U; as noted in Section 12.2.4, it is known how to construct permutation hash functions that simulate this assumption while preserving time and space guarantees.

We use the first $\log(n/\phi)$ bits of each key to assign it to a random one of n/ϕ bins $R_1, \ldots, R_{n/\phi}$. We maintain an array A of n/ϕ $O(\log \phi)$-bit counters $A_1, \ldots, A_{n/\phi}$, where each A_i is always in the range $[0, 100\phi]$. Whenever we insert an element x that maps to some bin R_i, we examine the counter A_i. If $A_i < 100\phi$, then we declare x to be ***standard*** and we increment A_i; otherwise, we declare x to be ***non-standard***, and we leave A_i unchanged. Similarly, we decrement A_i whenever we delete a standard element x that belongs to bin R_i, but we do not decrement A_i when we delete a non-standard element.

We take different approaches to storing standard versus non-standard elements. Non-standard elements are stored in a secondary ***backyard hash table*** B, constructed via Theorem 128 to incur $O(\log^{(k+1)} n)$ wasted bits per key. The large number of wasted bits per key is okay because only a small number of elements will reside in B.

Standard elements, on the other hand, are stored as follows. For any given bin R_i, we encode the set of standard elements that reside in that bin using a single $o(\log n)$-bit integer E_i (we will describe how to construct E_i later). Importantly, the number of bits used for E_i is a strict function of the number A_i of elements being encoded.

Since different E_is have different sizes, we cannot store them contiguously in an array. Instead, we again make use of Theorem 128. We maintain 100ϕ dynamically-resized hash tables $H_1, \ldots, H_{100\phi}$, each of which is parameterized to incur $O(\log^{(k+1)} n)$ wasted bits per key. For each bin R_i, we store the key-value pair (i, E_i) in hash-table H_{A_i}. Notice that this construction ensures that each hash table H_i storing fixed-size keys and values. Also notice that, even though the H_is incur a relatively large number of wasted bits per key, there are only $O(n/\phi)$ total elements in the H_is, one for each of the n/ϕ bins.

To complete the description of the data structure, we show that it is possible to encode each E_i space efficiently.

Lemma 133. *For any bin R_i, the set of standard keys in that bin can be encoded using $\log \binom{\phi 2^s}{A_i} + O(1) = o(\log n)$ bits. Moreover, the encodings can be updated/queried in constant time using $O(\sqrt{n})$ bits of metadata.*

Proof. Recall that keys are $\log n + s$ bits. All of the keys in R_i agree on their first $\log(n/\phi)$ bits, so they differ in only their final $s + \log \phi$ bits. The number of possibilities for the A_i keys encoded by E_i is

$$\binom{\phi 2^s}{A_i}.$$

If we can show that this is $2^{o(\log n)}$, then the lemma follows from the Method of Four Russians (see discussion in Section 12.4). To complete the proof, observe that

$$\begin{aligned}\binom{\phi 2^s}{A_i} &\leq \binom{(\log\log n) 2^{o((\log n/\phi))}}{O(\phi)} \\ &\leq \log\left((\log\log n) 2^{o(\log n/\phi)}\right)^{O(\phi)} \\ &\leq (\log\log n)^{O(\log\log n)} 2^{o(\log n)} \\ &\leq \log 2^{o(\log n)}.\end{aligned}$$

∎

We now proceed to bound the space consumption of the hash table. We begin with a simple approximation for binomial coefficients.

Lemma 134. *For all $a = \omega(b)$, we have*

$$\log \binom{a}{b} = b \log a - b \log b + b \log e \pm o(b),$$

and for all $a \geq b$, we have

$$\log \binom{a}{b} \leq b \log a - b \log b + b \log e + o(b).$$

Proof. If $a = \omega(b)$, then

$$\begin{aligned}
\log \binom{a}{b} &= \log \left(\frac{a \cdot (a-1) \cdots (a-b+1)}{b!} \right) \\
&= \log \left(\frac{a^b}{b!} \right) \pm o(b) \\
&= b \log a - \log(b!) \pm o(b) \\
&= b \log a - b \log b + b \log e \pm o(b),
\end{aligned}$$

where the final step follows from Stirling's inequality. By a similar sequence of arguments, if $a \geq b$, then

$$\begin{aligned}
\log \binom{a}{b} &= \log \left(\frac{a \cdot (a-1) \cdots (a-b+1)}{b!} \right) \\
&\leq \log \left(\frac{a^b}{b!} \right) \\
&= b \log a - \log(b!) \pm o(b) \\
&= b \log a - b \log b + b \log e \pm o(b).
\end{aligned}$$

∎

Next we bound the number of elements in the backyard, at any given moment.

Lemma 135. *With high probability in n, the number of non-standard elements is $O(n/2^{\phi})$ at any given moment.*

Proof. Break the bins $R_1, R_2, \ldots, R_{n/\phi}$ into $m = (n/\phi)/\operatorname{polylog} n$ collections $C_1, \ldots, C_m$, each consisting of some number $\ell = \operatorname{polylog} n$ of bins. The assignments of keys to bins are negatively correlated, so we can use a Chernoff bound for negatively correlated random variables [212] to deduce that, with high probability in n, the number of keys assigned to any given collection is at most $2\ell\phi$ at any given moment.

Let $a_1, \ldots, a_q$ be the set of keys currently present. By a union bound, we have that with high probability in n, every chunk C_i had at most $2\ell\phi$ keys assigned to it during each of the time steps in which $a_1, \ldots, a_q$ were inserted. Condition on this being the case, and further condition on which specific keys hash to which C_is.

Define $c_1, \ldots, c_m$ so that c_i is the number of non-standard keys currently in C_i. Having conditioned on which keys hash to which collections, the c_is are independent. We will show that $\mathbb{E}[c_i] = O(\ell\phi/2^{\phi})$, meaning that $\mathbb{E}[\sum_i c_i] = O(n/2^{\phi})$. Since each c_i is guaranteed to be at most $2\ell\phi = \operatorname{polylog} n$, we can apply a Chernoff bound to the sum $\sum_i c_i$ to deduce that the total number of non-standard elements is $O(n/2^\phi)$ with high probability.

We conclude the proof by establishing that $\mathbb{E}[c_i] = O(\ell/2^\phi)$. By linearity of expectation, it suffices to show that any given key x has a $O(1/2^\phi)$ probability of being non-standard. This follows by applying a Chernoff bound (again for negatively correlated random variables) to the number of keys that hash to x's bin. When x is inserted, there are at most $2\ell\phi$ keys in x's collection, each of which has a $1/\ell$ probability of hashing to x's bin, so by a Chernoff bound we have that the probability of there being already $100\phi - 1$ keys in x's bin is at most $1/2^\phi$. This completes the proof. ∎

We can now bound the total space consumed by the hash table.

Lemma 136. *The total space consumed by the hash table is* $\log\binom{n2^r}{n} + o(n)$ *bits, with high probability in* n.

Proof. The array A of counters uses $O((n/\phi)\log\phi) = o(n)$ bits. Let J_0 be the number of items in the backyard. Then the backyard uses space

$$\log\binom{n2^s}{J_0} + O(J_0 \log^{(k+1)} J_0)$$

bits. By Lemma 135, we have $J_0 \le O(n/2^\phi) = o(n/\log^{(k+1)} n)$, so the total space consumed by the backyard is at most

$$\log\binom{n2^s}{J_0} + o(n)$$

bits.

For $i \in [n]$, let J_i be the number of elements in each H_i. Notice that $\sum_{i=1}^{n} J_i = n/\phi$. Since H_i stores $\log n$-bit keys, stores $\log\binom{\phi 2^s}{A_i} + O(1)$-bit values ,and wastes $O(\log^{(k+1)} n)$ bits per key, the total space taken by a given H_i is

$$O(J_i \log^{(k+1)} n) + \log\binom{n}{J_i} + J_i \log\binom{\phi 2^s}{A_i}$$

bits.[16]

Putting the pieces together, the total space consumed by the hash table is

$$\sum_{i=1}^{100\phi} O(J_i \log^{(k+1)} n) + \sum_{i=1}^{100\phi} \log\binom{n}{J_i} + \sum_{i=1}^{n/\phi} \log\binom{\phi 2^s}{A_i} + \log\binom{n2^s}{J_0} + o(n).$$

[16]Here we ignore the $n^{1-\Omega(1)}$ total bits used for Method of Four Russians and for storing hash functions.

Since $\sum_i J_i = n/\phi = o(n/\log^{(k+1)} n)$, this is at most

$$\sum_{i=1}^{100\phi} \log \binom{n}{J_i} + \sum_{i=1}^{n/\phi} \log \binom{\phi 2^s}{A_i} + \log \binom{n 2^s}{J_0} + o(n).$$

Applying Lemma 134, the space is at most

$$\begin{aligned}
&\sum_{i=1}^{100\phi} J_i(\log n - \log J_i + O(1)) + \sum_{i=1}^{n/\phi} A_i \left(\log(\phi 2^s) - \log A_i + \log e + o(1)\right) \\
&\qquad\qquad\qquad + J_0 \left(\log(n2^s) - \log J_0 + O(1)\right) + o(n) \\
&= \sum_{i=1}^{100\phi} J_i(\log n - \log J_i) + \sum_{i=1}^{n/\phi} A_i \left(\log(\phi 2^s) - \log A_i + \log e\right) \\
&\qquad\qquad\qquad + J_0 \left(\log(n2^s) - \log J_0\right) + o(n) \\
&= \sum_{i=1}^{100\phi} J_i(\log n - \log J_i) + \sum_{i=1}^{n/\phi} A_i \left(\log(\phi 2^s) - \log A_i\right) \\
&\qquad\qquad\qquad + J_0 \left(\log(n2^s) - \log J_0\right) + n \log e + o(n).
\end{aligned}$$

By Jensen's inequality, the above quantity is maximized by setting $J_1, \ldots, J_\phi$s to be equal (so $J_i \geq \frac{n}{2\phi^2}$ for all i) and by setting $A_1, \ldots, A_{n/\phi}$ to be equal (so $A_i =$

$(n - J_0)/(n/\phi)$ for all i). Thus the number of bits used by the hash table is at most

$$\left(\sum_i J_i\right) \cdot \left(\log n - \log \frac{n}{2\phi^2}\right) + \left(\sum_i A_i\right) \cdot \left(\log(\phi 2^s) - \log \frac{n - J_0}{n/\phi}\right)$$
$$+ J_0 \left(\log(n2^s) - \log J_0\right) + n \log e + o(n)$$
$$= \frac{n}{\phi} \cdot \left(\log n - \log \frac{n}{2\phi^2}\right) + (n - J_0) \cdot \left(\log(\phi 2^s) - \log \frac{n - J_0}{n/\phi}\right)$$
$$+ J_0 \left(\log(n2^s) - \log J_0\right) + n \log e + o(n)$$
$$= \frac{n}{\phi} \cdot \Theta(\log \phi) + (n - J_0) \cdot \left(\log(\phi 2^s) - \log \frac{n - J_0}{n/\phi}\right)$$
$$+ J_0 \left(\log(n2^s) - \log J_0\right) + n \log e + o(n)$$
$$= (n - J_0) \cdot \left(\log(\phi 2^s) - \log \frac{n - J_0}{n/\phi}\right) + J_0 \left(\log(n2^s) - \log J_0\right) + n \log e + o(n)$$
$$= (n - J_0) \left(\log(\phi 2^s) - \log \phi\right) + J_0 \left(\log(n2^s) - \log J_0\right) + n \log e + o(n)$$
$$= (n - J_0)s + J_0 \left(\log(n2^s) - \log J_0\right) + n \log e + o(n)$$
$$= ns + J_0 \left(\log n - \log J_0\right) + n \log e + o(n).$$

Since $J_0 = o(n)$, this is

$$ns + n \log e + o(n),$$

which by Lemma 134 is

$$\log \binom{n2^s}{n} + o(n).$$

∎

Since insertions/deletions take time $O(k)$ and queries take time $O(1)$, the preceding lemma implies Theorem 132.

We conclude the section with several simple corollaries. The first corollary extends the theorem to store key-value pairs.

Corollary 137. *Let k, n be parameters. Let $\phi = \Theta(\log^{(k)} n)$. Let $U = [2^{\log n + s_1}]$ and $V = [2^{s_2}]$ for some s_1, s_2 satisfying $s_1 \geq \omega(1)$ and $s_1 + s_2 \leq o(\log n) \cap O(\log n/\phi)$.*

There exists a fixed-capacity dictionary that stores up to n keys from U at a time, each of which is associated with a value in V; that supports insertions/deletions in time $O(k)$ (with high probability in n); that supports queries in time $O(1)$; and that uses a total of

$$\log \binom{|U|}{n} + ns_2 + o(n)$$

bits of space (with high probability in n).

Proof. This follows from the same sequence of arguments as before, but now we associate values with keys as well. ∎

The second corollary extends the theorem to support dynamic resizing. Note that, since we are interested in keys whose lengths are very close to n, it does not make sense to talk about n changing by a large factor (indeed, this would take us out of the small-key regime). Thus, we focus on n in a range $[N/2, N]$ for some N.

Corollary 138. *Let k, N be parameters. Let $\phi = \Theta(\log^{(k)} N)$. Let $U = [2^{\log N + s_1}]$ and $V = [2^{s_2}]$ for some s_1, s_2 satisfying $s_1 \geq \omega(1)$ and $s_1 + s_2 \leq o(\log N) \cap O((\log N)/\phi)$.*

There exists a fixed-capacity dictionary that stores up to N keys from U at a time, each of which is associated with a value in V; that supports insertions/deletions in time $O(k)$ (with high probability in N); that supports queries in time $O(1)$; and that (with high probability in N) uses a total of

$$\log \binom{|U|}{n} + ns_2 + o(n)$$

bits of space if $n \in [N/2, N]$ is the number of keys currently present.

Proof. This guarantee is already true of our current data structure. Indeed, every component of the data structure except for the array A of counters is a dynamically-resizable hash table. The array A of counters takes $o(n)$ space, so we only need to worry about the total space consumed by the dynamically-resizable hash tables. The proof of Theorem 132 immediately extends to arbitrary $n \in [N/2, N]$ to bound the total space by

$$\log \binom{u2^{s_1}}{n} + ns_2 + o(n)$$

bits. ∎

12.6.3 Constructing Optimal Filters

In this section, we apply our results to the problem of constructing space-efficient filters. A filter has three parameters: a maximum capacity n, and a false positive rate ε (which we will assume is an inverse power of two), and a universe U of keys. A filter must support insertions/deletions/queries on a dynamic set $S \subseteq U$ of up to n keys. Unlike a dictionary, however, a filter is permitted to sometimes return false positives on queries: if a key $x \notin S$ is queried, the filter must correctly return that $x \notin S$ with probability $1 - \varepsilon$, but it is permitted to incorrectly return that $x \in S$ with probability ε.

Information theoretically, a static filter (i.e., a filter that supports only queries) must use at least $n \log \varepsilon^{-1}$ bits. It is known [249] that there exist values of ε for which a dynamic filter must use at least $n \log \varepsilon^{-1} + \Omega(n)$ bits, but it remains an open question whether there exists a dynamic filter that uses at most $n \log \varepsilon^{-1} + O(n)$ bits for all ε. We now establish that, as long as $\log \varepsilon^{-1}$ is slightly sublogarithmic in n, then such a dynamic filter does, in fact, exist. We also give extremely succinct filters for the setting where $\varepsilon^{-1} = \Theta(\log n)$, bringing the number of wasted bits per key to be the same as what we have achieved for the dictionary problem.

Reducing the filter problem to the dictionary problem. We begin by reviewing the standard technique for constructing a filter using a dictionary (see, e.g., [52, 82, 93, 122, 245, 294]). We hash keys $x \in U$ to $(\log n + \log \varepsilon^{-1})$-bit fingerprints $f(x)$. We store the fingerprints $\{f(x) \mid x \in S\}$ in a hash table, and to answer a query for a key x, we simply check whether $f(x)$ is in the hash table. If a key $x \notin S$ is queried, then the probability of a false positive is at most

$$\sum_{y \in S} \Pr[f(x) = f(y)] = n \cdot \frac{\varepsilon}{n} = \varepsilon.$$

Notice, however, that the fingerprints $\{f(x) \mid x \in S\}$ form a *multi-set*, rather than a set, so we cannot actually store them directly in a hash table. Our solution is to store one copy of each fingerprint in a hash table $\mathcal{A}$, and then to store any duplicate fingerprints in a secondary hash table $\mathcal{B}$ that is capable of supporting multi-sets. Whenever we insert a new key x, we first try to place $f(x)$ in $\mathcal{A}$, and if it is already there, we place it in $\mathcal{B}$; whenever we delete a key x, we first try to delete (one copy of) $f(x)$ from $\mathcal{B}$, and if it is not there, we delete it from $\mathcal{A}$; and whenever we query a key x, we can just check whether $f(x) \in \mathcal{A}$.

The hash table $\mathcal{B}$ will be significantly smaller than $\mathcal{A}$, meaning that it does not have to be highly space efficient. Thus we are able to use of past work on multi-set dictionaries to implement $\mathcal{B}$:

Lemma 139 (Theorem 1 of [93])**.** *Let $\varepsilon^{-1} \in [\omega(1), O(\log n)]$. There exists a high-probability constant-time hash table that stores an arbitrary multi-set of m keys in $(1 + o(1))m \log \varepsilon^{-1}$ bits.*

In fact, Lemma 139 is stronger than what we need—it would suffice for us to have a multi-set dictionary using $o(m\varepsilon^{-1})$ bits. Indeed, we can bound $m = |\mathcal{B}|$ by $O(\varepsilon n + \log n)$ with high probability:

Lemma 140. *At any given moment $D = |\{x \in S \mid f(x) = f(y) \text{ for some } y \in S \setminus \{x\}\}|$ satisfies $D = O(\varepsilon n + \log n)$ with high probability in n.*

Proof. Let $x_1, \ldots, x_n$ denote the keys in S, and let Y_i be the 0-1 random variable indicating whether $f(x_i) = f(x_j)$ for some $j < i$. Notice that $D \leq 2 \sum_i Y_i$.

The Y_is are independent, and each Y_i satisfies $\Pr[Y_i = 1] \leq \varepsilon$. Therefore we can apply a Chernoff bound to deduce that $D = O(\varepsilon n + \log n)$ with high probability in n. ∎

By Lemma 140, if $\varepsilon = o(1)$, then the total number of bits used by $\mathcal{B}$ is $o(n)$ with high probability in n. On the other hand, if $\mathcal{A}$ is implemented using a hash table that wastes r bits per key, then it uses a total of at most

$$\log \binom{n\varepsilon^{-1}}{n} + nr$$

bits. If we again assume that $\varepsilon = o(1)$, then by Lemma 134, this is equal to

$$n \log \varepsilon^{-1} + nr + n \log e$$

bits.

Applying Theorems 129 and 132 to construct $\mathcal{A}$, we arrive at the following result:

Theorem 141. *Let $\varepsilon^{-1} \in [\omega(1), O(\log n)]$ be an inverse power of two, and let $k \in [\log^* n]$ be a parameter. One can construct a filter that has false-positive rate at most ε, that supports queries in constant time, that supports insertions/deletions in time $O(k)$, and that uses space at most*

$$\begin{cases} n \log \varepsilon^{-1} + n \log e + o(n) & \text{if } \varepsilon^{-1} \leq \frac{\log n}{\log^{(k)} n} \text{ and } \log^{(k)} n = \omega(1) \\ n \log \varepsilon^{-1} + O(n) + O(n \log^{(k)} n) & \text{otherwise} \end{cases}$$

bits. The time and space guarantees hold for each operation with high probability in n.

We remark that the size $|U|$ of the universe does not matter, since we can use hash function with $O(\log n)$ description bits (see, e.g., Theorem 14 of [297]) to map $|U|$ to a universe of size $\operatorname{poly}(n)$, while avoiding collisions with probability $1 - 1/\operatorname{poly}(n)$. The $1/\operatorname{poly}(n)$ collision probability can then easily be absorbed into ε.[17]

[17] Notice that the same approach is not legal for hash tables, since the failure probability for a hash table must be with respect to *running time*, rather than with respect to *correctness*. That is, hash tables are never allowed to return false positives.

Chapter 13

A Hash Table Without Hash Functions

13.1 Introduction

A ***dictionary*** is any data structure that supports insertions, deletions, and queries on a set S of up to n ***keys***; dictionaries often also allow for a user to store a value associated with each key, which can then be retrieved during queries. Unless stated otherwise, we will assume a machine word of $\Theta(\log n)$ bits, which means that keys/values are also $O(\log n)$ bits. We will also require implicitly that a dictionary should take at most linear space (i.e., $O(n \log n)$ bits) and that a dictionary should be explicit (i.e., it can be initialized in time $O(n)$). In fact, the dictionaries in this chapter have the stronger property that they can be initialized in constant time.

Randomized dictionaries are often also referred to as ***hash tables***.[1] A hash table is said to have ***failure probability*** ε if each operation takes constant time with probability at least $1 - \varepsilon$, and is said to ***succeed with high probability*** if $\varepsilon \leq 1/\operatorname{poly} n$.

A central open question is whether there exists a deterministic constant-time dictionary. A remarkable success in this direction is Pătraşcu and Thorup's dynamic fusion node [306], which builds on older work by Fredman and Willard [186] in order to construct a deterministic constant-time dictionary for very small sets of keys—that is, sets S of at most $\operatorname{polylog} n$ keys that are $\Theta(\log n)$ bits each. For sets of $\Theta(n)$ keys, it is widely believed that (even non-explicit) deterministic constant-time dictionaries are impossible [341], but we are still very far from having lower bounds to establish this (see [153, 207, 295, 318, 337] for other related work on this question).

In this chapter, we consider a natural relaxation of this question: What is the smallest failure probability that a hash table can offer [75, 198, 199]? We present the first hash table to achieve a significantly sub-polynomial failure probability. And we show that such a hash table can even be made ***succinct***, meaning that it uses space within a $(1 + o(1))$ factor of the information-theoretic optimum.

Past work on super-high-probability guarantees. The study of probabilistic guarantees for hash tables has, up until now, been intimately tied to the study of hash-function families [123, 156, 162, 218, 250, 260, 277, 291, 293, 305, 326, 327]. If one has access to fully-random hash functions, then it is known [75, 198, 199] how to achieve substantially sub-polynomial failure probabilities. However, as observed by Goodrich, Hirschberg, Mitzenmacher, and Thaler [199], the known techniques for simulating constant-time hash functions with high independence [162, 293, 327] are *themselves* randomized constructions that introduce an additional $1/\operatorname{poly}(n)$ probability of failure. Efforts at reducing these failure probabilities [199] have only been able to do so at the cost of $\omega(1)$ evaluation times.

Of the known families of constant-time hash functions, the only one that has been

[1]Hash tables are sometimes also informally defined as any solution to the dictionary problem that makes use of hash functions. We intentionally take a more open-ended perspective as to the definition of a hash table, so that we include data structures that accomplish the same goal as traditionally accomplished by hash tables, but using different means.

successfully used to obtain a hash table with sub-polynomial failure probabilities is tabulation hashing [305]. Although the standard analyses of tabulation hashing include a $1/\operatorname{poly}(n)$ failure probability, it has been noted by [305] that, in some parameter regimes, the true failure probability is actually sub-polynomial. Indeed, one can extend the techniques of [305] to show that tabulation hash functions are load-balancing with probability $1 - 1/2^{\operatorname{polylog} n}$, thereby allowing one to construct a hash table that has a failure probability of $1/2^{\operatorname{polylog} n}$. To the best of our knowledge, this remains the smallest failure probability to be achieved by any hash table.

This chapter: hash tables with nearly optimal failure probabilities. We introduce a simple data structure, which we call the ***amplified rotated trie***, that offers a failure probability of $1/n^{n^{1-\varepsilon}}$ for an arbitrarily small positive constant ε of our choice. Barring a deterministic constant-time dictionary, this is the close to the strongest guarantee that one could hope for: if there were to exist a hash table with failure probability $1/n^{\varepsilon n}$, for some positive constant $\varepsilon > 0$, then that would imply the existence of a (non-explicit) deterministic constant-time dictionary. Our result improves significantly over the previous state-of-the-art of $1/2^{\operatorname{polylog} n}$.

Our second result is that, with a few small modifications, the same data structure can be used to obtain a very different guarantee. The resulting hash table, which we call the ***budget rotated trie***, uses $\widetilde{O}(\log n)$ random bits to support constant-time operations with high probability in n. This guarantee, which has also been achieved using more classical hashing-based techniques in previous work by Dietzfelbinger et al. [155], serves as a natural dual to the one above — rather than trying to minimize failure probability, while using up to $O(n)$ random bits, one tries to minimize random bits while maintaining a standard $1/\operatorname{poly}(n)$ failure probability.

An interesting feature of budget rotated tries is that they are able to make use of so-called "gradually-increasing-independence hash functions" [123, 260]. These hash functions, introduced originally by Celis, Reingold, Segev, and Wieder [123] (and subsequently made more efficient by Meka, Reingold, Rothblum, and Rothblum [260]) can be used to distribute n balls roughly evenly across n bins using only $O(n \log\log n)$ random bits, but come with the seemingly significant drawback that they require $\Theta((\log\log n)^2)$ time to evaluate. As a consequence, past work on applying these hash functions to classical hash tables [315] has incurred $\omega(1)$ time per operation. Our approach suggests that such gradually-increasing-independence may be more broadly applicable to than was previously thought, and can be used in the design of constant-time data structures.

Achieving succinctness. Finally, we turn our attention to space efficiency. There has also been a great deal of work on how to construct a ***succinct hash table*** (see, e.g., [75, 85, 245, 313]), that is, a hash table that stores n key/values pairs from a universe U in space

$$(1 + o(1))\mathcal{B}(|U|, n)$$

bits, where $\mathcal{B}(|U|, n) = \log \binom{|U|}{n}$ is the information-theoretic lower bound on the size of any hash table.

In Section 13.6, we show that the data structures in this chapter can also be made succinct, in the parameter regime where keys/values are $(1 + \Theta(1)) \log n$ bits. More generally, we give a black-box transformation that can be applied to any dictionary in order to obtain a succinct dictionary whose probabilistic guarantees are nearly the same as the original's. The new dictionary uses $\mathcal{B}(|U|, n) + O(n(\log n)/\log\log n)$ bits.

Interestingly, the transformation itself makes use of our (non-succinct) budget rotated trie as a critical algorithmic component. The transformation also makes use of a reduction due to Raman and Rao [313], and can be seen as a constant-time and randomness-efficient version of the succinct dictionary given in [313] (which guaranteed only constant expected-time operations).

Applying our transformation, we obtain two data structures: we get a succinct hash table that uses $O(\log n(\log\log n)^3) = \tilde{O}(\log n)$ random bits, while supporting constant-time operations with high probability; and a succinct hash table with a failure probability of $1/n^{n^{1-\varepsilon}}$, where ε is an arbitrarily small positive constant of our choice.

Circumventing the hash-function bottleneck. At the core of our results is a simple but powerful observation: that it is possible to construct a hash table *that does not use hash functions*, and that is consequently free of the limitations that hamper known hash-function constructions. In particular, we begin our exposition by constructing a simple randomized dictionary that we call a ***rotated radix trie***. Like standard hash tables, the rotated radix trie uses linear space and is constant-time (with high probability). But unlike standard hash tables, which rely on randomness supplied by hash functions, the rotated radix trie uses randomness directly embedded into the data structure. The rotated radix trie then serves as the basis for both the amplified rotated trie and the budget rotated trie.

Outline. The rest of the chapter proceeds as follows. Section 13.2 presents basic preliminaries and conventions—these are specialized for the setting of non-succinct dictionaries, and slightly different conventions are established later on in Section 13.6 for discussing succinct dictionaries. Section 13.3 presents and analyzes the rotated radix trie. Building on this, Section 13.4 gives a hash table that achieves failure probability $1/n^{n^{1-\varepsilon}}$ and Section 13.5 gives a high-probability hash table using $O(\log n \log\log n)$ random bits; in Appendix 13.A, we show that the latter guarantee can also be extended to the case where machine words are $\omega(\log n)$ bits. Finally, in Section 13.6, we show how to transform any linear-space hash table into a succinct hash table, while nearly preserving the randomization guarantees of the data structure.

13.2 Preliminaries and Conventions (for Non-Succinct Dictionaries)

We now present several preliminary definitions and conventions for discussing (non-succinct) dictionaries. These conventions are used throughout the chapter, except in Section 13.6 where we consider succinct dictionaries. We end up using slightly different conventions when discussing non-succinct versus succinct dictionaries because, in the non-succinct setting, there are a number of standard simplifications that one can make without loss of generality (but which do not hold in the succinct setting).

Keys, values, and dictionaries. Let $U = [\text{poly}(n)]$ be the set of all possible $\Theta(\log n)$-bit keys, and let $V = [\text{poly}(n)]$ be the set of all possible $\Theta(\log n)$-bit values. A ***dictionary*** is a data structure that stores a set of keys from U, and that associates each key x with a value $y \in V$. Dictionaries support three operations: $\text{Insert}(x, y)$ adds key x to the set, if it is not already there, and sets the corresponding value to y; $\text{Delete}(x)$ removes x; and $\text{Query}(x)$ reports whether key x is present, returning the corresponding value if so.

When discussing non-succinct dictionaries, we focuses (without loss of generality) on fixed-capacity dictionaries, that is, dictionaries that are permitted to have up to n keys at a time. Such dictionaries can be used to implement dynamically-resized dictionaries by simply rebuilding the dictionary (in a deamortized fashion) whenever its size changes by a constant factor. Unless stated otherwise, we shall require implicitly that dictionaries must use at most linear space (i.e., $O(n \log n)$ bits) and have $O(1)$ initialization time.

Standard techniques for simplifying dictionaries. There are several standard reductions that can be used to simplify the problem of maintaining a linear-space dictionary.

We can assume without loss of generality that the lifespan of a dictionary is only $O(n)$ operations. Indeed, longer sequences of operations can be broken into phases of size $O(n)$, and the dictionary can be rebuilt from scratch during each phase (i.e., all of the elements are gradually moved from one instance of the dictionary to another new instance of the dictionary). The rebuild cost can be spread across the phase so that the asymptotic running times of operations are preserved.[2]

Since the lifespan of each phase is only $O(n)$ operations, we can implement deletions with the following trivial approach: simply mark elements as deleted, and defer the actual removal of those elements until the next rebuild. As a consequence, when designing the dictionary that will be used to implement each phase, we can assume without loss of generality that the only operations performed are insertions/queries.

We will therefore assume throughout the chapter that, whenever we are discussing

[2]For our purposes, rebuilds *do not* sample new random bits. Once a dictionary's random bits are chosen, they are fixed forever.

a non-succinct dictionary, the sequence of operations being performed has length $O(n)$ and consists exclusively of inserts/queries.

Randomization. Randomized dictionaries are given access to a stream of random bits—the dictionary can access the next $\Theta(\log n)$ bits of the stream in time $O(1)$. When analyzing a randomized dictionary, the goal is to bound the ***failure probability*** for any given operation. We emphasize that, in this context, failure does not refer to lack of correctness, but instead to lack of timeliness. A dictionary ***fails*** whenever an operation takes super-constant time. (Later, in Section 13.6, when we consider succinct dictionaries, we will also allow for failures with respect to space consumption.)

All of our dictionaries share the property that, once a failure occurs, all of the rest of the operations (in the current phase of $O(n)$ inserts/queries) also fail. We will not bother to explicitly specify the dictionary's behavior when a failure occurs, since at that point it is okay for each of the remaining operations in the phase to take linear time.

We remark that randomized data structures are analyzed against *oblivious* adversaries, meaning that the sequence of insertions/deletions/queries being performed is determined independently of the random bits that the dictionary uses. We also remark that the failure probability of a dictionary is determined on a per-operation basis. For example, if a dictionary has failure-probability p and is used for $1/p$ operations, then it is reasonable that some failures should occur.[3]

13.3 A Warmup Data Structure: The Rotated Trie.

In this section, we present a simple randomized constant-time dictionary, called the ***rotated radix trie***, that serves as the basis for the data structures in later sections.

The starting place: an n-ary radix trie. The starting place for our data structure will be the classic n-ary radix trie. Each internal node of the trie can be viewed as an array of size n, where the j-th entry of the array stores either a pointer to child j, if such a child exists, or a null character otherwise. The leaves of the trie correspond to the keys in the data structure (and are where we store values). In general, there is a leaf with root-to-leaf path $j_1, j_2, j_3, \ldots, j_d$ if and only if the key $j_1 \circ j_2 \circ j_3 \circ \cdots \circ j_d \in [n^d] = [U]$ is present.

What makes the n-ary radix trie an interesting starting place is that the trie deterministically supports constant-time operations. What it does not support is space efficiency: there may be as many as $\Theta(n)$ internal nodes, each of which is an array of size n, and which collectively require space $\Omega(n^2)$ to implement.

[3]Moreover, failures may be correlated between steps (and between phases). For example, if we are using r random bits, and an adversary guesses them, then they can force failures all the time with probability $1/2^r$.

Using randomness to save space: the rotated radix trie. We now add randomness to our data structure in a very simple way. Label the internal nodes of the trie by $1, 2, \ldots, m$ for some $m \in O(n)$, and refer to the array used to implement each internal node $i \in [m]$ as A_i. When the data structure is initialized, we assign to each internal node i a ***random rotation*** r_i selected uniformly at random from $\{0, 1, \ldots, n-1\}$. The rotation r_i is stored as part of the node i.

The purpose of r_i is to apply a random cyclic rotation to the array A_i. That is, if a pointer would have been stored in position j of A_i, it is now stored in position $((j + r_i) \bmod n)$ of A_i instead.

Finally, having rotated each of the arrays A_i by r_i, we now overlay the arrays $A_1, A_2, \ldots, A_m$ on top of one another, and we store the contents of all of them in a single array A of size n. Of course, the j-th position of A may be responsible for storing elements from multiple A_is. As long as the number of elements stored in each entry is relatively small, then this is fine: we simply implement each entry of A as a dynamic fusion node [306], which is a deterministic constant-time linear-space dictionary capable of storing up to $\ell = \operatorname{polylog} n$ key/value pairs at a time.

If, prior to collapsing the arrays into a single array A, the the j-th position of rotated array A_i stored a pointer to array $A_{i'}$, then afterwards the dynamic fusion node $A[j]$ stores the key-value pair $(i, (i', r_{i'}))$. In this setting, we refer to the pair $(i', r_{i'})$ as a ***pointer*** to $A_{i'}$, since it dictates which array $A_{i'}$ we are pointing at and where to find the entries of $A_{i'}$. Similarly, if prior to collapsing the arrays, the j-th position of the rotated array A_i stored a pointer p directly to a value (rather than to another array $A_{i'}$), then the dynamic fusion node $A[j]$ stores the key-value pair (i, p).

Analyzing the rotated radix trie. To analyze the rotated radix trie, we must show that, with high probability in n, each entry of A is responsible for storing entries from at most $\ell = \operatorname{polylog} n$ different A_is.

Let us first establish some conventions that will be useful throughout the rest of the chapter. When discussing a radix trie, we will refer to the arrays $A_1, A_2, \ldots, A_m$ as the ***nodes*** (or sometimes as the ***internal nodes***), and we will refer to the non-null entries of each A_i (i.e., the entries containing pointers) as ***balls***.

In total, there are $O(n)$ balls in the trie. Each ball b is specified by a pair $(s, c) \in [m] \times [n]$, where $s \in [m]$ is the ***source node*** for the ball (i.e., the node containing the ball), and $c \in [n]$ is the ***child index*** of the ball (i.e., the index in A_i where b is logically stored). The effect of randomly rotating the arrays A_i and then overlaying them to obtain a single array A is that each ball $b = (s, c)$ gets mapped to position $\phi(s, c) := c + r_s$ in A. We refer to the entries of A as ***bins***, so each ball b gets mapped to bin $\phi(b)$. The dynamic fusion node for a each bin $j \in [n]$ stores the set of key-value pairs (b, p) where b ranges over the balls satisfying $\phi(b) = j$, and p is the pointer corresponding to the ball b.

For $i \in [m]$ and $j \in [n]$, let $X_{i,j}$ be the 0-1 random variable indicating whether node i places a ball into bin j. The $X_{i,j}$s are not independent across the bins j, but they are independent across the nodes i, since each $X_{i,j}$ is a function of the random bits r_i. Therefore, the number Y_j of balls in bin j, which is given by $Y_j = \sum_{i=1}^{m} X_{i,j}$, is a sum of independent indicator random variables.

Each of the $O(n)$ balls has probability $1/n$ of being in bin j, so $\mathbb{E}[Y_j] = O(1)$. Thus, by a Chernoff bound, we have that $Y_j \le \operatorname{polylog} n$ with high probability in n. The Chernoff bound actually tells us that $Y_j \le \operatorname{polylog} n$ with probability $1/n^{\operatorname{polylog} n}$, so we have even achieved a slightly sub-polynomial probability of failure.

Putting the pieces together. If we implement deletions as in Section 13.2, then we obtain the following result:

Proposition 142. *The rotated radix trie is a randomized linear-space dictionary that can store up to n $\Theta(\log n)$-bit keys/values at a time, and that supports each operation in constant time with probability* $1 - 1/n^{\operatorname{polylog} n}$.

It's worth taking a moment to remark on how to initialize our data structure. The random rotations r_i can be initialized lazily, so that r_i is generated the first time that the node i is used. Additionally, we do not have to actually pay the cost of initializing any arrays, since we can use standard techniques to simulate zero-initialized arrays in constant time (see [43] or Problem 9 of Section 1.6 of [92]). Thus our rotated radix trie can be initialized in constant time.

Taking stock of our situation. The rotated radix trie does not, on its own, make any significant progress on either of the problems that we care about: (1) achieving super-high probability guarantees, and (2) using a near-logarithmic number of random bits. We have achieved a *slightly* sub-polynomial failure probability, but we are nowhere near our goal of $1/n^{n^{1-\varepsilon}}$.

What makes the rotated radix trie useful, however, is that the role of randomness in the data structure is remarkably simple. The *only* sources of randomness are the rotational offsets $r_1, r_2, \ldots, r_m$. In this sense, the rotated radix trie deviates from the standard mold for how to design a constant-time dictionary. The randomness in the data structure isn't used to hash elements, but is instead used to apply random rotations to sparse arrays.

Since the role of randomness will be important in later sections, we conclude the current section by discussing an important subtlety in how the randomness is re-purposed over time. Consider how the data structure evolves over a large period of time containing many insertions/deletions. As the shape of the trie changes, each array A_i will be repurposed to represent different parts of the trie. This means that the way in which the random rotation r_i interacts with the key space also changes over time, with the same r_i applying to a different node in the trie (and thus a different part of the key space) at different times. The re-purposing of r_is has an interesting consequence: even if two points in time t_1 and t_2 store the exact same set S of key/value pairs as one-another, the shape of the rotated trie may differ considerably between the two times.

13.4 The Amplified Rotated Trie

In this section, we modify the rotated radix trie to reduce its probability of failure (i.e., the probability that a given operation takes super-constant time) to $1/n^{n^{1-\varepsilon}}$, for a positive constant ε of our choice. We will refer to this new data structure as the ***amplified rotated radix trie***.

Storing overflow balls in a (non-rotated) trie. Whenever a ball b is inserted into a bin j that already contains $\ell = \operatorname{polylog} n$ other balls, the ball b is regarded as an ***overflow ball***. Since each bin is a dynamic fusion node with capacity ℓ, we cannot store the overflow balls in the bins.

We instead store the overflow balls in a secondary data structure Q that is implemented as a n^{δ}-ary trie, for some positive constant $\delta > 0$.

The secondary data structure Q supports inserts/queries on overflow balls in constant time. On the other hand, Q is not space efficient. If there are q overflow balls, then Q may use as much as qn^{δ} space. To establish that our dictionary uses linear space, we must show that

$$\Pr[q \geq n^{1-\delta}] \leq O\left(1/n^{n^{1-\varepsilon}}\right). \tag{13.1}$$

The problem: dependencies between balls with shared source nodes. Our current data structure does not yet satisfy (13.1), however. This is because, whenever multiple balls share the same source node, their assignments become closely linked. Suppose, for example, that the rotated trie R has only 2ℓ internal nodes, and that each internal node $i \in \{1, 2, \ldots, 2\ell\}$ contains $\Theta(n/\ell)$ balls $(i, 1), (i, 2), \ldots, (i, \Theta(n/\ell))$. With probability $1/n^{2\ell} = 1/2^{\operatorname{polylog} n}$, each internal node $i \in \{1, 2, \ldots, 2\ell\}$ has random rotation $r_i = 0$. This results in bins $1, 2, \ldots, \Theta(n/\ell)$ each containing 2ℓ balls—in other words, half of the balls in the system are overflow balls. This means that

$$\Pr[q \geq \Omega(n)] \geq 1/2^{\operatorname{polylog} n}.$$

That is, our failure probability using Q the store overflow balls is *no better* than the failure probability that we achieved in Section 13.3 without Q.

Reducing the dependencies. What makes the above pathological example possible is that it is possible to have only a small number of internal nodes in our rotated trie. This makes it so that there are only a small number of random bits that affect the rotated trie's structure, preventing us from achieving any super-high probability guarantees.

To fix this problem, we reduce the fanout of our rotated radix trie from n to n^{δ}. Now each internal node can contain at most n^{δ} balls, so there are guaranteed to be at least $n^{1-\delta}$ internal nodes. This ensures that there are always at least $n^{1-\delta} \log n$

random bits affecting the trie's structure.

We remark that, since the fanout of the rotated trie is now n^δ, each ball is determined by a pair (s, c) where $s \in [m]$ is a source node and $c \in [n^\delta]$ is a child index. Nonetheless, the mapping ϕ from balls to bins works exactly as before: we map ball (s, c) to bin $\phi(s, c) = ((r_s + c) \bmod n)$ where $r_s \in [n]$ is selected at random.

Bounding the number of overflow balls. Of course, there are still dependencies between the number q_j of overflow balls in different bins $j \in [n]$. To handle these dependencies, we make use of a tool from probabilistic combinatorics.

Call a function $f : [0, 1)^m \to \mathbb{R}$ ***L-Lipschitz*** if for every pair of inputs of the form $\vec{x} = (x_1, \ldots, x_i, \ldots, x_m)$ and $\vec{x'} = (x_1, \ldots x'_i, \ldots, x_m)$, we have $|f(\vec{x}) - f(\vec{x'})| \leq L$. McDiarmid's inequality [256] tells us that if f is L-Lipschitz and $X_1, X_2, \ldots, X_m \in [0, 1)$ are independent random variables, then for any $t \geq 0$,

$$\Pr[|f(X_1, \ldots, X_m) - \mathbb{E}[f(X_1, \ldots, X_m)]| \geq t] \leq 2e^{-2t^2/(mL^2)}.$$

To apply McDiarmid's inequality to our situation, define $f(r_1, \ldots, r_m) := q$ to be the number of overflow balls. Observe that f is n^δ-Lipschitz, since each r_i can determine the outcome of at most n^δ different balls. Since $\mathbb{E}[q] = \frac{1}{\operatorname{poly} n}$, it follows by McDiarmid's inequality that

$$\begin{aligned}\Pr[f(r_1, \ldots, r_m) \geq n^{1-\delta}] &\leq e^{-\Omega(n^{2-2\delta}/(mn^{2\delta}))} \\ &= e^{-\Omega(n^{2-2\delta}/n^{1+2\delta})} \\ &= e^{-\Omega(n^{1-4\delta})}.\end{aligned}$$

For any $0 < \varepsilon \leq 1$, we can set $\delta = \varepsilon/5$ so that

$$\begin{aligned}\Pr[q \geq n^{1-\delta}] &\leq e^{-\Omega(n^{1-4\delta})} \\ &\leq O\left(n^{-n^{1-\varepsilon}}\right).\end{aligned}$$

This establishes (13.1). If we implement deletions as in Section 13.2, then we arrive at the following theorem.

Theorem 143. *The $n^{\varepsilon/5}$-ary amplified rotated radix trie is a randomized linear-space dictionary that can store up to n $\Theta(\log n)$-bit keys/values at a time, and that supports each operation in constant time with probability $1 - O\left(1/n^{n^{1-\varepsilon}}\right)$.*

We remark that there is a strong sense in which the amplified rotated radix trie is nearly optimal. In particular, for any constant $\varepsilon > 0$, if there were to exist a randomized linear-space dictionary with failure probability of $1/n^{\varepsilon n}$, that would imply the existence of a deterministic (though non-explicit) linear-space constant-time dictionary.

Lemma 144. *Let $\varepsilon > 0$ be any positive constant and assume a machine word of size $w = \Theta(\log n)$ bits. Suppose there exists randomized linear-space dictionary that stores up to n $\Theta(\log n)$-bit keys/values at a time and has failure probability $1/n^{\varepsilon n}$. Then there also exists a deterministic (not-necessarily explicit) dictionary with the same guarantees.*

Proof. To distinguish the randomized dictionary from the deterministic dictionary that we are constructing, we will refer to the former as a hash table and the latter as a dictionary.

As noted in Section 13.2, by rebuilding our dictionary once every $O(n)$ operations, we can assume without loss of generality that the lifespan of the dictionary is at most $O(n)$ operations. We will implement the dictionary using a hash table with capacity $n' = cn$ for some large positive constant c to be determined later. This means that the hash table has failure probability

$$1/n^{\varepsilon n'} = 1/n^{\varepsilon cn}.$$

Each operation takes place on a $\Theta(\log n)$-bit key/value pair, so there are at most $n^{O(1)}$ options for what a given operation could be. The total number of $O(n)$-long operation sequences is therefore at most $n^{O(n)}$. Since our hash table has failure probability $1/n^{\varepsilon cn}$, its total failure probability on any given sequence of $O(n)$ operations is at most $O(n)/n^{\varepsilon cn} \le 1/n^{\varepsilon cn/2}$ The probability that there exists *any* sequence of operations on which our hash table fails to be constant-time is therefore at most

$$\frac{n^{O(1)}}{n^{\varepsilon cn/2}},$$

which if c is taken to be a sufficiently large constant, is at most $1/2$. Thus there exists some choice of random bits for which our hash table is constant-time on *every* sequence of operations. By hard-coding in this choice of random bits, we arrive at a deterministic constant-time dictionary.

Note that, since the hash table spends total time $O(n)$ on the $O(n)$ operations, the number of random bits that it can use is at most $O(nw) = O(n \log n)$ bits—thus the deterministic dictionary can hard-code the random bits in linear space. ∎

Although one typically assumes a machine-word size of $\Theta(\log n)$ bits, it is also an interesting question what the strongest achievable probabilistic guarantees are in the setting where machine words (as well as keys/values) are of some size $w = \omega(\log n)$ bits. On one hand, the larger key size makes it so that Lemma 144 no longer applies, so in principle, one might be able to achieve a failure probability of $1/n^{\omega(n)}$. On the other hand, from an upper-bound perspective, it is not even known how to achieve a *sub-polynomial failure probability* in this setting [75, 198, 199, 305]. Here, the main obstacle appears to be unavoidably about hash functions: can one construct a family of hash functions from $[2^w]$ to $[\text{poly}(n)]$ such that for any given n-element set $S \subseteq [2^w]$, we have that $\max_{x \in S} |\{y \in S \mid h(x) = h(y)\}| \le \text{polylog}\, n$ with probability $1/n^{\omega(1)}$? If such a family were to exist, then it could be directly combined with Theorem 143 to construct a dictionary that achieves sub-polynomial failure probability for any key-size w. We conjecture that no such family of hash functions exists, and moreover, that a sub-polynomial failure probability is not possible for word sizes $w = \omega(\log n)$ bits.

13.5 The Budget Rotated Trie

In this section, we present a dictionary that uses only $O(\log n \log \log n)$ random bits, while guaranteeing that each operation takes constant time with probability $1 - 1/\operatorname{poly}(n)$ (i.e., with high probability in n). We will refer to the data structure as the ***budget rotated trie***. In Appendix 13.A, we further extend the budget rotated trie to support keys that are $\omega(\log n)$ bits, while still using only $O(\log n \log \log n)$ bits of randomness.

We remark that the guarantee achieved by the budget rotated trie is not novel—in fact, a previous approach by Dietzfelbinger, Gil, Matias, and Pippenger [155] can be used to achieve $O(\log n)$ random bits for the setting of $\Theta(\log n)$-bit keys that we are considering. Nonetheless, we believe that the construction for the budget rotated tries is interesting in its own right, both because of its relationship to the amplified rotated trie, and also because of the surprising way in which it is able to make use of gradually-increasing-independence hash functions. Additionally, the specific structure of the budget rotated trie will prove useful in our quest for succinctness in Section 13.6.

Our starting place is again the rotated trie, and as in Section 13.4, we will take the fanout of the trie to be n^{δ} for some constant δ; in fact, it will suffice to simply use $\delta = 1/4$.

Reducing the number of random bits to $O(n/\operatorname{polylog} n)$. To transform the n^{δ}-ary rotated trie into a budget rotated trie, our first modification will be to reduce the number of random bits from $O(n \log n)$ to $O(n/\operatorname{polylog} n)$. Of course, this may not seem like much progress, but we shall see later that the distinction is important.

Recall that, in a rotated trie, each ball b (i.e., each non-null entry in an internal node) contains a pointer to either a leaf (i.e., an actual key/value pair) or another internal node (i.e., a child). We now add a third option: if the ball should be pointing at another internal node x, but if the subtree rooted at x contains fewer than $\ell = \operatorname{polylog} n$ total keys, then we store that subtree as a dynamic fusion node z. If the size of the subtree rooted at x subsequently surpasses ℓ, then we create an actual internal node for x—in this case, any elements stored in the fusion node z remain in z, and the ball b now stores two pointers, one to x and one to z. In other words, there are now three possible states for a ball: it can contain a pointer to a leaf; it can contain a pointer to a dynamic fusion node; or it can contain two pointers, one to a dynamic fusion node and one to another internal node of the trie.

The point of this modification is that we only create an internal node x if the subtree rooted at x contains at least $\ell = \operatorname{polylog} n$ elements. Importantly, this means that the total number of internal nodes m is at most $O(n/\ell) = n/\operatorname{polylog} n$. The number of random bits needed for the rotations $r_1, r_2, \ldots, r_m$ is therefore also $n/\operatorname{polylog} n$.

Changing the balls-to-bins mapping. Our next modification is to change how we map the balls to bins. Recall that each ball b is specified by a pair (s, c), where $s \in [m]$ is the source node of the ball and $c \in [n^\delta]$ is the child index. In the standard rotated trie, we map balls to bins using the function

$$\phi(s, c) = (c + r_s) \bmod n.$$

We will now instead map balls to bins using the function

$$\psi(s, c) = (c + a_s (\bmod n^\delta)) \cdot n^{1-\delta} + b_s,$$

where a_s is selected at random from $[n^\delta]$ and b_s is selected at random from $[n^{1-\delta}]$.

When can think about ψ as follows. We break the bins into groups $G_1, \ldots, G_{n^\delta}$ of size $n^{1-\delta}$, and we use the random value $a_s \in [n^\delta]$ to assign the ball to a random group. Once the ball is assigned to a group G_i, it is then assigned to the b_s-th bin in that group. Importantly, the assignments are designed so that each source node s assigns *at most one* of its balls to any given group G_i. There will never be two balls b_1, b_2 in group G_i that both obtain their assignments b_s from the same source node.

Since the number m of internal nodes may be as large as $n/\operatorname{polylog} n$, we cannot afford to generate $a_1, a_2, \ldots, a_m \in [n^\delta]$ and $b_1, b_2, \ldots, b_m \in [n^{1-\delta}]$ truly at random. Fortunately, as we shall now see, the roles of the a_is and b_is have been carefully designed so that both sequences can be generated using a small number of "seed" random bits.

Generating the $\boldsymbol{a_i}$s with $\boldsymbol{O(1)}$-independent hash functions. Let k be a sufficiently large positive constant, and select a random hash function $g : [n] \to [n^\delta]$ from a family of k-independent hash functions. Since $k = O(1)$, the function g can be specified using $O(\log n)$ random bits, and can be evaluated in time $O(1)$. We compute the a_is by

$$a_i := g(i).$$

To analyze the number of balls in each group G_i, we use a well-known tail bound for k-independent random variables (see, e.g., [69] or [166]).

Lemma 145 (Lemma 2.2 of [69]). *Let $k \geq 4$ be an even integer. Suppose $X_1, \ldots, X_m$ are k-wise independent 0-1 random variables. Let $X = \sum_i X_i$. Then, for any $t \geq 0$,*

$$\Pr[|X - \mathbb{E}[X]| \geq t] \leq 2\left(\frac{nk}{t^2}\right)^{k/2}.$$

Define X_j to be the event that source-node j sends a ball to group G_i. The X_js are k-independent, so we have by Lemma 145 that

$$\Pr[|G_i| - \mathbb{E}[|G_i|] \geq n^{0.75}] \leq 2\left(\frac{kn}{n^{1.5}}\right)^{k/2} \leq n^{-\Omega(k)} = 1/\operatorname{poly}(n).$$

Since $\delta = 0.25$, it follows that

$$\Pr[|G_i| - \mathbb{E}[|G_i|] \geq n^{1-\delta}] \leq 1/\operatorname{poly}(n).$$

Since each of the $O(n)$ balls is equally likely to be in any group, we have that $\mathbb{E}[|G_i|] = O(n^{1-\delta})$. Thus

$$\Pr[|G_i| \leq O(n^{1-\delta})] \geq 1 - 1/\operatorname{poly}(n).$$

That is, each group G_i contains at most $O(n^{1-\delta})$ balls with high probability in n.

Generating the $\boldsymbol{b_i}$s with increasing-independence hash functions. To generate the b_is without using a large number of random bits, we make use of a more sophisticated family of hash functions. Call a family $\mathcal{H}(t)$ of hash functions $h : [\operatorname{poly}(t)] \to [t]$ ***load-balancing*** if it can be used to map t balls to t bins with maximum load $\operatorname{polylog} t$; that is, for any fixed set $S \subseteq \operatorname{poly}(t)$ of size t, and for any fixed $i \in [t]$, if we select a random $h \in \mathcal{H}$, then

$$|\{s \in S \mid h(s) = i\}| \leq \operatorname{polylog} t$$

with probability $1 - 1/\operatorname{poly}(t)$.

Celis, Reingold, Segev, and Wieder [123] showed how to construct a load-balancing family $\mathcal{H}(t)$ of hash functions such that each $h \in \mathcal{H}$ can be described with $O(\log t \log \log t)$ random bits and can be evaluated in time $O(\log t \log \log t)$. The family $\mathcal{H}$ is referred to as having "gradually-increasing-indepenence" because each $h \in \mathcal{H}$ is actually the composition of $\Theta(\log \log t)$ hash functions $h_1, \ldots, h_{\Theta(\log \log t)}$ with different levels of independence: each h_i determines $\Theta((3/4)^i \log t)$ bits of h, and each h_i is $(1/\operatorname{poly} t)$-close to being $\Theta((4/3)^i)$-independent.

The family $\mathcal{H}$ comes with a tradeoff. It is able to achieve a maximum load of $\operatorname{polylog} t$ (in fact, it even achieves maximum load $O(\log t / \log \log t)$) using on $O(\log t \log \log t)$ bits, but it requires super-constant time to evaluate. Subsequent work [260] has improved the evaluation time from $O(\log t \log \log t)$ to $O((\log \log t)^2)$. It seems unlikely that the evaluation time can be improved to $O(1)$, however, since $\Omega(\log \log t)$ time is needed just to read the random bits used to evaluate the hash function.

The super-constant evaluation time makes it so that hash functions with gradually-increasing independence are not suitable for direct use in constant-time hash tables [315]. We get around this problem by using h not as a hash function but as a pseudo-random number generator. Specifically, we select a random $h : [m] \to [n^{1-\delta}]$ from

$\mathcal{H}(n^{1-\delta})$, and we use h to initialize the b_is as

$$b_i := h(i).$$

Since h takes time $O((\log\log n)^2)$ to evaluate, each b_i now takes time $O((\log\log n)^2)$ to initialize. Recall, however, that we only create a new internal node x in our rotated trie once there are more than $\ell = \operatorname{polylog} n$ records that want to reside in that node's subtree; the first $\ell = \operatorname{polylog} n$ insertions that wish to use x are instead placed into a dynamic fusion node that acts as a proxy for x. As a result, we can afford to spend up to ℓ time *initializing* the node x, and we can spread that time across the ℓ insertions that trigger x's initialization. Since $\ell = \omega((\log\log n)^2)$, we can initialize $b_i = h(i)$ without any problem.

Analyzing the maximum load. Recall that, with probability $1 - 1/\operatorname{poly}(n)$, each group G_i contains at most $O(n^{1-\delta})$ balls. Furthermore, each of the balls have different source nodes than one another. If a ball has source-node s, then it is placed in the b_s-th bin of G_i.

Let $S_i \subseteq [m]$ be the set of source nodes that assign balls to G_i. Then for each $r \in [n^{1-\delta}]$, the number $g_{i,r}$ of balls in the r-th bin of G_i is given by

$$g_{i,r} = |\{s \in S_i \mid h(s) = r\}|.$$

Since $h : \operatorname{poly}(n) \to [n^{1-\delta}]$ is from a load-balancing family of hash functions, we are guaranteed to have

$$g_{i,r} \le \operatorname{polylog} n^{1-\delta} \le \operatorname{polylog} n$$

with high probability in n.

Putting the pieces together. The fact that each bin contains at most $\operatorname{polylog} n$ balls (with high probability) means that, as in the standard rotated trie, each bin can be implemented with a dynamic fusion node. Operations on our dictionary therefore take time $O(1)$ with high probability in n. If we implement deletions as in Section 13.2, then we arrive at the following theorem.

Theorem 146. *The budget rotated trie is a randomized linear-space dictionary that can store up to n $\Theta(\log n)$-bit keys/values at a time, that uses $O(\log n \log\log n)$ random bits, and that supports each operation in constant time with probability $1 - 1/\operatorname{poly}(n)$.*

We conclude the section by observing that there is a strong sense in which the guarantee achieved by the budget rotated trie is optimal. In particular, if there were to exist a hash table failure probability $1/n^c$ but that used fewer than $c \log n$ random bits, then there would also necessarily exist a deterministic linear-space constant-time dictionary.

Lemma 147. *Suppose there exists a randomized linear-space dictionary that can store up to n $\Theta(\log n)$-bit keys/values at a time, that uses $c \log n$ random bits, but* *that has a failure probability smaller than $1/n^c$. Then there exists a deterministic dictionary with the same guarantees.*

Proof. To distinguish the randomized dictionary from the deterministic dictionary that we are constructing, we will refer to the former as a hash table. Let R denote the $c \log n$ random bits used by the hash table. Define $\mathcal{D}$ to be the deterministic dictionary obtained by setting $R = 0$. Suppose for contradiction that D is not constant time. Then there exists some sequence of operations such that the final operation on D takes super-constant time. This means that, with probability at least $1/n^c$, that same operation would have taken super-constant time in our hash table. But the hash table has failure probability smaller than $1/n^c$, a contradiction. ∎

13.6 Achieving Succinctness

In this section, we turn our attention to space efficiency. Throughout the section, we will use $c_1 \log n$ to denote the size in bits of each key, we will use $c_2 \log n$ to denote the size in bits of each value, and will assume that $c = c_1 + c_2$ is a positive constant larger than 1. Here, unlike in previous sections, we use n to denote the *current size*, at any given moment, and we allow n to change dynamically over time, so long as the key/value length remains $\Theta(\log n)$ bits at all times after initialization—this means that the constants c_1, c_2, c also change dynamically.[4]

The dictionaries that we have described in previous sections are already optimal up to constant factors, using a total of $\Theta(n \log n)$ bits to store n key/value pairs. We shall now strive to achieve optimal space consumption up to low-order terms, that is, to use a total of

$$(1 + o(1)) \log \binom{2^{c \log n}}{n} = cn \log n - n \log n + o(n \log n) \tag{13.2}$$

bits. Such a dictionary is referred to as ***succinct*** [313].

In fact, we will prove a much more general result: that any constant-time dictionary can be *transformed* into a succinct constant-time dictionary, while (nearly) preserving the random-bit usage and failure probability of the original dictionary.

Define an $(r(n), p(n), s(n))$***-dictionary*** to be any constant-time dictionary that uses $O(r(n))$ random bits, achieves failure probability $O(p(n))$ per operation, and uses space $cn \log n - n \log n + O(s(n))$ bits. Since we are interested in dictionaries that automatically resize as the number of keys change, one should think of $r(n)$, $p(n)$, and $s(n)$ as functions rather than fixed values. Note that, in the context of space-efficient time-efficient dictionaries, a failure event could be in terms of either time (an operation takes $\omega(1)$ time) or space (the dictionary fails to fit in $cn \log n -$

[4]Note that keys trivially must have length at least $\log n$ bits, so the key/value size is necessarily $\Omega(\log n)$. If we assume that values are asymptotically no larger than keys, then one can enforce the bound of keys/values having length $O(\log n)$ by treating the dictionary as containing an extra $2^{\varepsilon w}$ dummy elements, where w is the key/value length and ε is a small positive constant.

$n \log n + O(s(n))$ bits)—and a failure probability of $p(n)$ means that, at any given moment, the probability of a failure occurring should be at most $O(p(n))$.

The main result of the section can be stated as follows.

Theorem 148. *Let ε be a small positive constant. Suppose that $r(n)$ and $p(n)$ are nondecreasing functions satisfying $r(n) \leq O(n)$ and $\exp(-n^{1-\varepsilon}) \leq p(n) \leq 1/\operatorname{polylog}(n)$. Given an $(r(n), p(n), n \log n)$-dictionary, one can construct a $(r'(n), p'(n), s'(n))$-dictionary with*

$$r'(n) = r(n) + (\log p(n)^{-1}) \cdot (\log \log n)^3,$$

$$p'(n) = p(n/\log \log n),$$

and

$$s'(n) = \frac{n \log n}{\log \log n}.$$

Applying Theorem 148 to Theorems 143 and 146, we get the following succinct versions of the theorems.

Corollary 149. *Let $0 < \varepsilon < 1$ be a positive constant. There exists a $(r(n), p(n), s(n))$-dictionary that uses $r(n) = O(n)$ random bits, that incurs a failure probability $p(n) = \exp(-n^{1-\varepsilon})$, and that incurs an additive space overhead of $s(n) = O\left(\frac{n \log n}{\log \log n}\right)$ bits compared to the information-theoretical optimum.*

Corollary 150. *There exists a $(r(n), p(n), s(n))$-dictionary that uses $r(n) = O(\log n (\log \log n)^3) = \tilde{O}(\log n)$ random bits, that incurs a failure probability $p(n) = 1/\operatorname{poly}(n)$, and that incurs an additive space overhead of $s(n) = O\left(\frac{n \log n}{\log \log n}\right)$ bits compared to the information-theoretical optimum.*

The rest of the section will be spent proving Theorem 148. We begin in Subsection 13.6.1 by presenting a reduction due to Raman and Rao [313], which transforms the problem of constructing a succinct dictionary into a different problem, which we call the ***many-sets problem***. Then, in Subsection 13.6.2, we show how to solve the many-sets problem using an $(r(n), p(n), O(n \log n))$-dictionary, while approximately preserving the randomness guarantees of the dictionary—an interesting feature of our solution is that it makes extensive use of the budget rotated trie constructed in Section 13.5.

13.6.1 Reduction to the Many-Sets Problem

An important tool in our proof of Theorem 148 will be a reduction due to Raman and Rao [313]. This reduction transforms the problem of constructing a succinct dynamic dictionary into a different problem that we call the many-sets problem.

Let $\delta > 0$ be a small positive constant of our choice. The ***many-sets problem***, with parameter δ, is defined as follows. Let $S_1, S_2, \ldots, S_m$ be (dynamically changing) sets of $O(\log n)$-bit key/value pairs, satisfying $\sum_i |S_i| = n$, $m \leq n/\operatorname{polylog}(n)$, and

$|S_i| \leq n^\delta$ for all $i \in [m]$, where n and m are permitted to evolve dynamically over time. We will use $\gamma_i = O(\log n)$ to denote the size of each key/value pair in S_i (so different S_is may have differently sized key/value pairs).

Any solution to the ***many-sets problem*** must support the following operations in constant time:

- INSERT(i, x, y), which inserts key/value pair (x, y) into S_i;
- DELETE(i, x), which deletes x (and its corresponding value) from S_i;
- and QUERY(i, x), which returns the value associated with x in S_i, or declares that $x \notin S_i$.

A many-sets solution is said to be an $(r(n), p(n), s(n))$***-solution*** if it uses $r(n)$ random bits, has failure probability $p(n)$ per insertion, and uses total space

$$\sum_i |S_i| \cdot (\gamma_i + O(\log |S_i|)) + s(n)$$

bits.

The following reduction is given (implicitly) in Section 3 of [313]:

Theorem 151. *The problem of constructing a $(r(n), p(n), s(n) + n \log n / \log\log n)$-dictionary reduces deterministically to the problem of constructing an $(r(n), p(n), s(n))$-solution to the many-sets problem, where the parameter δ is a positive constant of our choice.*

Understanding the reduction. Although we defer the full proof of Theorem 151 to [313], we take a moment to briefly describe the high-level structure here. The key insight is to make use of tries in a clever way (that differs substantially from how they are used elsewhere in this chapter). Different nodes of the trie have different fanouts: if a node has $r > \operatorname{polylog} n$ keys in it and has depth $O(1)$ in the trie, then it has fanout $r/\operatorname{polylog} n$. If a node has either $r \leq \operatorname{polylog} n$ keys in it, or has sufficiently large constant depth in the trie, then the node is a ***leaf***, and the elements of the node correspond to a set S_i in the many-sets problem.

Unlike for the data structures in this chapter, the internal nodes of this trie are implemented using straightforward arrays: if a node has fanout f, it is implemented using an array of size f. Fortunately, by choosing the fanout f to be $r/\operatorname{polylog} n$, where r is the number of elements stored by the node, one ensures that the arrays used to implement internal nodes have cumulative size $n/\operatorname{polylog} n$.

A key insight is that the trie allows for us to shave bits off of keys: if a node x has fanout f, then we can remove the high-order $\log f$ bits from each of the keys stored in x's children (these bits are now stored implicitly in the trie path). Raman and Rao [313] show that, if a leaf has size $|S_i|$, then the length γ_i of each of the key/value pairs in S_i will satisfy $\gamma_i \leq c \log n - \log n - q \log |S_i| + O(\log\log n)$ bits for a positive constant q of our choice (depending on the maximum depth of the trie). This is why, for the many-sets problem, it is okay to use (amortized) space $\gamma_i + O(\log |S_i|)$ bits per key in S_i.

We remark that, in [313], Theorem 151 was proved with *amortized* constant-time operations. The amortization came from the fact that, whenever a node x's size changes substantially in the trie, it must be rebuilt; however, by spreading the rebuild across $\Theta(|S_i|)$ operations (that each modify the node x), the rebuild can trivially be deamortized to take $O(1)$ time per operation (and without compromising space efficiency, since each element is in only one version of the node at a time).

Before continuing, it is worth taking a moment to understand where the bounds $m \le n/\operatorname{polylog}(n)$ and $|S_i| \le n^\delta$ (which are assumed in the many-sets problem) come from. The bound $m \le n/\operatorname{polylog}(n)$ comes from the fact that each internal node of the trie has fanout $r/\operatorname{polylog}(n)$, where r is its size; since the trie has depth $O(1)$, this means that the total number of leaves (and thus the number of S_is) is at most $O(n/\operatorname{polylog} n)$. The bound $|S_i| \le n^\delta$ comes from the fact that, if a given S_i were to have size $> n^\delta$, then at each node on its root-to-leaf path, each element in the set would have $\delta \log n - O(\log \log n)$ bits shaved off; but assuming that the trie has sufficiently large constant depth, this means that *all* of the bits are shaved off from the elements of S_i, which is a contradiction.

A starting place: Raman and Rao's solution to the many-sets problem. Raman and Rao [313] give a simple solution to the many-sets problem that incurs constant expected time per operation, and which will serve also as the starting point for our construction. At a high level, each S_i is stored in a two-part structure, consisting of a ***skeleton*** A_i and a ***storage array*** B_i.[5]

The storage array B_i is logically implemented as a dynamically-resized array that stores the elements of S_i (both the keys and values) contiguously. The dynamic resizing of the $\{B_i\}$s can be implemented (and deamortized) to take $O(1)$ time per operation while ensuring that, in aggregate, the B_i arrays use space within a factor of $1 + 1/\operatorname{polylog}(n)$ of optimal [313] (see, also, [116], for a more detailed treatment of succinct dynamic arrays).

The skeleton A_i allows for queries to S_i to find the appropriate key/value in B_i. In particular, the skeleton is a dictionary that maps the $\Theta(\log |S_i|)$-bit hash $h(x)$ for each key $x \in S_i$ to the index $j \in B_i$ at which x appears.[6]

The good news is that, within the skeleton A_i, both the keys (i.e., hashes $h(x)$) and the values (i.e., indices in B_i) are only $O(\log |S_i|)$ bits—this means that A_i can be implemented using any standard linear-space dictionary, without worrying about succinctness. The bad news is that there are two ways in which an insertion into A_i could fail: (1) the hash $h(x)$ of the element being inserted satisfies $h(x) = h(x')$ for

[5]The names *skeleton* and *storage array* are conventions that we are establishing here for ease of discussion, and were not used in the original paper [313].

[6]As noted by by [313], there is a subtle issue that one must be careful about for deletions. Whenever a deletion occurs, a gap is created in some position j of B_i. To remove this gap, one must move the final element x in B_i to position j. This means that we must also update the index that is stored for element x in A_i to be j.

some other $x' \in S_i$; or (2) the dictionary used to implement A_i fails.

Assuming that A_i is implemented to have a $1/\operatorname{poly}(|S_i|)$ failure probability (i.e., to be a w.h.p. dictionary), the probability of any given insertion into S_i failing is $1/\operatorname{poly}(|S_i|)$. Of course, each $|S_i|$ can be arbitrarily small, which is why the data structure given in [313] offered constant expected-time operations, rather than a high-probability guarantee.

13.6.2 Proof of Theorem 148

By the reduction in the previous subsection (Theorem 151), it suffices for us to prove the following proposition:

Proposition 152. *Let ε be a small positive constant. Suppose that $r(n)$ and $p(n)$ are nondecreasing functions satisfying $r(n) \le O(n)$ and $\exp(-n^{1-\varepsilon}) \le p(n) \le 1/\operatorname{polylog}(n)$. Given an $(r(n), p(n), n \log n)$-dictionary, one can construct a $(r'(n), p'(n), s'(n))$-solution to the many-sets problem, with parameter $\delta = 2\varepsilon$, and with*

$$r'(n) = r(n) + (\log p(n)^{-1}) \cdot (\log \log n)^3,$$

$$p'(n) = p(n/\log \log n),$$

and

$$s'(n) = \frac{n \log n}{\log \log n}.$$

We will prove Proposition 152 by adapting the skeleton/storage-array approach outlined in the previous section. The basic idea will be to implement each A_i as a budget rotated trie, and then to share random bits between A_is in just the right way so that (1) the total number of random bits used a small; but (2) with good probability, only an $O(1/\log \log n)$ fraction of the elements present at any given moment will have experienced a failure when they were inserted. We then store the elements that experience failures in a backyard data structure implemented using the $(r(n), p(n), O(n \log n))$-dictionary that we are given.

Preliminaries on how to discuss the $\boldsymbol{A_i}$s. Since the size of a given skeleton A_i may fluctuate over time, the skeleton A_i will need to be rebuilt every time that its size changes by a constant factor. These rebuilds can be deamortized to take constant time per operation (and since the dictionary A_i is permitted to use $O(|S_i| \log |S_i|)$ bits, it is acceptable for the rebuilds to add a constant-factor space overhead to A_i).

At any given moment, define the ***skeletal size*** a_i of S_i to be the size that S_i was the most recent time that A_i was (logically) rebuilt. The skeletal size satisfies $a_i = \Theta(|S_i|)$ at all times, but has the convenient property that it only changes when a rebuild occurs. The skeletal size a_i will also be used to determine how we implement A_i (sets with different skeletal sizes may be implemented differently from one another)—this means that, each time A_i is rebuilt (and a_i changes), the implementation of A_i

may also change.

Defining a backyard structure. An important component of the data structure will be a ***backyard dictionary*** T, which is used to store a small number of keys/values space inefficiently. We implement T using the $(r(n), p(n), n \log n)$-dictionary that we are given, and we assume without loss of generality that $|T| \geq n/\log\log n$ at all times (if $|T| < n/\log\log n$, then we can pad it with dummy elements to bring the size up). This means that, at any given moment, T uses at most $O(r(n))$ random bits, has failure probability at most $O(p(n/\log\log n))$, and uses space at most

$$O\left((|T| + n/\log\log n) \cdot \log n\right)$$

bits. In order for T to meet the constraints of Proposition 152 we will need to establish that, with probability $1 - O(p'(n))$, we have

$$|T| \leq O(n/\log\log n) \tag{13.3}$$

at any given moment.

It is worth taking a moment to describe precisely what information T stores for each key/value pair (x, y) that it stores belonging to a given skeleton A_i: it stores the pair (i, x) as a key and it stores y as a value. Since different A_is have different key/value sizes (all of which are $\Theta(\log n)$), we pad the sizes of the keys/values to all be a fixed length $\Theta(\log n)$ in the backyard.[7]

Storing small A_is in the backyard automatically. Let c be a sufficiently large positive constant. We handle the A_is satisfying $a_i \leq \log^c n$ by simply placing *all* of their elements in the backyard T. The number of such A_is is trivially at most m, which by the definition of the many-sets problem, is at most $n/\operatorname{polylog} n \leq O(n/\log^{2c} n)$. The number of elements that these A_is contribute to the backyard is therefore at most $O(\log^c n \cdot n/\log^{2c} n) = o(n/\log^c n)$. Throughout the rest of the section, we will focus exclusively on A_is satisfying $a_i > \log^c n$.

Partitioning the remaining A_is into groups $G_{j,k}$. We say that each A_i is in ***category*** $j = \lfloor \log a_i \rfloor$. Within each category j, we partition the A_is (satisfying $a_i > \log^c n$) into

$$t_j = \frac{(\log\log n)^2 \log p(n)^{-1}}{j} \tag{13.4}$$

[7]As an additional subtlety, whenever an element is stored in the backyard, it should also be stored in the appropriate storage array B_i, that way whenever A_i is rebuilt, it can determine which of its elements are currently in the backyard, it can remove those elements from the backyard, and it can include those elements in the rebuild of A_i.

groups, $G_{j,1}, G_{j,2}, \ldots, G_{j,t_j}$ such that, for each group $k \in [t_j]$,

$$\sum_{A_i \in G_{j,k}} a_i \leq O(n/t_j).$$

Note that such a partition is feasible only if and only if we can guarantee that $a_i \leq O(n/t_j)$ for each i—fortunately, this follows from

$$\begin{aligned} a_i \leq n^{\delta} &\leq O\left(n \cdot \frac{(\log\log n)^2}{jn^{1-\varepsilon}}\right) && (\text{since } \varepsilon = \delta/2) \\ &\leq O\left(n \cdot \frac{(\log\log n)^2 \log p(n)^{-1}}{j}\right) && (\text{since } \exp(-n^{1-\varepsilon}) \leq p(n)) \\ &= O(n/t_j). \end{aligned}$$

For any given category j, the only difference between how we implement each of the groups $G_{j,1}, G_{j,2}, \ldots, G_{j,t_j}$ is that each group $G_{j,k}$ is implemented using a different sequence $R_{j,k}$ of $\Theta(j \log j)$ random bits.

Implementing each A_i. We now describe how to implement a given skeleton $A_i \in G_{j,k}$ using the $\Theta(j \log j) = \Theta(\log a_j \log\log a_j) = \Theta(\log |A_j| \log\log |A_j|)$ random bits $R_{j,k}$. Recall that we store the A_is satisfying $a_i \leq \log^c n$ in the backyard, so we need only focus here on A_is satisfying $a_i > \log^c n$.

Define a hash function $h_{j,k}$ mapping the elements $x \in A_i$ to $\Theta(\log |A_i|)$-bit string $h(x)$. The hash function is implemented using the family of hash functions given in Lemma 154 of Appendix 13.A, meaning that the hash function uses $\Theta(\log |A_j|)$ random bits, and avoids collisions on A_i with probability $1 - 1/\operatorname{poly}(|A_i|)$ (note that, since $a_i > \log^c n$, the precondition for the lemma is met). The skeleton A_i will map hashes $h_{j,k}(x)$, for $x \in A_i$, to indices in B_i.

We implement the dictionary A_i using a budget rotated trie (Theorem 146)—this uses $\Theta(\log |A_j| \log\log |A_j|)$ random bits and has failure probability $1/\operatorname{poly}(|A_j|)$ per insertion. We remark that, in addition to making use of the probabilistic guarantees offered by the budget rotated trie, we will also be making use of the especially simple way in which the data structure experiences failures: the only possible failure mode is that one of the bins in the trie overflows. This will allow for us to gracefully handle when the A_js fail: we store the element that experienced failure in a backup data structure, and we allow the A_j that caused the failure to continue as though that insertion never happened.

In more detail, there are two reasons that an insertion into A_i might fail: (1) the hash $h_{j,k}(x)$ of the element being inserted satisfies $h_{j,k}(x) = h_{j,k}(x')$ for some other $x' \in S_i$; or (2) the budget rotated trie used to implement A_i fails (i.e., the insertion would cause one of the bins in the trie to overflow). The probability of either of these events occurring at any given insertion is $1/\operatorname{poly}(|A_i|) = 1/\operatorname{poly}(2^j)$. Whenever an insertion into A_i fails, we store the key/value pair in the backyard data structure T instead. This means that our full data structure has two possible failure modes: the case where T itself fails, and the case where T becomes too large, violating (13.3).

Analyzing the probability of a failure. The backyard data structure T has failure probability at most $O(p(n/\log\log n))$, by design. Thus, our task is to bound the probability that (13.3) fails.

Lemma 153. *With probability* $1 - \operatorname{poly}(p(n))$, *the category* j *contributes at most* $O(n/(\log\log n)^2)$ *elements to* T*, at any given moment.*

Proof. We have already established that the A_is satisfying $a_i \le \log^c n$ deterministically contribute at most $O(n/(\log\log n)^2)$ elements to T (because they deterministically *have* at most $O(n/(\log\log n)^2)$ elements). Thus we focus here on the A_is satisfying $a_i > \log^c n$ (note that these are all in categories j satisfying $j > \Omega(\log\log n)$.

Now consider a category j satisfying $j = \Omega(\log\log n)$. As shorthand, we will use t to denote t_j and G_k to denote $G_{j,k}$. For each group G_k, $k \in [t]$, define X_k to be the number of elements that skeletons in G_k contribute to the backyard. We have that $X_k \le O(n/t)$ deterministically; and, since each element x in each $A_i \in G_k$ has a $1/\operatorname{poly}(2^j)$ probability of failure, we have that

$$\mathbb{E}[X_k] = \frac{n}{t\operatorname{poly}(2^j)}. \tag{13.5}$$

Finally, since each of the groups $G_1, G_2, \ldots, G_t$ use different random-bit sequences, the X_ks are independent. Defining $X'_k = X_k/\Theta(n/t)$, the number of elements that G_k contributes to the backyard can be expressed as $\Theta(\frac{n}{t} \cdot Y)$, where

$$Y = \sum_{k=1}^{t} X'_k.$$

The X'_k's are independent random variables in the range $[0,1]$ and, by (13.5),

$$\mathbb{E}[Y] = \sum_{k=1}^{t} \mathbb{E}[X'_k] = \sum_{k=1}^{t} \Theta(t/n)\mathbb{E}[X_k] = t/\operatorname{poly}(2^j). \tag{13.6}$$

Applying a Chernoff bound to Y, we get

$$\Pr[Y > (1+D)\mathbb{E}[Y]] \le \left(\frac{e^D}{(1+D)^{1+D}}\right)^{\mathbb{E}[Y]},$$

which for $D \ge \Omega(1)$ implies

$$\Pr[Y_j > D\mathbb{E}[Y]] \le D^{-\Omega(D\cdot\mathbb{E}[Y])}.$$

Set

$$\begin{aligned} D &= \frac{t}{(\log\log n)^2 \mathbb{E}[Y]} \\ &= \frac{\operatorname{poly}(2^j)}{(\log\log n)^2} && \text{(by (13.6))} \\ &\geq 2^{\alpha j} && \text{(since } j \geq \Omega(\log\log n)\text{)}, \end{aligned}$$

where α is a positive constant of our choice. Then we have

$$\begin{aligned} \Pr\left[Y_j > \frac{t}{(\log\log n)^2}\right] &\leq D^{-\Omega(D\cdot\mathbb{E}[Y])} \\ &= \exp\left(-\alpha j \cdot \Omega(D \cdot \mathbb{E}[Y])\right) && \text{(since } D \geq 2^{\alpha j}\text{)} \\ &= \exp\left(-\alpha j \cdot \Omega(t/(\log\log n)^2)\right) \\ &= \exp\left(-\alpha j \cdot \Omega(\log p(n)^{-1}/j)\right) && \text{(by (13.4))} \\ &= \exp\left(-\alpha \cdot \Omega(\log p(n)^{-1})\right) \\ &= \operatorname{poly}(p(n)). \end{aligned}$$

Thus, we have with probability $1 - \operatorname{poly}(p(n))$ that

$$Y \leq \frac{t}{(\log\log n)^2}.$$

The number of elements that category j contributes to the backyard is therefore at most

$$O\left(\frac{n}{t} \cdot Y\right) \leq O(n/(\log\log n)^2),$$

as desired. ∎

Putting the pieces together, the total size of T is $O(n/\log\log n)$ with probability

$$1 - O(p(n/\log\log n)) - (\log\log n) \cdot \operatorname{poly}(p(n)) = 1 - O(p(n/\log\log n)).$$

Bounding the number of random bits. Finally, we count the number of random bits used by the data structure. The backyard uses $O(r(n))$ random bits, so it suffices to bound the number of random bits used by the skeletons in each category. Note that different categories can use the same random bits as one another (since we do not require independence between categories), so it suffices to bound the number of random bits used by any given category of skeletons. In category j, there are t_j groups, each of which uses $\Theta(j \log j) = O(j \log\log n)$ random bits. The total number of random bits used by the category is therefore

$$O(j(\log\log n)t_j) = O\left(j \log\log n \cdot \frac{(\log\log n)^2 \log p(n)^{-1}}{j}\right) = O\left((\log\log n)^3 \log p(n)^{-1}\right).$$

In total, the number of random bits used by the data structure is

$$O(r(n)) + O\left((\log\log n)^3 \log p(n)^{-1}\right).$$

This completes the proof of Proposition 152, and thus also the proof of Theorem 148.

Appendices

13.A Universe Reduction Using $O(\log n)$ Random Bits

In this section, we extend the budget rotated trie to support keys from a universe U of super-polynomial size. Throughout the section, we set $U = [2^u]$ for some $u = n^{o(1)}$, and we assume that machine words are $\Theta(u)$ bits.

To support large keys, the natural approach is to first hash elements from U to a smaller universe U' of polynomial size, an then to store the $\Theta(\log n)$-bit keys in a hash table along with pointers to the full keys/values. Past work on load-balancing hash functions [123] has used a pair-wise independent hash function $h : [2^u] \to [\text{poly}(n)]$ to perform this reduction. This requires the use of $\Theta(u)$ random bits, which when u is large, is significantly larger than $\log n \log \log n$.

An appealing alternative to using pairwise-independent hash functions would be to instead use Pagh's construction [297] (which, in turn, is based on an earlier construction by Fredman, Komlós, and Semerédi [185]) of $(1 + o(1))$-universal hash functions that require only $O(\log n + \log \log u)$ random bits. The only minor problem with this construction is that it is not fully explicit. The construction requires access to a random prime number $p \in [\text{poly}(n)]$, but the only known time-efficient high-probability approaches to constructing such a prime number require $\omega(\log n \log \log n)$ random bits (see discussion in [183]).

Fortunately, this issue is relatively straightforward to solve. For completeness, we now give a construction for a simple family of hash functions that can be initialized in time $o(n)$ and used for universe reduction.

Lemma 154. *Let $n > u^c$ for a sufficiently large positive constant c and let $S \subseteq [2^u]$ be a set of size n. Let $\mathcal{P}$ be the set of prime numbers in the range $[n^{2/c}]$. Select $p_1, p_2, \ldots, p_{c^2}$ independently and uniformly at random from $\mathcal{P}$, and define the function $h : [2^u] \to [n^{2c}]$ by*

$$h(x) = (x \bmod p_1 p_2 \cdots p_{c^2}).$$

With probability $1 - 1/\text{poly}(n)$, h is injective on S.

Proof. The probability that $|h(S)| \neq S$ satisfies

$$\Pr[|h(S)| \neq S] \leq \sum_{s_1, s_2 \in S} \Pr[|s_1 - s_2| \text{ divisible by all of } p_1, p_2, \ldots, p_{c^2}],$$

where s_1 and s_2 are implicitly taken to be distinct. Since the p_is are independent, this is

$$\sum_{s_1, s_2 \in S} \left(\Pr[|s_1 - s_2| \text{ divisible by } p_1]\right)^{c^2}.$$

The quantity $|s_1 - s_2|$ is an element of $U = [2^u]$, and can thus have at most u distinct prime factors. Therefore,

$$\begin{aligned}\Pr[|h(S)| \neq S] &\leq \sum_{s_1,s_2 \in S} \left(\frac{u}{|\mathcal{P}|}\right)^{c^2} \\ &\leq \sum_{s_1,s_2 \in S} \left(\frac{n^{1/c}}{|\mathcal{P}|}\right)^{c^2}.\end{aligned}$$

By the Prime Number Theorem, the set $\mathcal{P}$ of primes in the range $[n^{2/c}]$ has size $\Omega(n^{2/c}/\log n)$. Therefore,

$$\begin{aligned}\Pr[|h(S)| \neq S] &\leq \sum_{s_1,s_2 \in S} O\left(\frac{n^{1/c}}{n^{2/c}/\log n}\right)^{c^2} \\ &\leq O\left(\sum_{s_1,s_2 \in S} \left(\frac{\log n}{n^{1/c}}\right)^{c^2}\right) \\ &\leq O\left(\sum_{s_1,s_2 \in S} \frac{\log^{c^2} n}{n^c}\right) \\ &\leq O\left(\frac{n^2 \log^{c^2} n}{n^c}\right) \\ &\leq 1/\operatorname{poly}(n).\end{aligned}$$

∎

Since all of the prime numbers in $[n^\varepsilon]$ can be enumerated in time $O(n^{2\varepsilon})$, we get the following corollary:

Corollary 155. *Let $u = n^{o(1)}$. For any constant $\delta > 0$, there exists an explicit family $\mathcal{H}$ of constant-time hash functions $h : [2^u] \to [\operatorname{poly}(n)]$ such that (a) a random function $h \in \mathcal{H}$ can be constructed in time $O(n^\delta)$ using $O(\log n)$ random bits; and (b) for any fixed set $S \subseteq U$ of size n, and for a random $h \in \mathcal{H}$, we have that $|h(U)| = |U|$ with probability $1 - 1/\operatorname{poly}(n)$.*

We can use Corollary 155 to construct a version of the budget rotated trie that supports large universes.

Theorem 156. *Let $u = n^{o(1)}$, suppose that keys/values are u bits, and assume a machine word of size at least $\Omega(u)$ bits. The budget rotated trie uses $O(\log n \log\log n)$ random bits, it uses $O(nu)$ bits of space, and it supports insert/delete/query operations on up to n keys/values at a time. The data structure can be initialized in time $O(n^\varepsilon)$, for a positive constant ε of our choice, and each insert/delete/query operation takes constant time with probability $1 - 1/\operatorname{poly}(n)$.*

To eliminate the $O(n^\varepsilon)$ initialization cost, we can also construct a dynamic version of the same data structure, where there is some upper bound N on the data structure's size, but where the true size n changes over time. Every time that the data structure's size changes by a constant factor, we rebuild it based on the new value of n. Each rebuild takes time $O(n)$ (with high probability in n), but the cost of a rebuild can be spread across $\Theta(n)$ operations. The properties of this new data structure can be summarized with the following corollary.

Corollary 157. *Let $u = N^{o(1)}$, suppose that keys/values are u bits, and assume a machine word of size at least $\Omega(u)$ bits. The dynamic budget rotated trie uses $O(\log N \log \log N)$ random bits and supports insert/delete/query operations on up to N keys/values at a time. If it is storing n key/value pairs, then it uses $O(nu)$ bits of space, and each insert/delete/query operation takes constant time with probability $1 - 1/\operatorname{poly}(n)$.*

Chapter 14

Tight Bounds for Monotone Minimal Perfect Hashing

14.1 Introduction

The **monotone minimal perfect hash function** (MMPHF) problem is the following indexing problem. Given a set $S = \{s_1, \ldots, s_n\}$ of n distinct keys from a universe U of size u, create a data structure $\mathbf{D}$ that answers the following query:

$$\textsc{Rank}(q) = \begin{cases} \text{rank of } q \text{ in } S & q \in S \\ \text{arbitrary answer} & \text{otherwise.} \end{cases}$$

The name of the problem comes from interpreting the data structure $\mathbf{D}$ as a hash function: given a sorted array $A = [a_1, \ldots, a_n]$, $\mathbf{D}$ is a function mapping each a_i to its position i. Such a hash function is *minimal*, meaning that it maps n items to n distinct positions, and *monotone*, meaning that whenever $a_i < a_j$ we have $\mathbf{D}(a_i) < \mathbf{D}(a_j)$, and vice versa.

It may seem strange at first glance that $\mathbf{D}$ is permitted to return arbitrary answers on negative queries. A key insight, however, is that this relaxation allows for asymptotic improvements in space efficiency: whereas the set $\mathcal{S}$ would require $\Omega(n \log(u/n))$ bits to encode, Belazzougui, Boldi, Pagh and Vigna [64] show that it is possible to construct an MMPHF $\mathbf{D}$ using as few as $O(n \log\log\log u)$ bits, while supporting $O(\log\log u)$-time queries.

The remarkable space efficiency of MMPHF makes it useful for a variety of practical applications (e.g., in security [110], key-value stores [244] and information retrieval [280]). A high-performance implementation can be found in the Sux4J library [63, 108]. MMPHF has also been widely used in the theory community for the design of space-efficient combinatorial pattern-matching algorithms (see, e.g., [62, 65–68, 127, 191, 203]).

Despite the widespread use of MMPHF, it remains an open question [64, 109, 154] to determine the optimal bounds for solving this problem. The best lower bound achieved so far [63, 154] is $\Omega(n)$ bits (which follows immediately from the same lower bound for minimal perfect hashing [258]). Even disregarding applications (and the running time to answer queries), the information-theoretic question as to how many bits a MMPHF requires has been posed as a problem of independent combinatorial interest [154].

Our result. We fully settle this question by establishing the following result:

Result 158 (Formalized in Theorem 162)**.** *Any data structure (deterministic or randomized) for monotone minimal perfect hashing of any collection of n elements from a universe of size u requires $\Omega(n \log\log\log u)$ expected bits to answer every query correctly. The lower bound holds whenever u is at least $n^{1+1/\sqrt{\log n}}$ and at most* $\exp(\exp(\text{poly}(n)))$.

Thus, somewhat surprisingly, the $O(n \log\log\log u)$ bound achieved by [63] is asymptotically optimal. We also note that the boundary conditions on u in Result 158 are natural in the following sense. There are two trivial solutions for the

MMPHF. One encodes the entire input set S in $O(u)$ bits and the other builds a perfect hash table mapping from elements of S to their rank using $O(n \log n)$ expected space. Thus, when u is very small, for example $u = O(n)$, the first solution achieves $O(u) = o(n \log\log\log u)$ bits. On the other hand, when u is very large, that is when u is even beyond $\exp(\exp(\text{poly}(n)))$, then the $O(n \log n)$-bit solution uses $o(n \log\log\log u)$ bits. Our lower bound in Result 158 covers almost the entire range in between.

The lower bound achieved by Result 158 is remarkably general: it applies independently of the running time of the data structure; and it applies even to randomized data structures that are permitted to store their random bits for free.

Our techniques. The most intuitive approach toward proving a lower bound of d bits on the size of an MMPHF is to encode a d-bit string into the state of the data structure. This approach is already hindered by the fact that MMPHFs only support *positive queries*, however. If the user already knows which elements are in the input, then the MMPHF encodes no interesting information — but if the user only has partial information about the input, then the user can only get useful information from a small portion of possible MMPHF queries. The previous $\Omega(n)$ lower bound of [63, 154, 258] addresses this as follows: consider any bit-string $x \in \{0, 1\}^d$ and define:

$$S(x) := \{3, 6, \ldots, 3d\} \cup \{3i + 1 \mid i \in [d], x_i = 1\} \cup \{3i - 1 \mid i \in [d], x_i = 0\}.$$

For every $i \in [d]$, firstly, $3i$ belongs to $S(x)$ and thus is a positive query, and secondly, $\textsc{Rank}(3i) = 2 \cdot (i - 1) + x_i$. This allows us to recover x from any MMPHF for $S(x)$, proving a lower bound of $d = \Omega(n)$ bits for MMPHF on size-n subsets of universe $[3n+1]$. This approach, however, seems to be stuck at proving any $\omega(n)$ lower bound as these "direct encodings" ignore the delicate interaction between different elements in the input set[1].

To get around these obstacles, we take a fundamentally different approach to proving Result 158. We construct a "conflict graph" G whose vertices are all the possible inputs to an MMPHF problem for a fixed n and u. Two vertices are adjacent in G if they cannot have the same MMPHF representation, that is, if the vertices share an element but with a different rank. Any MMPHF induces a proper coloring of this graph, where the color of a vertex corresponds to its MMPHF representation. As a result, the *chromatic number* of the conflict graph is a lower bound on how many different MMPHF representations we must have, which implies that some input must have a representation of size at least $\log \chi(G)$ bits. This reduces our task to the combinatorial problem of lower bounding $\chi(G)$.[2]

[1] *Any* lower bound of d bits for a data structure immediately implies an encoding of d-bit strings in the state of the data structure by just assigning one bit-string to each state. This means that there is never a formal proof that one *cannot* encode a bit-string in a data structure and still prove a lower bound.

[2] Slightly more care must be taken when bounding the *expected* size of a MMPHF that is permitted to take different sizes on different inputs.

The problem of bounding chromatic number of graphs defined over these types of set-systems has a rich history in the discrete math literature; see, e.g. [167, 170, 190, 328]. For instance, Erdős and Hajnal [170] study *shift-graphs* that have vertices corresponding to n-element subsets of $[u]$ and edges between vertices $(a_1, a_2, \ldots, a_n)$ and $(a_2, \ldots, a_n, a_{n+1})$ for all $a_1 < a_2 < \ldots < a_{n+1}$. They prove that the chromatic number of the shift-graph is $(1 + o(1)) \cdot \log^{(n-1)}(u)$, namely, the $(n-1)$-th iterated logarithm of u. The shift-graph is a subgraph of our conflict graph. Thus, by taking $u = 2 \upuparrows (n+1)$, i.e., the *tower* of twos of height $n+1$, we can have $\chi(G) = 2^{\omega(n)}$, and thus prove an $\omega(n)$ lower bound for MMPHF on n-subsets of (extremely large) universes of size $u = 2 \upuparrows (n+1)$. This is the starting point of our approach. We now need to dramatically decrease the size of the universe, while also dramatically increasing the bound on the chromatic number by considering the conflict graph itself, and not only its shift-subgraph.

To lower bound the chromatic number of the conflict graph, we consider the relaxation of this problem via *fractional colorings* (see Section 14.2.2). Given that this latter problem can be formulated as a linear program (LP), a natural way for proving a lower bound on its value is to exhibit a feasible *dual* solution instead[3]. This corresponds to the following problem: exhibit a distribution on vertices of the graph so that for any independent set, the probability that a vertex sampled from the distribution belongs to the independent set is bounded by p; this then implies that the fractional chromatic number (and in turn the chromatic number) are lower bounded by $1/p$. The main technical novelty of our work lies in the introduction of a highly non-trivial such distribution and the analysis of this probability bound for each independent set (we postpone the overview of this part to Section 14.4.1 after we setup the required background). This allows us to lower bound the (fractional) chromatic number of the conflict-graph by $\Omega(n \log n)$ when the universe is of size $u = 2^{2^{\text{poly}(n)}}$ which gives an $\Omega(n \log\log\log u)$ lower bound for MMPHF on such universes.

Working with fractional colorings, beside being an immensely helpful analytical tool, has several additional benefits for us. Firstly, unlike standard (integral) colorings, fractional colorings admit a natural *direct product* property over a certain union of graphs; this allows us to extend the lower bound for MMPHF from universes of size doubly exponential in n (which are admittedly not the most interesting setting of parameters), all the way down to universes of size $n^{1+o(1)}$. Secondly, unlike the (integral) chromatic number, which yields a lower bound only on the space of deterministic MMPHFs, we show that lower bounding the fractional chromatic number allows us to prove a lower bound even for randomized MMPHFs that have access to their randomness for free. We believe this technique, namely, defining a proper conflict graph and bounding its fractional coloring by exhibiting a feasible dual solution, may be applicable to many other data structure problems and is therefore interesting in its own right.

[3]This is an inherently different technique than the one used in [170] for the shift-graph, as it is known that the fractional chromatic number of the shift-graph is $O(1)$ (see, e.g. [328]).

14.2 Preliminaries

Notation. For any integer $t \geq s \geq 1$, we let $[t] := \{1, \ldots, t\}$ and let $[s, t] = \{s, \ldots, t\}$. For a tuple $(X_1, \ldots, X_t)$, we further define $X_{<i} := (X_1, \ldots, X_{i-1})$ and $X_{-i} := (X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_t)$.

14.2.1 Problem Definition and Model of Computation

For any integer $n, u \geq 1$, we let $\mathbf{D}(n, u)$ be an MMPHF indexing algorithm for size-n subsets of $[u]$. That is, if $\mathcal{S}_{n,u} = \{S \subseteq [u] \text{ s.t. } |S| = n\}$ then for all $S \in \mathcal{S}_{m,u}$, $\mathbf{D}(S)$ is the MMPHF index for S.

For any fixed choice of random bits r, we use $\mathbf{D}^r$ to denote the resulting MMPHF with random bits r. Note that for any fixed choice of r, $\mathbf{D}^r$ is deterministic. For any $S \in \mathcal{S}_{n,u}$ and randomness r, define $d^r(S)$ as the size in bits of the MMPHF index $\mathbf{D}^r(S)$. Define:

$$d(n, u) := \max_{S \in \mathcal{S}_{n,u}} \mathop{\mathbb{E}}_{r} \left[d^r(S) \right].$$

When n and u are clear, we drop them and refer simply to $\mathbf{D}$ and d.

In this definition of size, we are giving the MMPHF a big advantage: we are not charging the algorithm for storing its randomness. In other words, the algorithm has access to a tape of random bits chosen independent of the input that it can use for both creating the index as well as answering the queries. Furthermore, we also allow the algorithm unbounded computation time. Thus, the only measure of interest for us is the *size* of the index. Finally, any deterministic MMPHF in this model is simply a randomized MMPHF that ignores its random bits and thus we will only focus on randomized MMPHFs from now on.

14.2.2 Fractional Colorings

A key tool that we use in establishing our lower bound is the notion of a **fractional coloring** of a graph. We now review the basics of fractional colorings, which we need in our proofs.

Let $G = (V, E)$ be any undirected graph. A proper coloring of G is any assignment of colors to vertices of G so that no edge is monochromatic. The chromatic number $\chi(G)$ is the minimum number of colors in any proper coloring of G. The fractional relaxation of chromatic number can then be defined as follows.

Let $\mathcal{I}(G) \subseteq 2^V$ denote the set of all independent sets in G, and for any vertex $v \in V$, define $\mathcal{I}(G, v)$ as the set of all independent sets that contain the vertex v. A fractional coloring of G is any assignment of $x = (x_1, \ldots, x_{\mathcal{I}(G)}), 0 \leq x_i \leq 1$, to the independent sets of G satisfying the following constraint:

$$\text{for every vertex } v \in V\text{:} \quad \sum_{I \in \mathcal{I}(G,v)} x_I \geq 1.$$

The ***value*** $|x|$ of a fractional coloring x is given by $\sum_{I \in \mathcal{I}(G,v)} x_I$.

The ***fractional chromatic number*** $\chi_f(G)$ is the minimum value of any fractional coloring of G. This quantity can be formalized as a linear program (LP):

$$\chi_f(G) := \min_{x \in \mathbb{R}_{\geq 0}^{\mathcal{I}(G)}} \sum_{I \in \mathcal{I}(G)} x_I \quad \text{subject to} \quad \sum_{v \in \mathcal{I}(G,v)} x_I \geq 1 \quad \forall v \in V. \tag{14.1}$$

Any proper coloring of G with k colors induces a solution x of value k to this LP, where x_I is set to 1 for the independent sets I that correspond to color classes in the coloring. Thus the LP given by Equation (14.1) is indeed a relaxation of the original coloring problem.

Fact 159. *For any graph G, $\chi_f(G) \leq \chi(G)$.*

It is worth mentioning that at the same time $\chi(G) = O(\log |V(G)|) \cdot \chi_f(G)$ using the standard randomized rounding argument (we do not use this direction explicitly in this chapter).

A primal-dual analysis of the fractional-chromatic-number LP implies the following results. These results are standard but we provide proofs in Section 14.A for completeness.

Proposition 160. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be arbitrary graphs. Define $G_1 \vee G_2$ as a graph on vertices $V_1 \times V_2$ and define an edge between vertices (v_1, v_2) and (w_1, w_2) whenever (v_1, w_1) is an edge in G_1 <u>or</u> (v_2, w_2) is an edge in G_2. Then, $\chi_f(G_1 \vee G_2) = \chi_f(G_1) \cdot \chi_f(G_2)$.*

Proposition 160 allows us to determine χ_f of a product of several graphs by focusing on each individual graph separately.

Proposition 161. *For any graph $G = (V, E)$,*

$$\chi_f(G) = \max_{\text{distribution } \mu \text{ on } V} \; \min_{I \in \mathcal{I}(G)} \left(\Pr_{v \sim \mu} (v \in I) \right)^{-1}.$$

Proposition 161 provides us with a tool to lower bound χ_f by finding a suitable distribution on the vertices so that no independent set has a significant probability of being by the distribution.

14.3 A Lower Bound for MMPHF via Fractional Colorings

We can now formally state the main theorem of this chapter.

Theorem 162 (Formalization of Result 158)**.** *For any $n, u \in \mathbb{N}^+$ such that $n \cdot 2^{\sqrt{\log n}} \leq u \leq 2^{n^{n^2+n}}$, and for any MMPHF algorithm $\mathbf{D}(n,u)$,*

$$d(n,u) = \Omega(n \log\log\log u).$$

The rest of the chapter presents the proof of Theorem 162. We spend the rest of the section reframing the theorem in terms of the fractional chromatic number of a certain graph associated with MMPHF problem—we will then show how to lower bound the fractional chromatic number in the next section.

14.3.1 Conflict Graph and its Fractional Chromatic Number

Let $m \geq 1$ be an integer and define $M := 2^{m^{m^2+m}}$. Define the graph $G(m) := (V(m), E(m))$ as:

- The vertex set is $V(m) = \mathcal{S}_{m,M}$, that is, the size-m subsets of $[M]$. We denote each vertex $v \in V(M)$ by the m-tuple $v := (v_1, \ldots, v_m)$ where $0 < v_1 < v_2 < \cdots < v_m \leq M$.
- The edge set $E(m)$ is defined as follows. Let $v = (v_1, \ldots, v_m)$ and $w = (w_1, \ldots, w_m)$ be any two vertices in $V(M)$. Then, there is an edge $(v,w) \in G(m)$ iff there exists some pair of indexes $i \neq j \in [m]$ such that $v_i = w_j$.

We refer to $G(m)$ as the **conflict graph** of m. The following lemma clarifies our interest in this graph by showing that fractional chromatic number of $G(m)$ can be used to lower bound size of any MMPHF (for certain parameters of input).

Lemma 163. *Let $m \geq 1$ be an integer and let $M = 2^{m^{m^2+m}}$. Consider any MMPHF $\mathbf{D}(m,M)$. Then*

$$d(m,M) \geq (\log \chi_f(G(m)) - 2)/2.$$

Proof. Consider any two vertices $v, w \in G(m)$. If there is an edge between v and w, then there exists an element $z = v_i = w_j, i \neq j$. Therefore for every choice of randomness r, $\mathbf{D}^r(v) \neq \mathbf{D}^r(w)$, because query z must return i on $\mathbf{D}^r(v)$ and j on $\mathbf{D}^r(w)$. This implies that for every r, the set of vertices v with the same $\mathbf{D}^r(v)$ form an independent set in $G(m)$ (and the collection of these sets is a coloring of $G(m)$). We use $\mathcal{I}^r$ to denote these independent sets in $G(m)$ for this choice of r.

On the other hand, by Proposition 161, there exists a distribution μ on $V(m)$ such that

$$\chi_f(G(m)) = \min_{I \in \mathcal{I}(G(m))} \left(\Pr_{v \sim \mu} (v \in I) \right)^{-1}. \tag{14.2}$$

Let us fix that distribution. Under this distribution, by the definition of d,

$$d = d(m,M) = \max_{v \in V(m)} \mathbb{E}_r[d^r(v)] \geq \mathbb{E}_{v \sim \mu} \mathbb{E}_r [d^r(v)] = \mathbb{E}_r \mathbb{E}_{v \sim \mu} [d^r(v)].$$

An averaging argument now implies that there exists a choice r^* of random bits such that

$$\mathop{\mathbb{E}}_{v\sim\mu}\left[d^{r^*}(v)\right] \leq d.$$

By Markov's inequality, with probability at least $1/2$, for $v \sim \mu$, we have that $d^{r^*}(v) \leq 2d$.

Recall that $\mathbf{D}^{r^*}(v)$ corresponds to an independent set in $\mathcal{I}_{r^*}$. Moreover, there can be at most $2^{2d+1} - 2$ independent sets I in $\mathcal{I}_{r^*}$ such that for all $v \in I$, $d^{r^*}(v) \leq 2d$; this is because there are at most $2^{2d+1} - 2$ choices for $\mathbf{D}^{r^*}(v)$ across all $v \in V(m)$ that can use up to $2d$ bits in their index (as the number of non-empty binary strings of length at most $2d$ is $2^{2d+1} - 2$). Since a random $v \sim \mu$ belongs to one of these $2^{2d+1} - 2$ independent sets with probability at least half, we necessarily have some independent set $I \in \mathcal{I}_{r^*}$ where

$$\Pr_{v\sim\mu}(v \in I) \geq \frac{1}{2\cdot(2^{2d+1}-2)} \geq \frac{1}{2^{2d+2}}.$$

Plugging in this bound in Equation (14.2), we have,

$$\chi_f(G(m)) \leq 2^{2d+2},$$

which implies that $d \geq (\log \chi_f(G(m) - 2)/2$, concluding the proof. ∎

Lemma 163 reduces our task of proving Theorem 162 to establishing a lower bound on $\chi(G(m))$. This will be accomplished by the following lemma, which we prove in Section 14.4.

Lemma 164. *There is an absolute constant $\eta > 0$ such that for every sufficiently large $m \geq 1$,*

$$\chi_f(G(m)) \geq m^{\eta\cdot m}.$$

By plugging in the lower bound of $\chi_f(G(m))$ from Lemma 164 inside Lemma 163, we get that for any sufficiently large $n \geq 1$ and universe size $u = 2^{m^{m^2+m}}$, the lower bound on the MMPHF problem is $\Omega(n \log n) = \Omega(n \log\log\log u)$ as $\log n = \Theta(\log\log\log u)$ here.

Thus Lemmas 164 and 163 can be combined to prove Theorem 162 modulo a serious caveat: the lower bound only holds for instances of the problem wherein the universe size is larger than doubly exponential in n, which is admittedly not the most interesting setting of the parameters. In the next subsection, we use a simple graph product argument (plus Proposition 160) to extend this lower bound to the whole range of parameters u considered by Theorem 162.

14.3.2 Extending the MMPHF Lower Bound to Small Universes

For every pair of integers $m, \ell \geq 1$, define $G(m, \ell) = (V(m, \ell), E(m, \ell))$ as the **ℓ-offset conflict graph** where the vertex set $V(m, \ell)$ is the set of all size-m subsets of $[\ell+1, M+\ell]$, and the edge set $E(m, \ell)$ is defined as in normal conflict graphs. (Thus $G(m, 0) = G(m)$.)

Furthermore, for every integer $m, k \geq 1$, we define the **k-fold conflict graph**, denoted by $G^{\oplus k}(m)$, as the graph:

$$G^{\oplus k}(m) = (V^{\oplus k}(m), E^{\oplus k}(m)) := G(m, 0) \vee G(m, M) \vee G(m, 2M) \vee \cdots \vee G(m, (k-1)M),$$

where '$\vee$' denotes the graph product in Proposition 160. The direct interpretation of the nodes of $V^{\oplus k}(m)$ is a product of tuples from disjoint ranges, but we can also interpret it as a single tuple of length $k \cdot m$. This way, $G^{\oplus k}(m)$ is a subset of the conflict graph on km-size subsets of $[k \cdot M]$ and it makes sense to compute $\mathbf{D}(v)$ for any $v \in V^{\oplus k}(m)$.

Therefore, by Lemma 163, we again have a lower bound of $\Omega(\log \chi_f(G^{\oplus k}(m)))$ for MMPHF on tuples of length $n = km$ from a universe of size $u = kM$.

By Proposition 160, combined with Lemma 164, we have,

$$\log \chi_f(G^{\oplus k}(m)) = k \cdot \log \chi_f(G(m)) \geq \Omega(k \cdot m \cdot \log m) = \Omega(n \log m).$$

Consider a choice of

$$m = (\log\log n)^{1/6} \quad \text{and} \quad k = n/(\log\log n)^{1/6},$$

which in turn gives us

$$u = k \cdot 2^{m^{m^2+m}} \ll k \cdot 2^{2^{m^3}} = \frac{n}{(\log\log n)^{1/6}} \cdot 2^{2^{\sqrt{\log\log n}}} \ll n \cdot 2^{\sqrt{\log n}}.$$

By the above equation, we have a lower bound of $\Omega(n \log\log\log u)$ for MMPHF given that in this case, $\log m = \Theta(\log\log\log u)$. Thus, so far, we have proven Theorem 162 on both its boundary cases, namely, when $u = n \cdot 2^{\sqrt{\log n}}$ and when $u = 2^{n^{n^2+n}}$. The proof can now be extended to the full range of the parameters in the middle by re-parameterizing k appropriately; see Section 14.B for the complete argument.

We conclude that in order to finish the proof of Theorem 162, we need only establish Lemma 164.

14.4 Fractional Chromatic Number of Conflict Graphs

In this section, we establish a lower bound on the fractional chromatic number of the conflict graph $G(m)$ for any (large enough) $m \geq 1$, and we thereby prove Lemma 164.

Proposition 161 gives us a clear path for proving the lower bound on $\chi_f(G(m))$ given by Lemma 164: we can design a distribution μ on vertices of $V(m)$ and then, for every independent set $I \in \mathcal{I}(G(m))$, we can upper bound the probability that v sampled from μ belongs to I. As χ_f in Proposition 161 is maximum over all possible distributions, our distribution provides a lower bound for $\chi_f(G(m))$.

To continue, we need the following simple interpretation of the (maximal) independent sets in $G(m)$.

Observation 165. *Any maximal independent set I in $G(m)$ can be identified by a function $f_I : [M] \to [m]$ such that for every vertex $v = (v_1, \ldots, v_m) \in I$, $f_I(v_i) = i$.*

Proof. Consider any two vertices $v, w \in I$. Since there is no edge between $v = (v_1, \ldots, v_m)$ and $w = (w_1, \ldots, w_m)$ in $G(m)$, whenever $v_i = w_j$, we necessarily have that $i = j$. Thus, any element of $e \in [M]$ can only appear in a single index $i_e \in [m]$ throughout all vertices $v \in I$ (or does not appear at all in v). We can thus define $f_I(e)$ to be i_e, giving us a functino f_I with the desired property.

To complete the proof, we show that f_I uniquely identifies I. If we define I' to be the set of vertices $v = (v_1, \ldots, v_m) \in I$ satisfying $f_I(v_i) = i$ for all i, then I' is an independent set satisfying $I \subseteq I'$. Since I is assumed to be maximal, it follows that $I = I'$, meaning that we can recover I from f_I. ∎

Observation 165 allows us to reduce Lemma 164 to the following lemma about m-tuples of increasing integers. Proving Lemma 166 is the main technical contribution of our work.

Lemma 166. *There is an absolute constant $\eta > 0$ such that for any sufficiently large $m \geq 1$ and $M = 2^{m^{m^2+m}}$, the following is true. There exists a distribution on m-tuples of increasing numbers $X_1 < \cdots < X_m$ from $[M]$ such that for any function $f : [M] \to [m]$,*

$$\Pr_{(X_1,\ldots,X_m)} (\forall i \in [m] : f(X_i) = i) \leq m^{-\eta \cdot m}.$$

Before proving Lemma 166, we show how it implies Lemma 164.

Proof of Lemma 164 (assuming Lemma 166). Any choice of $(X_1, \ldots, X_m)$ in Lemma 166 can be mapped to a unique vertex $v \in G(m)$ and vice versa. Thus, $(X_1, \ldots, X_m)$ induces a distribution μ on vertices $V(m)$: sample $(X_1, \ldots, X_m)$ and return the vertex $v = (v_1, \ldots, v_m)$ where $v_i = X_i$ for all $i \in [m]$. Moreover, for any maximal independent set $I \in \mathcal{I}(G)$, by Observation 165, the vertex corresponding to $(X_1, \ldots, X_m)$ belongs to I iff $f_I(X_i) = i$ for all $i \in [m]$. Thus,

$$\Pr_{v \sim \mu}(v \in I) = \Pr_{(X_1,\ldots,X_m)}(\forall i \in [m] : f(X_i) = i) \leq m^{-\eta \cdot m}.$$

As every independent set of $G(m)$ is a subset of some maximal independent set, the upper bound continues to hold for every independent set in $G(m)$.

By Proposition 161,

$$\chi_f(G(m)) \geq \min_{I \in \mathcal{I}(G(m))} \Big(\Pr(v \in I) \Big)^{-1} \geq m^{\eta \cdot m},$$

concluding the proof. ▮

The rest of the section proves Lemma 166. We start with a high-level overview in Section 14.4.1. We then define the distribution that we will use for the proof of Lemma 166 (Section 14.4) and analyze it to establish Lemma 166 (Section 14.4.3). The probability distribution that we construct in these sections should be viewed intuitively as a "hard" input distribution on inputs to the MMPHF problem (in the spirit of Yao's minimax principle).

14.4.1 A High-Level Overview of the Proof

The proof of Lemma 166 is quite dense and requires both a highly delicate probability distribution and several intricate technical arguments. Thus, before getting into the details of this proof, we provide a (very) high-level overview of the logic behind it. In order to convey the intuition, we omit many details from this subsection, instead limiting ourselves to an informal discussion.

The distribution in Lemma 166 is roughly as follows: we start with a "window" $\texttt{Win}_1$ which is the interval $[1 : M]$, and then sample X_1 uniformly at random from $\texttt{Win}_1$. We then pick window $\texttt{Win}_2$ to be $[X_1 + 1 : X_1 + w_2]$ for an integer $w_2 > 1$ chosen randomly from a carefully designed distribution. Similarly to before, X_2 will be chosen uniformly from $\texttt{Win}_2$. We continue like this by picking a new window $\texttt{Win}_i = [X_{i-1}+1 : X_{i-1}+w_i]$ for each $i \in [m]$ by sampling each w_i from a distribution that is constructed based on $(w_1, \ldots, w_{i-1})$, and then sampling X_i from $\texttt{Win}_i$. Note that, by design, we will satisfy $X_1 < X_2 < \ldots < X_m$.

The key property that this distribution achieves can be explained informally as follows. For any index $i \in [m]$, there is a recursive partitioning of the window $\texttt{Win}_i$ into "dense" and "sparse" intervals, where an interval $I \subseteq \texttt{Win}_i$ is dense (with respect to the function f and the index i) if at least an $\Omega(1/m)$ fraction of entries $j \in I$ satisfy $f(j) = i$, and otherwise I is sparse. The central property that our distribution ensures is that, if the random choice of X_i places it in a dense interval, then (with very high probability) the *final window* $\texttt{Win}_m$ will itself end up being dense (i.e., for at least a $2/m$ fraction of $j \in \texttt{Win}_m$, $f(j) = i$).

Establishing this property is quite challenging and involves defining the distribution of w_i's in a highly non-uniform manner (in terms of their values); this is also the source of the doubly exponential dependence of range M on the number of indices m. We postpone the details on how this property can be achieved to the actual proof and focus on why it is a useful property for us.

The analysis of the distribution now uses the property in a potential-function style argument. For each X_i, it is either sampled from a sparse interval or a dense one. If X_i is sampled from a sparse interval I, then no matter the past iterations, the probability that $f(X_i) = i$ is at most $(2/m)$, since at most $(2/m)$ fraction of I can have value $f(j) = i$ by the definition of it being sparse. On the other hand, if X_i is chosen from a dense interval, then at least a $(2/m)$ fraction of entries of $\mathtt{Win}_m$ should be mapped to i by f as well (by our property). Seeing $\mathtt{Win}_m$ as a potential function now, we have that this latter step can only happen for $(m/2)$ iterations $i \in [m]$—indeed, each time that this happens for some i, we commit some $(2/m)$ fraction of indices $j \in \mathtt{Win}_m$ to having $f(j) = i$, and these sets indices must be disjoint. As a result, we have that only at least $(m/2)$ iterations $i \in [m]$ sample X_i from a sparse interval. Thus,

$$\Pr(f(X_1) = 1, \ldots, f(X_m) = m) \leq \prod_{\substack{i:\ X_i \text{ chosen from} \\ \text{a sparse interval}}} \Pr\left(f(X_i) = i \mid f(X_1) = 1, \ldots, f(X_{i-1}) = (i-1)\right)$$
$$\leq O\left(\frac{1}{m}\right)^{m/2} = m^{-\Omega(m)},$$

as desired for the proof of Lemma 166.

The main challenge in formalizing the above argument is the design and analysis of the distribution so that the property discussed above holds. Note also that the property cannot hold deterministically—another challenge is to show that it holds with such high probability that the risk of the property ever failing (across the entire construction) can be ignored.

14.4.2 The Hard Input Distribution in Lemma 166

The distribution is defined as follows.

<u>The distribution in Lemma 166:</u>

(i) Let $k = m^m$, $S_0 = k^{m+1}$, and $X_0 = 0$.

(ii) For $i = 1$ to m:

(a) Sample two random numbers Y_i from $[2^{S_{i-1}}]$ and Z_i from $[k-1]$ uniformly at random.

(b) Define the random variables of iteration i as:

$$X_i = X_{i-1} + Y_i \qquad \text{and} \qquad S_i = S_{i-1} - k^{m-i+1} \cdot Z_i.$$

(*iii*) Return $(X_1, \ldots, X_m)$ as the resulting random variables.

To avoid ambiguity, we use lower case letters (s_i, x_i, y_i, z_i) to denote realizations of random variables (S_i, X_i, Y_i, Z_i) for $i \in [m]$.

We have the following basic observation on the range of numbers created in this distribution.

Observation 167. *Every choice of* $(X_1, \ldots, X_m)$ *and* $(S_1, \ldots, S_m)$ *satisfy the following properties:*

(*i*) Monotonicity: *for all* $i \in [m]$, $X_i > X_{i-1}$ *and* $S_i \leq S_{i-1} - m^m$ *(and* S_i, X_i *are integers).*

(*ii*) Boundedness: *for every* $i \in [m]$, $X_m \leq X_i + (m-i) \cdot 2^{S_i}$ *and* $S_m \geq S_i - k^{m-i+1} \geq 0$.

Proof. Monotonicity of X_i's holds as Y_i's are positive. Monotonicity for S_i's holds because Z_i's are positive and $k^{m-i+1} \geq k^{m-m+1} \geq k = m^m$, meaning that we always have $S_i \leq S_{i-1} - m^m$.

For part (*ii*), we have,

$$X_m = X_i + \sum_{j=i+1}^{m} Y_j \leq X_i + \sum_{j=i+1}^{m} 2^{S_{j-1}} \leq X_i + (m-i) \cdot 2^{S_i},$$

which proves the boundedness of X_i's. For S_i's,

$$S_m = S_i - \sum_{j=i+1}^{m} k^{m-j+1} \cdot Z_j \geq S_i - k^m \cdot (k-1) \cdot \sum_{j=i}^{m-1} k^{-j} \geq S_i - k^{m-i+1}.$$

(as $\sum_{j=i}^{m-1} k^{-j} \leq \sum_{j=i}^{\infty} k^{-j} = k^{-i+1} \cdot (k-1)^{-1}$)

Finally, by this bound, we have $S_m \geq S_0 - k^{m+1} \geq 0$ as $S_0 = k^{m+1}$. ∎

When discussing $(X_1, \ldots, X_m)$, we will also need some further definitions:

- For any realization $(s_{<i}, x_{<i})$, we define the **window** of iteration $i \in [m]$, $\texttt{Win}_i := \texttt{Win}_i(s_{<i}, x_{<i})$, as the support of the random variable X_i conditioned on $(s_{<i}, x_{<i})$, i.e.,

  $$\texttt{Win}_i := \texttt{Win}_i(s_{<i}, x_{<i}) = [x_{i-1} + 1 : x_{i-1} + 2^{s_{i-1}}].$$

  Notice that $|\texttt{Win}_i(s_{<i}, x_{<i})| = 2^{s_{i-1}}$ and $\texttt{Win}_i$ is determined by $(s_{<i}, x_{<i})$.

- Similarly, for any fixed choice of $(s_{<i}, x_{<i})$, consider the following numbers:

$$w_{i,j} := 2^{s_{i-1} - j \cdot k^{(m-i+1)}} \quad \text{for all } j \in \{0, \ldots, k\}. \tag{14.3}$$

  This way, $|\texttt{Win}_{i+1}(s_{<i}, x_{<i})|$ is chosen uniformly at random from $\{w_{i,1}, \ldots, w_{i,k-1}\}$ (depending solely on the choice of $Z_i \in [k-1]$ which also determines S_i). Moreover, the ratio of $w_{i,j}$ and $w_{i,j+1}$ is fixed for any $j \in \{0, \ldots, k-1\}$ and we define this quantity as

$$r_i := 2^{k^{m-i+1}} = \frac{w_{i,j}}{w_{i,j+1}} \quad \text{for any } j \in \{0, \ldots, k-1\}. \tag{14.4}$$

Observation 168. *For any fixed $(s_{<i}, x_{<i})$, the random variables $|\texttt{Win}_{i+1}|, \ldots, |\texttt{Win}_m|$ will be supported on the interval $[2^{m^m} \cdot w_{i,Z_i+1}, w_{i,Z_i}]$.*

Proof. By definition,

$$|\texttt{Win}_{i+1}| = 2^{S_i} = 2^{S_{i-1} - k^{m-i+1} \cdot Z_i} = w_{i,Z_i}.$$

Moreover, by Observation 167, for any $j \in \{i+1, \ldots, m\}$, we have $|\texttt{Win}_j| \le |\texttt{Win}_{i+1}|$. Thus each of these windows can have length at most w_{i,Z_i}, proving the upper bound side.

For the lower bound, for any $j \in \{i+1, \ldots, m\}$, we have,

$$\begin{aligned} |\texttt{Win}_j| \ge |\texttt{Win}_m| = 2^{S_{m-1}} &\ge 2^{S_i - k^{m-i+1} + m^m} \qquad \text{(by part } (ii) \text{ of Observation 167)} \\ &= 2^{m^m} \cdot 2^{S_i} \cdot 2^{-k^{m-i+1}} = 2^{m^m} \cdot w_{i,Z_i} \cdot r_i^{-1} = 2^{m^m} \cdot w_{i,Z_i+1}. \end{aligned}$$

This concludes the proof. ▮

We need one final definition for now:

- For the function $f : [M] \to [m]$, we define the **density** of index $i \in [m]$ in f over a window $\texttt{Win}$, denoted by $\texttt{density}_f(\texttt{Win}, i)$, as

$$\texttt{density}_f(\texttt{Win}, i) := \frac{|\{j \in \texttt{Win} : f(j) = i\}|}{|\texttt{Win}|},$$

  namely, the fraction of entries of the window that are equal to i.

Observation 169. *For any choice of $(s_{<i}, x_{<i})$, we have,*

$$\Pr\left(f(X_i) = i \mid s_{<i}, x_{<i}\right) = \texttt{density}_f(\texttt{Win}_i(s_{<i}, x_{<i}), i).$$

Proof. Conditioned on $(s_{<i}, x_{<i})$, X_i is chosen uniformly at random from $\texttt{Win}_i(s_{<i}, x_{<i})$. The observation therefore follows from the definition of $\texttt{density}_f(\texttt{Win}_i(s_{<i}, x_{<i}), i)$. ▮

14.4.3 Analysis of the Hard Distribution (and Proof of Lemma 166)

We prove Lemma 166 by individually considering each iteration in the distribution.

Lemma 170. *For any iteration $i \in [m]$ and conditioned on any choice of $(s_{<i}, x_{<i})$, <u>at least one</u> of the following two conditions is true:*

$$(i)\ \Pr\left(f(X_i) = i \mid s_{<i}, x_{<i}\right) \leq \frac{101}{m} \quad \text{or}$$
$$(ii)\ \Pr\left(\mathtt{density}_f(\mathtt{Win}_m, i) < \frac{2}{m} \mid s_{<i}, x_{<i}\right) < \frac{1}{k^{1/3}}.$$

The main bulk of this section is to prove Lemma 170. We then show at the end of the section that this lemma easily implies Lemma 166. To continue, we need some definitions.

Definition 171. *The* **window-tree** *of iteration $i \in [m]$ for $(s_{<i}, x_{<i})$, denoted by $\mathcal{T}_i := \mathcal{T}(s_{<i}, x_{<i})$, is the following rooted tree with $k+1$ levels (the root is at level 0):*

- (i) *Every non-leaf node α of the tree has r_i many child-nodes.*
- (ii) *Every node α at a level $\ell \in \{0, \ldots, k\}$ is associated with a window $\mathtt{Win}(\alpha)$ of length $w_{i,\ell}$.*
- (iii) *The root α_r is associated with the window $\mathtt{Win}(\alpha_r) := \mathtt{Win}_i(s_{<i}, x_{<i})$. The windows associated with child-nodes of a node α at level ℓ partition $\mathtt{Win}(\alpha)$ of length $w_{i,\ell}$ into equal-size windows of length $w_{i,\ell+1}$ (recall that α has $r_i = w_{i,\ell}/w_{i,\ell+1}$ child-nodes). Moreover, the left most child-node receives the window in the partition with the smallest starting point, the next child-node on the right receives the next window with smallest part, and so on.*
- (iv) *The* **density** *of a node α with respect to any function $f : [M] \to [m]$ is defined as*
$$\mathtt{density}_f(\alpha) := \mathtt{density}_f(\mathtt{Win}(\alpha), i).$$

One way we use the window-tree in our analysis is to consider the process of sampling X_i (which is uniform over $\mathtt{Win}_i(s_{<i}, x_{<i})$ at this stage) as traversing the window-tree via a root-to-leaf path. This is formalized in the following observation.

Observation 172. *The distribution of X_i conditioned on $(s_{<i}, x_{<i})$ can be alternatively seen as: (i) Sample a root-to-leaf path $\alpha_0, \alpha_1, \ldots, \alpha_k$ where α_0 is the root of $\mathcal{T}_i$ and where each $\alpha_{\ell+1}$ is a child-node of α_ℓ chosen uniformly at random; then, (ii) sample X_i uniformly at random from $\mathtt{Win}(\alpha_k)$. We refer to $\alpha_0, \ldots, \alpha_k$ as the* **sampling path** *of X_i.*

Proof. X_i is distributed uniformly over $\mathtt{Win}_i$ and leaf-nodes of $\mathcal{T}_i$ form an equipartition of $\mathtt{Win}_i$. ∎

In addition, we define a pruning procedure for any window-tree $\mathcal{T}$ as follows.

Definition 173. *Fix a function $f : [M] \to [m]$ and a window-tree $\mathcal{T}_i$ for some $i \in [m]$. We say that a node $\alpha \in \mathcal{T}_i$ is* **sparse** *iff*

$$\mathtt{density}_f(\alpha) \leq \frac{100}{m}.$$

We have the following procedure for pruning $\mathcal{T}_i$: Start from the root down to the leaf-nodes and prune any sparse node of the tree, as well as all of that node's sub-tree. We refer to a sparse node that was pruned on its own (i.e., any node that is sparse and has no sparse ancestors) as a **directly pruned** *node and to other pruned nodes (i.e., nodes with sparse ancestors) as* **indirectly pruned**.

Finally, for $\ell \in \{0, \ldots, k\}$, define p_ℓ as the fraction of directly pruned nodes at level ℓ of the tree over all level-ℓ nodes that are not indirectly pruned.

It is worth noting that pruning is deterministic conditioned on $(s_{<i}, x_{<i})$.

With these definitions, we can now start proving Lemma 170. This will be done by considering some different cases handled by the following claims. The first (and easiest) case is when most nodes of the window-tree are pruned, in which case we achieve property (i) of Lemma 170.

Claim 174 (Case I: "Many Directly Pruned Nodes")**.** *Suppose*

$$\prod_{\ell=0}^{k}(1 - p_\ell) \leq \frac{1}{m}.$$

Then, for any choice of $(s_{<i}, x_{<i})$,

$$\Pr_{X_i}(f(X_i) = i \mid s_{<i}, x_{<i}) \leq \frac{101}{m}.$$

Proof. Let W_{rem} denote the subset of $\mathtt{Win}_i$ that remains after removing windows of all pruned leaf-nodes from $\mathtt{Win}_i$. We have that

$$|W_{\text{rem}}| = \frac{\#\text{ leaf-nodes of } \mathcal{T}_i \text{ that are not pruned}}{\#\text{ leaf-nodes of } \mathcal{T}_i} \cdot |\mathtt{Win}_i| = \prod_{\ell=0}^{k}(1-p_\ell) \cdot |\mathtt{Win}_i| \leq \frac{|\mathtt{Win}_i|}{m},$$

where the second equality is because at each level ℓ of the tree, the number of not pruned nodes drops by a factor of $(1 - p_\ell)$ by the definition of p_ℓ.

Let *DP* denote the set of all nodes in the tree $\mathcal{T}_i$ that were directly pruned. Note that the windows $\mathtt{Win}(\alpha)$ for $\alpha \in \mathit{DP}$ partition $\mathtt{Win}_i \setminus W_{\text{rem}}$. This implies that

$\texttt{density}_f(\texttt{Win}_i, i)$ equals

$$\frac{1}{|\texttt{Win}_i|} \cdot \left(|W_{\text{rem}}| \cdot \texttt{density}_f(W_{\text{rem}}, i) + \sum_{\alpha \in DP} \texttt{density}_f(\alpha) \cdot |\texttt{Win}(\alpha)| \right)$$

(by the definition of $\texttt{density}_f(\cdot)$ function)

$$\leq \frac{1}{|\texttt{Win}_i|} \cdot \left(|W_{\text{rem}}| + \sum_{\alpha \in DP} \frac{100}{m} \cdot |\texttt{Win}(\alpha)| \right)$$

(as $\texttt{density}_f(\alpha) \leq 100/m$ by the definition of sparsity, and $\texttt{density}_f(W_{\text{rem}}, i) \leq 1$)

$$\leq \frac{1}{m} + \frac{100}{m} = \frac{101}{m}.$$

(as $|W_{\text{rem}}| \,/\, |\texttt{Win}_i| \leq 1/m$ as established above, and $\sum_{\alpha \in DP} |\texttt{Win}(\alpha)| \leq |\texttt{Win}_i|$)

By Observation 169, we have,

$$\Pr_{X_i}(f(X_i) = i \mid s_{<i}, x_{<i}) = \texttt{density}_f(\texttt{Win}_i, i) \leq \frac{101}{m},$$

concluding the proof. ∎

We now consider the complementary case, while also taking the randomness of Z_i into account. Recall that Z_i is uniform over $[k-1]$ and that $|\texttt{Win}_{i+1}| = w_{i,Z_i}$. For any fixed realization z_i of Z_i, recall the sampling-path-based process of sampling X_i outlined in Observation 172. Consider the first z_i vertices in this path, namely, $\alpha_0, \ldots, \alpha_{z_i-1}$ that start from the root and end at a level $z_i - 1$ node of $\mathcal{T}_i$.

Define Event $\mathcal{E}(s_{<i}, x_{<i}, z_i, X_i)$ to be the event that none of the nodes in $\alpha_0, \ldots, \alpha_{z_i-1}$ are pruned. Event $\mathcal{E}(s_{<i}, x_{<i}, z_i, X_i)$ depends only on the choice of X_i (to traverse the root-to-leaf path), and is conditioned on $s_{<i}, x_{<i}$ (which determine the window-tree $\mathcal{T}_i$) and z_i (which determines the level of the tree that we focus on). To avoid clutter, when it is clear from the context, we refer to this event simply by $\mathcal{E}_i$.

We partition the remaining cases based on whether or not the event $\mathcal{E}_i$ happens.

Claim 175 (Case II: "A Pruned Node on the Sampling Path")**.** *Fix any choice of z_i and $(s_{<i}, x_{<i})$. In the case that the event $\mathcal{E}_i$ does not happen, we have,*

$$\Pr_{X_i}(f(X_i) = i \mid s_{<i}, x_{<i}, z_i, \overline{\mathcal{E}(s_{<i}, x_{<i}, z_i, X_i)}) \leq \frac{100}{m}.$$

Proof. After conditioning on $(s_{<i}, x_{<i}, z_i)$, the event $\mathcal{E}_i$ is only a function of the sampling process of X_i outlined in Observation 172. Assuming $\mathcal{E}_i$ does not happen, we know that there exists a *unique* node α_j on the path $\alpha_0, \ldots, \alpha_{z_i-1}$ such that α_j is sparse and is directly pruned. By additionally conditioning on the subpath $\alpha_0, \ldots, \alpha_j$, we have that X_i is chosen uniformly at random from $\texttt{Win}(\alpha_j)$ at this point. Thus, if

we define $\mathcal{S}$ to be the event that $(\alpha_1, \ldots, \alpha_j)$ is on the sampling path, we have that

$$\begin{aligned}
&\Pr_{X_i}(f(X_i) = i \mid s_{<i}, x_{<i}, z_i, \overline{\mathcal{E}_i}) \\
&= \sum_{j=0}^{z_i-1} \sum_{\substack{(\alpha_1,\ldots,\alpha_j): \\ \alpha_j \text{ is directly pruned}}} \Pr_{X_i}(f(X_i) = i \wedge \mathcal{S} \mid s_{<i}, x_{<i}, z_i, \overline{\mathcal{E}_i}) \\
&\qquad \text{(as these subpaths partition all possible choices for } \mathcal{E}_i \text{ to not happen)} \\
&= \sum_{j=0}^{z_i-1} \sum_{\substack{(\alpha_1,\ldots,\alpha_j): \\ \alpha_j \text{ is directly pruned}}} \Pr_{X_i}(\mathcal{S} \mid s_{<i}, x_{<i}, z_i, \overline{\mathcal{E}_i}) \cdot \frac{|\{t \in \mathtt{Win}(\alpha_j) : f(t) = i\}|}{|\mathtt{Win}(\alpha_j)|} \\
&\qquad \text{(as } X_i \text{ is chosen uniformly from } \mathtt{Win}(\alpha_j) \text{ under these conditions)} \\
&= \sum_{j=0}^{z_i-1} \sum_{\substack{(\alpha_1,\ldots,\alpha_j): \\ \alpha_j \text{ is directly pruned}}} \Pr_{X_i}(\mathcal{S} \mid s_{<i}, x_{<i}, z_i, \overline{\mathcal{E}_i}) \cdot \mathtt{density}_f(\mathtt{Win}(\alpha_j), i) \\
&\qquad \text{(by the definition of } \mathtt{density}_f) \\
&\leq \sum_{j=0}^{z_i-1} \sum_{\substack{(\alpha_1,\ldots,\alpha_j): \\ \alpha_j \text{ is directly pruned}}} \Pr_{X_i}(\mathcal{S} \mid s_{<i}, x_{<i}, z_i, \overline{\mathcal{E}_i}) \cdot \frac{100}{m}. \\
&\qquad \text{(as } \alpha_j \text{ needs to be sparse to be directly pruned)}
\end{aligned}$$

This can now be further upper bounded by $100/m$ as the probability terms are summing over all disjoint events that can lead to $\overline{\mathcal{E}_i}$ (conditioned on this event) and thus add up to one. ∎

Finally, we have the following case which handles the situation when $\mathcal{E}_i$ happens. The following claim is the heart of the proof.

Claim 176 (Case III: "No Pruned Nodes on the Sampling Path")**.** *Fix any choice of z_i and $(s_{<i}, x_{<i})$. In the case that the event $\mathcal{E}_i$ happens, we have,*

$$\Pr_{X_i}\left(\mathtt{density}_f(\mathtt{Win}_m, i) < \frac{2}{m} \mid s_{<i}, x_{<i}, z_i, \mathcal{E}(z_i, X_i)\right) < 4 \cdot \left(p_{z_i} + p_{z_i+1} + \frac{m}{r_i}\right).$$

Proof. Throughout this proof, we always condition on $s_{<i}, x_{<i}, z_i, \mathcal{E}(z_i, X_i)$ and thus may not mention this explicitly in the probability terms. Let us first list the information we have so far:

- The node α_{z_i-1} on the sampling path is not pruned as we conditioned on the event $\mathcal{E}(z_i, X_i)$ (we emphasize that α_{z_i-1} is a random variable and is not fixed yet just by these conditions).

- Window $\mathtt{Win}_m$ is going to have size at least $2^{m^m} \cdot w_{i,z_i+1}$ and at most w_{i,z_i} by Observation 168.

- By Observation 167,

  $$X_m \leq X_i + (m-i) \cdot 2^{S_i} = X_i + (m-i) \cdot w_{i,z_i}. \text{ (by the definition of } w_{i,z_i} = 2^{S_i}\text{)}$$

- $\mathtt{Win}_m$ starts at X_m and ends at $X_m + |\mathtt{Win}_m|$. We can think of the process of sampling $\mathtt{Win}_m$ as first sampling its length $|\mathtt{Win}_m|$, then sampling the **offset** $O_{i,m} := X_m - X_i = \sum_{j=i+1}^{m} Y_j$ conditioned on $|\mathtt{Win}_m|$, and then sampling X_i conditioned on $O_{i,m}$, and $|\mathtt{Win}_m|$.

- We further have that X_i conditioned on $O_{i,m}$ and $|\mathtt{Win}_m|$ is still uniform over $\mathtt{Win}(\alpha_{z_i-1})$. This is because $|\mathtt{Win}_m|$ is only a function of $Z_{i+1}, \ldots, Z_m$, and $X_m - X_i$ is only a function of $Y_{i+1}, \ldots, Y_m$, while X_i is only a function of Y_i; finally, Y_i is independent of $Y_{i+1}, \ldots, Y_m$ and $Z_{i+1}, \ldots, Z_m$ and is chosen uniformly from $[2^{s_{i-1}}]$.

In the following, we condition on any fixed choice of offset $o_{i,m}$ for $O_{i,m}$ and on $|\mathtt{Win}_m|$. We have already established that

$$2^{m^m} \cdot w_{i,z_i+1} \leq |\mathtt{Win}_m| \leq w_{i,z_i} \qquad \text{and} \qquad o_{i,m} \leq (m-i) \cdot w_{i,z_i}. \tag{14.5}$$

Moreover, the distribution of $\mathtt{Win}_m$ conditioned on $o_{i,m}, |\mathtt{Win}_m|$ (and $s_{<i}, x_{<i}, z_i, \mathcal{E}_i$ that we always condition on in this proof), is $X_i + o_{i,m}$ for X_i chosen randomly from $\mathtt{Win}(\alpha_{z_i-1})$. Moreover, given that $o_{i,m} \leq (m-i) \cdot w_{i,z_i}$ while $|\mathtt{Win}(\alpha_{z_i-1})| = w_{i,z_i-1} = r_i \cdot w_{i,z_i}$ and $r_i = 2^{k^{m-i+1}} \geq 2^k$ as $i \leq m$, the distribution of X_i and $X_i + o_{i,m}$ are quite close to each other modulo a negligible factor. Thus, for intuition, we can think of X_i itself as the distribution of starting point for $\mathtt{Win}_m$ in this context (although we will of course take this difference into account explicitly in the proof). We now use this information to prove the claim. To simplify the exposition, we are going to separate the analysis based on level z_i and level z_{i+1} of the window-tree.

Analysis on level z_i of the window-tree. Firstly, since $|\mathtt{Win}_m| \leq w_{i,z_i}$, and each node at level z_i of the window-tree $\mathcal{T}_i$ has a window of length w_{i,z_i}, we get that $\mathtt{Win}_m$ intersects with windows of at most two *consecutive* nodes at level z_i of $\mathcal{T}_i$, which are solely determined by the choice of X_i. We use $\beta_1(X_i)$ and $\beta_2(X_i)$ to denote these two nodes with β_1 being the one where the starting point of $\mathtt{Win}_m$, namely, $X_i + o_{i,m}$, lies in, and $\beta_2(X_i)$ being the one containing the endpoint $X_i + o_{i,m} + |\mathtt{Win}_m|$ (note that it is possible that $\beta_2 = \beta_1$).

We prove that with high probability, neither of these nodes are pruned. Let us focus on $\beta_1(X_i)$ first (the analysis is almost identical for $\beta_2(X_i)$ and we can then apply the union bound). For any $\ell \in \{0, \ldots, k-1\}$, let $P(\ell)$ (resp. $DP(\ell)$) denote the set of pruned (resp. directly pruned) nodes at level ℓ of $\mathcal{T}_i$; similarly, for a node $\alpha \in \mathcal{T}_i$, let $P(\alpha)$ (resp. $DP(\alpha)$) denote the set of child-nodes of α that are pruned (resp. directly pruned). For any fixed choice of α_{z_i-1} on the sampling path of X_i,

$$\Pr_{X_i}\left(\beta_1(X_i) \text{ is pruned} \mid \alpha_{z_i-1}\right) \tag{14.6}$$

$$= \sum_{\beta\in P(z_i)} \Pr_{X_i}\left(\beta_1(X_i) = \beta \mid \alpha_{z_{i-1}}\right)$$

(as β_1 is in level z_i and $P(z_i)$ is the set of all pruned nodes of this level)

$$= \sum_{\beta\in P(\alpha_{z_i-1})} \Pr_{X_i}\left(\beta_1(X_i) = \beta \mid \alpha_{z_{i-1}}\right) + \sum_{\substack{\beta\in \\ P(z_i)\setminus P(\alpha_{z_i-1})}} \Pr_{X_i}\left(\beta_1(X_i) = \beta \mid \alpha_{z_{i-1}}\right) \tag{14.7}$$

$$\leq |P(\alpha_{z_i-1})| \cdot \frac{1}{r_i} + (m-i)\cdot\frac{1}{r_i}, \tag{14.8}$$

where the last inequality holds because of the following reasoning. Firstly, the probability that $\beta_1(X_i)$ is equal to any fixed node β at level z_i is at most $1/r_i$. This is because

$$\Pr\left(\beta_1(X_i) = \beta \mid \alpha_{z_{i-1}}\right) = \Pr\left(X_i + o_{i,m} \in \mathtt{Win}(\beta) \mid \alpha_{z_i-1}\right) \leq \frac{|\mathtt{Win}(\beta)|}{|\mathtt{Win}(\alpha_{z_i-1})|} = \frac{1}{r_i},$$

because X_i is chosen uniformly from $\mathtt{Win}(\alpha_{z_i-1})$, and $|\mathtt{Win}(\beta)| = |\mathtt{Win}(\alpha_{z_i-1})|/r_i$ as β is at level z_i. This immediately implies the first term in the RHS of Equation (14.8). For the second term, for $\beta_1(X)$ to intersect with a node β not in the subtree of α_{z_i-1}, we need to have $X_i + o_{i,m} \notin \mathtt{Win}(\alpha_{z_i-1})$, while we know $X_i \in \mathtt{Win}(\alpha_{z_i-1})$. As $o_{i,m} \leq (m-i)\cdot w_{i,z_i}$ by Equation (14.5), and any node at level z_i has a window of length w_{i,z_i}, we get that there are most $(m-i)$ choices of β outside child-nodes of α_{z_i-1} that can also become $\beta_1(X_i)$. The second part of RHS in Equation (14.8) now follows from this and the upper bound of $1/r_i$ on the probability of each node.

Finally, by taking the expectation over the choice of α_{z_i-1},

$$\Pr_{X_i}\left(\beta_1(X_i) \text{ is pruned}\right) = \mathop{\mathbb{E}}_{\alpha_{z_i-1}}\left[\Pr_{X_i}\left(\beta_1(X_i) \text{ is pruned} \mid \alpha_{z_i-1}\right)\right]$$

(by the law of total probability, over the choice of α_{z_i-1} in the sampling path)

$$\leq \mathop{\mathbb{E}}_{\alpha_{z_i-1}}\left[\frac{|P(\alpha_{z_i-1})|}{r_i}\right] + \frac{(m-i)}{r_i} \qquad \text{(by Equation (14.8))}$$

$$= p_{z_i} + \frac{(m-i)}{r_i},$$

where in the final equality, we used the fact that α_{z_i-1} is chosen from non-pruned nodes (by conditioning on $\mathcal{E}_i$), and thus $|P(\alpha_{z_i-1})|/r_i$ is the fraction of pruned nodes over all *not* indirectly pruned at level z_i, which by definition is p_{z_i}.

Doing the same exact analysis, we can bound the probability that $\beta_2(X_i)$ is pruned also as

$$\Pr_{X_i}\left(\beta_2(X_i) \text{ is pruned}\right) \leq p_{z_i} + \frac{(m-i)+1}{r_i},$$

where the $+1$ term in the RHS compared to the one for β_1 comes from the fact that $\beta_2(X_i)$ can have $(m-i+1)$ choices outside subtree of α_{z_i-1} (because we are now considering $X_i + o_{i,m} + |\mathtt{Win}_m| \leq X_i + (m-i+1)\cdot w_{i,z_i}$ instead). By the union bound on the probabilities for $\beta_1(X_i)$ and $\beta_2(X_i)$,

$$\Pr_{X_i}\left(\text{either of } \beta_1(X_i) \text{ or } \beta_2(X_i) \text{ is pruned}\right) \leq 2\cdot p_{z_i} + 2\cdot\frac{m}{r_i}. \tag{14.9}$$

Analysis on level z_i+1 of the window-tree. For the next step, let $\gamma_1(X_i),\ldots,\gamma_t(X_i)$ denote the child-nodes of $\beta_1(X_i)$ and $\beta_2(X_i)$ such that $\mathtt{Win}(\gamma_j(X_i))$ is *entirely* contained in $\mathtt{Win}_m$. Again, the choice of $\gamma_1,\ldots,\gamma_t$ is only a function of X_i. Moreover, since $|\mathtt{Win}_m| \geq 2^{m^m}\cdot w_{i,z_i+1}$ by Equation (14.5), while the window of each node at level z_i+1 is of size w_{i,z_i+1}, we have that $t \geq 2^{m^m}-2$ always. We now bound the probability that each γ_j is (directly) pruned, for $j\in[t]$. This part of the analysis is quite similar to that of level z_i with only minor changes.

For any choice of $\beta_1(X_i)$ and $\beta_2(X_i)$,

$$\begin{aligned}\Pr_{X_i}\left(\gamma_j(X_i) \text{ is directly pruned} \mid \beta_1,\beta_2\right) &= \sum_{\substack{\gamma\in\\ DP(\beta_1)\cup DP(\beta_2)}} \Pr_{X_i}\left(\gamma_j(X_i)=\gamma \mid \beta_1,\beta_2\right)\\ &\quad\text{(as } \mathtt{Win}_m\subseteq\mathtt{Win}(\beta_1)\cup\mathtt{Win}(\beta_2)\text{, so } \gamma_j \text{ has must use child-nodes of } \beta_1 \text{ or } \beta_2)\\ &\leq \Big(|DP(\beta_1)|+|DP(\beta_2)|\Big)\cdot\frac{1}{r_i},\end{aligned} \tag{14.10}$$

where we are again going to argue that the probability that $\gamma_j(X_i)$ is equal to any fixed node γ is at most $1/r_i$ conditioned on the choice of β_1 and β_2. For $\gamma_j(X_i)$ to be equal to a node γ we need to have that $X_i + o_{i,m} + (j-1)\cdot w_{i,z_i+1}\in\mathtt{Win}(\gamma)$; this is because $\gamma_j(X_i)$ appears after $(j-1)$ nodes of level z_i+1 that are fully inside $\mathtt{Win}_m$ and each such window has length w_{i,z_i+1} (note that this is a necessary but not a sufficient condition). Thus,

$$\begin{aligned}&\Pr_{X_i}\left(\gamma_j(X_i)=\gamma \mid \beta_1,\beta_2\right) \leq \Pr_{X_i}\left(X_i+o_{i,m}+(j-1)\cdot w_{i,z_i+1}\in\mathtt{Win}(\gamma) \mid \beta_1,\beta_2\right)\\ &\leq \frac{|\mathtt{Win}(\gamma)|}{w_{i,z_i}} = \frac{1}{r_i},\end{aligned}$$

where the last inequality is because conditioned on $\mathtt{Win}_m$ intersecting with β_1,β_2, X_i is chosen uniformly at random from a window of length w_{i,z_i} (equal to length of $\mathtt{Win}(\beta_1)$ and $\mathtt{Win}(\beta_2)$); the final equality also uses that $|\mathtt{Win}(\gamma)| = w_{i,z_i+1} = w_{i,z_i}/r_i$. Hence (14.10).

We can now deduce that

$$\begin{aligned}
&\mathop{\mathbb{E}}_{X_i}[\#\text{ of }\gamma_1(X_i),\dots,\gamma_t(X_i)\text{ that are directly pruned}]\\
&\quad=\mathop{\mathbb{E}}_{\beta_1,\beta_2}\mathop{\mathbb{E}}_{X_i}[\#\text{ of }\gamma_1(X_i),\dots,\gamma_t(X_i)\text{ that are directly pruned}\mid\beta_1,\beta_2]\\
&\qquad\qquad\qquad\text{(by the law of total probability over the choices of }\beta_1,\beta_2)\\
&\quad=\mathop{\mathbb{E}}_{\beta_1,\beta_2}\left[|DP(\beta_1)|+|DP(\beta_2)|\cdot\frac{t}{r_i}\right], \qquad (14.11)
\end{aligned}$$

where the last inequality is by Equation (14.10).

Let $\overline{P}(z_i)$ denote the set of not pruned nodes in level z_i and let $\hat{P}(z_i)$ denote the set of nodes in level z_i whose parents are not pruned. Since we are conditioning on $\mathcal{E}_i$, we know that X_i is uniformly random from the interval $\cup_{\beta\in\hat{P}(z_i)}\texttt{Win}(\beta)$. It follows that $X_m = X_i + o_{i,m}$ is uniformly random in a range whose size is also $\ell=\sum_{\beta\in\hat{P}(z_i)}|\texttt{Win}(\beta)|$. Thus, for any level-$z_i$ node β, we have that

$$\Pr[\beta_1=\beta]=\Pr[X_m\in\texttt{Win}(\beta)]\le\frac{|\texttt{Win}(\beta)|}{\ell}=\frac{w_{i,z_i}}{\ell}=\frac{1}{|\hat{P}(z_i)|}.$$

Summing over the level-(z_i+1) nodes that are directly pruned, we have that

$$\mathbb{E}|DP(\beta_1)|=\sum_{\gamma\in DP(z_i+1)}\Pr[\beta_1\text{ is the parent of }\gamma]\le\frac{|DP(z_i+1)|}{|\hat{P}(z_i)|}\le\frac{|DP(z_i+1)|}{|\overline{P}(z_i)|},$$

using the upper bound established above on the probability that β_1 is any fixed node. Note that

$$p_{z_i+1}=\frac{|DP(z_i+1)|}{r_i\cdot|\overline{P}(z_i)|},$$

i.e., the number of directly pruned nodes in level z_i+1 divided by the number of nodes with not pruned parents. Therefore, $\mathbb{E}\,|DP(\beta_1)|\le r_i\cdot p_{z_i+1}$. By the same reasoning (but applied to β_2, which contains the endpoint of X_m), we have that $\mathbb{E}\,|DP(\beta_2)|\le r_i\cdot p_{z_i+1}$.

Thus, we can use Equation (14.11) to conclude that

$$\mathop{\mathbb{E}}_{X_i}[\#\text{ of }\gamma_1(X_i),\dots,\gamma_t(X_i)\text{ that are directly pruned}]\le 2p_{z_i+1}\cdot t.$$

By Markov's inequality,

$$\Pr_{X_i}(\text{more than }t/2\text{ of }\gamma_1(X_i),\dots,\gamma_t(X_i)\text{ are directly pruned})\le 4\cdot p_{z_i+1}. \qquad (14.12)$$

Finally, by considering the possibility that at least one of β_1 or β_2 could be pruned also we have,

$$\begin{aligned}
&\Pr_{X_i}(\text{more than } t/2 \text{ of } \gamma_1, \ldots, \gamma_t \text{ are pruned}) \\
&\qquad \leq \Pr_{X_i}(\text{more than } t/2 \text{ of } \gamma_1(X_i), \ldots, \gamma_t(X_i) \text{ are directly pruned}) \\
&\qquad\qquad + \Pr(\text{either of } \beta_1 \text{ or } \beta_2 \text{ are pruned}) \\
&\qquad \leq 4p_{z_i+1} + 2p_{z_i} + \frac{2m}{r_i}, \qquad (14.13)
\end{aligned}$$

by Equation (14.9) and Equation (14.12).

Concluding the proof. Let us now condition on the event that at least $t/2$ of nodes $\gamma_1, \ldots, \gamma_t$ are *not* pruned, namely, the complement of the event in Equation (14.13). Given that $\texttt{Win}_m$ can have intersection with at most two other level-(z_i+1) nodes beside $\gamma_1, \ldots, \gamma_t$, conditioned on the above event, we have,

$$\texttt{density}_f(\texttt{Win}_m, i) \geq \frac{(t/2) \cdot 100/m}{t+2} \geq \frac{100}{3m} > \frac{2}{m},$$

as $t \geq 2^{m^m} - 2 \gg 1$. Thus, by Equation (14.13), we have,

$$\Pr_{X_i}\left(\texttt{density}_f(\texttt{Win}_m, i) \leq \frac{2}{m}\right) \leq 2p_{z_i} + 4 \cdot p_{z_i+1} + \frac{2m}{r_i} < 4\left(p_{z_i} + p_{z_i+1} + \frac{m}{r_i}\right),$$

concluding the proof. ∎

Claims 174, 175, and 176 now cover all possible cases and allow us to prove Lemma 170.

Proof of Lemma 170. Fix the tree $\mathcal{T}_i$ and consider its pruning process. If $\prod_{\ell=0}^{k}(1-p_\ell) \geq 1/m$, we achieve the first condition of the lemma by Claim 174 and are thus done. Now consider the complement case. In this case, we have,

$$\frac{1}{m} < \prod_{\ell=0}^{k}(1-p_\ell) \leq \exp\left(-\sum_{\ell=0}^{k} p_\ell\right),$$

which implies that $\sum_{\ell=0}^{k} p_\ell \leq \ln m$. Recall that the choice of Z_i in the distribution is uniform over $[k-1]$ regardless of conditioning on $(s_{<i}, x_{<i})$. Thus, we have,

$$\mathbb{E}_{Z_i}[p_{Z_i} + p_{Z_i+1}] \leq \frac{1}{k-1} \cdot \sum_{\ell=1}^{k-1} p_\ell + \frac{1}{k-1} \cdot \sum_{\ell=2}^{k} p_\ell \leq \frac{2}{k-1} \sum_{\ell=0}^{k} p_\ell \leq \frac{2\ln m}{(k-1)}.$$

By Markov's inequality, we have,

$$\Pr_{Z_i}\left(p_{Z_i} + p_{Z_i+1} \geq \frac{4 \cdot \ln m}{k^{1/2}}\right) \ll \frac{1}{k^{1/3}}.$$

We can now condition on any choice z_i of Z_i such that $p_{z_i} + p_{z_i+1} \leq (4 \ln m)/k^{1/2}$. At this point, either event $\mathcal{E}(z_i)$ does not happen, in which case, by Claim 175, we again obtain condition (i) of the lemma; or the event $\mathcal{E}(z_i)$ happens, which by Claim 176 and the choice of r_i in Equation (14.4) implies

$$\Pr_{X_i}\left(\texttt{density}_f(\texttt{Win}_m, i) \leq \frac{2}{m} \mid s_{<i}, x_{<i}\right) \leq 4 \cdot \left(\frac{4 \cdot \ln m}{k^{1/2}} + \frac{m}{2^{k^{m-i}}}\right) \ll \frac{1}{k^{1/3}},$$

as $i \leq m-1$ and thus $m/2^{k^{m-i}} \leq m/2^k \ll 1/k^{1/3}$, as $k = m^m$. Taking the union bound over the above two events, we also obtain condition (ii) of the lemma. ∎

Finally, we use this lemma to conclude the proof of Lemma 166.

Proof of Lemma 166. Let $T_1, T_2 \subseteq [m]$ denote, respectively, the iterations in which condition (i) or condition (ii) of Lemma 170 happens. Note that T_1 and T_2 are random variables over the randomness of S_i's and X_i's. We first claim that with high probability $|T_2| < m/2$. This is because for any iteration $i \in T_2$ and any choice of $(s_{<i}, x_{<i})$ of prior iterations, by Lemma 170,

$$\Pr_{X_i}\left(\texttt{density}_f(\texttt{Win}_m, i) \leq \frac{2}{m} \mid s_{<i}, x_{<i}\right) \leq \frac{1}{k^{1/3}}.$$

A union bound on at most m choices for indices on T_2 then implies that with probability at least $1 - m/k^{1/3}$, we have $\texttt{density}_f(\texttt{Win}_m, i) > \frac{2}{m}$ for all $i \in T_2$. But then conditioned on this event, the size of T_2 cannot be $m/2$ or larger as otherwise $\texttt{Win}_m$ contains $m/2$ disjoint sets each of which contains than a $2/m$ fraction of the window, which is a contradiction. Thus,

$$\Pr(|T_2| \geq m/2) \leq \frac{m}{k^{1/3}} \ll \frac{1}{k^{1/4}}. \qquad \text{(as } k = m^m\text{)}$$

We condition on the complement of this event in the following, namely, that $|T_2| < m/2$. Let $\{i_1, \ldots, i_{m/2}\}$ denote the first $m/2$ indices of T_1 which by the conditioning on the size of T_2 is well defined. We have,

$$\begin{aligned}
&\Pr(\text{for all } j \in [m/2]\text{: } f(X_{i_j}) = i_j) \\
&= \prod_{j \in [m/2]} \Pr\left(f(X_{i_j}) = i_j \mid f(X_{i_1}) = i_1, \ldots, f(X_{i_{j-1}}) = i_{j-1}\right) \\
&\leq \left(\frac{101}{m}\right)^{m/2}.
\end{aligned}$$

(since these are type (i) iterations and we can apply condition (i) of Lemma 170)

Putting these two together, combined with the value of $k = m^m$, implies that,

$$\Pr_{(X_1,\ldots,X_m)}(\forall i \in [m] : f(X_i) = i) \leq \frac{1}{k^{1/4}} + \left(\frac{101}{m}\right)^{m/2} \leq m^{-\eta \cdot m},$$

for some constant $\eta > 0$ (taking $\eta = 1/100$ certainly suffices). This concludes the proof. $\blacksquare$

Appendices

14.A Proofs of Standard Results in Fractional Coloring

We prove Propositions 160 and 161 here for completeness. These proofs are standard; see, e.g. [321]. We start by presenting the dual view of fractional colorings that is the key to these proofs.

The dual view of fractional colorings. Given that $\chi_f(G)$ is defined as a solution to an LP, we can use duality to also express $\chi_f(G)$ via the following LP:

$$\chi_f(G) := \max_{y \in \mathbb{R}_{\geq 0}^{V(G)}} \sum_{v \in G} y_v \quad \text{subject to} \quad \sum_{v \in I} y_v \leq 1 \quad \forall I \in \mathcal{I}(G). \tag{14.14}$$

This LP is a fractional relaxation of the *clique number* of G, namely, the size of the largest clique in G (since, in any integral solution to this LP, the y-values that are 1 must be on the vertices of a clique). Interestingly, although the chromatic number and clique size are not duals, their relaxations are.

Proposition (Restatement of Proposition 160)**.** *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be arbitrary graphs. Define $G_1 \vee G_2$ as a graph on vertices $V_1 \times V_2$ and define an edge between vertices (v_1, v_2) and (w_1, w_2) whenever (v_1, w_1) is an edge in G_1 or (v_2, w_2) is an edge in G_2. Then, $\chi_f(G_1 \vee G_2) = \chi_f(G_1) \cdot \chi_f(G_2)$.*

Proof of Proposition 160. We first prove that

$$\chi_f(G_1 \vee G_2) \geq \chi_f(G_1) \cdot \chi_f(G_2). \tag{14.15}$$

Let $y^1 \in \mathbb{R}^{V_1}$ and $y^2 \in \mathbb{R}^{V_2}$ be optimal solutions to the dual LP given by Equation (14.14) for G_1 and G_2, respectively. Consider the assignment $y \in \mathbb{R}^{V_1 \times V_2}$ where $y_{u_1,u_2} = y^1_{u_1} \cdot y^2_{u_2}$. We clearly have that

$$\sum_{(u_1,u_2) \in V_1 \times V_2} y_{u_1,u_2} = \left(\sum_{u_1 \in V_1} y^1_{u_1} \right) \cdot \left(\sum_{u_2 \in V_2} y^2_{u_2} \right) = \chi_f(G_1) \cdot \chi_f(G_2).$$

We now argue that y is also a valid solution to the dual LP given by Equation (14.14) for $G_1 \vee G_2$. Fix any independent set $I \in \mathcal{I}(G_1 \wedge G_2)$. By the definition of the product, we know that I can be written as a product set, namely, $I = I_1 \times I_2$ for $I_1 \in \mathcal{I}(G_1)$ and $I_2 \in \mathcal{I}(G_2)$. Thus,

$$\sum_{(u_1,u_2) \in I} y_{u_1,u_2} = \left(\sum_{u_1 \in I_1} y^1_{u_1} \right) \cdot \left(\sum_{u_2 \in I_2} y^2_{u_2} \right) \leq 1 \cdot 1 = 1,$$

where the inequality is by the constraint of dual LP for y^1 and y^2 each. Thus, y is a solution to the dual LP for $G_1 \vee G_2$, proving Equation (14.15).

We now prove that

$$\chi_f(G_1 \vee G_2) \leq \chi_f(G_1) \cdot \chi_f(G_2), \tag{14.16}$$

using the primal LP instead. Let $x^1 \in \mathbb{R}^{\mathcal{I}(G_1)}$ and $x^2 \in \mathbb{R}^{\mathcal{I}(G_2)}$ be optimal solutions to primal LP from Equation (14.1) for G_1 and G_2, respectively. Consider the assignment $x \in \mathbb{R}^{\mathcal{I}(G_1 \vee G_2)}$ where $x_I = x^1_{I_1} \cdot x^2_{I_2}$, using the fact from the previous part that $I = I_1 \times I_2$ for $I_1 \in \mathcal{I}(G_1)$ and $I_2 \in \mathcal{I}(G_2)$.

We again clearly have that

$$\sum_{(u_1,u_2) \in \mathcal{I}(G_1 \vee G_2)} x_I = \left(\sum_{I_1 \in \mathcal{I}(G_1)} x^1_{I_1} \right) \cdot \left(\sum_{I_2 \in \mathcal{I}(G_2)} x^2_{I_2} \right) = \chi_f(G_1) \cdot \chi_f(G_2),$$

so it remains to prove that x is a valid solution to the primal LP from Equation (14.1) for $G_1 \vee G_2$. Fix any vertex $(u_1, u_2) \in V_1 \times V_2$ and consider all independent sets $I_1 \subseteq V_1$ that contain u_1 and $I_2 \subseteq V_2$ that contain u_2. Then, $I_1 \times I_2$ is also an independent set in $G_1 \vee G_2$ that contains (u_1, u_2). Thus,

$$\sum_{I \in \mathcal{I}(G_1 \vee G_2) : (u_1,u_2) \in I} x_I \geq \left(\sum_{I_1 \in \mathcal{I}(G_1,u_1)} x^1_{I_1} \right) \cdot \left(\sum_{I_2 \in \mathcal{I}(G_2,u_2)} x^2_{I_2} \right) \geq 1 \cdot 1 = 1,$$

where the inequality is by the constraint of primal LP from Equation (14.1) for x^1 and x^2 each. Thus, x is a solution to the primal LP from Equation (14.1) for $G_1 \vee G_2$, proving Equation (14.16). ∎

Proposition (Restatement of Proposition 161)**.** *For any graph $G = (V, E)$,*

$$\chi_f(G) = \max_{\text{distribution } \mu \text{ on } V} \; \min_{I \in \mathcal{I}(G)} \left(\Pr_{v \sim \mu} (v \in I) \right)^{-1}.$$

Proof of Proposition 161. Let μ be any distribution on $V(G)$ and define $b := \max_{I \in \mathcal{I}(G)} \Pr(v \in I)^{-1}$. Create $y \in \mathbb{R}^{V(m)}$ such that $y_v = b \cdot \mu(v)$ for every vertex $v \in V(m)$ where $\mu(v)$ is the probability of vertex v under the distribution μ. We claim that y is a feasible dual solution in Equation (14.14).

For every independent set $I \in \mathcal{I}(G)$,

$$\sum_{v \in I} y_v = b \cdot \sum_{v \in I} \mu(v) = b \cdot \Pr_{v \sim \mu} (v \in I) \leq 1,$$

by the definition of b. Thus y is a feasible dual solution. Moreover,

$$\sum_{v \in V(G)} y_v = b \cdot \sum_{v \in V(G)} \mu(v) = b.$$

As the dual LP in Equation (14.14) is a maximization LP, we have that $\chi_f(G) \geq b = \max_{I \in \mathcal{I}(G)} \Pr_\mu(v \in I)^{-1}$, for any distribution μ on the vertices.

Conversely, let y be any optimal solution to the dual LP and let $c := \sum_{v \in V} y_v$. Define a distribution μ on the vertices V by setting $\mu(v) = y_v/c$. For any independent set $I \in \mathcal{I}(G)$, we have,

$$\Pr_{v \in \mu}(v \in I) = \sum_{v \in I} \mu(v) = \sum_{v \in I} y_v/c \leq 1/c,$$

where the final inequality is because y is a feasible dual solution. Thus, there exists a distribution μ such that $\chi_f(G) = c \leq \max_{I \in \mathcal{I}(G)} \Pr_\mu(v \in I)^{-1}$.

Combining these two parts concludes the proof. ∎

14.B Covering The Full Range of the Universe Size

We now generalize the proof of Theorem 162 to the full parameter range specified in the theorem. Consider u and n satisfying

$$n 2^{2^{\sqrt{\log\log n}}} \leq u \leq 2^{n^{n^2+n}}.$$

Notice that, on the lower-bound side, we are actually covering a slightly larger range (and therefore proving a slightly stronger result) than required to establish Theorem 162.

Set

$$m = (\log\log u)^{1/6} \quad \text{and} \quad k = n/m = n/(\log\log u)^{1/6}.$$

Note that the setting of k implicitly requires that $(\log\log u)^{1/6} \leq n$, which follows from the fact that $(\log\log u)^{1/6} \leq (n^2+n)^{1/6} \leq \sqrt{n}$.

The k-fold conflict graph $G^{\oplus k}(m)$ has $\log \chi_f(G^{\oplus k}(m)) = \Omega(n \log m) = \Omega(n \log\log\log u)$ as already argued in Section 14.3.2. To complete the proof, we must establish that the graph $G^{\oplus k}(m)$ has vertices that are subsets of a universe whose size u' satisfies $u' \leq u$. Solving for u', we have that

$$u' = kM = \frac{n}{(\log\log u)^{1/6}} \cdot 2^{m^{m^2+m}} \leq n \cdot 2^{2^{m^3/2}} \leq n \cdot 2^{2^{\sqrt{\log\log u}/2}}.$$

From here, we can consider two cases. Case 1 is that $u \geq n^2$. In this case, we have

$$u \geq n^2 \geq n \cdot 2^{2^{\sqrt{\log\log n}}} \geq n \cdot 2^{2^{\sqrt{\log\log u}/2}} \geq u'.$$

Case 2 is that $u < n^2$. In this case, we can use the fact that $2^{2^{\sqrt{\log\log u}/2}} \leq 2^{\log u/2} \leq \sqrt{u}$ to conclude that $u' \leq n \cdot 2^{2^{\sqrt{\log\log u}/2}} \leq n\sqrt{u} \leq u$, where the final inequality follows from the case that we are in. In either case, we have that

$$\frac{u}{u'} \geq 1,$$

which completes the proof of Theorem 162 for any choice of u between $n \cdot 2^{2^{\sqrt{\log\log n}}}$ and $2^{n^{n^2+n}}$.

Finally, we remark that the term $2^{n^{n^2+n}}$ in the upper bound is not tight and can be replaced by any other $2^{2^{\mathrm{poly}(n)}}$ term; this is simply because for any $u = 2^{2^{\mathrm{poly}(n)}}$, $\log\log\log u = \Theta(\log n)$ and thus for any larger universe size u also, we can simply focus on the smallest $2^{n^{n^2+n}}$ numbers in the universe and still obtain the same asymptotic lower bound. The lower bound term is also not tight and can be replaced with $n \cdot 2^{2^{(\log\log n)^{\varepsilon}}}$ for any constant $\varepsilon \in (0, 1/2)$ by the same argument.

Part V

How Many Bits Does It Take to Write Down a Pointer?

Chapter 15

Introduction

How many bits does it take to store a pointer? If we know nothing about the pointer except that it references an element in an array of size n, then there is a lower bound of $\log n$ bits.

For many (and perhaps even most) uses of pointers, however, this information-theoretic lower bound does not apply. As we shall see in this part of the thesis, even a small amount of prior information about a pointer (e.g., a node's predecessor in a linked list) can be used to defeat the $\log n$ lower bound.

In this part of the thesis, we introduce a general-purpose tool, which we call the ***tiny pointer***, for compressing pointers. In settings where pointers are used, tiny pointers can often be used instead to eliminate almost all of the space overhead of pointers.

What is a tiny pointer? Suppose n or more users (i.e., Alice, Bob, etc.) are sharing an array A of size n. A user can request a location in A via a function $\textsc{Allocate}()$, which returns a pointer p to a location that is now reserved exclusively for that user, if there is an available location; the user can later relinquish the memory location by calling a function $\textsc{Free}(p)$. Each user promises only to allocate at most one memory location at a time.[1] For example, if Alice calls $\textsc{Allocate}()$ to get a pointer p, she must call $\textsc{Free}(p)$ before calling $\textsc{Allocate}()$ again.

How large do the pointers p need to be? The natural answer is that each pointer uses $\log n$ bits. However, the fact that each pointer has a distinct owner makes it possible to compress the pointers to $o(\log n)$ bits. A critical insight is that the same pointer p can mean different things to different users, via the following scheme. A user k can call $\textsc{Allocate}(k)$ in order to get a tiny pointer p; they can dereference the tiny pointer p by computing a function $\textsc{Dereference}(k, p)$ whose value depends only on k, p, and random bits; and they can free a tiny pointer p by calling a function $\textsc{Free}(k, p)$.

[1] A user k can request more than one location by creating a unique label ℓ for each of their allocations. In this case, we simply treat the "user" for the allocation as the concatenation $k \circ \ell$, so the user k can have multiple allocations without violating the uniqueness requirement.

The reason that tiny pointers are not constrained by the information-theoretic lower bound of $\log n$ bits is that k and p *together* encode the allocated location, rather than p alone. Thus this scheme provides a mechanism for how to use information already available about a pointer (namely, who "owns" the pointer) to compress the pointer to size $o(\log n)$ bits.

We refer to the algorithms for the functions $\textsc{Allocate}(k)/\textsc{Dereference}(k,p)/\textsc{Free}(k,p)$, along with the array A and any associated metadata M, as a ***dereference table***. We will often refer to the users (i.e., the owners of tiny pointers) as ***keys*** and to the data stored at the allocated locations pointed at by the tiny pointers as ***values***. A dereference table that stores q-bit values in an array of nq bits (and using $O(n)$ bits of metadata) is said to support ***load factor*** $1-\delta$ if the table is capable of storing $(1-\delta)n$ values at a time.

An ideal dereference table would simultaneously support a load factor with $\delta = o(1)$, tiny-pointer sizes of $o(\log n)$, and constant-time operations with high probability. As we shall see, not only is such a guarantee possible, but there is a rich tradeoff curve between the load factor and the tiny-pointer size.

Chapter 16: Optimal tiny-pointer constructions In Chapter 16, we first develop a comprehensive theory of tiny pointers.

We consider both ***fixed-size tiny pointers*** (where all of the tiny pointers have the same size in bits) and ***variable-size tiny pointers*** (where every tiny pointer is of bounded expected size, but different tiny pointers may have different sizes).

For fixed-size tiny pointers, we show that for any load factor $1-\delta \in \Omega(1)$, there is a lower bound of $\Omega(\log\log\log n)$ on the tiny-pointer size s. On the other hand, parameterizing by δ, we show that it is possible to achieve a fixed tiny-pointer size $s = O(\log\log\log n + \log\delta^{-1})$, and we give a lower bound showing that this tradeoff curve is tight.

We show that the $\log\log\log n$ barrier can be eliminated by instead using variable-size tiny pointers. We prove that for any load factor $1-\delta$, it is possible to achieve average tiny-pointer size $s = O(1+\log\delta^{-1})$, and again we prove that this tradeoff curve is tight for all δ.

For variable-size tiny pointers, our construction offers a remarkably strong concentration bound on each tiny pointer's size: if the expected size is k, then the probability of any given allocation returning a tiny pointer of size greater than $k+j$ for any $j>0$ is *doubly exponentially small* in j.

All of our dereference-table constructions guarantee constant-time operations with high probability, that is, with probability $1 - 1/\operatorname{poly} n$. Thus, tiny pointers can be integrated into data structures while incurring only a constant-factor time overhead.

Chapter 17: Using tiny pointers to get tiny data structures. In addition to constructing dereference tables with tiny pointers, we show that such dereference tables can be used to obtain improved solutions for a number of classic problems:

- A data structure storing n v-bit values for n keys with constant-time modifications and queries can be implemented to take space $nv + O(n \log^{(r)} n)$ bits, for any constant $r > 0$, as long as the user stores a tiny pointer of expected size $O(1)$ with each key—here, $\log^{(r)} n$ is the r-th iterated logarithm.[2]
- Any binary search tree storing n sortable keys in n nodes can be made succinct with constant time overhead, and can be made within $O(n)$ bits of optimal with $O(\log^* n)$-time modifications. This holds even for rotation-based trees such as the splay tree, which is conjectured to be dynamically optimal.
- Any fixed-capacity key-value dictionary storing v-bit values can be made stable (i.e., items do not move once inserted) with constant time overhead an additive $O(\log v)$-bit space overhead per value.
- Any key-value dictionary that requires uniform-size values can be made to support arbitrary-size values with constant time overhead and with an additional space consumption of $\log^{(r)} n + O(\log j)$ bits per j-bit value, where $r > 0$ is an arbitrary constant.
- Given an external-memory array A of size $(1+\varepsilon)n$ containing a dynamic set of up to n key-value pairs, it is possible to maintain an internal-memory stash of size $O(n \log \varepsilon^{-1})$ bits so that the location of any key-value pair in A can be computed in constant time (and with no IOs).

What unifies these problems is that each is easy to solve space-inefficiently with pointers, and the difficulty in solving them space-efficiently stems from the challenge of eliminating the pointer overhead.

A theme throughout our applications of tiny pointers is the importance of having access to the full tradeoff curve of optimal tiny-pointer constructions. This is because of the need to balance two types of space overheads: that of storing the tiny pointers themselves, and that of storing the dereference table. The former is determined by tiny-pointer size and the latter is determined by load factor.

Relationship to dynamic perfect hashing. In order to understand what makes the tiny-pointer abstraction powerful, let us take a moment to consider the following (more classical) approach to removing pointer overhead in the setting where each value has a unique owner: construct a dynamic perfect hash function that maps keys to slots in a densely packed array, and replace pointer dereferences with queries to this hash function. Such an approach has a certain elegance because it removes the pointers entirely. However, it also hits a fundamental bottleneck: any dynamic perfect hash function mapping n $(1+\Theta(1))\log n$-bit keys to $(1+\delta)n$ slots must use $\Theta(n \log\log n + n \log \delta^{-1})$ bits of metadata [44, 146].

The $n \log\log n$-bit term means that dynamic perfect hashing cannot be used to simulate pointers of size any smaller than $\log\log n$ bits. What makes our results on tiny pointers surprising is that, by *reducing* the lengths of pointers (rather than attempting to eliminate them entirely), one can blast through the $n \log\log n$ lower bound, enabling both our bounds on tiny pointers and the data-structural applications

[2]That is, $\log^{(1)} n := \log n$ and $\log^{(i+1)} n := \log\log^{(i)} n$.

that we present.

Relationship to other work in this thesis. An interesting feature of the results in this part of the thesis is that they make heavy use of the results/technique developed in earlier parts. The balls-and-bins techniques from Chapter 10 (and especially Lemma 104) play a critical role in the tiny-pointer constructions in Chapter 16. Then, in our applications of tiny pointers (Chapter 17) we find that the very space-efficient hash table results (Chapter 12) come into play. Indeed, one of the main technical insights in our applications is the remarkable symbiosis between tiny pointers and very space-efficient hash tables, in which tiny pointers can be used to bring the very strong space bounds from Chapter 12 to other seemingly quite different problems.

Chapter 16

From Balls and Bins to Tiny Pointers

In this chapter, we present optimal constructions for both fixed-size and variable-size tiny pointers. All of our constructions support constant-time operations with high probability, while determining a tradeoff curve between the load factor $1-\delta$ of the dereference table and the size s of each tiny pointer.

Theorems 177 and 178 combine to establish an optimal bound of $s = \Theta(\log\log\log n + \log \delta^{-1})$ for the fixed size case.

Theorem 177. *For every $\delta \in (0,1)$ there is a dereference table that (i) succeeds on each allocation w.h.p., (ii) has load factor at least $1-\delta$, (iii) has constant-time updates w.h.p., and (iv) has tiny pointers of size $O(\log\log\log n + \log \delta^{-1})$.*

Theorem 178. *Consider a universe $\mathcal{U}$ of keys, where $\mathcal{U}$ is assumed to have a sufficiently large polynomial size. If a dereference table supports fixed-sized tiny pointers of size s and load factor $1-\delta = \Omega(1)$, then $s = \Omega(\log\log\log n + \log \delta^{-1})$.*

Theorems 179 and 180 then combine to establish an expected optimal bound of $s = \Theta(\log \delta^{-1})$ for the variable-size case.

Theorem 179. *For every $\delta \in (0,1)$, there exists a dereference table that (i) succeeds on each allocation w.h.p., (ii) has load factor at least $1-\delta$, (iii) has constant-time updates w.h.p., and (iv) has tiny pointer size $O(P + \log \delta^{-1})$, where P is a random variable such that $\Pr(P \geq i) \leq 2^{-2^{\Omega(i)}}$ for all i. In particular, the tiny pointer size is $O(1 + \log \delta^{-1})$ in expectation.*

Theorem 180. *Consider a universe $\mathcal{U}$ of keys, where $\mathcal{U}$ is assumed to have a sufficiently large polynomial size. If a dereference table supports variable-sized tiny pointers of expected size s and load factor $1-\delta = \Omega(1)$, then $s = \Omega(\log \delta^{-1})$.*

The structure of the chapter is as follows. We begin with preliminaries, including a formal definition of the tiny-pointer abstraction (Section 16.1). We then present the upper bounds for the fixed-size case (Section 16.2) and the variable-size case (Section 16.3), after which we prove both lower bounds (Section 16.4).

Relationship to balls and bins. An important strand of our story will be the relationship between tiny-pointer/dereference-table constructions and the balls-and-bins results from Chapter 10.

To see why this is the case, it may be helpful to consider the following very basic dereference-table construction: Break the array into bins of size polylog(n); assign each element x in the dereference table to a random bin via a hash function; and within that bin, assign the element to an arbitrary vacant slot. Assuming the load factor $1 - \delta$ is at most $1 - 1/\log n$, we can use simple Chernoff bounds to argue that every element will map to a bin that has room for it. We can then construct the tiny pointer for a given element to simply be the *position where the element is stored in its bin.* Since each bin has size $\operatorname{polylog} n$, this simple construction leads to $O(\log \operatorname{polylog} n) = O(\log \log n)$-bit tiny pointers (although a bit more work is needed to make the construction time efficient).

Thus, even the basic SINGLECHOICE strategy can be transformed into a nontrivial tiny-pointer scheme! Given this relationship, it should come as no surprise that the optimal construction for fixed-size tiny pointers will end up following *almost immediately* from the ICEBERG balls-and-bins scheme that we gave in Chapter 10. Our construction for variable-size tiny pointers will then take the same set of techniques and extend them further.

16.1 Preliminaries

Operations. A dereference table with q-bit-values is a data structure that supports the following operations:

- CREATE(n, q): The procedure creates a new dereference table, and returns a pointer to an array with n slots, each of size q bits. We call this array the ***store***.
- ALLOCATE(k): Given a key k, the procedure allocates a slot in the store to k, and returns a bit string p, which we call a ***tiny pointer***.
- DEREFERENCE(k, p): Given a key k and a tiny pointer p, the procedure returns the index of the slot allocated to k in the store. If p is not a valid tiny pointer for k (i.e., p was not returned by a call to ALLOCATE(k)), then the procedure may return an arbitrary index in the store.
- FREE(k, p): Given a key k and a tiny pointer p, the procedure deallocates slot DEREFERENCE(k, p) from k. The user is only permitted to call this function on pairs (k, p) where p is a valid tiny pointer for k (i.e., p was returned by the most recent call to ALLOCATE(k)).

We say a key k is ***present*** if it has been allocated more recently than it has been freed; in this case the tiny pointer p returned by the most recent call to ALLOCATE(k) is said to be k's tiny pointer. The user is only permitted to allocate at most one tiny pointer p to each key k. That is, each time that ALLOCATE(k) is called to obtain some tiny pointer p, the function FREE(k, p) must be called before ALLOCATE(k) can be called again.

We say that slot i in the store is ***occupied*** if there is a present key k with tiny pointer p such that $\textsc{Dereference}(k, p) = i$, and otherwise we say it is ***free***. We typically refer to the parameter n (i.e., the number of slots in the store) as the table's ***size*** or ***capacity***.

Guarantees. Dereference tables provide the following guarantees:

- For any two present keys $k_1 \neq k_2$ with tiny pointers p_1 and p_2, respectively, $\textsc{Dereference}(k_1, p_1) \neq \textsc{Dereference}(k_2, p_2)$.
- $\textsc{Dereference}(k, p)$ only depends on k, p, random bits, and the parameter n.

The second property ensures that the act of dereferencing a tiny pointer is similar to the act of dereferencing a standard pointer; in both cases, one does not need to access the data structure being pointed into in order to perform the dereference. This ends up being important for several of our applications later. In particular, it ensures that in external-memory applications, each dereference incurs only a single I/O; and it ensures that in data-structure applications, the locations pointed at by tiny pointers are stable (i.e., once a tiny pointer p is allocated to a key k, the location that is being pointed at does not change).

Metadata information. The dereference table may store metadata in order to perform updates (allocations and frees) efficiently. Metadata can either be stored as part of the store, or in an auxiliary data structure that is permitted to consume up to $O(n)$ bits. In other words, the dereference table is allowed to use $O(n)$ bits (i.e., $O(1)$ bits of overhead per slot) of metadata for "free", without that counting towards the space consumption of the store, but any additional metadata must count towards the space consumption of the store. Note that the dereference table is not allowed to store metadata in any slot of the store that is currently allocated.

Failure probability. We will permit allocations to have a small failure probability. That is, each allocation is permitted to fail with probability $1/\operatorname{poly}(n)$, in which case the allocation simply returns a failure message rather than a tiny pointer. In general, if a random event occurs with probability $1 - 1/\operatorname{poly}(n)$, we say that it occurs ***with high probability (w.h.p.)***.

We remark that, when analyzing dereference tables, we shall always assume that the sequence of allocations, frees, and dereferences are determined by an oblivious adversary (i.e., the sequence is determined ahead of time, rather than adapting to the behavior of the dereference table). One consequence of this is that, if a given allocation fails, the only effect on the operation sequence is that the corresponding call to $\textsc{free}$ is removed.

Load factor. Any implementation of a dereference table must also specify an additional parameter $\delta \in [0, 1]$ dictating how full the table is allowed to be. This means that the dereference table can support up to $(1-\delta)n$ allocations at a time—the quantity $1-\delta$ is referred to as the table's ***load factor***. If the $\textsc{Allocate}$ function is called when there are already $(1-\delta)n$ allocations performed, then the dereference table is permitted to fail the allocation.[1]

Since dereference tables can use up to $O(n)$ space for metadata, the total amount of space consumed by a dereference table may be as large as $nq+O(n) = (1-\delta)nq + \delta nq + O(n)$. The first term $(1-\delta)nq$ is space that allocations can make use of, and the other terms $\delta nq + O(n)$ are wasted space. Note that there is no point in considering $\delta \ll 1/q$, since this just makes it so that they wasted space is dominated by metadata. Thus, when constructing a reference table with some load factor $1-\delta$, we shall always implicitly assume that $q \geq \Omega(\delta^{-1})$.

Hashing and independence. Our dereference-table constructions will all make use of hash functions. For simplicity, we shall treat hash functions as being uniform and fully independent. This assumption is without loss of generality since there are already known families of hash functions [162, 293] that simulate n-independence with constant-time evaluation and $O(n)$ random bits, and there are already well understood techniques [52, 245] for applying these families to data structures that require poly n-independence[2]. These known techniques can easily be applied directly to all of our data structures; the only caveat is that the families of hash functions being used [162,293] introduce their own additional $1/\operatorname{poly}(n)$ failure probability to the data structure. So, even if a data structure offers sub-polynomial failure probability under the assumption of fully random hash functions, if we wish to use an explicit family of hash functions, then we must allow for a $1/\operatorname{poly}(n)$ failure probability.

16.2 Upper Bound for Fixed-Size Pointers

In this section, we give optimal constructions for fixed-size tiny pointers. We prove the following theorem:

Theorem 177. *For every $\delta \in (0,1)$ there is a dereference table that (i) succeeds on each allocation w.h.p., (ii) has load factor at least $1-\delta$, (iii) has constant-time updates w.h.p., and (iv) has tiny pointers of size $O(\log\log\log n + \log \delta^{-1})$.*

In particular, for $\delta = 1/\log\log n$, we get tiny pointers of size $O(\log\log\log n)$. Thus, we can doubly-exponentially beat raw $\log n$-bit pointers, while still supporting a load factor of $1 - o(1)$.

The proof is the simplest of our tiny-pointer constructions, and can be viewed essentially as a *reinterpretation* of the ICEBERG balls-to-bins scheme presented in Chapter 10. Indeed, one way to think about the construction is that we will partition our array into bins, we will place the elements into those bins using ICEBERG with

[1]Note that, even though a dereference table only guarantees the ability to store up to $(1-\delta)n$ allocations at a time, we still use the terms "size" and "capacity" of a dereference table to refer to n, rather than $(1-\delta)n$, since n represents the total number of q-bit entries in the store.

[2]The basic idea is to simply replace the data structure of capacity n with $n^{1-\varepsilon}$ data structures of capacity n^{ε}. Each element x in the full data structure gets hashed at random to one of the $n^{1-\varepsilon}$ data structures, each of which only requires $\operatorname{poly}(n^{\varepsilon}) = o(n)$ independence.

three bin-choice hash functions h_1, h_2, h_3, and then we will define the tiny pointer for a given element x to be the pair (i, j), $i \in [3]$, such that x is in position j of bin $h_i(x)$. The fact that ICEBERG achieves strong load-balancing guarantees will allow for us to ensure a high load factor in each bin, while keeping the bins small so that the tiny pointers are also small.

Rather than state the fixed-size construction directly as an application of ICEBERG, it will be helpful to describe it as the combination of two building blocks (indeed, these same building blocks will then be useful for constructing variable-size tiny pointers).

The first building block: load-balancing tables. A load-balancing table is a simple type of dereference table that has a very specific internal representation, and that, unlike normal dereference tables, is permitted to fail on calls to ALLOCATE with a non-negligible probability. Roughly speaking, if a load-balancing table has load factor $1 - \delta$, then the load-balancing table is permitted to fail on a δ-fraction of allocations.

Load-balancing tables are implemented as follows. If the store is of some size m, then we partition it into m/b buckets of size $b = \Theta(\delta^{-2} \log \delta^{-1})$. To allocate a key k, we hash k into one of the buckets, using a hash function h. If bucket $h(k)$ contains a free slot, then we allocate any free slot $i \in [b]$ within that bucket, and we return i as the tiny pointer. Otherwise, all b slots in the bucket are occupied, and the allocation fails. The function DEREFERENCE(k, p) can then be implemented to simply return the p-th slot in bin $h(k)$.

Load-balancing tables will serve as a building block in the dereference tables that we construct. The basic idea is that we can use a load-balancing table to handle all but a δ-fraction of allocations, and the remaining allocations can be handled via some other mechanism. Thus, we will need the following lemma which bounds the total number of failed allocations at any given moment:

Lemma 181. *Consider a load-balancing table with size m and load factor $1 - \delta$. Consider a sequence of allocations and frees such that no more than $(1 - \delta)m$ allocations are made at any given moment. If an allocation fails, and the allocation would have been freed at some time t, then we consider the allocation to be* **alive** *up until that time t. At any given moment, the number of allocations that have failed and are still alive is $O(\delta m)$ with probability at least $1 - \exp(-\operatorname{poly}(\delta)m)$.*

We have already seen the proof of Lemma 181 earlier in the thesis. Indeed, it follows directly from Lemma 104 in Chapter 10.

We remark that in all of our applications of Lemma 181, we will have w.l.o.g. that $\log \delta^{-1} = o(\log m)$ (since, otherwise, we would have $\log \delta^{-1} = \Omega(\log m)$ and so could just use standard $O(\log m)$-bit pointers). Thus the probability bound offered by the lemma will always be at least $1 - \exp(m^{1-o(1)}) \geq 1 - 1/\operatorname{poly}(m)$.

To conclude our discussion of load-balancing tables, we must describe how to implement allocations and frees in constant time. Here, there are two cases, depending on how b compares to the size n of the dereference table that the load-balancing table is being used within.

If $b \leq \log n$, then we can store a b-bit bitmap for each bucket indicating which slots in the bucket are free; and we can use standard bit-manipulation on the bitmap to implement the allocation and free functions in constant time.

We take a different approach if $b \geq \log n$. In this case, we claim that without loss of generality, $q = \omega(\log b)$, where q is the size in bits of the elements being stored (we will prove this claim in a moment). This claim means that we can keep track of which slots are free in each bucket of a load-balancing table as follows: we simply store a ***free list*** in each bucket, that is, a linked list consisting of all the free slots, where each free slot contains a pointer to the next free slot in the list. This is possible since each free slot is q bits and each pointer in the linked list needs only $\log b = o(q)$ bits. The $\log b$-bit base pointers of the m/b linked lists can be stored in an auxiliary metadata array of size $O((m/b) \cdot \log b) \leq O(m)$, where m is the size of the load-balancing table. The free lists allow for us to implement the allocation and free functions in constant time.

To prove that this free-list approach works, it remains to show that $q = \omega(\log b)$ without loss of generality. Let $1 - \delta$ be the load factor of the full dereference table (that the load-balancing table is part of) and let $1 - \delta'$ be the load factor of the load-balancing table. Since $b \geq \log n$, we must have $\delta'^{-1} = \tilde{\Omega}(\sqrt{\log n})$. In all of our constructions of dereference tables, if we use a load-balancing table with load factor $1 - \delta'$ satisfying $\delta'^{-1} = \tilde{\Omega}(\sqrt{\log n})$ (or even $\delta'^{-1} = \omega(\log \log n)$), we will always have $\log \delta^{-1} \geq \Omega(\log \delta'^{-1})$. Recall that, if a dereference table has load factor $1 - \delta$, then it is assumed that the dereference table is storing objects of size $q \geq \Omega(\delta^{-1})$ bits. Thus, we have that $q = \omega(\log \delta^{-1}) = \omega(\log \delta'^{-1}) = \omega(\log b)$, as desired.

The second building block: a power-of-two-choices dereference table. To compensate for the high failure probability of load-balancing tables, we develop our second building block: a simple dereference table that supports $O(\log \log \log n)$-bit tiny pointers and, unlike a load-balancing table, has low failure probability. The downside of this second building block is that it only supports a very small load factor.

Lemma 182. *There exists a δ satisfying $1 - \delta = \Theta(1/\log \log n)$, such that there is a dereference table that (i) succeeds on each allocation w.h.p., (ii) has load factor at least $1 - \delta$, (iii) has constant-time updates w.h.p., and (iv) has tiny pointers of size $O(\log \log \log n)$.*

Proof. We partition the store into buckets of size $b = \Theta(\log \log n)$. When ALLOCATE(k) is called, the key k is hashed to two buckets $h_1(k), h_2(k) \in [1, n/b]$. The key k is allocated a slot in whichever of the two buckets contains the most free slots. The tiny pointer p is $1 + \log b = O(\log \log \log n)$ bits, and indicates which slot in the two buckets was allocated.

We can think of the allocations as balls that are inserted into bins using the power-of-two-choices rule [351, 369], with the same ball possibly being inserted/deleted/reinserted over time. Since the load factor is $\Theta(1/\log \log n)$, the expected number of balls in each bin is $O(1)$. In this setting, it is known that, w.h.p., the number of balls in the fullest bin is $O(\log \log n)$ [369]. Thus allocations succeed w.h.p.

Finally, to implement allocations and frees in constant time, we can just use a bitmap to keep track of which slots in each bucket are free; since each bucket is only $O(\log \log n)$ slots, the bitmaps are each only $O(\log \log n)$ bits, and thus each bitmap fits into a machine word. Using standard bit manipulation, the bitmaps can be used to keep track of which slots are free in constant time per allocation/free (and to find a free slot for a given allocation also in constant time). The bitmaps consume a total of $O(n)$ bits of space. ∎

Putting the pieces together. Of course, power-of-two-choices dereference tables are not very useful on their own, because they only support $o(1)$ load factors. We now show how to combine them with load-balancing tables in order to prove Theorem 177.

Proof of Theorem 177. Since we are willing to have tiny pointers of size $\Theta(\log \log \log n + \log \delta^{-1})$, we can assume without loss of generality that $\delta = o\left(\frac{1}{\log \log n}\right)$.

We store a $1 - \delta^2$ fraction of the allocations in a load-balancing table of size $m = (1 - \delta/2)n$ slots that supports load factor $1 - \delta^2/c$ for some sufficiently large positive constant c; we call this the ***primary table***. Allocations that fail in the primary table are stored in a secondary table implemented with Lemma 182 to have size $n' := \delta n/2$ slots and support load factor $1 - \delta' := \Theta(1/\log \log n')$. If an allocation fails in the secondary table, or if the load factor of the secondary table ever exceeds $\Theta(1/\log \log n')$, then the allocation fails in the full dereference table as well. Note that the total size (in terms of slots) of the primary and secondary tables is n. See Figure 1 for a picture of the layouts of the two tables.

Since both the primary and secondary tables are constant time, so is the full dereference table. Additionally, each allocation can return a tiny pointer that is either in the primary table or in the secondary table (plus 1 bit of information indicating which table it is being pointed into). Since the primary and secondary tables both have tiny pointers of size $O(\log \log \log n + \log \delta^{-1})$, the claim about tiny-pointer size is also proven.

Our final task is to bound the probability of a given allocation failing. Lemma 181 tells us that the number of allocations in the secondary table will be at most $\delta^2 n$ at any given moment w.h.p. Since the secondary table has $n' = \Theta(\delta n/2)$ slots, and since $\delta = o\left(\frac{1}{\log \log n}\right)$, it follows that the number of allocations in the secondary table at any given moment is $o(n'/\log \log n) = o(n'/\log \log n')$ with high probability. We therefore get from Lemma 182 that the allocations in the secondary table each succeed with high probability in n'. Without loss of generality, $n' \geq \sqrt{n}$ (since otherwise $\delta \leq O(1/\sqrt{n})$, and we can just use standard $\log n$-bit pointers). Thus the allocations in the secondary table each succeed with high probability in n. ∎

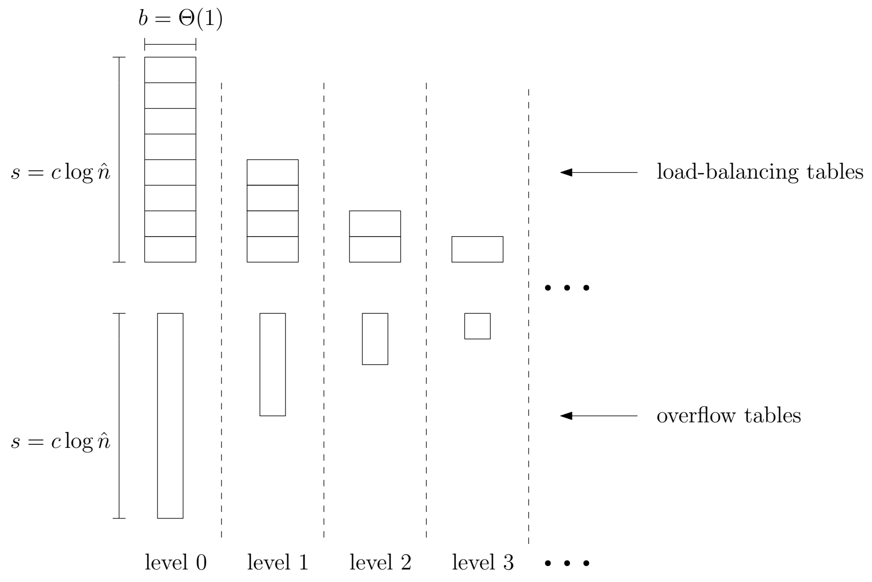


Figure 1: A pictoral representation of the layouts of the primary and secondary tables. The primary table is implemented to support load factor $1 - \Theta(\delta^2)$, so that only $\delta^2 n$ allocations overflow to the secondary table at a time. The secondary table is implemented to have size $n' = \delta n/2$ and to support a (much sparser) load factor of $\Theta(1/\log\log n') = \omega(\delta)$, so that it can successfully store all of the overflowed allocations from the primary table.

16.3 Upper Bounds for Variable-Sized Pointers

In this section, we give optimal constructions for variable-size tiny pointers. We prove the following theorem:

Theorem 179. *For every $\delta \in (0, 1)$, there exists a dereference table that (i) succeeds on each allocation w.h.p., (ii) has load factor at least $1 - \delta$, (iii) has constant-time updates w.h.p., and (iv) has tiny pointer size $O(P + \log\delta^{-1})$, where P is a random variable such that $\Pr(P \geq i) \leq 2^{-2^{\Omega(i)}}$ for all i. In particular, the tiny pointer size is $O(1 + \log\delta^{-1})$ in expectation.*

We can assume without loss of generality that $1-\delta < \alpha$ for some sufficiently small positive constant α of our choice (if $1-\delta > \alpha$, we can reset $\delta = 1-\alpha = \Theta(1)$ without changing the guarantee of the theorem).

Observe that, using the same primary/secondary-table construction as in the proof of Theorem 177, we can immediately reduce to the case where the load factor is a positive constant of our choice. Indeed, suppose that we could implement a dereference table T with load factor α for some positive constant $\alpha > 0$ and average tiny pointer size $O(1)$. Then we can use T as the secondary table in the construction: if the entire dereference table supports load factor $1-\delta$, then the requirement from the secondary table is that it must be able to support $\delta^2 n$ elements using $\delta n/2$ slots. So as long as $\delta < \alpha/2$ (which is without loss of generality) then T suffices.

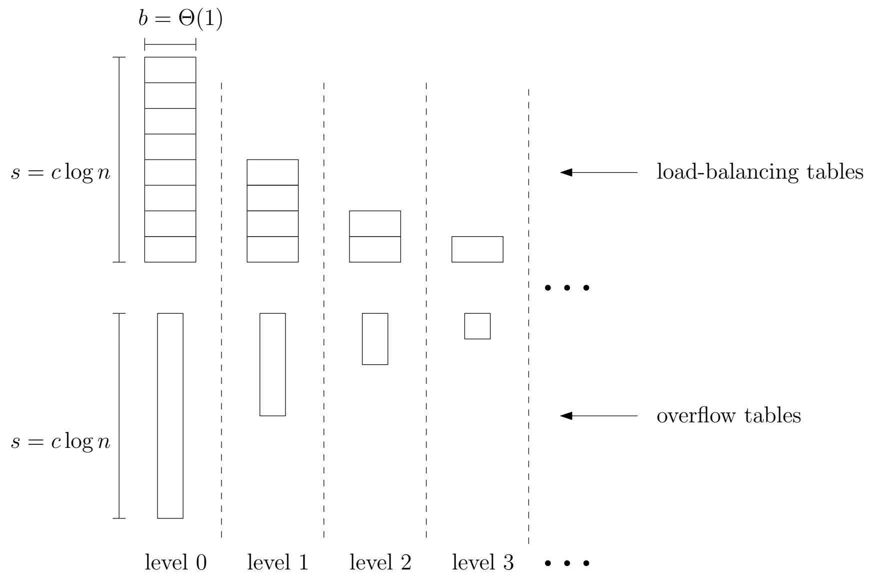


Figure 2: A pictoral representation of the layout used to implement each container of size $\Theta(\log n)$. When an allocation fails in the i-th load-balancing table, it either proceeds to the $(i + 1)$-th load-balancing table (if $L_{i+1} < s_{i+1}$) or it proceeds to the i-th overflow array (which is deterministically guaranteed to have a free slot).

Thus our task of proving Theorem 179 reduces to the task of proving the following proposition.

Proposition 183. *There exists a dereference table that (i) succeeds w.h.p. (ii) has load factor* $\Omega(1)$, *(iii) has constant-time updates w.h.p. in* n, *and (iv) has tiny pointer size* P, *where* P *is a random variable satisfying* $\Pr(P \geq i) \leq 2^{-2^{\Omega(i)}}$ *for all* i.

For ease of discussion, throughout the rest of the section, we use n to denote the maximum number of items that can be stored in the dereference table (rather than the number of slots), and we aim to construct a dereference table with $O(n)$ slots.

Constructing the dereference table. We now describe our construction for the dereference table that we use to prove Proposition 183. The dereference table initially hashes every key into one of $n/\log n$ ***containers***, so that, at all times, any container has $\log n$ items in expectation. We deterministically limit the number of elements in each container to $s = c \log n$ items, for some large enough constant $c > 1$ to be determined later. When a key is hashed into a container that already has $c \log n$ items, the allocation fails.

Each container is managed independently, and its allocations/frees are performed using a scheme with $\log_2 s$ levels, as follows. For every $0 \leq i < \log_2 s$, the ith level is a load-balancing table with $s_i := s/2^i$ buckets, each with b slots, for some large enough constant $b \geq 2$ to be determined.

The basic idea is that, when an allocation in level i fails due to bucket fullness, we recursively attempt the allocation in the next level $i + 1$ (which uses a different hash function than does level i). Intuitively, as long as b is a sufficiently large constant, then each level should succeed on at least 1/2 of its allocations, which is why the next level $i + 1$ can afford to be half the size of the previous one.

The problem with this basic construction is that if even just a few consecutive levels behave badly, resulting in $\omega(s_i)$ elements being sent to some level i, then there may not be room for those elements in all of the levels $i, \ldots, \log_2 s$ combined. On the other hand, our construction must be able to handle such bad scenarios, because most of the levels are so small that we cannot offer high-probability guarantees on their behavior. Thus, we must modify the construction so that, when a level behaves badly, the effects of that are isolated.

To do this, we add a fallback structure to each level, that we call ***overflow array***, to prevent excessive occupancy. The overflow array in each level i has s_i slots (the same number of slots as the load-balancing table at that level). Let L_i be the random variable denoting the number of values currently stored in levels i or larger, including their overflow arrays. Whenever an allocation at some level i fails (due to bucket fullness), we recursively allocate in the next level only if $L_{i+1} < s_{i+1}$, and otherwise, we place the value in any available slot in the overflow array of level i. The result of this is that we deterministically guarantee $L_i \leq s_i$ for every level i (including level 0, for which this is trivial, since $s_0 = s$).

Importantly, no overflow array can ever run out of space: since $L_i \leq s_i$ (deterministically), the total number of elements in the overflow array for level i is also a guaranteed to be a most s_i, which is precisely the capacity of the overflow array.

We are now ready to describe the full allocation algorithm. See Figure 2 for a picture of the layout used to implement each container.

```
ALLOCATE($k$):
  1. Hash $k$ into one of the $n/\log n$ containers.
  2. If the selected container is already at full capacity $s$, fail.
  3. Else, allocate $k$ in the selected container:
     (a) For each $i = 0, 1, \ldots, \log_2 s - 1$:
           i. Increment $L_i$.
          ii. Try to allocate $k$ in the $i$th load-balancing table.
         iii. If the allocation succeeds:
                • Let $j$ be the chosen slot within the chosen bucket.
                • Return (level $i$, load-balancing table, bucket slot $j$).
          iv. If $L_{i+1} \geq s_{i+1}$:
                • Pick any free slot in the $i$-th overflow array.
                • Let $j$ be the chosen slot in the array.
                • Return (level from the back $\log_2 s - 1 - i$, overflow
                  array, slot $j$).
```

Notice that, if an allocation ends up using a slot j in some bucket in the i-th level's load-balancing table, then the tiny pointer encodes: the quantity i, which is $O(\log i)$ bits; the fact that the allocation used the load-balancing table rather than the overflow array, which is $O(1)$ bits; and the quantity j, which is $O(\log b) = O(1)$ bits. The total length of the tiny pointer is $O(\log i)$ in this case.[3]

On the other hand, if an allocation ends up using the j-th slot in the i-th level's overflow array, then the tiny pointer encodes: the quantity $\log_2 s - 1 - i$, which is $O(\log(\log_2 s - 1 - i))$ bits; the fact that the allocation used the overflow array rather than the load-balancing table, which is $O(1)$ bits; and the quantity j, which is $O(\log s_i)$ bits. Importantly, in this case, we elect to encode $\log_2 s - 1 - i$, rather than the equivalent quantity i. This allows us to bound the total size of the tiny pointer by $O(\log(\log_2 s - 1 - i)) + O(1) + O(\log s_i) = O(\log \log(s/2^i) + \log s_i) = O(\log \log s_i + \log s_i) = O(\log s_i)$. Thus, when an allocation uses the overflow array in level i, we can bound the tiny-pointer size by $O(\log s_i)$.

Implementing operations in constant time. The information in the tiny pointers allows for dereferences to easily be performed in time $O(1)$. Performing allocations and frees in time $O(1)$ is slightly more difficult, however.

Let us start by considering the naïve approach to implementing allocations and see why this is too slow. We must first identify which container to use (this just requires us to evaluate a hash function, taking constant time). We must then determine which level we will be using; if we end up using level i, then this takes time $\Theta(i)$, which is too slow when $i = \omega(1)$.

We solve this problem as follows. Let d to be some sufficiently large positive constant. We will implement levels $0, 1, \ldots, d-1$ using the naive approach, and then we will implement the levels $d, \ldots, \log_2 s$ using the Method of Four Russians (i.e., the "lookup-table approach"). Notice that the total number of slots in the levels $d, \ldots, \log_2 s$ is at most $4s_d/2^d \le (\log n)/10$. Thus the entire state of which slots are free in those levels can be encoded in $(\log n)/10$ bits; we store this quantity as metadata for each container, totaling to $O(n)$ bits of metadata across all $n/\log n$ containers. Moreover, the hashes $h_1(k), h_2(k), \ldots, h_{\log_2 s}(k)$ that are used to select a bucket in each level together represent only $O((\log \log n)^2)$ bits (and can be implemented to just be the first $O((\log \log n)^2)$ bits of a single hash function). Thus, the entire state of levels $d, \ldots, \log_2 s$, plus all of the information about the hashes $h_1(k), h_2(k), \ldots, h_{\log_2 s}(k)$, can be encoded in an integer ϕ of $(\log n)/2$ bits that can be constructed in time $O(1)$. This means that we can pre-construct a lookup table of size $2^{(\log n)/2} = \sqrt{n}$ that we can use to determine, for any given value of ϕ, which level the allocation should use. The lookup table takes a negligible amount of metadata space, allows for allocations to be performed in time $O(1)$, and can be constructed in time $\tilde{O}(\sqrt{n})$ during the dereference table's creation.

Now that we have specified how to implement allocations, frees are simple to implement, since they just update the metadata to reflect that the slot has been

[3]We follow the convention that $\log i = \Omega(1)$ for all i, so $\log 0$ and $\log 1$ are set to 1.

freed (this just flips a single bit in the metadata).

We have now fully specified the construction and implementation of our dereference table. It remains to analyze its properties, namely the probability of failure, the load factor, and the tiny-pointer sizes.

Probability of failure. The only way that an allocation can fail is if there is no room in the container that it hashes to, i.e., the container has $c \log n$ elements already. Otherwise, if the container has fewer than $c \log n$ elements, then the allocation is guaranteed to succeed (but, of course, it is not guaranteed to result in a small tiny pointer).

On average, $\log n$ items hash to any particular container, so by a Chernoff bound the maximum size across all containers is at most $c \log n$ w.h.p. in n for some positive constant c. By the union bound, this holds for all of the $n/\log n$ containers simultaneously, w.h.p. in n. Thus, if we pick $s = c \log n$ for some large enough constant c, at any point in time, all containers will be below capacity w.h.p. in n.

Load factor. Next, we verify that the total number of slots is $O(n)$. The dereference table for each container uses space $O(\sum_i s_i) = O(s_0) = O(s) = O(\log n)$ slots, and there are $n/\log n$ containers. Hence, the total space is $O(n)$, so the load factor is $\Omega(1)$, as desired.

Tiny pointer size. To conclude the proof of Proposition 183, we analyze the tiny pointer size of a given allocation, conditioned on the event that the allocation doesn't fail. The size of the tiny pointer depends on where the key ends up allocated. Specifically, it is:

- $O(\log i)$ if the key is allocated in the ith load-balancing table;
- $O(\log s_i)$ if the key is allocated in the ith overflow array.

Fix an arbitrary container to be the one where the allocation takes place, and consider the following events:

- $\mathcal{B}_i$: the key is allocated in the ith load-balancing table;
- $\mathcal{O}_i$: the key is allocated in the ith overflow array;
- $\mathcal{L}_i$: $L_i < s_i$.

We will condition on two events: (i) that the element picks the container we fixed, and (ii) that the container contains fewer than $c \log n$ elements (i.e., the allocation doesn't fail). We will drop the conditioning notation for clarity. Let P be the size of the output tiny pointer. Then, by the law of conditional expectation,

$$\mathbb{E}[P] \le \sum_i \Pr(\mathcal{B}_i) \cdot O(\log i) + \sum_i \Pr(\mathcal{O}_i) \cdot O(\log s_i). \tag{16.1}$$

We bound each term separately. On the one hand,

$$\begin{aligned}\Pr\left(\mathcal{B}_i\right) &\le \Pr\left(\overline{\mathcal{B}_0}, \mathcal{L}_1, \overline{\mathcal{B}_1}, \ldots, \mathcal{L}_{i-1}, \overline{\mathcal{B}_{i-1}}\right) \\ &\le \Pr\left(\overline{\mathcal{B}_0}\right) \cdot \Pr\left(\overline{\mathcal{B}_1} \mid \overline{\mathcal{B}_0}, \mathcal{L}_1\right) \cdots \Pr\left(\overline{\mathcal{B}_{i-1}} \mid \overline{\mathcal{B}_0}, \mathcal{L}_1, \ldots, \overline{\mathcal{B}_{i-2}}, \mathcal{L}_{i-1}\right). \qquad (16.2)\end{aligned}$$

For every j, the load factor of level j is at most $1/b$, because there are $L_j < s_j$ items, s_j buckets, and each bucket has capacity b. This means that at most $1/b$ of the bins are full, deterministically, so the probability that a full bucket is chosen at most $1/b$. Hence, every term in Equation (16.2) is bounded by $1/b$, and

$$\Pr\left(\mathcal{B}_i\right) \le 1/b^i \le 1/2^i.$$

On the other hand,

$$\Pr[\mathcal{O}_i] \le \Pr[\overline{\mathcal{L}_{i+1}}].$$

We can bound the latter probability using Lemma 181. By construction, the load-balancing table in level i always has at most s_i allocations made to it (including the failed ones, since $L_i \le s_i$ and L_i counts both the elements in level i and the elements in levels $i+1, i+2, \ldots$); moreover, the allocations and frees performed on the table are independent of the randomness used in the table. Assuming that the bucket-size b is a sufficiently large constant, it follows that we can apply Lemma 181 to deduce that, with probability at least

$$1 - \exp(-\operatorname{poly}(b) s_i) = 1 - \exp(-\Omega(s_i)),$$

the number of failed allocations at level i at any given moment is at less than $s_i/2 = s_{i+1}$ (and hence $\mathcal{L}_{i+1}$ holds). Thus, we can conclude that

$$\Pr[\mathcal{O}_i] \le 1/2^{\Omega(s_i)}.$$

Putting the pieces together,

$$\mathbb{E}\left[P\right] = \sum_i \frac{O(\log i)}{2^i} + \sum_i \frac{O(\log s_i)}{2^{\Omega(s_i)}} = O(1).$$

Notice that these calculations show that a tiny pointer of size $O(\log \ell)$ has probability $2^{-\Omega(\ell)}$, or, equivalently, a tiny pointer of size $O(\ell)$ has probability $2^{-2^{\Omega(\ell)}}$. This suggests that the tiny pointer size decays at a doubly-exponential rate. We prove this next. For any ℓ,

$$\begin{aligned}\Pr(P \geq \ell) &\leq \sum_{i:\, O(\log i) \geq \ell} \Pr(\mathcal{B}_i) + \sum_{i:\, O(\log s_i) \geq \ell} \Pr(\mathcal{O}_i) \\ &= \sum_{i \geq 2^{\Omega(\ell)}} \Pr(\mathcal{B}_i) + \sum_{s_i \geq 2^{\Omega(\ell)}} \Pr(\mathcal{O}_i) \\ &= \sum_{i \geq 2^{\Omega(\ell)}} \frac{1}{2^i} + \sum_{s_i \geq 2^{\Omega(\ell)}} \frac{1}{2^{\Omega(s_i)}}.\end{aligned}$$

Both sums are dominated by their first terms, and are thus $1/2^{2^{\Omega(\ell)}}$. Therefore,

$$\Pr(P \geq \ell) \leq \frac{1}{2^{2^{\Omega(\ell)}}},$$

which completes the proof of Proposition 183. As discussed earlier, Proposition 183, in turn, implies Theorem 179.

Bounding sums of tiny-pointer sizes. In our applications of tiny pointers, a common way of using variable-size pointers will be to pack $\Theta\left(\frac{\log n}{\log \delta^{-1}}\right)$ of them into $\Theta(\log n)$ bits. Therefore, we conclude this section by proving a bound of the total number of bits consumed by a set S of $O(\log n / \log \delta^{-1})$ tiny pointers.

Proposition 184. *Using the construction in Theorem 179, for any set S of $O\left(\frac{\log n}{\log \delta^{-1}}\right)$ tiny pointers, the sum of their sizes will be $O(\log n)$ bits w.h.p.*

Proof. With high probability, all of the allocations for S succeed. This means that we can ignore the case where allocations fail, so when an allocation fails, we shall treat it as contributing a tiny pointer of size 0.

Let K be the set of keys corresponding to the tiny pointers in S. The easy case is if every key $k \in S$ hashes to a different container; in this case, we can analyze each container separately to conclude that each tiny pointer ALLOCATE(k) independently has length $O(\log \delta^{-1} + P_k)$ bits, where $\Pr[P_k > \ell] \leq 2^{-2^{\Omega(\ell)}}$. Applying a Chernoff bound for sums of independent geometric random variables, we can conclude that $\sum_{k \in K} P_k \leq O(\log n)$ w.h.p., and thus that the total number of bits consumed by S is $O(\log n)$.

What if some of the keys $k \in K$ hash to the same container as others $k' \in K$? Then we can no longer analyze the lengths of the resulting tiny pointers independently. Let X denote the set of such keys k. Since each tiny pointer is deterministically at most $O(\log n)$ bits, we can complete the proof by establishing that, with w.h.p., $|X| = O(1)$.

Let $k_1, k_2, \ldots$ denote the keys in K, and let X_i be the indicator random variable for the event that k_i hashes to the same container as one of $k_1, k_2, \ldots, k_{i-1}$. Then $|X| \leq 2\sum_i X_i$. On the other hand, each X_i independently satisfies $\Pr[X_i] \leq (i - 1)/n \leq |S|/n \leq O(\log n/n)$. Thus $\sum_i X_i$ is a sum of independent indicator random variables with total mean $O(\log^2 n/n)$. Applying a Chernoff bound, we conclude that

$\sum_i X_i = O(1)$ w.h.p./, which completes the proof. ∎

16.4 Lower Bounds

In this section we prove that the bounds in Theorems 177 and 179 are tight. We begin by proving a lower bound for variable-size tiny pointers, since it is then used as part of the proof for the fixed size case.

What makes the lower bound for variably sized tiny pointer tricky is that any single tiny pointer might be very small. For example, the dereference table could have a single special slot that corresponds to the tiny pointer 0 (for every key), and then if the dereference table ever wanted to make a single tiny pointer small, it could allocate the special slot. Thus, our proof treats different types of slots differently: for each slot j, we define a potential function $\phi(j)$ indicating how "useful" that slot is to a random insertion. The idea is that insertions that use slots j with small potentials $\phi(j)$ must, on average, have relatively large tiny pointers; but insertions that use slots j with large potentials $\phi(j)$ must be rare, since only a relatively small fraction of the slots can have large potentials, and the number of insertions into them can be bounded by the number of deletions out of them.

Theorem 180. *Consider a universe $\mathcal{U}$ of keys, where $\mathcal{U}$ is assumed to have a sufficiently large polynomial size. If a dereference table supports variable-sized tiny pointers of expected size s and load factor $1 - \delta = \Omega(1)$, then $s = \Omega(\log \delta^{-1})$.*

Proof. Let $\mathcal{U}$ be a universe of size n^c where c is a sufficiently large constant. Let $\delta < 1/4$. Let T be a dereference table with n slots and load factor $1 - \delta$ (i.e., it is capable of allocating up to $(1 - \delta)n$ slots to keys from $\mathcal{U}$ at a time). Moreover, suppose that T guarantees an expected tiny-pointer length of at most μ. Then we wish to show that

$$\mu \geq \Omega(\log \delta^{-1}).$$

To simplify our discussion, we shall think of a key $k \in \mathcal{U}$ as residing in the location that is allocated to it. Thus allocations correspond to insertions, and frees correspond to deletions.

Consider a workload in which the table is initialized to contain $(1 - \delta)n$ arbitrary elements, and then we alternate between insertions and deletions for $n^{c/2}$ steps. Each insertion selects a random element of $\mathcal{U}$ (with high probability in n, we never insert an element that is already present), and each deletion selects a random element out of those present.

We treat tiny pointers as taking values in $\mathbb{N}$. If the tiny pointer takes value i, then it uses $\Omega(\log i)$ bits. For each element $x \in \mathcal{U}$, let $h_i(x)$ denote the position where x would reside in T if x had a tiny pointer with value i. Set $\ell = \delta^{-1}/32$. For each position $j \in [n]$ in the table, define the ***potential*** $\phi(j)$ to be

$$\phi(j) = \frac{|\{u \in \mathcal{U}, i \in [\ell] \mid h_i(u) = j\}|}{|\mathcal{U}|}.$$

Call an insertion ***safe*** if the element x that is inserted is inserted into one of positions $h_1(x), \dots, h_\ell(x)$. Call an insertion ***resource efficient*** if the element x that is inserted is inserted into a position j satisfying $\phi(j) \le \frac{4\ell}{n}$.

The probability that a given insertion is both safe and resource efficient is at most

$$\begin{aligned}
&\sum_{\substack{\text{empty position } j \in [n] \\ \phi(j) \le \frac{4\ell}{n}}} \sum_{i=1}^{\ell} \Pr_{x \in \mathcal{U}}[h_i(x) = j] \\
&= \sum_{\substack{\text{empty position } j \in [n] \\ \phi(j) \le \frac{4\ell}{n}}} \sum_{i=1}^{\ell} \frac{1}{|\mathcal{U}|} \sum_{x \in \mathcal{U}} \mathbb{I}_{h_i(x)=j} \\
&= \sum_{\substack{\text{empty position } j \in [n] \\ \phi(j) \le \frac{4\ell}{n}}} \phi(j) \\
&\le \sum_{\text{empty position } j \in [n]} \frac{4\ell}{n} \\
&= \delta n \frac{4\ell}{n} \\
&= \frac{1}{8}.
\end{aligned}$$

It follows that the expected number of insertions that are safe and resource efficient is at most $n^{c/2}/8$.

Next we bound the expected number of insertions A that are safe but not resource efficient. Rather than bound A directly, we instead examine the number of *deletions* B where the deleted element is deleted from a position j satisfying $\phi(j) > \frac{4\ell}{n}$. Note, in particular, that

$$A \le B + n.$$

By the definition of $\phi(j)$, we have that $\sum_{j=1}^{n} \phi(j) = \ell$. It follows that $|\{j \in [n] \mid \phi(j) > \frac{4\ell}{n}\}| \le n/8$. Each random deletion therefore has probability at most $\frac{n/8}{(1-\delta)n} \le 1/4$ of removing an element in a position j satisfying $\phi(j) > \frac{4\ell}{n}$. Thus $\mathbb{E}[B] \le n^{c/2}/4$ which means that

$$\mathbb{E}[A] \le n^{c/2}/4 + n \le n^{c/2}/2.$$

Since the expected number of insertions that are safe and resource efficient is at most $n^{c/2}/8$ and the expected number of insertions that are safe and resource inefficient is at most $n^{c/2}/2$, the expected number of insertions that are safe is at most $\frac{5}{8} n^{c/2}$. The expected number of insertions that are *not* safe is therefore at least $\frac{3}{8} n^{c/2}$. Each unsafe insertion results in a tiny pointer of length at least $\Omega(\log \ell) = \Omega(\log \delta^{-1})$ bits. Since a constant fraction of the insertions are expected to result in a tiny pointer of length at least $\Omega(\log \delta^{-1})$, we must have $\mu \ge \Omega(\log \delta^{-1})$. ∎

Next we prove a lower bound for fixed-sized tiny pointers, which shows that the bound in Theorem 177 is tight.

Theorem 178. *Consider a universe $\mathcal{U}$ of keys, where $\mathcal{U}$ is assumed to have a sufficiently large polynomial size. If a dereference table supports fixed-sized tiny pointers of size s and load factor $1 - \delta = \Omega(1)$, then $s = \Omega(\log\log\log n + \log \delta^{-1})$.*

It suffices to prove that $s = \Omega(\log\log\log n)$, since we have already shown that $s = \Omega(\log \delta^{-1})$.

The proof re-purposes a classic balls-and-bins lower bound. Say that a ball-placement rule is ***sequential*** if balls are placed sequentially, without knowledge of future ball arrivals, and if balls are never moved after being placed.

Theorem 185 (Theorem 2 in [351]). *Suppose that m balls are placed sequentially into m bins using an arbitrary sequential ball placement rule choosing d bins for each ball at random according to an arbitrary probability distribution on $[m]^d$. Then the number of balls in the fullest bin is $\Omega((\log\log m)/d)$ w.h.p.*

We now prove Theorem 178.

Proof of Theorem 178. Assume for contradiction that there exists a dereference table with load factor $1 - \delta = \Omega(1)$ and that supports fixed-size tiny pointers of size $s = o(\log\log\log n)$ bits. Let n be the number of slots in the dereference table, and let $m = (1 - \delta)n$ be the maximum number of allocations that the dereference table can support at a time; assume without loss of generality that $1/(1 - \delta) \in \mathbb{N}$, so n is a multiple of m. Finally, let $S = 2^s$, and observe that, by assumption, $S = o(\log\log n)$—and since $m = \Theta(n)$, $S = o(\log\log m)$.

Recall that $\mathcal{U}$ is the universe from which the keys are taken. For each key $x \in \mathcal{U}$, define the sequence $h_1(x), h_2(x), \ldots, h_S(x) \in [m]$ so that $h_i(x) = \lfloor \frac{m}{n} \text{Dereference}(x, i) \rfloor$. Note that, by the definition of the Dereference function, the sequence $h_1(x), h_2(x), \ldots, h_S(x)$ is a function of only on x, i, n, and the random bits of the dereference table—therefore, the sequence is predetermined by the coin flips, and is independent of the sequence of allocations/deallocations that are performed. Let $R \in [m]^S$ be a random variable obtained by selecting $x \in \mathcal{U}$ at random and setting $R = \langle h_1(x), h_2(x), \ldots, h_S(x)\rangle$; and let $\mathcal{R}$ be the probability distribution for R.

We will now construct a sequential ball-placement rule for mapping m balls to m bins. Our rule independently assigns each ball a random bin sequence $\langle h_1, h_2, \ldots, h_S\rangle \sim \mathcal{R}$ of S bins. Equivalently, we can think of the m balls as being m keys $x_1, x_2, \ldots, x_m$, where each x_i is selected uniformly and independently at random from $\mathcal{U}$, and each x_i has a bin sequence of $\langle h_1(x), h_2(x), \ldots, h_S(x)\rangle \in [m]^S$.

Since $|\mathcal{U}| \geq \text{poly}(n)$, we have that with high probability in n, the x_i's are distinct. Our ball placement rule uses our dereference table to decide where to place balls. To place ball x_i into a bin, we compute $p_i = \text{Allocate}(x_i)$, and we place x_i into the p_i-th bin in x_i's bin sequence, which is given by bin

$$h_{p_i}(x_i) = \left\lfloor \frac{m}{n} \text{Dereference}(x_i, p_i) \right\rfloor \in [m].$$

In summary, we have constructed a sequential ball placement rule that places m balls sequentially into m bins and that chooses a set of $d = S$ bins for each ball according to a probability distribution $\mathcal{R}$ over $[m]^d$. By Theorem 185, we can deduce that the fullest bin contains at least

$$\Omega\left((\log\log m)/d\right) = \Omega\left((\log\log m)/S\right) = \omega(1)$$

balls with high probability in m.

On the other hand, the dereference table guarantees that $\textsc{Dereference}(x_i, p_i) \in [n]$ is unique for each $i \in [m]$. The number of balls x_i satisfying

$$\left\lfloor \frac{m}{n} \textsc{Dereference}(x_i, p_i) \right\rfloor = j$$

for a given j is therefore at most $\frac{n}{m} = O(1)$. This means that the number of balls in any given bin is also $O(1)$. Since the dereference table succeeds with high probability in n, we can deduce that there are $O(1)$ balls in the fullest bin with high probability in n. This contradicts the fact that the number of balls in the fullest bin is $\omega(1)$, thereby completing the proof by contradiction. ∎

Chapter 17

Five Applications to Data Structures

In this chapter, we give five data-structural applications of tiny pointers. The first application is to the classic data-structural problem of storing a dynamic set of values associated with keys. The next three applications are each black-box transformations in which we show how to remove space inefficiency from large classes of data structures. And the final application is a new data structure for a classic problem in external-memory storage.

Overcoming the $\Omega(\log \log n)$-bit lower bound for the cost of data retrieval. Our first application revisits the classic ***data-retrieval problem*** [44, 146, 157, 159], in which a data structure must store a v-bit value for each of the k-bit keys in some set S, and must answer queries that retrieve the value associated with a given key.[1] In the static case, where the keys/values are given up front, it is possible to solve the retrieval problem with $O(1)$-time queries using $nv + O(\log n)$ bits of space [157, 159]; but in the dynamic case where keys/values are inserted/deleted over time, and there are up to n keys/value pairs present at a time (with keys taken from some large polynomial-size universe), it is known that any solution to the retrieval problem must use a lower bound of $nv + \Omega(n \log \log n)$ bits of space, even if super-constant-time operations are allowed [44, 146]. This means that the number of metadata bits per value is $\Omega(\log \log n)$ on average, even if the values are of size $v = o(\log \log n)$.

We show that, by just slightly modifying the specification of the retrieval problem, we can completely dissolve the $\Omega(\log \log n)$-wasted-bits-per-item lower bound. Suppose, in particular, that whenever the user inserts a key/value pair (x, y), they are given back a small *hint* h that they are responsible for storing. (We will guarantee that the hint has constant expected size.) In the future, when the user wishes to recover the value y for x, they present both the key x and the hint h to the retrieval data structure. We call this the ***relaxed retrieval problem*** and we refer to the hints as ***tiny retrievers***.

The relaxed retrieval problem can also be viewed as a relaxation of the tiny-pointer

[1]Note that queries are required to be for a key $x \in S$—the data structure is allowed to return an arbitrary value if $x \notin S$.

problem: the tiny retriever h is analogous to a tiny pointer, except that the pair (x, h) does not have to fully encode the position of y—instead, the relaxed-retrieval data structure can make use of not just x and h, but can also make use of a small auxiliary data structure whose purpose is to help recover y.

Given that we have already stated tight bounds for tiny pointers, it is tempting to assume that the same bounds should hold for tiny retrievers. We find that this is not so. We show how to construct tiny retrievers of expected size $O(1)$, while supporting queries in constant time (with high probability), and allowing for the following tradeoff curve: using time $\Theta(r)$ for insertions/deletions, the size of the data structure becomes $nv+O(n(1+\log^{(r)} n))$ bits. So, with constant-time operations, we can achieve size, say, $nv + O(n \log\log\log\log\log n)$, and with $O(\log^* n)$-time operations, we can achieve size $nv + O(n)$. Moreover, in the special case where the value length v is sub-logarithmic, satisfying $v \le \frac{\log n}{\log^{(r)} n}$, the space consumption reduces to $nv + O(n)$ bits, even for constant r.

Remarkably, our construction for tiny retrievers is *itself a direct application of tiny pointers*—in fact, tiny retrievers are simply variable-length tiny pointers of $O(1)$ expected size. This is because the ability to construct $O(1)$-length tiny pointers into an array with $\Theta(n)$ entries ends up allowing for us to reduce the relaxed retrieval problem to the dictionary problem, for which highly space-efficient solutions are known [85].

We remark that the distinction between tiny pointers and tiny retrievers ends up being significant in several of our applications below. In some cases, tiny retrievers offer a path to remarkable (and unexpected) space efficiency, while in other cases, the smooth tradeoff curve and pointer-like behavior offered by tiny pointers makes them a better fit. The advantage of tiny retrievers is that they offer a steep tradeoff between time and space; the advantage of tiny pointers is that they offer indirection-less reference to elements, as well as a flexible tradeoff between different *types* of space consumption (pointer size and load factor).

Succinct rotation-based binary search trees. To describe our second application, we first take a digression into the world of succinct binary trees. Since there are at most 4^n ordered binary trees on n nodes, the pointer structure of a binary tree can be encoded in $O(n)$ bits. This observation has led to a great deal of work on optimal (and near-optimal) encodings of binary trees [130, 142, 176, 184, 276, 281, 304, 313]. Apart from navigation, state-of-the-art trees also support a wide variety of query operations (e.g., subtree size [130, 176, 276, 281, 313], depth [130, 281], lowest-common ancestor [130, 281], level ancestor [130, 281], etc.), while also supporting basic dynamism (e.g., inserting/removing leaves [130, 176, 276, 281, 313], inserting a node in the middle of an edge [130, 176, 276, 281, 313], compacting a path of length two [130, 176, 276, 281, 313], etc.).

One natural form of dynamic operation has proven elusive, however: the known succinct binary trees do not efficiently support rotations. The lack of support for rotations is especially important for binary *search trees*, which store a set of n sortable keys in n nodes. Almost all dynamic balanced binary search trees (e.g., AVL trees [40], red-black trees [204], splay trees [329], treaps [49, 325], etc.) rely on rotations when modifying the tree. None of these tree structures can be encoded with the known succinct-tree techniques.

We give a randomized black-box approach for transforming dynamic binary search trees into succinct data structures. If there are n keys in the succinct search tree, each of which is k bits long, then the size of the succinct search tree will be $nk+O(n\log^{(r)} n)$ bits. The transformation induces only a constant-factor time overhead on query operations, and only an $O(r)$-factor time overhead on tree modifications. So, for example, if we set $r = O(\log^* n)$, then edge traversals take time $O(1)$, edge insertions/deletions take time $O(\log^* n)$, and the tree structure is encoded using $O(n)$ bits. In contrast, the previous state of the art [281] for implementing rotations in space-efficient binary search trees also encoded the tree structure in $O(n)$ bits (actually, $2n + o(n)$ bits) but required $\tilde{\Omega}(\log n)$ time to implement a single rotation.

When r is set to be $O(1)$, the fact that running times are preserved means that other properties, such as dynamic optimality, are as well. For example, if the splay tree [329] is dynamically optimal (as the widely believed Dynamic-Optimality Conjecture [329] posits), then so is the succinct splay tree.

Space-efficient stable dictionaries. Our third application is a black-box approach for transforming any key-value dictionary that stores its values in a fixed-capacity array into a stable dictionary with the same operation set and with only a constant-factor time overhead. If the original dictionary stores v-bit values, then the new stable dictionary also stores v-bit values, and uses $O(\log v)$ extra bits of space per value than does the original data structure.

Formally, a key-value dictionary (e.g., a binary search tree, hash table, etc.) is ***stable*** if whenever a key-value pair is inserted, the position in which the value is stored never changes. (This property is sometimes also referred to as referential integrity [320] or value stability [75].) Stability ensures that users can maintain pointers into a data structure without those pointers becoming invalidated by changes to the data structure [205, 320]. Stability is a strict requirement in many library data structures [132–139] (and it is a core reason why high-performance languages such as C++ use chained hashing [132,133], which is stable, instead of more space-efficient alternatives, such as linear probing [224, 309] or cuckoo hashing [161, 182, 299]).

Empirical research on achieving stability in space-efficient hash tables dates back to the 1980s [205,320] (see also discussion in Knuth's Volume 3 [226]) and the resulting techniques have been built into widely-used hash tables released by Google [38] and Facebook [173]. On the theoretical side, if a data structure is storing k-bit keys and v-bit values, where $k, v = O(\log n)$, it is known how to achieve stability at the cost of an extra $\Theta(\log\log n)$ bits of space per value [146], but it is not known whether $\Omega(\log\log n)$ bits per value are *necessary.*[2] Our result shows that it is not—stability can be achieved with $O(\log v)$ extra bits per value. This is especially noteworthy in

[2]Interestingly, there are several specific approaches for which $\Omega(\log\log n)$ bits per value are known to be necessary, for example if stability is achieved via perfect hashing (see Theorem 2 of [146]).

cases where the value-size v is small[3]. Our result applies to arbitrary fixed-capacity dictionaries, including, for example, the succinct splay tree constructed above.

Space-efficient dictionaries with variable-size values. Our fourth application is a black-box approach for transforming any key-value dictionary (designed to store fixed-size values) into a dictionary that can store different-sized values for different keys. The resulting data structure induces a constant-factor time overhead and offers the following guarantee on space efficiency. Let $\log^{(r)} n$ be the r-th iterated logarithm and set r to be a positive constant of our choice. The new data structure incurs an additive space overhead of only $O(\log^{(r)} n + \log |x|)$ bits for each value x. (Interestingly, the iterated logarithm $\log^{(r)} n$ in this application comes from an entirely different source than in our previous applications.)

The ability to store variable-length values also yields a simple solution to the ***multi-set problem***, which is the problem of how to design a space-efficient constant-time hash table that stores multi-sets of keys (rather than just sets). The multi-set problem was first posed as an open question by Arbitman et al. [52], who gave a succinct constant-time hash table capable of storing sets but not multi-sets. A series of subsequent works gave solutions to the multi-set problem, first in the case of random multi-sets [95], and then very recently for arbitrary multi-sets [96]. The known solutions come with a drawback, however: the bound on space is the same for duplicate keys as it is for non-duplicate keys. So, if there are m_i copies of some key, then they are permitted to take m_i times as much space as a single copy would, even though, in principle, $m_i - 1$ of the copies could be encoded using an $\log m_i$-bit counter. Our transformation gives a simple alternative solution that avoids this drawback and that can even be applied directly to the original hash table of Arbitman et al. [51]: by storing the multiplicity of each key as a (variable-length) value, one can support arbitrary multisets at an additional space cost of only $\log^{(r)} n + \log m_i + O(\log \log m_i)$ bits per key, where m_i is the multiplicity of the key and r is a positive constant of our choice; this is remarkably space efficient considering the fact that $\log m_i$ bits are needed just to store the multiplicity. A nice feature of our solution is that it also applies directly to other dictionaries such as, for example, the succinct splay tree discussed earlier in the section.

An optimal internal-memory stash. Our final application of tiny pointers revisits one of the oldest problems in external-memory data structures: the problem of maintaining a small internal-memory stash that allows for one to directly locate where elements reside in a large external-memory array.

In more detail, the problem can be described as follows [197]. We are given an (initially blank) external-memory array with $(1+\varepsilon)n$ slots, for some parameters ε, n. We must maintain a dynamically changing set S of key-value pairs (where keys are

[3]One especially remarkable consequence is the following: if we wish to store $O(1)$ control bits associated with each key in a data structure, and we wish for the positions of those bits to be stable so that a third party who does not have access to the data structure can still access/modify the control bits, then we can accomplish this with only $O(1)$ extra bits of space per item.

distinct) in the array, such that each time a key-value pair (x, y) is inserted into S, the pair (x, y) is assigned some permanent position where it resides in the external-memory array. We must then also maintain a small internal-memory data structure X, known as a ***stash***, that can be used to recover, for each key x, precisely where its key-value pair (x, y) is stored in the external-memory array. A stash enables queries to be performed in a *single* access to external memory.

Work on designing space-efficient and time-efficient stashes dates back to the late 1980s [197, 235, 236], and is also closely related to the problem of designing space-efficient page tables in operating systems [35, 36, 71]. The best-known theoretical results are due to Gonnet and Larson [197], who give a stash that uses only $O(n \log \varepsilon^{-1})$ bits of space. A consequence is that, if $\varepsilon = \Theta(1)$, the stash uses only $O(n)$ bits.

Gonnet and Larson's result comes with several drawbacks, however [197]. First, the stash only offers provable guarantees in the setting where insertions/deletions to S are random; in the case where S is modified by an arbitrary sequence of insertions/deletions/queries, the problem of designing a space-efficient stash remains open. Second, the internal-memory operations on the stash of [197] are not constant-time in the RAM model (or even constant expected time, when $\varepsilon = o(1)$).

By combining tiny pointers with modern techniques for constructing space-efficient filters, we show that it is possible to construct a stash of size $O(n \log \varepsilon^{-1})$ bits that supports constant-time operations in the RAM model (not just in expectation, but even with high probability) and that supports *arbitrary* sequences of insertions/deletions/queries.

17.1 Some General-Purpose Techniques for Using Tiny Pointers

Before diving into specific applications, we briefly discuss several preliminary definitions and techniques that will be useful in multiple of the applications.

Key-value dictionaries. Several of our applications will perform black-box transformations in order to add new features (namely, stability and variable-sized values) to key-value dictionaries. Formally, a ***key-value dictionary*** (often just called a ***dictionary***) is any data structure that stores key-value pairs (e.g., a hash table or a tree), where each key appears at most once. Typically, a key-value dictionary supports insertions, deletions, and queries, where queries, in particular, return the value associated to some key. Depending on the data structure, additional operations may also be supported, for example successor queries, which return the successor to some key.

We say that a key-value dictionary uses a ***value array*** if it designates some contiguous chunk of memory (that can be extended or shrunk over time) whose purpose is to store the values corresponding to keys. If values are k bits long, then the value array can be viewed as a array of k-bit objects.

In our applications, we will restrict ourselves to dictionaries that store their values in value arrays. For simplicity, we will assume that the dictionary uses a single value array, although all of our results can also easily be applied to a dictionary that makes use of many separately-allocated value arrays (as long as each individual value array is at least $\Omega(\log n)$ bits). The reason that we assume a single value array is because, to the best of our knowledge, all of the known space-efficient key-value dictionaries can easily be implemented in this format, so we choose to avoid introducing unnecessary complication to the results.

How to store value arrays of tiny pointers. A theme in several of our applications will be to modify a value array so that, rather than storing values directly, we instead store tiny pointers of some size k. Recall, however, that tiny pointers of size $k = o(\log \log \log n)$ bits are not fixed-size, meaning that some tiny pointers may require more than k bits. Nonetheless, if we are willing to use a value-array that is a constant-factor larger, then there is a simple trick, which we call ***chunked pointer storage***, that lets us interact with these variable-length tiny pointers in the same way that we would interact with fixed-length tiny pointers.

Break the value array into contiguous chunks of $O(\log n/k)$ tiny pointers. By Proposition 184, the total number of bits used by the tiny pointers in each chunk is $O(\log n)$ with high probability in n. Thus each chunk can be stored in $O(\log n)$ bits, meaning that the entire value array can be stored in $O(nk)$ bits.

There is, however, the remaining issue of how to efficiently access and modify the j-th tiny pointer in a given chunk. For each chunk, we can store an additional $O(\log n)$-bit bitmap where the bits that are set to 1 indicate the positions in the chunk where tiny pointers begin and end. To efficiently find the j-th tiny pointer, it suffices to find the j-th and $j+1$-th 1s in the bitmap. (The tiny pointer can then be extracted, modified, and reinserted, in constant time using standard bit manipulation on the bitmap and the chunk.) The problem of finding the j-th 1 in a $O(\log n)$-bit bitmap is easily solved with the method of four Russians [53]: simply store an auxiliary lookup table of size $\sqrt{n}$ that allows for such queries to be answered in a $(\log n)/2$-bit bitmap in a single lookup, and then perform $O(1)$ lookups to perform such a query in an $O(\log n)$-bit bitmap.

How to dynamically resize a data structure using tiny pointers. Several of our applications will also encounter the problem of using tiny pointers in a data structure whose size dynamically changes over time. Of course, this means that we must also dynamically resize dereference tables. Our applications will take the following approach, which we call ***zone-aggregated resizing***.

Consider a value array storing tiny pointers to k-bit items in a dereference table (and assume k bits fit in $O(1)$ machine words). Suppose that we wish to maintain the dereference table at a load factor of $1 - \Theta(1/k)$, that way the number of bits wasted per item stored is $O(1)$; note that this means that the tiny pointers in the value array are $\Theta(\log k)$ bits on average. Further suppose, however, that the value array dynamically changes size over time (meaning that elements must be added and removed from the dereference table). For our discussion here, we will assume that the value array itself is dynamically resized to always be at a load factor of at least $\Omega(1)$.

How can we update the dereference table to maintain a load factor of $1 - \Theta(1/k)$ while the number of items changes over time? Rather than just using a single dereference table, we use k dereference tables, and add $\Theta(\log k)$ bits to each tiny pointer in order to indicate which dereference table is being pointed into (this doesn't change the asymptotic size of the tiny pointers). We can grow and shrink the capacity (i.e., number of slots) of the dereference tables by either (a) rebuilding the smallest dereference table to double its size, or (b) rebuilding the largest dereference table to halve its size. If we assume for the moment that rebuilding a dereference table takes time proportional to the table's size, then the rebuilds can be de-amortized to take time $O(1)$ per operation (i.e., per modification to the dereference tables), while maintaining the desired load factor of $1 - \Theta(1/k)$.

The problem with rebuilding a dereference table is that all of the tiny pointers into that dereference table become invalidated. The actual construction of the new dereference table can easily be performed in linear time, but how do we update the tiny pointers in the value array? If the value array has size n, then the dereference table being rebuilt consists of only $\Theta(n/k)$ items. We want to identify where the tiny pointers to those items are in the value array in time $\Theta(n/k)$ rather than time $\Theta(n)$.

The solution to this issue is very simple: break the value array into contiguous ***zones*** each of which consists of k values. Within each zone, maintain k linked lists, where the i-th linked list contains the tiny pointers that point into the i-th dereference table. Importantly, because these linked lists are within a zone of size k, the pointers *within* each linked list only require $\Theta(\log k)$ bits each; thus the linked lists do not asymptotically increase the size of the value array. On the other hand, in order to find all of the tiny pointers for a given dereference table, one can simply look at one linked list in each of the $\Theta(n/k)$ zones, allowing for all $\Theta(n/k)$ of the tiny pointers to be identified in time $\Theta(n/k)$.

For reasons that we shall see later, one of our applications will also require us to use larger zones of size $\text{poly}(k)$ rather than just of size k. For now, we simply remark that using larger zones of size $\text{poly}(k)$ still allows for the linked-list overhead of each tiny pointer to be bounded by $\Theta(\log k)$ bits, and that the time needed to identify the tiny pointers to a dereference table of size j is only

$$O(j + n/\operatorname{poly}(k)), \tag{17.1}$$

since the number of linked lists that must be examined is only $O(n/\operatorname{poly}(k))$.

17.2 Overcoming the $\Omega(\log \log n)$-Bit Lower Bound for Data Retrieval

Our first application revisits the classic retrieval problem [44, 146, 157, 159], in which a data structure must store a v-bit value for each of the k-bit keys in some set S, and must answer queries that retrieve the value associated with a given key. Here, we address the dynamic version of the problem, where the data structure must support the functions $\textsc{Insert}(x, y)$ (which inserts a new $x \in [2^k]$ into S and associates it with value $y \in [2^v]$), $\textsc{Delete}(x)$ (which removes some $x \in S$ from S), and $\textsc{Query}(x)$ (which returns the value y corresponding to x for some $x \in S$), allowing for the set S to grow up to some maximum size n. Note that, in the retrieval problem, it is the *user's responsibility* to ensure that every invocation of $\textsc{Insert}$ is on a key $x \notin S$, every invocation of $\textsc{Query}$ is on a key $x \in S$, and every invocation of $\textsc{Delete}$ is on a key $x \in S$.

It is known that, if $k = (1 + \Omega(1)) \log n$ bits, then any solution to the dynamic retrieval problem must use at least $nv + \Omega(n \log \log n)$ bits of space [44], regardless of the time complexity, and even if $v = 1$. It is further known that, if $k = \Theta(\log n)$ and $v = O(\log n)$, then the $nv + \Theta(n \log \log n)$ space bound can be accomplished by a randomized constant-time data structure [146].

We will now show that, by slightly relaxing the retrieval problem, we can use tiny pointers to obtain significantly better space bounds. In the ***relaxed retrieval problem***, the insertion/deletion/query operations are modified to work as follows. The operation $\textsc{Insert}(x, y)$ now returns a ***tiny retriever*** r which the user must remember. In the future, if the user wishes to query x (and they have not yet deleted x), they call $\textsc{Query}(x, r)$ to obtain the value y. Finally, if the user ever wishes to remove x from the set S, then the user calls $\textsc{Delete}(x, r)$.

The role of the tiny retriever is similar to that of a tiny pointer—it acts as a hint to assist the data structure. Unlike for tiny pointers, however, the pair (x, r) does not have to fully encode the position of y; instead, query operations $\textsc{Query}(x, r)$ can use auxiliary metadata, beyond just x and r, to determine the value y. We shall now see that this distinction is very important, allowing for us to do better than *both* the lower bound for the retrieval problem [44] and our lower bound for the tiny-pointer problem (Theorem 180). At the same time (almost paradoxically), it is our construction for variable-size tiny pointers that allows for us to get around both of these lower bounds.

Theorem 186. *Consider the relaxed retrieval problem with k-bit keys, v-bit values, and a maximum capacity of n key/value pairs. Let $r \in [\log^* n]$ be a parameter. There is a solution to the relaxed retrieval problem that uses tiny retrievers of expected size $O(1)$, and that with high probability in n: takes constant time per query, takes $O(r)$ time per insertion/deletion, and uses total space $nv + O(n \log^{(r)} n)$ bits.*

Furthermore, if $\log^{(r)} n = \omega(1)$ and $v \le \frac{\log n}{\log^{(r)} n}$, then the space consumption becomes $nv + O(n)$ bits.

The above theorem comes with an interesting tradeoff curve: constant-time insertions/deletions can achieve a space consumption of, for example, $nv + O(n \log\log\log\log\log n)$ bits, and $O(\log^* n)$-time insertion/deletions can achieve space consumption $nv + O(n)$ bits. Moreover, if v is slightly sub-logarithmic, then even constant-time insertions/deletions can achieve $nv + O(n)$ bits.

We remark that the tiny retrievers in Theorem 186 are, in fact, variable-size tiny pointers as constructed in Theorem 179. They therefore satisfy the doubly-exponential tail inequality given by Theorem 179, as well as the concentration inequality given by Proposition 184.

Proof. We shall make use of Theorem 179 to construct a dereference table T with $2n$ slots. What makes our application of Theorem 179 unusual, however, is that we will not store anything in the store (if fact, we need not even allocate space for it). Instead, we will take advantage of the fact that $\textsc{Dereference}(x, p)$ is a $(1+\log n)$-bit number that has been uniquely allocated to x.

To implement the operation $\textsc{Insert}(x, y)$, we call $\textsc{Allocate}(x)$ to obtain a tiny pointer p of expected size $O(1)$ (note that p will also be our tiny retriever). Define $s_x = \textsc{Dereference}(x, p)$ to be the slot number in $[2n]$ allocated to x. The main property that we will exploit is that $s_x \neq s_{x'}$ for all other $x' \in S$. To complete the $\textsc{Insert}$ operation, we insert the key/value pair (s_x, y) into a succinct hash table H (whose specifications we will describe later). Queries and deletes are then implemented as follows: $\textsc{Query}(x, p)$ returns $H[\textsc{Dereference}(x, p)]$; and $\textsc{Delete}(x, p)$ deletes key $\textsc{Dereference}(x, p)$ from H and calls $\textsc{Free}(x, p)$ on the dereference table T.

The correctness of the data structure follows from the fact that, for each $x \in S$ with tiny retriever p, $\textsc{Dereference}(x, p)$ is unique. The dereference table uses space only $O(n)$ bits and supports constant-time operations (with high probability). Thus, to prove the theorem, it remains to analyze the hash table H.

We construct H using the most space-efficient known construction for a hash table [85]. If H is storing up to n keys from a universe U and values are v bits, then it supports the following guarantees with high probability: queries are constant-time, insertions/deletions take time $O(r)$, and the total space consumption is

$$\log \binom{|U|}{n} + nv + O(n \log^{(r)} n)$$

bits. If, in addition, $\log^{(r)} n = \omega(1)$ and $v \leq \frac{\log n}{\log^{(r)} n}$, then the space becomes $\log \binom{|U|}{n} + nv + O(n)$ bits.

Our use of tiny pointers ensures that the keys in H are from the very small universe $U = [2n]$. So

$$\log \binom{|U|}{n} = \log \binom{2n}{n} = O(n)$$

by Stirling's approximation. This completes the proof of the theorem. ∎

A remark on resizing. In Subsection 17.3, we shall see an application of tiny retrievers to the problem of constructing succinct binary search trees. In this application, we will want to have two relaxed-retrieval data structures whose sizes sum to at most n. Here, we can take advantage of the fact that the hash table H used above actually offers a dynamically-resizing guarantee: if, at any given moment, the hash table has size m, then it uses space at most

$$\sqrt{n} + \log \binom{2n}{m} + mv + O(m \log^{(r)} n),$$

with high probability in n. The full retrieval data structure (consisting of the hash table H and the dereference table T) therefore uses space at most

$$\log \binom{2n}{m} + mv + O(n + m \log^{(r)} n).$$

By Stirling's inequality, this is at most

$$m \log n - m \log m + mv + O(n + m \log^{(r)} n).$$

Thus, if we have two relaxed-retrieval data structures, one of size $m_1 \le n$ and one of size $m_2 \le n$, and $m = m_1 + m_2 = \Theta(n)$, then their total space consumption will be at most

$$\begin{aligned} &(m_1 + m_2) \log n - m_1 \log m_1 - m_2 \log m_2 + (m_1 + m_2)v + O((m_1 + m_2) \log^{(r)} n) \\ =\, &m \log n - m_1 \log m_1 - m_2 \log m_2 + mv + O(m \log^{(r)} n). \end{aligned}$$

By Jensen's inequality, $m_1 \log m_1 + m_2 \log m_2 \ge (m_1 + m_2) \log \frac{m_1+m_2}{2} = m \log \frac{m}{2} = m \log n - O(n)$. Thus the total space is at most

$$\begin{aligned} &m \log n - (m \log n - O(n)) + mv + O(m \log^{(r)} n) \\ &= mv + O(m \log^{(r)} n) \\ &= mv + O(m \log^{(r)} m) \end{aligned}$$

This, of course, is the same bound that we get for a single relaxed-retriever data structure of size m.

The reason that this matters is that it allows for a simple way to perform dynamic resizing: every time that the size m of a data structure changes by a factor of two, we move all of the elements in the current relaxed-retrieval data structure D_1 into a new relaxed-retrieval data structure D_2 (parameterized as having capacity $n = \Theta(m)$ based on the new value of m). As we move elements from D_1 to D_2, the total space consumption of D_1 and D_2 will continue to be $mv + O(m \log^{(r)} m)$ bits. Note that, to move an element from D_1 to D_2, we will need to generate a new tiny retriever for that element (since we are deleting the element from D_1 and inserting it into D_2). In our binary-search-tree application, this will be easy to do by simply running through all of the elements and relocating them one by one. Furthermore, since the work of constructing D_2 can be spread across $\Theta(n)$ operations, it can be achieved at a cost of $O(r)$ per insertion/deletion.

17.3 Succinct Binary Search Trees

Our second application is a black-box approach for transforming dynamic binary search trees into succinct data structures. If there are n elements in the succinct search tree, each of which is k bits long, then the size of the succinct search tree will be at most $nk + O(n + n\log^{(r)} n)$ bits, where $r > 0$ is an arbitrary parameter. Path traversals in the tree incur only a constant-factor overhead, and modifications to the tree incur only an $O(r)$-factor overhead.

An advantage of our approach is that it can be applied to rotation-based search trees. This includes, for example, red-black trees [204], splay trees [329], etc. If the dynamic-optimality conjecture [329] is true, meaning that the splay tree is dynamically optimal, then our succinct splay tree is also dynamically optimal when $r = O(1)$.

Theorem 187. *Consider any binary search tree storing a-bit keys and b-bit values, where every node is associated with a distinct key, and where each node has pointers to its children. For any $r > 0$, the tree can be implemented to offer the following guarantees with high probability in the tree size n: the tree takes space $na+nb+O(n+n\log^{(r)} n)$ bits, traversals from parents to children take time $O(1)$, and modifications to the tree (i.e., adding or removing an edge) take time $O(r)$.*

We remark that, information theoretically, the tree use consume $n(a + b)$ bits of space. And since the keys are distinct, $na = \Omega(n \log n)$. Thus, for any $r > 1$, the search tree above is succinct.

Proof. We will make use of our solution to the relaxed retrieval problem (Theorem 186). However, the key/value pairs (x, y) that we will store in the relaxed-retrieval data structure will be a bit unusual in that y will take the following form: y contains x's b-bit value, along with two tiny retrievers r_1 and r_2. Since r_1 and r_2 are themselves variable-length tiny pointers of expected size $O(1)$, this means that y is also variable-length. On the other hand, the relaxed-retrieval data structure is designed for *fixed-length* values. Fortunately, we can store the tiny retrievers r_1 and r_2 with the following method. Recall that, in our construction for the relaxed retrieval problem, we create a dereference table with $2n$ slots, but we do not actually store anything in the dereference table's store. We now change this so that the store is a value array with $2n$ slots that stores the tiny retrievers r_1 and r_2 for each item in the dereference table (so, if p is the tiny pointer for x, then r_1, r_2 are in the $\textsc{Dereference}(x, p)$-th position of the value array). Using the chunked pointer storage technique, we can ensure that the total size of the value array is $O(n)$ bits, even though the pointers that it stores are variable length.

We now describe our encoding of the binary search tree: Each node in the search tree stores the key-value pair (x, y) corresponding to that node along with two tiny retrievers r_1 and r_2. The tiny retriever r_1 is for the left child and uses $x \circ 0$ as its key (so $\text{Query}(x \circ 0, r_1)$ returns the left child of x), and the tiny retriever r_2 is for the right child and uses $x \circ 1$ as its key (so $\text{Query}(x \circ 1, r_1)$ returns the right child of x). Note that, if the left child (resp. right child) does not exist, then we simply set r_1 (resp. r_2) to null.

Let us begin by assuming that our binary search tree has a fixed capacity of n keys/values, so we can use a relaxed-retrieval data structure with capacity n. Then our relaxed-retrieval data structure uses $na + nb + O(n + n\log^{(r)} n)$ bits. Navigating from a node to its child takes time $O(1)$ (since it requires a single query to the relaxed-retrieval data structure) and adding/removing an edge (x, z) from a node x to a child z takes time $O(r)$, with high probability, since it requires only a single insert/delete to the relaxed-retrieval data structure; importantly, if z is the root of some subtree, the act of setting z to be x's child *does not* require any nodes besides z to inserted/deleted in the relaxed-retrieval data structure.

Finally, let us modify our data structure so that it dynamically resizes as a function of the current number n of key/value pairs. For this, we can simply use the resizing approach outlined in Section 17.2. Every time that n changes by a constant factor, we rebuild the relaxed-retrieval data structure to have capacity $\Theta(n)$ for the new value of n. (Note that this does not require us to rebuild the tree; it just requires us to update the tiny retrievers used in each node.) For each relaxed retriever in the binary search tree, we can store an extra bit indicating which of the two relaxed-retrieval data structures it uses—this preserves correctness. As observed in Section 17.2 the act of moving items from the old relaxed-retrieval data structure to the new one does not violate our desired space guarantee: the total number of bits used by our search tree remains $na + nb + O(n + n\log^{(r)} n)$ at all times. And, by spreading the work of rebuilding the relaxed-retrieval data structure across $\Theta(n)$ operations, we maintain the property that each edge insertion/deletion takes time $O(r)$. Thus the theorem is proven. ∎

17.4 Space-Efficient Stable Dictionaries

Using tiny pointers, we give a black-box approach for transforming any fixed-capacity key-value dictionary into a ***stable*** dictionary, meaning that the position in which a value is stored never changes after the value is inserted. If the original dictionary stored k-bit values, then the new dictionary also stores k-bit values, and uses at most $O(\log k)$ extra bits of space per value than the original data structure.

Theorem 188. *Consider a fixed-capacity key-value dictionary data structure T that stores its values in a value array of some size m. Let v denote the size of each value in bits.*

It is possible to construct a new data structure T' with the same operations and asymptotics (with high probability) as T, but with the additional property that T' is stable. Moreover, the total space consumed by T' is guaranteed (with high probability in m) to be at most $O(m \log v)$ more bits than T.

Proof. To construct T', we simply replace the value array for T with an array of m tiny pointers, each of size $\Theta(\log v)$ bits. (If $\log v < \log\log\log n$, then the chunked-storage technique can be used to handle the fact that different tiny pointers have different sizes.) The tiny pointers point into a dereference table of size $(1 + 1/v)m$ that stores the m v-bit values. (So the load factor is $1 - \Theta(1/v)$.) If a tiny pointer points at the value y corresponding to a key x, then the tiny pointer uses x as its key. This ensures stability, since even if the location in which the tiny pointer is stored changes, the tiny pointer does not have to change (and the value y does not have to move).

The array of tiny pointers consumes $O(m \log v)$ space. Whereas the value array in T consumes mv bits, the dereference table in T' consumes $(1 + 1/v)mv$ bits, which is only $O(m)$ more bits then used in T. Thus the claim on space efficiency is proven. Since tiny pointers only add constant time per access/modification of the value, the asymptotics are (with high probability in m) the same for both T and T'. ∎

17.5 Space-Efficient Dictionaries with Variable-Size Values

Our fourth application is a black-box approach for transforming any key-value dictionary (designed to store fixed-size values) into a dictionary that can store different-sized values for different keys. The resulting data structure offers the following remarkable guarantee on space efficiency. Let $\log^{(r)} n = \log\log\cdots\log n$ denote the r-th iterated logarithm of n. Let r be a positive constant of our choice, and let m be the number of entries in the value array used by the original dictionary (at some given moment). The new dictionary, which allows for values to be arbitrary lengths, replaces the value array for T with a data structure that consumes at most

$$O(m \log^{(r)} m) + \sum_{i=1}^{m} (v_i + O(\log v_i))$$

bits, where $v_1, v_2, \ldots, v_m$ denote the lengths in bits of the values being stored.

Theorem 189. *Consider a key-value dictionary data structure T that stores its values in a value array, and that is designed to store fixed-length keys. Let r be a positive constant of our choice.*

It is possible to construct a new data structure T' with the same operations and asymptotics (with high probability) as T, but with the additional property that T' can store values of arbitrary lengths (up to $O(1)$ machine words).

At any given moment, if T would have been using a value array of size m, and the machine word size w satisfies $w \le m^{o(1)}$, then the total space consumed by T' to *implement the value array is guaranteed (with high probability in m) to be at most*

$$O(m \log^{(r)} m) + \sum_{i=1}^{m} (v_i + O(\log v_i)) \tag{17.2}$$

bits, where $v_1, v_2, \ldots$ are the sizes of the values.

We remark that the limitation on value-size to be $O(1)$ machine words is simply so that each value can be written/read in constant time, that way it is easy to discuss how the asymptotics of T and T' compare. The same techniques work for even larger values without modification, as long as one is willing to spend the necessary time to read/write values that are of super-constant size.

Proof of Theorem 17.2. Values in T' are stored with up to r levels of indirection. If a value is k bits, then it is pointed at by a tiny pointer p_1 of size $O(\log k)$ bits. The tiny pointer p_1 is, in turn, pointed at by a tiny pointer p_2 of size $O(\log \log k)$ bits, and so on, with pointers of size $O(\log \log \log k), O(\log \log \log \log k), \ldots, O(\log^{(r)} n)$. That is, every value is stored at the end of a linked list of length $O(1)$, where the base pointer of the linked list is $O(\log^{(r)} n)$ bits, and each subsequent pointer is exponentially larger than the previous one.

For each tiny pointer of some size j in the data structure, we must also store $O(j)$ extra bits of information indicating (a) whether the tiny pointer is pointing at another tiny pointer or at a final value, and (b) what the size is of the tiny-pointer/value being pointed at. Throughout the rest of the proof, we will count these $O(j)$ extra bits as being part of the size of the tiny pointer.

Since there are both values and tiny pointers of many different sizes, we must use a different dereference table for each size-class of tiny-pointer and the different dereference table for each size-class of values being stored. (Note that the dereference tables storing tiny pointers may need to use the chunked-storage technique to handle variable-sized tiny pointers, so the same dereference table should not be used to store both tiny pointers and values.)

The problem of dynamically resizing all of the dereference tables simultaneously is slightly tricky. Consider a dereference table A (to A could also be the value array) that stores j-bit tiny pointers for some j. There are $K = 2^{\Theta(j)}$ different dereference tables $B_1, B_2, \ldots, B_k$ that these tiny pointers can point into (depending on the size of the object being pointed at, and whether the object is a tiny pointer or a value). Each B_i must individually be dynamically resized. We will maintain what we call the ***dynamic-sizing invariant***, which guarantees that each B_i is either (a) at a load factor $1 - O(1/j')$, where j' is the size of the objects stored in B_i, or (b) is at most a $o(1/(Kj))$-fraction the size (in bits) of A.

To implement the dynamic-sizing invariant, we dynamically resize each B_i using zone-aggregated resizing (recall from Section 17.1 that this means B_i is broken into multiple components, and each component is occasionally rebuilt so that its size either doubles or halves). To allow for components of each B_i to be rebuilt efficiently, we break A into zones of size $\text{poly}(K)$, meaning by (17.1) from Section 17.1 that a given component (of some B_i) consisting of s entries can be rebuilt in time

$$|A|/\operatorname{poly}(K) + s,$$

where $|A|$ is the number of entries in A. We perform dynamic resizing on B_i differently depending on whether it is very small (its components contain fewer than $|A|/\operatorname{poly}(K)$ elements each) or not:

- If the components contain $s = \Omega(|A|/\operatorname{poly}(K))$ elements each, then we perform zone-aggregated resizing (exactly as in Section 17.1) to keep B_i at a load factor $1 - O(1/j')$, where j' is the size of the objects stored in B_i. In this case, the time needed to rebuild a component of size s is $\Theta(s)$, so the dynamic resizing of B_i can be deamortized to take $O(1)$ time per operation (on B_i). Note that, here, B_i is in case (a) of the dynamic-resizing invariant.

- If the components contain fewer than $|A|/\operatorname{poly}(K)$ elements each, then we perform zone-aggregated resizing to keep each component of B_i at a capacity of $\Theta(|A|/\operatorname{poly}(K))$ (even as $|A|$ changes over time, and *regardless* of whether the number of elements per component may be significantly smaller than $|A|/\operatorname{poly}(K)$). Note that, here, B_i is in case (b) of the dynamic-resizing invariant.

  When B_i is in this regime, we cannot amortize the work spent rebuilding B_i to the operations that are performed on B_i. Instead, we spread out the work spent rebuilding components of B_i in the following way: for every $\Theta(K)$ work that is spent on A we also spend $O(1)$ time on resizing B_i. Since B_i is more than a factor of K smaller than A, this is sufficient time to keep B_i in a state where each component has capacity $\Theta(|A|/\operatorname{poly}(K))$.

  From the perspective of A, every time that we spend constant time on insertions/deletions/rebuilding A, we also may spend constant time performing rebuild-work on one of the B_is (which, in turn, may recursively lead us to spend constant time on rebuilding one of the dereference tables pointed at by B_i, etc.). Importantly, since chains of tiny pointers are at most $r \leq O(1)$ long, the time spent on rebuilds only introduces a constant-factor overhead on running time per operation.

The resizing approach described above guarantees the dynamic-sizing invariant while incurring only a constant-factor time overhead per operation. Next we use the invariant to bound the space consumption of T'. The dereference tables B_i in case (a) are implemented space-efficiently enough that the empty slots in them take negligible space compared to the actual objects stored in them (i.e., the empty slots add $O(1)$ bits per object), and the dereference tables B_i in case (b) are small enough that they take negligible space compared to the size of the parent dereference table A (i.e., they cumulatively add $o(1)$ bits per slot in A). It follows that the total space consumed by dereference tables will be at most the sum of the sizes of the objects being stored in the dereference tables, plus $O(1)$ bits per object; this, in turn, means that the space used by T' to store values/tiny pointers is given by (17.2).

Next, we bound the time-overhead of T' when compared to T. We have already shown that the time-overhead of performing dynamic-resizing on dereference tables is $O(1)$ per operation. Since values are stored with at most $r = O(1)$ levels of indirection, the time needed to access/modify a value is also $O(1)$. Thus T' has the same time asymptotics as T.

Finally, we argue that the dereference tables used by T' succeed at their allocations with high probability.[4] There are several approaches that we could take to doing this; the simplest is to just add one more modification to how we perform dereference-table resizing: whenever a dereference table gets down to size $\Theta(\sqrt{m})$, we do not ever resize it to be any smaller.[5] This means that some dereference tables could be very sparse, containing $\sqrt{m}$ slots, but containing far fewer elements. Since there are only $O(w) = m^{o(1)}$ different dereference tables (recall that w is the machine-word size), the net space consumption of the dereference tables of size $\Theta(\sqrt{m})$ is $o(m)$ bits. The fact that every dereference table has size at least $\Omega(\sqrt{m})$ means that all of the dereference tables offer high probability guarantees, as desired. ∎

17.6 An Optimal Internal-Memory Stash

Our final application of tiny pointers revisits one of the oldest problems in external-memory data structures: the problem of maintaining a small internal-memory ***stash*** that allows for one to locate where elements reside in a large external-memory data structure.

The problem can be formalized as follows. We must store a dynamically changing set S of up to n key-value pairs, where each key-value pair can be stored in one machine word, and where each key is unique. We are given an ***external memory*** consisting of $(1+\varepsilon)n$ machine words, where the key-value pairs S are to be stored. In addition to storing key-value pairs in external memory, we must maintain a small internal-memory data structure X, which we will refer to as the ***stash***, that supports the following operations:

- **Query**(k)**:** Using only information in the stash data structure, returns the position in external memory where the key k and its corresponding value v are stored.

- **Insert**(k, v)**:** Inserts the key-value pair (k, v), placing the pair somewhere in external memory, and updating the stash.

- **Delete**(k, v)**:** Removes the key/value pair (k, v) from the external-memory array, and updates the stash.

[4]There are many different ways that one could handle allocation failures, including, for example, performing batch-rebuilds of the data structure.

[5]However, since m may dynamically change over time, we do need to spend constant time per operation resizing dereference tables of size $\Theta(\sqrt{m})$ so that they stay size $\Theta(\sqrt{m})$ as m changes.

The important feature of a stash is that queries can be completed with a single access to external memory. On the other hand, in order for a stash to be useful, several other objectives must be achieved:

- **Compactness:** The stash X needs to be as small as possible, that way it can fit into an internal memory with limited size.
- **Efficient inserts and deletes:** Although a stash prioritizes queries, insertions and deletions should ideally also require only $O(1)$ accesses/modifications to external memory.
- **RAM efficiency:** Finally, so that computational overhead does not become a bottleneck, the operations on a stash should be as efficient as possible in the RAM model, ideally taking time $O(1)$.

A concrete example of a stash that is used in real-world systems is the ***page table*** [35, 36, 71], which is an operating-system-level dictionary that maps virtual page addresses to where their corresponding physical pages reside in memory. The page table is accessed for every address translation, so it is performance critical and thus highly optimized. Additionally, it is important that the page table be space-efficient, so that it may be effectively cached in the processor cache hierarchy. Note that, although page tables get to select where physical pages reside in memory, they do not get to move physical pages that have already been placed; thus any stash that is used as a page table must also be stable. For this reason, past work [197, 235, 236] has typically included stability as an additional criterion for a stash.

Work on designing space-efficient and time-efficient stashes dates back to the late 1980s [197, 235, 236]. The best-known theoretical results are due to Gonnet and Larson [197], who give a stable stash that uses only $O(n \log \varepsilon^{-1})$ bits. A remarkable consequence of this is that, when $\varepsilon = \Theta(1)$, it is possible to construct a stash using only $O(n)$ bits.

Gonnet and Larson's result comes with several significant drawbacks, however [197], which have proven difficult to fix. First, due to its reliance on stable uniform probing [234] as a mechanism for determining where keys/values should reside, the stash only offers provable guarantees in the setting where insertions/deletions are performed *randomly*. Second, the data structure is not constant-time in the RAM model, instead taking expected time $\Theta(\varepsilon^{-1})$.

Using tiny pointers, we show that modern techniques for constructing filters can easily be adapted in order to construct a stable stash of size $O(n \log \varepsilon^{-1})$ bits that supports constant-time operations in the RAM model (with high probability) and that supports arbitrary sequences of insertions/deletions/queries.

Theorem 190. *It is possible to construct a stable stash that supports constant-time operations in the RAM model, that stores up to m keys/values in an external-memory array of size $(1+\varepsilon)m$, and that uses only $O(m \log \varepsilon^{-1})$ bits of internal-memory space. All of the guarantees for the stash hold with high probability in m.*

Proof. The starting point for our design is the adaptive filter of Bender et al. [82]. Like a stash, their filter is a space-efficient internal-memory data structure that summarizes the state of an external-memory key-value dictionary. Unlike a stash, their filter does not indicate where in external memory each key/value is stored. Instead, the filter answers containment queries with the following guarantee: each positive query is guaranteed to return true, and each negative query is guaranteed to return false with probability at least $1 - \varepsilon$ (for some parameter ε). The size of their internal-memory data structure is only $(1 + o(1))m \log \varepsilon^{-1} = O(m \log \varepsilon^{-1})$ bits, where m is the capacity of the filter.[6]

The basic idea behind the adaptive filter of [82] is to store a ***fingerprint*** for each key x, where each fingerprint is taken to be some prefix of the hash $h(x)$. Different keys have different-length fingerprints, and the invariant maintained by the filter is that no fingerprint is a prefix of any other fingerprint. To maintain this invariant while also keeping the fingerprints as small as possible, the filter will sometimes change the lengths of $O(1)$ different fingerprints during a given insertion/deletion; to change the length of a fingerprint, the key corresponding to that fingerprint must first be fetched from external memory, that way the hash $h(x)$ of that key can be recomputed.[7]

The fingerprints in the filter are stored as follows. The first $\lg n$ bits of each fingerprint are called the ***quotient***, and these bits are used to assign the key to one of n bins; importantly, the fact that the bin-choice encodes the quotient of each of the keys in the bin means that the data structure does not have to explicitly store the quotients of the fingerprints. The next $\log \varepsilon^{-1}$ bits of each fingerprint are called the ***baseline bits***, and these bits are included for every fingerprint in the data structure. Finally, any subsequent bits in a fingerprint are called the ***adaptivity bits***, and these bits are added/removed in order to maintain the prefix-freeness invariant. A central piece of [82]'s analysis is to show that there are only $O(m)$ adaptivity bits in total, and that these bits can be stored efficiently.

We now describe how to modify the filter to be a stash. In addition to storing a fingerprint for each key, we now also store a tiny pointer with expected size $\Theta(\log \varepsilon^{-1})$. These tiny pointers are easy to store, since the filter has already made room for $\log \varepsilon^{-1}$ baseline bits for each key. Of course, different tiny pointers may have different lengths, but this issue can easily be resolved by either using the chunked-storage technique described in Section 17.1 (or by adapting the techniques already used in [82] to handle variable-length fingerprints).

One minor difficulty is that the filter assumes access to an external-memory dictionary (rather than just a dereference table) that way it can lookup keys in order to modify their fingerprints. In the case of our stash, however, these lookups can easily be performed using the tiny pointers that are already stored, so one does not need a full dictionary in external memory.

The fact that the tiny pointers have size $\Theta(\log \varepsilon^{-1})$ means that external memory

[6]In fact, their data structures also dynamically resizable, but for our application that will not be necessary.

[7]The original data structure also sometimes updates the lengths of fingerprints during negative queries, but such updates are not needed for the purposes of our data structure.

can be implemented as a dereference table with load factor $1 - \varepsilon$. The fact that the original adaptive filter supported constant-time operations (with high probability in m) translates to the stash also supporting constant-time operations. And the fact that the original adaptive filter used space $O(m \log \varepsilon^{-1})$ bits in internal memory also translates the same guarantee for the stash. Thus the theorem is proven.

Part VI

A Strong Theory of Strong History Independence

Chapter 18

Introduction

A data structure ALG is said to be ***strongly history independent*** (or ***uniquely representable***) if its state is *fully determined by its current set of elements.* That is, given the current set S of elements that the data structure stores, and given the random bits R that the data structure uses, one can reconstruct the data structure $\text{ALG}(S, R)$. Critically, such a data structure *cannot* depend on the order in which elements were inserted, or on the history of what other elements were present in the past. This makes history independence an appealing *security guarantee* [70, 105, 106, 119, 194, 209, 263, 278, 279]. If an adversary obtains the states of the data structure at some set of times $\{t_1, \ldots, t_k\}$, then all that the adversary learns about the history of the data structure is what the data structure contained at each of those times. The adversary cannot deduce anything further about what sequences of insertions/deletions occurred *between* the times.

Although this part of the thesis will be primarily about strong history independence, it is worth also mentioning the closely related notion of *weak* history independence [70, 74, 119, 209, 263, 279], which says that the current set S of elements fully determines the *probability distribution* $\mathcal{ALG}(S)$ from which the current data structure state is drawn. Weak history independence ensures that, if an adversary obtains the data structure at a *single* point in time, then all that they learn is which elements are present. However, if the adversary obtains the data structure at multiple points $t_1 < t_2$ in time, then the *relationship* between the states of the data structure at those points may leak additional information about what occurred in the time interval $[t_1, t_2]$.

Variable-Size Stateless Allocation. One of the most basic algorithmic questions surrounding strong history independence is that of ***Variable-Size Stateless Allocation*** [194,200,279], which can be viewed as the strongly history-independent analogue of classical memory allocation. Given a set $S \subseteq [U]$ and a size function $\pi : S \to \mathbb{N}$ satisfying $\sum_{x \in S} \pi(x) \leq (1 - \varepsilon)n + 1$, one must construct an allocation ϕ_S mapping the elements $x \in S$ to disjoint intervals $\phi_S(x) \subseteq [0, n]$ of size $|\phi_S(x)| = \pi_S(x)$. We can think of ϕ_S as a type of memory allocator, mapping each element x a memory chunk of size $\pi(x)$ in an array of size n.

The goal is to minimize the *overhead* of performing an insertion/deletion on S. If an insertion, transforms input (S, π) into input (S', π'), adding a new element x of some size $\pi'(x) = r$, then the ***overhead*** of the insertion is given by

$$1 + \sum_{\substack{a \in S \\ \phi_{(S,\pi),R}(a) \neq \phi_{(S',\pi'),R}(a)}} \frac{\pi(a)}{r}, \tag{18.1}$$

i.e., $1/r$ times the sum of the sizes of the elements (including x) whose allocations change due to the insertion. One should think of (18.1) as measuring the multiplicative overhead of performing the insertion, when compared to the r work that would be needed to simply write an object of size r into an array.

Note that the Variable-Size Stateless Allocation Problem comes with three parameters: the universe size U, the total array size n, and a load-factor parameter ε dictating the maximum amount of space $(1-\varepsilon)n+1$ that can be allocated at a time.

The Variable-Size Stateless Allocation Problem has remained one of the most basic open questions in the field of randomized data structures since it was first introduced in STOC '01 by Naor and Teague [279]. They presented a *weakly* history-independent solution that achieved load factor $1 - \varepsilon = 1/2$ and overhead $O(\log n)$ [279]. The authors posed an open problem of whether any scheme could do better—for example, achieving an overhead that is purely a function of ε^{-1}.

In fact, the authors of [279] put quite a bit of emphasis on this open problem. The final sentence of their abstract states simply: "The main open problem we leave is whether it is possible to implement a variable-size record scheme with low overhead." This problem has remained open for more than two decades, not just in the setting of strong history independence, but even in the somewhat easier setting of *weak* history independence.

It is worth noting that the same allocation problem is interesting even for *non-history-independent* data structures. In this context, the problem is typically referred to as the storage reallocation problem [81], and has a folklore solution that achieves overhead $O(\varepsilon^{-1})$. One of the major technical surprises from this thesis will be that our algorithms for the strongly history-independent case will actually *beat* this folkore bound. Thus we will see our third example in this thesis of a case where history-independent algorithms are able to bypass the seemingly natural barriers that their non-history-independent counterparts were previously stuck at.

Fixed-Size Stateless Allocation. The Variable-Size Stateless Allocation Problem has also been studied extensively in the Fixed-Size case where every object $x \in S$ has the same size $\pi(x) = 1$. Here, the load-factor constraint becomes $|S| \leq (1 - \varepsilon)n + 1$, and the allocation function ϕ simply maps each element $x \in S$ to a distinct position $\phi(x) \in \{1, \ldots, n\}$. The overhead of an insertion/deletion of some element x is the number of elements (including the x) whose allocation changes due to the insertion/deletion.

The Fixed-Size Stateless Allocation Problem is widely believed to be one of the simplest examples of a setting in which there is a separation between strong and weak history independence. Indeed, if all that one wishes for is weak history independence, then one can use reservoir sampling [242, 279] to achieve $O(1)$ time (independent of ε) per insertion/deletion while preserving weak history independence (and, in particular, while preserving the invariant that each element is in a random slot). On the other hand, to achieve strong history independence, it is widely believed that the time per insertion/deletion should be a function of ε^{-1} [105, 124, 125, 194, 279]. However, proving this has remained one of the most basic open problems in the area.

The Fixed-Size Stateless Allocation Problem naturally arises in a variety of different settings, both directly in the context of history independence [105, 278, 279], and also in earlier work on unique representations of hash tables [45, 124, 125], and more recent work on distributed stateless worker-task assignment [334, 335].[1]

In these settings, there has been a long line of work on solutions with expected costs of the form $\Omega(\varepsilon^{-1})$ [45, 105, 124, 125, 278, 279], and it would be natural to assume that this bound should be optimal. However, establishing *any* nontrivial lower bounds has proven to be remarkably challenging. For $\varepsilon = 1/n$ (meaning that the array can be completely full), Su, Su, Dornhaus, and Lynch established that the worst-case cost of performing a combined insertion/deletion is at least 2 [334].[2] Subsequently, in ICALP '20, Su and Wein [335] improved this lower bound to 4, assuming a sufficiently large universe size (roughly $U = \text{Tower}(n)$). All of these lower bounds apply only to the *worst-case* cost of the allocation, and only to $\varepsilon = 1/n$. It has remained open to prove *any* nontrivial lower bound for the expected cost, or for $U = \text{poly}(n)$.

Chapter 19. Strong Upper Bounds for Stateless Allocation

In Chapter 19, we give new upper bounds for both the Variable-Size and Fixed-Size Stateless Allocation Problems.

Recall that, in the variable-size case, the prior state of the art was a *weakly* history-independent solution that supported $\varepsilon = 1/2$ with overhead $O(\log n)$. In the same parameter regime, where $\varepsilon = 1/2$, we give a *strongly* history-independent solution that achieves $O(1)$ expected overhead.

Assuming that the maximum size of any element is $O(\varepsilon^4 n)$, the solution extends to support a load factor of $1 - \varepsilon$ with an expected overhead of $O(1 + \log \varepsilon^{-1})$.[3] To the best of our knowledge, this is the first algorithm to beat the folklore $O(\varepsilon^{-1})$ bound

[1] In the context of distributed worker-task assignment, strong history independence is of interest not just as a security property but also as a type of *fault recovery* [335]. If some subset of the agents (i.e., some subset of the numbers $[n]$) suffer a fault in which they lose their memories, they can reconstruct which tasks (i.e., which elements in S) they are assigned to, based only on what the current set of tasks is.

[2] Note that in the model of [334], the set S may actually be a *multiset* of size n However, these two problems are equivalent up to changes in the universe size U. Indeed, as long as $U \geq n$ then the multi-set version contains the non-multi-set version; and for any universe size U, one can construct a multi-set solution using a non-multi-set on a universe of size $U' = Un$.

[3] We remark that, if one wishes to remove the $O(\varepsilon^4 n)$ limitation on maximum object size, one can trivially achieve this at the cost of increasing the expected overhead to $O(\varepsilon^{-4})$.

for the *non*-history-independent version of the problem (see, e.g., discussion in [81]).

In the fixed-size case, the prior states of the art were strongly history-independent solutions that incurred expected overhead $\Omega(\varepsilon^{-1})$ [45, 105, 124, 125, 278, 279]. Our solution for the variable-size case extends immediately to this case: assuming $|S| \geq \Omega(\varepsilon^{-1})$, we are able to support a load factor of $1 - \Theta(\varepsilon)$ while achieving an expected overhead of $O(1 + \log \varepsilon^{-1})$.

In the distributed-computing community, researchers have had a great deal of interest, in particular, in the fixed-size case where $\varepsilon = n^{-1}$ (i.e., the array can be fully saturated), and where the allocation must be *fully deterministic* [334,335]. This is sometimes called the ***memoryless worker-task assignment problem***, since it captures the problem of assigning n workers S to tasks $\{1, 2, \ldots, n\}$ in a memoryless fashion with deterministic worst-case bounds on the cost of workers leaving/entering the set. This version of the problem would seem, *a priori*, to be significantly harder, since we cannot use randomization to achieve strong history independence. At the same time, efforts to prove lower bounds [334,335] have stalled at $\Omega(1)$.

Our final result in the chapter is a (deterministic) solution to the memoryless worker-task assignment problem with worst-case overhead $O(\log^2 n)$ per insertion/deletion (where the workers S are assumed to come from a polynomial-size universe $[U]$). Our solution is non-constructive, making use of the probabilistic method. However, making use of techniques from the derandomization literature, we are also able to give a constructive solution with $\operatorname{polylog} n$ cost. To the best of our knowledge, these constructions are the first instances of a dynamic data-structural problem having a non-trivial *deterministic* strongly history-independent solution.

Chapter 20. Strong Lower Bounds for Stateless Allocation

The Fixed- and Variable- Size Stateless Allocation Problems have proven remarkably resistant to lower bounds. Indeed, the only known bounds, achieved by [334,335], are of the form $\Omega(1)$ and apply to the *worst-case* (rather than expected) overhead for the fixed-size problem with $\varepsilon^{-1} = n$.

In Chapter 20, we establish a nearly tight lower bound for the Fixed-Size Stateless Allocation Problem. We show that any Stateless Allocation scheme must incur expected cost $\Omega(\log \varepsilon^{-1} / \log \log \varepsilon^{-1})$. This matches our upper bound of $O(1 + \log \varepsilon^{-1})$ up to an $O(\log \log \varepsilon^{-1})$ factor.

Of course, since the fixed-size case is a special instance of the variable-size case, our lower bound for the former extends immediately to the latter. Thus our bounds for the variable-size case are also tight up to a factor of $O(\log \log \varepsilon^{-1})$.

Our lower bound provides a concrete separation between the weak and strong history independence for the Fixed-Size Stateless Allocation Problem. The proof offers a natural framework for how to chain together indistinguishability arguments in order to establish lower bounds for strong history independence—we expect that similar ideas will likely be useful for establishing lower bounds for related problems in subsequent work.

Chapter 21. Efficient Data-Structural Implementations

Up until now, we have focused on optimizing only the overhead of an allocator. However, even if an allocator exhibits low expected overhead, it may still be difficult to implement the allocator *time efficiently*. In Chapter 21, we show that for a large range of ε, it is possible to construct time-efficient allocators for both the fixed- and variable- size cases.

Formally, we can capture this as a data-structural problem: We wish to construct a strongly history-independent hash table that maps each element $x \in S$ to a contiguous sub-array of size $\pi(x)$. The hash table is said to achieve load factor $1 - O(\varepsilon)$ if the hash table uses space $(1 + O(\varepsilon))m$, where $m = \sum_{x \in S} \pi(x)$. The hash table is said to achieve (time) overhead $O(L)$ if the expected time to insert/delete of an item x is bounded by $O(L\pi(x))$. Ideally, the hash table should not incur any query overhead, i.e., the time to query an item x should be $O(1)$ (w.h.p.). We assume that $\pi(x) \geq 1$ for each $x \in S$, and, in particular, that the element x is stored explicitly at the beginning of the sub-array allocated to it.

Even in the fixed-size case, even though Chapter 19 gives a solution with expected cost $O(1 + \log \varepsilon^{-1})$, it is unclear a priori whether this solution can be used in data-structural settings. Indeed our solution can be viewed as a special case of open addressing, which naively requires $\Omega(\varepsilon^{-1})$ time per insertion/deletion (even if $o(\varepsilon^{-1})$ *rearrangements* actually occur). Nonetheless, we show that one can overcome these issues using data-structural techniques. We give a strongly history-independent hash table that, at a load factor of $1 - \varepsilon$ (where $\varepsilon^{-1} \leq \log^{1/10} m$), supports insertions and deletions in $O(1 + \log \varepsilon^{-1})$ expected time and queries in constant time (w.h.p.).

Combining this with our aforementioned lower bound, we conclude that the optimal insertion cost for any strongly history-independent hash table is $\tilde{\Theta}(\log \varepsilon^{-1})$. We remark that, prior to our work, the previous state of the art, due to Blelloch and Golovin in FOCS '07 [105], achieved an expected insertion cost of $\Theta(\varepsilon^{-2})$.

Next, we turn our attention to the variable-size case, where we show that it is possible to produce a time-efficient implementation of our $O(\varepsilon^{-1})$-overhead allocator. That is, we give a strongly history-independent hash table (Theorem 243) that maps each key $x \in S$ to a contiguous subarray of size $\pi(x)$ and that, at any load factor $1 - \varepsilon$ satisfying $\varepsilon^{-1} \leq \log^{1/10} n$, supports queries in time $O(1)$ (w.h.p.), and supports insertions/deletions for items of sizes up to $O(\varepsilon^4 n)$ with expected overhead $O(1 + \log \varepsilon^{-1})$. In the case where $\varepsilon^{-1} = O(1)$, this is the first known solution to achieve $O(1)$ expected overhead [279].

18.1 Conventions

When discussing discrete intervals, we will use $[k]$ to denote $\{1, 2, \ldots, k\}$ and $[a, b]$ to denote $\{a, a+1, a+2, \ldots, b\}$. When discussing real intervals, we will use $[a, b)$ to denote the half-open interval from a to b, and $[k]$ to denote $[0, k)$.

We will typically use $S \subseteq [U]$ to denote the set of keys being allocated and

$\pi : S \to \mathbb{N}$ to denote the size function being used. Note that, although each key $x \in S$ is $\log U$ bits, when we refer to the *size* of x, we mean $\pi(x)$ (not $\log U$). In the fixed-size version of the problem, we will omit discussion π, since each key $x \in S$ implicitly has size 1.

We can think of a stateless allocation algorithm ALG as having four inputs: the random bits R that it should use, the set S of keys that it must allocate, the size function π, and the load-factor parameter ε. In most cases, we will treat R and ε as global variables, and refer to the allocation produced by ALG as either ALG(S) (for the fixed-size version of the problem) or ALG(S, π) (for the variable-size version of the problem). When convenient, we will often simply use ϕ to denote the allocation function produced by ALG.

The allocation ALG(S, π) can be viewed as either mapping each key $x \in S$ to an *interval* $[a_x, b_x] \subseteq [n]$ of size $b_x - a_x = \pi(x)$, or as assigning the key x to the *sub-array* $A[a_x, a_x + 1, \ldots, b_x]$ of an array A of size n. In most contexts, we will take the interval-assignment perspective (since, of course, stateless allocation can also be applied to non-data-structural settings, e.g., [102,334,335]); but in some cases where it is convenient, we will use the array/sub-array nomenclature instead. As a convention, we will typically use m to denote the sum $\sum_{x \in S} \pi(x)$ of the sizes of the elements in S (or, in some cases, to denote an upper bound on the sum), and we will typically use n to denote the size of the array being used for the allocation problem. Although in general, we will be interested in $m = (1 - \varepsilon)n$ (i.e., the load factor is ε), in some of our upper-bound constructions it will be helpful to instead take the more relaxed perspective of requiring only that $m = (1 - \Theta(\varepsilon))n$.

In some cases, we will allow for keys $x \in S$ to have real-valued sizes $\pi(x) \geq 1$. Of course, this means that each key $x \in S$ will be assigned to an interval $\phi(x)$ with real-valued endpoints (rather than integer endpoints). Note that, if each key $x \in S$ has an integer size $\pi(x) \in \mathbb{N}$, then the endpoints of $\phi(x)$ can be trivially rounded to integers (simply replace each endpoint q with $\lfloor q \rfloor$), while preserving the fact that different keys are assigned to disjoint intervals and that each key x is assigned to an interval of size $\pi(x)$.

Finally, as a convention, we say that an event E occurs *with high probability (w.h.p.)* in a variable k if the event occurs with probability $1 - 1/k^c$ for an arbitrarily large positive constant c of our choice. Equivalently, one can say that E occurs with probability $1 - 1/\operatorname{poly}(k)$.

Chapter 19

Strong Upper Bounds for Stateless Allocation

In this chapter, we give efficient solutions to both the Fixed-Size and Variable-Size Stateless Allocation Problems.

Section 19.1. Achieving $O(1 + \log \varepsilon^{-1})$ expected overhead. We begin by constructing an allocator that can handle objects with power-of-two sizes. The allocator supports load factor $1 - \varepsilon$ while achieving expected overhead $O(1 + \log \varepsilon^{-1})$.

Theorem 191. *Algorithm 4 is a solution to the Variable-Size Stateless Allocation Problem in the setting where objects have power-of-two sizes that divide n. The algorithm achieves load factor $1 - \varepsilon$ with expected overhead $\Theta(1 + \log \varepsilon^{-1})$ per insertion/deletion.*

As a corollary, we immediately get a new bound for Fixed-Size Stateless Allocation. This is the first solution to achieve $o(\varepsilon^{-1})$ expected overhead.

Corollary 192. *Algorithm 4 is a solution to the Fixed-Size Stateless Allocation Problem that, on input sets S of size at least $\Omega(\varepsilon^{-1})$, achieves load factor $1 - O(\varepsilon)$ with expected overhead $\Theta(1 + \log \varepsilon^{-1})$ per insertion/deletion.*

Both of these results also extend to the setting where n changes over time based on $\sum_{x \in S} \pi(S)$, so that the total memory used at any given moment is $(1 + O(\varepsilon)) \sum_{x \in S} \pi(x)$.

An intriguing aspect of our solution for power-of-two sizes is that the solution can be viewed as a variation on uniform probing. In classical uniform probing, each element x has a random sequence $h_1(x), h_2(x), \ldots \in [n]$ of positions where it could go, and the element is placed in the first available position from that sequence. Of course, uniform probing on its own is *not* strongly history independent, but there are several well-studied variations [124, 125, 279] that are, including the $O(\varepsilon^{-1})$-overhead fixed-size allocator given by Naor and Teague [279]. What is interesting about the variation presented in Algorithm 4 is that it is able to (1) achieve *sub-linear* overhead in ε^{-1} and (2) handle different items having very different sizes *without any asymptotic* *blowup in overhead.* The approach will likely also be of interest to the wider hashing community.

Finally, we show that with additional techniques our results can be extended to handle arbitrary (not-necessarily-power-of-two) size objects. This leads to the main theorem of the section:

Theorem 193. *Let U, ε^{-1} be positive integers. Consider inputs (S, π) where $S \subseteq [U]$, and where $\pi(x) \in [1, \varepsilon^4 \sum_{x \in S} \pi(x)]$ for all $x \in S$. Then one can construct a stateless allocator that uses space $(1 + O(\varepsilon)) \sum_{x \in S} \pi(x)$ and that incurs expected overhead $O(1 + \log \varepsilon^{-1})$ per insertion and deletion.*

This result improves the state of the art even for non-history-independent solutions, which previously stood at $O(\varepsilon^{-1})$ [81], as well as for weakly history-independent solutions, which previously stood at $O(\log n)$ for $\varepsilon = 1/2$ [279]. As we shall see in the next chapter, the $O(1 + \log \varepsilon^{-1})$ overhead bound is tight up to $O(\log \log \varepsilon^{-1})$ factors for both the fixed-size and variable-size versions of the problem.

Section 19.2. Achieving polylog n worst-case overhead for worker-task assignment

In the second part of the chapter, we turn our attention to a special case of the Fixed-Size Stateless Allocation Problem that has received recent attention in the distributed computing community. This version of the problem, known as ***memoryless worker-task assignment***, focuses on the case of $\varepsilon^{-1} = n$ and considers the *worst-case* overhead of an insertion/deletion. It is the worst-case nature of this problem that makes it so interesting—intuitively it seems very difficult to construct a strongly history-independent data structure that offers worst-case (and therefore deterministic) guarantees.

In order to be consistent past works on the memoryless worker-task assignment problem [334, 335], we adapt here the conventions that are normally used to discuss the problem (as we shall see, these conventions end up being convenient for discussing our solution, anyway).

Formally, the (memoryless) worker-task assignment problem is defined as follows. There are w workers $1, 2, \ldots, w$ and t tasks $1, 2, \ldots, t$. A ***worker-task assignment function*** ϕ is a function that takes as input a multiset T of w tasks, and produces an assignment of workers to tasks such that the number of workers assigned to a given task $\tau \in T$ is equal to the multiplicity of τ in T.

Two task multisets T_1, T_2 of size w are said to be *adjacent* if they agree on exactly $w - 1$ elements; that is, $|T_1 \setminus T_2| = |T_2 \setminus T_1| = 1$.[1] The ***switching cost*** between two adjacent task multisets T_1, T_2 of size w is defined as the number of workers whose assignment changes between $\phi(T_1)$ and $\phi(T_2)$. The ***switching cost of*** ϕ is defined to be the maximum switching cost over *all* pairs of adjacent task multisets. The goal of

[1]Let $m_A(i)$ denote the number of times element i appears in multiset A. Then, for any two multisets A and B, we define multisets $A \setminus B$, $A \cup B$, and $A \cap B$ to be such that $m_{A \setminus B}(i) = \max(0, m_A(i) - m_B(i))$, $m_{A \cup B}(i) = \max(m_A(i), m_B(i))$, and $m_{A \cap B}(i) = \min(m_A(i), m_B(i))$, for every element i.

the worker-task assignment problem is to design a worker-task assignment function with the minimum possible switching cost.

Su, Su, Dornhaus, and Lynch [334] initiated the study of the worker-task assignment problem and observed that assigning the workers to tasks in numerical order achieves a switching cost of $\min(t-1, w)$. They also proved a lower bound of 2 on switching cost, and showed a matching upper bound in the case where $w \leq 6$ and $t \leq 4$. Subsequent work by Su and Wein [335], in ICALP 2020, pushed further on the lower-bound side of the problem. They proved that a switching cost of 2 is not always possible in general. They show that, if $t \geq 5$ and $w \geq 3$, then any worker-task assignment function must have switching cost at least 3; and if t is sufficiently large in terms of w (i.e., it is a tower of height $w-1$), then the switching cost must be at least 4. The bounds by [334] and [335] have until now remained state-of-the-art. It remains unknown whether the optimal switching cost is small (it could be as small as 4) or large (it could be as large as $\min(t-1, w)$). And even achieving a lower bound of 4 on switching cost [335] has required a quite involved argument.

In Section 19.2, we establish that it is possible to construct a worker-task assignment function with $O(\log w \log(wt))$ switching cost. This resolves the open question as to whether memoryless worker can allocate themselves to tasks with strong worst-case guarantees.

Theorem 194. *There exists a worker-task assignment function that achieves switching cost $O(\log w \log(wt))$.*

As a corollary, we restate this result in the context of the Fixed-Size Stateless Allocation Problem.

Corollary 195. *There exists a solution to the Fixed-Size Stateless Allocation Problem that handles sets $S \subseteq [\operatorname{poly} n]$ of size up n, and produces an injection $\phi : S \to [n]$ with deterministic worst-case overhead $O(\log^2 n)$.*

Theorem 194 is proven via the probabilistic method and is thus non-constructive. By replacing random hash functions with strong dispersers, however, we show that one can construct an explicit worker-task assignment function with polylogarithmic switching cost.

Theorem 196. *There is an explicit worker-task assignment function that achieves switching cost $O(\operatorname{polylog}(wt))$.*

Both Theorems 194 and 196 continue to hold in the more general setting where the size of T changes over time. That is, T is permitted to be any multiset of $[t]$ of size w or smaller. Two task multisets T_1, T_2 of different sizes are considered adjacent if they satisfy $\big||T_1| - |T_2|\big| = 1$ and $|(T_1 \cup T_2) \setminus (T_1 \cap T_2)| = 1$. If $|T| < w$, then our worker-task assignment function assign workers $1, \ldots, |T|$ to tasks, and leaves workers $|T|+1, \ldots, w$ unassigned.

19.1 Achieving $O(1 + \log \varepsilon^{-1})$ Expected Overhead

In this section, we construct stateless allocators with $O(1+\log \varepsilon^{-1})$, culminating with the proof of Theorem 193.

19.1.1 Achieving $O(1 + \log \varepsilon^{-1})$ Overhead with Power-of-Two Sizes

We begin by showing that, for objects whose sizes are powers of two, one can achieve load factor $1 - \varepsilon$ while incurring expected overhead $O(1 + \log \varepsilon^{-1})$. Our approach, which can be viewed as a (surprisingly simple) variation on classical uniform probing, is presented in Algorithm 4.

Algorithm 4 Input is a set $S \subseteq [U]$ and a size function $\pi : S \to \{1, 2, 4, 8, \ldots\}$ such that $\sum_{x \in S} \pi(x) \leq (1 - \varepsilon)n + 1$. Allocates the elements $x \in S$ to disjoint intervals $\phi(x) \subseteq [n]$ satisfying $|\phi(x)| = \pi(x)$. Assumes that $\pi(x)$ divides n for all $x \in S$.

```
1: procedure PowersOfTwoAllocator($S, \pi$)
2:     Sort elements of $S$ in decreasing order by size, and then within each size class
       sort them by ID.
3:     Call the sorted elements $x_1, \ldots, x_k$.
4:     For each element $x_i$, let $h_1(x_i), h_2(x_i), \ldots$ be random integers in
       $\{1, 2, \ldots, n/\pi(x_i)\}$.
5:     for $i \in \{1, 2, \ldots, k\}$ do
6:         for $j \in \{1, 2, \ldots\}$ do
7:             if Slots $h_j(x_i) \cdot \pi(x_i) + 1, \ldots, h_j(x_i) \cdot \pi(x_i) + \pi(x_i)$ are all free then
8:                 Allocate $x_i$ to $\phi(x_i) := [h_j(x_i) \cdot \pi(x_i) + 1, h_j(x_i) \cdot \pi(x_i) + \pi(x_i)]$.
9:                 Mark slots $h_j(x_i) \cdot \pi(x_i) + 1, \ldots, h_j(x_i) \cdot \pi(x_i) + \pi(x_i)$ as occupied.
10:                Break out of for loop.
       return $\phi$
```

Consider $S \subseteq [U]$, and let $S' = S \cup \{x\}$ for some $x \in [U] \setminus S$. Let π be a function mapping each $s \in S$ to a size $\pi(s)$, and suppose that $\sum_{s \in S} \pi(s) \leq (1 - \varepsilon)n + 1$. Let ϕ and ϕ' be the allocations produced by Algorithm 4 on S and S', respectively, using π as the size function.

Let $x'_1, x'_2, \ldots, x'_{k+1}$ be the elements of S' sorted in decreasing order of size, and then sorted lexicographically (i.e., by name/ID) within each size class. Let q be the rank of x in this sorted list, i.e., $x'_q = x$. For $i \in \{1, 2, \ldots, k+1\}$, let $A_i = \bigcup_{t \in [i] \setminus [q]} \phi(x'_t)$ denote the cumulative set of slots that ϕ allocates to $\{x'_1, \ldots, x'_i\} \setminus \{x\}$, and let $A'_i = \bigcup_{t \in [i]} \phi'(x'_t)$ denote the cumulative set of slots that ϕ' allocates to $\{x'_1, \ldots, x'_i\}$.

We now establish one of the key properties achieved by Algorithm 4, namely that $|A'_i \setminus A_i| = \pi(x)$ for all $i \geq q$.

Lemma 197. *For each $i \in [q, k+1]$, $A'_i = A_i \cup B_i$ for some set $B_i \subseteq [n]$ of size $\pi(x)$. Moreover, B_i can be written as the union of aligned intervals of size $\pi(x'_i)$ (i.e.,* *intervals of the form $[\ell\pi(x_i) + 1, \ell\pi(x_i) + \pi(x_i)]$ for some integer ℓ).*

Proof. For all $i < q$, $\phi(x_i) = \phi(x'_i)$. Thus $A'_q = A_q \cup \phi'(x)$. That is, for $i = q$, the lemma holds with $B_i = \phi'(x)$. Note that, by design, $B_i = \phi'(x)$ is an aligned interval of size $\pi(x'_q)$ (i.e., $\pi(x)$).

Now consider $i > q$, and suppose by induction that the lemma holds for $i-1$. When Algorithm 4 is deciding $\phi(x'_i)$, the set of free slots is given by $[n] \setminus A_{i-1}$. In contrast, when Algorithm 4 is deciding $\phi'(x'_i)$, the set of free slots is given by $[n] \setminus A_{i-1} \cup B_{i-1}$. Because $\phi(x'_i)$ is an aligned interval of size $\pi(x'_{i-1})$, we have either that $\phi(x'_i) \subseteq B_{i-1}$ or that $\phi(x'_i) \cap B_{i-1} = \emptyset$. If $\phi(x'_i) \cap B_{i-1} = \emptyset$, then we will have $\phi'(x'_i) = \phi(x'_i)$, implying that $A'_i = A_i \cup B_{i-1}$ and therefore that the lemma holds with $B_i = B_{i-1}$. On the other hand, if $\phi(x'_i) \subseteq B_{i-1}$, then we will have $A_i = A_{i-1} \cup \phi(x'_i)$ and that $A'_i = A'_{i-1} \cup \phi'(x'_i)$. This means that $A'_i = A_i \cup (B_{i-1} \setminus \phi(x'_i)) \cup \phi'(x'_i)$. That is, $A'_i = A_i \cup B_i$ where $B_i = (B_{i-1} \setminus \phi(x'_i)) \cup \phi'(x'_i)$. Since $\phi(x'_i)$ and $\phi'(x'_i)$ are both aligned intervals of size $\pi(x'_i)$, we have that $|B_i| = |B_{i-1}| = |\pi(x)|$ and that B_i is a union of aligned intervals of size $\pi(x'_i)$. ∎

Using Lemma 197 as a building block, we can now analyze the probability that the allocations $\phi(x'_i)$ and $\phi'(x'_i)$ disagree for a given $i > q$.

Lemma 198. *For $i > q$, we have that*

$$\Pr[\phi(x'_i) \neq \phi'(x'_i)] = \frac{\pi(x)}{n - \sum_{\ell=1}^{i-1} \pi(x'_\ell)}.$$

Proof. By Lemma 197, we have that $A'_{i-1} = A_{i-1} \cup B_{i-1}$ for some set B_{i-1} of size exactly $\pi(x)$ that can be written as the union of aligned intervals of size $\pi(x'_{i-1})$. Since $\pi(x'_{i-1})$ is divisible by $\pi(x'_i)$, B_{i-1} can also be written as a union of aligned intervals of size $\pi(x'_i)$. Call the set of these intervals $\overline{B}$.

The event $\phi(x'_i) \neq \phi'(x'_i)$ occurs if and only if $\phi(x'_i) \in \overline{B}$. Let $\overline{I}$ be the set of aligned intervals of size $\pi(x'_i)$ in $[n] \setminus A_{i-1}$. Then $\phi(x'_i)$ is a uniformly random element of $\overline{I}$, where the randomness is determined by the hashes $h_1(x'_i), h_2(x'_i), \ldots$. Critically these hashes are independent of both $\overline{I}$ and $\overline{B}$ (which are determined by $x'_1, \ldots, x'_{i-1}$). Thus, if we condition on any outcome for the random variables $\overline{I}$ and $\overline{B} \subseteq \overline{I}$, then the probability that $\phi(x'_i) \in \overline{B}$ is exactly

$$\frac{|\overline{B}|}{|\overline{I}|} = \frac{\pi(x)}{n - \sum_{\ell=1}^{i-1} \pi(x'_\ell)}.$$

This completes the proof of the lemma. ∎

Finally, we can analyze the expected overhead achieved by Algorithm 4.

Theorem 191. *Algorithm 4 is a solution to the Variable-Size Stateless Allocation Problem in the setting where objects have power-of-two sizes that divide n. The* *algorithm achieves load factor $1-\varepsilon$ with expected overhead $\Theta(1+\log \varepsilon^{-1})$ per insertion/deletion.*

Proof. The expected overhead of the algorithm is given by

$$1 + \sum_{i \in [k+1] \setminus \{q\}} \frac{\pi(x'_i)}{\pi(x)} \cdot \Pr[\phi(x'_i) \neq \phi'(x'_i)].$$

For $i < q$, we have deterministically that $\phi(x'_i) = \phi'(x'_i)$. Thus the expected overhead is given by

$$1 + \sum_{i \in (q,k+1]} \frac{\pi(x'_i)}{\pi(x)} \cdot \Pr[\phi(x'_i) \neq \phi'(x'_i)],$$

which by Lemma 198 is

$$\begin{aligned}
&1 + \sum_{i \in (q,k+1]} \frac{\pi(x'_i)}{\pi(x)} \cdot \frac{\pi(x)}{n - \sum_{\ell=1}^{i-1} \pi(x'_\ell)} \\
&= 1 + \sum_{i \in (q,k+1]} \frac{\pi(x'_i)}{n - \sum_{\ell=1}^{i-1} \pi(x'_\ell)} \\
&\leq 1 + \sum_{i \in (q,k+1]} \sum_{\ell=1}^{\pi(x'_i)} \frac{1}{n - \sum_{\ell=1}^{i-1} \pi(x'_\ell) - \ell + 1} \\
&\leq 1 + \sum_{t=1}^{(1-\varepsilon)n+1} \frac{1}{n-t+1} \\
&= 1 + \sum_{t=\varepsilon n}^{n} \frac{1}{t} \\
&= \Theta(1 + \log \varepsilon^{-1}).
\end{aligned}$$

∎

19.1.2 Supporting Dynamic Resizing

Next, we extend Algorithm 4 to support dynamic resizing, i.e., so that n changes over time to always satisfy $n = (1+\Theta(\varepsilon))\sum_{x \in S} \pi(x)$. In addition to requiring that the elements in S have power-of-two sizes, we will also require that each $s \in S$ satisfies

$$\pi(s) \leq 0.5\varepsilon \sum_{x \in S} \pi(x) \qquad (19.1)$$

for a sufficiently small positive constant c.

The key to supporting dynamic resizing is to make use of an old result from the hashing literature: Larson introduced linear hashing [233] which allows for one to dynamically increase/decrease the number of bins that a hash function h maps to. The key property that makes linear hashing useful is that, if there are k bins, then we can increase that number to $k' = k + \Theta(\varepsilon k)$ while achieving the following guarantee: each $h(x)$ has probability at most $O(\varepsilon)$ of changing; and if $h(x)$ changes, then the new value of $h(x)$ is in the range $(k, k']$. (Moreover, the size of the increase $k' - k$ is guaranteed to be a power of two between $k\varepsilon/2$ and $k\varepsilon$, inclusive.)

Remark 199. *Although this subsection is not concerned with time efficiency, it is worth taking a moment to discuss the time efficiency of linear hashing in the RAM model. Although Larson's original scheme required logarithmic time to evaluate h, we subsequently showed in [75] that it is possible to design a variation on the scheme (which we call* **waterfall addressing***) that offers $O(1)$-time evaluation so long as $\varepsilon^{-1} \leq \operatorname{polylog} n$. This will be useful later on, in Chapter 21, when we are designing time-efficient data-structural implementations of our allocation schemes.*

We can use linear hashing to compute each $h_i(x_j)$. This allows us to perform a ***resize operation*** in which we increase/decrease n by $\Theta(\varepsilon n)$. Consider, in particular, a resize operation that increases n to $n' = n + \Theta(\varepsilon n)$. Let $h_i(x_j)$ and $h_i'(x_j)$ be the values of $h_i(x_j)$ before/after the resizing operation. What linear hashing guarantees is that $\Pr[h_i(x_j) \neq h_i'(x_j)] = O(\varepsilon)$ and that, if $h_i(x_j) \neq h_i'(x_j)$ then it is because $h_i'(x_j) \in (n/\pi(x), n'/\pi(x)]$.

Let ϕ denote the allocation prior to the resize operation (using the original n) and ϕ' denote the allocation after the resize operation (using $n' = n + \Theta(\varepsilon n)$). Define the ***cost*** of a resize operation to be $\sum_{x \in S} \pi(x) \cdot \mathbb{I}(\phi(x) \neq \phi'(x))$. We will prove the following proposition.

Proposition 200. *Consider Algorithm 4 implemented with linear hashing. Consider input sets S consisting of items with power-of-two sizes. Consider a resize operation in which n is increased by $\Theta(\varepsilon n)$ to become n', and suppose that both n and n' are divisible by $\pi(x)$ for every $x \in S$. The expected sum of sizes of items whose allocations change due to the resize is $O(\varepsilon n(1 + \log \varepsilon^{-1}))$.*

Before we give the proof of Proposition 200, it is worth taking a moment to see how we can use it in order to obtain a dynamically-resizable version of Theorem 191.

Theorem 201. *There is a solution to the Variable-Size Stateless Allocation Problem in the setting where objects have power-of-two sizes that achieves the following guarantees. So long as every $s \in S$ satisfies $\pi(s) \leq 0.5\varepsilon \sum_{x \in S} \pi(x)$, then the algorithm performs allocations within the interval $[0, (1 + O(\varepsilon)) \sum_{x \in S} \pi(x)]$ and incurs expected overhead $\Theta(1 + \log \varepsilon^{-1})$ per insertion/deletion.*

Proof. We will make use of Proposition 200 to perform resizes. One thing we must be careful about is the divisibility constraint in Algorithm 4, which requires that the value of n we are currently using must always be divisible by $\pi(x)$ for every $x \in s$. Fortunately, linear hashing always changes n by powers of two between $\varepsilon n/2$ and εn. Thus, if we start with a value of n that satisfies the divisibility constraint, and if whenever a resize operation is performed every $s \in S$ satisfies

$$\pi(s) \le 0.5\varepsilon \sum_{x \in S} \pi(x),$$

then the divisibility constraint will continue to hold after the resize. As this argument handles the divisibility constraint, we will ignore the constraint for the rest of the proof.

Although Proposition 200 bounds the cost of a resize operation, it does not tell us when to perform resize operations. We must do this in a way that is strongly history independent, and that offers the following probabilistic guarantee: if an element x of size $\pi(x)$ is inserted, it triggers a resize with probability $O(\pi(x)/(\varepsilon n))$, where n is the current value of n. This will then imply, by Proposition 200, that the expected resizing cost from the insertion is

$$O\left(\frac{\pi(x)}{\varepsilon n}\varepsilon n(1 + \log \varepsilon^{-1})\right) = O(\pi(x)(1 + \log \varepsilon^{-1})).$$

In other words, the expected *overhead* is still $O(1 + \log \varepsilon^{-1})$.

To determine when resize operations should occur, we use a standard resizing trick for strongly history-independent data structures. Let $n_1 < n_2 < n_3 < \cdots$ be all of the possible values that n can take. Let $r_0 = 0$ and let r_i, $i \ge 1$, be a uniformly random value between n_i and n_{i+1}. Let j be the unique integer such that $r_{j-1} \le \sum_{x \in S} \pi(x) < r_j$. Then we set $n = n_{j+2}$.

This value of n ensures, by design, that $\sum_{x \in S} \pi(x) = (1 - \Theta(\varepsilon))n$ at any given moment. Moreover, it is now easy to analyze the probability of a given insertion (of some element x with size $\pi(x)$) causing a resize operation. The resize operation occurs if there is some r_i such that the insertion of x causes $\sum_{s \in S} \pi(x)$ to toggle from being $< r_i$ to $\le r_i$. The randomness of the r_is ensures that this occurs with probability $O(\pi(x)/(\varepsilon n))$ (because the up to two r_is that $\sum_{s \in S} \pi(x)$ is at risk of crossing are each drawn at random from an interval of size $\Theta(\varepsilon n)$). This is exactly the probability that we needed to guarantee an expected overhead of $O(1 + \log \varepsilon^{-1})$, completing the proof. ∎

In the rest of the subsection, we prove Proposition 200. In addition to using $\phi, \phi', h, h', n, n'$ as defined above, we will need a few additional notations.

Let $x_1, x_2, \ldots$ be the elements in S in the sorted order used by Algorithm 4. Call x_i ***special*** if ϕ allocates x_i using h_j, but $h_j(x_i) \ne h'_j(x_i)$ (i.e., $h'_j(x_i)$ is in $(n/\pi(x_i), n'/\pi(x_i)]$). By the guarantees of linear hashing, we know that each x_i has probability $O(\varepsilon)$ of being special. Let P_i be the subset of $x_1, x_2, \ldots, x_i$ that are special.

Let Δ_i (resp. Δ'_i) be the set of slots that, after the processing of $x_1, \ldots, x_i$, are free in ϕ (resp. ϕ') but not free in ϕ' (resp. ϕ). And Δ_0 (resp. Δ'_0) to be the set of slots that, at the beginning of the algorithm, are free in ϕ (resp. ϕ') but not free in ϕ' (resp. ϕ). This means that $|\Delta_0| = 0$ and $|\Delta'_0| = |\{n+1, \ldots, n'\}| = O(\varepsilon n)$.

We begin with a simple alignment observation, which follows by the same reasoning as Lemma 197.

Observation 202. *Each of Δ_{i-1} and Δ'_{i-1} can be written as the union of aligned intervals of size $\pi(x_i)$. Furthermore, if F_{i-1} is the set of slots that are free in both ϕ and ϕ' at the time that x_i is being processed, then F_{i-1} can also be written as the union of aligned intervals of size $\pi(x_i)$.*

As an immediate corollary, we also get the following observation:

Observation 203. *Whenever an element x_i is being processed by ϕ and ϕ', at least one of the following must hold: $\phi(x_i) \subseteq \Delta_{i-1}$, $\phi'(x_i) \subseteq \Delta'_{i-1}$, x_i is special, or $\phi(x_i) = \phi'(x_i)$.*

In particular, if none of the first three options hold, then ϕ and ϕ' must use the same hash function $h_j(x_i) = h'_j(x_i)$ to assign x_i to the same intervals as each other.

Building on these observations, we can now prove a more nontrivial fact, namely a bound on the expected cumulative size of $\Delta_i \cup \Delta'_i$.

Lemma 204. $\mathbb{E}[|\Delta_i| + |\Delta'_i|] \leq O(\varepsilon n)$.

Proof. We begin by relating $|\Delta_{i-1}| + |\Delta'_{i-1}|$ to $|\Delta_i| + |\Delta'_i|$.

No matter what, we have the trivial inequality $|\Delta_i|+|\Delta'_i| \leq |\Delta_{i-1}|+|\Delta'_{i-1}|+2\pi(x_i)$. This is the inequality that we will use if x_i is special.

On the other hand, if x_i is *not special*, then by Observation 203 we can consider three cases:

- **Option 1: $\phi(x_i) \subseteq \Delta_{i-1}$.** This means that $|\Delta_i| = |\Delta_{i-1}| - \pi(x_i)$. On the other hand, we have trivially that $|\Delta'_i| \leq |\Delta'_{i-1}| + \pi(x_i)$. Thus $|\Delta_i| + |\Delta'_i| \leq |\Delta_{i-1}| + |\Delta'_{i-1}|$.
- **Option 2: $\phi'(x_i) \subseteq \Delta'_{i-1}$.** This means that $|\Delta'_i| = |\Delta'_{i-1}| - \pi(x_i)$. On the other hand, we have trivially that $|\Delta_i| \leq |\Delta_{i-1}| + \pi(x_i)$. Thus $|\Delta_i| + |\Delta'_i| \leq |\Delta_{i-1}| + |\Delta'_{i-1}|$.
- **Option 3: $\phi(x_i) = \phi'(x_i)$.** In this case, $|\Delta_i| + |\Delta'_i| = |\Delta_{i-1}| + |\Delta'_{i-1}|$.

In all three cases, $|\Delta_i| + |\Delta'_i| \leq |\Delta_{i-1}| + |\Delta'_{i-1}|$.

As the only case in which $|\Delta_i| + |\Delta'_i|$ increases over $|\Delta_{i-1}| + |\Delta'_{i-1}|$ is when x_i is special (and even in this case the increase is at most $2\pi(x_i)$), we have that

$$\mathbb{E}[|\Delta_i| + |\Delta'_i|] \leq \mathbb{E}[|\Delta_0| + |\Delta'_0|] + 2\mathbb{E}\left[\sum_{x \in P_i} \pi(x)\right] \leq O(\varepsilon n).$$

∎

We can now bound the probability that ϕ and ϕ' disagree on x_i.

Lemma 205.

$$\Pr[\phi(x_i) \neq \phi'(x_i)] = \frac{O(\varepsilon n)}{n - \sum_{\ell=1}^{i-1} \pi(x'_\ell)}.$$

Proof. The probability that x_i is special is at most $O(\varepsilon) \leq \frac{O(\varepsilon n)}{n - \sum_{\ell=1}^{i-1} \pi(x'_\ell)}$. On the other hand, if x_i is not special, then Lemma 203 tells us that x_i gets different allocations in ϕ and ϕ' with probability

$$\begin{aligned}
&\Pr[\phi(x_i) \subseteq \Delta_{i-1} \text{ or } \phi'(x_i) \subseteq \Delta'_{i-1}] \\
&\leq \Pr[\phi(x_i) \subseteq \Delta_{i-1}] + \Pr[\phi'(x_i) \subseteq \Delta'_{i-1}] \\
&= \frac{\mathbb{E}[|\Delta_{i-1}|]}{n - \sum_{j=1}^{i-1} \pi(x_j)} + \frac{\mathbb{E}[|\Delta'_{i-1}|]}{n' - \sum_{j=1}^{i-1} \pi(x_j)} \\
&\leq \frac{\mathbb{E}[|\Delta_{i-1}| + |\Delta'_{i-1}|]}{n - \sum_{j=1}^{i-1} \pi(x_j)} \\
&\leq \frac{O(\varepsilon n)}{n - \sum_{j=1}^{i-1} \pi(x_j)},
\end{aligned}$$

where the final inequality follows from Lemma 204. ∎

Finally, proceeding as in Section 19.1.1, but with Lemma 205 in place of Lemma 198, we can conclude that the expected cost of a resize operation is $O(\varepsilon n \log \varepsilon^{-1})$, as desired.

19.1.3 Allocating Items with Sizes in $[1, 2)$

Our next step is to consider objects with sizes in the range $[1, 2)$. In this subsection, the objects that we consider will have *real-valued* sizes. Thus, we will allocate them to disjoint intervals $\phi(x)$ whose sizes (and end points) are not necessarily integral. Later on, in Section 19.1.4, we will apply these allocators to settings where objects have integer sizes between two powers of two—in this case, the endpoints of each allocated interval $\phi(x) = [a, b)$ can also be made integral by replacing the continuous interval $[a, b)$ with the discrete interval $[\lfloor a \rfloor, \lfloor b \rfloor]$.

Proposition 206. *Let ε^{-1}, U be positive integer parameters. Consider sets $S \subseteq [U]$ and size functions $\pi : [U] \to [1, 2)$ such that $|S| \geq \Omega(\varepsilon^{-3})$. Finally, let $m = \sum_{s \in S} \pi(s)$. There exists a Stateless Allocation Algorithm that allocates the elements $s \in S$ to disjoint intervals $\phi(s) \subseteq [(1+O(\varepsilon))m]$ of size $\pi(s)$, and that incurs expected overhead $O(1 + \log \varepsilon^{-1})$.*

Note that Proposition 206, like Theorem 201, offers a space guarantee that changes dynamically with $\sum_{s \in S} \pi(s)$.

Proof. Partition elements into size classes $C_1, C_2, \ldots, C_{1/\varepsilon}$ where C_i contains elements of sizes in $[1 + (i-1)\varepsilon, 1 + i\varepsilon)$. We will treat the elements in C_i as having size exactly $1 + i\varepsilon$.

We will think of memory as broken into ***blocks*** of size $B = \varepsilon^{-1}$. Let r you uniformly random real number in $[0, B]$. We will allocate blocks to size classes so that, at any given moment, size class C_i has at least

$$\left\lceil \frac{(1+\varepsilon)|C_i| + r}{\lfloor B/(1+i\varepsilon) \rfloor} \right\rceil \tag{19.2}$$

blocks allocated to it. The size class will treat each block as an array with $\lfloor B/(1+i\varepsilon) \rfloor$ entries, each of which can store one item of size $(1+i\varepsilon)$. Combined, the blocks allocated to the size class represent an array of size at least $|C_i|$, which the size class can use to perform to store its elements. We call this the ***block array*** for the size class.

This algorithmic approach breaks our original allocation problem into two separate allocation problems: (1) the problem of allocating blocks to size classes; and (2) the problem of allocating the elements within each size class to entries of its block array.

We solve both allocation problems using Theorem 201 with a load factor of $1 - \Theta(\varepsilon)$. Note that, in order for us to be able to apply the theorem, we need that the array size is at least $\Omega(\varepsilon^{-1})$ times the size of the item being allocated—for the first allocation problem, this follows from the assumption that $|S| \geq \Omega(\varepsilon^{-2}) \geq \Omega(B\varepsilon^{-1})$, and for the second allocation problem, this follows from the fact that $B \geq \Omega(\varepsilon^{-1})$.

Using Theorem 201, we can bound the allocations in each of the two allocation problems as follows. For the first allocation problem, each allocation/deallocation of a block incurs $O(1 + \log \varepsilon^{-1})$ expected overhead relative to the block size, and $O(B \log \varepsilon^{-1})$ expected overhead relative to the sizes of the elements being inserted/deleted. By the randomness of r, each insertion/deletion has probability $O(1/|B|)$ of triggering a block allocation/deallocation. So the expected overhead from allocating/deallocating blocks is $O(1+\log \varepsilon^{-1})$ per insertion/deletion. For the second allocation problem, we can simply directly apply Theorem 201 to get an overhead of $O(1 + \log \varepsilon^{-1})$ overhead per insertion/deletion.

It remains to analyze the total space used by our allocation scheme. Note that, by (19.2) (and since $B = \varepsilon^{-1}$), each size class C_i uses total space $(1 + O(\varepsilon))|C_i|(1 + i\varepsilon) + O(\varepsilon^{-1})$ for its blocks. This, in turn, evaluates to

$$(1 + O(\varepsilon)) \sum_{x \in C_i} \pi(x) + O(\varepsilon^{-1}).$$

Summing over the size classes, the total space used by all allocated blocks is a most

$$(1 + O(\varepsilon)) \sum_{x \in S} \pi(x) + O(B\varepsilon^{-2}).$$

Since the allocation of blocks to size classes is performed using an allocator with load factor $1 - \Theta(\varepsilon)$, the total space used by the algorithm is

$$\begin{aligned}(1 + O(\varepsilon) &\left((1 + O(\varepsilon)) \sum_{x \in S} \pi(x) + O(B\varepsilon^{-2})\right) \\ &= (1 + O(\varepsilon)) \sum_{x \in S} \pi(x) + O(B\varepsilon^{-2}).\end{aligned}$$

Finally, since $\sum_{x \in S} \pi(x) \geq \varepsilon^{-3}$ we can conclude that our allocator uses at most $(1 + O(\varepsilon)) \sum_{x \in S} \pi(x)$ space, as desired. ∎

19.1.4 Allocating Objects of Arbitrary Sizes

Finally, we can construct an allocator that can handle objects with arbitrary sizes (of up to $O(\varepsilon^4 m)$). The basic idea will be (1) to partition the objects into groups $C_1, C_2, \ldots$, where each group C_i contains objects whose sizes are within a factor of two of which other; (2) to construct an allocator ϕ_i for each group C_i using Proposition 206; and (3) to implement each ϕ_i using slabs of memory that have power-of-two sizes, and that are allocated to ϕ_i via Theorem 201.

Let m, U be positive integers, let ε^{-1} be a power of two, and let $\rho > 0$ be an upper bound on object size. We shall restrict ourselves to inputs (S, π) where $\sum_{x \in S} \pi(x) \leq m$, and where $\pi(x) \in [1, \rho]$ for all $x \in S$. Subject to these constraints, we will construct a stateless allocator that uses space $(1 + O(\varepsilon))m + O(\varepsilon^{-3}\rho)$ and that incurs expected overhead $O(\varepsilon^{-1})$. Note that, if $\rho \leq O(\varepsilon^4 m)$, then this implies a space bound of $(1 + O(\varepsilon^{-1}))m$.

For $i \in \{0, 1, 2, \ldots, \lfloor \lg \rho \rfloor\}$, define the ***group*** $C_i = \{x \in S \mid 2^i \leq \pi(x) < 2^{i+1}\}$. We assume without loss of generality that $|C_i| \geq \varepsilon^{-3}$ for all $i \leq \lg \rho$, as we can add ε^{-3} dummy elements to each C_i while increasing $\sum_{x \in S} \pi(x)$ by at most $O(\varepsilon^{-3}\rho)$. Under this ***minimum-group-size assumption***, it suffices to show that our allocator uses space $(1 + O(\varepsilon))m$.

Let ϕ_i be the allocation that Proposition 206 would produce for C_i (note that Proposition 206 requires $|C_i| \geq \Omega(\varepsilon^{-3})$, hence the need for the minimum-group-size assumption).

Let $\ell_i = \varepsilon^{-1} 2^i$. We use Theorem 201 to allocate ***slabs*** $L_1^{(i)}, L_2^{(i)}, \ldots$ of size ℓ_i to the group C_i. These slabs are used to implement ϕ_i as follows: if ϕ_i allocates an object $x \in S$ to $[a_x, b_x]$, then the object is actually allocated to slab $L^{(i)}_{\lfloor a_x/(\ell_i - 2^i) \rfloor}$, occupying the sub-interval $[t, t + \pi(x)]$ where $t = a_x \mod (\ell_i - 2^i)$.

It is straightforward to bound the space consumed by the slabs that implement ϕ_i:

Lemma 207. *Assume the minimum-group-size assumption. At any given moment, the slabs allocated to ϕ_i will have cumulative size at most $(1 + O(\varepsilon)) \sum_{x \in C_i} \pi(x)$.*

Proof. By construction, if ϕ_i were implemented in a contiguous array (instead of using slabs), it would use space $q = (1 + O(\varepsilon)) \sum_{x \in C_i} \pi(x)$. On the other hand, the number of slabs that ϕ_i uses is bounded by $q/(\ell_i - 2^i)$. As each of the slabs has size ℓ_i, the cumulative size of the slabs is

$$\frac{\ell_i q}{\ell_i - 2^i} = \frac{\varepsilon^{-1} 2^i q}{\varepsilon^{-1} 2^i - 2^i} = (1 + O(\varepsilon))q.$$

∎

Next we bound the frequency with which slabs are allocated and deallocated to a given group.

Lemma 208. *Assume the minimum-group-size assumption. Each insertion/deletion in C_i changes the number of slabs allocated to ϕ_i by $O(\varepsilon)$ in expectation.*

Proof. The number of slabs allocated to ϕ_i changes whenever ϕ_i performs a resizing-related rebuild. Recall that, if $\sum_{x \in C_i} \pi(x) = m_i$ for some m_i, then the probability of a given insertion/deletion in C_i triggering a resizing-related rebuild is $O(1/(\varepsilon m_i))$; moreover, if such a rebuild occurs, then the memory usage of ϕ_i changes by $O(\varepsilon m_i)$, so the number of slabs that must be added or removed is $O(\varepsilon^2 m_i)$. Thus, whenever an insertion or deletion occurs in C_i, the expected number of slabs that must be allocated or removed from C_i (via Theorem 201) is $O(\varepsilon)$. ∎

We can now analyze our algorithm. Note that, once again, our space guarantee is a function of $\sum_{x \in S} \pi(x)$, rather than a fixed m.

Theorem 209. *Let U, ε^{-1} be positive integers, and let $\rho > 0$ be an upper bound on object size. Consider inputs (S, π) where $S \subseteq [U]$, where $\pi(x) \in [1, \rho]$ for all $x \in S$. Finally, let m denote $\sum_{x \in S} \pi(x)$. Then one can construct a stateless allocator that uses space $(1 + O(\varepsilon))m + O(\varepsilon^{-3}\rho)$ and that incurs expected overhead $O(1 + \log \varepsilon^{-1})$ per insertion and deletion.*

Proof. We assume without loss of generality that ε^{-1} is a power of two. As discussed above, we can also make the minimum-group-size assumption without loss of generality (this is the reason for the $O(\varepsilon^{-3}\rho)$ term in the space bound). Under this assumption, Lemma 207 tell us that the total space consumed by the slabs allocated to groups $\{C_i\}$ is at most $(1+O(\varepsilon))m$. Since the slabs are allocated via Algorithm 4, which itself has load factor $1-\varepsilon$, the total space used by our algorithm is $(1+O(\varepsilon))m$.

Now consider an insertion into some group C_i. There are two sources of overhead: (1) changes to the allocation ϕ_i; and (2) the cost of allocating new slabs to ϕ_i via Theorem 201. By Proposition 206, the changes to ϕ_i contribute $O(1 + \log \varepsilon^{-1})$ to expected overhead. By Lemma 208, the expected number of new slabs that must be allocated by Algorithm 4 (as a result of the insertion) is $O(\varepsilon)$. Each such slab has size $\ell_i = \Theta(\varepsilon^{-1})$ and is allocated with expected overhead $O(1+\log \varepsilon^{-1})$ (by Theorem 191). So the expected overhead incurred per insertion/deletion to allocate/deallocate new slabs is $O(1 + \log \varepsilon^{-1})$. This completes the proof. ∎

As an immediate corollary, we get:

Theorem 193. *Let U, ε^{-1} be positive integers. Consider inputs (S, π) where $S \subseteq [U]$, and where $\pi(x) \in [1, \varepsilon^4 \sum_{x \in S} \pi(x)]$ for all $x \in S$. Then one can construct a stateless allocator that uses space $(1 + O(\varepsilon)) \sum_{x \in S} \pi(x)$ and that incurs expected overhead $O(1 + \log \varepsilon^{-1})$ per insertion and deletion.*

19.2 Achieving polylog n Worst-Case Overhead for Worker-Task Assignment

In this section, we prove polylog n worst-case bounds on the overhead (i.e., switching cost) needed to solve the memoryless worker-task assignment problem. As discussed in the introduction of the chapter, we will adapt for this section the conventions that are typically used [334, 335] to discuss the problem in the distributed-computing literature.

We will prove two results. In Subsection 19.2.1, we give a non-constructive solution with log-squared worst-case cost.

Theorem 194. *There exists a worker-task assignment function that achieves switching cost $O(\log w \log(wt))$.*

Then, in Subsection 19.2.2, we show how to use derandomization techniques in order to obtain a constructive solution with polylogarithmic cost.

Theorem 196. *There is an explicit worker-task assignment function that achieves switching cost $O(\text{polylog}(wt))$.*

Prior to these solutions, it remained an open question to achieve any bounds that were sublinear in both w and t [334, 335].

Review of problem statement. As review, before we continue into the section, we take a moment to repeat the formal problem statement for memoryless worker-task assignment.

There are w workers $1, 2, \ldots, w$ and t tasks $1, 2, \ldots, t$. A ***worker-task assignment function*** ϕ is a function that takes as input a multiset T of w tasks, and produces an assignment of workers to tasks such that the number of workers assigned to a given task $\tau \in T$ is equal to the multiplicity of τ in T.

Two task multisets T_1, T_2 of size w are said to be *adjacent* if they agree on exactly $w - 1$ elements; that is, $|T_1 \setminus T_2| = |T_2 \setminus T_1| = 1$.[2] The ***switching cost*** between two adjacent task multisets T_1, T_2 of size w is defined as the number of workers whose assignment changes between $\phi(T_1)$ and $\phi(T_2)$. The ***switching cost of*** ϕ is defined to be the maximum switching cost over *all* pairs of adjacent task multisets. The goal of the worker-task assignment problem is to design a worker-task assignment function

[2]Let $m_A(i)$ denote the number of times element i appears in multiset A. Then, for any two multisets A and B, we define multisets $A \setminus B$, $A \cup B$, and $A \cap B$ to be such that $m_{A \setminus B}(i) = \max(0, m_A(i) - m_B(i))$, $m_{A \cup B}(i) = \max(m_A(i), m_B(i))$, and $m_{A \cap B}(i) = \min(m_A(i), m_B(i))$, for every element i.

with the minimum possible switching cost.

Of course, if we restrict ourselves to sets T_1 and T_2 (rather than multi-sets), and if we consider the cost of a single insertion/deletion (rather than switching cost), then this becomes the Fixed-Size Stateless Allocation Problem with $n = w$, $U = t$, and $\varepsilon^{-1} = n$, and where our focus is on *worst-case* overhead. It is this worst-case nature of the problem that makes memoryless worker-task assignment so interesting—we must construct low-overhead strongly history-independent allocations, but they cannot be randomized.

19.2.1 Achieving Switching Cost $O(\log w \log(wt))$

Recall that the task multiset T is a multiset of elements from $[t]$. When discussing a multiset S, we use $m_S(x)$ to denote the multiplicity of each element x in S. If S consists of elements in the range $[j]$ for some j, then we say that $S \subseteq [j]$ and we define $|S| = \sum_{i=1}^{j} m_S(i)$. As a convention, we say that a set of workers $A \subseteq [w]$ is ***assigned to*** a multiset of tasks $B \subseteq [t]$ if for each $j \in [t]$ the number of workers in A that are assigned to task j is $m_B(j)$. We define union and setminus in the standard way for multisets, that is, $m_{A\cup B}(x) = m_A(x)+m_B(x)$ and $m_{A\setminus B}(x) = \max(0, m_A(x)-m_B(x))$. Finally, we define a ***sub-multiset*** of a multiset A to be any multiset B such that $m_B(x) \le m_A(x)$ for all x.

In this subsection, we prove the following theorem.

Theorem 194. *There exists a worker-task assignment function that achieves switching cost $O(\log w \log(wt))$.*

We demonstrate the existence of such a function via the probabilistic method, showing that there is a randomized construction that produces a low-switching cost worker-task assignment function with nonzero probability.

From multisets to sets. We begin by showing that, without loss of generality, we can restrict our attention to task multisets T that are sets (rather than multisets). We reduce from the multiset version of the problem with w workers and t tasks to the set version of the problem with w workers and wt tasks.

Lemma 210. *Define $n = wt$. Let ϕ be a worker-task assignment function that assigns workers $[w]$ to task sets $T \subseteq [n]$ (note that ϕ is defined only on task* sets *T, and not on multisets). Let s be the switching cost of ϕ (considering only pairs of adjacent subsets of $[n]$, rather than adjacent sub-multisets). Then there exists a worker-task assignment function ϕ' assigning workers $[w]$ to task multisets $T \subseteq [t]$, such that ϕ' also has switching cost s.*

Proof. When discussing the assignment function ϕ, we think of its input task-set T as consisting of elements from $[t] \times [w]$ rather than elements of $[tw]$.

With this in mind, we construct ϕ' as follows. Given a task multiset $T \subseteq [t]$, define the set $\mathbf{S}(T) \subseteq [t] \times [w]$ to be $\bigcup_{i=1}^{t} \{(i,1), \ldots, (i, m_T(i))\}$, where $m_T(i)$ is the multiplicity of i in T. The worker-task assignment ϕ produces some bijection

$\psi_{\mathbf{S}(T)} : [w] \to \mathbf{S}(T)$. Similarly, ϕ' should produce some bijection $\psi'_T : [w] \to T$. This bijection is defined naturally by projection: if $\psi_{\mathbf{S}(T)}$ assigns worker j to task (i, x), let ψ'_T assign worker j to task i.

We now compute the switching cost of ϕ'. Let T and T' be two adjacent task multisets, so $T' = T \cup \{a\} \setminus \{b\}$ for some $a, b \in [t]$. Then $\mathbf{S}(T') = \mathbf{S}(T) \cup \{(a, m_T(a) + 1)\} \setminus \{(b, m_T(b))\}$, and so $\mathbf{S}(T')$ is adjacent to $\mathbf{S}(T)$. Since ϕ has switching cost s, $\psi_{\mathbf{S}(T)}$ and $\psi_{\mathbf{S}(T')}$ agree on $w - s$ workers. By construction, ψ'_T and $\psi'_{T'}$ must agree on these $w - s$ workers as well, and so it too has switching cost at most s. ∎

In the remainder of the section, we will make the assumption that T is a subset of $[n]$, and we will show how to design an assignment function with switching cost $O(\log w \log n)$ on all pairs of adjacent subsets of $[n]$. By Lemma 210, setting $n = wt$ then implies Theorem 194.

Designing an assignment function as an algorithm. It will be helpful to think of the function we construct for assigning workers to tasks as an algorithm $\mathcal{A}$, which we call the ***multi-round balls-to-bins algorithm***. The algorithm $\mathcal{A}$ takes as input a set $T \subseteq [n]$ of tasks with $|T| = w$ and must produce a bijection from the workers $[w]$ to T.

The algorithm constructs this bijection in stages. Each stage is what we call a ***partial assignment algorithm***, which takes as input the current sets of workers and tasks that have yet to be matched and assigns some subset of these workers to some subset of the tasks. Formally, we define a partial assignment algorithm to be any function ψ which accepts as input any pair of sets $T \subseteq [n], W \subseteq [w]$ with $|T| = |W|$ and produces a matching between some subset of T and some subset of W. After applying ψ to (T, W), there may remain some unmatched elements $T' \subseteq T$, $W' \subseteq W$. We call (T, W) the ***worker-task input*** to ψ and (T', W') the ***worker-task output***. Since a matching must remove exactly as many elements from T as it does from W, we must also have $|W'| = |T'|$. Consequently, there is a natural notion of the ***composition*** of two partial assignment algorithms: the composition $\psi' \circ \psi$ applies ψ and then ψ', letting the worker-task output of ψ be the worker-task input to ψ'.

The algorithm. At a very high level, the algorithm $\mathcal{A}$ can be summarized as follows. For each i from 1 to $c \log w$, repeat the following hashing procedure $c \log n$ many times. Initialize a hash table consisting of $w/(1.1)^i$ bins and randomly hash each unassigned worker and each unassigned task into this table. For each bin that contains at least one worker and one task, assign the minimum worker in that bin to the minimum task in that bin.

In more detail, our algorithm $\mathcal{A}$ is the composition of $\log_{1.1} w$ partial-assignment algorithms,

$$\mathcal{A} = \mathcal{A}_1 \circ \mathcal{A}_2 \circ \cdots \circ \mathcal{A}_{\log_{1.1} w}.$$

Let c be a large positive constant. Each $\mathcal{A}_i$ is itself the composition of $c \log n$ partial-assignment algorithms,

$$\mathcal{A}_i = \mathcal{A}_{i,1} \circ \mathcal{A}_{i,2} \circ \cdots \circ \mathcal{A}_{i,c\log n}.$$

Designing the parts. Each $\mathcal{A}_{i,j}$ assigns workers to tasks using what we call a $w/(1.1)^i$***-bin hash***, which we define as follows.

For a given parameter k, a k***-bin hash*** selects functions $h_1 : [w] \to [k]$ and $h_2 : [n] \to [k]$ independently and uniformly at random. For each worker $\omega \in [w]$, we say that ω is ***assigned*** to bin $h_1(\omega)$. Similarly, for each $\tau \in [n]$ we say τ is assigned to $h_2(\tau)$. These functions are then used to construct a partial assignment. Given a worker-task input (W, T), we restrict our attention to only the assignments of workers in W and tasks in T. In each bin $\kappa \in [k]$ with at least one worker and one task assigned, match the smallest such worker to the smallest such task. Importantly, once h_1 and h_2 are fixed, the algorithm $\mathcal{A}_{i,j}$ uses this same pair of hash functions for every worker-task input, which (as we will see later) is what allows it to make very similar assignments for similar inputs and achieve low switching cost.

We set each $\mathcal{A}_{i,j}$ to be an independent random instance of the k-bin hash, where $k = w/(1.1)^i$. Formally, this means that the algorithm $\mathcal{A} = \mathcal{A}_{1,1} \circ \cdots \circ \mathcal{A}_{\log_{1.1} w, c\log n}$ is a random variable whose value is a partial-assignment function. Our task is thus to prove that, with non-zero probability, $\mathcal{A}$ fully assigns all workers to tasks and has small switching cost.

Analyzing the algorithm. In Section 19.2.1, we show that $\mathcal{A}$ *deterministically* has switching cost $O(\log w \log n)$.

Although $\mathcal{A}$ always has small switching cost, the algorithm is *not* always a legal worker-task assignment function. This is because the algorithm may sometimes act as a *partial* worker-task assignment function, leaving some workers and tasks unassigned.

In Section 19.2.1, we show that with probability greater than 0 (and, in fact, with probability $1 - 1/\operatorname{poly} n$), the algorithm $\mathcal{A}$ succeeds at fully assigning workers to tasks for *all* worker-task inputs (W, T). Theorem 194 follows by the probabilistic method.

Bounding the Probability of Failure

Call a partial-assignment algorithm ψ ***fully-assigning*** if for every worker/task input (W, T), ψ assigns all of the workers in W to tasks in T. That is, ψ never leaves workers unassigned.

Proposition 211. *The multi-round balls-to-bins algorithm $\mathcal{A}$ is fully-assigning with high probability in n. That is, for any polynomial $p(n)$, if the constant c used to define $\mathcal{A}$ is sufficiently large, then $\mathcal{A}$ is fully-assigning with probability at least $1 - O(1/p(n))$.*

Proposition 211 tells us that with high probability in n, $\mathcal{A}$ succeeds at assigning all workers on *all* inputs. We remark that this is a much stronger statement than saying that $\mathcal{A}$ succeeds with high probability in n on a *given* input (W, T).

The key to proving Proposition 211 is to show that each $\mathcal{A}_i$ performs what we call $(w/(1.1)^i)$***-halving***. A partial-assignment function ψ is said to perform k***-halving*** if for every worker/task input (W, T) of size at most $1.1k$, the worker-task output (W', T') for $\psi(W, T)$ has size at most k.

If every $\mathcal{A}_i$ performs $w/(1.1)^i$-halving, then it follows that

$$\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_{\log_{1.1} w}$$

is a fully-assigning algorithm. Thus our task is to show that each $\mathcal{A}_i$ performs $w/(1.1)^i$-halving with high probability in n.

We begin by analyzing the k-bin hash on a given worker/task input (W, T).

Lemma 212. *Let ψ a randomly selected k-bin hash. Let (W, T) be a worker/task input satisfying $|W| = |T| \le 1.1k$, and let (W', T') be the worker/task output of $\psi(W, T)$. The probability that (W', T') has size k or larger is $2^{-\Omega(k)}$.*

Proof. We may assume that $|W| = |T| \ge k$, else the conclusion is trivially true. Let X be the random variable denoting the number of worker/task assignments made by $\psi(W, T)$. Equivalently, X counts the number of bins to which at least one worker is assigned and at least one task is assigned—call these the ***active bins***. We will show that $\Pr[X < \frac{k}{8}] \le 2^{-\Omega(k)}$. Since $|W| = |T| \le 1.1k$, this immediately implies that $|W'| = |T'| \le 1.1k - 0.125k \le k$ with probability $1 - 2^{-\Omega(k)}$, as desired.

We begin by computing $\mathbb{E}[X]$. For each bin $j \in [k]$, the probability no workers are assigned to bin j is $(1 - 1/k)^{|W|} \le (1 - 1/k)^k \le 1/e$. Similarly, the probability that no tasks are assigned to bin j is at most $(1 - 1/k)^{|T|} \le 1/e$. The probability of bin j being active is therefore at least $1 - 2/e \ge 1/4$. By linearity of expectation, $\mathbb{E}[X] \ge k/4$.

Next we show that the random variable X is tightly concentrated around its mean. Because the bins that are active are not independent of one-another, we cannot apply a Chernoff bound. Instead, we employ McDiarmid's inequality:

Theorem 213 (McDiarmid '89 [256])**.** *Let $A_1, \ldots, A_m$ be independent random variables over an arbitrary probability space. Let F be a function mapping $(A_1, \ldots, A_m)$ to $\mathbb{R}$, and suppose F satisfies,*

$$\sup_{a_1, a_2, \ldots, a_m, \overline{a_i}} |F(a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_m) - F(a_1, a_2, \ldots, a_{i-1}, \overline{a_i}, a_{i+1}, \ldots, a_m)| \le R,$$

for all $1 \le i \le m$. That is, if $A_1, A_2, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m$ are fixed, then the value of A_i can affect the value of $F(A_1, \ldots, A_m)$ by at most R. Then for all $S > 0$,

$$\Pr[|F(A_1, \ldots, A_m) - \mathbb{E}[F(A_1, \ldots, A_m)]| \ge R \cdot S] \le 2e^{-2S^2/m}.$$

The number of active bins X is a function of at most $2.2 \cdot k$ independent random variables (namely, the hashes $h_1(\omega)$ for each $\omega \in W$ and the hashes $h_2(\tau)$ for each $\tau \in T$). Each of these random variables can individually change the number of active bins by at most one. It follows that we can apply McDiarmid's inequality with $R = 1$ and $m = 2.2k$. Taking $S = k/8$, we obtain

$$\Pr[|X - \mathbb{E}[X]| \geq k/8] \leq e^{-\Omega(k)}.$$

Since $\mathbb{E}[X] \geq k/4$, we have that $\Pr[X < k/8] \leq e^{-\Omega(k)}$, which completes the proof of the lemma. ∎

Our next lemma shows that each $\mathcal{A}_i$ is k-halving with high probability in n, where $k = w/(1.1)^i$.

Lemma 214. *Let $\psi_1, \ldots, \psi_{c \log n}$ be independent random k-bin hashes, and let $\psi = \psi_1 \circ \cdots \circ \psi_{c \log n}$. With high probability in n, ψ is k-halving. That is, every worker-task input (W, T) with $|W| = |T| \leq 1.1k$ has a worker task output (W', T') with $|W'| = |T'| \leq k$.*

Proof. Fix an arbitrary worker-task input (W, T) with $|W| = |T| \leq 1.1k$. Let (W_i, T_i) denote the worker-task output after applying the first i rounds, $\psi_1 \circ \cdots \circ \psi_i$. Let p_i denote the probability that $|W_i| = |T_i| > k$.

First, we observe that $p_i \leq e^{-\Omega(k)} p_{i-1}$ for all $i > 1$. Indeed, for $|W_i| = |T_i| > k$, we must necessarily have $|W_{i-1}| = |T_{i-1}| > k$, which occurs with probability p_{i-1}, but in this situation, the probability that ψ_i produces a worker-task output of size greater than k is a further $e^{-\Omega(k)}$ by Lemma 212.

The probability that ψ fails to reduce the size of (W, T) to k or smaller is thus at most

$$p_{c \log n} \leq e^{-\Omega(ck \log n)} \leq n^{-\Omega(ck)}, \tag{19.3}$$

where c is treated as a parameter.

On the other hand, the number of possibilities for input pairs (W, T) satisfying $|W| = |T| \leq 1.1k$ is

$$\sum_{j=0}^{1.1k} \binom{w}{j} \binom{n}{j} \leq 1.1k \cdot w^{1.1k} n^{1.1k} \leq n^{O(k)}. \tag{19.4}$$

Combining (19.3) and (19.4), the probability that there exists *any* pair (W, T) of size $1.1k$ or smaller which fails to have its size reduced to k or smaller is at most $n^{O(k) - c\Omega(k)}$. If c is selected to be a sufficiently large constant, then it follows that ψ performs k-halving with probability at least $1 - n^{-\Omega(k)}$. ∎

We now prove Proposition 211.

Proof of Proposition 211. By Lemma 214, each algorithm $\mathcal{A}_i$ is $(w/(1.1)^i)$-halving with high probability in n. By a union bound, it follows that all of $\mathcal{A}_i \in \{\mathcal{A}_1, \ldots, \mathcal{A}_{\log_{1.1} w}\}$ are $(w/(1.1)^i)$-halving with high probability in n. If this occurs, then

$$\mathcal{A} = \mathcal{A}_1 \circ \cdots \circ \mathcal{A}_{\log_{1.1} w}$$

is fully-assigning, as desired. ∎

Bounding the switching cost

Recall that two worker/task inputs (W_1, T_1) and (W_2, T_2) are said to be ***unit distance*** if

$$W_1 \setminus W_2| + |W_2 \setminus W_1| + |T_1 \setminus T_2| + |T_2 \setminus T_1| \leq 2.$$

A partial-assignment algorithm ψ is ***s-switching-cost bounded*** if for all unit-distance pairs of worker/task inputs (W_1, T_1) and (W_2, T_2), the set of assignments made by $\psi(W_1, T_1)$ deterministically differs from the set of assignments made by $\psi(W_2, T_2)$ by at most s.

In this section, we prove the following proposition.

Proposition 215. *The multi-round balls-to-bins algorithm is $O(\log w \log n)$-switching-cost bounded.*

We begin by showing that each of the algorithms $\mathcal{A}_{i,j}$ are $O(1)$-switching-cost bounded.

Lemma 216. *For any k, the k-bin hash algorithm is $O(1)$-switching-cost bounded.*

Proof. Let ψ denote the k-bin hash algorithm. Consider unit-distance pairs of worker/task inputs (W_1, T_1) and (W_2, T_2). Changing W_1 to W_2 can change the assignments made by ψ for at most a constant number of bins. Similarly changing T_1 to T_2 can change the assignments made by ψ for at most a constant number of bins. Thus $\psi(W_1, T_1)$ differs from $\psi(W_2, T_2)$ by at most $O(1)$ assignments. ∎

Recall that $\mathcal{A}$ is the composition of the $O(\log w \log n)$ partial-assignment algorithms $\mathcal{A}_{i,j}$'s. The fact that each $\mathcal{A}_{i,j}$ is $O(1)$-switching-cost bounded does not directly imply that $\mathcal{A}$ is $O(\log w \log n)$-switching-cost bounded, however, because switching cost does not necessarily interact well with composition. In order to analyze $\mathcal{A}$, we show that each $\mathcal{A}_{i,j}$ satisfies an additional property that we call being composition-friendly.

A partial-assignment algorithm ψ is ***composition-friendly***, if for all unit-distance pairs of worker/task inputs (W_1, T_1) and (W_2, T_2), the corresponding worker/task outputs (W_1', T_1') and (W_2', T_2') are also unit-distance.

Lemma 217 shows that each $\mathcal{A}_{i,j}$ is composition-friendly.

Lemma 217. *For any k, the k-bin hash is composition-friendly.*

Proof. Although the algorithm ψ is formally only defined on input (W, T) for which $|W| = |T|$, we will abuse notation here and consider ψ even on worker/task input (W, T) satisfying $|W| \neq |T|$.[3] Define the ***difference-score*** of a pair of worker/task

[3]Indeed, the definition of the k-bin hash does not require a worker-task input with $|W| = |T|$. The only reason we require this equality in general is to simplify calculations, as in practice the algorithm will only be run on worker-task inputs of equal size.

inputs $I_1 = (W_1, T_1), I_2 = (W_2, T_2)$ to be the quantity

$$d(I_1, I_2) = |W_1 \setminus W_2| + |W_2 \setminus W_1| + |T_1 \setminus T_2| + |T_2 \setminus T_1|.$$

We will show the stronger statement that the difference-score $d(O_1, O_2)$ of the corresponding worker/task outputs $O_1 = (W_1', T_1'), O_2 = (W_2', T_2')$ satisfies

$$d(O_1, O_2) \leq d(I_1, I_2). \tag{19.5}$$

It suffices to consider only two special cases: the case in which $W_2 = W_1 \cup \{\omega\}$ for some worker ω and $T_2 = T_1$; and the case in which $T_2 = T_1 \cup \{\tau\}$ for some task τ and $W_2 = W_1$. Iteratively applying these two cases to transform I_1 into I_2 implies inequality 19.5.

For this purpose, the roles of W and T are identical, so suppose without loss of generality that $W_2 = W_1 \cup \{\omega\}$ for some worker ω and $T_2 = T_1$. Recall that the assignment of workers and tasks to buckets is determined by some hash functions h_1, h_2 and in particular is the same whether we input W_1 or W_2. We first assign (only) the elements of W_1 and T_1 to their respective buckets, and then look at how including the assignment of ω changes the worker-task output. If h_1 assigns ω to either a bin with no tasks or a bin which already has some lexicographically smaller worker, then we will have $W_2' = W_1' \cup \{w\}$ and $T_2' = T_1'$. If h_1 assigns worker ω to a bin with no other workers and at least one task, we let the smallest such task be τ and see $W_2' = W_1'$ and $T_2' = T_1' \setminus \{\tau\}$. Finally, if h_1 assigns ω to a bin with only larger workers and at least one task, we let the minimal such worker be γ, and we see $W_2' = W_1' \cup \{\gamma\}$ and $T_2' = T_1'$. In all three cases, $d(O_1, O_2) = 1$, as desired. ∎

Next, we will show that composing composition-friendly algorithms has the effect of summing switching costs.

Lemma 218. *Suppose that partial-assignment algorithms $\psi_1, \psi_2, \ldots, \psi_k$ are all composition-friendly, and that each ψ_i is s_i-switching-cost bounded. Then $\psi_1 \circ \psi_2 \circ \cdots \circ \psi_k$ is composition-friendly and is $(\sum_i s_i)$-switching-cost-bounded.*

Proof. By induction, it suffices to prove the lemma for $k = 2$. Let $I_1 = (W_1, T_1)$ and $I_2 = (W_2, T_2)$ be unit-distance worker/task inputs.

For $i \in \{1, 2\}$, let $I_i' = (W_i', T_i')$ be the worker/task output for $\psi_1(W_i, T_i)$, and let $I_i'' = (W_i'', T_i'')$ be the worker/task output for $\psi_2(W_i', T_i')$.

Since ψ_1 is composition friendly, its outputs I_1' and I_2' are unit distance. Since I_1' and I_2' are unit distance, and since ψ_2 is composition friendly, the outputs I_1'' and I_2'' of ψ_2 are also unit distance. Thus $\psi_1 \circ \psi_2$ is composition friendly.

Since the inputs I_1 and I_2 to ψ_1 are unit-distance, $\psi_1(I_1)$ and $\psi_1(I_2)$ differ in at most s_1 worker-task assignments. Since the inputs I_1' and I_2' to ψ_2 are also unit distance, $\psi_2(I_1')$ and $\psi_2(I_2')$ differ in at most s_2 worker-task assignments. Thus the composition $\psi_1 \circ \psi_2$ is $(s_1 + s_2)$-switching-cost bounded, as desired. ∎

We can now prove Proposition 215.

Proof of Proposition 215. By Lemma 216, each $\mathcal{A}_{i,j}$ is $O(1)$-switching-cost bounded. By Lemma 217, each $\mathcal{A}_{i,j}$ is composition friendly. Since $\mathcal{A}$ is the composition of the $O(\log w \log n)$ different $\mathcal{A}_{i,j}$'s, it follows by Lemma 218 that $\mathcal{A}$ is $O(\log w \log n)$-switching-cost bounded. ∎

19.2.2 Derandomizing the Construction

In this subsection, we derandomize the multi-round balls-to-bins algorithm to prove the following theorem.

Theorem 196. *There is an explicit worker-task assignment function that achieves switching cost* $O(\mathrm{polylog}(wt))$.

To this end, we use pseudorandom objects called *strong dispersers*. Intuitively, a disperser is a function such that the image of any not-too-small subset of its large domain (e.g., workers or tasks) is a dense subset of its small co-domain (e.g., bins). Since this requirement is hard to satisfy directly, dispersers are defined with a second argument, called the seed. For a strong disperser, the density requirement is satisfied only in expectation over the seed. The standard way to define strong dispersers (Definition 219 below) is in the language of random variables. We follow with an equivalent alternative Definition 220, more convenient for our purposes.

Definition 219 (Strong dispersers)**.** *For $k \in \mathbb{N}$, $\varepsilon \in \mathbb{R}_+$, a (k, ε)-strong disperser is a function $Disp : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ such that for any random variable X over $\{0,1\}^n$ with min-entropy at least k we have*

$$|\operatorname{Supp}((Disp(X, U_d), U_d))| \geq (1 - \varepsilon) \cdot 2^{m+d}.$$

Here Supp denotes the support of a random variable, U_d denotes the uniform distribution on $\{0,1\}^d$, and the min-entropy of a random variable X is defined as $\min_x(-\log_2(\Pr[X = x]))$. We will use a simple fact that any distribution which is uniform on a 2^k-element subset of the universe and assigns zero probability elsewhere (called *flat k-source* in pseudorandomness literature) has min-entropy k. Interestingly, every distribution with min-entropy at least k is a convex combination of such distributions (see, e.g., Lemma 6.10 in [344], first proved in [126]), which makes the following definition equivalent.

Definition 220 (Strong dispersers, alternative definition)**.** *For $k \in \mathbb{N}$, $\varepsilon \in \mathbb{R}_+$, a (k, ε)-strong disperser is a function $Disp : [N] \times [D] \to [M]$ such that for any subset $S \subseteq [N]$ of size $|S| \geq 2^k$ we have*

$$|\{(Disp(s, d), d) : s \in S, d \in [D]\}| \geq (1 - \varepsilon) \cdot M \cdot D.$$

We use efficient explicit strong dispersers constructed by Meka, Reingold and Zhou [261].

Theorem 221 (Theorem 6 in [261])**.** *For all $N = 2^n$, $k \in \mathbb{N}$, and $\varepsilon \in \mathbb{R}_+$, there exists an explicit (k, ε)-strong disperser $Disp : [N] \times [D] \to [M]$ with $D = 2^{O(\log n)} =$*

$\operatorname{polylog} N$ *and* $M = 2^{k - 3\log n - O(1)} = 2^k \cdot \Omega(1/\log^3 N)$.

Designing the algorithm We begin with applying Lemma 210 in order to be able to restrict our attention to task sets (rather than multisets), at the expense of increasing the number of tasks from t to wt. For convenience, we round up the new number of tasks to the closest power of two $N = 2^{\lceil \log wt \rceil}$.

Our explicit algorithm $\mathcal{E}$ has the same structure as the randomized algorithm $\mathcal{A}$, i.e. it is the composition of $\log w$ partial assignment algorithms

$$\mathcal{E} = \mathcal{E}_1 \circ \mathcal{E}_2 \circ \cdots \circ \mathcal{E}_{\log w}.$$

Each $\mathcal{E}_i$ is responsible for bringing down the number of unassigned workers to the next power of two, and is composed of a number of explicit sub-algorithms $\mathcal{E}_{i,j}$'s. Contrary to $\mathcal{A}_{i,j}$'s, sub-algorithms $\mathcal{E}_{i,j}$'s are not identical copies for a fixed i. However, the chain of distinct sub-algorithms has to be copied $O(\log^3 N)$ times. We reflect this introducing the $\widehat{\mathcal{E}_i}$ notation:

$$\mathcal{E}_i = \underbrace{\widehat{\mathcal{E}_i} \circ \widehat{\mathcal{E}_i} \circ \cdots \circ \widehat{\mathcal{E}_i}}_{O(\log^3 N)\text{ times}}, \quad \text{where} \quad \widehat{\mathcal{E}_i} = \mathcal{E}_{i,1} \circ \mathcal{E}_{i,2} \circ \cdots \circ \mathcal{E}_{i,\operatorname{polylog} N}.$$

The key difference between the randomized and explicit algorithm is that $\mathcal{E}_{i,j}$'s, instead of using random hash functions h_1, h_2, use explicit functions obtained from strong dispersers. Another notable difference is that $\mathcal{A}_{i,j}$'s use k bins to deal with input sets of size in $[k, 1.1k]$, while $\mathcal{E}_{i,j}$'s have to use polylogarithmically less bins, limiting the number of worker-task pairs that can be assigned by a single sub-algorithm and, as a consequence, forcing us to compose a larger number of sub-algorithms.

Let us fix $i \in [\log w]$, and denote $k_i = \lceil \log w \rceil - i$. Let $Disp_i : [N] \times [D_i] \to [M_i]$ be the $(k_i, 1/4)$-strong disperser given by Theorem 221. Recall that $D_i = \operatorname{polylog} N$, $M_i = 2^{k_i} \cdot \Omega(1/\log^3 N)$, and N is large enough so that all workers and all tasks are elements of $[N]$. We will have $\widehat{\mathcal{E}_i} = \mathcal{E}_{i,1} \circ \mathcal{E}_{i,2} \circ \cdots \circ \mathcal{E}_{i,D_i}$. For each $j \in [D_i]$, sub-algorithm $\mathcal{E}_{i,j}$ assigns workers and tasks to M_i bins. Each worker $\omega \in W$ is assigned to bin $Disp_i(\omega, j)$, and each task $\tau \in T$ is assigned to bin $Disp_i(\omega, j)$. Then, like in the randomized strategy, for each active bin (i.e. one which was assigned nonempty sets of workers and tasks) the smallest worker and the smallest task in that bin get assigned to each other.

Analyzing switching cost In Section 19.2.1, where we analyze the switching cost of randomized multi-round balls-to-bins algorithm, we do not exploit the fact that the hash functions h_1, h_2 are random. Actually, as we already remark, our switching cost bound is deterministic and thus works for any choice of functions h_1, h_2. Therefore the same analysis works for the explicit algorithm. Namely, each sub-algorithm $\mathcal{E}_{i,j}$ is $O(1)$-switching cost bounded and composition-friendly (Lemmas 216 and 217 generalize trivially), thus the switching cost of $\mathcal{E}$ depends only on the number of sub-algorithms, which is $\operatorname{polylog} N = \operatorname{polylog} wt$, as desired.

Proving the algorithm is fully-assigning We begin by analyzing the number of worker/task assignments made by $\widehat{\mathcal{E}_i} = \mathcal{E}_{i,1} \circ \cdots \circ \mathcal{E}_{i,D_i}$.

Lemma 222. *Let (W,T) be a worker/task input satisfying $|W| = |T| \geq 2^{k_i}$. Then $\widehat{\mathcal{E}_i}(W,T)$ makes at least $M_i/4$ worker/task assignments.*

Proof. By the definition of dispersers, the two images

$$\{(Disp_i(\omega, j), j) : \omega \in W, j \in [D_i]\}, \quad \text{and} \quad \{(Disp_i(\tau, j), j) : \tau \in T, j \in [D_i]\}$$

have size at least $(3/4) \cdot M_i \cdot D_i$. Since they are both subsets of $[M_i] \times [D_i]$, their intersection has size at least $(1/2) \cdot M_i \cdot D_i$. By the pigeonhole principle, there must exist $j \in D_i$ such that

$$|Disp_i(W,j) \cap Disp_i(T,j)| \geq M_i/2. \tag{19.6}$$

Let us fix such j, and look at the execution of $\mathcal{E}_{i,j}$. For each bin $b \in Disp_i(W,j) \cap Disp_i(T,j)$, if b is not active, then all workers $\{\omega \in W \mid Disp_i(\omega, j) = b\}$ or all tasks $\{\tau \in T \mid Disp_i(\tau, j) = b\}$ must have been already assigned by $(\mathcal{E}_{i,1} \circ \cdots \circ \mathcal{E}_{i,j-1})(W,T)$. Thus, each bin in $Disp_i(W,j) \cap Disp_i(T,j)$ either is active – and contributes one worker and one task to the assignment – or is inactive and testifies that at least one worker or at least one task is assigned by earlier sub-algorithms. Let c_a denote the number of active bins, c_w denote the number of inactive bins testifying for a worker assigned by earlier sub-algorithms, and c_t denote the number of inactive bins testifying for a task. We have $c_a + c_w + c_t \geq M_i/2$, by Inequality (19.6). It follows that the number of worker/task assignments made by $(\mathcal{E}_{i,1} \circ \cdots \circ \mathcal{E}_{i,j})(W,T)$ is at least $c_a + \max(c_w, c_t) \geq c_a + \frac{1}{2}(c_w + c_t) \geq M_i/4$, as desired. ∎

Recall that $M_i = 2^{k_i} \cdot \Omega(1/\log^3 N)$. Thus, Lemma 222 implies that each $\mathcal{E}_i$ – which is a composition of $O(\log^3 N)$ copies of $\widehat{\mathcal{E}_i}$ – when given a worker/task input of size at most $2 \cdot 2^{k_i}$ returns a worker/task output of size at most 2^{k_i}. It follows that $\mathcal{E} = \mathcal{E}_1 \circ \mathcal{E}_2 \circ \cdots \circ \mathcal{E}_{\log w}$ is fully-assigning, which concludes the proof of Theorem 196.

Chapter 20

Strong Lower Bounds for Stateless Allocation

In this chapter, we prove a lower bound of $\Omega(\log \varepsilon^{-1}/\log\log \varepsilon^{-1})$ on the expected cost incurred by any fixed-size stateless allocation algorithm.

Theorem 223. *Let $\varepsilon \geq 1/n$. Any fixed-size stateless allocation algorithm* ALG *must incur $\Omega(\log \varepsilon^{-1}/\log\log \varepsilon^{-1})$ expected cost per insertion/deletion.*

Our lower bound matches the upper bounds achieved in Chapter 19, for both the fixed-size and variable-size versions of the problem, up to $O(\log\log \varepsilon^{-1})$ factors. Prior to our result, the best known lower bound stood at $\Omega(1)$ [334, 335] for the *worst-case* cost incurred by any fixed-size stateless allocation algorithm.

An immediate consequence of our lower bound is that there is a separation between what can be achieved by weakly history-independent strategies (which get $O(1)$ overhead for the fixed-size case) and strongly history-independent strategies (which must incur $\Omega(1 + \log \varepsilon^{-1})$ expected overhead).

20.1 Proof of Theorem 223

Let $\varepsilon \geq 1/n$ and let $m = (1-\varepsilon)n$. For any set $S \subseteq [1, 2m]$ satisfying $|S| \leq m+1$, let ALG(S) denote the allocation of elements S to slots $[1, n]$ produced by allocation algorithm ALG (note that the random bits used by the algorithm are left implicit). As a convention, we will use the terms *slot* and *position* interchangeably, and we will refer to ALG(S) as either an *allocation* (of elements to slots) or a *state* (referring to the state of which elements are in which slots).

We will consider a sequence of insertions/deletions as follows: We will start with $S_0 = \{m+1, \ldots, 2m\}$ and then we will perform a sequence of m deletion/insertion pairs where the i-th deletion deletes $m+i$ and the i-th insertion inserts some $\pi_i \in [1, m]$.

We will specify our sequence of operations by an ***input vector*** $\pi = \langle \pi_1, \pi_2, \ldots, \pi_m \rangle$ defined to be any permutation of the numbers $1, \ldots, m$. For any input vector π and for any $i \in [0, m]$, define $S_i(\pi) = \{m + i + 1, \ldots, 2m\} \cup \{\pi_1, \pi_2, \ldots, \pi_i\}$. That is, we can reach S_i from S_{i-1} by deleting $m + i$ and inserting π_i.

For any input vector π and any $i \in [1, m]$ define the ***positions-touched set*** $X_i(\pi) \subseteq [1, n]$ to be the set of positions whose contents change between $\text{ALG}(S_{i-1}(\pi))$ and $\text{ALG}(S_i(\pi))$; and define the ***values-touched set*** $Y_i(\pi) \subseteq [1, 2m]$ to be the set of elements whose position changes between $\text{ALG}(S_{i-1}(\pi))$ and $\text{ALG}(S_i(\pi))$ (this includes the elements $(S_i(\pi) \setminus S_{i-1}(\pi)) \cup (S_{i-1}(\pi) \setminus S_i(\pi))$ that are inserted/deleted). Finally, define the ***kick-out chain*** $K_i(\pi)$ to be the sequence $\langle K_i^{(1)}, K_i^{(2)}, \ldots \rangle$ given by: $K_i^{(0)} = \pi_i$; $K_i^{(1)} =$ position of $K_i^{(0)}$ in $\text{ALG}(S_i(\pi))$; $K_i^{(2)} =$ element that was in position $K_i^{(1)}$ in $\text{ALG}(S_{i-1}(\pi))$ (the chain ends if no such element exists); $K_i^{(3)} =$ position of $K_i^{(2)}$ in $\text{ALG}(S_i(\pi))$ (the chain ends if no such position exists); $K_i^{(4)} =$ element that was in position $K_i^{(3)}$ in $\text{ALG}(S_{i-1}(\pi))$ (the chain ends if no such element exists); $K_i^{(5)} =$ position of $K_i^{(4)}$ in $\text{ALG}(S_i(\pi))$ (the chain ends if no such position exists); etc.

That is, the kick-out chain keeps track of the chain of moves that π_i's insertion triggered between $\text{ALG}(S_{i-1}(\pi))$ and $\text{ALG}(S_i(\pi))$: the insertion of $K_i^{(0)} = \pi_i$ into position $K_i^{(1)}$ displaced an element $K_i^{(2)}$ that was then moved to position $K_i^{(3)}$, which displaced an element $K_i^{(4)}$ that was then moved to position $K_i^{(5)}$, etc. The chain ends when we either displace an element into a free slot (in which case that slot is the final member of the chain) or when we displace the element that is deleted between $\text{ALG}(S_{i-1}(\pi))$ and $\text{ALG}(S_i(\pi))$) (in which case that element is the final member of the chain). Note that, for even j we have $K_i^{(j)} \in Y_i(\pi)$ and for odd j we have $K_i^{(j)} \in X_i(\pi)$. However, not all elements of $Y_i(\pi)$ and $X_i(\pi)$ necessarily appear as elements of the kick-out chain, since $\text{ALG}(S_{i-1}(\pi))$ and $\text{ALG}(S_i(\pi))$ can differ from each other beyond just the differences captured by the kick-out chain.

For each $i \in [n]$, define $\text{slide}(\pi, i, 1), \text{slide}(\pi, i, 2), \ldots$ so that permutation $\text{slide}(\pi, i, j)$ is obtained by sliding π_i from position i to position j in π. That is, if $j \leq i$, then,

$$\text{slide}(\pi, i, j) = \langle \pi_1, \pi_2, \ldots, \pi_{j-1}, \pi_i, \pi_j, \pi_{j+1}, \ldots, \pi_{i-1}, \pi_{i+1}, \pi_{i+2}, \ldots, \pi_n \rangle$$

and if $j > i$, then

$$\text{slide}(\pi, i, j) = \langle \pi_1, \pi_2, \ldots, \pi_{i-1}, \pi_{i+1}, \pi_{i+2}, \ldots, \pi_j, \pi_i, \pi_{j+1}, \pi_{j+2} \ldots, \pi_n \rangle.$$

Throughout the proofs, it will be helpful to make use of the following basic identity: If $i, q, r \in [m]$ such that either $i < q, r$ or $i \geq q, r$, then

$$S_i(\pi) = S_i(\text{slide}(\pi, q, r)). \tag{20.1}$$

Finally, say that π_i is (π, j)***-robust*** in a time window $[t_0, t_1] \subseteq [1, m]$ if all of the kick-out-chains $\{K_t(\text{slide}(\pi, i, t)) \mid t \in [t_0, t_1]\}$ agree on their first j entries (and have length at least j). That is, if we slide π_i to be inserted at any time $t \in [t_0, t_1]$ (instead of at time i), then the kick-out chain triggered by the insertion of π_i has the same first j positions no matter which time t we pick. As a formality, we say that for $j < 0$, all π_is are (π, j)-robust in all time windows.

To prove our lower bound, we will show that if ALG has low cost, then $\mathbb{E}[|K_{m/2}(\pi)|]$ must be $\Omega(\log \varepsilon^{-1} / \log\log \varepsilon^{-1})$. Rather than directly analyzing $|K_{m/2}(\pi)|$, however, we will instead show that with probability at least $1/2$, $\pi_{m/2}$ is $(\pi, \Omega(\log \varepsilon^{-1} / \log\log \varepsilon^{-1}))$-robust in some interval T containing $m/2$. This offers an indirect path by which we can obtain a lower bound on $|K_{m/2}(\pi)|$.

The main task in completing the proof will be to understand how the (π, j)-robustness property interacts with the stateless allocation algorithm ALG. We will show that, if one considers time intervals $T_2 \subseteq T_1$ satisfying $|T_1| \gg |T_2| \gg \varepsilon n$, then no matter what ALG does, the fraction of $\pi_i \in T_2$ that are $(\pi, j-1)$-robust in T_2 will be only slightly smaller (in expectation) than the fraction of $\pi_i \in T_1$ that are (π, j)-robust in T_1. This allows us to perform induction on j in order to lower bound the probability of $\pi_{m/2}$ being $(\pi, \Omega(\log \varepsilon^{-1} / \log\log \varepsilon^{-1}))$-robust in the appropriate interval T.

We begin with two lemmas (one for odd j and one for even j) that characterize the situations in which an element π_i that is $(\pi, j-1)$-robust in an interval T_1 will also be (π, j)-robust in a smaller interval T_2 (the specifics of T_2 will differ slightly between the lemmas). At a high level, these lemmas tell us that, if ALG is to *prevent* π_i to from being (π, j)-robust in T_2, then ALG must touch $K_i^{(j-1)}(\pi)$ during some insertion/deletion in time-window T_2.

Lemma 224. *Let π be an input vector, let $[t_0, t_1] \subseteq [1, m]$ be a time window, and let $j > 0$ be odd. Suppose that π_{t_0} is $(\pi, j-1)$-robust in $[t_0, t_1]$, suppose that $K_{t_0}(\pi)$ has length at least j, and let $\langle K^{(1)}, K^{(2)}, \ldots, K^{(j)} \rangle$ denote the first j elements of $K_{t_0}(\pi)$. Then π_{t_0} is (π, j)-robust in $[t_0, t_1]$ iff $K^{(j-1)} \notin Y_{t_0+1}(\pi) \cup Y_{t_0+2}(\pi) \cup \cdots \cup Y_{t_1}(\pi)$, meaning that element $K^{(j-1)}$ is not touched by any of the deletion/insertion pairs performed in the time window $[t_0 + 1, t_1]$.*

Proof. Begin by considering the case that $K^{(j-1)}$ is deleted by some deletion in the time-window $[t_0, t_1]$. That is, there is some $t \in [t_0, t_1]$ such that $K^{(j-1)} \in S_{t-1}(\pi)$ but $K^{(j-1)} \notin S_t(\pi)$. By the $(\pi, j-1)$-robustness of π_{t_0} in $[t_0, t_1]$, we know that kick-out chains $K_t(\text{slide}(\pi, t_0, t)) = K_t(\pi)$ and $K_{t_0}(\pi)$ have the same first $j-1$ entries $K^{(1)}, K^{(2)}, \ldots, K^{(j-1)}$. Since $K^{(j-1)}$ is deleted between $S_{t-1}(\pi)$ and $S_t(\pi)$, we can conclude that $K^{(j-1)}$ is the final element of $K_t(\text{slide}(\pi, t_0, t))$. Thus the lemma is satisfied, since we have both that $K^{(j-1)} \in Y_t(\pi)$ (since $K^{(j-1)}$ is deleted between $S_{t-1}(\pi)$ and $S_t(\pi)$) and that π_{t_0} is not (π, j)-robust in $[t_0, t_1]$ (since $|K_t(\text{slide}(\pi, t_0, t)| = j-1$).

Suppose for the rest of the proof that K_{j-1} is not deleted by any deletion in the time window $[t_0, t_1]$. Since π_{t_0} is already known to be $(\pi, j-1)$-robust in $[t_0, t_1]$, the condition for it being (π, j)-robust in $[t_0, t_1]$ is that the j-th entries in the kick-out chains

$$\{K_t(\text{slide}(\pi, t_0, t)) \mid t \in [t_0, t]\}$$

are all the same. These j-th entries correspond to the positions where $K^{(j-1)}$ resides in $\text{ALG}(S_t(\text{slide}(\pi, t_0, t))$ for $t \in [t_0, t_1]$ (and since $K^{(j-1)}$ is not deleted in the time window, we know that each of the kick-out chains has a j-th entry). Notice, however, that $\text{ALG}(S_t(\text{slide}(\pi, t_0, t)) = \text{ALG}(S_t(\pi))$ by (20.1), so the j-th entries in the kick-out chains actually correspond to the positions in which $K^{(j-1)}$ resides in $\text{ALG}(S_t(\pi))$ for $t \in [t_0, t_1]$. Therefore, π_{t_0} is (π, j)-robust in $[t_0, t_1]$ iff $K^{(j-1)}$ resides in the same position in all of $\text{ALG}(S_{t_0}(\pi)), \text{ALG}(S_{t_0+1}(\pi)), \ldots, \text{ALG}(S_{t_1}(\pi))$. But this, in turn, holds iff $K^{(j-1)}$ is *not* in any of the values-touched sets $Y_{t_0+1}(\pi) \cup Y_{t_0+2}(\pi) \cup \cdots \cup Y_{t_1}(\pi)$. ∎

Lemma 225. *Let π be an input vector, let $[t_0, t_1] \subseteq [1, m]$ be a time window, and let $j \geq 0$ be even. Suppose that π_{t_1} is $(\pi, j-1)$-robust in $[t_0, t_1]$, suppose that $K_{t_1}(\pi)$ has length at least j, and let $\langle K^{(1)}, K^{(2)}, \ldots, K^{(j)} \rangle$ denote the first j elements of $K_{t_1}(\pi)$. Then π_{t_1} is (π, j)-robust in $[t_0, t_1]$ iff $K^{(j-1)} \notin X_{t_0}(\pi) \cup X_{t_0+1}(\pi) \cup \cdots \cup X_{t_1-1}(\pi)$, meaning that position $K^{(j-1)}$ is not touched by any of the deletion/insertion pairs performed in the time window $[t_0, t_1 - 1]$.*

Proof. Begin by considering the case that position $K^{(j-1)}$ is empty in at least one of the states $\text{ALG}(S_{t_0-1}(\pi)), \ldots, \text{ALG}(S_{t_1-1}(\pi))$. We know that position $K^{(j-1)}$ is *not empty* in state $\text{ALG}(S_{t_1-1}(\pi))$ (since the kick-out chain $K_{t_1}(\pi)$ has length j and therefore displaces some item from position $K^{(j-1)}$). Therefore $K^{(j-1)}$ must be in one of the positions-touched sets $X_{t_0}, \ldots, X_{t_1-1}$. On the other hand, since $K^{(j-1)}$ is empty in some state $\text{ALG}(S_{t-1}(\pi))$ with $t \in [t_0, t_1 - 1]$, we know that the kick-out chain $K_t(\text{slide}(\pi, t_1, t))$ cannot displace any item from position $K^{(j-1)}$. Thus $K_t(\text{slide}(\pi, t_1, t)) = \langle K^{(1)}, \ldots, K^{(j-1)} \rangle$ has length $j-1$, implying that π_{t_1} is *not* (π, j)-robust in $[t_0, t_1]$. This is consistent with the lemma, since we have both that $K^{(j-1)} \in X_{t_0}(\pi) \cup X_{t_0+1}(\pi) \cup \cdots \cup X_{t_1-1}(\pi)$ and that π_{t_1} is not (π, j)-robust in $[t_0, t_1]$.

Suppose for the rest of the proof that position $K^{(j-1)}$ is not empty in any of states $\text{ALG}(S_{t_0-1}(\pi)), \text{ALG}(S_{t_0}(\pi)), \ldots, \text{ALG}(S_{t_1-1}(\pi))$. By (20.1), this is equivalent to $K^{(j-1)}$ not being empty in any of states $\text{ALG}(S_{t_0-1}(\text{slide}(\pi, t_1, t_0)), \text{ALG}(S_{t_0}(\text{slide}(\pi, t_1, t_0 + 1)), \ldots, \text{ALG}(S_{t_1-1}(\text{slide}(\pi, t_1, t_1))$. This implies that each of the kick-out chains in

$$\{K_t(\text{slide}(\pi, t_1, t)) \mid t \in [t_0, t_1]\}$$

contains a j-th entry. Namely, the j-th entry of $K_t(\text{slide}(\pi, t_1, t))$ is the element in position $K^{(j-1)}$ of $\text{ALG}(S_{t-1}(\text{slide}(\pi, t_1, t))$, which by (20.1) is also the element in position $K^{(j-1)}$ of $\text{ALG}(S_{t-1}(\pi))$. Observe that π_{t_1} is (π, j)-robust in $[t_0, t_1]$ iff the j-th entries of these kick-out chains are all equal, that is, iff the element in position $K^{(j-1)}$ is the same for all of $\text{ALG}(S_{t-1}(\pi))$ with $t \in [t_0, t_1]$. This, in turn, occurs iff $K^{(j-1)}$ is *not* in any of the positions-touched sets $X_{t_0}(\pi), \ldots, X_{t_1-1}(\pi)$. ∎

Next we show that, if two elements π_i and π_k are both $(\pi, j-1)$-robust in some time interval T (where $i, k \in T$), then $K_i^{(j-1)}(\pi)$ and $K_k^{(j-1)}(\pi)$ must be distinct.

Lemma 226. *Let π be an input vector, let $[t_0, t_1] \subseteq [1, m]$ be a time window, and let $j \geq 0$. Define Q to be the set of $\pi_i \in [t_0, t_1]$ that are (π, j)-robust in $[t_0, t_1]$. Then we have that $K_i^{(j)}(\pi) \neq K_k^{(j)}(\pi)$ for every distinct $\pi_i, \pi_k \in Q$.*

Proof. Suppose $\pi_i, \pi_k \in Q$, where $i < k$. If j is odd, then Lemma 224 says that element $K_i^{(j-1)}(\pi)$ cannot be in $Y_{i+1}(\pi), \ldots, Y_{t_1}(\pi)$. Since $K_i^{(j-1)}(\pi)$ resides in position $K_i^{(j)}(\pi)$ of $\text{ALG}(S_i(\pi))$, it follows that $K_i^{(j)}(\pi) \notin X_{i+1}(\pi), \ldots, X_{t_1}(\pi)$. Thus $K_i^{(j)}(\pi) \notin X_k(\pi)$, meaning that $K_i^{(j)}(\pi) \neq K_k^{(j)}(\pi)$, as desired. Similarly, if j is even, then Lemma 225 says that position $K_k^{(j-1)}(\pi)$ cannot be in $X_{t_0}(\pi), \ldots, X_{k-1}(\pi)$. It follows that the element $K_k^{(j)}(\pi)$ that resides in position $K_k^{(j-1)}(\pi)$ in state $\text{ALG}(S_{k-1}(\pi))$ cannot be in any of $Y_{t_0}(\pi), \ldots, Y_{k-1}(\pi)$. Thus $K_k^{(j)}(\pi) \notin Y_i(\pi)$, meaning that $K_k^{(j)}(\pi) \neq K_i^{(j)}(\pi)$. ∎

Next, we make an indistinguishability argument: Given that an element π_i is $(\pi, j-1)$-robust in an interval T_1, the probability of it being (π_i, j)-robust in some $T_2 \subseteq T_1$ is independent of i.

Lemma 227. *Let π be a random permutation of $[1, m]$, let $j > 0$, let $[t_0, t_1] \subseteq [1, m]$ be a time window, and let $[t_0', t_1'] \subseteq [t_0, t_1]$ be a time window contained in $[t_0, t_1]$. Define $E_\pi(i)$ to be the event that π_i is $(\pi, j-1)$-robust in $[t_0, t_1]$ but is not (π, j)-robust in $[t_0', t_1']$. Then,*

$$\Pr[E_\pi(1)] = \Pr[E_\pi(2)] = \cdots = \Pr[E_\pi(m)].$$

Proof. Observe that event $E_\pi(i)$ is equivalent to event $E_{\pi'}(1)$ for $\pi' = \text{slide}(\pi, i, 1)$. However, if π is a random permutation then π' is too. Thus $\Pr[E_{\pi'}(1)] = \Pr[E_\pi(1)]$. This implies that $\Pr[E_\pi(i)] = \Pr[E_\pi(i)]$, as desired. ∎

At this point, we have established the core properties of (π, j)-robustness that will allow us to complete the proof. Consider two time intervals $T_2 \subseteq T_1$, and say that ALG (j, T_1, T_2)***-squashes*** an element π_i, $i \in T_1$, if π_i is $(\pi, j-1)$-robust in T_1 but is not (π, j)-robust in T_2. Lemmas 224 and 225 tell us that (if T_1 and T_2 are defined appropriately), then the only way that ALG can (j, T_1, T_2)-squash a given element π_i is if ALG touches $K_i^{(j-1)}(\pi)$ during time window T_2. Lemma 226 then tells us that, for every π_i that ALG (j, T_1, T_2)-squashes, the value of $K_i^{(j-1)}(\pi)$ will be *distinct*. This means that the only way for ALG to (j, T_1, T_2)-squash a large number of elements is if ALG touches a large number of elements/slots during T_2 (which would force ALG to incur a large cost during T_2). Thus we can assume that ALG (j, T_1, T_2)-squashes only a small fraction of the elements $\pi_i \in \{\pi_i \mid i \in T_1\}$. Finally, Lemma 227 tell us that ALG has no systematic control over *which* elements $\pi_i \in \{\pi_i \mid i \in T_1\}$ progress from being $(\pi, j-1)$-robust in T_1 to being (π, j)-robust in T_2. From this, we can conclude that, for a given element π_i, $i \in T_2$, the probability of π_i being (π, j)-robust in T_2 is only slightly smaller than that of π_i being $(\pi, j-1)$-robust in T_1. The full argument requires some care (as well as some special handling of odd vs even j) and is presented in the following two lemmas.

Lemma 228. *Let s be the expected cost per insertion/deletion of* ALG. *Let π be a random permutation of $[1, m]$, let $j > 0$ be odd, let $[t_0, t_1] \subseteq [1, m]$ be a time window, and let $[t_0', t_1]$ be a smaller time window also ending at t_1. Define $E(i)$ to be the event that π_i is $(\pi, j-1)$-robust in $[t_0, t_1]$ but is not (π, j)-robust in $[t_0', t_1]$. Then, for all $i \in [m]$,*

$$\Pr[E(i)] \le \frac{2s(t_1 - t_0') + 1}{t_0' - t_0 + 1}.$$

Proof. Let Q be the set of $i \in [t_0, t_0']$ for which $E(i)$ occurs. By Lemma 227, $p = \Pr[E(i)]$ is the same for all i. It follows that

$$\mathbb{E}[|Q|] = \mathbb{E}\left[\sum_{i \in [t_0, t_0']} \mathbb{I}(E(i))\right] = (t_0' - t_0 + 1)p. \tag{20.2}$$

For $i \in [t_0, t_0']$, if $E(i)$ occurs, then we must have that (1) π_i is $(\pi, j-1)$-robust in $[t_0, t_1]$ and (2) π_i is not (π, j)-robust in $[t_0', t_1]$. By the $(\pi, j-1)$-robustness of π_i in $[t_0, t_1]$, we know that the element $K_t^{(j-1)}(\text{slide}(\pi, i, t))$ is the same for all $t \in [t_0, t_1]$—call this element k_i. In order for π_i to be *non-*(π, j)-robust in $[t_0', t_1]$, it must also be non-$(\text{slide}(\pi, i, t_0'), j)$-robust in $[t_0', t_1]$. But Lemma 224 tells us that, in order for this to occur, we need either (1) that $|K_{t_0'}(\text{slide}(\pi, i, t_0'))| = j - 1$, in which case $k_i = K_{t_0'}^{(j-1)}(\text{slide}(\pi, i, t_0'))$ must be the element $m + t_0'$ that is deleted by the t_0'-th deletion; or (2) that $k_i = K_{t_0'}^{(j-1)}(\text{slide}(\pi, i, t_0'))$ is contained in at least one of $Y_{t_0'+1}(\text{slide}(\pi, i, t_0')), Y_{t_0'+2}(\text{slide}(\pi, i, t_0')), \ldots, Y_{t_1}(\text{slide}(\pi, i, t_0'))$. Combined, conditions (1) and (2) can be rewritten as $k_i \in \{m + t_0'\} \cup \bigcup_{t \in [t_0'+1, t_1]} Y_t(\text{slide}(\pi, i, t_0'))$. By (20.1), this, in turn, is equivalent to $k_i \in \{m + t_0'\} + \bigcup_{t \in [t_0'+1, t_1]} Y_t(\pi)$. Recalling that $k_i = K_i^{(j-1)}(\pi)$, we can conclude that: in order for $E(i)$ to occur, we must have both that π_i is (π, j)-robust in $[t_0, t_1]$ and that $K_i^{(j-1)}(\pi) \in \{m + t_0'\} \cup \bigcup_{t \in [t_0'+1, t_1]} Y_t(\pi)$.

Recall that Q is the set of $i \in [t_0, t_0']$ for which $E(i)$ occurs. Since every $\pi_i \in Q$ is $(\pi, j-1)$-robust in $[t_0, t_1]$, we have by Lemma 226 that each $\pi_i \in Q$ has a distinct $k_i = K_i^{(j-1)}(\pi)$. Thus each $\pi_i \in Q$ contributes a distinct element to $\{m + t_0'\} \cup \bigcup_{t \in [t_0'+1, t_1]} Y_t(\pi)$. This means that

$$|Q| \le 1 + \left|\bigcup_{t \in [t_0'+1, t_1]} Y_t(\pi)\right|.$$

The right side can be bounded above by one plus the total cost that the algorithm incurs in time interval $[t_0'+1, t_1]$, which in expectation is at most $2s(t_1 - t_0') + 1$. Thus

$$\mathbb{E}[|Q|] \le 2s(t_1 - t_0') + 1.$$

Combining this with (20.2), we have that

$$(t_0' - t_0 + 1)p \le 2s(t_1 - t_0') + 1,$$

which completes the proof of the lemma. ∎

Lemma 229. *Let s be the expected cost per insertion/deletion of* ALG. *Let π be a random permutation of $[1, m]$, let $j > 0$ be even, let $[t_0, t_1] \subseteq [1, m]$ be a time window, and let $[t_0, t_1']$ be a smaller time window also starting at t_0. Define $E(i)$ to be the event that π_i is $(\pi, j-1)$-robust in $[t_0, t_1]$ but is not (π, j)-robust in $[t_0, t_1']$. Then, for all $i \in [m]$,*

$$\Pr[E(i)] \le \frac{2s(t_1' - t_0) + \varepsilon n}{t_1 - t_0' + 1}.$$

Proof. Let Q be the set of $i \in [t_1', t_1]$ for which $E(i)$ occurs. By Lemma 227, $p = \Pr[E(i)]$ is the same for all i. It follows that

$$\mathbb{E}[|Q|] = \mathbb{E}\left[\sum_{i \in [t_1', t_1]} \mathbb{I}(E(i))\right] = (t_1' - t_1 + 1)p. \tag{20.3}$$

For $i \in [t_1', t_1]$, if $E(i)$ occurs, then we must have that (1) π_i is $(\pi, j-1)$-robust in $[t_0, t_1]$ and (2) π_i is not (π, j)-robust in $[t_0, t_1']$. By the $(\pi, j-1)$-robustness of π_i in $[t_0, t_1]$, we know that the position $K_t^{(j-1)}(\text{slide}(\pi, i, t))$ is the same for all $t \in [t_0, t_1]$—call this position k_i. In order for π_i to be *non-*(π, j)-robust in $[t_0, t_1']$, it must also be non-$(\text{slide}(\pi, i, t_1'), j)$-robust in $[t_0, t_1']$. But Lemma 224 tells us that, in order for this to occur, we need either (1) that $|K_{t_1'}(\text{slide}(\pi, i, t_1'))| = j - 1$, in which case $k_i = K_{t_1'}^{(j-1)}(\text{slide}(\pi, i, t_1'))$ must be one of the εn positions that are empty in state $\text{ALG}(S_{t_1'-1}(\text{slide}(\pi, i, t_1'))) = \text{ALG}(S_{t_1'-1}(\pi))$ (where the equality follows from (20.1)); or (2) that $k_i = K_{t_1'}^{(j-1)}(\text{slide}(\pi, i, t_1'))$ is contained in at least one of $X_{t_0}(\text{slide}(\pi, i, t_1')), X_{t_0+1}(\text{slide}(\pi, i, t_1')), \ldots, X_{t_1'-1}(\text{slide}(\pi, i, t_1'))$. Defining R to be the εn positions that are empty in state $\text{ALG}(S_{t_1'-1}(\pi))$, conditions (1) and (2) can be rewritten as $k_i \in R \cup \bigcup_{t \in [t_0, t_1'-1]} X_t(\text{slide}(\pi, i, t_1'))$. By (20.1), this, in turn, is equivalent to $k_i \in R + \bigcup_{t \in [t_0, t_1'-1]} X_t(\pi)$. Recalling that $k_i = K_i^{(j-1)}(\pi)$, we can conclude that: in order for $E(i)$ to occur, we must have both that π_i is (π, j)-robust in $[t_0, t_1]$ and that $K_i^{(j-1)}(\pi) \in R \cup \bigcup_{t \in [t_0, t_1'-1]} X_t(\pi)$.

Recall that Q is the set of $i \in [t_1', t_1]$ for which $E(i)$ occurs. Since every $\pi_i \in Q$ is $(\pi, j-1)$-robust in $[t_0, t_1]$, we have by Lemma 226 that each $\pi_i \in Q$ has a distinct $k_i = K_i^{(j-1)}(\pi)$. Thus each $\pi_i \in Q$ contributes a distinct element to $R \cup \bigcup_{t \in [t_0, t_1'-1]} X_t(\pi)$. This means that

$$|Q| \le |R| + \left| \bigcup_{t \in [t_0, t_1'-1]} X_t(\pi) \right| = \varepsilon n + \left| \bigcup_{t \in [t_0, t_1'-1]} X_t(\pi) \right|.$$

The right side can be bounded above by εn plus the total cost that the algorithm incurs in the time interval $[t_0, t'_1 - 1]$, which in expectation is at most $2s(t'_1 - t_0) + \varepsilon n$. Thus

$$\mathbb{E}[|Q|] \leq 2s(t'_1 - t_0) + \varepsilon n.$$

Combining this with (20.3), we have that

$$(t_1 - t'_1 + 1)p \leq 2s(t'_1 - t_0) + \varepsilon n,$$

which completes the proof of the lemma. ∎

The previous two lemmas considered odd and even j, respectively. We now combine them to obtain a single lemma that holds for all j.

Lemma 230. *Let c be a sufficiently large positive constant. Let s be the expected cost per insertion/deletion of* ALG *and let $\overline{s} > s$. Let π be a random permutation of $[1, m]$, and let $j > 0$. Consider a time window $[t_0, t_1] \subseteq [1, m]$ of size $r = t_1 - t_0 + 1$ satisfying $r > c\overline{s}\varepsilon n$. Furthermore, let $[t'_0, t'_1] = [t_0 + r/2 - r/(c\overline{s}^2), t_0 + r/2 + r/(c\overline{s}^2)]$ be the interval representing the middle $2/(c\overline{s}^2)$-fraction of $[t_0, t_1]$. Let E be the event that π_1 is $(\pi, j-1)$-robust in $[t_0, t_1]$ but π_1 is not (π, j)-robust in $[t'_0, t'_1]$. Then $\Pr[E] \leq 1/\overline{s}$.*

Proof. Suppose j is odd. We have that

$$\Pr[E] \leq \Pr[\pi_1 \text{ is } (\pi, j-1)\text{-robust in } [t_0, t'_1] \text{ but } \pi_1 \text{ is not } (\pi, j)\text{-robust in } [t'_0, t'_1]],$$

which by Lemma 228 is at most

$$\frac{2s(t'_1 - t'_0) + 1}{t'_0 - t_0 + 1}.$$

By assumption, $t'_0 - t_0 = \Omega(c\overline{s}^2(t'_1 - t'_0))$. Thus

$$\Pr[E] \leq O\left(\frac{s}{c\overline{s}^2}\right).$$

Setting c to be a sufficiently large positive constant, and using the fact that $\overline{s} \geq s$, we get $\Pr[E] \leq 1/\overline{s}$.

Now suppose j is even. We have that

$$\Pr[E] = \Pr[\pi_1 \text{ is } (\pi, j-1)\text{-robust in } [t'_0, t_1] \text{ but } \pi_1 \text{ is not } (\pi, j)\text{-robust in } [t'_0, t'_1]],$$

which by Lemma 229 is at most

$$\frac{2s(t'_1 - t'_0) + \varepsilon n}{t_1 - t'_1 + 1}.$$

By assumption, $t_1 - t'_1 = \Omega\left(c\overline{s}^2(t'_1 - t'_0)\right)$ and $t_1 - t'_1 = \Omega\left(c\overline{s}\varepsilon n\right)$, which means that

$$\Pr[E] = O\left(\frac{2s(t'_1 - t'_0) + \varepsilon n}{c\overline{s}^2(t'_1 - t'_0) + c\overline{s}\varepsilon n}\right).$$

Setting c to be a sufficiently large positive constant, and using the fact that $\overline{s} \geq s$, we get $\Pr[E] \leq 1/\overline{s}$. ∎

By applying Lemma 230 repeatedly, we can conclude that, with reasonably large probability, $\pi_{m/2}$ will be (π, j)-robust (for a reasonably large j) in some time interval containing $m/2$. This allows us to indirectly obtain a lower bound on $|K_{m/2}(\pi)|$.

Lemma 231. *Let π be a random permutation of $[1, m]$, let $j > 0$, and let c be a sufficiently large positive constant. Suppose* ALG *has cost $s \leq \overline{s}$ per insertion/deletion, and suppose that*

$$\left(c\overline{s}^2\right)^{j+1} \leq \varepsilon^{-1}. \tag{20.4}$$

Then

$$\Pr[|K_{m/2}(\pi)| \geq j] \geq 1 - j/\overline{s}.$$

Proof. Define $[a_0, b_0] = [1, m]$, and then recursively define

$$[a_\ell, b_\ell] = \left[a_{\ell-1} + (b_{\ell-1} - a_{\ell-1}) \cdot \left(1/2 - \frac{1}{c\overline{s}^2}\right), a_{\ell-1} + (b_{\ell-1} - a_{\ell-1}) \cdot \left(1/2 + \frac{1}{c\overline{s}^2}\right)\right]$$

for $\ell \in [1, j]$. From (20.4), we can deduce that for every $\ell \in [0, j]$,

$$|b_\ell - a_\ell| > c\overline{s}\varepsilon n,$$

which will allow us to apply Lemma 230 in a moment.

For $\ell \in [0, j]$, define

$$p_\ell = \Pr[\pi_1 \text{ is } (\pi, \ell)\text{-robust in } [a_\ell, b_\ell]].$$

Then $p_0 = 1$, and for all $\ell \in [1, j]$ we have by Lemma 230 that $p_\ell \geq p_{\ell-1} - 1/\overline{s}$. Thus $p_j \geq 1 - j/\overline{s}$. On the other hand, by Lemma 227, we know that

$$\Pr[\pi_{m/2} \text{ is } (\pi, j)\text{-robust in } [a_j, b_j]] = \Pr[\pi_1 \text{ is } (\pi, \ell)\text{-robust in } [a_\ell, b_\ell]] = p_j \geq 1 - j/\overline{s}.$$

But $\pi_{m/2}$ being (π, j)-robust implies that $|K_{m/2}(\pi)| \geq j$. Thus $\Pr[|K_{m/2}(\pi)| \geq j] \geq 1 - j/\overline{s}$, as desired. ∎

We can now prove our lower bound on the expected cost incurred by ALG.

Theorem 223. *Let $\varepsilon \geq 1/n$. Any fixed-size stateless allocation algorithm* ALG *must incur $\Omega(\log \varepsilon^{-1} / \log\log \varepsilon^{-1})$ expected cost per insertion/deletion.*

Proof. Let s be the expected cost per insertion/deletion of ALG and suppose for contradiction that $s = o(\log \varepsilon^{-1} / \log\log \varepsilon^{-1})$. Let $\overline{s} = \log \varepsilon^{-1}$, let c be a sufficiently large positive constant and let $j = \log \varepsilon^{-1}/(c \log\log \varepsilon^{-1})$. By design, we have

$$\left(c\overline{s}^2\right)^{j+1} \le \varepsilon^{-1}.$$

Consider a random permutation π of $[1,m]$. By Lemma 230, we have that

$$\Pr[|K_{m/2}(\pi)| \ge j] \ge 1 - j/\overline{s} \ge 1/2.$$

But that means that there is a deletion/insertion pair with an expected cost of at least $j/2$. Thus $s \ge j/2 = \Omega(\log \varepsilon^{-1}/\log\log \varepsilon^{-1})$, a contradiction. ∎

Chapter 21

Efficient Data-Structural Implementations

Until now, we have concerned ourselves only with the overhead of an allocator. An upper bound of L on the expected overhead means that, for an insertion/deletion of an item x with size $\pi(x)$, the cumulative sizes of the items that are rearranged will be $O(L\pi(x))$ in expectation. Notice, however, that actually determining *which* items to move may be difficult do efficiently. Indeed, although our allocators in Chapter 19 achieve $O(1 + \log \varepsilon^{-1})$ expected overhead, a naive implementation of these allocators even in the *fixed-size case* would take time $\Omega(L\pi(x) + \varepsilon^{-1})$ per insertion/deletion.

In this chapter, we consider the data-structural version of the stateless allocation problem, in which our goal is to achieve expected *time* $O(L\pi(x))$ for the insertion/deletion of a given key x. Additionally, the data structures that we present will support constant-time queries.

Formally, we wish to construct a strongly history-independent hash table that maps each of its keys $x \in S$ to a contiguous interval $\phi(x)$ of size $\pi(x)$, that supports constant-time queries (determining $\phi(x)$ for a given key x), and that supports efficient insertions/deletions. The ***overhead*** of such a hash table is said to be bounded by a quantity L if the insertion/deletion of a key x of size $\pi(x)$ is guaranteed to take expected time $O(L\pi(x))$. The hash table is said to achieve ***load factor*** $1 - O(\varepsilon)$ if it uses space at most $(1 + O(\varepsilon)) \sum_{x \in S} \pi(x)$. As a convention, we will assume that $\pi(x) \geq 1$ includes the space needed to store x—that, is the first machine word in the interval $\phi(x)$ is used to store the key x explicitly. We are interested both in the ***fixed-size case***, where $\pi(x) = \pi(y) = \Theta(1)$ for every $x, y \in S$, and the ***variable-size case***, where different elements can have different sizes.

We will typically use m to either denote $\sum_{x \in S} \pi(x)$ or denote an upper bound on $\sum_{x \in S} \pi(x)$.

In the fixed-size case, we show that for any $\varepsilon^{-1} \leq \log^{1/10} m$, one can construct a strongly history independent hash table with load factor $1 - \varepsilon$ and overhead $O(1 + \log \varepsilon^{-1})$. By the lower bound given in Chapter 20, the overhead achieved by this construction is optimal up to a factor of $O(\log \log \varepsilon^{-1})$.

Next, we turn our attention to the variable-size case. We show that, for any $\varepsilon^{-1} \le \log^{1/10} m$, if we assume that each item $x \in S$ has size $\pi(x) \le \varepsilon^4 m$, then we can construct a strongly history independent hash table with load factor $1 - \Theta(\varepsilon)$ and overhead $O(1 + \log \varepsilon^{-1})$. This gives a time-efficient realization of the bounds from Chapter 19.

Throughout the chapter, except for when otherwise specified, we measure space in machine words.

Chapter overview. We begin in Section 21.1 by constructing efficient strongly history-independent hash tables both for fixed-size objects and for relatively small objects (i.e., $\pi(x) \le \log^{1/10} m$). Here we are able to make use of several techniques that have been popularized in the succinct data-structure literature in recent years, namely the use of frontyard/backyard constructions [52, 75, 76, 82, 94, 146] and the use of the Method of Four Russians [54, 75, 76, 82, 85, 105]. Then, in Section 21.2, we give a solution that is efficient for large objects but not for small ones. Finally, in Section 21.3, we show how to combine these results to get a single solution that, for an object x of size $\pi(x) \le \varepsilon^7 x$, supports insertions/deletions in expected time $O((1 + \log \varepsilon^{-1})\pi(x))$, and queries in constant time (w.h.p.).

We remark that, throughout the chapter, we will assume access to fully random hash functions. This assumption is without loss of generality in our setting, since one can make use of (what are at this point a standard set of) known techniques [158, 162, 291, 326] for efficiently simulating fully-random hash functions in data-structural applications—for a detailed discussion of how to apply these techniques, see, e.g., [85, 158].

21.1 An Efficient Allocator for Small Objects

Let m be an upper bound on $\sum_{x \in S} \pi(x)$. In this section, we consider the special case where objects are guaranteed to have sizes at most $\log^{1/10} m$. In fact, we give a general-purpose construction for how to take an *arbitrary* stateless allocator $\mathcal{A}$ and make it time efficient in this setting.

Let ε, m, R, U be parameters satisfying $\varepsilon^{-1} \le \log^{1/10} m$ and $R \le \log^{2/10} m$. We will consider sets $S \subseteq [U]$ of keys, and size functions $\pi : S \to \mathbb{N}$ satisfying $\pi(x) \in [1, R]$ for every $x \in X$ and satisfying $\sum_{x \in S} \pi(x) \le m$. Finally, let $\mathcal{A}$ be a stateless allocation algorithm that, given as input a set of items $T \subseteq [\log^4 m]$ and a size function $\pi : T \to [1, R]$ satisfying $\sum_{x \in T} \pi(x) \le (1 + \varepsilon) \log^{9/10} m$, produces an allocation mapping the items in $x \in T$ to disjoint intervals $\phi(x) \subseteq [0, (1 + 2\varepsilon) \log^{9/10} m]$ of size $\pi(x)$.

We will show how to construct a strongly history independent hash table that stores items with sizes in $[1, R]$, and that incurs overhead matching that is achieved by the black-box algorithm $\mathcal{A}$. Plugging in the allocators from Chapter 19, we get an $O(1 + \log \varepsilon^{-1})$-overhead hash table for fixed-size objects (Theorem 235) and an $O(1 + \log \varepsilon^{-1})$-overhead hash table for small variable-size objects (Theorem 236).

We remark that, for convenience, we assume that the capacity parameter m is fixed. However, at the end of the section, we will describe how to extend the data structure to support dynamic resizing.

A frontyard/backyard construction. Partition the first $(1+2\varepsilon)m$ slots of the array into $m/\log^{9/10} m$ bins $B_1, B_2, \ldots, B_{m/\log^{9/10} m}$ of size $b = (1+2\varepsilon)\log^{9/10} m$ each. Let $h : [U] \to [m/\log^{9/10} m]$ be a random hash function, and say that bin B_i ***owns*** keys

$$O_i = \{x \in S \mid h(x) = i\}.$$

Let $g : [U] \to [12 \log\log m]$ be a random pairwise independent hash function. Say that bin B_i contains a ***fingerprint collision*** if $g(x) = g(y)$ for some pair of distinct $x, y \in O_i$.

Say that a bin B_i is ***overflowed*** if either $\sum_{x \in O_i} \pi(x) > (1+\varepsilon)\log^{9/10} m$ or if B_i contains a fingerprint collision. At a high level, keys $x \in S$ that map to overflowed bins $B_{h(x)}$ will be stored in a ***backyard*** data structure, to be described in a moment. Keys $x \in S$ that map to non-overflowed bins $B_{h(x)}$ will be allocated space within their bin; these keys are said to reside in the ***frontyard***.

Implementing the frontyard. Consider the keys O_i in some *non-overflowed* bin B_i. Recall that $\sum_{x \in O_i} \pi(x) \le (1+\varepsilon)\log^{9/10} m$ and that bin B_i is a sub-array of size $(1+2\varepsilon)\log^{9/10} m$. We can therefore use $\mathcal{A}$ to construct an allocation ϕ_i mapping the fingerprint $g(x)$ of each key $x \in O_i$ to a disjoint interval $\phi_i(g(x))$ in B_i.

There are two difficulties that we encounter here: (1) how do we efficiently *implement* $\mathcal{A}$, so that if the algorithm incurs overhead ℓ on an allocation of size k, we take time $O(\ell k)$; and (2) how do we support queries allowing us to evaluate ϕ_i in constant time?

We solve both of these difficulties with a lookup-table approach (a.k.a., the so-called Method of Four Russians). Observe that, for each $x \in O_i$, both the fingerprint $g(x)$ and the size $\pi(x) \in [1, R]$ can be written using $O(\log\log n)$ bits. It follows that the entire input to ϕ_i can be written using $O(|O_i| \log\log n) = O(\log^{9/10} n \log\log n)$ bits. As there are $n^{o(1)}$ such inputs, we can construct a global lookup table (shared across all of the bins) that tells us how to implement insertions, deletions, and queries. Given an input I and a new key y with fingerprint $g(y)$ to be inserted, the lookup table $L(\textit{insert } g(y) \textit{ into } I)$ tells us what rearrangements need to be performed to get the new allocation (post-insertion), and what the new input I' is after the insertion (and whether the insertion causes a fingerprint collision). Similarly, given an input I and a key y with fingerprint $g(y)$ to be deleted, the lookup table $L(\textit{delete } g(y) \textit{ from } I)$ tells us what rearrangements need to be performed to get the new allocation (post-deletion), and what the new input I' is after the deletion. And, given an input I and a key y with fingerprint $g(y)$, the lookup table $L(\textit{query } g(y) \textit{ in } I)$ evaluates $\phi_i(g(y))$. (Note that negative queries can be detected using the fact that each key $x \in O_i$ is stored at the beginning of the interval $\phi_i(g(x))$ allocated to it.)

With this lookup table approach, we have the following guarantees: If the expected overhead of $\mathcal{A}$ is L, then the expected time to insert/delete a key x of size $\pi(x)$ in the frontyard is $O(L\pi(x))$ (as the time is dominated by the cost of rearrangements); and the time to query a key x is $O(1)$ (deterministically).

Implementing the backyard. Before describing how to implement the backyard, we begin with two lemmas bounding the size of the backyard.

Lemma 232. *The probability of a given item $x \in S$ being in the backyard is at most $o(1/\log^{10} m)$.*

Proof. For $x \in S$ to be in the backyard, we would need either that

$$\sum_{y \in O_{h(x)}} \pi(y) \geq (1+\varepsilon)\log^{9/10} m. \tag{21.1}$$

or that

$$\exists y, z \in S \text{ s.t. } h(x) = h(y) = h(z) \text{ and } g(y) = g(z) \text{ and } y \neq z. \tag{21.2}$$

To bound the probability of (21.1), observe that $Z := \sum_{y \in O_{h(x)} \setminus \{x\}} \pi(y)/R$ has expected value at most $(\log^{9/10} m)/R$, and is a sum of independent random variables that are each at most 1. As $\varepsilon \geq 1/\log^{1/10} m$ and $R \leq \log^{2/10} m$, we have that

$$\begin{aligned}
&\Pr\left[\sum_{y \in O_{h(x)} \setminus \{x\}} \pi(y) \geq (1+\varepsilon)\log^{9/10} m - R\right] \\
&\leq \Pr[Z \geq (1+\varepsilon)(\log^{9/10} m)/R - 1] \\
&\leq \Pr[Z \geq \mathbb{E}[Z] + \sqrt{\mathbb{E}[Z]} \cdot \log^{\Omega(1)} m],
\end{aligned}$$

which by a Chernoff bound is $o\left(\frac{1}{\log^{10} m}\right)$.

To bound the probability of (21.2), observe that there are at most $\binom{m}{2} \leq m^2$ options for y and z, each of which has probability at most $\left(\frac{\log^{9/10} m}{m}\right)^2$ of satisfying $h(y) = y(z) = h(x)$, and probability at most $1/\log^{12} m$ of satisfying $g(y) = g(z)$. Thus the probability of (21.2) is at most

$$m^2 \cdot \left(\frac{\log^{9/10} m}{m}\right)^2 \cdot \frac{1}{\log^{12} m} = o\left(\frac{1}{\log^{10} m}\right).$$

∎

Lemma 233. *Let X be the set of keys $x \in S$ that map to overflowed bins $B_{h(x)}$.*

With probability at least $1 - 1/\operatorname{poly}(m)$,

$$|X| \le m/\log^{10} m.$$

Proof. By Lemma 232, $\mathbb{E}[|X|] \le o(m/\log^{10} m)$. For each $x \in S$, we define $P_x = (g(x), h(x))$. Then X is a determined by $\le m$ independent random variables $\{P_x \mid x \in S\}$, and changing the value of any given P_x can change X by at most $b = O(\log^{9/10} m)$. Thus, by McDiarmid's inequality, we have for all $t \ge 0$ that $\Pr[X \ge \mathbb{E}[X] + bt\sqrt{m}] \le e^{t^2/2}$. It follows that $|X| \le m/\log^{10} m$ with probability $1 - 1/\operatorname{poly}(m)$. ∎

Since the backyard is so small, we can implement the backyard as follows: partition the keys into $O(\frac{1}{\varepsilon}\log n)$ groups $G_1, G_2, \ldots$, where each key x in the backyard goes to group $\lceil \log_{(1+\varepsilon)} \pi(x) \rceil$. Implement each group G_i as a strongly history independent hash table capable of storing up to $2m/\log^{10} m$ elements of size $(1+\varepsilon)^i$ each, using Blelloch and Golovin's construction [105]. Critically, each G_i contains at most $m/\log^{10} m$ elements, so the hash table is at a load factor of at most $1/2$, and therefore incurs $O(1)$ expected overhead (and worst-case constant-time queries). Finally, implement one more hash table J, also with capacity $2m/\log^{10} m$, that maps each key x to its size $\pi(x)$ (so that queries can determine which hash table G_i to find key x's allocation in).

Assuming that the backyard contains at most $m \log^{10} m$ elements, the space used to implement these hash tables will be at most $O(\varepsilon^{-1} \log m \cdot m/\log^{10} m) \le O(m/\log^8 m)$.

Finally, for each bin that is in the backyard, we maintain a linked list traversing the bin's elements (in the backyard) in sorted order (by key). This linked list ensures that, if we ever need to bring the bin back to the frontyard (i.e., it is no longer overflowed), we can recover its elements in time proportional to the size of the bin. The linked lists consume $O(1)$ extra space per backyard element[1]. Thus the space to implement the backyard remains $O(m/\log^8 m)$ with high probability in m.

In general, if an insertion/deletion of some key x touches an overflowed bin $B_{h(x)}$ (or if the insertion/deletion changes the overflow-state of the bin $B_{h(x)}$), then the insertion/deletion will take at most $O(\sum_{y \in B_{h(x)}} \pi(y))$ expected time. We can deduce by a Chernoff bound that, even if we condition on $B_{h(x)}$ being overflowed, $\sum_{y \in B_{h(x)}} \pi(y)$ has expected value $O(b) = O(\log^{9/10} m)$ (and is, in fact, bounded above by a geometric random variable with mean $O(b)$). On the other hand, we have by Lemma 232 that each insertion/deletion has less than a $1/\log^{10} m$ probability of touching an overflowed bin—thus the expected time per insertion/deletion spent handling overflowed bins is $O(1)$.

Putting the pieces together. We can now prove the following proposition.

[1]The base pointer for the linked list can be stored directly in the bin B_i.

Proposition 234. *Let ε, m, R, U be parameters satisfying $\varepsilon^{-1} \le \log^{1/10} m$ and $R \le \log^{1/10} m$. Consider input sets $S \subseteq [U]$ of keys, and size functions $\pi : S \to \mathbb{N}$ satisfying $\pi(x) \in [1, R]$ for every $x \in X$ and satisfying $\sum_{x \in S} \pi(x) \le m$. Let $\mathcal{A}$ be a stateless allocation algorithm that, given as input a set of items $T \subseteq [\log^4 m]$ and a size function $\pi : T \to [1, R]$ satisfying $\sum_{x \in T} \pi(x) \le (1+\varepsilon) \log^{9/10} m$, produces an allocation mapping the items in $x \in T$ to disjoint intervals $\phi(x) \subseteq [0, (1+2\varepsilon) \log^{9/10} m]$ of size $\pi(x)$. Finally, let L be the overhead achieved by $\mathcal{A}$.*

Then we can construct a strongly history-independent hash table that maps the elements $x \in S$ to disjoint intervals $\phi(x)$ of size $\pi(x)$ in $[0, (1+O(\varepsilon))m]$, that incurs $O(L)$ expected overhead per insertion/deletion, and that supports $O(1)$-time queries (with probability $1 - 1/\operatorname{poly}(m)$).

Proof. The frontyard bins consume $(1 + O(\varepsilon))m$ total space, and the backyard consumes $O(m/\log^8 m) = O(\varepsilon m)$ space. Thus the total space consumption is $(1 + O(\varepsilon))m$.

To analyze overhead, observe that insertions/deletions in non-overflowed bins incur $O(L)$ expected overhead, since they simulate $\mathcal{A}$ time efficiently using lookup tables. On the other hand, the expected amount of time spent by a given insertion/deletion on overflowed bins is $O(1)$. Finally queries in both the front and backyards take time $O(1)$, completing the proof. ∎

Two consequences. For the fixed-size case, if we implement $\mathcal{A}$ as an allocator with overhead $O(1 + \log \varepsilon^{-1})$, then we arrive at the following theorem.

Theorem 235. *Let ε, m, R, U be parameters satisfying $\varepsilon^{-1} \le \log^{1/10} m$ and $R = O(1)$. Consider input sets $S \subseteq [U]$ and size functions $\pi : S \to \mathbb{N}$ satisfying $\pi(x) = R$ for all $x \in S$ and satisfying $\sum_{x \in S} \pi(x) \le m$.*

Then we can construct a strongly history-independent hash table that maps the elements $x \in S$ to disjoint intervals $\phi(x)$ of size R in $[0, (1+O(\varepsilon))m]$, that incurs $O(1 + \log \varepsilon^{-1})$ expected overhead per insertion/deletion, and that supports $O(1)$-time queries (with probability $1 - 1/\operatorname{poly}(m)$).

We remark that the choice of $1/10$ in the constraint $\varepsilon^{-1} \le \log^{1/10} m$ is somewhat arbitrary. One can easily replace this with a larger number, and with a more involved construction, one could likely replace the constraint with $\varepsilon^{-1} \le \log m / \log \log m$—we leave this as a possible direction for future work

For the variable-size case, if we implement $\mathcal{A}$ using Theorem 193, then we arrive at the following theorem.

Theorem 236. *Let ε, m, R, U be parameters satisfying $\varepsilon^{-1} \le \log^{1/10} m$ and $R \le \log^{2/10} m$. Consider input sets $S \subseteq [U]$ and size functions $\pi : S \to \mathbb{N}$ satisfying $\pi(x) \in [1, R]$ for all $x \in S$ and satisfying $\sum_{x \in S} \pi(x) \le m$.*

Then we can construct a strongly history-independent hash table that maps the elements $x \in S$ to disjoint intervals $\phi(x)$ of size $\pi(x)$ in $[0, (1+O(\varepsilon))m]$, that incurs $O(1 + \varepsilon^{-1})$ expected overhead per insertion/deletion, and that supports $O(1)$-time *queries (with probability* $1 - 1/\operatorname{poly}(m)$*).*

Note that Theorem 236 can only handle relatively small keys—we will extend it to handle larger keys later on in the section.

Using waterfall addressing [75] to supporting dynamic resizing. For convenience, we have focused in this section on a fixed-capacity hash table, that is, we have assumed that $\sum_{x \in S} \pi(x) \le m$ for some fixed parameter m. We now describe how to add dynamic resizing (i.e., m can change over time) while achieving the same bounds. Here, we assume that the data structure resides on an infinite tape, and that we wish to use only a prefix of the tape at any moment.

Recall the "standard" resizing trick for strongly history-independent data structures: For $i > 0$, define r_i to be a random number between $(1+\varepsilon)^{i-1}$ and $(1+\varepsilon)^i$; then, at any given moment, we use $(1+\varepsilon)^{\min\{i \mid r_i > \sum_{x \in S} \pi(x)\}}$ as our value for the parameter m. Whenever this quantity changes, a ***resizing rebuild*** is required.

One way to implement a resizing rebuild (which will not work here) is to simply rebuild the entire data structure in time $O(\sum_{x \in S} \pi(x))$. Since the probability of a given insertion/deletion triggering a resizing rebuild is $O(\varepsilon \sum_{x \in S} \pi(x)))$, this approach adds $O(\varepsilon^{-1})$ expected time per insertion/deletion.

This simplistic approach to implementing resizing rebuilds does not for our hash tables, since we wish to achieve an overhead of $O(1 + \log \varepsilon^{-1})$. Note that the backyard is not the problem here, since it has size much smaller than εm—the difficulty is implementing a resizing rebuild on the frontyard.

To implement resizing rebuilds more efficiently (without actually rebuilding the entire frontyard), we can make use of a more heavyweight tool that my collaborators and I developed in our work on succinct hashing: Waterfall addressing [75] (which can be viewed as a time-efficient variation on linear hashing [233], allows for one to dynamically increase/decrease the number of bins that a hash function h maps to. The key properties that make waterfall addressing useful here are that, if we increase the number of bins by an $\varepsilon \ge \Omega(\log \log n / \log n)$ fraction, then (1) each element has its hash change with probability only $O(\varepsilon)$; and (2) the hashes whose elements change can be identified in expected time $O(m \log \log n / \log n)$ using an auxiliary data structure that consumes space $O(\log \log n)$ bits per element (moreover, this data structure can straightforwardly be implemented history independently); and (3) the hash function can be evaluated in constant time. If we use waterfall addressing, then a rebuild that increases/decreases m by a $(1+\varepsilon)$ factor can identify the elements that need to move in time $O(m \log \log n / \log n) = O(\varepsilon m)$ time, and can then move them to their new bins in expected time $O(\varepsilon m \log \varepsilon^{-1})$ (since the sum of the sizes of the elements moved is $O(\varepsilon m)$ in expectation, and inserting/deleting in a bin has overhead $O(1 + \log \varepsilon^{-1})$). This preserves the $O(1 + \log \varepsilon^{-1})$ overhead bounds in our theorems while supporting dynamic resizing:

Corollary 237. *Let* ε, R, U *be parameters satisfying* $R = O(1)$*. Consider input sets* $S \subseteq [U]$ *and size functions* $\pi : S \to \mathbb{N}$ *satisfying* $\pi(x) = R$ *for all* $x \in S$*, and satisfying* $\varepsilon^{-1} \le \log^{1/10} \sum_{x \in S} \pi(x)$*.*

Then we can construct a strongly history-independent hash table that maps the elements $x \in S$ to disjoint intervals $\phi(x)$ of size R in $[0, (1 + O(\varepsilon)) \sum_{x \in S} \pi(x)]$, that incurs $O(1 + \log \varepsilon^{-1})$ expected overhead per insertion/deletion, and that supports $O(1)$-time queries (with probability $1 - 1/\operatorname{poly}(\sum_{x \in S} \pi(x))$).

Corollary 238. *Let ε, R, U be parameters. Consider input sets $S \subseteq [U]$ and size functions $\pi : S \to \mathbb{N}$ satisfying $\pi(x) \in [1, R]$ for all $x \in S$, satisfying $\varepsilon^{-1} \leq \log^{1/10} \sum_{x \in S} \pi(x)$ and $R \leq \log^{2/10} \sum_{x \in S} \pi(x)$.*

Then we can construct a strongly history-independent hash table that maps the elements $x \in S$ to disjoint intervals $\phi(x)$ of size $\pi(x)$ in $[0, (1 + O(\varepsilon)) \sum_{x \in S} \pi(x)]$, that incurs $O(1 + \varepsilon^{-1})$ expected overhead per insertion/deletion, and that supports $O(1)$-time queries (with probability $1 - 1/\operatorname{poly}(\sum_{x \in S} \pi(x))$).

21.2 An Efficient Allocator for Large Objects

Now we turn our attention to storing a set S of large objects x satisfying $\pi(x) \geq \varepsilon^{-2}$. This case is much easier since we can amortize the time/space costs to the fact that the items we are moving around are quite large.

Let us begin by considering Algorithm 4, which stores items with power-of-two sizes and incurs expected overhead $O(1 + \log \varepsilon^{-1})$. Since we are considering only items with sizes $\geq \varepsilon^{-2}$, we can think of space as being allocated at the granularity of ***meta-slots*** of size ε^{-2}. We will assume that $\varepsilon^{-1} \leq n^{1/4}$, meaning that there are at least $\sqrt{n}$ meta-slots.

Let $x_1, x_2, \ldots$ denote the elements of S in the order that they are inserted by Algorithm 4, and let n be the size of the array used by the algorithm (so we are guaranteed that $\sum_{x \in S} \pi(x) \leq (1 - \varepsilon)n + 1$).

Say that a key $x \in S$ ***wishes for*** a meta-slot i if, in Algorithm 4, x would have preferred being assigned to an interval beginning with meta-slot i, rather than the interval that the key is currently assigned to. For each meta-slot i, we will store a linked list Q_i consisting of the keys that wish for that meta-slot (sorted, say, lexicographically).

We can perform space allocation for the linked lists Q_i using Blelloch and Golovin's hash table [105] set to load factor 1/2. Additionally, we have a hash table of size $O(\varepsilon^2 n)$ for keeping track of where each item $x \in S$ is currently allocated. The next lemma tells us that, with high probability in n, the total space consumed by the Q_is will be $O(\varepsilon n)$.[2]

Lemma 239. *The total space consumed by the Q_is is $O(\varepsilon n)$ with probability $1 - 1/\operatorname{poly}(n)$.*

Proof. Define w_i to be the number of meta-slots that x_i wishes for. By design, even if we condition on the outcomes of $w_1, w_2, \ldots, w_{i-1}$, the random variable w_i is

[2]In the low-probability event that this bound fails, we can give up on insertion/deletion/query time and just use an arbitrary strongly history-independent allocator.

bounded above by a geometric random variable with mean $O(\varepsilon^{-1})$ (as each iteration of the inner for-loop for Algorithm 4 succeeds with probability at least ε). Thus $\mathbb{E}[\sum_j w_j] = O(\varepsilon^{-1}|S|) = O(\varepsilon n)$, where the final inequality uses the fact that each element of S has size at least ε^{-2}. By a Chernoff bound, we have that $\sum_j w_j \le O(\varepsilon n)$ with probability $1 - 1/\operatorname{poly}(n)$. ∎

We can also bound the expected size of each Q_i. Note that this bound does not follow directly from Lemma 239 because the Q_is are not symmetric (meta-slots i that are divisible by large powers of two will likely have a larger list Q_i than meta-slots i that are not).

Lemma 240. *Each Q_i has expected size $O(\varepsilon^{-1})$.*

Proof. Consider a meta-slot i that is occupied by an element x_j. Now consider the insertion of an element x_k, where $k \ge j$. When x_k is inserted by Algorithm 4, there are guaranteed to be at least $\varepsilon n/\pi(x_k)$ (aligned) intervals of size $\pi(x_k)$ where x_k could go. Define H to be the set of hashes corresponding to those intervals, and let w be the hash corresponding to the aligned interval of size $\pi(x_k)$ that begins with meta-slot i (if such an interval exists). Then x_k will wish for meta-slot i if and only if the sequence $h_1(x_k), h_2(x_k), \ldots$ contains w before it contains any of the elements of H. This happens with probability $1/(|H|+1) = O(\pi(x_k)/(\varepsilon n))$. It follows by linearity of expectation that $\mathbb{E}[Q_i] \le \sum_k O(\pi(x_k)/(\varepsilon n)) \le O(\varepsilon^{-1})$. ∎

For a given input (S, π), and a given $x \in S$, let $\psi(S, \pi, x)$ be the index j such that Algorithm 4 on input (S, π) assigns x using $h_j(x)$. If $x \notin S$, then we define $\psi(S, \pi, x) = 0$. Define $\overline{h}_j(x)$ to be the set of meta-slots that appear in the interval $[(h_j(x) - 1) \cdot \pi(x) + 1, h_j(x)\pi(x)]$.

Let (S, π) and (S', π) be inputs such that $S = S' \cup \{y\}$ for some element y—we will think of S as being reached from S' via the insertion of y. Let

$$P = \bigcup_{x \in S} \bigcup_{\psi(S', \pi, x) < j \le \psi(S, \pi, x)} \overline{h}_j(x).$$

One can think of P as consisting of all of the meta-slots i with the property that: S allocates some element x to an interval containing i, but S' *does not* allocate the same element x to that interval.

Our next lemma tells us that, even though the set P is determined by a randomized process that *interacts* with the Q_is (and is therefore *not* independent of them), the expected size of Q_is for $i \in P$ is still small.

Lemma 241. *We have that*

$$\mathbb{E}\left[\sum_{i \in P} |Q_i|\right] \le O(\pi(y) \log \varepsilon^{-1}).$$

Proof. Consider some x_j such that $\psi(S, \pi, x_j) \neq \psi(S', \pi, x_j)$. Let $a = \psi(S', \pi, x_j)$. On input (S, π), when Algorithm 4 gets around to processing x_j, the inner for loop will fail to allocate x_j using any of $h_1, \ldots, h_a$. The inner for loop will then attempt to use each of $h_{a+1}, h_{a+2}, \ldots$ until it finds a set $\overline{h}_{a+k}(x_j)$ of meta-slots that are all free. Critically, each of $h_{a+1}(x_j), h_{a+2}(x_j), \ldots$ are independent uniformly random elements of $[n/\pi(x_j)]$. For each $k > a \in \mathbb{N}$, let

$$q_k(x_j) = \begin{cases} |Q_{h_k(x)}| & \text{if } k \leq \psi(S, \pi, x_j) \\ 0 & \text{otherwise.} \end{cases}$$

By Lemma 240, $\mathbb{E}[Q_{h_k(x)}] = O(\varepsilon^{-1})$. Since $h_k(x)$'s value is independent of the event $k \leq \psi(S, \pi, x_j)$, it follows that $\mathbb{E}[q_k(x_j)] \leq O(\varepsilon^{-1}) \cdot \Pr[k \leq \psi(S, \pi, x_j)]$, which in turn is at most $O(\varepsilon^{-1}) \cdot (1 - \varepsilon)^{k-a-1}$. Summing of $k > a$,

$$\mathbb{E}\left[\sum_{k>a} q_k(x_j)\right] \leq O(\varepsilon^{-2}).$$

Thus we have

$$\begin{aligned} \mathbb{E}[P] &\leq \mathbb{E}\left[\sum_{j|\psi(S,\pi,x)\neq\psi(S',\pi,x)} \quad \sum_{k>\psi(S',\pi,x)} q_k(x_j)\right] \\ &\leq O\left(\mathbb{E}\left[\sum_{j|\psi(S,\pi,x)\neq\psi(S',\pi,x)} \varepsilon^{-2}\right]\right). \end{aligned}$$

Recall that $y = S' \setminus S$ is the element inserted between S and S'. As Algorithm 4 incurs expected overhead $O(1 + \log \varepsilon^{-1})$, the expected sum of the sizes of the elements whose intervals are reassigned because of the insertion is at most $O(\pi(y) \log \varepsilon^{-1})$. Since each element has size at least ε^{-2}, the expected *number* of elements that have their intervals reassigned is at most $O(\varepsilon^2 \pi(y) \log \varepsilon^{-1})$. It follows that

$$\mathbb{E}\left[\sum_{j|\psi(S,\pi,x)\neq\psi(S',\pi,x)} O(\varepsilon^{-2})\right] \leq O(\pi(y) \log \varepsilon^{-1}),$$

which completes the proof. ∎

We now describe how the Q_is can be used to efficiently implement Algorithm 4. The main role of the linked list Q_i is as follows: Suppose an item x is allocated to some interval I, and that x is then *de-allocated* from that interval (either because x is deleted, or because x is reassigned to a different interval). We need to determine which (if any) elements should be reassigned to reside in I. We can do this by considering each meta-slot $i \in I$, and examining the list Q_i.

We must also be careful to also account for the cost of updating the Q_is. If, during an insertion, an item x is moved from its j-th-choice interval to its j'-th-choice interval, then all of $Q_{h_j(x)}, Q_{h_{j+1}(x)}, \ldots, Q_{h_{j'}(x)}$ must be updated.

In total, the algorithm may spend up to $O(|Q_i|)$ time on each Q_i with $i \in P$. By Lemma 241, it follows that the time overhead incurred per insertion/deletion is $O(1 + \log \varepsilon^{-1})$.

Putting the pieces together, we can implement Algorithm 4 to store items of size $\pi(x) \geq \varepsilon^{-2}$ at a load factor of $1 - O(\varepsilon)$ while supporting an insertion/deletion of an item x in expected time $O(\pi(x) \log \varepsilon^{-1})$, and while supporting queries in constant time with probability $1 - 1/\operatorname{poly}(n)$.

Note that the same guarantee extends directly to the resizable version of Algorithm 4 in Theorem 201 (as the resize operation can easily be implemented in time $O(\varepsilon^{-1} n)$), and can therefore also easily be achieved for the algorithm given by 206 (as it uses Theorem 201 as its main algorithmic building block). Finally, combining these algorithms, we get a time-efficient implementation of Theorem 193:

Proposition 242. *Let U, ε^{-1} be positive integers and let $\rho > 0$ be an upper bound on object size. Consider inputs (S, π) where $S \subseteq [U]$, where $\varepsilon^{-4} \leq \sum_{x \in S} \pi(x)$, where $\pi(x) \in [\varepsilon^{-2}, \rho]$ for all $x \in S$. Then one can construct a strongly history-independent hash table that uses space $(1 + O(\varepsilon)) \sum_{x \in S} \pi(x) + O(\varepsilon^{-3} \rho)$, that incurs expected time overhead $O(1 + \log \varepsilon^{-1})$ per insertion and deletion, and that supports queries in constant time with probability $1 - 1/\operatorname{poly}(\sum_{x \in S} \pi(x))$.*

21.3 Putting the Pieces Together

Combining together Corollary 238 with Proposition 242, we can now construct a general-purpose solution to the variable-size case, under the assumption that each element $x \in S$ has size at most $O(\varepsilon^4 \sum_{x \in S} \pi(x))$.

Theorem 243. *Let ε, U be parameters. Consider input sets $S \subseteq [U]$ and size functions $\pi : S \to \mathbb{N}$. Let m denote $\sum_x \pi(x)$, and assume that $\pi(x) \in [1, \varepsilon^4 m]$ for all $x \in S$, and that $\varepsilon^{-1} \leq \log^{1/10} m$.*

Then we can construct a strongly history-independent hash table that maps the elements $x \in S$ to disjoint intervals $\phi(x)$ of size $\pi(x)$ in $[0, (1 + O(\varepsilon))m]$, that incurs $O(1 + \log \varepsilon^{-1})$ expected overhead per insertion/deletion, and that supports $O(1)$-time queries (with probability $1 - 1/\operatorname{poly}(m)$).

The construction is very simple: We store items $x \in S$ with sizes between 1 and ε^{-2} in a data structure A implemented using Corollary 238; and we store items $x \in S$ with sizes greater than ε^{-2} in a data structure B implemented using Proposition 242.

Note that each of A and B are dynamically resized based on the sum of the sizes of the elements they contain—in order so that insertions/deletions offer high-probability guarantees in terms of $m = \sum_{x \in S} \pi(x)$, we add $\Theta(\sqrt{m})$ unit-size dummy elements to A and $\Theta(\sqrt{m})$ ε^{-2}-sized dummy elements B, so that their sizes are guaranteed to be at least $\Omega(\sqrt{m})$ (which means that they offer high-probability bounds as a function of $\sqrt{m}$ and thus also as a function of m).

Finally, we must be careful about how to place A and B in memory together, given that each is dynamically resized. The solution is to allocate A's memory *using* B, in chunks of size $\Theta(\varepsilon^{-2})$. Note that, whenever space is added/removed to A, it is already assumed in the analysis of A that we incur an $O(1 + \log \varepsilon^{-1})$ overhead on adding/removing that space, so we can afford to allocate/deallocate that space in B.

Define m_1 (resp. m_2) to be the sum of the sizes of the elements stored in A (resp. B). By construction, A uses space at most $(1 + O(\varepsilon))m_1$ and B (which stores A within it) uses space at most $(1 + O(\varepsilon))(m_2 + |A|) + O(\varepsilon^{-3}\rho)$, where $\rho = \varepsilon^4 m$ is an upper bound on the maximum object size. Thus the total space used is $(1 + O(\varepsilon))m$.

Finally, since each of A and B supports constant-time queries with probability $1 - 1/\operatorname{poly}(m)$, and incurs $O(1 + \log \varepsilon^{-1})$ expected overhead per insertion/deletion, the same is true for C. This completes the proof of Theorem 243.

Bibliography

[35] Amd64 architecture programmer's manual volume 2: System programming. `https://www.amd.com/system/files/TechDocs/24593.pdf`. Accessed: 07/04/2021.

[36] Intel®64 and ia-32 architectures software developer's manual combined volumes 3a, 3b, 3c, and 3d: System programming guide. `https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-3a-3b-3c-and-3d-system-programming-guide.html`. Accessed: 07/04/2021.

[37] Anders Aamand, Jakob Bæk Tejs Knudsen, and Mikkel Thorup. Load balancing with dynamic set of balls and bins. In *Symposium on Theory of Computing (STOC)*, pages 1262–1275, 2021.

[38] Google's Abseil C++ library. `https://abseil.io/`. Accessed: 2020-11-06.

[39] Abseil, 2017. Accessed: 2020-11-06.

[40] George M Adel'son-Vel'skii and Evgenii Mikhailovich Landis. An algorithm for organization of information. In *Doklady Akademii Nauk*, volume 146, pages 263–266. Russian Academy of Sciences, 1962.

[41] Georgii Maksimovich Adelson-Velskii and Evgenii Mikhailovich Landis. An algorithm for organization of information. In *Doklady Akademii Nauk*, volume 146, pages 263–266. Russian Academy of Sciences, 1962.

[42] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[43] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms.* Pearson Education India, 1974.

[44] Stephen Alstrup, Gerth Brodal, and Theis Rauhe. Optimal static range reporting in one dimension. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing (STOC)*, pages 476–482, 2001.

[45] Ole Amble and Donald Ervin Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, January 1974.

[46] Arne Andersson. Improving partial rebuilding by using simple balance criteria. In *Proc. Workshop on Algorithms and Data Structures (WADS)*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer, 1989.

[47] Arne Andersson and Tony W. Lai. Fast updating of well-balanced trees. In John R. Gilbert and Rolf G. Karlsson, editors, *Proc. 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 447 of *Lecture Notes in Computer Science*, pages 111–121, July 1990.

[48] Arne Andersson and Thomas Ottmann. Faster uniquely represented dictionaries. In *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 642–649, 1991.

[49] Cecilia R Aragon and Raimund G Seidel. Randomized search trees. In *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 540–545, 1989.

[50] Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 107–118, 2009.

[51] Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 107–118, Berlin, Heidelberg, 2009.

[52] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 787–796. IEEE, 2010.

[53] Vladimir L'vovich Arlazarov, Yefim A Dinitz, MA Kronrod, and IgorAleksandrovich Faradzhev. On economical construction of the transitive closure of an oriented graph. In *Doklady Akademii Nauk*, volume 194, pages 487–488. Russian Academy of Sciences, 1970.

[54] Vladimir L'vovich Arlazarov, Yefim A Dinitz, MA Kronrod, and IgorAleksandrovich Faradzhev. On economical construction of the transitive closure of an oriented graph. In *Doklady Akademii Nauk*, volume 194, pages 487–488. Russian Academy of Sciences, 1970.

[55] Attractive Chaos Blog. Deletion from hash tables without tombstones, December 2019. Accessed 22-May-2021.

[56] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. In *Symposium on theory of computing (STOC)*, pages 593–602, 1994.

[57] Martin Babka, Jan Bulánek, Vladimír Cunát, Michal Koucký, and Michael E. Saks. On online labeling with polynomially many labels. In *ESA*, volume 7501 of *Lecture Notes in Computer Science*, pages 121–132. Springer, 2012.

[58] Martin Babka, Jan Bulánek, Vladimír Cunát, Michal Koucký, and Michael E. Saks. On online labeling with large label set. *SIAM J. Discret. Math.*, 33(3):1175–1193, 2019.

[59] Nikhil Bansal and Ohad Feldheim. Well-balanced allocation on general graphs. In *Symposium on Theory of Computing (STOC)*, 2022.

[60] Daniel Bauer. Columbia COMS W3134: Data structures in Java — Lecture 12: Introduction to hashing, October 2015.

[61] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indexes. In *Software pioneers*, pages 245–262. Springer, 2002.

[62] Djamal Belazzougui. Linear time construction of compressed text indices in compact space. In *Proceedings of the forty-sixth Annual ACM Symposium on Theory of Computing*, pages 148–193, 2014.

[63] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Theory and practice of monotone minimal perfect hashing. *Journal of Experimental Algorithmics (JEA)*, 16:3–1, 2008.

[64] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In *SODA*, pages 785–794. SIAM, 2009.

[65] Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Transactions on Algorithms (TALG)*, 16(2):1–54, 2020.

[66] Djamal Belazzougui, Travis Gagie, Veli Mäkinen, and Marco Previtali. Fully dynamic de bruijn graphs. In *International symposium on string processing and information retrieval*, pages 145–152. Springer, 2016.

[67] Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms (TALG)*, 10(4):1–19, 2014.

[68] Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms (TALG)*, 11(4):1–21, 2015.

[69] Mihir Bellare and John Rompel. Randomness-efficient oblivious sampling. In *Proceedings 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 276–287. IEEE, 1994.

[70] Michael A Bender, Jonathan W Berry, Rob Johnson, Thomas M Kroeger, Samuel McCauley, Cynthia A Phillips, Bertrand Simon, Shikha Singh, and David Zage. Anti-persistence on persistent storage: History-independent sparse tables and dictionaries. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 289–302, 2016.

[71] Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martín Farach-Colton, Rob Johnson, William Kuszmaul, Don Porter, Guido Tagliavini, Janet Vorobyeva, and Evan West. Paging and the address-translation problem. In *Proc. 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, July 2021.

[72] Michael A. Bender, Richard Cole, Erik D. Demaine, and Martin Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proc. 10th European Symposium on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pages 139–151, 2002.

[73] Michael A Bender, Richard Cole, Erik D Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proc. 10th European Symposium on Algorithms (ESA)*, pages 152–164. Springer, 2002.

[74] Michael A Bender, Alex Conway, Martín Farach-Colton, Hanna Komlós, William Kuszmaul, and Nicole Wein. Online list labeling: Breaking the log 2 n barrier. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 980–990. IEEE, 2022.

[75] Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. All-purpose hashing. *arXiv preprint arXiv:2109.04548*, 2021.

[76] Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. Tiny pointers. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 477–508. SIAM, 2023.

[77] Michael A Bender, Rathish Das, Martín Farach-Colton, Rob Johnson, and William Kuszmaul. Flushing without cascades. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 650–669. SIAM, 2020.

[78] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 399–409, Redondo Beach, California, 12–14 November 2000.

[79] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[80] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 53(2):115–136, November 2004.

[81] Michael A Bender, Martin Farach-Colton, Sándor P Fekete, Jeremy T Fineman, and Seth Gilbert. Cost-oblivious storage reallocation. *ACM Transactions on Algorithms (TALG)*, 13(3):1–20, 2017.

[82] Michael A Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 182–193. IEEE, 2018.

[83] Michael A Bender, Martin Farach-Colton, Rob Johnson, Rus C Kraner, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. Don't thrash: how to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, 2012.

[84] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string B-trees. In *Proc. 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 233–242. ACM, 2006.

[85] Michael A Bender, Martín Farach-Colton, John Kuszmaul, William Kuszmaul, and Mingmou Liu. On the optimal time/space tradeoff for hash tables. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1284–1297, 2022.

[86] Michael A Bender, Martín Farach-Colton, and William Kuszmaul. Achieving optimal backlog in multi-processor cup games. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 1148–1157, 2019.

[87] Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro. Insertion sort is $O(n \log n)$. *Theory of Computing Systems*, 39(3):391–397, 2006.

[88] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, Tsvi Kopelowitz, and Pablo Montes. File maintenance: When in doubt, change the layout! In *Procedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 1503–1522, Barcelona, Spain, 16–19 January 2017.

[89] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. 17th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 228–237, 2005.

[90] Michael A. Bender and Haodong Hu. An adaptive packed-memory array. *ACM Trans. Database Syst.*, 32(4):26:1–26:43, November 2007.

[91] Michael A Bender and William Kuszmaul. Randomized cup game algorithms against strong adversaries. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2059–2077. SIAM, 2021.

[92] Jon Bentley. *Programming pearls.* Addison-Wesley Professional, 2016.

[93] Ioana O Bercea and Guy Even. A dynamic space-efficient filter with constant time operations. In *17th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2020).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[94] Ioana O Bercea and Guy Even. Dynamic dictionaries for multisets and counting filters with constant time operations. *Algorithmica*, pages 1–19, 2022.

[95] Ioana Oriana Bercea and Guy Even. Fully-dynamic space-efficient dictionaries and filters with constant number of memory accesses. *CoRR*, abs/1911.05060, 2019.

[96] Ioana Oriana Bercea and Guy Even. A space-efficient dynamic dictionary for multisets with constant time operations. *CoRR*, abs/2005.02143, 2020.

[97] Petra Berenbrink, Artur Czumaj, Matthias Englert, Tom Friedetzky, and Lars Nagel. Multiple-choice balanced allocation in (almost) parallel. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 411–422. Springer, 2012.

[98] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: the heavily loaded case. In *Symposium on Theory of Computing (STOC)*, pages 745–754, 2000.

[99] Petra Berenbrink, Tom Friedetzky, Zengjian Hu, and Russell Martin. On weighted balls-into-bins games. *Theoretical Computer Science*, 409(3):511–520, 2008.

[100] Petra Berenbrink, Tom Friedetzky, Peter Kling, Frederik Mallmann-Trenn, Lars Nagel, and Chris Wastell. Self-stabilizing balls and bins in batches. *Algorithmica*, 80(12):3673–3703, 2018.

[101] Petra Berenbrink, Tom Friedetzky, Peter Kling, Frederik Mallmann-Trenn, Lars Nagel, and Christopher Wastell. Self-stabilizing balls & bins in batches: The power of leaky bins. In *Symposium on Principles of Distributed Computing (PODC)*, pages 83–92, 2016.

[102] Aaron Berger, William Kuszmaul, Adam Polak, Jonathan Tidor, and Nicole Wein. Memoryless worker-task assignment with polylogarithmic switching cost. In *49th International Colloquium on Automata, Languages, and*

Programming (ICALP 2022). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[103] Richard S. Bird and Stefan Sadnicki. Minimal on-line labelling. *Inf. Process. Lett.*, 101(1):41–45, 2007.

[104] Guy E. Blelloch and Daniel Golovin. Strongly history-independent hashing with applications. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007)*, pages 272–282, Providence, Rhode Island, USA, 21–23 October 2007.

[105] Guy E Blelloch and Daniel Golovin. Strongly history-independent hashing with applications. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*, pages 272–282. IEEE, 2007.

[106] Guy E Blelloch, Daniel Golovin, and Virginia Vassilevska. Uniquely represented data structures for computational geometry. In *Algorithm Theory–SWAT 2008: 11th Scandinavian Workshop on Algorithm Theory, Gothenburg, Sweden, July 2-4, 2008. Proceedings 11*, pages 17–28. Springer, 2008.

[107] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[108] P Boldi and S Vigna. Sux4j 1.0. 2008.

[109] Paolo Boldi. Minimal and monotone minimal perfect hash functions. In Giuseppe F. Italiano, Giovanni Pighizzini, and Donald Sannella, editors, *Mathematical Foundations of Computer Science 2015 - 40th International Symposium, MFCS 2015, Milan, Italy, August 24-28, 2015, Proceedings, Part I*, volume 9234 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2015.

[110] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Annual Cryptology Conference*, pages 578–595. Springer, 2011.

[111] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *European Symposium on Algorithms (ESA)*, pages 684–695. Springer, 2006.

[112] Maury Bramson, Yi Lu, and Balaji Prabhakar. Randomized load balancing with general service time distributions. *ACM SIGMETRICS performance evaluation review*, 38(1):275–286, 2010.

[113] Graham Brightwell and Malwina Luczak. The supermarket model with arrival rate tending to one. *arXiv preprint arXiv:1201.5523*, 2012.

[114] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 39–48, San Francisco, California, USA, 6–8 January 2002.

[115] Andrei Broder and Michael Mitzenmacher. Using multiple hash functions to improve ip lookups. In *Conference on Computer Communications (INFOCOM)*, volume 3, pages 1454–1463. IEEE, 2001.

[116] Andrej Brodnik, Svante Carlsson, Erik D Demaine, J Ian Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Workshop on Algorithms and Data Structures*, pages 37–48. Springer, 1999.

[117] Nathan Bronson and Xiao Shi. Open-sourcing F14 for faster, more memory-efficient hash tables, 25 April 2019. Accessed: 2020-11-06.

[118] Niv Buchbinder and Erez Petrank. Lower and upper bounds on obtaining history independence. In *Advances in Cryptology*, pages 445–462, 2003.

[119] Niv Buchbinder and Erez Petrank. Lower and upper bounds on obtaining history independence. *Information and Computation*, 204(2):291–337, 2006.

[120] Jan Bulánek, Michal Koucký, and Michael Saks. Tight lower bounds for the online labeling problem. In *Proceedings of the 44th ACM Symposium on Theory of Computing (STOC'12)*, pages 1185–1198, New York, New York, USA, 19–22 May 2012.

[121] Jan Bulánek, Michal Koucký, and Michael E. Saks. On randomized online labeling with polynomially many labels. In *Proc. International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 7965 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2013.

[122] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM Symposium on Theory of Computing (STOC)*, pages 59–65, 1978.

[123] L Elisa Celis, Omer Reingold, Gil Segev, and Udi Wieder. Balls and bins: Smaller hash families and faster evaluation. In *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 599–608, 2011.

[124] Pedro Celis. *Robin hood hashing.* University of Waterloo, 1986.

[125] Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin Hood hashing (preliminary report). In *26th Annual Symposium on Foundations of Computer Science (FOCS'85)*, pages 281–288, Portland, Oregon, USA, 21–23 October 1985.

[126] Benny Chor and Oded Goldreich. Unbiased bits from sources of weak randomness and probabilistic communication complexity. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 429–442, 1985.

[127] Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. Dictionary matching in a stream. In *Algorithms-ESA 2015*, pages 361–372. Springer, 2015.

[128] Richard Cole, Alan Frieze, Bruce M. Maggs, Michael Mitzenmacher, Andréa W Richa, Ramesh Sitaraman, and Eli Upfal. On balls and bins with deletions. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 145–158. Springer, 1998.

[129] Richard Cole, Bruce M. Maggs, Friedhelm Meyer auf der Heide, Michael Mitzenmacher, Andréa W. Richa, Klaus Schröder, Ramesh K. Sitaraman, and Berthold Vöcking. Randomized protocols for low congestion circuit routing in multistage interconnection networks. In *Symposium on Theory of Computing (STOC)*, pages 378–388. ACM, 1998.

[130] Joshimar Cordova and Gonzalo Navarro. Simple and efficient fully-functional succinct trees. *Theor. Comput. Sci.*, 656(PB):135–145, December 2016.

[131] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, USA, 3rd edition, 2009.

[132] cpppreference std::unordered_map. `https://en.cppreference.com/w/cpp/container/unordered_map`. Accessed: 2020-11-06.

[133] gcc-mirror/gcc libstdc++-v3 unordered_map.h. `https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/unordered_map.h`. Accessed: 2020-11-06.

[134] cpppreference std::unordered_set. `https://en.cppreference.com/w/cpp/container/unordered_set`. Accessed: 2020-11-06.

[135] gcc-mirror/gcc libstdc++-v3 unordered_set.h. `https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/unordered_set.h`. Accessed: 2020-11-06.

[136] cpppreference std::map. `https://en.cppreference.com/w/cpp/container/map`. Accessed: 2020-11-06.

[137] gcc-mirror/gcc libstdc++-v3 stl_map.h. `https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/stl_map.h`. Accessed: 2020-11-06.

[138] cpppreference std::set. `https://en.cppreference.com/w/cpp/container/set`. Accessed: 2020-11-06.

[139] gcc-mirror/gcc libstdc++-v3 stl_set.h. `https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/stl_set.h`. Accessed: 2020-11-06.

[140] Michael Dahlin. Interpreting stale load information. *IEEE Transactions on parallel and distributed systems*, 11(10):1033–1047, 2000.

[141] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill Education, 2006.

[142] Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti. On succinct representations of binary trees. *Mathematics in Computer Science*, 11:177–189, 2017.

[143] Lilian de Greef. UW CSE 373: Data structures and algorithims — Lecture 7: Hash table collisions, Summer 2017.

[144] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[145] Erik Demaine. MIT 6.897: Advanced data structures – Lecture 10, Spring 2012.

[146] Erik D Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Pǎtraşcu. De dictionariis dynamicis pauco spatio utentibus. In *Latin American Symposium on Theoretical Informatics (LATIN)*, pages 349–361. Springer, 2006.

[147] Erik D. Demaine and Charles E. Leiserson. MIT 6.046J/18.401J: Introduction to algorithms – Lecture 7: Hashing I, October 2005.

[148] William E Devanny, Jeremy T Fineman, Michael T Goodrich, and Tsvi Kopelowitz. The online house numbering problem: Min-max online list labeling. In *Proc. 25th European Symposium on Algorithms (ESA)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[149] Paul F. Dietz. Maintaining order in a linked list. In *Proc. 14th Annual ACM Symposium on Theory of Computing (STOC)*, pages 122–127, New York, NY, USA, 1982.

[150] Paul F Dietz, Joel I Seiferas, and Ju Zhang. A tight lower bound for on-line monotonic list labeling. In *Scandinavian Workshop on Algorithm Theory*, pages 131–142. Springer, 1994.

[151] Paul F Dietz, Joel I Seiferas, and Ju Zhang. A tight lower bound for online monotonic list labeling. *SIAM Journal on Discrete Mathematics*, 18(3):626–637, 2004.

[152] Paul F Dietz and Ju Zhang. Lower bounds for monotonic list labeling. In *Scandinavian Workshop on Algorithm Theory*, pages 173–180. Springer, 1990.

[153] M Dietzfelbinger, A Karlin, K Mehlhorn, FM auf der Heide, H Rohnert, and RE Tarjan. Dynamic perfect hashing: upper and lower bounds. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 524–531. IEEE, 1988.

[154] Martin Dietzfelbinger et al. 4.3 space complexity of monotone minimal perfect hashing. *Dagstuhl Reports, Vol. 7, Issue 5 ISSN 2192-5283*, page 19, 2018.

[155] Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable. In *International Colloquium on Automata, Languages, and Programming*, pages 235–246. Springer, 1992.

[156] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 6–19, 1990.

[157] Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming (ICALP)*, pages 385–396, 2008.

[158] Martin Dietzfelbinger and Michael Rink. Applications of a splitting trick. In *Automata, Languages and Programming: 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I 36*, pages 354–365. Springer, 2009.

[159] Martin Dietzfelbinger and Stefan Walzer. Constant-time retrieval with o (log m) extra bits. In *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[160] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. In *Proceedings of the 32nd international conference on Automata, Languages and Programming (ICALP)*, pages 166–178, 2005.

[161] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1-2):47–68, June 2007.

[162] Martin Dietzfelbinger and Philipp Woelfel. Almost random graphs with simple hash functions. In *Proceedings of the thirty-fifth Annual ACM Symposium on Theory of Computing*, pages 629–638, 2003.

[163] Peter C Dillinger and Stefan Walzer. Ribbon filter: practically smaller than bloom and xor. *arXiv preprint arXiv:2103.02515*, 2021.

[164] Alexander S Douglas. Techniques for the recording of, and reference to data in a computer. *The Computer Journal*, 2(1):1–9, 1959.

[165] Adam Drozdek and Donald L. Simon. *Data Structures in C.* PWS, Boston, Massachusetts, USA, 1995.

[166] Devdatt P Dubhashi and Alessandro Panconesi. *Concentration of measure for the analysis of randomized algorithms.* Cambridge University Press, 2009.

[167] Dwight Duffus, Hannon Lefmann, and Vojtěch Rödl. Shift graphs and lower bounds on ramsey numbers rk (l; r). *Discrete Mathematics*, 137(1-3):177–187, 1995.

[168] Marie Durand, Bruno Raffin, and François Faure. A packed memory array to keep moving particles sorted. In *VRIPHYS*, pages 69–77. Eurographics Association, 2012.

[169] Yuval Emek and Amos Korman. New bounds for the controller problem. *Distributed Computing*, 24(3-4):177–186, 2011.

[170] Paul Erdős and András Hajnal. On chromatic number of graphs and set-systems. *Acta Math. Acad. Sci. Hungar*, 17(61-99):1, 1966.

[171] Jeff Erickson. UIUC CS473: Algorithms — Lecture 5: Hash tables, 2017.

[172] Patrick Eschenfeldt and David Gamarnik. Supermarket queueing system in the heavy traffic regime. short queue dynamics. *arXiv preprint arXiv:1610.03522*, 2016.

[173] Facebook's F14 hash table. `https://engineering.fb.com/2019/04/25/developer-tools/f14/`. Accessed: 2020-11-06.

[174] Rolf Fagerberg, David Hammer, and Ulrich Meyer. On optimal balance in B-trees: What does it cost to stay in perfect shape? In *ISAAC*, volume 149 of *LIPIcs*, pages 35:1–35:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[175] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[176] Arash Farzan and J. Ian Munro. Succinct representation of dynamic trees. *Theoretical Computer Science*, 412(24):2668 – 2678, 2011. Selected Papers from 36th International Colloquium on Automata, Languages and Programming (ICALP).

[177] Ohad N Feldheim and Ori Gurel-Gurevich. The power of thinning in balanced allocation. *Electronic Communications in Probability*, 26:1–8, 2021.

[178] Jon Feldman, Aranyak Mehta, Vahab Mirrokni, and Shan Muthukrishnan. Online stochastic matching: Beating 1-1/e. In *Symposium on Foundations of Computer Science (FOCS)*, pages 117–126. IEEE, 2009.

[179] Gene Fisher. CalPoly CSC103: Fundamentals of computer science – hashing, 2001.

[180] Philippe Flajolet, Patricio Poblete, and Alfredo Viola. On the analysis of linear probing hashing. *Algorithmica*, 22(4):490–515, December 1998.

[181] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G Spirakis. Space efficient hash tables with worst case constant access time. In *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science (STOC)*, pages 271–282, 2003.

[182] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, December 2005.

[183] Pierre-Alain Fouque and Mehdi Tibouchi. Close to uniform prime number generation with fewer random bits. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 991–1002. Springer, 2014.

[184] Gianni Franceschini and Roberto Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In *Workshop on Algorithms and Data Structures (WADS)*, pages 114–126. Springer, 2003.

[185] Michael L. Fredman, Janos Komlos, and Endre Szemeredi. Storing a sparse table with O(1) worst case access time. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS)*, SFCS '82, page 165–169. IEEE Computer Society, 1982.

[186] Michael L Fredman and Dan E Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the twenty-second annual ACM Symposium on Theory of Computing (STOC)*, pages 1–7, 1990.

[187] Michael J Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–19. Springer, 2004.

[188] Alan Frieze and Michal Karonski. *Introduction to Random Graphs.* Cambridge University Press, 2015.

[189] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS'99)*, pages 285–297, 17–19 October 1999.

[190] Zoltan Füredi, Péter Hajnal, Vojtech Rödl, and William T Trotter. Interval orders and shift graphs. 1992.

[191] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *Journal of the ACM (JACM)*, 67(1):1–54, 2020.

[192] Anna Gál, Meena Mahajan, Rahul Santhanam, and Till Tantau. Computational complexity of discrete problems (dagstuhl seminar 21121). In *Dagstuhl Reports*, volume 11. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[193] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 165–174. ACM/SIAM, 1993.

[194] Daniel Golovin. *Uniquely represented data structures with applications to privacy.* PhD thesis, Carnegie Mellon University, 2008.

[195] Daniel Golovin. B-treaps: A uniquely represented alternative to B-trees. In *Proc. 36th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 487–499. 2009.

[196] Daniel Golovin. The B-skip-list: A simpler uniquely represented alternative to B-trees. *arXiv preprint arXiv:1005.0662*, 2010.

[197] Gaston H. Gonnet and Per-Åke Larson. External hashing with limited internal storage. *J. ACM*, 35(1):161–184, 1988.

[198] Michael T Goodrich, Daniel S Hirschberg, Michael Mitzenmacher, and Justin Thaler. Fully de-amortized cuckoo hashing for cache-oblivious dictionaries and multimaps. *arXiv preprint arXiv:1107.4378*, 2011.

[199] Michael T Goodrich, Daniel S Hirschberg, Michael Mitzenmacher, and Justin Thaler. Cache-oblivious dictionaries and multimaps with negligible failure probability. In *Mediterranean Conference on Algorithms*, pages 203–218. Springer, 2012.

[200] Michael T Goodrich, Evgenios M Kornaropoulos, Michael Mitzenmacher, and Roberto Tamassia. Auditable data structures. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 285–300. IEEE, 2017.

[201] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design and Applications.* Wiley, Hoboken, New Jersey, USA, 2015.

[202] David Gries and Doug James. Cornell CS210: Object-oriented programming and data structures — recitation week 8: Hashing, Fall 2014.

[203] Roberto Grossi, Alessio Orlandi, and Rajeev Raman. Optimal trade-offs for succinct string indexes. In *International Colloquium on Automata, Languages, and Programming*, pages 678–689. Springer, 2010.

[204] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 8–21, 1978.

[205] Takao Gunji and E Goto. Studies on hashing part-1: A comparison of hashing algorithms with key deletion. *J. Information Processing*, 3(1):1–12, 1980.

[206] Bernhard Haeupler, Vahab S Mirrokni, and Morteza Zadimoghaddam. Online stochastic weighted matching: Improved approximation algorithms. In *International Workshop on Internet and Network Economics*, pages 170–181. Springer, 2011.

[207] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. *Journal of Algorithms*, 41(1):69–85, October 2001.

[208] Jason D Hartline, Edwin S Hong, Alexander E Mohr, William R Pentney, and Emily C Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005.

[209] Jason D Hartline, Edwin S Hong, Alexander E Mohr, William R Pentney, and Emily C Rocke. Characterizing history independent data structures. *Algorithmica*, 42:57–74, 2005.

[210] Micha Hofri and Alan G. Konheim. Padded lists revisited. *SIAM Journal on Computing*, 16(6):1073–1114, 1987.

[211] F. R. A. Hopgood and J. Davenport. The quadratic hash method when the table size is a power of 2. *The Computer Journal*, 15(4):314–315, 1972.

[212] Russell Impagliazzo and Valentine Kabanets. Constructive proofs of concentration bounds. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 617–631. Springer, 2010.

[213] Alon Itai and Irit Katriel. Canonical density control. *Informatino Processing Letters*, 104(6):200–204, December 2007.

[214] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *Proceedings 8th International Colloquium on Automata, Languages, and Programming (ICALP 1981)*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, Acre (Akko), Israel, July 1981.

[215] Svante Janson and Alfredo Viola. A unified approach to linear probing hashing with buckets. *Algorithmica*, 75(4):724–781, August 2016.

[216] Morten Skaarup Jensen and Rasmus Pagh. Optimality in external memory hashing. *Algorithmica*, 52(3):403–411, November 2008.

[217] Rosa M. Jiménez and Conrado Martínez. On deletions in open addressing hashing. In *Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 23–31, New Orleans, Louisana, USA, 8–9 January 2018. SIAM.

[218] Eyal Kaplan, Moni Naor, and Omer Reingold. Derandomized constructions of k-wise (almost) independent permutations. *Algorithmica*, 55(1):113–133, 2009.

[219] Irit Katriel. Implicit data structures based on local reorganizations. Master's thesis, Technion – Israel Inst. of Tech., Haifa, May 2002.

[220] Krishnaram Kenthapadi and Rina Panigrahy. Balanced allocation on graphs. In *Symposium on Discrete Algorithms (SODA)*, volume 6, pages 434–443, 2006.

[221] Gregory Kesden. CMU 15-310: System-level software development — hashing review, 2007. Accessed 31-May-2021.

[222] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. Fast and scalable inequality joins. *The VLDB Journal*, 26(1):125–150, 2017.

[223] Jon Kleinberg and Éva Tardos. *Algorithm Design.* Pearson Addison-Wesley, Boston, Massachusetts, USA, 2006.

[224] Don Knuth. Notes on "open" addressing, 1963.

[225] Donald Knuth. personal communication, December 2022.

[226] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[227] Donald Ervin Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms.* Addison-Wesley, 3rd edition, 1997.

[228] Alan G. Konheim and Benjamin Weiss. An occupancy discipline and applications. *SIAM Journal on Applied Mathematics*, 14(6):1266–1274, November 1966.

[229] Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *Proc. 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 283–292, 2012.

[230] Robert L. Kruse. *Data Structures and Program Design.* Prentice-Hall Inc, Englewood Cliffs, New Jersey, USA, 1984.

[231] William Kuszmaul. Achieving optimal backlog in the vanilla multi-processor cup game. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1558–1577. SIAM, 2020.

[232] William Kuszmaul. How asymmetry helps buffer management: achieving optimal tail size in cup games. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 1248–1261, 2021.

[233] Per-Åke Larson. Performance analysis of linear hashing with partial expansions. *ACM Transactions on Database Systems*, 7(4):566–687, December 1982.

[234] Per-Åke Larson. Analysis of uniform hashing. *J. ACM*, 30(4):805–819, 1983.

[235] Per-Åke Larson. Linear hashing with separators - A dynamic hashing scheme achieving one-access retrieval. *ACM Trans. Database Syst.*, 13(3):366–388, 1988.

[236] Per-Åke Larson and Ajay Kajla. File organization: Implementation of a method guaranteeing retrieval in one access. *Commun. ACM*, 27(7):670–677, 1984.

[237] Christoph Lenzen, Merav Parter, and Eylon Yogev. Parallel balanced allocations: The heavily loaded case. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 313–322, 2019.

[238] Dean De Leo and Peter A. Boncz. Fast concurrent reads and updates with PMAs. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 8:1–8:8. ACM, 2019.

[239] Dean De Leo and Peter A. Boncz. Packed memory arrays - rewired. In *35th IEEE International Conference on Data Engineering (ICDE)*, pages 830–841. IEEE, 2019.

[240] Dean De Leo and Peter A. Boncz. Teseo and the analysis of structural dynamic graphs. *Proc. VLDB Endowment 14*, 14(6):1053–1066, 2021.

[241] Harry R. Lewis and Larry Denenberg. *Data Structures and Their Algorithms.* HarperCollins Publishers, New York, New York, USA, 1991.

[242] Kim-Hung Li. Reservoir-sampling algorithms of time complexity o (n (1+ log (n/n))). *ACM Transactions on Mathematical Software (TOMS)*, 20(4):481–493, 1994.

[243] Tianxiao Li, Jingxun Liang, Huacheng Yu, and Renfei Zhou. Tight cell-probe lower bounds for dynamic succinct dictionaries. In *64th Annual Symposium on Foundations of Computer Science (FOCS)*, 2023.

[244] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13, 2011.

[245] Mingmou Liu, Yitong Yin, and Huacheng Yu. Succinct filters for sets of unknown sizes. In *47th International Colloquium on Automata, Languages, and Programming (ICALP)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[246] Dimitrios Los and Thomas Sauerwald. Balanced allocations in batches: Simplified and generalized. *arXiv preprint arXiv:2203.13902*, 2022.

[247] Dimitrios Los and Thomas Sauerwald. Balanced allocations with incomplete information: The power of two queries. In *Innovations in Theoretical Computer Science Conference (ITCS)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[248] Dimitrios Los, Thomas Sauerwald, and John Sylvester. Balanced allocations: Caching and packing, twinning and thinning. In *Symposium on Discrete Algorithms (SODA)*, pages 1847–1874. SIAM, 2022.

[249] Shachar Lovett and Ely Porat. A lower bound for dynamic approximate membership data structures. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 797–804. IEEE, 2010.

[250] Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.

[251] Malwina J Luczak and Colin McDiarmid. On the maximum queue length in the supermarket model. *The Annals of Probability*, 34(2):493–527, 2006.

[252] Malwina J Luczak and James Norris. Strong approximation for the supermarket model. *The Annals of Applied Probability*, 15(3):2038–2061, 2005.

[253] Michael Main and Walter Savitch. *Data Structures and Other Objects Using C++*. Addison-Wesley, Boston, Massachusetts, USA, 2001.

[254] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European conference on Computer Systems (EuroSys'12)*, pages 183–196, Bern, Switzerland, 10–13 April 2012.

[255] W. D. Maurer. An improved hash code for scatter storage. *Communications of the ACM*, 11(1):35–38, 1968.

[256] Colin McDiarmid. On the method of bounded differences. *Surveys in combinatorics*, 141(1):148–188, 1989.

[257] Michael McMillan. *Data Structures and Algorithms with JavaScript.* O'Reilly, Sebastopol, California, USA, 2014.

[258] Kurt Mehlhorn. On the program size of perfect and universal hash functions. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 170–175. IEEE, 1982.

[259] Kurt Mehlhorn, Peter Sanders, and Peter Sanders. *Algorithms and data structures: The basic toolbox*, volume 55. Springer, 2008.

[260] Raghu Meka, Omer Reingold, Guy N Rothblum, and Ron D Rothblum. Fast pseudorandomness for independence and load balancing. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 859–870. Springer, 2014.

[261] Raghu Meka, Omer Reingold, and Yuan Zhou. Deterministic Coupon Collection and Better Strong Dispersers. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM)*, volume 28 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 872–884, 2014.

[262] Haim Mendelson and Uri Yechiali. A new approach to the analysis of linear probing schemes. *Journal of the ACM*, 27(3):474–483, July 1980.

[263] Daniele Micciancio. Oblivious data structures: applications to cryptography. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 456–464, 1997.

[264] Shyamal Mitra. UT CS 313E: Elements of software design — hashing, Spring 2021.

[265] Michael Mitzenmacher. How useful is old information (extended abstract)? In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 83–91, 1997.

[266] Michael Mitzenmacher. Studying balanced allocations with differential equations. *Combinatorics, Probability and Computing*, 8(5):473–482, 1999.

[267] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[268] Michael Mitzenmacher, Andrea W. Richa, and Ramesh Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.

[269] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, 2nd edition, 2017.

[270] Michael Molloy and Bruce Reed. Graph coloring and the probabilistic method. *New York I Springer*, 23:1329–356, 2002.

[271] Robert Morris. Scatter storage techniques. *Communications of the ACM*, 11(1):38–44, January 1968.

[272] Christian Worm Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 618–627. ACM/SIAM, 2003.

[273] Christian Worm Mortensen, Rasmus Pagh, and Mihai Pătraçcu. On dynamic range reporting in one dimension. In *Proceedings of the thirty-seventh annual ACM Symposium on Theory of Computing (STOC)*, pages 104–111, 2005.

[274] Dave Mount. UMD CMSC 420: Data structures — lecture 11: Hashing — handling collisions, Spring 2019.

[275] Debankur Mukherjee, Sem C Borst, Johan SH Van Leeuwaarden, and Philip A Whiting. Universality of power-of-d load balancing in many-server systems. *Stochastic Systems*, 8(4):265–292, 2018.

[276] J. Ian Munro, Venkatesh Raman, and Adam J. Storm. Representing dynamic binary trees succinctly. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 529–536, USA, 2001. Society for Industrial and Applied Mathematics.

[277] Moni Naor and Omer Reingold. On the construction of pseudorandom permutations: Luby-Rackoff revisited. *Journal of Cryptology*, 12(1):29–66, 1999.

[278] Moni Naor, Gil Segev, and Udi Wieder. History-independent cuckoo hashing. In *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35*, pages 631–642. Springer, 2008.

[279] Moni Naor and Vanessa Teague. Anti-persistence: History independent data structures. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 492–501, 2001.

[280] Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys (CSUR)*, 46(4):1–47, 2014.

[281] Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3), May 2014.

[282] Yakov Nekrich. Space efficient dynamic orthogonal range reporting. *Algorithmica*, 49(2):94–108, 2007.

[283] Yakov Nekrich. Orthogonal range searching in linear and almost-linear space. *Computational Geometry*, 42(4):342–351, 2009.

[284] Yakov Nekrich. Searching in dynamic catalogs on a tree. *Computing Research Repository (CoRR)*, abs/1007.3415, 2010.

[285] Yakov Nekrich. A dynamic stabbing-max data structure with sub-logarithmic query time. *Computing Research Repository (CoRR)*, abs/1109.3890, 2011.

[286] A. Newell and J. C. Shaw. Programming the Logic Theory Machine. In *Proceedings of the Western Joint Computer Conference: Techniques for Reliability*, Los Angeles, California, USA, 26–28 February 1957.

[287] Allen Newell and Herbert A. Simon. The logic theory machine—a complex information processing system. *IRE Transactions on Information Theory*, 2(3):61–79, September 1956.

[288] Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. In *Proc. 4th Annual ACM Symposium on Theory of Computing (STOC)*, pages 137–142, 1972.

[289] Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[290] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Symposium on Operating Systems Principles (SOSP)*, pages 29–41, 2011.

[291] Anna Ostlin and Rasmus Pagh. Uniform hashing in constant time and linear space. In *Proceedings of the thirty-fifth annual ACM Symposium on Theory of Computing (STOC)*, pages 622–628, 2003.

[292] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[293] Anna Pagh and Rasmus Pagh. Uniform hashing in constant time and optimal space. *SIAM Journal on Computing*, 38(1):85–96, 2008.

[294] Anna Pagh, Rasmus Pagh, and S Srinivasa Rao. An optimal bloom filter replacement. In *Proceedings of the sixteenth annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 823–829, 2005.

[295] Rasmus Pagh. Faster deterministic dictionaries. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 487–493, USA, February 2000.

[296] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[297] Rasmus Pagh. Dispersing hash functions. *Random Structures & Algorithms*, 35(1):70–82, 2009.

[298] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Proceedings of the 9th European Symposium on Algorithms (ESA 2001)*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133, University of Aaarhus, Denmark, 28–31 August 2001. Springer.

[299] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *European Symposium on Algorithms (ESA)*, pages 121–133. Springer, 2001.

[300] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[301] Rasmus Pagh, Gil Segev, and Udi Wieder. How to approximate a set without knowing its size in advance. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 80–89. IEEE, 2013.

[302] Rasmus Pagh, Zhewei Wei, Ke Yi, and Qin Zhang. Cache-oblivious hashing. *Algorithmica*, 69(4):864–883, 2014.

[303] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluç. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proc. 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1372–1385, 2021.

[304] Mihai Patrascu. Succincter. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313. IEEE, 2008.

[305] Mihai Pătraşcu and Mikkel Thorup. The power of simple tabulation hashing. *Journal of the ACM (JACM)*, 59(3):1–50, 2012.

[306] Mihai Patrascu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proceedings of the 2014 IEEE 55th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 166–175, 2014.

[307] Christos Pelekis. Lower bounds on binomial and Poisson tails: An approach via tail conditional expectations. arXiv:1609.06651, September 2016.

[308] Yuval Peres, Kunal Talwar, and Udi Wieder. The (1+ β)-choice process and weighted balls-into-bins. In *Symposium on Discrete Algorithms (SODA)*, pages 1613–1619. SIAM, 2010.

[309] W. W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, April 1957.

[310] William Pugh. *Incremental computation and the incremental evaluation of functional programs.* PhD thesis, Cornell University, 1988.

[311] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[312] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Proc. 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 315–328, 1989.

[313] Rajeev Raman and Satti Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 357–368, 2003.

[314] Vijayshankar Raman. Locality preserving dictionaries: Theory and application to clustering in databases. In *Proc. 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 337–345, 1999.

[315] Omer Reingold, Ron D Rothblum, and Udi Wieder. Pseudorandom graphs in data structures. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 943–954. Springer, 2014.

[316] Stefan Richter, Victor Alvarez, and Jens Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB – Porceedings of the VLDB Endowment*, 9(3):96–107, November 2015. The 42nd International Conference on Very Large Data Bases, New Delhi, India.

[317] J. P. Royston. Expected normal order statistics (exact and approximate). *Journal of the Royal Statistical Society*, 31:161–165, 1982.

[318] Milan Ružić. Uniform deterministic dictionaries. *ACM Transactions on Algorithms*, 4(1):1–23, March 2008.

[319] Michael Saks. Online labeling: Algorithms, lower bounds and open questions. In *International Computer Science Symposium in Russia (CSR)*, volume 10846, pages 23–28. Springer, 2018.

[320] Peter Sanders. Hashing with linear probing and referential integrity. *arXiv preprint arXiv:1808.04602*, 2018.

[321] Edward R Scheinerman and Daniel H Ullman. *Fractional graph theory: a rational approach to the theory of graphs*. Courier Corporation, 2011.

[322] Keith Schwarz. Stanford CS166: Data structures — linear probing, Spring 2021.

[323] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, Massachusetts, USA, 1983.

[324] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, Reading, Massachusetts, USA, 1990.

[325] Raimund Seidel and Cecilia R Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, 1996.

[326] A Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 20–25, 1989.

[327] Alan Siegel. On universal classes of extremely random constant-time hash functions. *SIAM Journal on Computing*, 33(3):505–543, 2004.

[328] Gábor Simonyi and Gábor Tardos. On directed local chromatic number, shift graphs, and borsuk-like graphs. *Journal of Graph Theory*, 66(1):65–82, 2011.

[329] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[330] Peter Smith. *Applied Data Structures with C++*. Jones & Bartlett Learning, 2004.

[331] Lawrence Snyder. On uniquely represented data structures. In *Proc. 18th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 142–146, 1977.

[332] Thomas A. Standish. *Data Structures, Algorithms, and Software Principles in C*. Addision-Wesley, Reading, Massachusetts, USA, 1995.

[333] Volker Stemann. Parallel balanced allocations. In *Symposium on Parallel algorithms and Architectures (SPAA)*, pages 261–269, 1996.

[334] Hsin-Hao Su, Lili Su, Anna Dornhaus, and Nancy Lynch. Ant-inspired dynamic task allocation via gossiping. In *Stabilization, Safety, and Security of Distributed Systems: 19th International Symposium, SSS 2017, Boston, MA, USA, November 5–8, 2017, Proceedings 19*, pages 157–171. Springer, 2017.

[335] Hsin-Hao Su and Nicole Wein. Lower bounds for dynamic distributed task allocation. In *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[336] David G. Sullivan. Harvard CS S-111: Intensive introduction to computer science using Java — unit 9, part 4: Hash tables, Summer 2021.

[337] Rajamani Sundar. A lower bound for the dictionary problem under a hashing model. In *Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 612–621, San Juan, Puerto Rico, USA, October 1991.

[338] Rajamani Sundar and Robert Endre Tarjan. Unique binary search tree representations and equality-testing of sets and sequences. In *Proc. 22nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 18–25, 1990.

[339] Kunal Talwar and Udi Wieder. Balanced allocations: the weighted case. In *Symposium on Theory of Computing (STOC)*, pages 256–265, 2007.

[340] Kunal Talwar and Udi Wieder. Balanced allocations: A simple proof for the heavily loaded case. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 979–990. Springer, 2014.

[341] Mikkel Thorup. Mihai Pătraşcu: Obituary and open problems. *Bulletin of EATCS*, 1(109), 2013.

[342] Julio Toss, Cicero Augusto de Lara Pahins, Bruno Raffin, and João Luiz Dihl Comba. Packed-memory quadtree: A cache-oblivious data structure for visual exploration of streaming spatiotemporal big data. *Computers & Graphics*, 76:117–128, 2018.

[343] Jean-Paul Tremblay and Paul G. Sorenson. *An Introduction to Data Structures with Applications*. McGraw-Hill, 1984.

[344] Salil P. Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1–3):1–336, 2012.

[345] Alfredo Viola. Exact distribution of individual displacements in linear probing hashing. *ACM Transactions on Algorithms (TALG)*, 1(2):214–242, October 2005.

[346] Alfredo Viola. Distributional analysis of the parking problem and Robin Hood linear probing hashing with buckets. *Discrete Mathematics and Theoretical Computer Science (DMTCS)*, 12(2), January 2010.

[347] Alfredo Viola and Patricio V. Poblete. The analysis of linear probing hashing with buckets (extended abstract). In *Algorithms — ESA '96, Fourth Annual European Symposium*, volume 1136 of *Lecture Notes in Computer Science*, pages 221–233. Springer, 1996.

[348] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[349] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CsUR)*, 33(2):209–271, 2001.

[350] Berthold Vöcking. How asymmetry helps load balancing. In *Foundations of Computer Science (FOCS)*, page 131, 1999.

[351] Berthold Vöcking. How asymmetry helps load balancing. *Journal of the ACM (JACM)*, 50(4):568–589, 2003.

[352] Nikita Dmitrievna Vvedenskaya, Roland L'vovich Dobrushin, and Fridrikh Izrailevich Karpelevich. Queueing system with selection of the shortest of two queues: An asymptotic approach. *Problemy Peredachi Informatsii*, 32(1):20–34, 1996.

[353] Mark Allen Weiss. *Data Structures and Problem Solving using C++*. Addison-Wesley, Reading, Massachusetts, USA, 2000.

[354] Jay Wengrow. *A Common-Sense Guide to Data Structures and Algorithms*. The Pragmatic Programmers, 2017.

[355] Brian Wheatman and Randal Burns. Streaming sparse graphs using efficient dynamic sets. In *IEEE BigData*, pages 284–294. IEEE, 2021.

[356] Brian Wheatman and Helen Xu. Packed compressed sparse row: A dynamic graph representation. In *HPEC*, pages 1–7. IEEE, 2018.

[357] Brian Wheatman and Helen Xu. A parallel packed memory array to store dynamic graphs. In *Proc. Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 31–45. SIAM, 2021.

[358] Udi Wieder et al. Hashing, load balancing and multiple choice. *Foundations and Trends® in Theoretical Computer Science*, 12(3–4):275–379, 2017.

[359] Wikipedia contributors. Linear probing, 2021. Accessed 31-May-2021.

[360] Wikipedia contributors. Linked list, 2021. Accessed 22-May-2021.

[361] Wikipedia contributors. Primary clustering, 2021. Accessed 22-May-2021.

[362] Wikipedia contributors. Quadratic probing, 2021. Accessed 31-May-2021.

[363] Dan E. Willard. Inserting and deleting records in blocked sequential files. Technical Report TM81-45193-5, Bell Labs Tech Reports, 1981.

[364] Dan E. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC'82)*, pages 114–121, San Francisco, California, USA, May 1982.

[365] Dan E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD'86)*, pages 251–260, Washington, DC, USA, May 1986.

[366] Dan E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, April 1992.

[367] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1976.

[368] Niklaus Wirth. *Algorithms and Data Structures*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1986.

[369] Philipp Woelfel. Asymmetric balanced allocation with simple hash functions. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 424–433. ACM Press, 2006.

[370] Huacheng Yu. Nearly optimal static las vegas succinct dictionary. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1389–1401, 2020.

[371] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: Substrate support for path splitting and migration. *ACM SIGCOMM Computer Communication Review*, 38(2):17–29, 2008.

[372] Ju Zhang. *Density control and on-line labeling problems*. PhD thesis, University of Rochester, 1993.