On the Optimal Time/Space Tradeoff for Hash Tables

Michael A. Bender
Stony Brook University
USA
bender@cs.stonybrook.edu

Martín Farach-Colton
Rutgers University
USA
martin@farach-colton.com

John Kuszmaul
Yale University
USA
john.kuszmaul@yale.edu

William Kuszmaul
MIT
USA
kuszmaul@mit.edu

Mingmou Liu
Nanyang Technological University
Singapore
mingmou.liu@ntu.edu.sg

ABSTRACT

For nearly six decades, the central open question in the study of hash tables has been to determine the optimal achievable tradeoff curve between time and space. State-of-the-art hash tables offer the following guarantee: If keys/values are $\Theta(\log n)$ bits each, then it is possible to achieve constant-time insertions/deletions/queries while wasting only $O(\log \log n)$ bits of space per key when compared to the information-theoretic optimum—this bound has been proven to be optimal for a number of closely related problems (e.g., stable hashing, dynamic retrieval, and dynamically-resized filters).

This paper shows that $O(\log \log n)$ wasted bits per key is not the end of the line for hashing. In fact, for any $k \in [\log^* n]$, it is possible to achieve $O(k)$-time insertions/deletions, $O(1)$-time queries, and the k-th iterated logarithm $O(\log^{(k)} n)$ wasted bits per key (all with high probability in n), while also supporting dynamic resizing as the size of the table changes. We further show that this tradeoff curve is the best achievable by any of a large class of hash tables, including any hash table designed using the current framework for making constant-time hash tables succinct.

Our result holds for arbitrarily large keys/values, and in the case where keys/values are very small, we can tighten our bounds to $o(1)$ wasted bits per key. Building on this, we obtain a constant-time dynamic filter that uses $n\left\lceil \log \epsilon^{-1} \right\rceil + n \log e + o(n)$ bits of space for a wide choice of false-positive rates ϵ, resolving a long-standing open problem for the design of dynamic filters.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**.

KEYWORDS

hash tables, succinct, balls and bins, data structures

STOC '22, June 20–24, 2022, Rome, Italy

ACM ISBN 978-1-4503-9264-8/22/06…$15.00
https://doi.org/10.1145/3519935.3519969

ACM Reference Format:
Michael A. Bender, Martín Farach-Colton, John Kuszmaul, William Kuszmaul, and Mingmou Liu. 2022. On the Optimal Time/Space Tradeoff for Hash Tables. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing (STOC '22), June 20–24, 2022, Rome, Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3519935.3519969

1 INTRODUCTION

A ***hash table*** [21] (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 [1–3, 10, 11, 11, 12, 18–22, 28, 29, 31, 32, 34]. Whereas early hash tables [20, 21] required $\omega(1)$ time per operation in order to support a load factor of $1 - o(1)$, modern hash tables [2, 3, 22] 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 [32] with constant expected-time operations, and which after a long line of work [2, 3, 10, 22] has now also been achieved [3] 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] [10, 13, 25] and fully-dynamic approximate

[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 [13], and dynamic value-retrieval requires $nv + \Theta(n \log \log n)$ bits [10, 25].

set membership[2] [9, 22, 30]. And it is known that *stable* hash tables [3, 10] (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 operations yield major space savings?). Tight answers to these questions would close off one of the longest-standing directions of research in the field of data structures.

A Steep Tradeoff Curve Between Time and Space. In this paper, we present a data structure that achieves a much stronger time/space tradeoff than existing hash tables, and we prove a matching lower bound establishing that our hash table is optimal across a large family of data structures that includes all existing fast succinct hash tables.

We show that it is possible to achieve significantly fewer than $O(\log \log n)$ wasted bits per key. In fact, 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.

As we mentioned above, we prove that the tradeoff curve on which these hash tables sit is tight for a large class of data structures, including all known dynamic succinct hash tables [2–4, 7, 22, 32], and more generally, any hash table that makes implicit (or explicit) use of "augmented open addressing" (discussed in more detail in Section 2.1).

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

$$n \log \epsilon^{-1} + n \log e + o(n)$$

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 [4, 7, 9, 22, 27].

Outline. In Section 2, we present a detailed discussion of our results, and we give a brief overview of the techniques that are used to achieve them. In Section 3, we establish tight upper and lower bounds for the so-called ***probe-complexity problem***—the upper bound, in particular, serves as the main algorithmic building block for all of our data structural results. Due to space limitations, we defer the data-structural applications of our balls-and-bins results to the full version of this paper [6]—several of the applications require significant additional technical ideas, which we are only able to touch upon here.

2 OVERVIEW OF RESULTS AND TECHNIQUES

This section presents an overview of the main results and techniques in the paper.

2.1 The Relationship Between 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 [2–4, 7, 22, 32]), 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 3, we will give tight upper and lower bounds for the probe-complexity problem.

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 $m = (1+\epsilon)n$, 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 $\epsilon = o(1)$ (we call the quantity $1 - \epsilon$ 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 [21]. An important difference is that the query-routing data structure allows for queries

[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 $n \log \epsilon^{-1}$ bits [9], an optimal resizable filter requires $n \log \epsilon^{-1} + \Omega(n \log \log n)$ space [30], which is known to be optimal for $\epsilon^{-1} \leq \text{polylog } n$ [22, 30].

[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.

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 [3, 22, 32], a hash table might use augmented open addressing on bins of size 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 [2–4, 7, 22, 32] 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 $m = (1+\epsilon)$ slots. The balls-to-slots scheme is measured by two objectives: the average ***probe complexity*** of the balls, 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 [21], double hashing [21], or Cuckoo hashing [29]. Cuckoo hashing [29] (and its variants [1, 14, 18]) 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 $1 - \epsilon < 1/2$. Generalizations of Cuckoo hashing (i.e., d-ary Cuckoo hashing [18] and Cuckoo hashing with d-slot bins [14]) are able to support higher load factors $1 - \epsilon$, but at the cost of incurring a super-constant switching cost of at least $\Omega(\log \epsilon^{-1})$. 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 [2]).

To achieve small probe complexity and switching cost, while also supporting $\epsilon = o(1)$, 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 [24]). 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 [10, 25] 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 [2–4, 7, 22, 32]. Raman and Rao [32] 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 [2, 4, 7, 22], which allowed for them to overcome the query-routing bottleneck, but resulted in a worse space utilization. Recently, [3] 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 our paper 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.

Theorem 1 (Probe Complexity Upper Bound)**.** *For any $k \in [\log^* n - 1]$, the k-kick tree is a balls-to-slots scheme that stores*

up to n balls in n slots, achieves worst-case switching cost $k+1$, and achieves expected average probe complexity $O(\log^{(k+1)} n)$.

We prove that this tradeoff between switching cost and probe complexity is asymptotically optimal (as long as the load factor $1-\epsilon$ 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. This lower bound is established via an intricate potential-function argument that we consider to be one of the main technical contributions of the paper.[4]

Theorem 2 (Probe Complexity Lower Bound)**.** *Suppose the universe U has sufficiently large polynomial size. Let $\epsilon = 1/\log^{(b)} n$ where $b \le (\log^* n)/4$. Consider any balls-to-slots scheme that stores up to $(1-\epsilon)n+1$ balls in n slots, and that achieves expected average probe complexity $O(\mathrm{tow}(a))$ (across all balls in the system at any given moment). The expected amortized switching cost per insertion/deletion must be at least*

$$\Omega(\log^* n - a - b).$$

Interpreting our lower bound as a statement about augmented open-addressed hash tables, we can conclude that if there exists a hash table with a better time/space tradeoff curve than the hash tables in this paper, it would have to fundamentally avoid the use of augmented open addressing, and would instead require an entirely new approach to succinct hashing.

Corollary 3 (Augmented Open Addressing Lower Bound)**.** *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 $\Omega(k)$.*

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 paper, 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. The full details can be found in the extended version [6] of the paper.

An Improved Query Router (Section 4 of [6]). 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.

[4]Note that, in Section 3, where we present Theorem 2, we adapt the convention that n is the number of slots, rather than balls. This is in keeping with the balls-and-bins literature which, unfortunately, follows the opposite convention from the hash-table literature—hence the reformulation of n in Theorem 2.

Supporting Dynamic Resizing (Section 5.2 of [6]). Past approaches [3, 22, 32] have performed resizing at a granularity of $1+1/\mathrm{polylog}\, n$ factors.[5] This has required the data to be partitioned into $\mathrm{polylog}\, n$ 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.

Combining our results, we arrive at the following theorem:

Theorem 4 (Succinct Resizable Hashing)**.** *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.

Handling Large Keys/Values (Section 6.1 of [6]). Now consider the setting where the keys and values are u and v bits long, respectively, for some potentially large u, v satisfying $u+v \le n^{o(1)}$. 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 [2, 22, 32]. 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 [32], which achieves $O(\log(u+v))$ wasted bits per key with constant expected-time insertions—subsequent work [2, 22] 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 $u \ge \omega(\log n)$ 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 paper is agnostic to key/value size: with $O(k)$-time per insertion/deletion, we can achieve $O(\log^{(k)} n)$ wasted bits per key.

[5]The specific ways in which resizing has been implemented have differed, with some papers [22, 32] performing resizing at a per-bin level, and others [3] performing it globally.

Theorem 5 (Large Keys). *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 $k \geq 0$.*

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, and $a+b \leq n^{o(1)}$, 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.

Handling Small Keys/Values (Section 6.2 of [6]). In addition to considering large keys, past work [1, 10, 22, 32] 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.

Theorem 6 (Small keys). *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.

Prior to our work, this type of guarantee was only known to be possible in the much smaller regime of $t = \tilde{O}((\log n)^{1/3})$ [2, 32]. 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.

2.3 An Application to Optimal Dynamic Filters

Finally, in Section 6.3 of the extended paper [6], 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 $\mathcal{F}(n, \epsilon) = n \log \epsilon^{-1}$ bits [9].

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 $\mathcal{F}(n, \epsilon) + nr$ 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 $r = \Omega(\log \log n)$ [30].

Filters tend to be used in applications where space efficiency is a central concern; the result is that most applications select ϵ such that $\log^{-1} \epsilon$ is very small (for a practical discussion of filters, see, e.g., [5, 16, 17]). 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 [8], which supports $O(\epsilon^{-1})$-time insertions and achieves $r = O(\log \epsilon^{-1})$ (the Bloom filter does not support deletions). After a long line of work [4, 7–9, 22, 27], contemporary filters are able to achieve much stronger bounds than this. Indeed, there are now a number of filters [4, 7, 27] that and that achieve

$$r = o(\log \epsilon^{-1})$$

wasted bits per key for all ϵ satisfying

$$\log \epsilon^{-1} \in [\omega(1), O(\log n)],$$

while supporting constant-time insertions/deletions/queries either in expectation [27] or in the worst case [3, 7] (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 [23] (namely, $\epsilon = \Theta(1)$), 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

$$r = \log e + o(1) = O(1) \tag{1}$$

wasted bits per key for all inverse-power-of-two ϵ satisfying

$$\log \epsilon^{-1} \in [\omega(1), \log n / \log^{(k)} n],$$

while supporting worst-case constant-time insertions/deletions/queries (with high probability). The total space used by the data structure is therefore

$$n \log \epsilon^{-1} + n \log e + o(n)$$

bits. This resolves the question of whether $r = O(1)$ is achievable in all cases except for when $\log^{-1} \epsilon$ is very close to $\log n$. Finally, we show that for any value of $\log^{-1} \epsilon$ (including $\log^1 \epsilon = \Theta(\log n)$), 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. Combining our results, we arrive at the following theorem:

Theorem 7 (Succinct Filters). *Let $\epsilon^{-1} \in [\omega(1), O(\log n)]$ 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 $n \log \epsilon^{-1} + n \log e + o(n)$ bits if $\epsilon^{-1} \leq (\log n)/\log^{(k)} n$ and $\log^{(k)} n = \omega(1)$; and space at most $n \log \epsilon^{-1} + O(n) + O(n \log^{(k)} n)$ bits otherwise. The time and space guarantees hold for each operation with high probability in n.*

We remark that the specific constant $\log e$ that we achieve in Equation (1) is information-theoretically optimal for any filter that is constructed by storing fingerprints in a hash table. (This includes all modern dynamic filters [4, 7, 9, 22, 27].) 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.

2.4 Preliminaries

We conclude the section by formalizing several preliminaries that we will need throughout the paper.

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. We say that an event occurs with high probability (w.h.p.) in n if it occurs with probability $1 - 1/\text{poly}(n)$.

Simulating Fully Random Hash Functions. Whereas early work on hash tables [11, 12, 19] was bottlenecked by the known families of hash functions, there are now well-established techniques [1, 3, 15, 26, 33] for simulating fully random hash functions in hash tables. Notably, Siegel [33] showed that for some positive $\epsilon > 0$, there is a family of constant-time hash functions that can be constructed in time $o(n)$ and that is n^ϵ-independent.[6] In the context of hash tables, this can be amplified to simulate $\text{poly}(n)$-independence [1, 3] 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 $\text{poly}(m)$-independent. Thus we can assume without loss of generality that we have access to $\text{poly}(n)$-independent 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 $\text{polylog}(n)$ slabs of memory at a time).

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[7], and let $\epsilon \in [0, 1)$ 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 $(1-\epsilon)n + 1$ 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 $i = \text{argmin}_j\{h_j(x) = r\}$.

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 3.1), achieving probe-complexity $O(\log^{(k)} n)$ with switching cost $O(k)$. This holds even when $\epsilon = 1/n$, meaning that there are up to n balls present at a time (and there are up to $n - 1$ balls present prior to any given insertion).

We then also prove a matching lower bound in Subsection 3.2: any balls-to-slots scheme that supports $\epsilon \leq 1/\log^{(O(1))}(n)$ with expected average probe complexity $O(\log^{(k)} n)$ must incur average switching cost $\Omega(k)$. Note that, whereas our upper bound supports $\epsilon = 1/n$ (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.

3.1 A Balls-to-Slots Scheme with Small Average Probe Complexity

In this section, we fix $\epsilon = 1/n$, 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.

[6] 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 $\text{poly}(n)$-size universe using a pairwise independent hash function.

[7] 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 paper we follow the hash-table convention.

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 8. *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

$$\mathbb{E}[\log s_j] \le \log s_k + \sum_{i\in[0,k)} \Pr[g_{i+1}(x) \text{ saturated}] \cdot \log s_i. \quad (2)$$

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 $s(y) \ge i+1$. For a given $y \ne x$,

$$\begin{aligned}
&\Pr[g_{i+1}(y) = g_{i+1}(x) \text{ and } s(y) \ge i+1] \\
&= \Pr[g_{i+1}(y) = g_{i+1}(x)] \cdot \Pr[s(y) \ge i+1] \\
&= \frac{1}{n/s_{i+1}} \cdot (1 - \Pr[s(y) < i+1]) \\
&= \frac{1}{n/s_{i+1}} \cdot \left(1 - 1/(\log^{(i+1)} n)^2\right).
\end{aligned}$$

The number Y of such y therefore satisfies

$$\begin{aligned}
\mathbb{E}[Y] &\le n \cdot \frac{1}{n/s_{i+1}} \cdot \left(1 - 1/(\log^{(i+1)} n)^2\right) \\
&= s_{i+1} \cdot \left(1 - 1/(\log^{(i+1)} n)^2\right) \\
&= (\log^{(i+1)} n)^6 - (\log^{(i+1)} n)^4.
\end{aligned}$$

Since Y is a sum of independent indicator random variables, we can apply a Chernoff bound to deduce that

$$\begin{aligned}
\Pr[Y \ge (\log^{(i+1)} n)^6] &\le e^{-\Omega(\log^{(i+1)} n)^2} \\
&\le O\left(\frac{1}{(\log^{(i)} n)^2}\right).
\end{aligned}$$

This is an upper bound on the probability that $g_{(i+1)}(x)$ is saturated. Thus, by (2),

$$\begin{aligned}
\mathbb{E}[\log s_j] &\le \log s_k + \sum_{i\in[0,k)} \frac{1}{(\log^{(i)} n)^2} \cdot \log s_i \\
&= O\left(\log^{(k+1)} n + \sum_{i\in[0,k)} \frac{1}{(\log^{(i)} n)^2} \cdot \log^{(i+1)} n\right) \\
&= O(\log^{(k+1)} n).
\end{aligned}$$

This completes the proof of the lemma. □

It's worth taking a moment to understand the bottlenecks in the Lemma 8. 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 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 8 is:

Theorem 1 (Probe Complexity Upper Bound)**.** *For any $k \in [\log^* n - 1]$, the k-kick tree is a balls-to-slots scheme that stores up to n balls in n slots, achieves worst-case switching cost $k+1$, and achieves expected average probe complexity $O(\log^{(k+1)} n)$.*

In the extended version of the paper [6], we also show how to transform our bound on expected average probe complexity into a high-probability bound.

Theorem 9. *For any $k \in [\log^* n - 1]$, there is a balls-to-slots scheme with $\epsilon = 1/n$ 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.*

3.2 A Lower-Bound on the Relationship Between Switching Cost and 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 $q(h, i, j) < \text{poly}(i)$ 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 $\epsilon = 1/n$ 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 10. *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 $\epsilon = 1/n$. 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 $M = \text{poly}(n)$ insertions and the same number of deletions. The insertions and deletions alternate; each insertion inserts a random ball (which with probability $1 - 1/\text{poly}(n)$ 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 $i \le \text{tow}(L)$, 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 $i \ge n$, 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 11. *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

$$\begin{aligned}\mathbb{E}[J_*] &= \mathbb{E}[J_0] + LM - \Theta(M) - \mathbb{E}[\psi] \\ &= \mathbb{E}[J_0] + \Theta(LM) - \mathbb{E}[\psi],\end{aligned}$$

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

$$\Theta(LM) - \mathbb{E}[\psi] \le O(n \log^* n).$$

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 10 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 11 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 10. The main challenge is therefore to construct a potential function ϕ with the three desired properties.

Constructing the Potential Function ϕ. 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

$$\phi = \max_{S \in \mathcal{S}} (I(S) - c|S|)$$

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 ϕ. 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 12. *Let a_1 be a random ball in A_1 and let a_2 be a random ball in A_2. With probability $1 - 1/\text{poly}(L)$, 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

$$\begin{aligned}\mathbb{E}[d_i(b)] &= 2 \cdot n \Pr_{x \in U}[h_k(x) = b \text{ for some } k \le \text{tow}(L+i)] \\ &= 2 \cdot q(h, \text{tow}(L+i), b) \\ &\le 2 \cdot \text{poly}(\text{tow}(L+i)) \\ &\le (\text{tow}(L+i))^c/2.\end{aligned}$$

Applying a Chernoff bound, it follows that for all $D \ge (\text{tow}(L+i))^c$, we have

$$\Pr[d_i(b) \ge D] \le \frac{1}{2^{\Omega(D)}}.$$

Thus

$$\begin{aligned}\mathbb{E}[d_i(b) \cdot \mathbb{I}_{d_i(b) \ge (\text{tow}(L+i))^c}] &\le \frac{1}{2^{\Omega((\text{tow}(L+i))^c)}} \\ &= \frac{1}{2^{\text{poly}(\text{tow}(L+i))}}.\end{aligned}$$

This means that the expected sum S of the degrees of the high-degree slots in G_i satisfies

$$\mathbb{E}[S] \le n/2^{\text{poly}(\text{tow}(L+i))}.$$

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 $\text{poly}(\text{tow}(L+i))$. It follows that the number λ of nodes in G_i that are within distance $O(L)$ of a high-degree node satisfies

$$\mathbb{E}[\lambda] \le S \cdot \text{poly}(\text{tow}(L+i))^{O(L)},$$

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

$$\mathbb{E}[\lambda] \le \frac{n}{2^{\text{poly}(\text{tow}(L+i))}} \cdot \text{poly}(\text{tow}(L+i))^{O(L)}.$$

The above quantity is dominated by its first factor, so

$$\mathbb{E}[\lambda] \le \frac{n}{2^{\text{poly}(\text{tow}(L+i))}}.$$

Applying Markov's inequality, we have that with probability $1 - 1/2^{\text{poly}(\text{tow}(L+i))} \ge 1 - 1/\text{poly}(L)$,

$$\lambda \le \frac{n}{2^{\text{poly}(\text{tow}(L+i))}}.$$

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

$$\begin{aligned}&\Pr\left[\lambda > \frac{n}{2^{\text{poly}(\text{tow}(L+i))}}\right] + \frac{\frac{n}{2^{\text{poly}(\text{tow}(L+i))}}}{\Theta(n)} \\ &= \frac{1}{\text{poly}(L)} + \frac{1}{2^{\text{poly}(\text{tow}(L+i))}} \\ &\le \frac{1}{\text{poly}(L)}.\end{aligned}$$

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 $\ge i+3$.

Lemma 13. *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 $\epsilon = 1/n$, 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 $\ge i+3$.

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,

$$\mathbb{E}[|X|] \le \frac{O(n)}{\text{tow}(L+i+1)}, \tag{3}$$

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,

$$|X| \le \frac{n \,\text{poly}(L)}{\text{tow}(L+i+1)}$$

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

$$\frac{n \,\text{poly}(L)}{\text{tow}(L+i+1)} \,\text{tow}(L+i)^{O(L)} \ll \frac{n}{\text{poly}(L)}. \tag{4}$$

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 14 (Establishing Property 1 for ϕ)**.** *Each insertion/deletion increases ϕ by at most $1/\text{poly}(L)$ 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 $\ge i+3$—the probability of this occurring is therefore an upperbound on the expected increase to ϕ_{i+3} due to the insertion.

By Lemma 12, we have with probability $1 - 1/\text{poly}(L)$ 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 13, we have that with probability $1 - 1/\text{poly}(L)$, that within the graph G_i', a is not within distance $O(L)$ of any node with level $\ge i+3$ (besides a itself). Thus, with probability $1 - 1/\text{poly}(L)$, we have that in G_i, a is not within distance $O(L)$ of any node with level $\ge i+3$ (besides a itself). This establishes that, with probability $1 - 1/\text{poly}(L)$, 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 $O(1/\text{poly}(L))$.

Now consider the deletion of a random ball $a \in A_1$. By the same reasoning as in the preceding paragraph, with probability $1-1/\text{poly}(L)$, 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 $1-1/\text{poly}(L)$ 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 $1/\text{poly}(L)$.

In either case, the expected increase to

$$\phi = \sum_{i=3}^{L} \phi_i$$

is at most $\sum_{i=3}^{L} 1/\text{poly}(L) = 1/\text{poly}(L)$. Thus the lemma is proven. □

Next we analyze the effect that a given move by the adversary has on ϕ.

Lemma 15 (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 10 is proven.

Generalizing to Arbitrary ϵ and Large Probe Lengths. So far we have assumed for simplicity that $\epsilon = 1/n$ and that the balls-to-slots scheme being analyzed achieves expected average probe complexity $O(1)$. Next, we generalize our lower bound to consider $\epsilon \geq 1/n$ and probe complexity $\omega(1)$.

Theorem 16. *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 $\epsilon = 1/\log^{(b)} n$ where $b \leq (\log^* n)/4$.*[8] *The*

[8] In this notation, if $b = 0$, then $\epsilon = 1/n$.

expected amortized switching cost per insertion/deletion must be at least

$$\Omega(\log^* n - a - b).$$

We defer the proof of Theorem 16 to the extended paper [6].

Removing the Near-Uniformity Assumption. The lower bound also extends to handle 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 17. *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, $c(A, h') \leq O(c(A, h) + n)$.*

We defer the proof of Lemma 17 to the extended paper [6].

By the preceding lemma, the assumption in Theorem 16 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 2 (Probe Complexity Lower Bound)**.** *Suppose the universe U has sufficiently large polynomial size. Let $\epsilon = 1/\log^{(b)} n$ where $b \leq (\log^* n)/4$. Consider any balls-to-slots scheme that stores up to $(1-\epsilon)n + 1$ balls in n slots, and that achieves expected average probe complexity $O(\text{tow}(a))$ (across all balls in the system at any given moment). The expected amortized switching cost per insertion/deletion must be at least*

$$\Omega(\log^* n - a - b).$$

Corollary 18. *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 $\epsilon = 1/n$. 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 3 (Augmented Open Addressing Lower Bound)**.** *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 $\Omega(k)$.*

ACKNOWLEDGMENTS

This research was supported in part by NSF grants CSR-1938180, CCF-2106999, CCF-2118620, CCF-2118832, CCF-2106827, CCF-1725543, CSR-1763680, CCF-1716252 and CNS-1938709. Kuszmaul is funded by an NSF GRFP fellowship and a Fannie and John Hertz Fellowship. Mingmou is currently supported by Singapore Ministry of Education (AcRF) Tier 1 grant RG75/21, and was previously supported by Singapore Ministry of Education (AcRF) Tier 2 grant MOE2018-T2-1-013.

This research was also partially sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was 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.

REFERENCES

[1] Yuriy Arbitman, Moni Naor, and Gil Segev. 2009. De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP)*. 107–118.

[2] Yuriy Arbitman, Moni Naor, and Gil Segev. 2010. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 787–796.

[3] Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2021. All-Purpose Hashing. *arXiv preprint arXiv:2109.04548* (2021).

[4] Michael A Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. 2018. Bloom filters, adaptivity, and the dictionary problem. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 182–193.

[5] Michael A Bender, Martin Farach-Colton, Rob Johnson, Rus C Kraner, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. 2012. Don't thrash: how to cache your hash on flash. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1627–1637.

[6] Michael A Bender, Martín Farach-Colton, John Kuszmaul, and William Kuszmaul. 2021. On the Optimal Time/Space Tradeoff for Hash Tables. *arXiv preprint arXiv:2111.00602* (2021).

[7] Ioana O Bercea and Guy Even. 2020. 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.

[8] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[9] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. 1978. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM Symposium on Theory of Computing (STOC)*. 59–65.

[10] Erik D Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Pătraşcu. 2006. De dictionariis dynamicis pauco spatio utentibus. In *Latin American Symposium on Theoretical Informatics*. Springer, 349–361.

[11] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. 1990. 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)*. 6–19.

[12] M Dietzfelbinger, A Karlin, K Mehlhorn, FM auf der Heide, H Rohnert, and RE Tarjan. 1988. Dynamic perfect hashing: upper and lower bounds. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 524–531.

[13] Martin Dietzfelbinger and Rasmus Pagh. 2008. Succinct Data Structures for Retrieval and Approximate Membership. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming (ICALP)*. 385–396.

[14] Martin Dietzfelbinger and Christoph Weidling. 2005. Balanced allocation and dictionaries with tightly packed constant size bins. In *Proceedings of the 32nd international conference on Automata, Languages and Programming (ICALP)*. 166–178.

[15] Martin Dietzfelbinger and Philipp Woelfel. 2003. Almost random graphs with simple hash functions. In *Proceedings of the thirty-fifth annual ACM Symposium on Theory of Computing (STOC)*. 629–638.

[16] Peter C Dillinger and Stefan Walzer. 2021. Ribbon filter: practically smaller than Bloom and Xor. *arXiv preprint arXiv:2103.02515* (2021).

[17] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 75–88.

[18] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G Spirakis. 2003. Space Efficient Hash Tables with Worst Case Constant Access Time. In *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science (STOC)*. 271–282.

[19] Michael L. Fredman, Janos Komlos, and Endre Szemeredi. 1982. 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)*. IEEE Computer Society, 165–169.

[20] Don Knuth. 1963. Notes on "open" addressing. (1963).

[21] Donald E. Knuth. 1973. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley.

[22] Mingmou Liu, Yitong Yin, and Huacheng Yu. 2020. Succinct Filters for Sets of Unknown Sizes. In *47th International Colloquium on Automata, Languages, and Programming (ICALP)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[23] Shachar Lovett and Ely Porat. 2010. A lower bound for dynamic approximate membership data structures. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 797–804.

[24] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

[25] Christian Worm Mortensen, Rasmus Pagh, and Mihai Pătraçcu. 2005. On dynamic range reporting in one dimension. In *Proceedings of the thirty-seventh annual ACM Symposium on Theory of Computing (STOC)*. 104–111.

[26] Anna Ostlin and Rasmus Pagh. 2003. Uniform hashing in constant time and linear space. In *Proceedings of the thirty-fifth annual ACM Symposium on Theory of Computing (STOC)*. 622–628.

[27] Anna Pagh, Rasmus Pagh, and S Srinivasa Rao. 2005. An optimal Bloom filter replacement. In *Proceedings of the sixteenth annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 823–829.

[28] Rasmus Pagh. 2001. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.* 31, 2 (2001), 353–363.

[29] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *Journal of Algorithms* 51, 2 (May 2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[30] Rasmus Pagh, Gil Segev, and Udi Wieder. 2013. How to approximate a set without knowing its size in advance. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 80–89.

[31] Mihai Patrascu. 2008. Succincter. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE, 305–313.

[32] Rajeev Raman and Satti Srinivasa Rao. 2003. Succinct dynamic dictionaries and trees. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*. 357–368.

[33] A Siegel. 1989. 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)*. 20–25.

[34] Huacheng Yu. 2020. Nearly optimal static Las Vegas succinct dictionary. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*. 1389–1401.