1 Persistent Trees

Full copy bad.
Fat nodes method:

- replace each (single-valued) field of data structure by list of all values taken, sorted by time.
- requires $O(1)$ space per data change (**unavoidable** if keep old date)
- to lookup data field, need to search based on time.
- store values in binary tree
- checking/changing a data item takes $O(\log m)$ time after m updates
- slowdown of $O(\log m)$ in structure access.

Path copying:

- much of data structure consists of fixed-size *nodes* conencted by *pointers*
- can only reach node by traversing pointers starting from *root*
- changes to a node only visible to *ancestors* in pointer structure
- when change a node, copy it and ancestors (back to root of data structure
- keep list of roots sorted by update time
- $O(\log m)$ time to find right root (or const, if time is integers)
- same access time as original structure
- *additive* instead of multiplicative $O(\log m)$.
- modification time and space usage equals number of ancestors: possibly huge!

Combined Solution (trees only):

- in each node, store 1 *extra* time-stamped field
- if full, overrides one of standard fields for any accesses later than stamped time.
- access rule
  - standard access, just check for overrides while following pointers
  - constant factor increase in access time.
- update rule:
  - when need to change/copy pointer, use extra if available.

  - otherwise, make new copy of node with new info, and recursively modify parent.

- Analysis
  - *live* node: pointed at by *current* root.
  - potential function: number of *full* live nodes.
  - copying a node is free (new copy not full, pays for copy space/time)
  - pay for filling an extra pointer (do only once, since can stop at that point).
  - amortized space per update: $O(1)$.

Power of twos: Like Fib heaps. Show binary tree of modifications.
Application: persistent trees.

- amortized cost $O(1)$ to change a field.
- splay tree has $O(\log n)$ amortized field change per access.
- $O(\log n)$ space per access!
- drawback: rotations on access mean unbounded space usage.

Red-black trees:

- aggressive rebalancers
- store red/black bit in each node
- use red/black bit to rebalance.
- depth $O(\log n)$
- search: standard binary tree search; no changes
- update: causes changes in red/black fields on path to item, $O(1)$ rotations.
- result: $(\log n)$ space *per insert/delete*
- geometry does $O(n)$ changes, so $O(n \log n)$ space.
- $O(\log n)$ query time.

Improvement:

- red-black bits used only for updates
- only need current version of red-black bits
- don't store old versions: just overwrite
- only updates needed are for $O(1)$ rotations

- so $O(1)$ space per update
- so $O(n)$ space overall.

Result: $O(n)$ space, $O(\log n)$ query time for planar point location.
Extensions:

- method extends to arbitrary pointer-based structures.
- $O(1)$ cost per update for any pointer-based structure with any constant indegree. s
- full persistence with same bounds.

2 Suffix Trees

Crochemore and Rytter, *Text Algorithms*
Gusfield: Algorithms on Strings, Trees, and Sequences.
Weiner 73 "Linear Pattern-matching algorithms" IEEE conference on automata and switching theory
McCreight 76 "A space-economical suffix tree construction algorithm" JACM 23(2) 1976
Chen and Seifras 85 "Efficient and Elegegant Suffix tree construction" in Apostolico/Galil *Combninatorial Algorithms on Words*
Another "search" structure, dedicated to strings.
Basic problem: match a "pattern" to "text"

- goal: decide if a given string ("pattern") is a substring of the text
- possibly created by concatenating short ones, eg newspaper
- application in IR, also computational bio (DNA seqs)
- if pattern avilable first, can build DFA, run in time linear in text
- if text available first, can build suffix tree, run in time linear in pattern.

First idea: binary tree on strings. Inefficient because run over pattern many times.
Tries:

- Idea like bucket heaps: use bounded alphabet Σ.
- used to store dictionary of strings
- trees with children indexed by "alphabet"
- time to search equal length of query string
- insertion ditto.

- optimal, since even hashing requires this time to hash.
- but better, because no "hash function" computed.
- space an issue:
  - using array increases stroage cost by $|\Sigma|$
  - using binary tree on alphabet increases search time by $\log |\Sigma|$
  - ok for "const alphabet"
- size in worst case: sum of word lengths (so pretty much solves "dictionary" problem.
-

But what about substrings?

- idea: trie of all n^2 substrings
- equivalent to trie of all n suffixes.
- put "marker" at end, so no suffix contained in other (otherwise, some suffix can be an internal node, "hidden" by piece of other suffix)
- means one leaf per suffix
- Naive construction: insert each suffix
- basic alg:
  - text $a_1 \cdots a_m$
  - define $s_i = a_i \cdots a_m$
  - for $i = 1$ to m
  - insert s_i
- time, space $O(m^2)$

Better construction:

- note trie size may be much smaller: $aaaaaaa$.
- algorithm with time $O(|T|)$
- idea: avoid repeated work by "memoizing"
- (also shades of finger search tree idea)
- suppose just inserted aw
- next insert is w
- big prefix of w might already be in trie

- avoid traversing: skip to end of prefix.

Suffix links:

- any node in trie corresponds to string
- arrange for node corresp to ax to point at node corresp to x
- suppose just inserted aw.
- walk up tree till find suffix link
- follow link (puts you on path corresp to w)
- walk down tree (adding nodes) to insert rest of w

Memoizing: (save your work)

- can add suffix link to every node we walked up
- (since walked up end of aw, and are putting in w now).
- charging scheme: charge traversal up a node to creation of suffix link
- traversal up also covers (same length) traversal down
- once node has suffix link, never passed up again
- thus, total time spent going up/down equals number of suffix links
- one suffix link per node, so time $O(|T|)$

Suffix Trees:

- problem: maybe $|T|$ is large (m^2)
- compress paths in suffix trie
- path on letters $a_i \cdots a_j$ corresp to substring of text
- replace by edge labelled by (i, j) (*implicit nodes*)
- gives tree where every node has indegree at least 2
- in such a tree, size is order number of leaves $= O(m)$
- Search still works:
  - preserves invariant: *at most* one edge starting with given character leaves a node
  - store edges in array indexed by starting character.
  - walk down same as trie, but use indexing into text to find chars
  - called "slowfind" for later

Construction:

- obvious: build suffix trie, compress
- drawback: may take m^2 time and intermediate space
- better: use original construction idea, work in compressed domain.
- as before, insert suffixes in order $s_1, \ldots, s_m$
- compressed tree of what inserted so far
- to insert s_i, walk down tree
- at some point, path diverges from what's in tree
- may force us to "break" an edge (show)
- tack on *one* new edge for rest of string (cheap!)

MacReight 1976

- use suffix link idea of up-link-down
- problem: can't suffix link every character, only explict nodes
- want to work proportional to *real* nodes traversed
- need to skip characters inside edges (since can't pay for them)
- introduced "fastfind"
  - idea: fast alg for descending tree if *know* string present in tree
  - just check first char on edge, then skip number of chars equal to edge "length"
  - may land you in middle of edge (specified offset)
  - cost of search: number of *explicit* nodes in path
  - pay for with suffix links

Analysis:

- suppose just inserted string aw
- sitting on its leaf, which has *parent*
- invariant: every internal node except for parent of current leaf has suffix link to another explicit node
- plausible?
  - suppose s_j and s_k diverge (creating explicit node) at v

  - claim s_{j+1} and s_{k+1} diverge at suffix(v), creating another explicit node.
  - only problem if s_{k+1} not yet present
  - happens only if k is current suffix
  - only blocks parent of current leaf.
- insertion step:
  - consider parent p_i and *grandparent* (parent of parent) g_i of current node
  - g_i to p_i link has string w_1
  - p_i to l_i link w_2
  - go up to grandparent
  - follow suffix link
  - *fastfind* w_1
  - claim: know w_1 is present in tree!
  - create suffix link from p_i (preserves invariant)
  - *slowfind* w_2 (stopping when leave current tree)
  - break current edge
  - add new edge for rest of w_2

Analysis:

- Break into two costs: from suf(g_i) to suf(p_i) (w_1), then suf(p_i) to p_{i+1} (w_2).
- slowfind cost is time to get from suf(p_i) to p_{i+1} (plus const)
- (note p_{i+1} is descendant of suf(p_i))
- so total cost $O(\sum |p_{i+1} - p_i| + 1) = O(n)$
- what about time to find suf(p_i)?
- fastfind costs less than slowfind, so at most $|g_{i+1}| - |g_i|$ to reach g_{i+1}. Sums to linear
- Done if g_{i+1} below suf(p_i) (double counts, but who cares)
- what if g_{i+1} above suf(p_i)?
- can only happen if suf(p_i) = p_{i+1} (this is only node below g_{i+1})
- in this case, fastfind takes 1 step to go from g_{i+1} to p_{i+1} (landing in middle of edge)
- only case where fastfind necessary, but can't tell in advance.

Weiner algorithm: insert strings "backwards", use prefix links. Ukonnen online version.
Suffix arrays
Applications:

- preprocess bottom up, storing first, last, num. of suffixes in subtree
- allows to answer queries: what first, last, count of w in text in time $O(|w|)$.
- enumerate k occurrences in time $O(w + |k|)$ (traverse subtree, binary so size order of number of occurences.
- longest common substring. work bottom up, deciding if subtree contains suffixes of both strings. Take deepest node satisfying.