1 Persistent Data Structures

"Making Data Structures Persistent" by Driscoll, Sarnak, Sleator and Tarjan Journal of Computer and System Sciences 38(1) 1989

Idea: be able to query and/or modify past versions of data structure.

• ephemeral: changes to struct destroy all past info

• partial persistence: changes to most recent version, query to all past versions

• full persistence: queries and changes to all past versions (creates “multiple worlds” situation)

Goal: general technique that can be applied to any data structure.
Application: planar point location.

• planar subdivision
  – \( n \) segments meeting only at ends
  – defines set of polygons
  – query: “what polygon contains this point”

• numerous special-purpose solutions

• One solution:
  – vertical line through each vertex
  – divides into slabs
  – in slab, segments maintain one vertical ordering
  – find query point slab by binary search
  – build binary search tree for slab with “above-below” queries
  – \( n \) binary search trees, size \( O(n^2) \), time \( O(n^2 \log n) \)

• observation: trees all very similar

• think of \( x \) axis as time, slabs as “epochs”

• at end of epoch, “delete” segments that end, “insert” those that start.

• over all time, only \( n \) inserts, \( n \) deletes.

• must be able to query over all times
Persistent sorted sets:

- find\((x, s, t)\) find (largest key below) \(x\) in set \(s\) at time \(t\)
- insert\((i, s, t)\) insert \(i\) in \(s\) at time \(t\)
- delete\((i, s, t)\).

We use partial persistence: updates only in “present”
Implement via persistent search trees.
Result: \(O(n)\) space, \(O(\log n)\) query time for planar point location.

2 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
- multiplicative slowdown of \(O(\log m)\) in structure access.

Path copying:

- much of data structure consists of fixed-size nodes connected 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) (additive slowdown)
- 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 field 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 red-black trees:

• aggressive rebalancers

• amortized cost $O(1)$ to change a field.

• 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.