Persistent Data Structures

Inge Li Gørtz
Persistent Data Structures

• **Ephemeral data structures.** A modification destroys the old version.

• **Persistent data structures.** Modifications nondestructive. Each modification creates a new version. All versions coexists.

  • **Partially persistent data structures.** Can access all versions, but only update the latest version.

  • **Fully persistent data structures.** Can access and modify all versions (branching). An update operation only operates on a single version at a time (cannot combine several versions).
Partially persistent data structures. Can access all versions, but only update the latest version.

• Versions: linear ordering.
Full persistence

- Fully persistent data structures. Can access and modify all versions.

- Versions: form a tree.
Overview

• Algorithmic applications

• Partial persistence. Three methods on balanced binary search tree.
  • Fat node.
  • Path copying.
  • Node copying.

• Full persistence. Main idea.
Algorithmic Applications
Planar Point Location

- **Planar point location.** Euclidean plane subdivided into polygons by \( n \) line segments that intersect only at their endpoints. Query: given a query point \( p \) determine which polygon that contains \( p \).

- **Measure** data structure by three parameters:
  - Preprocessing time
  - Query time
  - Space
Planar point location: Example
Planar point location: Example

From slides by H. Kaplan
Planar Point Location

• Dobkin-Lipton.

• Partition plane into vertical slabs by drawing a vertical line through its endpoint.

• Within each slab the lines are totally ordered.

• Create a search tree, $T_i$, per slab containing the lines and associate with each line the polygon above it.

• Create another search tree, $T_x$, on the x-coordinates of the vertical lines.
Planar Point Location

• Queries:

1. Find appropriate slab i (using $T_x$).
2. Search slab to find polygon (using $T_i$).
Planar point location: Example

From slides by H. Kaplan
Planar Point Location

• Queries:
  1. Find appropriate slab $i$ (using $T_x$).
  2. Search slab to find polygon (using $T_i$).

• Query time: $O(\log n)$

• Space: $\Omega(n^2)$

• Preprocessing time: $\Omega(n^2)$
Planar Point Location

- # lines = $O(n)$, number of lines in each slab $O(n)$. 

mandag den 07. mar 2011
Planar Point Location

• Improve space bound?

• Key observation. The lists of lines in adjacent swaps are very similar.

  • Create search tree for first slab.

  • Obtain next one by deleting lines ending at corresponding vertex, and adding lines starting at that vertex.

• \#insertions/deletions all together?

• 2n
Planar Point Location

- **Sarnak and Tarjan.** Sweep line + partially persistent binary search tree:
  - Preprocessing time: $O(n \log n)$
  - Query time: $O(\log n)$
  - Space $O(n)$
Partiel Persistence
Simple methods

• Let \( m \) = #modifications (versions).

• **Copy** entire data structures before each modification: \( \Omega(n) \) time per update, \( \Omega(nm) \) space.

• Keep **log** of all updates (don’t do anything else when updating). When accessing version \( i \) perform first the \( i \) updates in order to obtain version \( i \): \( \Omega(i) \) time per access, \( O(m) \) space.

• **Hybrid method:** Store entire sequence of updates and in addition every \( k \)th version for some suitable \( k \). Either space or access time blows up by a factor of \( \sqrt{m} \).

• Better methods?
Fat Node Method

- **Fat nodes.** Keep record of all changes to each field of the data structure.

- **BST.**
  - Fat node contains same fields as ephemeral node. Every field can store many values, each tagged with a version number (version stamp).
Fat Node Method

• Update i:
  - Ephemeral update creates new node: create new fat node, each value of a field in the new node has version stamp i.
  - Ephemeral update change a value changes a field: store field value plus version stamp.

• Access version i:
  - Choose value with maximum version stamp no greater than i.
Fat Node Method

- Use binary search tree or y-fast trie to keep track of version stamps for each field.
- Time cost per access: $O(\log m)$ slowdown per node.
- Time and space cost per update: $O(1)$ per ephemeral operation.

- BST: using BST with $O(1)$ memory modifications per update for the version stamps.
  - Find: $O(\log^2 n)$
  - Insert/delete: $O(\log n)$ time
  - Space: $O(n)$
Path Copying Method

- **Observation:** All modifications occur on one path => suffices to copy one path.

- **Update:**
  - Create copy of node and its ancestors before changing it to point to new child.
  - Every modification creates a new root.
  - Maintain predecessor data structure of roots indexed by version stamp.
Path Copying Method

- Use binary search tree to find correct root.
- Time cost per access: $O(1)$ slowdown per node.
- Time and space cost per update: $O(\log n)$ per ephemeral operation.

- BST:
  - Find: $O(\log n)$
  - Insert/delete: $O(\log n)$ time
  - Space: $O(n \log n)$
Node Copying Method

- **Idea.** Similar to fat node method, but don’t make nodes too fat.

- **BST.**
  - Keep on extra pointer field in each node (with version stamp).
  - Ephemeral update allocate new node: allocate new node as well.
  - Ephemeral update change pointer: if extra pointer field empty use it, otherwise copy node. Try to store pointer to new node in parent. If extra pointer in parent occupied copy parent.... (cascade changes up).
Node Copying - Analysis

- **Time per access.** Slowdown $O(1)$ per access step.
- **Finding correct root.** Predecessor query.
- **Time and space per update.** Amortized $O(1)$ per ephemeral operation.

- **BST:** using BST with $O(1)$ memory modifications per update for the version stamps.
  - **Find:** $O(\log n)$
  - **Insert/delete:** $O(\log n)$ time
  - **Space:** $O(n)$
Node Copying - Analysis

- Amortized $O(1)$ copying per update.
- Live node: node reachable from root of most recent version.
- Full node: extra pointer field used.
- Potential function: $\Phi(D) = \#\text{full live nodes}$.

- Amortized space consumption = actual space + $\Delta\Phi$. 
Partially Persistent Data Structures


- Any data structure can be made partially persistent with slowdown $O(\log m)$ for queries and $O(1)$ for updates. The space cost is $O(1)$ for each ephemeral memory modification.

- Any data structure can be made fully persistent on a RAM with slowdown $O(\log \log m)$ for queries and expected slowdown $O(\log \log m)$ for updates. The space cost is $O(1)$ for each ephemeral memory modification.

- Any bounded-degree linked data structure can be made partially persistent with (worst-case) slowdown $O(1)$ for queries, amortized slowdown $O(1)$ for updates, and amortized space cost $O(1)$ per memory modification.
Fully Persistent Data Structures


  - Any data structure can be made fully persistent with slowdown $O(\log m)$ for both queries and updates. The space cost is $O(1)$ for each ephemeral memory modification.

  - Any bounded-degree linked data structure can be made fully persistent with (worst-case) slowdown $O(1)$ for queries, amortized slowdown $O(1)$ for updates, and amortized space cost $O(1)$ per memory modification.

- Dietz, 1989. Any data structure can be made fully persistent on a RAM with slowdown $O(\log \log m)$ for queries and expected slowdown $O(\log \log m)$ for updates. The space cost is $O(1)$ for each ephemeral memory modification.