Fully Persistent Arrays

Anders Kaseorg

andersk@mit.edu

6.851 Project Presentation
**Fully Persistent Array**

- \textit{create}(n) \rightarrow v
  Create an array of size \( n \) and returns its initial version identifier \( v \).

- \textit{lookup}(v, i) \rightarrow x
  Returns the item \( x \) stored at index \( i \) in version \( v \) of the array.

- \textit{update}(v, i, x') \rightarrow v'
  Creates a new version \( v' \) from version \( v \) by replacing the item at index \( i \) with \( x' \).
Possible Implementations

Let $m =$ number of updates.

- **Copying arrays**: each version is a separate copy of the array. $O(1)$ lookup, $O(n)$ update, $O(mn)$ space.

- **Trees (branching factor $b$)**: $O(\log_b n)$ lookup, $O(b \log_b n)$ update, $O(mb \log_b n)$ space.

- **Dietz, 1989**: $O(\lg \lg m)$ lookup, expected amortized $O(\lg \lg m)$ update, $O(m)$ space.
Idea (Dietz, 1989)

- Take an Euler tour of the version tree.
  - Insert new versions after their parent, then insert another copy of the parent (with the update reversed) after that.
- Identify a version by a reference to its tag in an order-query structure on this tour.
  - Tag length $O(\lg m)$.
  - Amortized $O(1)$ insert and delete.
This reduces lookups to the predecessor problem. We’re good at that now!

- Store updated items in a y-fast tree keyed by index–version pairs. $O(\log \log m)$ lookup.
- When the order-query structure needs to change a version’s tag, remove it and reinsert it into the y-fast tree.
Recall that the order-query structure is built with **data structure duct tape**—versions are clustered into blocks of size $\Theta(\lg m)$.

- Blocks given $(2 \lg m)$-bit tags.
- Versions are given $(\lg m)$-bit tags within their block.
Problem

- Although an insert takes amortized $O(1)$ time, it may implicitly change the tags of amortized $O(\lg m)$ versions by splitting or relabeling their block.

- So, array updates may cause $O(\lg m)$ deletions and insertions in the $y$-fast tree, requiring $O(\lg m \lg \lg m)$ time.
Recall that the y-fast tree is also built with data structure duct tape—updates are clustered into blocks of size $\Theta(\lg m)$.

- A representative of each block is stored in the prefix hash table structure, pointing to the block.
- Blocks are stored as little BSTs of height $O(\lg \lg m)$. 
Fix

- Within each update block, keys are stored purely based on order information.
- When version tags are changed, the order remains the same.
- So we only need to do extra work if a changed tag belongs to one of the representatives.
- All we need to do is prevent this from happening too often.
Fix

- h4xx0r the order-query structure so that the version of any representative is put into its own block.
  - One dumb way to do this is to insert $\lg n$ virtual versions after each of them.
  - The amortized cost of awarding such special treatment to a version is $O(\lg m)$. But that’s okay because we’re already paying $O(\lg m)$ to deal with it in the y-fast tree.
• Now an insert in the order-query structure can only affect the tags of amortized $O\left(\frac{1}{\lg m}\right)$ representatives.

• Therefore, the changed tags only cause us to do amortized $O(1)$ extra work per update, and the overall cost of update is amortized $O(\lg \lg m)$. 
• So, by tying together two pieces of duct tape, we get a fully persistent array with $O(\lg \lg m)$ time operations.
• Questions?