Prof. Erik Demaine  
TAs: Tom Morgan & Justin Zhang

**Topics:**  
- time travel: remembering/changing the past [THIS WEEK]  
- geometry: >1 dimension (maps, DB tables)  
- dynamic optimality: is there one best BST?  
- memory hierarchy: minimize cache misses  
- hashing: most used DS in CS  
- integers: beat $\lg n$ time/op, or prove impossible  
- dynamic graphs: changing computer/social network  
- strings: search for phrase in text (DNA, web)  
- succinct: reduce space to $\approx$ bare minimum

**Administration:**  
- video recording of lectures  
- requirements: attending lecture, $\approx$ weekly psets, scribing, project  
- signup sheet  
- listeners welcome  
- problem session (starting $\approx$ week 3)  
[ - scribe for today ]
Theme in this class: THE MODEL MATTERS

**Pointer machine:** model of computation

- **$O(1)$ fields**
  - 7
  - 42
- **root**
- **node/record/struct**
- **pointer**

- **field** = data item or pointer to node
- **operations**: $O(1)$ time each
  - $x = \text{new node}$
  - $x = y.\text{field}$
  - $x.\text{field} = y$
  - $x = y + z$ etc. (data operations)
  - [destroy $x$ (if no pointers to it)]

where $x, y, z$ are fields of root (or root)

$\Rightarrow$ constant working space

e.g. linked list, binary search tree (BST), most object-oriented programs
Temporal data structures:
- persistence
- retroactivity

Persistence:
- keep all versions of DS
- DS operations relative to specified version
- update creates (& returns) new version (never modify a version)
- 4 levels:
  ① partial persistence:
    - update only latest version
    ⇒ versions linearly ordered
  ② full persistence:
    - update any version
    ⇒ versions form a tree
  ③ confluent persistence:
    - can combine >1 given version into new V.
    ⇒ versions form a DAG
  ④ functional:
    - never modify nodes; only create new
    - version of DS represented by pointer
**Partial persistence**: [Driscoll, Sarnak, Sleator, Tarjan - JCSS 1989]

Any pointer-machine DS with \( \leq p = O(1) \) pointers to any node (in any version) can be made partially persistent with \( O(1) \) amortized multiplicative overhead \& \( O(1) \) space per change.

**Proof:**
- Store **reverse pointers** for nodes in latest version (only).
- Allow \( \leq 2p \) (version, field, value) mods. in a node (using that \( p = O(1) \)).
- To read node.field at version \( v \), check for mods with time \( \leq v \).
- When update changes node.field = \( x \):
  - If node not full: add mod. (now.field.x)
  - Else: create node' = node with mods applied.
- Change back pointers to node -> node'.
  - Found by following pointers.
  - Recursively change pointers to node' found via back pointers.

  \( - \text{add back pointer from } x \text{ to node}. \)

  \( - \text{potential } \Phi = c \cdot \sum \# \text{mods. in nodes in latest version} \)

  \( \Rightarrow \text{amortized cost } \leq c + c - 2cp + p \text{ recursions} \)

  Compute \( \text{mod.} \text{ if recurse} \leq 2c. \)
**Full persistence:** ditto

- Linearize tree of versions via Euler tour, marking begin & end of each subtree

- Parent sequence = linearized times
  - Time \( i \) makes change \( i \)
  - Time \( i \) unmakes change \( i \)

\[ \Rightarrow \text{version } i \text{ accumulates changes at times } t_i \]

- Store times in order-maintenance DS:
  
  - Insert item before/after specified item (like linked list)
  
- Relative order of 2 items? \(<\) or \(>\)
in \(O(1)\) time/op.

- Update to version \( i \) represented by 2 mods
  - at times \( t_i \) & \( t^{\ast} \)
  - Inserted after \( t_i \) \(\Rightarrow\) undo update or before \( t^{\ast} \)

- Allow \( \leq 2(d+p+1) \) mods. per node

- Forward & backward pointers now symmetric

- When node overflows \(\Rightarrow 2(d+p+1) + 1 \) mods.

  - Split into 2 nodes, each \( \approx \) half full
    - Find median time \( t^{\ast} \)
      - In sorted order of mod. times

  - Old node keeps mods. at \(< t^{\ast} \)
  - New node keeps mods. at \(> t^{\ast} \)

  & base version = old base + all mods. at \( \leq t^{\ast} \)

- Multiple mods. at same time

  \[ \begin{align*}
  t^{\ast} & \in \{ \text{excluded from each } \leq d+p+1 \text{ mod}. \} \\
  & \Rightarrow \text{each } \leq d+p+1 \text{ mod. including } t^{\ast}
  \end{align*} \]
- Use mods. to update reverses of $\leq d+p$ pointers in base version of node' (node ⇒ node')
- Directly update reverses of pointers in mods. of node' (their time > t' ⇒ don't care about node)

- Potential $\Phi = c \cdot \sum_{\text{node}} \# \text{mods. in second half of node}$
  i.e. don't count first $d+p+1$ mods.
- Update creates 2 mods. ⇒ $\Phi / 2c$
  ⇒ $O(1)$ amortized cost ignoring splits
- Split: $\Phi \setminus c(d+p+1)$ in node (empty second half)
  & $\Phi \uparrow c(d+p)$ to update reverses of base version of node'
  ⇒ net $\Phi \setminus c$
- Set $c$ to the work of one split (excluding any recursive splits)
  ⇒ amortized cost = 0
De-amortization:
- partial: $O(1)$ worst case [Brodal–NJC 1996]
- full: OPEN: $O(1)$ worst case?
Confluent persistence:
- after \( u \) confluent updates, can get size \( 2^u \)
- general transformation: [Fiat & Kaplan - J.Alg.2003]
  - \( d(v) \) = depth of version \( v \) in version DAG
  - \( e(v) = 1 + \log(\# \text{ paths from root to } v) \)
  - overhead: \( \log(\# \text{ updates}) + \max_v e(v) \) time & space
    can be up to \( u \)... 
  - still exponentially better than complete copy...
  - lower bound: \( \sum e(v) \) bits of space [Fiat & Kaplan]
  - \( \Omega(e(v)) \) for update if queries are free
  - construction makes \( \approx e(v) \) queries per update
  - \( O(\log^3 \max e(v)) \) update, \( O(\log^2 \max e(v)) \) query [Fiat & Kaplan]
  - OPEN: \( O(1) \) or even \( O(\log n) \) overhead per op.?
- disjoint transformation: [Collette, Tacono, Langerman - SODA 2012]
  - assume confluent ops. performed only on
    versions with no shared nodes
  - then \( O(\log n) \) overhead possible

Idea: each node in subtree of version DAG
- only some of those versions modify node
- 3 types of versions:
  - node modified ~ easy
  - along path between mods.
  - below a leaf ~ hard
  - fractional cascading [L3] & link-cut trees [L19]
Functional: [Okasaki - book 2003]
  - simple example: balanced BSTs
    - work top-down ⇒ no parent pointers
    - duplicate all changed nodes & ancestors before changing
    ⇒ \(O(lg n)\) / op.
  ⇒ link-cut trees too [Demaine, Langerman, Price]

  - e.g. deque with concat. in \(O(1)\) / op.
    double-ended queues [Kaplan, Okasaki, Tarjan - sicomp]
    + update & search in \(O(lg n)\) / op.
      [Brodal, Makris, Tsichlas - ESA 2006]
  - tries with local navigation & subtree copy/delete
    & \(O(1)\) fingers maintained to present
    [Demaine, Langerman, Price - Algorithmica 2010]

```
<table>
<thead>
<tr>
<th>method</th>
<th>time</th>
<th>space</th>
<th>modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>path copying</td>
<td>(lg \Delta)</td>
<td>(\emptyset)</td>
<td>depth</td>
</tr>
<tr>
<td>1. functional</td>
<td>(lg \Delta)</td>
<td>(lg \Delta)</td>
<td>(lg \Delta)</td>
</tr>
<tr>
<td>1. confluent</td>
<td>(lg lg \Delta)</td>
<td>(lg lg \Delta)</td>
<td>(lg lg \Delta)</td>
</tr>
<tr>
<td>2. functional</td>
<td>(lg \Delta)</td>
<td>(\emptyset)</td>
<td>(lg n)</td>
</tr>
<tr>
<td>2. confluent</td>
<td>(lg lg \Delta)</td>
<td>(\emptyset)</td>
<td>(lg n)</td>
</tr>
</tbody>
</table>
Beyond:

- functional: $\geq$ log separation from pointer machine [Pippinger - TPLS 1997]
- OPEN: lists with split & concatenate?

\textcolor{red}{\textbf{Solved}} by Blame Trees in 6.851 Spring'12
[Demaine, Panchekha, Wilson, Yang - WADS 2013]