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 ~ bare minimum

Administration:
- Video recording of lectures
- Requirements: attending lecture, ~weekly psets, scribing, project
- Signup sheet
- Listeners welcome
- Problem session (starting ~ week 3)
[- Scribe for today]
Theme in this class: **THE MODEL MATTERS**

**Pointer machine:** model of computation

- **Fields**
  - \( O(1) \)
  - field = data item OR pointer to node

- **Operations:** \( O(1) \) time each
  - \( x = \text{new node} \)
  - \( x = y.field \)
  - \( x.field = y \)
  - \( x = y + z \) 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 \([L_1]\)
- retroactivity \([L_2]\)

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
    empty mods. ↑ 2 now old
  - change back pointers to node \( \rightarrow \) node’
    found by following pointers
  - recursively change pointers to node \( \rightarrow \)’
    found via back pointers
  - add back pointer from \( x \) to node

- potential \( \Phi = c \cdot \sum \# \text{mods. in nodes in latest version} \)
  \( \Rightarrow \) amortized cost \( \leq c + c - 2cp + p \) recursions

compute mod. if recurse
\( \leq 2c \). \( \Box \)
Full persistence: ditto [Driscoll et al. 1989]
- linearize tree of versions via in-order traversal begin & end of subtree
- store sequence of b's & e's in order-maintenance DS:
  [L8: Dietz & Sleator - STOC 1987]
  - insert item before/after specified item (like linked list)
  - relative order of 2 items in O(1) time/op.
- version v ancestor of w \( \Rightarrow b_v < b_w < e_w < e_v \) \( \Rightarrow O(1) \) time/op.
- can tell which mods apply to specified version
- create child version of v via 2 inserts after b_v
- allow \( \leq 2(d+p+1) \) mods. per node (a \rightarrow 3)
- when changed node is full:
  - split into two nodes, each half full by making copy with half mods. applied, half left
  - recursively update pointers & back pointers to copy
- potential \( \Phi = -c \sum \# \text{empty mod. slots} \) (all nodes live)
  \( \Rightarrow \) charge \( \leq d + p + (d+p+1) \) recursions to \( \Phi \ \cap \ c \ \alpha \ (d+p+1) \) from rest from mods. \( \Rightarrow O(1) \) amortized
- actually splitting mod. version tree \( \frac{1}{3}; \frac{2}{3} \) \( \Delta \) new \( \triangleleft \) old

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: \([\text{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: \(\Omega(e(v))\) bits of space \([\text{Fiat & Kaplan}]
    \Rightarrow \Omega(e(v))\) 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 \([\text{Fiat & Kaplan}]
  - \(\text{OPEN}\): \(O(1)\) or even \(O(\log n)\) overhead per op.?

- disjoint transformation: \([\text{Collette, Tacono, Langeman - 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: \[\text{Okasaki - book 2003}\]
- simple example: balanced BSTs
  - work top-down \(\Rightarrow\) no parent pointers
  - duplicate all \underline{changed nodes & ancestors}
    before changing \(\Rightarrow O(lg n)\)/op.
- \(\Rightarrow\) link-cut trees too \[\text{Demaine, Langerman, Price}\]
- e.g. deque with concat. in \(O(1)\)/op.
  double-ended queues \[\text{Kaplan, Okasaki, Tarjan - sicomp}\]
  \(\Rightarrow\) update \& search in \(O(lg n)\)/op.
  \[\text{Brodal, Makris, Tsichlas - ESA 2006}\]
- \(\Rightarrow\) tries with local navigation \& subtree copy/delete
  \& \(O(1)\) fingers maintained to present.
  \[\text{Demaine, Langerman, Price - Algorithmica 2010}\]

Think: Subversion

<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 (lg \Delta)</td>
</tr>
<tr>
<td>1. functional</td>
<td>(lg \Delta)</td>
<td>(lg \Delta)</td>
<td>(lg \Delta) ((time \neq space))</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 \Delta) (\leq local) mods. (\leq locally ) cheap</td>
</tr>
<tr>
<td>2. confluent</td>
<td>(lg lg \Delta)</td>
<td>(\emptyset)</td>
<td>(lg n) (\leq globally) balanced</td>
</tr>
</tbody>
</table>
Beyond:

- **Functional**: \( \geq \log \) separation from pointer machine \[ \text{Pippinger - TPLS 1997} \]
- **OPEN**: bigger separation? \( \geq \) functional general transformations? \( \geq \) confluent

- **OPEN**: lists with split & concatenate?

\( \rightarrow \text{SOLVED} \) by Blame Trees in 6.851 Spring'12
\[ \text{Demaine, Panchekha, Wilson, Yang - WADS 2013} \]

- **OPEN**: arrays with copy & paste?