Topics:
- Time travel: remembering/changing the past
- 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: \( \text{beat } \log n \text{ time/op. or prove impossible} \)
- 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

0(1) fields

- field = data item or pointer to node
- operations: O(1) time each
  - x = 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)
⇒ constant working space

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

Persistenace:
- 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

think: time travel
branching-universe model

most of Terminator/Sarah Conner Chron.
movie Déjà Vu
part 1
Déjà Vu
part 2
Pullman's book Subtle Knife?
TV show Sliders
movie Primer?
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→node'
  - found by following pointers
- recursively change pointers to node→'
  - found via back pointers
  - add back pointer from \( x \) to node
- potential \( \Phi = c \Sigma_i \# \text{mods. in nodes in latest version} \) \( \Rightarrow \) amortized cost \( \leq c + c - 2cp + p \text{ recursions} \)
  - compute \( \mod. \) if recurse
  - \( \leq 2c \). □
Full persistence: ditto
- linearize tree of versions via in-order traversal, marking 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 \iff b_v < b_w < e_w < e_v
    \iff O(1) time/op.
  \iff 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
  - 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 \Xi, # empty mod. slots (all nodes live)
    \Rightarrow charge \leq d+p + (d+p+1) recursions to \Phi \setminus c 2(d+p+1)
    from rest from mods. \Rightarrow O(1) amortized
    - actually splitting mod. version tree \frac{1}{3} to \frac{2}{3}

De-amortization: (see L10)
- 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
  - \( e(v) = 1 + \lg (\# \text{ paths from root to } v) \)
  - overhead: \( \lg (\# \text{ updates}) + \max_v e(v) \) time & space
    can be up to \( u \)...
  - still exponentially better than complete copy...
  - lower bound: \( \geq e(v) \) bits of space [Fiat & Kaplan]
    \( \Rightarrow \Omega(e(v)) \) for update if queries are free
  - construction makes \( \approx e(v) \) queries per update
  - OPEN: \( O(1) \) or even \( O(\lg n) \) overhead per op.?

- disjoint transformation: [Collette, Iacono, Langerman - SODA]
  - assume confluent ops. performed only on versions with no shared nodes
  - then \( O(\lg n) \) overhead possible

Idea: each node in subtree of version DAG
- only some of those versions modify node
- 3 types of versions:
  - node modified \( \sim \) easy
  - along path between mods.
  - below a leaf \( \sim \) 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 ⇒ \( O(\log n) \)/op.
  - Link-cut trees too [Demaine, Langerman, Price]

- E.g. Deques with concat. in \( O(1) \)/op.
  - Double-ended queues [Kaplan, Okasaki, Tarjan - sicomp]
  - Update & search in \( O(\log n) \)/op.
  - [Brodal, Makris, Tsichlas - ESA 2006]

- Tries with local navigation & subtree copy/delete & \( O(1) \) fingers maintained to present;
  - [Demaine, Langerman, Price - Algorithmica 2010]

**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>( \varnothing )</td>
<td>D ∧ ( 3 ) local mods. cheap</td>
</tr>
<tr>
<td>1. Functional</td>
<td>( \lg \Delta )</td>
<td>( \lg \Delta )</td>
<td>D ∧ ( 3 ) globally balanced</td>
</tr>
<tr>
<td>1. Confluent</td>
<td>( \lg \lg \Delta )</td>
<td>( \lg \lg \Delta )</td>
<td>D ∧ ( 3 ) globally balanced</td>
</tr>
<tr>
<td>2. Functional</td>
<td>( \lg \Delta )</td>
<td>( \varnothing )</td>
<td>( \lg n ) ( 3 ) globally balanced</td>
</tr>
<tr>
<td>2. Confluent</td>
<td>( \lg \lg \Delta )</td>
<td>( \varnothing )</td>
<td>( \lg n ) ( 3 ) globally balanced</td>
</tr>
</tbody>
</table>
Beyond:

- **functional**: log separation from pointer machine [Pippinger - TPLS 1997]

- **OPEN**: bigger separation? general transformations? functional & confluent

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

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