Today: temporal data structures
- persistence
- retroactivity

Persistence:
- keep all versions of DS
- operations specify which version
- update creates 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 node ptr.
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/change)

Proof:
- store reverse pointers for nodes in latest version
- allow $\leq p+1$ (version, field, value) mods. in a node (using $p = O(1)$)
- to read node.field at version $v$, check for mods with time $\leq v$
- when update changes node.field $\leq x$
  - if node not full: add mod. (now.field, x)
  - else: create node with mods. applied ($&$ no mods.)
  update back pointers to this node (found via pointers)
  recursively change pointers to this node (found via back ptrs)
- $\Omega = \# full (p mods.) nodes in latest version
\Rightarrow O(p) = O(1)$ amortized cost per change

$O(1)$ worst-case [Brodal - NJC 1996] $O(\log (# changes))$ reading slowdown for $p$ unbounded
Full persistence: ditto [Driscoll et al. 1989]
- linearize tree of versions via in-order traversal, marking begin & end times of each version
- store begin & end times in order-maintenance DS: [L21: Dietz & Sleator - STOC 1987]
  - insert time before/after specified time
  - does time \( t \) precede time \( t' \)?
  - \( \Rightarrow \) is version \( V \) an ancestor of \( V' \)?
  - in \( O(1) \) time/op.
- \( \Rightarrow \) can tell which mods. apply to desired version
- when node is full, split into two nodes each roughly half full: (like B-tree node)
  then recursively update pointers & back pointers to this node
- allow up to \( 2(p+c+1) \) mods. \( (c=\#ptrs./node) \)
- \( \Rightarrow \) even if half full \( (p+c+1) \) & all \( c \) ptrs. move & all \( p \) back ptrs. move, still not full
- \( \Phi = \# \) full nodes \( \Rightarrow O(1) \) amortized cost

\[ \text{OPEN: } O(1) \text{ worst case?} \]
- \( O(lg \ lg \ n) \) fully persistent arrays \( \Rightarrow \) RAM DS [Dietz - WADS 1989]
  - matching lower bound [Patrascu - unpub. 2008]
  - \[ \text{OPEN: } \text{what about partial persistence?} \]
Confluent persistence:
- functional DSs [Okasaki - book 2003]
  e.g. deques with concat. in O(1)/op.
  double-ended queues [Kaplan, Okasaki, Tarjan - SICOMP 2000]
- logarithmic separation from pointer-machine DS [Pippenger - TPLS 1997]
- general transformation: [Fiat & Kaplan - J.Alg. 2003]
  \( d(v) = \text{depth of version } v \text{ in version } \text{DAG} \)
  \( e(v) = 1 + \log(\# \text{ paths from root to } v) \)
- overhead: \( \log(\#\text{updates}) + \max_v e(v) \) time\&space
- poor when \( e(v) \sim 2^{\#\text{updates}} \) e.g.:
  - can make exponential-size version in this way
  \Rightarrow \text{still exponentially better than naive}
  - \( \max_v e(v) \) lower bound \text{ASSUMING} all nodes addressable at all times \text{UNREASONABLE: normally have to navigate to nodes}
- tries with \( O(1) \) fingers,
  local nav. & subtree copy/delete [Demaine, Langerman, Price - Algorithmica]

<table>
<thead>
<tr>
<th>method</th>
<th>finger move time</th>
<th>space</th>
<th>modification (time = space)</th>
</tr>
</thead>
<tbody>
<tr>
<td>path copying</td>
<td>( \log \Delta )</td>
<td>( \emptyset )</td>
<td>depth</td>
</tr>
<tr>
<td>1. functional</td>
<td>( \log \Delta )</td>
<td>( \log \Delta )</td>
<td>( \log \Delta ) local mods. cheap</td>
</tr>
<tr>
<td>1. confluent</td>
<td>( \log \log \Delta )</td>
<td>( \log \log \Delta )</td>
<td>( \log \log \Delta ) globally balanced</td>
</tr>
<tr>
<td>2. functional</td>
<td>( \log \Delta )</td>
<td>( \emptyset )</td>
<td>( \log n )</td>
</tr>
<tr>
<td>2. confluent</td>
<td>( \log \log \Delta )</td>
<td>( \emptyset )</td>
<td>( \log n )</td>
</tr>
</tbody>
</table>
OPEN: better transformation with $O(1)$ fingers? maintained to present separations?
OPEN: functional transformations?
OPEN: lists with split & concatenate?
OPEN: arrays with e.g. copy & paste?
Retroactivity: [Demaine, Iacono, Langerman - T Alg. 2007]
- traditional DS formed by sequence of updates
- allow changes to that sequence
- maintain linear timeline

ops:
- Insert(t, "op(...)"'): retroactively do op() at time t
- Delete(t): retroactively undo op. at time t
- Query(t, "op()"'): execute query at time t

- partial retroactivity: Query only in present (last t)
- full retroactivity: Query at any time

Easy cases:
- commutative updates: x, y \equiv y \cdot x
  \Rightarrow \text{Insert}(t, x) \equiv x \text{ in present}
- invertible updates: x \cdot x^{-1} \equiv \emptyset
  \Rightarrow \text{Delete}(t) \equiv x^{-1} \text{ in present}
- e.g. hashing, or array with A[i] += \Delta
  \Rightarrow \text{partial retroactivity easy}
- \underline{search problem}: maintain set S of objects subject to
query(x, S) for object x \in S \text{ comm. & invertible}
- \underline{decomposable search problem}: [Bentley & Saxe - T Alg. 1980]
query(x, A \cup B) = f(query(x, A), query(x, B))
- e.g. nearest neighbor, successor, point location
- full retroactivity in O(lg n) overhead via segment tree
General transformations: [Demaine et al. 2003]

- rollback method: retro op. r time units in past with factor-r overhead via logging ("undo persistence")
- lower bound: \( \Omega(r) \) overhead can be necessary
- DS maintains two values \( X & Y \) initially \( \emptyset \)
  - \( setX(x) \): \( X \leftarrow x \)
  - \( addY(A) \): \( Y \leftarrow Y + A \)
  - \( mulXY() \): \( Y \leftarrow X \cdot Y \)
  - \( query() \): return \( Y \)
- \( O(1) \) time/op. in "straight-line program" model
- \( addY(a_n), mulXY(), addY(a_{n-1}), mulXY(), ..., addY(a_0) \) computes poly. \( a_nx^n + a_{n-1}x^{n-1} + ... + a_0 \) [Cramer's rule]
- \( Insert(t=0, "setX(x)") \) changes \( x \) value
- evaluating degree-\( n \) polynomial requires \( \Omega(n) \) worst-case arithmetic ops. in any field

independent of \( a_i \), preprocessing in "history-independent algebraic decision tree"

\( \Rightarrow integer \) RAM \( \Rightarrow generalized \) real RAM
[Frandsena, Hansenb, Miltersen – I&C 2001]

- cell-probe lower bound: \( \Omega(\sqrt[3]{lg r}) \)
- DS maintains \( n \) words; arithmetic updates +,
- compute FFT using \( O(n \lg n) \) ops.
- changing \( w_i \) requires \( \Omega(\sqrt{n}) \) cell probes
[Frandsena et al. 2001]

- \( \boxed{OPEN}: \Omega(\sqrt[3]{r/poly \lg r}) \) cell-probe lower bound?
Priority queues: [Demaine et al. 2003]

- partial retroactivity in $O(\log n)/\text{op.}$
- assume keys inserted only once
- $L$-view: insert = rightward ray
  delete-min = upward ray

- Insert \((t, \text{"insert}(k)\text{"})\) inserts into \(Q_{\text{now}}\)
  \(\max \{ k : k \in Q_{\text{now}} \} \) \(k\) deleted at time \(\geq t \frac{2}{3}\) hard to maintain

- bridge at time \(t\) if \(Q_t \subseteq Q_{\text{now}}\)
- if \(t'\) is the bridge preceding time \(t\)
  then \(\max \{ k' : k' \text{ deleted at time } \geq t \frac{2}{3}\}\)
  \(= \max \{ k' : k' \in Q_{\text{now}} \text{ inserted at time } \geq t' \}\)
- store $Q_{\text{now}}$ as balanced BST; one change/update
- store balanced BST on leaves = insertions, ordered by time, augmented with
  $\forall$ node $x$: $\max \{ k' \mid Q_{\text{now}} \downarrow k' \text{ inserted in } x \text{'s subtree}\}$
- store balanced BST on leaves = updates, ordered by time, augmented with
  0 for $\text{insert}(k)$ with $k \in Q_{\text{now}}$
  +1 for $\text{insert}(k)$ with $k \notin Q_{\text{now}}$
  -1 for $\text{delete-min}$
  & subtree sums
  $\Rightarrow$ bridge $= \text{prefix summing to } \emptyset$
  $\Rightarrow$ can find preceding bridge, change to $Q_{\text{now}}$ in $O(\log n)$ time

**Other structures:**
- queue: $O(1)$ partial, $O(\log m)$ full
- deque: $O(\log n)$ full
- union-find (incremental connectivity): $O(\log n)$ full
- priority queue: $O(\sqrt{m} \log m)$ full
  (via general partial $\rightarrow$ full transform, $\times O(\sqrt{m})$)
- successor: $O(\log m)$ partial, trivial
  $O(\log^2 m)$ full easy
  $O(\log m)$ full [Giora & Kaplan - 2009]

($\Rightarrow$ optimal dynamic vertical
ray shooting among horizontal line segments)

**OPEN:** better? general?