Today: temporal data structures II
- partial retroactivity
- full retroactivity
- nonoblvious retroactivity

Think: time travel

Retroactivity: [Demaine, Iacono, Langerman - T. Alg. 2007]
- traditional DS formed by sequence of updates
- allow changes to that sequence (destroying old ver.)
- maintain linear timeline

\[ t: \emptyset \rightarrow 1 \rightarrow 2 \rightarrow \ldots \rightarrow \text{now} \]

- ops:
  - Insert\( (t, \text{"op(--)"}) \): retroactively do op() at time \( t \)
  - Delete\( (t) \): retroactively undo op. at time \( t \)
  - Query\( (t, \text{"op()"}) \): execute query at time \( t \)
    (relative to current timeline only)

- time specified as index, or via order-maintenance DS
- partial retroactivity: Query only in present \( \leq t \)
- full retroactivity: Query at any time

Dr. Who, Timecop, Back to the Future

in Star Trek
Easy case:
- commutative updates: \( x, y \equiv y, x \)
  \( \Rightarrow \) Insert \((t, x)\) \( \equiv \) \( x \) in present
- invertible updates: \( x \cdot x^{-1} \equiv \emptyset \)
  \( \Rightarrow \) Delete \((t)\) \( \equiv \) \( x^{-1} \) in present
  \( \Rightarrow \) partial retroactivity easy (update in present)

- e.g. hashing, or array with \( A[i] \overset{\Delta}{=} i \)
- e.g. Search problem: maintain set \( S \) of objects subject to \( \text{query}(x, S) \) for object \( x \) & insert/delete objects
- decomposable search problem: [Bentley & Saxe - JACM 1980]
  \( \text{query}(x, A \cup B) = f(\text{query}(x, A), \text{query}(x, B)) \)
- e.g. nearest neighbor, successor, point location
- full retroactivity in \( O(lg n) \) factor overhead via segment tree:

\( \text{time} \)

- time interval maps to \( O(lg n) \) subtree intervals
- Insert/Delete modify element’s existence interval
  \( \Rightarrow O(lg n) \) updates to DSs in nodes
- Query combines \( O(lg n) \) searches via \( f \)
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 \( \emptyset \)
  - ops: \( X = x \), \( Y = \Delta \), \( Y = X \cdot Y \), query: return \( Y \)
  - \( O(1) \) time/op. in "straight-line program" model
  - \( Y^+ = a_n, X = X \cdot Y, Y^+ = a_{n-1}, X = X \cdot Y \ldots, Y^+ = a_0 \)

  computes poly. \( a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 \) [Cramer's rule]

- Insert \( t = \emptyset, "X = x" \) 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
  [Frandsen, Hansenb, Miltersen – I&C 2001]

- cell-probe lower bound: \( \Omega(\sqrt[4]{n^{2r}}) \)
  - DS maintains \( n \) words; arithmetic updates + & -
  - compute FFT using \( O(n \log n) \) ops.
  - changing \( w_i \) requires \( \Omega(n) \) cell probes
  [Frandsen et al. 2001]

- OPEN: \( \Omega(r/poly \log r) \) cell-probe lower bound?
Priority queues: \cite{DemaineIaconoLangerman2003}

- insert & delete-min, partially retroactive in $O(g n / \log n)$
- assume keys inserted only once
- L view: insert = rightward ray
  delete-min = upward ray

also Delete("delete-min")

- Insert($t$, "insert($k$)") inserts into $Q_{now}$
  max $\{k, k'\}$ $k'$ deleted at time $\geq t^3$
  hard to maintain

- bridge at time $t$ if $Q_t \subseteq Q_{now}$
- if $t'$ is the bridge preceding time $t$
  then $\max \{k' \mid k'$ deleted at time $\geq t^3\}$
  = $\max \{k' \in Q_{now} \mid k'$ 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 $x$: $\max \{ k' \in Q_{\text{now}} \mid k'$ inserted in $x$'s subtree $\}$
- store balanced BST on leaves = updates, ordered by time, augmented with
  $\begin{align*}
  0 & \text{ for insert}(k) \text{ with } k \in Q_{\text{now}} \\
  +1 & \text{ for insert}(k) \text{ with } k \notin Q_{\text{now}} \\
  -1 & \text{ for delete-min}
  \end{align*}$

& subtree sums

$\Rightarrow$ bridge = prefix summing to 0

$\Rightarrow$ 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
  $\begin{array}{c}
  \text{(via general partial $\rightarrow$ full transform, } \times O(\sqrt{m})
  \end{array}$
- successor: $O(\log m)$ partial via search
  $O(\log^2 m)$ full via decomposable search
  $O(\log m)$ full \cite{Giora & Kaplan}

\text{uses fractional cascading \cite{L3} \& van Emde Boas \cite{L11}}
Nonoblivious retroactivity: [Acar, Blelloch, Tangwongsan - CMU TR 2007]
- in algorithmic use of DS (e.g. priority queue in Dijkstra) updates performed depend on results of queries
⇒ put queries on timeline too
- retroactive update may change result of future queries
- new retro DS query: time of earliest error
- assume that algorithm corrects errors by further retroactive updates (e.g. Delete & re-Insert query)
in increasing time order always ≤ errors
- idea: just rerunning what’s changed of algorithm

Priority queue: insert, delete, & min in \(O(\log m)\) time/op.

\begin{itemize}
\item invariant: all crossings involve horiz. segments with left endpoint left of all errors
\item maintain lowest leftmost crossing
  \Rightarrow \text{leftmost lowest crossing}
\end{itemize}
- Assume keys inserted only once
- Maintain earliest floating error on each key row
- Maintain priority queue on all errors by time

⇒ Always know earliest error

- \text{Insert}(x, "min"): upward ray shot
  = fully retroactive successor (-∞) \leq O(lg m)
  = fully retroactive insert, delete, min
  (decomposable search problem ~ but then lg m)

- \text{Insert}(x, "insert(y)") / \text{Delete}(x, "delete(y)"):
  rightward ray shot to find earliest crossing
  (if lower than existing lower left crossing)
  = fully retroactive successor(x) \leq O(lg m)
  ... when all inserts are at time -∞

- \text{Insert}(x, "delete(y)") / \text{Delete}(x, "insert(y)"):
  - if was lowest crosser, find next by upward ray shot from leftmost crosser query
  - rightward ray shot to find earliest floater

- \text{Delete}(x, "min"):
  - if floating: rightward ray shot to next in row
  - if leftmost crosser: find next by upward ray shot for next min query (successor among queries)