6.851	Lecture 2	Feb. 9, 2012

TODAY: temporal data structures II

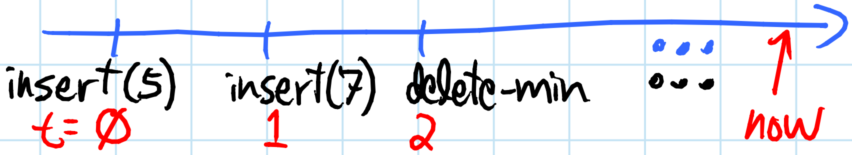
- partial retroactivity
- full retroactivity
- nonoblivious retroactivity

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

plastic timeline model



Dr. Who, Timecop, Back to the Future

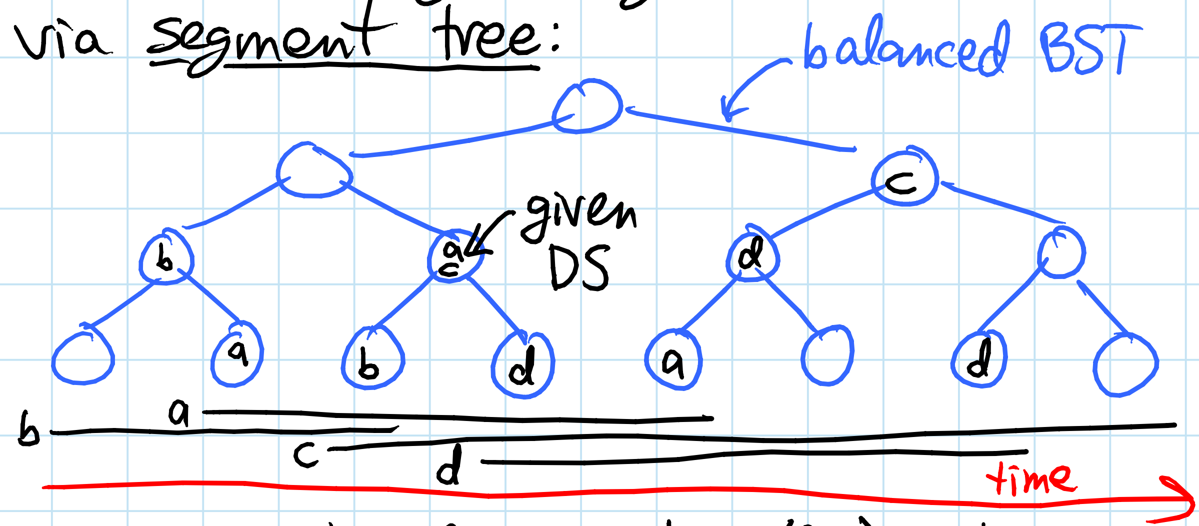
- ops:
  - Insert(t, "op(...)"): retroactively do op() at time t
  - Delete(t): retroactively undo op. at time t
  - Query(t, "op()"): execute query at time t (relative to current timeline only)
- time specified as index, or via order-mainenance DS
- partial retroactivity: Query only in present (last t)
- full retroactivity: Query at any time

"Q" in Star Trek

Easy case:

- commutative updates: $x, y \equiv y, x$
  $\Rightarrow$ Insert$(t, x) \equiv x$ in present
- \+ invertible updates: $x, x^{-1} \equiv \emptyset$
  $\Rightarrow$ Delete$(t) \equiv x^{-1}$ in present
- $\Rightarrow$ partial retroactivity easy (update in present)

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



- time interval maps to $O(\lg m)$ subtree intervals
- Insert/Delete modify element's existence interval
  $\Rightarrow O(\lg m)$ updates to DSs in nodes
- Query combines $O(\lg m)$ searches via f $\square$

union all ancestors of query time

General transformations: [Demaine et al. 2003]

- rollback method: retro. op. r time units in past with factor-r overhead via logging ("undo persistence")
  movie Retroactive

- lower bound: $\Omega(r)$ overhead can be necessary
  - DS maintains two values X & Y, initially $\emptyset$
  - ops: $X = x$, $Y \mathrel{+}= \Delta$, $Y = X \cdot Y$, query: return Y
  - $O(1)$ time/op. in "straight-line program" model
  - $Y \mathrel{+}= a_n$, $Y = X \cdot Y$, $Y \mathrel{+}= a_{n-1}$, $Y = X \cdot Y$, ..., $Y \mathrel{+}= a_0$ computes polyn. $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$ [Horner'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
    [Frandsena, Hansenb, Miltersen — I&C 2001]

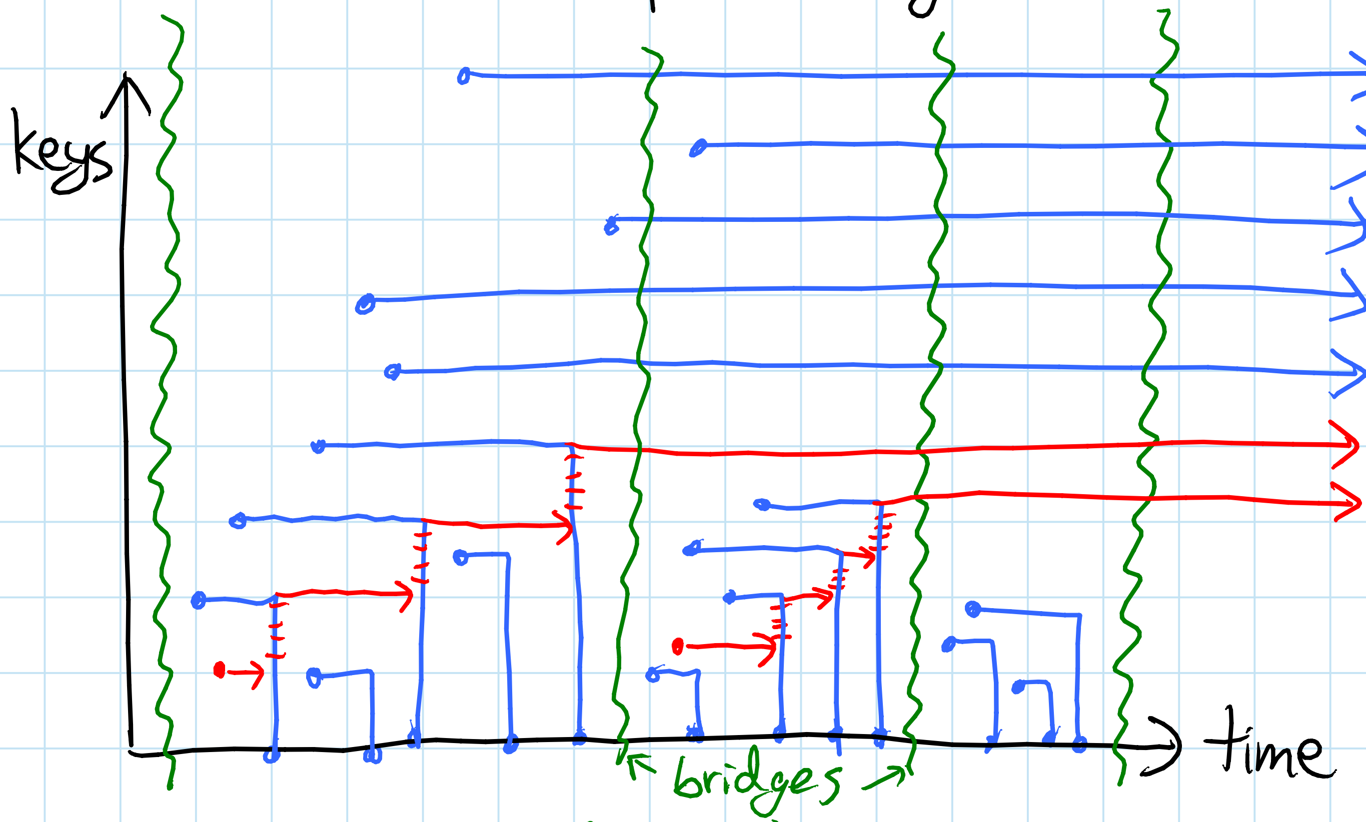
- cell-probe lower bound: $\Omega(\sqrt{r / \lg r})$
  - DS maintains n words; arithmetic updates $+$ & $\cdot$
  - compute FFT using $O(n \lg n)$ ops.
  - changing w_i requires $\Omega(\sqrt{n})$ cell probes
    [Frandsena et al. 2001]

- $\boxed{\text{OPEN}}$: $\Omega(r / \text{poly} \lg r)$ cell-probe lower bound?

Priority queues: [Demaine, Iacono, Langerman 2003]

insert & delete-min, partially retroactive in $O(\lg n)$/op.

- 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} $\max\{k, k' \mid k' \text{ deleted at time} \geq t\}$

hard to maintain

- bridge at time t if $Q_t \subseteq Q_{now}$
- if t' is the bridge preceding time t then $\max\{k' \mid k' \text{ deleted at time} \geq t\}$ $= \max\{k' \notin Q_{now} \mid k' \text{ inserted at time} \geq t'\}$

- store Q_{now} as balanced BST: one change/update
- store balanced BST on leaves = insertions, ordered by time, augmented with $\forall$node x: $\max\{k' \notin Q_{now} \mid k'$ inserted in x's subtree$\}$
- store balanced BST on leaves = updates, ordered by time, augmented with
  - 0 for insert(k) with $k \in Q_{now}$
  - $+1$ for insert(k) with $k \notin Q_{now}$
  - -1 for delete-min

  & subtree sums & subtree min/max prefix sums

$\Rightarrow$ bridge = prefix summing to $\emptyset$

$\Rightarrow$ can find preceding bridge, change to Q_{now} in $O(\lg n)$ time

Other structures:

- queue: $O(1)$ partial, $O(\lg m)$ full
- deque: $O(\lg m)$ full
- union-find (incremental connectivity): $O(\lg m)$ full
- priority queue: $O(\sqrt{m} \lg m)$ full $\boxed{\text{OPEN: better?}}$
  (via general partial→full transform, $\times O(\sqrt{m})$)
- successor: $O(\lg m)$ partial via search
  $O(\lg^2 m)$ full via decomposable search
  $O(\lg m)$ full [Giora & Kaplan – T.Algs. 2009]
  ↳ uses fractional cascading [L3]
  & van Emde Boas [L11]

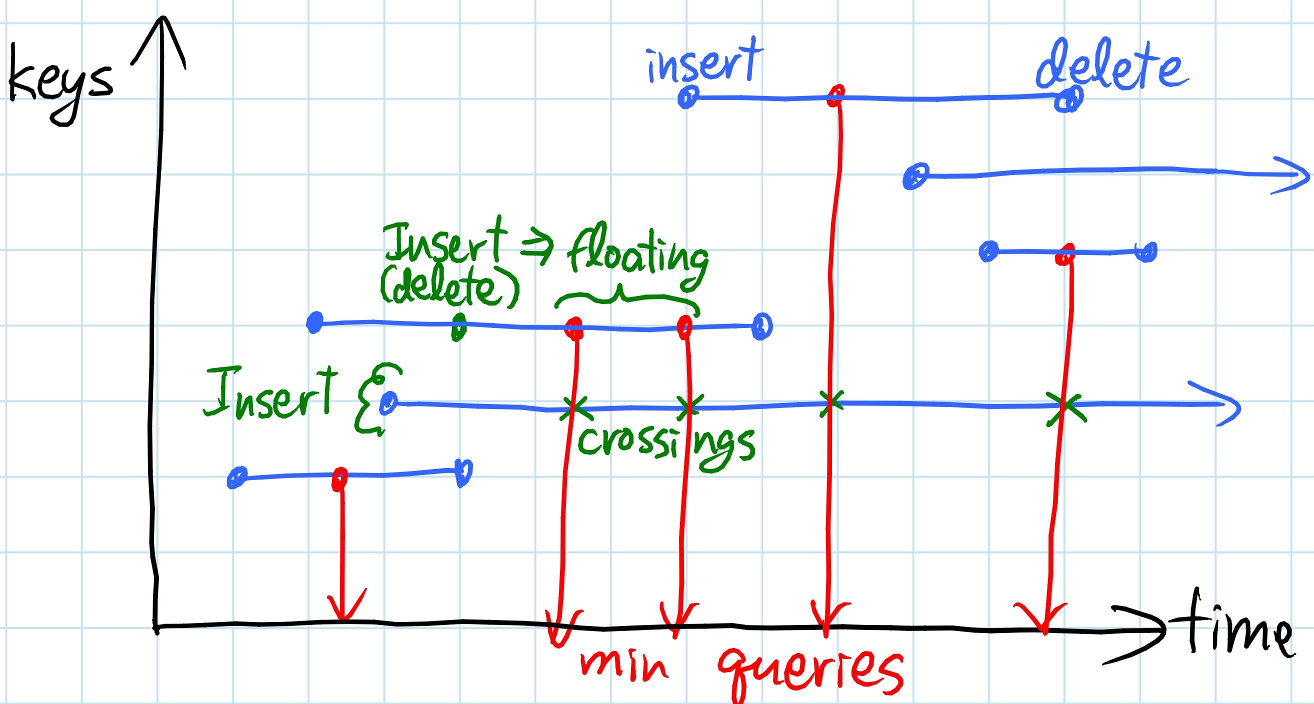
Nonoblivious retroactivity: [Acar, Blelloch, Tangwongsan - CMU TR 2007]

- in algorithmic use of DS (e.g. priority queue in Dijkstra) updates performed depend on results of queries

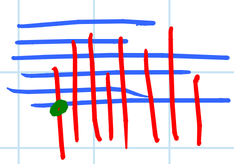
⇒ put queries on timeline too

- 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 $O(\lg m)$ time/op.



- invariant: all crossings involve horiz. segments with left endpoint left of all errors
- maintain lowest leftmost crossing = leftmost lowest crossing



- 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

- Insert(x, "min"): upward ray shot
  = fully retroactive successor($-\infty$) ← $O(\lg m)$
  = fully retroactive insert, delete, min
  (decomposable search problem ~ but then $\lg^2 m$)

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

- Insert(x, "delete(y)") / 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

- 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)