6.851: Advanced Data Structures Spring 2012

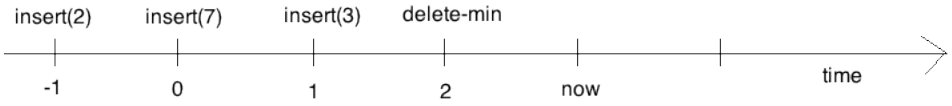
Lecture 2 — 02/09, 2012

Prof. Erik Demaine *Scribe: Erek Speed, Victor Jakubiuk*

1 Overview

The main idea of persistent data structures is that when a change is made in the past, an entirely new universe is obtained. A more science-fiction approach to time travel is that you can make a change in the past and see its results not only in the current state of the data structure, but also all the changes in between the past and now.

We maintain one timeline of updates and queries for a persistent data structure:



Usually, operations are appended at the end of the timeline (present time). With retroactive data structures we can do that in the past too.

2 Retroactivity

The following operations are supported by retroactive DS:

- *Insert(t, update)* - inserts operation "update" at time t
- *Delete(t)* - deletes the operation at time t
- *Query(t, query)* - queries the DS with a "query" at time t

Uppercase Insert indicates an operation on retroactive DS, lowercase update is the operation on the actual DS.

You can think of time t as integers, but a better approach is to use an order-maintanance DS to avoid using non-integers (in case you want to insert an operation between times t and $t+1$), as mentioned in the first lecture.

There are three types of retroactivity:

- *Partial* - Query always done at $t = \infty$ (now)
- *Full* - Query at any time t (possibly in the past)
- *Nonoblivious* - Insert, Delete, Query at any time t, also if an operation modifies DS, we must say which future queries are changed.

2.1 Easy case with commutativity and inversions

Assume the following hold:

- *Commutative updates:* $x.y = y.x$ (x followed by y is the same as y followed by x); that is the updates can be reordered $\Rightarrow$ Insert(t, op) = Insert(now, op).
- *Invertible updates:* There exists an operation x^{-1}, such that $x.x^{-1} = \emptyset \Rightarrow$ Delete(t, op) = Insert(now, op^{-1})

2.1.1 Partial retroactivity

These two assumptions allow us to solve some retroactive problems easily, such as:

- *hashing*
- *array* with operation $A[i] += \Delta$ (but no direct assignment)

2.2 Full retroactivity

First, lets define the **search problem**: maintain set S of objects, subject to insert, delete, query(x, S).

Decomposable search problem [1980, 2007]: same as the search problem, with a restriction that the query must satisfy: query$(x, A \cup B) = f($query(x, A),query$(x, B))$, for some function f computed in $O(1)$ (sets A and B may overlap). Examples of problems with such a function include:

- *Dynamic nearest neighbor*
- *Successor on a line*
- *Point location*

Claim 1. *Full Retroactivity for decomposable search problems (with commutativity and inversions) can be done in* $O(\lg m)$ *factor overhead both in time and space (where* m *is the number of operations) using* **segment tree** *[1980, Bentley and Saxon (??)]*

We want to build a balanced search tree on time (leaves represent time). Every element "lives" in the data structure on the interval of time, corresponding to its insertion and deletion. Each element appears in $\lg n$ nodes.

To query on this tree at time t, we want to know what operations have been done on this tree from the beginning of time to t. Because the query is decomposable, we can look at $\lg n$ different nodes and combine the results (using the function f).

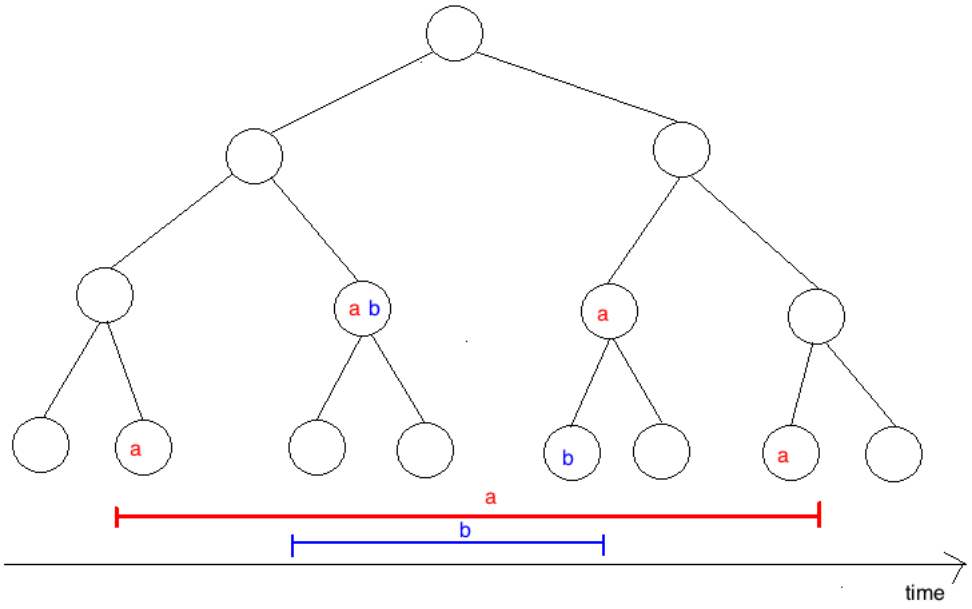


Figure 1: Segment Tree

2.3 General case of full retroactivity

Roll back method:

- write down a (linear) chain of operations and queries
- change r time units in the past with factor $O(r)$ overhead.

That's the best we can do in general.

Lower bound: $\Omega(r)$ overhead necessary.

Proof: Data Structure maintains 2 values (registers): X and Y, initially $\emptyset$. The following operations are supported: $X = x$, $Y+ = \Delta$, $Y = X.Y$, query 'Y?'. Perform the following operations (Cramer's rule):

$$Y+ = a_n, X = X.Y, Y+ = a_{n-1}, X = X.Y, \ldots, Y+ = a_0$$

which is equivalent to computing

$$Y = a_n X^n + a_{n-1} X^{n-1} + ... + a_0$$

Now, execute Insert$(t = 0, X = x)$, which changes where the polynomial is evaluated. This cannot be done faster than re-evaluating the polynomial. In history-independent algebraict decision tree, for any field, independent of pre-processing of the coefficients, need $\Omega(n)$ field operations (result from 2001), where n is the degree of the polynomial.

2.4 Cell-probe problem

How many integers (words) of memory do you need to read to solve your problem? (gives a lower bound on running time).

Claim: $\Omega(\sqrt{\frac{r}{\lg r}})$.

Open problem: $\Omega(r)$.

Proof of the lower bound claim:

- DS maintans n words (integers of $w \geq \lg n$ bits).
- Arithmetic operations are allowed.
- Query = what is the value of an integer?
- Compute FFT.
- Retroactively change the words $\Rightarrow$ Dynamic FFT.
- Changing x_i requires $\Omega(\sqrt{n})$ cell probes.

2.5 Priority Queues

Now, let us move onto some more positive results. Priority queues represent a DS where retroactive operations potentially create chain reactions but we have still obtained some nice results for. The main operations are *insert* and *delete-min* which we would like to retroactively *Insert* and *Delete*.

Claim 2. *It is possible to implement a partially retroactive priority queue with only $O(\lg n)$ overhead per partially retroactive operation.*

Because of the presence of *delete-min*, the set of operations on priority queues is non-commutative. The order of updates now clearly matters, and *Insert*ing a *delete-min* retroactively has the potential to cause a chain reaction which changes everything that comes afterward.

To develop an intuition for how our DS changes given a retroactive operation, it is helpful to plot it on a two dimensional plane. The x-axis represents time and y-axis represents key value. Every $insert(t, k)$ operation creates a horizontal ray that starts at point (t, k) and shoots to the right (See Fig. 2).

Every *delete-min*() operation creates a vertical ray that starts at $(t, -\infty)$ and shoots upwards, stopping at the horizontal ray of the element it deletes. Thus, the horizontal ray becomes a line segment with end points (t, k) and (t_k, k), where t_k is the time of key k's deletion.

This combinations of inserts and deletes creates a graph of nonintersecting upside down "L" shapes, where each L corresponds to an *insert* and the *delete-min*() that deletes it. Elements which are never deleted remain rightward rays. Figure 3 demonstrates this by adding a few *delete-min*s to our previous graph of only *insert*s.

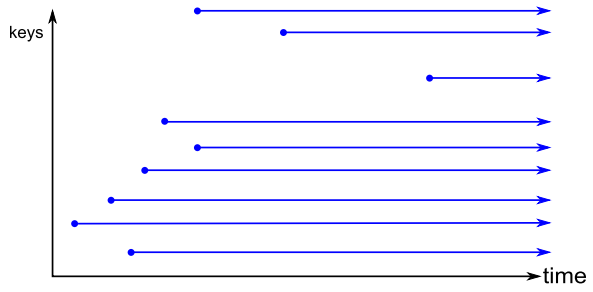


Figure 2: Graph of priority queue featuring only a set of *inserts*

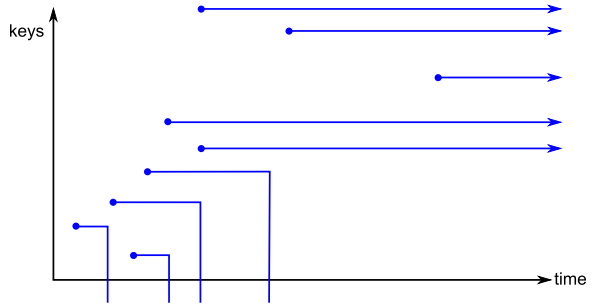


Figure 3: Adding *del-min*() operations leads to these upside down "L" shapes.

The rest of the discussion will focus on $Insert(t, \text{“}insert(k)\text{”})$. It should be easy enough to convince yourself that $Delete(t, \text{“}delete\text{-}min\text{”})$ has equivalent analysis if you think about Delete as inserting the deleted element at the time of deletion.

Consider the priority queue represented by figure 4. It's similar to the previous ones seen but with more inserts and deletes to better illustrate the chain-reactions of retroactive operations. Figure 5 shows what happens when elements are retroactively inserted. The retroactive operations and their chain reactions are shown in red. The cascading changes add up fast.

However, since we're only requiring partial retroactivity we only need to determine what element $Insert(t, \text{“}insert(k)\text{”})$ inserts into Q_{now} where Q_{now} is the priority queue at the present time. Naively, it is easy to see that the element inserted at Q_{now} is: $\max\{k, k' \mid k' \text{ deleted at time} \geq t\}$. That is, the element that makes it to the "end" is the biggest element that was previously deleted (ie. the end of the chain-reaction shown in Figure 3) or simply k if it is bigger than those (ie. the insert caused no chain reactions).

Problem: Maintaining "deleted" elements is hard. It requires us to maintain the various chain-reactions which isn't efficient. Instead, we would like to simply keep track of inserts. Such a transformation is possible as long as we define the new concept of "bridges".

Definition 3. *We define time t to be a bridge if $Q_t \subseteq Q_{now}$.*

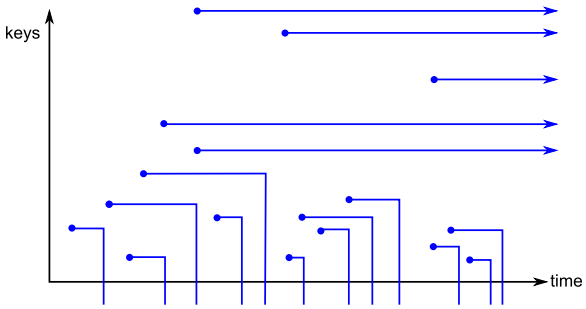


Figure 4: An L view representation of a priority queue with a more complicated set of updates

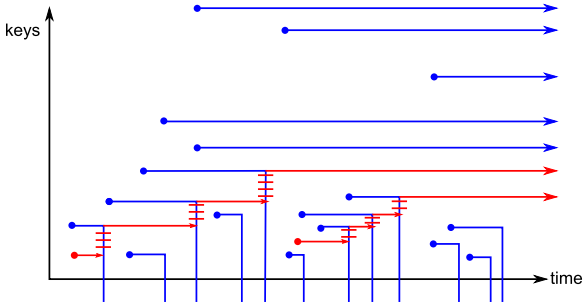


Figure 5: Retroactive inserts start at red dots and cause subsequent *delete-min* operations to effect different elements as shown.

This simply means that all of the elements at a bridge t' are also present at t_{now}. You can think of bridges as separating the chaotic chain-reactions that happen during retroactive $Insert$s as seen in figure 6.

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

With that transformation, we only need to maintain three data structures which will allow us to perform partially retroactive operations with only $O(\lg n)$ overhead.

- We will store Q_{now} as a balanced BST. It will be changed once per update.
- We will store a balanced BST where the leaves equal insertions, ordered by time, and augmented with $\forall$ node $x : \max\{k' \notin Q_{now} \mid k' \text{ inserted in } x\text{'s subtree}\}$.
-

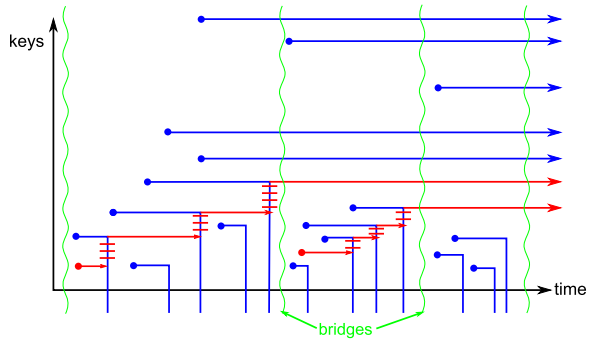


Figure 6: Bridges have been inserted as green wavy lines. Notice how they only cross elements present to the end of visible time.

References

[1] M. Fredman, J. Komlós, E. Szemerédi, *Storing a Sparse Table with $O(1)$ Worst Case Access Time*, Journal of the ACM, 31(3):538-544, 1984.