**6.851: Advanced Data Structures** Spring 2007

# Lecture 7 — March 5, 2007

*Prof. Erik Demaine* *Scribes: Aston Motes and Kevin Wang*

## 1 Overview

In previous lectures, we've discussed data structures that efficiently provided many nice properties. However, we have never discussed any data structures that provide the somewhat magical property of *time travel*.

In this lecture we discuss temporal data structures, which will allow us to view and/or modify our data structure at various points in the past, in addition to its present state. The specific behaviors of these data structures are defined below.

## 2 Temporal Data Structures

There are two primary models of temporal data structures. The first, called persistence, is based on the branching-universe model of time travel. In this model, going back in time and making changes creates a new branch of the data structure that differs from the original branch. The second, called retroactivity, works on the idea of round-trip time travel. Here, a time traveller goes back in time, makes a change, and then returns to observe the effects of his or her change. This model gives us a there is a linear timeline with no branching.

### 2.1 Persistence

We begin by describing the levels of desired persistence. With data structure persistence, we would like to keep all versions of the data structure available for updates and queries. Each persistence level, however will vary based on where updates are allowed and how branches and nodes are modified and created.

1. Partial Persistence – In this persistence model, we may query any previous version of the data structure, but we may only update the latest version. This implies a linear ordering among the versions.
2. Full Persistence – In this model, both updates and queries are allowed on any version of the data structure. The versions here form of a branching tree.
3. Confluent Persistence – In this model, we use combinators to combine input of $> 1$ previous versions to output a new single version. Rather than a branching tree, combinations of versions induce a DAG structure on the version graph.
4. Functional Persistence – This model takes its name from functional programming where objects are immutable. The versions in this model are likewise immutable, so revisions do not alter the existing nodes in the data structure, but create new ones instead. Okasaki discusses these as well as other functional data structures in his book [9].

Each of the succeeding levels of persistence imply the preceding ones. That is, Functional Persistence $\implies$ Confluent Persistence $\implies$ Full Persistence $\implies$ Partial Persistence. Functional implies confluent because we simply use the combinators to append a new combined version. Confluent implies full if we do not use combinators. Lastly, full implies partial if we only update the most recent version.

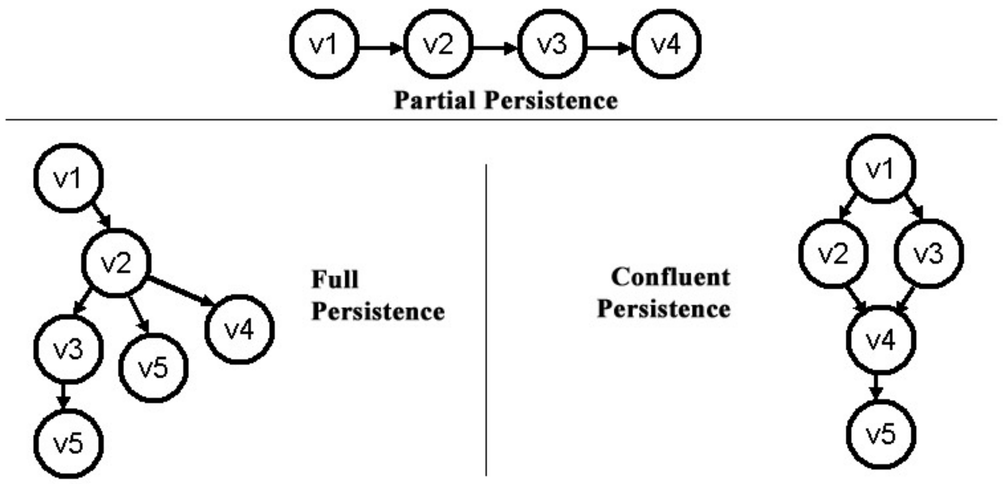

Figure 1: Abstract cartoons of the major persitence types.

### 2.1.1 Partial Persistence

This result is due to Driscoll, Sarnak, Sleator, and Tarjan [6]. We work within the pointer machine model and require $O(1)$ in-degree per node, meaning that $\leq p = O(1)$ nodes point to any node. Each node stores some data and a constant number of pointers to children, reverse pointers to parents, and version modification data (in a "modification box"). To maintain partial persistence, each node stores a reverse pointer to the parent node representing the most recent version of the data structure. A modification can be thought of as the tuple $(time, field, value)$, consisting of the time of the modification, the field being modified, and the new value.

An update on a field at some time $t$ can come across two cases:

- The node has space – We can simply add the modification $(t, field, value)$ to the modification box. All subsequent accesses of this modified node will check the modification box to override any initial data stored in the node.
- The node is full – We make a copy of the node, but using only the latest values. That is, we overwrite one of the nodes fields with the value that was stored in the modification box, and make the modification box of the new node empty. We propagate this change up to node's ancestors as follows: each ancestor makes a modification to change its child pointer to the newly created node. If that ancestor's modification box happens to be full, then we copy that node and propagate up. These changes propagate until we stop at the root.

Using $\Phi =$ the number of full latest nodes, we can prove constant time amortized updates. Further study by Brodal [2] has shown it to also be $O(1)$ in the worst case.

### 2.1.2 Full Persistence

This result is also due to [6]. We again assume a pointer machine with $\leq p$ incoming pointers per node. We can speak of the version list, which is equivalent to a pre-order traversal of the version tree. Deitz and Sleator showed we can insert and compare the order of nodes in this so called list-order data structure (the version list) in $O(1)$ time [4]. We will discuss this data structure in more depth in lecture 19. A related open question is whether we can support $O(1)$ worst-case full persistence?

Each node can store up to $2p$ modifications. When a node is full, we split it into two roughly half-full nodes (like B-trees). Using $\Phi =$ the number of full nodes in all versions, we get $O(1)$ amortized cost. Each data structure node is represented by a linked list of nodes, and there's a second phase of the operation to update reverse pointers. Deitz developed a fully persistent array that can be achieved in $O(\lg\lg n)$ $\times$ overhead in the word RAM model [5].
**OPEN:** Is there a matching lower bound for both full and partial persistence? This question may have been solved by Pǎtrascu et al. (unpublished).

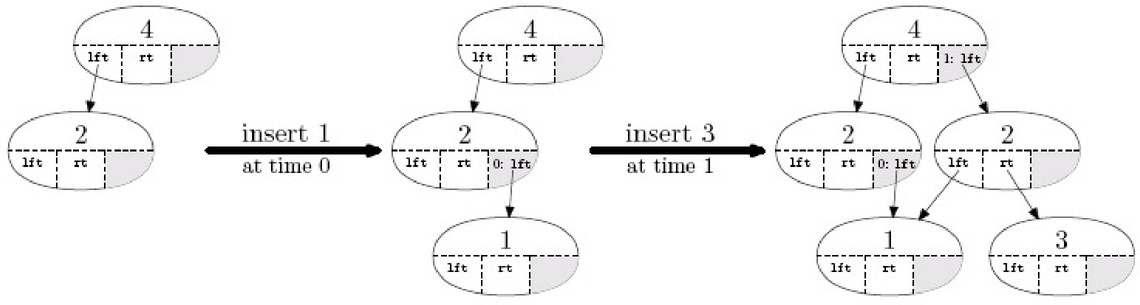

Figure 2: Persistence of a binary tree. We need a modification box the size of the in-degree of each data structure node (just one for trees).

### 2.1.3 Confluent Persistence

Confluent persistence has been explored in functional data structures [9]. Deques (double ended queues allowing stack and queue operations) with concatenation can be done in constant time per operation (Kaplan, Okasaki, and Tarjan [7]). We can create implicity exponential deques in polynomial time by recursively concatenating a node with itself. The general transformation due to Fiat and Kaplan [8] is as follows:

- $d(v)$ = depth of node $v$ in version DAG
- $e(v) = 1 + lg($ number of paths root to $v)$
- overhead: $\lg($ number of updates $) + max_v(e(v))$

- poor when $e(v) = 2^u$ where $u$ is the number of updates (see Figure 1). This is still exponentially better than the non-persistent model.

**OPEN:** When can you do better? Lists with split and concatenate? Trees? General pointer machine? Array with cut and paste? Special DAGs? Others?

## 2.2 Retroactivity

Retroactive data structures store a data structure under a sequence of operations. We would like to be able to go back in time, change an operation, and then observe the effects of that change in the current state of the data structure. The induced timeline is linear. Much of this work is due to Demaine, Iacono, and Langerman [3].

The allowed operations are

- Insert$(t, x)$ – Retroactively do operation $x$ at time $t$
- Delete$(t)$ – Retroactively undo operation at time $t$
- Query$(t, x)$ – Do query $x$ at time $t$

We define partial retroactivity as allowing queries at the present time and full retroactivity as allowing queries at any time. Some cases of partial retroactivity are easy to implement. If updates are commutative: $x \circ y = y \circ x$, then we can support retroactive insertion of operations at no additional asymptotic cost (implement $Insert(t, x)$ by executing $x$ at the present time). If updates, in addition to being commutative, are also invertible: $x \circ x^{-1} = NOP$, then we can support partial retroactivity at no additional asymptotic cost ($Insert(t, x)$ by executing $x$ at the present time and $Delete(t)$ by executing $x^{-1}$ at the present time where $x^{-1}$ is the inverse of the operation at time $t$).

### 2.2.1 The Rollback Method

There are a few general transformations that we can prove bounds for. One is the rollback method, in which we perform a retroactive operation at $r$ time units in the past. We can do this with a factor of $r$ overhead by keeping a log of all updates done to the DS such that every change can be reversed. The rollback method needs an $\Omega(r)$ lower bound. To see this, we examine a data structure that maintains two values, $X$ and $Y$, both initialized to 0. We can perform the following operations on our data structure:

- set$X(x)$ – Sets $X \leftarrow x$
- add$Y(\Delta)$ – Sets $Y \leftarrow Y + \Delta$
- mult$XY()$ – Sets $Y \leftarrow X \cdot Y$
- query() – Returns $Y$

Consider the following sequence of operations: add$Y(a_n)$, mult$XY()$, add$Y(a_{n-1})$, mult$XY()$, ..., add$Y(a_0)$. This is Horner's rule for evaluating the polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$. Now suppose we perform $Insert(t = 0,$

set$X(x_0)$) to change the $x$ value of the evaluated polynomial $x_0$. Frandsena, Hansen, and Miltersen in 2001 showed that evaluating a polynomial of degree $n$ requires $\Omega(n)$ time over any field, independent of any preprocessing of the $a_i$s. This holds in the "history-independent algebraic decision tree" model, which implies the same result for the integer RAM and generalized real RAM models [10]. This is somewhat disappointing result because it says that in the retroactive model, we can't do any persistence maintenance that's better than just going back in time, performing the new operation, and then re-executing all of the operations in our history past that point.

In the cell-probe model, Frandsena et al. also proved a lower bound of $\Omega(\sqrt{r/\lg r})$. They had a data structure that maintained $n$ words and supported arithmetic updates $(+/\cdot)$. Computing a fast fourier transform takes $O(n \lg n)$ time, but changing one weights $w_i$ of the FFT needs $\Omega(\sqrt{n})$ time, from which we derive the $\Omega(\sqrt{r/\lg r})$ lower bound. An open question is whether the tightest cell-probe lower bound is $\Omega(\sqrt{r/\mathrm{poly}\lg r})$.

### 2.2.2 Priority Queues

Let us turn our attention to partially retroactive priority queues. The defining features of priority queues is the delete-min() operation, which makes the set of operations on priority queues non-commutative. We can plot the status of our data structure in the 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. 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. It turns the horizontal ray into a line segment with endspoints $(t,k)$ and $(d_k,k)$, where $d_k$ is the time of key $k$'s deletion. This creates nonintersecting upsidedown "L" shapes, where each L corresponds to an *insert* and the *delete-min*() that deletes it. Refer to Figure 3 for an animation.

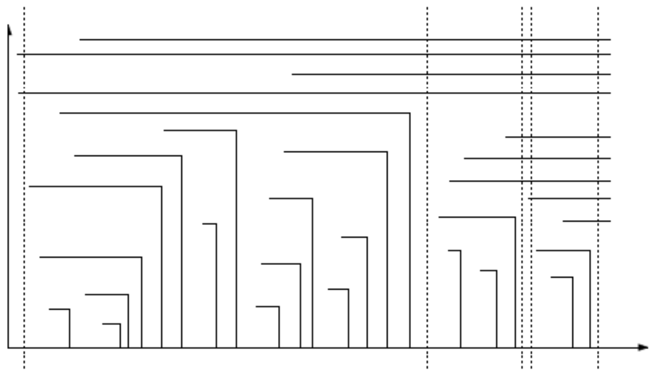

Figure 3: The "L" representation of a sequence of operations. Dotted horizontal lines represent *bridges*. [3]

Let $Q_0$ be the current state of our priority queue and $Q_t$ be its state at time $t$. We call time $t$ a *bridge* if $Q_t \subseteq Q_0$. There are four combinations of retroactive operations:

1. *Insert*$(t,$"*insert*$(k)$") – Insert key $k$ into $Q_t$. The resulting element we insert into $Q_0$ is the largest element that was deleted after time $t$. See Figure 4.
2. *Delete*$(t,$"*delete-min*()") – Undo the *delete-min* at time $t$. This is identical to re-inserting the element that was being deleted at the time of deletion (i.e. nullify the upwards *delete-min*() arrow by inserting

the appropriate key right at that time). Thus, it is the same as the above case and we insert into $Q_0$ the largest element that was deleted after time $t$.

3. $Insert(t,$"$delete\text{-}min()$"$)$ – Delete the minimum key at time $t$. The element we delete from $Q_0$ is the minimum value of $Q_{t'}$, where $t'$ is the first bridge after $t$. This essentially pushes the bridge forward in time. See Figure 5.

4. $Delete(t,$"$insert(k)$"$)$ – Undo the insertion of key $k$ at time $t$. If $k \in Q_0$ we remove it from there. If not, then we again delete the minimum value of $Q_{t'}$ where $t'$ is the first bridge after $t$. The idea is that since $k \notin Q_0$, it didn't make it to its next bridge. Therefore, removing the insertion of that number will cascade up the deletes before that bridge, so the minimum element from the that bridge will get removed by the last cascaded delete.

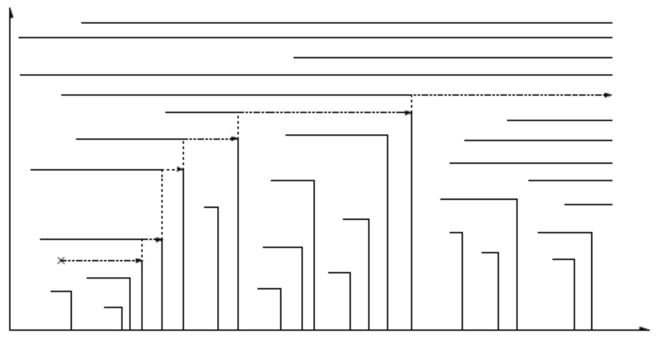

Figure 4: The $Insert(t,$"$insert(k)$"$)$ operation causes a cascade of changes of delete times, and one insertion into $Q_0$. [3]

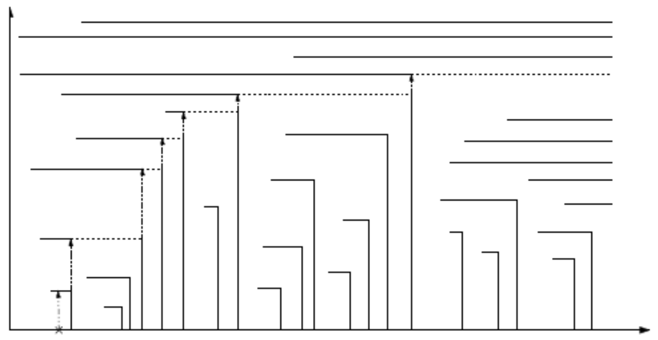

Figure 5: The $Insert(t,$"$delete\text{-}min()$"$)$ operation causes a cascade of changes of delete times, and one deletion into $Q_0$. [3]

We can perform all of these operations in $O(\lg m)$ worst-case, where $m$ is the total number of updates, present or retroactive, performed on the priority queue. We first store a balanced BST insertions keyed on insertion time. We augment each node of this tree with the max key $k' \notin Q_0$ over every node's subtree. This lets us find the maximum key among all elements inserted after a time $t'$ but not in $Q_0$ in $O(\lg m)$ time. We can also find the minimum key among all elements inserted before a time $t'$ and still in $Q_0$ in the same runtime if we maintain in every node the minimum of all keys in its subtree still in $Q_0$. These are useful for operations 1 and 2, and 3 and 4, respectively.

To find the last bridge before $t$ or the first bridge after $t$, we maintain a list of updates which we store in a modified $(a,b)$-tree developed by Fleischer [11]. If we assign a weight of 0 to $\mathit{insert}(k)$ for $k \in Q_0$, +1 to $\mathit{insert}(k)$ for $k \notin Q_0$, and -1 for $\mathit{delete\text{-}min}()$, every bridge in the tree corresponds to a prefix sum of 0. This allows us to find bridges in $O(\lg m)$ time, which we use for every operation.

## 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.

[2] Gerth Stlting Brodal: *Partially Persistent Data Structures of Bounded Degree with Constant Update Time*. Nord. J. Comput. 3(3): 238-255 (1996)

[3] Erik D. Demaine, John Iacono, Stefan Langerman: *Retroactive data structures*. SODA 2004: 281-290

[4] Paul F. Dietz, Daniel Dominic Sleator: *Two Algorithms for Maintaining Order in a List* STOC 1987: 365-372

[5] Paul F. Dietz: *Fully Persistent Arrays (Extended Array)*. WADS 1989: 67-74

[6] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, Robert Endre Tarjan: *Making Data Structures Persistent*. J. Comput. Syst. Sci. 38(1): 86-124 (1989)

[7] Haim Kaplan, Chris Okasaki, Robert Endre Tarjan: *Simple Confluently Persistent Catenable Lists*. SIAM J. Comput. 30(3): 965-977 (2000)

[8] Amos Fiat, Haim Kaplan: *Making data structures confluently persistent*. J. Algorithms 48(1): 16-58 (2003)

[9] Chris Okasaki: *Purely Functional Data Structures*. New York: Cambridge University Press, 2003.

[10] Gudmund Skovbjerg Frandsen, Gudmund Skovbjerg Frandsen, Peter Bro Miltersen: *Lower Bounds for Dynamic Algebraic Problems*. Information and Computation 171(2): 333-349 (2001)

[11] Rudolf Fleischer: *A Simple Balanced Search Tree with $O(1)$ Worst-Case Update Time*. Int. J. Found. Comput. Sci. 7(2): 137-150 (1996)