Non-oblivious Retroactive Data Structures

Umut A. Acar[1] **Guy E. Blelloch**[2] **Kanat Tangwongsan**[3]

December 11, 2007
CMU-CS-07-169

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]Toyota Technological Institute at Chicago (TTI-C), Chicago IL 60637. Email: *umut@tti-c.org*.
[2]Computer Science Department, Carnegie Mellon University, Pittsburgh PA 15213. Email: *blelloch@cs.cmu.edu*.
[3]Computer Science Department, Carnegie Mellon University, Pittsburgh PA 15213. Email: *ktangwon@cs.cmu.edu*.

Keywords: data structures, dynamization, retroactive data structures

Abstract

The idea of a retroactive version of a data structure is to maintain a time-ordered sequence of operations while allowing the user to revise the operation sequence by *invoking* and *revoking* (i.e., inserting and deleting, respectively) operations anywhere in the sequence—including backwards in time. In many applications of retroactivity, operations depend on the outcomes of previous queries, and therefore the data structures need to identify the queries whose outcome changes when a revision is performed retroactively. Existing notions of retroactivity, however, do not keep track of queries in the operation sequence. Therefore, they cannot efficiently identify the queries that become inconsistent as a result of a retroactive revision.

In this paper, we propose and study a new model of retroactivity, called *non-oblivious retroactivity*, where both updates and queries are maintained as part of the operation sequence. In this model, a revision to the operation sequence returns the earliest operation that becomes *inconsistent*, i.e., an operation whose return value differs from before. This mechanism enables the user to efficiently locate the affected operation and decide to perform further revisions as necessary to reestablish the consistency of the operation sequence. We investigate several non-oblivious data structures and prove some lower bounds in the proposed model.

1 Introduction

Consider a data type that supports update and query operations. The idea of a retroactive version of the data type is to maintain a time-ordered sequence of operations while allowing the user to revise the operation sequence by *invoking* and *revoking* operations, i.e., inserting and deleting (respectively), anywhere in the sequence, including back in time. The ability to revise the sequence of operations performed on a data structure is often helpful and sometimes critical. For example, if incorrect information is entered into a database, then it may be important to find all the decisions based on this information and revise them efficiently. Similar applications of retroactivity can be found in areas such as security and algorithm dynamization [DIL04].

Demaine, Iacono, and Langerman [DIL04] introduced two models of retroactivity, called partial and full retroactivity. The models differentiate between two kinds of operations: updates and queries. *Update operations* (e.g., enqueue) modify the underlying data structure but do not return a value. *Query operations* inspect the underlying data structure and return a value (e.g., front). Both models allow update operations to be revoked or invoked at any time, but they restrict queries. In partial retroactivity, queries can be invoked only at the most recent time; in full retroactivity, queries can be invoked at any time. Neither model maintains a record of queries—only a record of update operations is maintained. Therefore, queries cannot be revoked. Furthermore, when a revision back in time changes the outcome of a query operation, the models will not identify the queries that will now return a different value. In other words, the sequence of operations are oblivious to revisions. We therefore refer to these models of retroactivity as *oblivious*.

Incorporating query operations to retroactivity, however, has crucial benefits. As an example, consider a database where some incorrect information is mistakenly entered. Without the ability to identify the queries affected by this operation, it is difficult to correct the problem caused by the incorrect information: one has to keep a separate record of the query operations and re-perform them after each update to check if their return values change; this is inefficient. For the same reasons, oblivious retroactive data structures are also difficult to compose and to use as components in dynamic algorithms. For example, consider the dynamization of the Dijkstra's shortest-paths algorithm that uses a retroactive priority queue. If the user inserts an edge into the graph, then we can retroactively invoke a new insert operation on the priority queue based on that edge. This could affect a later deleteMin operation and could further cause other revisions. If the retroactive structure only tracks updates, however, there is no way to determine which operations are affected efficiently—all future operations are oblivious to the update. We see no way, for example, to effectively use Demaine et al.'s retroactive priority queue to dynamize Dijkstra's algorithm. Similarly we see no way to effectively use their retroactive union find to dynamize graph connectivity.

In this paper, we propose a model of *non-oblivious retroactivity* and proceed to design and analyze several non-oblivious retroactive data structures. Henceforth, we use the term "retroactive data structures" to refer to non-oblivious data structures unless otherwise stated. Our model maintains all operations (both queries and updates) as part of the operation sequence. When the user performs a retroactive revision by invoking or revoking an operation, the user is notified of the next operation that becomes *inconsistent*, i.e., an operation whose return value differs from before. This mechanism allows the user to propagate the effects of the revision through the operation sequence. For example, if a piece of incorrect information is entered into a database, then the erroneous operation can be revoked when discovered. After such a revision, the data structure will return the first inconsistent operation. The user can then take corrective actions, e.g., revoke the inconsistent operation. Based on this notion of propagation, our data structures can be used to dynamize static algorithms (cf. Section 3). Our data structures only report the operations that are inconsistent and no more.

All our results rely on an order-maintenance data structure to keep track of a time line consisting of time stamps. We write R for the number of operations in the operation sequence and T for the total number of allocated time stamps[1]. Table 1 summarizes the time bounds for each retroactive revision (except for ordered sets) and total

[1]This can be significantly larger than r when multiple retroactive data structures are used.

space usage of the data structures. All our bounds are expected amortized where the expectations are taken over internal randomization. We show matching lower bounds for stacks and priority queue.

Data Structure	Time	Space
Dictionary	$O(\log\log T)$	$O(R)$
Queue	$O(\log\log T)$	$O(R)$
Stack	$O(\log R/\log\log R)$	$O(R)$
Priority Queue	$O(\log R)$	$O(R)$
Ordered Set	‡ $O((m+1)\,n\,\log R)$	$O(R)$

Table 1: Time and space bounds for retroactive data structures. (‡: for a sequence of n revised updates and m revoked queries)

We note that the results we present are not comparable and sometimes quite different from those known for the oblivious case. For example for the oblivious fully retroactive priority queues, Demaine et al. achieve a $O(\sqrt{r}\log r)$ time per update for r operations. In the non-oblivious case, however, we are able to achieve $O(\log r)$ time. In this case the non-oblivious version seems easier since all that is required is that it report the next inconsistent operation. In priority queues, invoking a new insert operation back in time will make some future deleteMin inconsistent. By performing an additional deleteMin operation, the user can cancel the effect of the previous update and the data structure can reach a consistent state quickly. On the other hand, for oblivious, ordered dictionaries, Demaine et al. achieve $O(\log r)$ per update for r operations. In the non-oblivious case, we are able to achieve $O((m+1)\,n\,\log r)$ time, where n is the number of revised updates and m is the number of revoked queries. This matches the time previous bound when the queries are not maintained (as in the oblivious case) in the amortized sense. When queries are included, the problem seems harder.

2 Preliminaries and Notations

In this section, we define some notations and survey results that will be used throughout the paper. For $x_1 \leq x_2$, we denote by $[x_1, x_2]$ the closed interval between x_1 and x_2; that is, we define $[x_1, x_2] = \{x : x_1 \leq x \leq x_2\}$. Extending this notation, we write $([x_1, x_2], y)$ to denote a line segment connecting (x_1, y) and (x_2, y). Similarly, we define $(x, [y_1, y_2])$ to be the line segment connecting (x, y_1) and (x, y_2). It is possible that the right (or top) endpoint is infinite, in which case the notation represents a ray extending indefinitely from the starting point. We now move on to surveying results we need in this paper.

Order maintenance and ordered subsets. The order maintenance (OM) problem is to maintain an ordered set $\mathcal{T}$ while supporting the following operations: $\mathsf{insert}(x)$ inserts a new element y immediately following x in the ordering of $\mathcal{T}$ and returns it; $\mathsf{delete}(x)$ deletes the element x from $\mathcal{T}$; and $\mathsf{order}(x, y)$ for $x, y \in \mathcal{T}$ returns true if x precedes y in the ordering of $\mathcal{T}$. Dietz and Sleator [DS87] and Bender et al. [BCD+02] described algorithms that support all operations in $O(1)$ worst-case time. We make use of the OM data structure throughout this paper and will often refer to the elements of $\mathcal{T}$ as time stamps.

The ordered subsets (OS) problem is to maintain a set $\mathcal{T}$ supporting the OM operations, along with a set of subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ of $\mathcal{T}$, each supporting the following operations: $\mathsf{insert}(S_i, x)$ inserts $x \in \mathcal{T}$ into S_i, $\mathsf{delete}(S_i, x)$ deletes x from S_i, $\mathsf{findPrev}(S_i, x)$ for $x \in \mathcal{T}$ returns $\max\{e \in S_i | e \leq x\}$, and $\mathsf{findNext}(S_i, x)$ returns $\min\{e \in S_i | e \geq x\}$. The OS problem also supports adding (removing) an empty set to (from) $\mathcal{S}$. Assuming that each element of $\mathcal{T}$ belongs to at most one subset, Mortensen [Mor06] shows that the OS problem can be maintained in expected $O(\log\log|\mathcal{T}|)$ time per operation. Lifting this restriction, Blelloch and Vassilevska [BV07] show that the OS problem can be solved such the OM operations are $O(1)$ time, adding and removing sets from S take $O(1)$ worst-case time, insert and delete take $O(\log\log|\mathcal{T}|)$ expected amortized time, and $\mathsf{findNext}$ and $\mathsf{findPrev}$ take $O(\log\log|\mathcal{T}|)$ worst-case time; the total space is bounded by $O(|\mathcal{T}| + \sum_{i=1}^{n}(1 + |S_i|))$.

Dynamic orthogonal point location. The variant of dynamic orthogonal point-location problem we consider is to maintain a set S of segments of the form $([x_1, x_2], y)$ while supporting the following operations: $\mathsf{insert}(e)$ inserts a new segment e into S; $\mathsf{delete}(e)$ deletes the segment e from S; and $\mathsf{above}(x, y)$ reports the first segment of S that the ray extending from (x, y) hits. Note that when the segments are horizontal, the query—now called

$\mathsf{right}(x, y)$—can be analogously supported. This and related problems have been extensively studied in literature. We describe two results here. A simple modification of Mehlhorn and Näher's dynamic segment tree [MN90] can support all above operations in $O(\log |S| \log\log |S|)$ per operation, requiring $O(|S| \log |S|)$ space. Recently Blelloch [Ble07] shows a data structure for this problem that supports updates and queries in amortized $O(\log |S|)$ time per operation and uses linear space.

3 The Framework

Consider a data type D and let O be the set of operations defined on D and V be the set of values that these operations may return. A data type defines a function $F\colon O^* \to V$ that maps sequences of operations to the value returned by the last operation in the sequence starting with a predefined initial state. For example, if D is a queue of tasks, then the operation set O includes enqueue and dequeue operations and the return values consist of **null** (to be returned by enqueue and dequeue when the queue is empty) and all possible tasks. The queue data-type defines a function that maps an operation sequence to a member of V starting with an empty queue.

Given a total-ordered set of *time stamps* $\mathcal{T}$, we define a *trace* for D as a set $\mathcal{R} \subseteq O \times V \times \mathcal{T}$, where each time stamp appears at most once. Intuitively, we think of a trace as a sequence of operations together with their return values ordered with respect to time. For example, the trace for a sequence of operations on a queue consists of the operations along with the returned elements (if any) ordered in time. For $t \in \mathcal{T}$, we define $\textsc{Prefix}(\mathcal{R}, t)$ as the sequence of operations in $\mathcal{R}$ up to t (inclusive) in time order. Note that $\textsc{Prefix}(\mathcal{R}, t)$ contains only the operations (not the returned values nor the times). We say that an element (o, v, t) of the trace is *consistent* if $v = F(\textsc{Prefix}(\mathcal{R}, t))$—i.e., it is associated with the correct value given the preceding operations. We say that a trace is consistent if all elements are consistent.

For a data type $D = (O, V, F)$, the *retroactive version* D_r maintains a trace $\mathcal{R}$ for each instance and provides an operation for creating a new instance and revision operations for updating the trace:

- $\mathsf{new}()$: Returns an empty trace $\mathcal{R}$ for a new instance.
- $\mathsf{invoke}(\mathcal{R}, o, t)$: Updates the trace $\mathcal{R}$ by inserting (o, v, t), computes $v = F(\textsc{Prefix}(\mathcal{R}, t))$, and returns the tuple consisting of v and the time of the earliest inconsistent operation in $\mathcal{R}$.
- $\mathsf{revoke}(\mathcal{R}, t)$: Updates the trace $\mathcal{R}$ by removing the element with time t (if any) and returns the time of the earliest inconsistent operation in $\mathcal{R}$.

We refer to invoke and revoke operations as *revisions*. Although the model allows revisions at any time, for our results, we assume that revisions are applied as part of a *monotone revision sequence*—a sequence of revisions on an initially consistent trace such that (1) the times of the revisions is increasing, and (2) for each revision at time t, all operation at times before t are consistent.

Dynamization. As an example of how retroactive data structures can be used to dynamize static algorithms, consider Dijkstra's algorithm for single-source shortest paths. Suppose that we execute the algorithm on a given graph using a retroactive priority queue and a retroactive array that stores the computed distances (we initialize the elements of the array to infinity). Suppose now we insert an edge from u to v (deletions are symmetric). We go back in time and invoke an insert operation for that edge. If inserting the edge does not change the output, then we are done. Otherwise, this invoke operation will return the deleteMin operation that removes v. At this point, we re-perform the deleteMin operation to obtain the distance to v and update the result array if v's distance is still infinity (otherwise, v's distance has already been determined and we are done). We then relax all the edges outgoing from v by revoking the corresponding insert operations and invoking them again with the new distance. These operations will return further inconsistencies, which we will process by moving to the earliest one repeating until no more inconsistencies remain. Although we do not prove it here, the resulting algorithm closely matches the bounds of Ramalingam and Rep's dynamic shortest-paths algorithm [RR96]. In general, by identifying the operations that become inconsistent after a revision, non-oblivious retroactive data structures allow dynamization of static algorithms by using a straightforward propagation mechanism.

4 General Theory

We study non-oblivious retroactive data structures in relation to other data structures. A natural question to ask is when automatic retroactivity is possible. Despite the differences between the models, the rollback method described in Demaine et al. provides a general technique for automatic non-oblivious retroactivity with a slight modification. Instead of remembering only the update operations, we also remember the query operations, and thus we can trivially determine which operation is inconsistent when we "play back" the sequence. It is straightforward to prove the following theorem:

Theorem 4.1 *Given a data structure where each operation takes $T(n)$ worst-case time, the rollback method yields a (non-oblivious) retroactive data structure that supports the same set of operations retroactively in time $O(rT(n))$, where r is the number of operations occurring after the time that the action takes place.*

Unsurprisingly, the rollback method is in most cases far less efficient than a specially designed retroactive data structure for the task at hand. Perhaps more surprising is the result that there are data structures for which the rollback method is essentially the best possible. We consider an example of such data structures here. The data structure maintains a counter X and supports one operation $\mathsf{incr}()$, which increments X and returns the value of X right before the increment. Initially we perform a sequence of m $\mathsf{incr}()$ operations. Going back in time, we perform an $\mathsf{incr}()$ operation at the beginning of the sequence. This has a cascading effect that requires every $\mathsf{incr}()$ operation to be revoked and re-performed. For this reason, we cannot hope to be more efficient than the rollback method.

Finally, we note that any fully retroactive data structure can be made non-oblivious by applying the rollback method to the sequence of queries. This, however, is generally very inefficient, because it requires invoking all the queries after a revision even when unnecessary. Consider, for example, a stack data structure and a sequence of n push operations followed by n pop operations. Inserting a push operation at the start of this trace will require all pop operations to be re-executed, even though there are no inconsistent operations. Our results show that we can be much more efficient with a non-oblivious stack data structure (Section 6).

5 Dictionaries and Arrays

We now turn our attention to developing efficient retroactive data structures for specific problems. In this section, we discuss a dictionary data structure. Note that arrays are a special case of dictionaries when the identity mapping is used. Given a universe of keys K and a universe of values V, a dictionary maintains a set $D \subseteq K \times V$ in which each key appears at most once and supports the operations:

- $\mathsf{insert}(k, v)$: Add (k, v) to D, replacing the old value for k if one.
- $\mathsf{delete}(k)$: Delete any element with key k from D.
- $\mathsf{lookup}(k)$: Return $\{(k', v) \in D : k' = k\}$.

We assume a universal class of hash functions over K so hashing can be used to implement the dictionary.

Theorem 5.1 *A retroactive dictionary D_r with trace $\mathcal{R}$ over times $\mathcal{T}$ can be maintained in $O(|\mathcal{R}|)$ space so that all operations take $O(\log\log|\mathcal{T}|)$ expected amortized time.*

Proof: Our solution maintains a dictionary DK mapping each key k appearing in $\mathcal{R}$ to a set S_k, a dictionary DT mapping each time appearing in $\mathcal{R}$ to the key involved in that operation, and a set PQ storing each key that has an inconsistent operation. DK and DT use universal hashing (e.g., [WC79]) for constant expected amortized access and update. The sets $S_k \subseteq \mathcal{T} \times V \times \{\mathsf{insert}, \mathsf{delete}, \mathsf{lookup}\}$ contain the time, value at that time, and operation type for all the operations on k in $\mathcal{R}$. They are stored as ordered subsets of $\mathcal{T}$. We say that a key is *inconsistent* if it contains an inconsistent lookup. The set PQ contains all inconsistent keys k, and is stored as an ordered subset of $\mathcal{T}$ based on the time of the earliest inconsistent operation for k.

For $\mathsf{invoke}(\mathsf{insert}(k, v), t)$ we insert (t, v, insert) into S_k. For $\mathsf{invoke}(\mathsf{lookup}(k), t)$ we find v from the predecessor of t in S_k, insert (t, v, lookup) into S_k, and return v. For $\mathsf{invoke}(\mathsf{delete}(k), t)$ we insert $(t, \cdot, \mathsf{delete})$ into S_k. For any revoke at time t we use DT to find the appropriate key k and then delete the element at t from S_k. If performing an insert is on a new key k, then a new set S_k is inserted into DK, and if revoking an insert removes the last element from S_k then k is deleted from DK. For any revision on key k at time t let u be the successor operation in S_k. If the current value of u differs from its predecessor we insert u into PQ at time t, otherwise we delete u from PQ. All revisions update DT. Finally we report the minimum time in PQ, or ∞ if PQ is empty.

All operations described take at most $O(\log\log|\mathcal{T}|)$ amortized expected time for the ordered subset operations. The space is linear in the number of entries in all the S_k, which has a one-to-one correspondence to elements in $\mathcal{R}$, and the space for DK, DT and PQ which are all at most linear in the length of the trace. ■

6 Stacks

Consider a stack data structure S that supports the operations $\mathsf{push}(k)$, which pushes the key k to the top of S, and $\mathsf{pop}()$, which pops and returns the value at the top of S. In this section, we discuss a retroactive stack data structure and show a lower bound. We first consider a closely related problem called the dynamic ± 1 prefix-sum problem, which we use as a subroutine in the retroactive stack data structure.

6.1 Dynamic ± 1 prefix-sum problem

An instance of the dynamic ± 1 prefix-sum problem maintains a set $S \subseteq \mathcal{T} \times \{-1, +1\}$ where $\mathcal{T}$ is an total-ordered set of time stamps and supports the following operations: $\mathsf{insert}(t, v)$ inserts the pair (t, v) into S; $\mathsf{delete}(t)$ removes the unique pair with first coordinate t from S; $\mathsf{prefix\_sum}(t)$ reports $\sum_{(s,v):s\le t} v$; $\mathsf{pred}(t, r)$ reports $\max\{t' : t' < t \text{ and } r = \sum_{(s,v):s<t'} v\}$; and $\mathsf{succ}(t, r)$ reports $\min\{t' : t' > t \text{ and } r = \sum_{(s,v):s<t'} v\}$[2]. In this section, we develop a solution to this problem. In particular we show that an instance S of the dynamic ± 1 prefix-sum problem can be maintained in $O(|S|)$ space such that all operations take $O(\log|S|/\log\log|S|)$ time. It is easy to obtain an $O(\log|S|)$ upper bound for all the operations using a balanced binary tree with all the elements of S at the leaves and storing partial sums and minimum values for each subtree. However, to achieve the promised bound we need to develop a specialized search structure, called an S-structure, described below.

Lemma 6.1 (S-structure) *Let the word size be $u = O(\log N)$. Let ϵ be a sufficiently small constant. An $O(\log^\epsilon N)$-element array of integers in the range $[-O(N), O(N)]$ can be maintained in $O(\log^\epsilon N)$ space such that the following operations each takes amortized $O(1)$ time:* $\mathsf{incr}(a, b)$ *increments $A[i]$ for all $i \in \mathbb{Z} \cap [a, b]$;* $\mathsf{decr}(a, b)$ *decrements $A[i]$ for all $i \in \mathbb{Z} \cap [a, b]$;* $\mathsf{prev}(a, x)$ *reports $\max\{i \in \mathbb{Z} : i < a$ and $A[i] \ge x\}$; and* $\mathsf{next}(a, x)$ *reports $\min\{i \in \mathbb{Z} : i > a$ and $A[i] \ge x\}$.*

Proof: Pick $\delta = 1/\log^\epsilon N$, so $\log^\epsilon N < N^\delta < N$. Our construction relies on a lemma of Mortensen [Mor03] and the observation that an $O(\log^\epsilon N)$-element array of integers in the range $[0, N^\delta]$ can be represented in a single word such that one can increment any elements simultaneously in $O(1)$ time—given that an appropriate bitmask can be constructed in $O(1)$. In this proof we refer to the array we just described as a C-array ("compact array").

[2]This problem should be contrasted with the well-studied prefix-sums problem [Die89, PD04], in which one maintains a fixed-length array A and supports the operations $\mathsf{update}(i, v)$ which sets $A[i]$ to v, and $\mathsf{sum}(i)$ which reports $\sum_{j<i} A[i]$. The data structure does *not* support predecessor and successor queries.

For each entry $A[i]$, we maintain three quantities: (1) p_i is the number of increments modulo N^δ performed on $A[i]$, (2) m_i is the number of decrements modulo N^δ performed on $A[i]$, and (3) a_i is the "approximate" value of the entry $A[i]$. The following invariant relates these quantities: $A[i] = a_i \cdot N^\delta + (p_i - m_i)$, where $a_i, p_i, m_i \geq 0$. We maintain two C-arrays, $\mathcal{P}$ for p_i's and $\mathcal{M}$ for m_i's. We store the values a_i's in a structure described in Lemma 3.2 of Mortensen [Mor03]. Built on q-heap of Fredman and Willard [FW94], the structure of Mortensen can maintain an $O(\log^\epsilon N)$-element array of integers in the range $[-O(N^{1-\delta}), O(N^{1-\delta})]$ in $O(\log^\epsilon N)$ space such that updating the value of a specific element takes $O(1)$ and the query $\mathsf{report}(A, i, j) = \{k : i \leq A[k] \leq j\}$ takes $O(1)$ time.

The operation incr (decr resp.) is supported by bulk-incrementing the corresponding entries of the $\mathcal{P}$ array ($\mathcal{M}$ array resp.), which takes $O(1)$ time. Only when an overflow occurs do we need to perform extra work. If an entry p_i (or m_i) overflows, set it to 0 and update a_i. Note that multiple a_i's may need to be updated at the same time, but when this happens, we will have performed at least N^δ operations; since $N^\delta > \log^\epsilon N$, we can charge the cost of updating a_i's to those N^δ operations.

Let $\mathsf{lo}(x) = x \bmod N^\delta$ and $\mathsf{hi}(x) = \lfloor x/N^\delta \rfloor$. The $\mathsf{prev}(a, x)$ operation can be supported in $O(1)$ as follows: first compute $C_0 = \{i : m_i = \mathsf{hi}(x)\}$, $C_1 = \{i : m_i = 1 + \mathsf{hi}(x)\}$, and $C_{\geq 2} = \{i : m_i \geq 2 + \mathsf{hi}(x)\}$, each of which can be discovered in $O(1)$ time by a report query. Then we compute the following sets: $S_0 = \{i \in C_0 : p_i - m_i \geq \mathsf{lo}(x)\}$, and $S_1 = \{i \in C_1 : N^\delta + p_i - m_i \geq \mathsf{lo}(x)\}$, which can be accomplished by simple arithmetic operations and look-up tables of size $O(N)$. It is easy to verify that $S_0 \cup S_1 \cup C_{\geq 2} = \{i : A[i] \geq x\}$. We note that each of these sets can be compactly represented as a bit vector in a single word, and hence unions and intersections can be done in constant time. Finally, consider that $\max\{i : i > a$ an $A[i] \geq x\}$ corresponds to the most significant bit—in an appropriate bit-vector representation—of the set $(S_0 \cup S_1 \cup C_{\geq 2}) \cap [a-1]$. The msb can be computed in constant time using the technique of Fredman and Willard [FW94]. The next operation is symmetrical. ■

We are now in a position to prove the key theorem of this section.

Theorem 6.2 (Dynamic Prefix-Sum Search Structure) *An instance S of the dynamic ± 1 prefix-sum problem can be maintained in $O(|S|)$ space such that all operations take $O(\log |S| / \log\log |S|)$ time.*

Proof: We store the elements of S ordered by t at the leaves of a weight-balanced B-tree (WBB-tree) [AV03, Wil85, Die89] of order $B = \log^\epsilon N$. While other alternatives exist, our data structure is easiest to describe with a WBB-tree. An internal node v with the children $u_1, \ldots, u_B$ keeps track of the following information:

- $v.\textit{psum}$ is the sum of all values (the v component) stored inside this subtree.
- $v.\textit{prefix}[i]$ is the sum of the *psum*'s of the subtree $u_1, \ldots, u_i$ (i.e., $v.\textit{prefix}[i] = \sum_{j=1}^{i} u_j.\textit{psum}$). Let us define $v.\textit{prefix}[0] = 0$.
- $v.\textit{bottom}[i]$ records the following information. Let $\ell(u_i)$ be the set of times of all the operations stored at leaves of the subtree u_i. For reasons that will become apparent, we define $v.\textit{bottom}[i]$ as $(v.\textit{prefix}[i-1] - \min\{\mathsf{prefix\_sum}(t) : t \in \ell(u_i)\})$. Essentially, *bottom*$[i]$ records the value of the smallest $\mathsf{prefix\_sum}$ inside the subtree rooted at u_i in a form which is easy to update.

These fields are stored in the S-structures of Lemma 6.1 so that (bulk) updates and queries take $O(1)$ time. We now describe how to answer queries. For $\mathsf{prefix\_sum}(t)$, we sum up the appropriate values the *prefix* array of the nodes along the root-to-leaf path. For $\mathsf{pred}(t, r)$, the key observation is that the subtree rooted at v contains a time with prefix sum r if and only if the least prefix-sum value in the subtree is at most r. Starting at a leaf corresponding to t, we walk up the tree (towards the root) until we find a node that has the target prefix-sum value inside the subtree. We now follow the largest child that possesses the target value until we reach a leaf. The prev operation (Lemma 6.1) allows for determining the largest child that contains the target value in $O(1)$. The succ operation is symmetrical.

For insertions and deletions, we insert or delete a leaf in the tree and update the summary fields whose values change. Note that we can only affect values of the nodes along the path we traverse. Moreover, the values change by at most 1 and can be updated in bulk in $O(1)$ using operations of the S-structure. In a WBB-tree, a node is marked before a split, which then occurs at the marked position. Known bounds [AV03, Mor03] show that there is enough time to create two copies for each side before the split takes place. Using the global-rebuilding technique of Overmars [Ove83], we perform deletions by simply removing the leaves without reorganizing the tree; in the meantime, a new tree is gradually constructed. ■

6.2 Efficient Retroactive Stacks

We describe an efficient retroactive stack that builds on the dynamic ± 1 prefix-sums of the previous section.

Theorem 6.3 *A retroactive stack S_r with trace $\mathcal{R}$ can be maintained in $O(|\mathcal{R}|)$ space such that any revision operation takes amortized $O\left(\frac{\log|\mathcal{R}|}{\log\log|\mathcal{R}|}\right)$ time.*

Proof: We keep all stack operations in a dynamic prefix-sum search structure PSS of Theorem 6.2. Each stack operation is assigned a value: push is $+1$ and pop is -1. This translation gives that $\mathsf{prefix\_sum}(t)$, the prefix sum of up to time t, is the height of the stack at t. For a trace $\mathcal{R}$, we define the rank at time t to be the height of the stack at that time.

During a revision process, we maintain a (standard) stack S_I of inconsistent pop's. The stack has the property that the trace is consistent up to the time at the top of the stack. Thus, when the trace is consistent, S_I is empty. For $\mathsf{invoke}(\mathsf{push}(k), t)$, we insert $(t, +1)$ into PSS, use a succ query to compute the pop operation whose value now changes, and add it to S_I. If the new push operation has rank r, the pop operation whose value is affected has rank $r-1$. For $\mathsf{invoke}(\mathsf{pop}(), t)$, we insert $(t, -1)$ into PSS and use a pred query to determine the corresponding push. If the new pop has rank r, use a succ query to find the earliest next operation with rank $r-1$ and push it to the stack (if not at the stack's top already). Note that before this operation, the stack was consistent up to the time at the then top of the stack; invoking such an operation can introduce an earlier inconsistent time, which we place at the stack's top.

For $\mathsf{revoke}(t)$, we remove the corresponding entry from PSS. If the operation is a push, use a succ query to compute the pop operation that previously popped this key and push it to S_I if the element is not already there. When we revoke a pop, chances are that it is the pop with the earliest inconsistent time, in which case we remove t from the top of the stack and calculate the new inconsistent time. For all cases of revoking a pop, if the pop being revoked has rank r, locate the next operation with rank $r-1$ and push it to S_I (if not there already).

After revisions some operations may re-synchronize. We now clean up S_I and report the next inconsistent time. To clean up, check the pop operation at the top of S_I: we use a pred query to determine the value of a pop operation if one were to re-issue it at that time. If the value matches the original value of the pop operation, we discard it from the stack—the operation is no longer inconsistent. [3] Repeat this process until no more element is removed or the stack is empty. Finally we report the time at the top of S_I as the earliest inconsistent time.

In all above operations, the queries succ and pred take $O(\log|\mathcal{R}|/\log\log|\mathcal{R}|)$ per operation. Each operation on S_I takes $O(1)$. Therefore, any revision operation takes amortized $O(\log|\mathcal{R}|/\log\log|\mathcal{R}|)$ time. Since the revision sequence is monotone, each element of S_I can be discarded only once. Thus its cost can be charged to the operation that places it to the stack. ■

[3]This requires an equality check on the keys. Even if the keys don't accept equality, we can match by comparing the time of the corresponding push operations for the keys.

6.3 A Lower-bound

In this section, we show that the stack data structure presented in the previous section is the best possible up to constant factors by proving the following theorem:

Theorem 6.4 *There is a sequence of $m > n$ operations on a retroactive stack where $|\mathcal{R}| = O(n)$ that require $\Omega(\frac{m \log n}{\log \log n})$ time.*

The argument outlined below is a simple reduction from the marked-ancestor problem. The marked-ancestor problem is to maintain a data structure on a fixed rooted tree to support the following operations: *mark*(v) marks the node v, *unmark*(v) unmarks the node v, and *firstmarked*(v) reports the first marked node on the path from v to the root node. In FOCS'98, Alstrup et al. [AHR98] showed that,

Theorem 6.5 (Alstrup-Husfeldt-Rauhe) *In the cell-probe model with word size $O(\log n)$, there is a sequence of $m > n$ operations in the marked ancestor problem on n nodes that requires $\Omega(m \log n / \log \log n)$ time.*

We now briefly describe the reduction. Consider an Euler's tour on the tree. For each node v, let f_v and ℓ_v be the first and last times the tour visits v. Marking v (unmarking resp.) corresponds to invoking (revoking resp.) a pair of push at f_v and pop at ℓ_v. The query *firstmarked*(v) can be answered by considering the result of a pop at f_v[4]. We can support each operation in the marked-ancestor problem using a constant number of retroactive stack operations, concluding the proof.

7 Queues

Consider a *queue* data structure Q with the following operations: $\mathsf{enqueue}(e)$, which puts the element e to the end of Q, and $\mathsf{dequeue}()$, which removes and returns the element at the front of the queue. We prove the following:

Theorem 7.1 *A retroactive queue Q_r with trace $\mathcal{R}$ over times $\mathcal{T}$ can be maintained in $O(|\mathcal{R}|)$ space so that all operations take $O(\log \log |\mathcal{T}|)$ expected amortized time.*

Proof: We maintain two ordered sets indexed on their first component: $E \subseteq \mathcal{T} \times \mathcal{U} \times (\mathcal{T} \cup \{\mathbf{null}\})$ stores the times, values, and corresponding dequeue times of all enqueue operations; $D \subseteq \mathcal{T} \times (\mathcal{T} \cup \{\mathbf{null}\})$ stores the dequeue times and corresponding enqueue times of all dequeue operations. Both sets are kept as ordered-subset structures. Let $\mathsf{lookAt}(t)$ return the time of the enqueue operation at the front of the queue at time t and $\mathbf{null}$ if the queue is empty. Note that $\mathsf{lookAt}(t)$ can be computed in $O(\log \log |\mathcal{T}|)$ by a pred query on D and a succ query on E.

For $\mathsf{invoke}(\mathsf{enqueue}(e), t)$, we insert $(t, e, \mathbf{null})$ into E and set p to the dequeue pair of the enqueue right after t. For $\mathsf{invoke}(\mathsf{dequeue}(), t)$, we insert $(t, \mathsf{lookAt}(t))$ into D and set p to the successor of t in D. In both cases, we update the corresponding pair as necessary.

Revoking an operation removes the corresponding element from D or E depending on the operation. We then update the corresponding pair as necessary. If revoking an enqueue, we set p to the dequeue pair of the enqueue being revoked. If revoking a dequeue, we set p to the dequeue operation right after t.

To report the earliest inconsistent time, we consider p. If p is unset or if the enqueue pair of p is the same as what lookAt returns, the trace is consistent. Otherwise, we report p. ■

8 Ordered Sets

Given an ordered universe of keys K, an ordered set maintains $D \subseteq K$, with the operations:

[4]To restore the trace back to the original state, we need to invoke a pop and subsequently revoke both of them.

- $\mathsf{insert}(k)$: Insert k into D.
- $\mathsf{delete}(k)$: Delete k from D
- $\mathsf{succ}(k)$: Return $\min\{k' \in D : k' \geq k\}$.

It is straightforward to allow data to be associated with keys and to support a corresponding predecessor query, but we leave these out to simplify the exposition. We assume the comparison model for K and under this model an ordered dictionary has a lower bound of $\Omega(\log |D|)$ time for at least some of the operations. A retroactive version cannot do better. We show, however, that we can match the lower bounds within expectation and amortization.

Theorem 8.1 *A retroactive ordered set D_r with trace $\mathcal{R}$ can be maintained in $O(|\mathcal{R}|)$ space such that any revision sequence involving n revised updates (*insert *or* delete*) and m revoked* succ *operations takes $O((m+1)n \log |\mathcal{R}|)$ time.*

Proof: The retroactive version of the dictionary data structure has a natural geometric representation. Consider Figure 1 below representing a set of insert, delete, and succ operations over time.

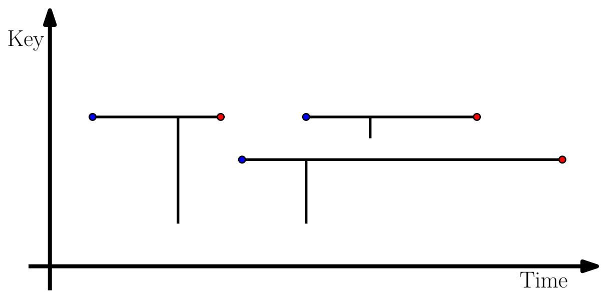

Figure 1: The geometric representation.

Let the vertical axis (y-axis) represent keys and the horizontal axis (x-axis) represent the times $\mathcal{T}$. An instance of a retroactive data structure can be represented as planar segments and points. A horizontal segment $([t_1, t_2], k)$ denotes the lifespan of the key k—it is inserted at t_1 and deleted at t_2. A vertical segment $(t, [k_1, k_2])$ denotes the query $\mathsf{succ}(k_1)$ at time t whose return key is k_2. Insertions and deletions are represented by points at the corresponding keys and times, and we also include points at the top of each vertical segment. As always, we assume that each operation occurs at a unique time (i.e., we can uniquely refer to an operation by its time). Since the standard setup and the geometric representation are equivalent, we will use them interchangeably in subsequent discussions as convenient.

The set of horizontal segments H and set of vertical segments V are stored as two point-location data structures (cf., Section 2). These structures support the queries $\mathsf{above}(k, t)$, which reports the first horizontal segment hit by the ray shooting upward from the point (t, k), and $\mathsf{right}(k, t)$, which reports the first vertical segment hit by the ray shooting rightward from the point (t, k). We also store all points in sets S_k indexed by their key k (y-coordinate). The sets can be accessed using a balanced tree on K, and each set can be stored as a balanced tree on $\mathcal{T}$. The geometric representation is the set $H \cup V \cup (\cup_k S_k)$.

For any revision involving an update (insert or delete) we immediately update the point corresponding to the revision and the horizontal segment it affects. This involves inserting, deleting, extending, or trimming the horizontal segment, and can be done using at most two insertions and deletions on the point location structure. This can leave the geometric interpretation inconsistent by, for example, introducing segment crossings. We cannot, however, afford to fix all these since a single update could create $\Theta(|\mathcal{R}|)$ such inconsistencies. The idea will be to only keep track of the earliest inconsistency for each key.

During a monotone revision sequence on the retroactive structure we therefore allow for the following two kinds of inconsistencies: an *open inconsistency* is a vertical segment for which the top point does not lie on a horizontal segment, and a *cross inconsistency* is a vertical segment that crosses multiple horizontal segments. We associate an open inconsistency with the key of its top point and a cross inconsistency with all the keys for horizontal segments it crosses. During the whole revision sequence we maintain for each key k the earliest inconsistency i_k associated with it, and also keep a priority queue PQ over time of all these earliest inconsistencies. This allows us to track the overall earliest inconsistency. In the discussion below we assume PQ is updated whenever i_k is updated.

When we invoke an insert, or revoke a delete on key k at time t we can insert or extend a horizontal segment, which can either create cross inconsistencies or fix open inconsistencies. If i_k is a open inconsistency we delete it. Because of the monotone revision sequence condition there cannot be any more inconsistencies on k. If i_k is a cross inconsistency, we do nothing (we already know the next inconsistency). If there is no i_k we search for the next cross inconsistency using $\mathsf{right}(k,t)$ and if there is one we insert it as i_k.

When we invoke a delete, or revoke an insert on key k at time t we can delete or trim a horizontal segment, which can create open inconsistencies or fix cross inconsistencies. If i_k is a cross inconsistency on a $\mathsf{succ}(k')$ operation at time t' we delete i_k—there cannot be more inconsistencies on k. If i_k is an open inconsistency we do nothing. If there is no i_k we check for the next open inconsistency in S_k and if there is one we insert it as i_k.

For an $\mathsf{invoke}(\mathsf{succ}(k),t)$ we do an $\mathsf{above}(k,t)$ to find k', add the vertical segment $(t,[k,k'])$, and return k'. No inconsistencies are affected. Consider a $\mathsf{revoke}(t)$ for an succ operation at t. The vertical segment associated with this operation could be involved in up to l inconsistencies where l is the number of revised updates so far in the monotone revision sequence (only these can create inconsistencies). This is bounded by n. For each key k that has an inconsistency at t (can be found in PQ) we need to identify the next inconsistency, if there is one, and update i_k. For cross inconsistencies we search for it with $\mathsf{right}(k,t)$ and and for an open inconsistency we use S_k.

All revisions except revoking a succ operation involve at most a constant number of point location, dictionary, ordered set, or priority queue operations which all take $O(\log|\mathcal{R}|)$ worst case time. The revoke of a succ operations takes at most $\log|\mathcal{R}|$ time per inconsistency for a total of $n\log|\mathcal{R}|$ time. The total time is therefore bounded by $(m+1)n\log|\mathcal{R}|$ total time. The space is bounded by the space required by the two point location structures and the sets S_k. This is all linear in the number of operations in the trace $\mathcal{R}$. ■

9 Priority Queues

In this section, we consider a priority queue data structure. Given a total-ordered universe of keys K, a priority queue maintains a set $PQ \subseteq K$ and supports the following operations:

- $\mathsf{insert}(k)$: insert a key k to the priority queue. We assume $k \notin PQ$.
- $\mathsf{min}()$: return the minimum key of PQ or **null** if PQ is empty. This operation does *not* delete the key.
- $\mathsf{delete}(k)$: delete the key k if existed.

Theorem 9.1 *A retroactive priority queue PQ_r with trace $\mathcal{R}$ over times $\mathcal{T}$ can be maintained in $O(|\mathcal{R}|)$ space so that all operations take $O(\log|\mathcal{R}|)$ expected amortized time.*

Proof: We use the same setup as that of the retroactive ordered sets (Section 8) except that now all vertical segments begin at $y = -\infty$ and correspond to min queries. Recall that an *open inconsistency* is a vertical segment for which the top point does not lie on a horizontal segment, and a *cross inconsistency* is a vertical segment that crosses multiple horizontal segments.

For all keys, we keep their earliest open inconsistent times. But, we maintain only a single cross inconsistency for the least key with a cross inconsistency. This cross consistency must appear before all other cross inconsistencies, because if a vertical segment (query) intersects a horizontal segment of a bigger key, the vertical segment intersects the segment of the smallest key — all vertical segments begin at $y = -\infty$. For this reason, when we revoke a query (min), at most one cross inconsistency has to be updated. Invoking an $\mathsf{insert}(k)$ or revoking a $\mathsf{delete}(k)$ may introduce more cross inconsistency; however, we only have to do a right query if the k is the minimum key with a cross inconsistency. ■

In this setting, we do not need a general point-location data structure: the above query always starts at $y = -\infty$, and the right query deals with a special type of segments—their bottom endpoints are at $-\infty$. In both cases, variants of McCreight's priority search tree [McC85, Wil00] can be used to achieve $O(\log |S|)$ time and $O(|S|)$ space with simple balanced tree operations by storing an additional value (minimum) at each internal node.

We recall that sorting n elements requires $\Omega(n \log n)$ in the comparison model. Since we can use a priority queue to sort, there is a sequence of $O(n)$ operations on a retroactive priority queue that require $\Omega(n \log n)$ time.

10 Discussions

It remains an interesting open problem to see if a retroactive ordered-set data structure can be maintained in $O(\log |\mathcal{R}|)$ time, matching the sorting lower-bound in the comparison model.

References

[AHR98] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *FOCS '98: Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 534–544, Washington, DC, USA, 1998. IEEE Computer Society.

[AV03] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32(6):1488–1508, 2003.

[BCD$^+$02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Lecture Notes in Computer Science*, pages 152–164, 2002.

[Ble07] Guy E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. (submitted to SODA'08), 2007.

[BV07] Guy E. Blelloch and Virginia Vassilevska. Ordered subsets with applications. (submitted to SODA'08), 2007.

[Die89] Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *WADS*, pages 39–46, 1989.

[DIL04] Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 281–290, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[DS87] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th ACM Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

[FW94] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.

[McC85] Edward M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.

[MN90] Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 5(2):215–241, 1990.

[Mor03] Christian Worm Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *SODA '03: Proceedings of the 14th annual ACM-SIAM symposium on discrete algorithms*, pages 618–627, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[Mor06] Christian Worm Mortensen. Fully dynamic orthogonal range reporting on ram. *SIAM J. Comput.*, 35(6):1494–1525, 2006.

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer, 1983.

[PD04] Mihai Pǎatraşcu and Erik D. Demaine. Tight bounds for the partial-sums problem. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 20–29, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[RR96] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph algorithms. *Theoretical Computer Science*, 158(1–2):233–277, 1996.

[WC79] Mark N. Wegman and Larry Carter. New classes and applications of hash functions. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, pages 175–182, 1979.

[Wil85] Dan E. Willard. Reduced memory space for multi-dimensional search trees (extended abstract). In *STACS*, pages 363–374, 1985.

[Wil00] Dan E. Willard. Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3):1030–1049, 2000.