Retroactive Data Structures (extended abstract)

Erik D. Demaine* John Iacono† Stefan Langerman‡

Abstract

We introduce a new data structuring paradigm in which operations can be performed on a data structure not only in the present but also in the past. In this new paradigm, called *retroactive data structures*, the historical sequence of operations performed on the data structure is not fixed. The data structure allows arbitrary insertion and deletion of operations at arbitrary times, subject only to consistency requirements. We initiate the study of retroactive data structures by formally defining the model and its variants. We prove that, unlike persistence, efficient retroactivity is not always achievable, so we go on to present several specific retroactive data structures.

1 Introduction

Suppose that we just discovered that an operation previously performed in a database was erroneous (e.g., from a human mistake), and we need to change the operation. In most existing systems, the only method to support these changes is to rollback the state of the system to before the time in question and then re-execute all of the operations from the modifications to the present. Such processing is wasteful, inefficient, and often unnecessary. In this paper we introduce and develop the notion of *retroactive data structures*, which are data structures that efficiently support modifications to the historical sequence of operations performed on the structure. Such modifications could take the form of retroactively inserting, deleting, or changing one of the operations performed at a given time in the past on the data structure in question.

After defining the model, we show that there is no general efficient transformation from non-retroactive structures into retroactive structures. We then turn to the development of specific retroactive structures. For some classes of data structures (commutative and invertible data structures, and data structures for decomposable search problems), we present general transformations to make data structures efficiently retroactive. For other data structures where the dependency between operations is stronger, efficient retroactivity requires the development of new techniques. In particular, we present a retroactive heap that achieves optimal bounds.

1.1 Comparison to persistence.

The idea of retroactive data structures is related at a high level to the classic notion of persistent data structures, because they both consider the notion of time, but otherwise they differ almost entirely.

A persistent data structure maintains several versions of a data structure, and operations can be performed on one version to produce a new version. In its simplest form, modifications can only be made to the last structure, thus creating a linear relationship amongst the versions. In full persistence [4], an operation can be performed on any past version to create a new version, thus creating a tree structure of versions. Confluently persistent structures [5] allow a new version to be created by merge-like operations on multiple existing structures, and thus the versions form a directed acyclic graph. The data structuring techniques for persistence represent a substantial cost savings over the naïve method of maintaining separate copies of all versions.

The key difference between persistent and retroactive data structures is that, in persistent data structures, each version is treated as an unchangeable archive. Each new version is dependent on the state of existing versions of the structure. However, because existing versions are never changed, the dependence relationship between two versions never changes. The user can view a past state of the structure, but changes in the past state can only occur by forking off a new version from a past state. Thus, the persistence paradigm is useful for maintaining archival versions of a structure, but is inappropriate for when changes must be made directly to the past state of the structure.

In contrast, the retroactive model we define allows changes to be made directly to previous versions. Because of the interdependence of the versions, such a change can radically affect the contents of all later versions. In effect we sever the relationship between time as perceived by a data structure, and time as perceived by the user of the data structure. Operations such as "Insert 42" now become

*MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, USA. edemaine@mit.edu.

†Polytechnic University, 5 MetroTech Center, Brooklyn NY 11201, USA. jiacono@poly.edu.

‡Chargé de recherches du FNRS, Université Libre de Bruxelles, Département d'informatique, ULB CP212, Belgium. Stefan.Langerman@ulb.ac.be.

"Insert at time 10 the operation 'Insert 42' ".

1.2 Motivation. In a real-world environment, large systems processing many transactions are commonplace. In such systems there are many situations where the need arises to alter the historical sequence of operations that were previously performed on the system. We now suggest several applications where a retroactive approach to data structures would help:

Simple Error. Data was entered wrong. The data should be corrected and all secondary effects of the data removed.

Security Breaches. Suppose some operations were discovered to have been maliciously performed by an unauthorized user. It is particularly important in the context of computer security to not only remove the malicious transactions, but also to act as if the malicious operation never occurred. For example, if the intruder modified the password file, not only should we restore that file, but we should also undo logins enabled by this modification.

Tainted Sources. In a situation where data enters a system from various automated devices, if one device is found to have malfunctioned, all of its transactions are invalid over a period of time and must be retroactively removed. For example, in a situation where many temperature readings from various weather stations are reported to a central computer, if one weather station's sensors are discovered to be intermittently malfunctioning, we wish to remove all of the sensor readings from the malfunctioning station because they are no longer reliable. Secondary effects of the readings, such as averages, historical highs and lows, along with weather predictions must be retroactively changed.

Disconnection. Continuing with the weather-station analogy of the previous paragraph, suppose the transmission system for one weather station is damaged, but later the data is recovered. We should then be able to retroactively enter the reports from the weather station, and see the effects of the new data on, for example, the current and past forecasts.

Online Protocols. In a standard client-server model, the server can be seen as holding a data structure, and clients send update or query operations. When the order of the requests is important (e.g., Internet auctions), the users can send a timestamp along with their requests. In all fairness, the server should execute the operations in the order they were sent. If a request is delayed by the network, it should be retroactively executed at the appropriate time in the past.

Settlements. In some cases it is mutually agreed upon by the parties involved to change some past transaction and all of its effects. We cite one example of such a settlement and describe how the traditional method of handing such settlements often fails in today's systems. Suppose you have two charge cards from one company. When a bill comes from one card, you pay the wrong account. Upon realizing the mistake, you call customer service and you reach a settlement in which the payment is transferred into the correct account. Unfortunately, the next month, you are charged a late fee for the late payment of the bill. You call customer service again, and the late fee is removed as per the previous agreement. The next month, interest from the (now removed) late fee appears on the bill. Once again you must call customer service to fix the problem. This sequence of events is typical and results from the system's inability to retroactively change the destination of the payment.

Intentional Manipulation of the Past. In Orwell's *1984* [10], the protagonist's job was to change the past editions of a newspaper to reflect the desires of the government to control the past. Retroactively changing several documents while maintaining consistency has many applications, some more dangerous than others.

Dynamization. Some static algorithms or data structures are constructed by performing on some dynamic data structure a sequence of operations determined by the input. For example, the point-location data structure of Sarnak and Tarjan [13] consists of performing a sequence of insertions and deletions on a persistent binary search tree. Making such data structures retroactive would allow us to dynamically modify the input by retroactively modifying the operation sequence, thus making static algorithms or data structures dynamic.

1.3 Time is not an ordinary dimension. One may think that the problem of retroactive data structures can be solved by adding one more dimension to the structure under consideration. For example, in the case of a min-heap, it would seem at first glance that one could create a two-dimensional variant of a heap, and the problem would be solved. The idea is to assign the key values of the items in the heap to the y axis and use the x axis to represent time. In this representation, each item in the heap is represented by a horizontal

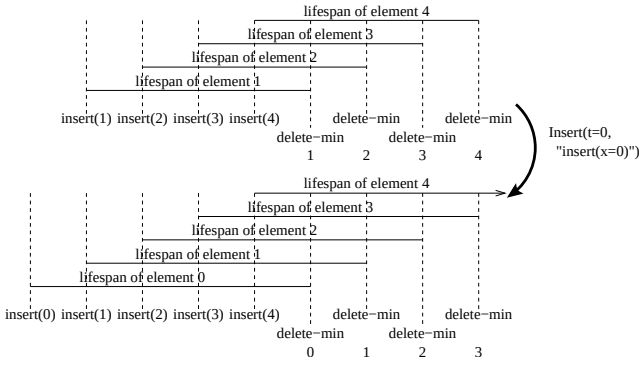

Figure 1: A single insertion of an insertion operation in a retroactive heap data structure can change the outcome of every delete-min operation and the lifespan of every element.

line segment. The left endpoint of this segment represents when an item is inserted into the heap and the right endpoint represents when the item is removed. If the only operations supported by the heap are insert() and delete-min(), then we have the additional property that there are no points below the right endpoint of any segment, because only minimal items are removed from the heap. While this seems to be a clean two-dimensional representation of a heap throughout time, retroactively adding and removing an operation in the heap cannot simply be implemented by adding or removing a line segment. In fact, the endpoints of all the segments could be changed by inserting a single operation, as illustrated in Figure 1.

Thus, while time can be drawn as a spatial dimension, that dimension is special in that complicated dependencies may exist, so that when small changes are made to some part of the diagram, changes may have to be made to the rest of the diagram. Thus, traditional geometric and high-dimensional data structures cannot be used directly to solve most retroactive data-structure problems. New techniques must be introduced to create retroactive data structures without explicitly storing every state of the structure, much like Chan's breakthrough dynamic convex hull algorithm [3], which does not explicitly store the hull.

1.4 Outline. The rest of the paper proceeds as follows. In Section 2, we further develop the model of retroactive data structures and explore the possible variants on the model. Next, Section 3 considers some basic problems about retroactivity, proving separations among the model variations and proving that automatic efficient retroactivity is impossible in general. In Section 4, we present two general transformations to construct an efficient retroactive version of a data structure, one for commutative invertible operations and one for any decomposable search problem. Finally, in Section 5, we discuss specific data structures for which we propose to create efficient retroactive structures. Table 1 in Section 5 gives a partial summary of the results obtained.

2 Definitions

In this section, we define the precise operations that we generally desire from a retroactive data structure.

2.1 Partial Retroactivity. Any data structural problem can be reformulated in the retroactive setting. In general, the data structure involves a sequence of updates and queries made over time. We define the list $U = [u_{t_1}, \ldots, u_{t_m}]$ of updates performed on the data structure, where u_{t_i} is the operation performed at time t_i, and $t_1 < t_2 < \cdots < t_m$.

The data structure is *partially retroactive* if, in addition to supporting updates and queries on the "current state" of the data structure (present time), it supports insertion and deletion of updates at past times as well. In other words, there are two operations of interest:

1. Insert(t, u): insert into U a new update operation u at time t (or before operation u_t if it already exists).
2. Delete(t): delete the past update operation u_t from the sequence U of updates.

Thus, the retroactive versions of standard insert(x) and delete(x) operations are Insert$(t,$ "insert(x)"$)$, Insert$(t,$ "delete(x)"$)$, and Delete(t), where t represents a moment in time. For example, if $t_{i-1} < t \leq t_i$, Insert$(t,$ "insert(x)"$)$ creates a new operation $u =$ insert(x), which inserts a specified element x, and modifies history to suppose that operation u occurred between operations $u_{t_{i-1}}$ and u_{t_i} in the past. Informally, we are traveling back in time to a prior state of the data structure, introducing or preventing an update at that time, and then returning to the present time.

All such retroactive changes on the operational history of the data structure potentially affect all existing operations between the time of modification and the present time. Particularly interesting is the (common) case in which local changes propagate in effect to produce radically different perceived states of the data structure. The challenge is to realize these perceived differences extremely efficiently by implicit representations.

2.2 Full Retroactivity. The definitions presented above capture only a partial notion of retroactivity: the ability to insert or delete update operations in the past, and to view the effects at the present time. Informally, we can travel back in time to modify the past, but we cannot directly observe the past. A data structure is *fully retroactive* if, in addition to allowing updates in the past, it can answer queries about the past. In some sense, this can be seen as making a partially retroactive version of a persistent version of the original structure. Thus, the standard search(x) operation, which finds an element x in the data structure, becomes Query$(t,$ "search(x)"$)$, which finds the element x in the state of the data structure at time t.

2.3 Running Times. When expressing the running times of retroactive data structures, we will use m for the total number of updates performed in the structure (retroactive or not), r for the number of updates before which the retroactive operation is to be performed (i.e., $t_{m-r} < t \leq t_{m-r+1}$), and n for the maximum number of elements present in the structure at any single time. Most running times in this paper are expressed in terms of m, but in many cases, it is possible to improve the data structures to express the running time of the operations in terms of n and r, so that retroactive operations performed at a time closer to present time are executed faster. These improvements are not detailed here due to space constraints.

2.4 Consistency. We assume that only valid retroactive operations are performed. For example, in a retroactive dictionary, a delete(k) operation for a key k must always appear after a corresponding insert(k) in the list U; and in a retroactive stack, the number of push() operations is always larger than the number of pop() operations for any prefix of U. The retroactive data structures we describe in this paper will not check the validity of recursive updates, but it is often easy to create a data structure to verify the validity of a recursive operation.

3 General Theory

The goal of this research is to design retroactive structures for abstract data types with performance similar to their non-retroactive counterparts. This section considers some of the most general problems about when this is possible.

Unless stated otherwise, the results will apply to the RAM model of computation (or Real-RAM when real values are used), and sometimes to the pointer model [18] as well. Lower bounds will be given in the straight-line-program model [16] or in the cell-probe model [20].

3.1 Automatic Retroactivity. A natural question in this area is the following: is there a general technique for converting any data structure in (e.g.) the pointer-machine model into an efficient partially retroactive data structure? Such a general technique would nicely complement existing methods for making data structures persistent [4, 5]. As described in the introduction, retroactivity is fundamentally different from persistence, and known techniques do not apply here.

One simple approach to this general problem is the *rollback method*. Here we store as auxiliary information all changes to the data structure made by each operation in such a way that every change could be reversed. For operations on the present, the data structure proceeds as normal, modulo some extra logging. When the user requests a retroactive change at a past time t with $t_{m-r} < t < t_{m-r+1}$, the data structure rolls back all changes made by operations $u_m, \ldots, u_{m-r+1}$, then applies the retroactive change as if it were the present, and finally re-performs all operations $u_{m-r+1}, \ldots, u_m$. Notice that these re-performances may act differently from how the operations were performed before, depending on the retroactive change. Because the changes made to the data structure are bounded by the time taken by the operations, a straightforward analysis proves the following theorem:

THEOREM 3.1. *Given any data structure that performs a collection of operations each in worst case $T(n)$ time, there is a corresponding retroactive data structure that supports the same operations in $O(T(n))$ time, and supports retroactive versions of those operations in $O(rT(n))$ time.*

The rollback method is widely used in database management systems (see e.g. [11]) and robust file systems for concurrency control and crash recovery. It has also been studied in the data structures literature under the name of unlimited undo or *backtracking* [9, 19].

Of course, this result, and its extension to operations with nonuniform costs, is far too inefficient for applications where r can be n or even larger—the total number of operations performed on the data structure. The goal would be to reduce the dependence on r in the running time of retroactive operations. We show that this is not possible in the straight-line-program model of computation:

THEOREM 3.2. *There exists a data structure in the straight-line-program model, supporting $O(1)$ time update operations, but the (partially) retroactive insertions of those operations require $\Omega(r)$ time, worst case or amortized.*

Proof: The data structure maintains two values X and Y, initially 0, and supports operations addX(c) and addY(c) which add the value c to the value X or Y, and mulXY(), which multiplies Y by X, and store the resulting value in Y. Queries return the value of Y.

Consider the sequence of $m = 2n + 1$ operations: [addY(a_n), mulXY(), addY(a_{n-1}), mulXY(), ..., mulXY(), addY(a_0)]. At the end of the sequence, $X = 0$ and $Y = 0$. We then retroactively insert the operation "addX(x)" at the very beginning of the sequence. The value of Y is now $a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$, which is a polynomial of degree n in x with arbitrary coefficients. By Motzkin's theorem [16], the computation of that polynomial for a given value of x requires $\Omega(n)$ multiplications, regardless of how much preprocessing time or space is used. Thus the retroactive insertion of the addX(x) operation requires that many multiplications. Because the retroactive modification can be repeated an arbitrary number of times, and each modification will have the same worst-case lower bound, the lower bound also applies to amortized data structures. □

The straight-line-program model counts only the number of arithmetic operations performed by a program. Our lower bound thus holds in more general models of computation such as the Real-RAM model or the algebraic-computation-tree model.

3.2 From Partial to Full Retroactivity. A natural general question about the two versions of retroactivity is whether partial retroactivity is indeed easier to support than full retroactivity. In other words, is it easier to answer queries only about the present? We first give a partial answer:

THEOREM 3.3. *In the cell-probe model, there exists a data structure supporting partially retroactive updates in $O(1)$ time, but fully retroactive queries of the past require $\Omega(\log n / \log \log n)$ time.*

Proof: The data structure is for the following problem: maintain a set of numbers subject to the update insert(c), which adds a number c to the set, and the query sum() which reports the sum of all of the numbers. For this problem, the only retroactive update operations are Insert(t, "insert(c)") and Delete(t), whose effects on queries about the present are to add or subtract a number to the current aggregate. Thus, a simple data structure solves partially retroactive updates in $O(1)$ time per operation. In contrast, to support queries at arbitrary times, we need to remember the order of the update operations and support arbitrary prefix sums. Thus, we obtain a lower bound of $\Omega(\log n / \log\log n)$ in the cell-probe model by a reduction from dynamic prefix sums [7]. □

On the other hand, we can show that it is always possible, at some cost, to convert a partially retroactive data structure into a fully retroactive one:

THEOREM 3.4. *Any partially retroactive data structure in the pointer-machine model with constant indegree, supporting $T(m)$-time retroactive updates and $Q(m)$-time queries about the present can be transformed into a fully retroactive data structure with amortized $O(\sqrt{m}\,T(m))$-time retroactive updates and $O(\sqrt{m}\,T(m) + Q(m))$-time fully retroactive queries using $O(mT(m))$ space.*

Proof: We define $\sqrt{m}$ *checkpoints* $t_1, \ldots, t_{\sqrt{m}}$ such that at most $2\sqrt{m}$ operations have occurred between consecutive checkpoints, and maintain $\sqrt{m}$ versions of the partially retroactive data structure $D_1, \ldots, D_{\sqrt{m}}$, where the structure D_i only contains the updates that occurred before time t_i. When a retroactive update is performed for time t, we perform the update on all structures D_i such that $t_i > t$. When a retroactive query is made at time t, we find the largest i such that $t \geq t_i$, and perform on D_i all updates that occurred between times t_i and t, and finally perform the query on the resulting structure.

In order to reduce space, we use persistent data structures [4]. Given a sequence of m operations, we perform the sequence on a fully persistent version of the partially retroactive data structure, and keep a pointer D_i to the version obtained after the first $i\sqrt{m}$ operations for $i = 1, \ldots, \sqrt{m}$. The retroactive updates branch off a new version of the data structure for each of the modified D_i. After $\sqrt{m}/2$ retroactive updates have been performed, we rebuild the entire structure in time $O(mT(m))$. This will ensure that the number of updates between any two checkpoints is always between $\sqrt{m}/2$ and $3\sqrt{m}/2$. The resulting data structure will have the claimed running times. The fully persistent version of the partially retroactive data structure after a rebuild will use at most $O(mT(m))$ space because it can use at most one unit of space for each computational step. The data structure will perform at most $\sqrt{m}/2$ retroactive updates between two rebuilds, each using at most $O(\sqrt{m}\,T(m))$ time and extra space, and so the space used by the fully retroactive data structure will never exceed $O(mT(m))$. □

4 Transformable Structures

In this section, we present some general transformations to make data structures partially or fully retroactive for several easy classes of problems.

4.1 Commutative Operations. To highlight the difficult case of nonlocal effects, we define the notion of *commutative operations*. A set of operation types is *commutative* if the state of the data structure resulting from a sequence of operations is independent of the order of those operations.

If a data structure has a commutative set of operations, performing an operation at any point in the past has the same effect as performing it in the present, so we have:

LEMMA 4.1. *Any data structure supporting a commutative set of operations allows the retroactive insertion of operations in the past (and queries in the present) at no additional asymptotic cost.*

We say that a set of operations is *invertible* if, for every operation u, there is another operation u' that negates the effects of operation u, that is, the sequence of operations $[u, u']$ doesn't change the state of the data structure.

LEMMA 4.2. *Any data structure supporting a commutative and invertible set of operations can be made partially retroactive at no additional asymptotic cost.*

For example, a data structure for *searchable dynamic partial sums* [12] maintains an array $A[1..n]$ of values, where sum(i) returns the sum of the first i elements of the array, search(j) returns the smallest i such that sum(i) $\geq j$, and update(i, c) adds the value c to $A[i]$. The state of the data structure at the present time is clearly independent of the order of the update operations, so it is commutative. Any operation update(i, c) is negated by the operation update($i, -c$), so the updates are also invertible, and so any data structure for searchable dynamic partial sums is automatically partially retroactive.

An important class of commutative data structures are for *searching problems*. The goal is to maintain a set S of objects under insertion and deletion operations, so that we can efficiently answer queries $Q(x, S)$ that ask some relation of a new object x with the set S. Because a set S is by definition unordered, the set of operations for a searching problem is commutative, given that the deletion of an object is always performed after its insertion. As long as the retroactive updates do not violate this consistency condition, we have:

LEMMA 4.3. *Any data structure for a searching problem can be made partially retroactive at no additional asymptotic cost.*

For example, dictionary structures, but also dynamic convex hull or planar width data structures can be stated as searching problems and are thus automatically partially retroactive. Note that these results can also be combined with Theorem 3.4 to obtain fully retroactive data structures.

4.2 Decomposable Searching Problems. A *searching problem* maintains a set S of objects subject to queries $Q(x, S)$ that ask some relation of a new object x with the set S. We already saw in Lemma 4.3 that data structures for searching problems are automatically partially retroactive. A searching problem is *decomposable* if there is a binary operator $\square$ (computable in constant time) such that $Q(x, A \cup B) = \square(Q(x, A), Q(x, B))$. Decomposable searching problems have been studied extensively by Bentley and Saxe [2]. In particular, they show how to transform a static data structure for such a problem into an efficient dynamic one. In this section, we show that data structures for decomposable searching problems can also be made fully retroactive.

THEOREM 4.1. *Any data structure for a decomposable searching problem supporting insertions, deletions, and queries in time $T(n)$ and space $S(n)$ can be transformed into a fully retroactive data structure with all operations taking time $O(T(m))$ if $T(m) = \Omega(n^\epsilon)$ for some $\epsilon > 0$, or $O(T(m) \log m)$ otherwise. The space used is $O(S(m) \log m)$.*

Proof: Every element that was ever inserted in the data structure can be represented by a segment on the time line, between its insertion time and its deletion time (or present time if it wasn't deleted). We maintain a segment tree [1], which is a balanced binary tree where the leaves correspond to the elementary intervals between consecutive endpoints of the segments, and internal nodes correspond to the union of the intervals of their children. Each segment is thus represented as the union of $O(\log m)$ intervals, each represented by one node of the tree, and each node of the tree will contain the set of segments it represents. For each node, we maintain that set in a data structure supporting the desired queries. Each retroactive update affects at most $O(\log m)$ of those data structures. Given a point t on the timeline, the set of segments containing that point can be expressed as the union of $O(\log m)$ sets from as many nodes. For a retroactive query Query(t, x), we query x in each of the $O(\log m)$ sets and compose the global result using the $\square$ operator. If $T(m) = \Omega(n^\epsilon)$, then the query and update times for a retroactive operation form a geometric progression and the total time is $O(T(n))$, otherwise, the total time is $O(T(m) \log m)$. $\square$

For example, dictionaries, dynamic point location, and nearest-neighbor query data structures solve decomposable searching problems thus can be made fully retroactive. Of course, in many cases, it will be possible to improve the fully retroactive data structures obtained through the application of Theorem 4.1. For example, any comparison-based dictionary where only exact search queries are performed can be made fully retroactive by storing with each key the times at which it was present in the structure. The resulting data structure will use $O(m)$ space and all operations can be performed in $O(\log m)$ time, a $\log m$ factor improvement in both time and space over the straightforward application of Theorem 4.1.

In other cases, though, improving upon the structures obtained from Theorem 4.1 seems rather difficult, as is the case for example with the dictionary problem allowing predecessor and successor queries. Indeed, we can view it as a geometric problem, in which we maintain a set of horizontal line segments, where the y coordinate of each line segment is the element's key and the x extent of the line segment is the element's lifetime. A faster retroactive data structure would immediately result in a faster data structure for dynamic planar point location for orthogonal regions, which may also play a role in general dynamic planar point location. In fact, this retroactive approach is hinted at as a research direction for dynamic planar point location by Snoeyink [15, p. 566].

5 Maintaining the Timeline

We showed in Theorem 3.2 in Section 3.1 that no general technique can turn every data structure into an efficient retroactive counterpart. This suggests that in order to obtain efficient data structures, we need to study special cases separately. In this section, we show how to construct retroactive data structures by maintaining a structure on top of the sequence U of update operations (the timeline). Table 1 gives a partial summary of our results.

In the following, we assume that the sequence U is maintained in a doubly linked list, and that when a retroactive operation is performed at time t, a pointer to the operation following time t in U is provided (such a pointer could for example have been stored during a previous operation). In the case where the pointer is not provided, it could easily

Data Structure	Partially Retroactive	Fully Retroactive
Dictionary (exact)	$O(\log m)$	$O(\log m)$
Dictionary (successor)	$O(\log m)$	$O(\log^2 m)$
Queue	$O(1)$	$O(\log m)$
Stack	$O(\log m)$	$O(\log m)$
DEQUE	$O(\log m)$	$O(\log m)$
Union/Find	$O(\log m)$	$O(\log m)$
Priority Queue	$O(\log m)$	$O(\sqrt{m} \log m)$

Table 1: Running times for retroactive versions of a few common data structures. m is the number of operations.

be found in $O(\log m)$ time by maintaining a binary search tree indexed by time on top of U.

5.1 Queues. A queue supports two update operations, enqueue(x) and dequeue() and two query operations, front() which returns the next element to be dequeued, and back() which returns the last element enqueued. Here we describe two data structure, one partially retroactive and one fully retroactive, that thus support the update operations Insert(t, "enqueue(x)"), Insert(t, "dequeue()"), Delete(t), and queries, Query(t, "front()"), and Query(t, "back()"). The partially retroactive data structure will only allow queries at the present time, that is, at time $t = 0$.

LEMMA 5.1. *There exists a partially retroactive queue data structure with all retroactive updates and present-time queries taking $O(1)$ time.*

Proof: The data structure maintains the enqueue operations in a doubly linked list, and two pointers: B will point to the last enqueued element in the sequence, and F to the next element to be dequeued. When an enqueue is retroactively inserted, if it occurs before the operation pointed to by F, we move that pointer to its predecessor. When an enqueue is removed, if it occurs before the operation pointed by F, we move that pointer to its successor. When a dequeue, retroactive or not, is performed, we move the front pointer to its successor, and when a dequeue is removed, we move the front pointer to its predecessor. The B pointer is only updated when we add an enqueue operation at the end of the list. The front() and back() operations return the items pointed by F and B, respectively. □

LEMMA 5.2. *There exists a fully retroactive queue data structure with all retroactive operations taking time $O(\log m)$ and present-time operations taking $O(1)$ time.*

Proof: We maintain two order-statistic trees T_e and T_d. The tree T_e stores the enqueue(x) operations sorted by time, and the T_d stores the dequeue() operations, sorted by time. The update operations can then be implemented directly in time $O(\log m)$, where m is the size of the operation sequence currently stored.

The Query(t, "front()") operation is implemented by querying T_d to determine the number d of dequeue() operations performed at or before time t. The operation then returns the item in T_e with time rank $d + 1$. The Query(t, "back()") operation uses t_e to determine the number e of enqueue() operations that were performed at or before time t, and simply returns the item in T_e with time rank e. Thus, both queries can executed in time $O(\log m)$.

Using balanced search trees supporting updates in worst-case constant-time [6], and by maintaining pointers into the trees to the current front and back of the queues, updates and queries at the current time can be supported in $O(1)$ time. Thus queues may be made efficiently fully retroactive without changing their asymptotic run-times. □

5.2 Doubly Ended Queues. A doubly ended queue (deque) maintains a list of elements and supports four update operations: pushL(x), popL() which insert or delete an element at the left endpoint of the list, pushR(x), popR(), which insert or delete an element at the right endpoint of the list, and two query operations left() and right() that return the leftmost or rightmost element in the list. The deque generalizes both the queue and the stack.

THEOREM 5.1. *There exists a fully retroactive DEQUE data structure with all retroactive operations taking time $O(\log m)$ and present-time operations taking $O(1)$ time.*

Proof: In a standard implementation of a DEQUE in an array A, we initialize variables $L = 1$ and $R = 0$. Then a pushR(x) operation increments R and places x in $A[R]$, popR() decrements R, pushL(x) decrements L and places x in $A[L]$, and popL() increments L. The operation left() returns $A[L]$ and right() returns $A[R]$.

In our retroactive implementation of a DEQUE, we also maintain L and R: if we maintain all pushR(x) and popR() operations in a linked list U_R and associate a weight of $+1$ to each pushR(x) operation and a weight of -1 to each popR(), then R at time t can be calculated as a weighted sum of a prefix of the list up to time t. The same can be done for L, maintaining the list U_L, and reversing the weights.

The values of the sums for all prefixes of U_R can be maintained in the modified (a, b)-tree of [6] with the elements of the list as leaves. Every node of the tree will contain a value r, and the sum of the r values along a path to a leaf will compute the sum of the prefix of U_R up to that element. After inserting an element with weight c in the list and in the tree, we set the r value in the leaf to c and walk along the path to the root, and add c to the r of all right siblings along the path. Deletions are processed symmetrically.

Finally, we have to describe how to extract $A[i]$ from the data structure, where $i = R$ at time t. For this, we augment each node of the tree with two values containing the minimum and maximum prefix sum values for all the leaves in its subtree. Note that these values can also be maintained after insertions and deletions by adding c to them whenever c is added to the r value of the same node, and updating them if an insertion occurs in their subtree.

To find the contents of $A[i]$ at time t, we find the last time $t' \leq t$ when R had value i. This can be done by finding the last operation in U_R before time t, walking up the tree, and walking back down the rightmost subtree for which i is between the minimum and maximum value. The same is done for U_L. □

5.3 Union-Find. A union-find data structure [17] maintains an equivalence relation on a set S of distinct elements,

that is, a partition of S into disjoint subsets (equivalence classes). The operation create(a) creates a new element a in S, with its own equivalence class, union(a, b) merges the two sets that contain a and b, and find(a) returns a unique representative element for the class of a. Note that the representative might be different after each update, so the main use of find(a) is two determine if two elements are in the same class. The Union-Find structure can be made fully retroactive, but to simplify the discussion, we replace the find(a) operation by a sameset(a, b) operation which determines if a and b are in the same equivalence class.

THEOREM 5.2. *There exists a fully retroactive Union-SameSet data structure supporting all operations in $O(\log m)$ time.*

Proof: The equivalence relation can be represented by a forest, where each equivalence class corresponds to a tree in the forest. The create(a) operation constructs a new tree in the forest with a unique node a, sameset(a, b) determines if the root of the trees of a and b are the same, and union(a, b) assumes a and b are not in the same tree, sets b as the root of the tree that contains it, and creates an edge between a and b. Such a forest can be maintained in $O(\log m)$ time per operation using the link-cut trees of Sleator and Tarjan [14], which maintain a forest and support the creation and deletion of nodes, edges, and the changing of the root of a tree.

In order to support retroactive operations, we modify the above structure by adding to each edge the time at which it was created. The link-cut tree structure also allows to find the maximum edge value on a path between two nodes. To determine whether two nodes are in the same set at time t, we just have to verify that the maximum edge time on the path from a to b is no larger than t. □

5.4 Priority Queues. More sophisticated than queues, stacks and deques is the *priority queue*, which supports operations insert(k), which inserts an element with key value k, delete-min() which deletes the element with smallest key, and the query find-min() which reports the current minimum key element. The delete-min() operation is particularly interesting here because of its dependence on all operations in the past: which element gets deleted depends on the set of elements when the operation is executed. More precisely, it is delete-min() that makes the set of operations non-commutative.

Priority queues seem substantially more challenging than queues and stacks. Figure 1 shows an example of the major nonlocal effects caused by a minor modification to the past in a priority queue. In particular, in this example, the lifetime of all elements change because of a single Insert$(t,$ "insert(k)") operation. Such cascading effects need to be succinctly represented in order to avoid the cost inherent in any explicit maintenance of element lifetimes.

Without loss of generality, we assume that all key values inserted in the structure are distinct. t_k will be used to denote the insertion time of key k, and d_k its deletion time. We write Q_t for the set of elements contained in the priority queue at time t, and so Q_0 is the set of elements in the queue in the present time. Let $I_{\geq t} = \{k | t_k \geq t\}$ be the set of keys inserted after time t, and $D_{\geq t} = \{k \notin Q_0 | d_k \geq t\}$ be the set of keys deleted after time t.

In order to construct a retroactive priority queue, we need to learn more about the structure of the problem. For this, we represent a sequence of updates by a planar figure where the x axis represents time, and the y axis represents key values. In this representation, each item k in the heap is represented by a horizontal line segment. The left endpoint (t_k, k) of this segment represents when an item is inserted into the heap and the right endpoint (d_k, k) represents when the item is removed. Similarly, a delete-min() operation is represented by a vertical ray shooting from $y = -\infty$ and stopping at the intersection with the horizontal segment representing the element it deletes. Thus, insert(k) operations paired with their corresponding delete-min() are together represented by upside-down "L" shapes, and no two "L" intersect, while elements still in the structure at present time (i.e. in Q_0) are represented by horizontal rays. See Figure 2.

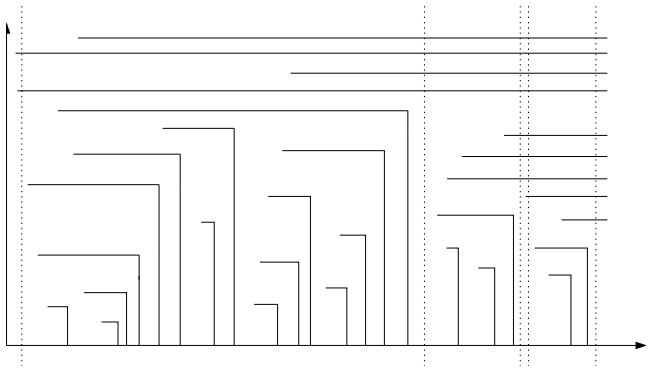

Figure 2: The "L" representation of a sequence of operations.

One obvious invariant of a priority queue data structure is that the number $|Q_0|$ of elements present in the queue is always equal to the number of inserts minus the number of delete-min operations. Thus, when we add an operation $u =$ "insert(k)" at time t in the past, one element will have to be added in Q_0. There are two possibilities: if the element k is not deleted between time t and present time, k can just be added to Q_0. Otherwise, the element k is deleted by some operation $u' =$ "delete-min()", but then the element that was supposed to be deleted by u' will stay in the structure a little longer until it is deleted by some other delete-min() operation and so on. So the insertion of operation u causes a cascade of changes which is depicted in Figure 3.

LEMMA 5.3. *After an operation Insert$(t,$ "insert(k)"), the*

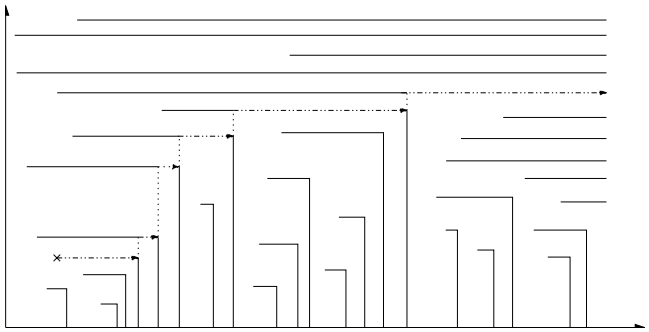

Figure 3: The Insert(t, "insert(k)") operation causes a cascade of changes of deletion times, and one insertion in Q_0.

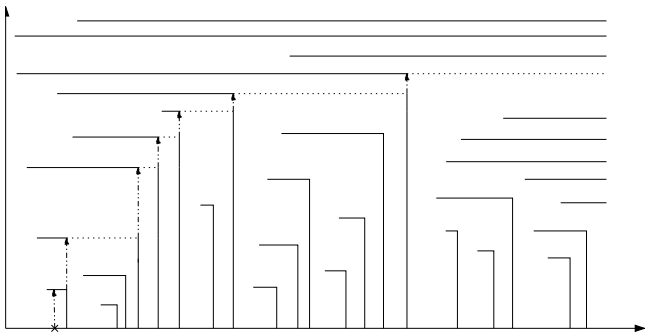

Figure 4: The Insert(t, "delete-min()") operation causes a cascade of changes of deletion times, and one deletion in Q_0.

element to be inserted in Q_0 is

$$\max(k, \max_{k' \in D_{\geq t}} k')$$

Proof: As discussed above, the retroactive insertion will cause several elements to extend the time they are present in the structure. Consider the chain of keys $k < k_1 < k_2 < \cdots < k_\ell$ whose life in the structure is extended. After the retroactive update, the extended pieces of horizontal segments are from (t, k) to (d_{k_1}, k), from (d_{k_i}, k_i) to $(d_{k_{i+1}}, k_i)$ for $i = 1, \ldots, \ell - 1$, and finally from (d_{k_ℓ}, k_ℓ) to $(0, k_\ell)$. They form a nondecreasing step function which, by construction, is not properly intersected by any of the (updated) vertical rays. The key that will be added to Q_0 at the end of the retroactive update is k_ℓ. Suppose there is a key $\hat{k}$ larger than k_ℓ in $D_{\geq t}$. This implies that $(d_{\hat{k}}, \hat{k})$ is above every segment of the step function. But then, the vertical ray from that point intersects the step function, which is a contradiction. In the particular case where k is never deleted, the step function is just one horizontal segment and the same argument holds. □

Note that removing a delete-min() operation has the same effect as re-inserting the element that was being deleted at the time of the deletion. So we have:

COROLLARY 5.1. *After an operation Delete(t), where the operation at time t is "delete-min()", the element to be inserted in Q_0 is*

$$\max_{k' \in D_{\geq t}} k'$$

Because $D_{\geq t}$ can change for many values of t each time an operation is performed, it would be quite difficult to maintain explicitly. The next lemma will allow us to avoid this task. We say that there is a *bridge* at time t if $Q_t \subseteq Q_0$. Bridges are displayed as dotted vertical lines in Figure 2.

LEMMA 5.4. *Let t' be the last bridge before t, then*

$$\max_{k' \in D_{\geq t}} k' = \max_{k' \in I_{\geq t'} - Q_0} k'$$

Proof: By definition of $D_{\geq t}$, any key k' in $D_{\geq t}$ is not in Q_0. If the same k' was inserted before time t', then $k' \in Q_{t'}$, but this would contradict the fact that t' is a bridge, and so $k' \in I_{\geq t'} - Q_0$. This shows that $D_{\geq t} \subseteq I_{\geq t'} - Q_0$, and so

$$\max_{k' \in D_{\geq t}} k' \leq \max_{k' \in I_{\geq t'} - Q_0} k'$$

Let $\hat{k} = \max_{k' \in I_{\geq t'} - Q_0} k'$, and suppose $\hat{k} > \max_{k' \in D_{\geq t}} k'$. This implies $\hat{k} \notin D_{\geq t}$, and so $t' < d_{\hat{k}} < t$. Because t' was the last bridge before time t, $d_{\hat{k}}$ cannot be a bridge, and so there is another key $k'' \in Q_{d_{\hat{k}}} - Q_0 \subseteq I_{\geq t'} - Q_0$, and $k'' > \hat{k}$ otherwise k'' would be deleted instead of $\hat{k}$. But this contradicts that $\hat{k}$ was maximum. □

We next study the effect of adding an operation u = "delete-min()" at time t in the past. In that case, one element will have to be removed from Q_0. Again, this operation will have a cascading effect: if it is not in Q_0, the key k that will be deleted by operation u was supposed to be deleted by the operation u' at time d_k, but as k is being deleted at time t by u, the operation u' will delete the next key up, and so on. See Figure 4.

LEMMA 5.5. *After an operation Insert(t, "delete-min()"), the element to be removed from Q_0 is*

$$\min_{k \in Q_{t'}} k$$

where t' is the first bridge after time t.

Proof: Consider the chain of keys $k_1 < k_2 < \cdots < k_\ell < k$ whose life in the structure is shortened, with $k_i \in D_{\geq t}$ and $k \in Q_0$. After the retroactive update, the shortened pieces of horizontal segments are from (t, k_1) to (d_{k_1}, k_1), from $(d_{k_{i-1}}, k_i)$ to (d_{k_i}, k_i) for $i = 2, \ldots, \ell$, and finally from (d_{k_ℓ}, k) to $(0, k)$. First, it must be clear that there is a bridge at d_{k_ℓ} because there is no key smaller than k in $Q_{d_{k_\ell}}$, and all keys larger than k in $Q_{d_{k_\ell}}$ are also in Q_0 because $k \in Q_0$. So we just have to show that there is no bridge t'' between times t and d_{k_ℓ}. For this we observe that the shortened segments

at times $t'' \in [t, d_{k_t})$ form a step function, and that none of the keys k_i corresponding to the steps are in Q_0, but they are in $Q_{t''}$. □

Because removing an "insert(k)" operation from time t has the same effect as adding a "delete-min()" operation directly after it, we also have:

COROLLARY 5.2. *After an operation Delete(t) where the operation at time t is u_t = "insert(k)", the element to be removed from Q_0 is k if $k \in Q_0$; otherwise, it is*

$$\min_{k' \in Q_{t'}} k'$$

where t' is the first bridge after time t.

Again, because we don't explicitly maintain Q_t for all t, we ease the computation by using that, if t' is a bridge, then $Q_{t'} = I_{\leq t'} \cap Q_0$.

THEOREM 5.3. *There exists a partially retroactive priority queue data structure with all retroactive updates taking time $O(\log m)$ and present-time queries taking $O(1)$ time.*

Proof: The history of all update operations is maintained in a doubly linked list, and the data structure will also explicitly maintain the set Q_0 in a binary search tree, and associating with each key a pointer to its insert operation in the list. After each retroactive update, an element will be inserted or deleted from Q_0 according to the rules described in the lemmas above. In order to decide which element to insert or delete, we need to be able to perform two types of operations:

A. find the last bridge before t or the first bridge after t;

B. find the maximum key in $I_{\geq t'} - Q_0$ or the minimum key in $I_{\leq t'} \cap Q_0$.

If we maintain the list of updates, assigning a weight of 0 to insert(k) operations with $k \in Q_0$, $+1$ to insert(k) with $k \notin Q_0$, and -1 to delete-min() operations, every bridge corresponds to a prefix with sum 0. So, using the data structure used in Theorem 5.1, we can answer the queries of type A in $O(\log m)$ time. Because every retroactive update adds or deletes at most one element from Q_0, only one weight change has to be performed in the structure, which also takes $O(\log m)$.

If we maintain the list of insertions augmented by the modified (a, b)-tree of [6], and store in each internal node the maximum of all keys in its subtree which are absent in Q_0, we can easily find the maximum key in $I_{\geq t'} - Q_0$ in $O(\log m)$ time by walking down the tree. The minimum key in $I_{\leq t'} \cap Q_0$ can also be maintained if we store in every internal node of the tree the minimum of all keys in its subtree which are in Q_0. Those values can be maintained in $O(\log m)$ time per retroactive update because each update changes at most one element of Q_0. □

Acknowledgments. We thank Michael Bender, Prosenjit Bose, Jean Cardinal, Alejandro López-Ortiz, Ian Munro, and the anonymous referees for helpful discussions and comments. In particular, López-Ortiz [8] considered some related notions.

References

[1] J. Bentley. Algorithms for Klee's rectangle problems. unpublished manuscript, Dept. of Computer Science, Carnegie-Mellon University, 1977.

[2] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformations. *J. Algorithms*, 1:301–358, 1980.

[3] T. M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *J. ACM*, 48:1–12, 2001.

[4] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[5] A. Fiat and H. Kaplan. Making data structures confluently persistent. In *Proc. 12th Ann. Symp. Discrete Algorithms*, pages 537–546, Washington, DC, January 2001.

[6] R. Fleischer. A simple balanced search tree with $O(1)$ worst-case update time. *International Journal of Foundations of Computer Science*, 7(2):137–149, 1996.

[7] M. L. Fredman and M. E. Saks. The cell probe complexity of dynamic data structures. In *Proc. of the 21st ACM Symposium on Theory of Computing*, pages 345–354, May 1989.

[8] A. López-Ortiz. Generalized infinite undo and speculative user interfaces. Tech. Rep. CS-2003-33, U. Waterloo, 2003.

[9] H. Mannila and E. Ukkonen. The set union problem with backtracking. In *Proc. 13th Int. Colloq. Automata, Languages and Programming*, LNCS 226, pages 236–243, 1986.

[10] G. Orwell. *1984*. Signet Classic, 1949.

[11] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2002.

[12] R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. In *Proc. 7th Workshop on Algorithms and Data Structures*, LNCS 2125, pages 426–437, 2001.

[13] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.

[14] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–381, 1983.

[15] J. Snoeyink. Point location. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 30, pages 559–574. CRC Press LLC, Boca Raton, FL, 1997.

[16] V. Strassen. Algebraic complexity theory. In *Handbook of Theoretical Computer Science*, volume A, chapter 11, pages 633–672. MIT Press, 1990.

[17] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, 1975.

[18] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci.*, 18:110–127, 1979.

[19] J. Westbrook and R. E. Tarjan. Amortized analysis of algorithms for set union with backtracking. *SIAM J. Comput.*, 18:1–11, 1989.

[20] A. C. Yao. Should tables be sorted? *J. ACM*, 28(3):615–628, 1981.