Polylogarithmic Fully Retroactive Priority Queues via Hierarchical Checkpointing

Erik D. Demaine, Tim Kaler, Quanquan Liu, Aaron Sidford, Adam Yedidia

MIT CSAIL, Cambridge, Massachusetts

Abstract. Since the introduction of retroactive data structures at SODA 2004 [1], a major open question has been the difference between partial retroactivity (where updates can be made in the past) and full retroactivity (where queries can also be made in the past). In particular, for priority queues, partial retroactivity is possible in $O(\log m)$ time per operation on a m-operation timeline, but the best previously known fully retroactive priority queue has cost $\Theta(\sqrt{m}\log m)$ time per operation.

We address this open problem by providing a general logarithmic-overhead transformation from partial to full retroactivity called "hierarchical checkpointing," provided that the given data structure is "time-fusible" (multiple structures with disjoint timespans can be fused into a timeline supporting queries of the present). As an application, we construct a fully retroactive priority queue which can insert an element, delete the minimum element, and find the minimum element, at any point in time, in $O(\log^2 m)$ amortized time per update and $O(\log^2 m \log\log m)$ time per query, using $O(m \log m)$ space. Our data structure also supports the operation of determining the time at which an element was deleted in $O(\log^2 m)$ time.

1 Introduction

Retroactivity. We can think of a data structure as being defined by a sequence of updates $u_1, u_2, \ldots, u_m$ applied to its initial (empty) state. Traditional data structures "live in the present" in the sense that the user can only append updates to this sequence, and ask queries about the final state of the data structure resulting from the entire update sequence. ***Retroactive data structures***, introduced at SODA 2004 [1], allow for updates to be inserted or deleted in the middle of the sequence, instead of just the end. Effectively, this feature enables the user to travel back in time and make a retroactive change to the data structure (similar to the movie *Back to the Future*). Thus we refer to the mutable update sequence as the ***timeline***.

We distinguish two forms of retroactivity. In ***partial retroactivity***, queries can be made only of the final version resulting from all of the updates in the timeline; effectively, retroactive updates must be propagated all the way through the timeline in order to answer such queries correctly. In ***full retroactivity***, queries can be made about the data structure at any time, corresponding to the result from a prefix of the timeline. In short, both forms of retroactivity enable modifying the past, and full retroactivity enables querying the past.

Known results. In some settings, retroactivity is easy to achieve. If updates commute with each other and have inverses, then retroactive updates can be moved to the end of the timeline, making partial (but not full) retroactivity easy. If updates are inserts and deletes, and the queries fall under Bentley and Saxe's decomposable search problems, then full retroactivity is possible with an $O(\log m)$ factor overhead [1].

Retroactivity becomes challenging when updates can have non-trival interactions. Here one retroactive update can have a propagated effect on potentially all later updates. In the extreme, when the data structure is a general-purpose computer, a retroactive update can require an $\Omega(m)$ factor overhead [1].

The more interesting middle ground is when the updates have some but limited influence on each other—a common scenario in many classic data structures. For example, logarithmic fully retroactive stacks (with push/pop), queues (with enqueue/dequeue), deques (with all four), union-find, dictionaries, and predecessor/successor structures all have logarithmic fully retroactive data structures [1, 2]. Of these results, predecessor/successor was the most challenging; the original paper [1] solved partial retroactivity in $O(\log m)$ but full retroactivity in $O(\log^2 m)$, which was later improved to $O(\log m)$ by Giora and Kaplan [2]. This problem is equivalent to dynamic rectilinear ray shooting, which was in fact the original motivation for defining retroactivity.

Challenges. A key open problem in retroactivity, posed at SODA 2004, is whether there is a difference in difficulty between obtaining partial versus full retroactivity. The only known upper bound on the separation is a conversion from partial to full retroactivity with $O(\sqrt{m})$ factor overhead [1]. Essentially, this conversion maintains $\Theta(\sqrt{m})$ checkpoints of the timeline using a partially retroactive data structure, and to query in between, replays the necessary $O(\sqrt{m})$ intervening updates. On the other hand, the only known data structural problem with a polynomial separation between the best partially retroactive and best fully retroactive data structures is priority queues (with insert and delete-min operations). The logarithmic partially retroactive priority queue [1] is one of the most sophisticated retroactive data structures, propagating potentially linear-length chain reactions in just logarithmic time. However, the existing approach appeared limited to partial retroactivity. Until now, the fastest known fully retroactive priority queue was the $O(\sqrt{m}\log m)$ bound that follows from the general conversion.

Our results. In this paper, we solve this 11-year-old open problem by constructing the first polylogarithmic fully retroactive priority queue. Specifically, our data structure supports inserting an element, deleting the minimum element, and finding the minimum element, at any time in the timeline, in $O(\log^2 m)$ amortized time per update and $O(\log^2 m \log\log m)$ time per query, using $O(m \log m)$ space. We also show how to support another natural query over the timeline: finding the time at which a given element gets deleted as the minimum (or finding that it remains in the structure in the present).

More importantly, we present a new general transformation from partial to full retroactivity with only a logarithmic factor overhead. This result shows a strong upper bound on the separation between partial versus full retroactivity, but it requires one additional assumption. Specifically, we call a (partially retroactive) data structure ***time-fusible*** if, given two such data structures representing two different timelines (contained in disjoint time intervals), it is possible to form a new (read-only) data structure representing the concatenation of those timelines. Roughly speaking, this assumption lets us apply the $O(\sqrt{m})$ checkpointing idea recursively in a binary tree structure built over the timeline, storing a partially retroactive data structure for the sub-timeline represented by each rooted subtree. Hence we call the transformation ***hierarchical checkpointing***. A retroactive query can then be answered by fusing $O(\log m)$ structures and asking a query about the present.

Our fully retroactive priority queue data structure is an application of this general technique. With some modifications, we show how to fuse two of the logarithmic partially retroactive priority queues from [1] in polylogarithmic time. Applying the general technique gives us a polylogarithmic bound on fully retroactive priority queues, but with worse bounds than those stated above. By a more careful analysis tailored to priority queues, we show how to further tune the hierarchical checkpointing analysis to improve the running time by a logarithmic factor and get the claimed bounds of $\tilde{O}(\log^2 m)$.

Organization. We organize the sections of this paper as follows. Section 2 introduces our hierarchical checkpointing framework in greater detail. Section 3 describes time-fusible partially retroactive priority queues whose timelines may be fused together in polylogarthmic time. Section 4 applies the technique of hierarchical checkpointing to obtain a fully retroactive priority queue with polylogarthmic overheads.

2 Hierarchical Checkpointing

In this section, we present our hierarchical checkpointing technique for transforming a time-fusible partially retroactive data structure into one that is fully retroactive while incurring only polylogarithmic overheads. In later sections, these results will be employed to design a fully retroactive priority queue with polylogarithmic overheads.

We begin by defining in Section 2.1 the notion of time fusibility for retroactive data structures. Then in Section 2.2 we describe the hierarchical checkpoint procedure and prove its correctness.

2.1 Definitions

Here we discuss the properties of partially retroactive data structures and the conditions necessary to use hierarchical checkpointing to obtain full retroactivity.

We define a ***retroactive update*** operation to be the insertion or deletion of a data structure operation at a particular time. These operations are:

– INSERT-OP(o, t): insert a data structure update operation o into the retroactive structure's timeline at time t.
– DELETE-OP(o, t): delete a data structure update operation o from the retroactive structure's timeline at time t.

We define a ***retroactive query*** operation to be one that can determine some aspect of the state of the retroactive data structure at some point in time. We use GET-VIEW(t) as the canonical query procedure when we describe our transformation.

– GET-VIEW(t): returns some aspect of the state of the retroactive data structure at time t.

For partially retroactive structures, query operations can only be performed in the present (i.e. $t = \infty$). Fully retroactive data structures, however, may be queried at any time t. It turns out, that a collection of partially retroactive data structures can be used to support fully retroactive query operations when it is possible to "fuse" their timelines. Formally, we say a partially retroactive data structure is ***time fusible*** if it has the following properties:

1. It supports a function, FUSE(d_1, d_2), that fuses the timelines of two instances d_1 and d_2 of the partially retroactive data structure, producing a version of the data structure that allows read-only queries and reflects the updates in both d_1 and d_2. FUSE(d_1, d_2) need only support fusion between structures containing updates that span disjoint and adjacent intervals of the timeline.
2. Sequences of operations made on it exhibit substring closure; in other words, given a valid sequence of operations, any contiguous subsequence of operations on the structure is also valid.

2.2 The Data Structure

In this section we describe how to transform a time-fusible partially retroactive data structure into one that is fully retroactive using our hierarchical checkpointing framework. Specifically, we obtain a fully retroactive data structure with $O(T(m) \log m + Q(m, k))$ query time and $O(A(m) \log^2 m)$ amortized update time, where $T(m)$ and $A(m)$ represent the merge and update time, respectively, in the original partially retroactive data structure, and $Q(m, k)$ is the query time of a time-fused structure consisting of k fusions and containing m updates.

The first step of our transformation is to build a ***checkpoint tree*** — a balanced binary search tree in which each node of the tree contains a partially retroactive data structure consisting of all the updates in the subtree rooted at that node. Our checkpoint tree is similar to a segment tree [**?**] in that each partially retroactive data structure can be viewed as a segment with endpoints given by the first and last chronological update in the structure. The structures in the leaves of our checkpoint tree each contain only one update, and the leaves are sorted by the time of their one update. The update operations INSERT-OP(o, t) or DELETE-OP(o, t) can be performed on the fully retroactive structure by inserting

into or deleting the update, o, from all of the partially retroactive structures in the search path. A query can be performed at time t by merging $O(\log n)$ disjoint partially retroactive structures obtained from the balanced binary tree such that the fused structure contains all updates in the time span $(-\infty, t]$.

Theorem 1. *Given a partially retroactive data structure that is time fusible, we may construct a fully retroactive version of the data structure using hierarchical checkpointing. This data structure will have an $O(A(m)\log^2 m)$ amortized update time and $O(T(m)\log m + Q(m,k))$ query time.*

We prove Theorem 1 in two parts below.

Lemma 1. *Our hierarchical checkpointing method produces a fully retroactive data structure with $O(A(m)\log^2 m)$ amortized update time.*

Proof. Let F be a fully retroactive data structure based on a time-fusible partially retroactive data structure P. Suppose that m updates have been inserted into F and that the update operation for P runs in $O(A(m))$ time.

We utilize a scapegoat tree [4] to represent the checkpoint tree for F. The checkpoint tree contains all updates to the fully retroactive structure at its leaves ordered by time. Each internal node, x, is associated with an instance of P that reflects the application of all updates in its subtree. To perform Insert-Op(o, t) or Delete-Op(o, t), we insert the update as a leaf in the checkpoint tree, and apply the update to the instances of P associated with nodes along the roof to leaf path in $O(A(m)\log m)$ time.

To rebalance the checkpoint tree, the tree rooted at the scapegoat node is rebuilt. We begin by obtaining a sorted list of the k updates ordered by time by performing an in-order walk of the subtree. We create a balanced binary tree with these k updates at the leaves, and initialize an empty instance of P for each internal node of the subtree. Then, we insert the update at each leaf into each of its $O(\log k)$ ancestors. Because applying an update to an instance of P takes $O(A(k))$ time, the total time required to rebuild a subtree containing k updates is $O(A(k)\log k)$. The total cost of an Insert-Op or Delete-Op operation for the fully retroactive structures is then the sum of the cost of an insertion or deletion and the amortized cost of rebuilding, $O(A(m)\log^2 m)$ amortized.

Lemma 2. *Our hierarchical checkpointing method produces a fully retroactive data structure with $O(T(m)\log m + Q(m,k))$ query time.*

Proof. Suppose that $T(m)$ is the time it takes to fuse any two instances of P, and $Q(m,k)$ is the time it takes to query an instance of P, where m is the total number of updates in P, and k is the number of components that were used to create the fused structure.

To perform Get-View(t), we first traverse the checkpoint tree to identify the $O(\log m)$ disjoint subtrees that represent the time interval $(-\infty, t]$. The time-fusible partially retroactive structures associated with these subtrees are then fused in-order, resulting in a single structure representing the interval $(-\infty, t]$.

We can fuse $O(\log m)$ P structures in $O(T(m) \log m)$ time. Querying this structure then takes $O(Q(m, k))$ time. Therefore, the total runtime of GET-VIEW(t) is $O(T \log m + Q(m, k))$.

3 Time-Fusible Partially Retroactive Priority Queue

In this section we present a partially retroactive priority queue that supports a polylogarithmic fusion operation. Specifically, we describe an algorithm that fuses $k = O(\log m)$ partially retroactive priority queues containing m updates in $O(k \log k \log m)$ time. This time-fusible partially retroactive priority queue enables the use of hierarchical checkpointing to obtain a fully retroactive priority queue with polylogarithmic overheads.

3.1 Partially Retroactive Priority Queues

We begin with an informal review of a partially retroactive priority queue data structure. To simplify our exposition, we treat the partially retroactive priority queue from [1] as a black box and maintain 2 auxillary data structures: Q_{now} containing the set of all keys remaining in the priority queue at time $t = \infty$, and Q_{del} containing all keys that were removed from the priority queue at some point in the past.

We assume that the partially retroactive priority queue returns, following each retroactive update, the keys which should be inserted or deleted from Q_{now} and Q_{del}. If a priority queue is empty at time t, then a delete-min operation will, by convention, insert a placeholder key of infinite weight into Q_{del}. It is known that, following a retroactive update at time t, it is only necessary to insert or delete a single key into Q_{now} and Q_{del} [1]. We can, therefore, synchronize our auxillary data structures Q_{now} and Q_{del} with the partially retroactive priority queue in $O(\log m)$ time. A proof of this claim and an in-depth description of the partially retroactive priority queue data structure can be found in [1, 5.4].

The auxillary Q_{now} and Q_{del} structures are maintained using weight-balanced B-trees [**?**,**?**,**?**] which for a balance factor $d > 4$ have the following properties:

- Insertion and deletion operations on a B-tree containing m elements take $O(\log m)$ time.
- For all non-root nodes u at height h the weight $w(u)$ of the subtree rooted at u is bounded as follows: $d^h/2 \leq w(u) \leq 2d^h$.
- The root r of a height-h tree has bounded weight $w(r)$: $d^{h-1} \leq w(r) \leq 2d^h$.
- Tree-split and concatenate operations on a size-m tree take $O(\log m)$ time.
- A height-h' subtree T' of a height-h weight-balanced B-tree T can be deleted to form the weight-balanced B-tree $T - T'$ in $O(d(h - h'))$ time.

A weight-balanced B-tree data structure possessing these properties is described in [**?**,**?**]. Specifically, we apply the result of [**?**] with balance factor $d = 8$ to maintain Q_{now} and Q_{del}.

3.2 Fusion Algorithm

Before describing our algorithm for fusion, let us better understand the structure of the problem by proving a mathematical relationship between two partially retroactive priority queues that represent two fusible (i.e. disjoint and adjacent) intervals of time.

Lemma 3. *Consider two partially retroactive priority queues Q_1 and Q_2 whose update times lie in the intervals $[a, b)$ and $[b, c)$ respectively. Then, the partially retroactive priority queue Q_3 containing all updates in Q_1 and Q_2 in the interval $[a, c)$ has the property that*

$$Q_{3,now} = Q_{2,now} \cup \textit{max-A}\,\{Q_{1,now} \cup Q_{2,del}\} \tag{1}$$

$$Q_{3,del} = Q_{1,del} \cup \textit{min-D}\,\{Q_{1,now} \cup Q_{2,del}\} \tag{2}$$

where $A = |Q_{1,now}| - |Q_{2,del}|$, $D = |Q_{2,del}|$ and max-C $\{S\}$ denotes the C largest elements in the set S.

Using Lemma 3 we can construct a time-fused representation of Q_3 from Q_1 and Q_2 in polylogarithmic time. We will represent each of $Q_{3,now}$ and $Q_{3,del}$ as a list of trees obtained via tree-split operations consistent with the application of Equation (1) and Equation (2). We say that a time-fusible partially retroactive priority queue has order k, and use the superscript notation Q^k, if Q^k_{now} and Q^k_{del} are represented as lists of at most k trees.

In Figure 1 we provide the pseudocode for FUSE which fuses two partially retroactive priority queues Q^k_1, Q^k_2 to obtain Q^{3k}_3. Step 1 computes the value of A from Lemma 3, and step 2 concatenates the list of trees representing $Q^k_{1,now}$ and and $Q^k_{2,del}$ to form a list L containing $2k$ trees. Step 3 computes a "split-key" x that is greater than A elements contained in trees of L. Next each tree in L is split in step 4 by performing a tree-split operation to divide each tree T_i into a tree $T_{i,<}$ containing all keys in T_i that are less than x and $T_{i,>}$ containing all keys in T_i that are greater than x. The trees $T_{i,>}$ for $i = 1, 2, \ldots, 2k$ combined with the trees in $Q_{2,now}$ contain the elements satisfying the relation of Equation (1) in Lemma 3, and similarly the trees in $Q_{1,del}$ and in $T_{i,<}$ for $i = 1, 2, \ldots, 2k$ contain the elements satisfying the relation of Equation (2).

The following theorem proves that FUSE fuses two partially retroactive priority queues of order k in $O(k \log m)$ time.

Theorem 2. *Consider two partially retroactive priority queues Q^k_1 and Q^k_2 with order k containing m operations. Then* FUSE(Q^k_1, Q^k_2) *runs in $O(k \log m)$ time.*

Proof. We first show that GETSPLITKEY runs in $O(k \log m)$ time. Our algorithm for finding the split key is an adaptation of the approach of Frederickson and Johnson to compute order statistics for sorted arrays [5].

Steps 1, 2, and 4 of GETSPLITKEY run in $O(k)$ time (step 4 uses linear-time weighted selection from [**?**]).

Step 3 finds a leftmost subtree T_{m_i} whose contents are contained in the range $(-\infty, m_i)$ and where the order statistic of m_i is in the range $(|T_i|/256, |T_i|/4)$.

GetSplitKey$(s, T_1, \ldots, T_k)$

1. If $N = \sum_i |T_i| < C$ (for constant C), sort $\bigcup_i T_i$ and return the sth element.
2. If $s < N/2$, set $s = N - s$ and "invert" the order of each T_i.
3. For each T_i, pick a leftmost subtree T_{m_i} containing keys in the range $(-\infty, m_i)$ where m_i has an order statistic in T_i contained in the range $(|T_i|/256, |T_i|/4)$.
4. Assign each m_i the weight $w_i = |T_i|$. Using weighted selection, select the $N/4$th element m_j among $m_1, m_2, \ldots, m_k$
5. For $m_i \leq m_j$, let $T_i' = T_i - T_{m_i}$. For $m_i > m_j$, let $T_i' = T_i$.
6. Set $s_{new} = s - \sum_i (|T_i| - |T_i'|)$ and return GetSplitKey$(s_{new}, T_1', \ldots, T_k')$.

(a)

Fuse(Q_1^k, Q_2^k)

1. $A = |Q_{1,now}| - |Q_{2,del}|$
2. Form a list of $2k$ trees $L = T_1, \ldots, T_{2k}$ by concatenating the list of k trees representing $Q_{1,now}$ with the k trees representing $Q_{2,del}$.
3. $x =$ GetSplitKey$(A, T_1, \ldots, T_{2k})$
4. For $i = 1, 2, \ldots 2k$, split the tree T_i on the key x to obtain 2 trees $T_{i,>}$ and $T_{i,<}$.
5. $Q_{3,now} = Q_{2,now} + T_{1,>}, \ldots, T_{2k,>}$
6. $Q_{3,del} = Q_{1,del} + T_{1,<}, \ldots, T_{2k,<}$
7. Return Q_3

(b)

Fig. 1: Pseudocode for (a) the GetSplitKey operation; and (b) the Fuse operation. GetSplitKey takes a key s and a list of k binary trees, and returns a key x such that s keys in $T_1, T_2, \ldots, T_k$ are less than x.

We show that step 3 runs in $O(k)$ time by showing that for each T_i such a subtree exists at a distance of at most 2 from the root. Consider a height-h weight-balanced B-tree with balance factor d, root node r, and an internal node u at height $h-2$. The weight-balance criteria for B-trees provided in Section 3.1 implies that the ratio $w(u)/w(r)$ is bounded in the range $(1/256, 1/4)$. The key m_i can, therefore, be found in $O(1)$ time by selecting the maximum key from the leftmost height-$(h-2)$ subtree of T_i.

Step 5 deletes the subtree T_{m_i} from T_i if $m_i \leq m_j$. The difference in the heights of T_{m_i} and T_i is at most 2, which allows $T - T_{m_i}$ to be obtained in $O(d)$ time while preserving weight-balance. For $d = 8$, this step runs in $O(k)$ time. Note that the subtrees deleted in this step contain elements whose order statistic is strictly less than $N/2$ and thus these subtrees can not contain the sth order statistic. To prove this we show that the order statistic of m_j, computed in step 4, is less than $N/2$. The key m_j is selected in step 4 such that $3N/4$ elements are contained in trees T_i for which $m_i > m_j$. For each such i, the key m_i is smaller than at least $3|T_i|/4$ of the elements in T_i. The key m_j is, therefore, smaller than at least $9N/16$ elements, and thus has an order statistic less than $N/2$.

Step 6 updates the value of s to reflect the reduced problem size, and recursively calls GetSplitKey. To bound the depth of the recursion, it is sufficient to show that step 5 eliminates a constant fraction of the elements. Since a total of $N/4$ elements are contained in trees T_i for which $m_i \leq m_j$, and at least $|T_i|/256$ elements in T_i are smaller than m_i, step 5 eliminates at least $N/1024$ elements. The recursion depth is, therefore, bounded by $O(\log N)$. Since $N = O(m)$, the total runtime of GetSplitKey is $O(k \log m)$

Next let us analyze the FUSE operation. Steps 1-2 and 5-6 of FUSE can be performed in $O(k)$ time. Step 3 to compute the split key runs in $O(k \log m)$ time, and step 4 may be performed in $O(k \log m)$ time by performing an $O(\log m)$ time tree split operation on each of k trees. The runtime of FUSE is bounded by the time to compute the split key, and therefore is $O(k \log m)$.

The bound proved in Theorem 2 depends on the order k of the two time-fusible partially retroactive priority queues Q_1^k, Q_2^k being merged. It turns out, that the fusion of k partially retroactive priority queues can be constructed efficiently while being represented using only $O(k)$ trees by combining trees in Q_{now} and Q_{del} that originated from a split operation on a common tree. The ability to perform such a reduction relies on the following lemma.

Lemma 4. *Let $Q_1, \ldots, Q_k$ denote k partially retroactive priority queues each with disjoint time intervals that increase monotonically with k. Let Q_* be a priority queue containing the updates in $Q_1, \ldots, Q_k$ applied consecutively. Then $Q_{*,now}$ and $Q_{*,del}$ consist of contiguous intervals of $Q_{i,now}$ and $Q_{i,del}$, i.e.*

$$Q_{*,now} = \cup_{i \in S_{now}} Q_{i,now}[a_i, b_i] \cup_{i \in S_{del}} Q_{i,del}[a_i', b_i'] \tag{3}$$

$$Q_{*,del} = \cup_{i \in T_{now}} Q_{i,now}[c_i, d_i] \cup_{i \in T_{del}} Q_{i,del}[c_i', d_i'] \tag{4}$$

for some sets $S_{now}, S_{del}, T_{now}, T_{del} \subseteq \{1, \ldots, k\}$ and elements a_i,a_i',b_i,b_i',c_i,c_i',d_i,d_i' where for a set S and $a, b \in S$ we let $S[a, b] = \{x \in S : a \leq x \leq b\}$.

The preceding lemma allows us to tweak the fusion algorithm to guarantee that the order of the fusion of k time-fusible partially retroactive priority queues is bounded by $2k$. This is accomplished by adding a post-processing step POST-FUSE immediately after the fusion procedure FUSE. After obtaining the fusion Q_3 of Q_1 and Q_2, the trees representing $Q_{3,now}$ are checked in POSTFUSE to identify pairs of split-trees that were obtained by splitting a common tree. By Lemma 4 the union of these intervals span disjoint intervals and these pairs of trees can, therefore, be concatenated in logarithmic time.

Lemma 5. *The fusion of k time-fusible partially retroactive priority queues has order bounded by $2k$ and runs in $O(k \log m)$ time when using the* POSTFUSE *procedure.*

To combine the results of this section, we prove the following theorem.

Theorem 3. *Consider $k = O(\log m)$ time-fusible partially retroactive priority queues. The time to fuse these k data structures is bounded by $O(k \log k \log m)$, and the time required to query this structure is $O(\log^2 m)$.*

Proof. We arrange the k time-fusible structures at the leaves of a balanced height-$\log k$ merge tree. By Lemma 5 the sum of the orders of time-fusible partially retroactive priority queues at level i in the merge tree is $O(k)$. The total work to perform fusions at level i is, therefore, $O(k \log m)$ Since there are $\log \log m$ levels in the merge tree the total time is $O(k \log m \log \log m)$. To query the fused structure we perform a query on each of the $O(\log m)$ trees representing Q_{now} which can be done in $O(\log^2 m)$ time.

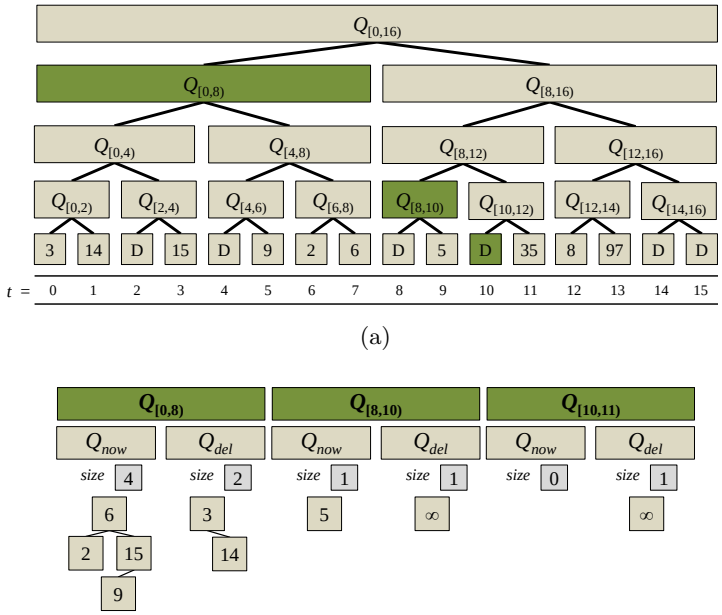

Fig. 2: Hierarchical checkpointing for fully retroactive priority queue. Illustration of the checkpoint tree for a fully retroactive priority queue with 16 operations.

4 Fully Retroactive Priority Queue

In this section we describe the design of a fully retroactive priority queue that uses hierarchical checkpointing. We begin in Section 4.1 by showing how to apply our technique of hierachical checkpointing using the time-fusible partially retroactive priority queue of Section 3. This yields a fully retroactive priority queue that supports retroactive updates in $O(\log^3 m)$ amortized time, retroactive queries in $O(\log^2 m \log\log m)$ time, and FIND-DELETION-TIME in $O(\log^3 m \log\log m)$ time. Next, in Section 4.2, we optimize our application of hierarchical checkpointing for priority queues to obtain $O(\log^2 m)$ amortized time updates, and $O(\log^2 m)$ time FIND-DELETION-TIME queries.

4.1 Obtaining Full Retroactivity using Hierarchical Checkpointing

Here we analyze the fully retroactive priority queue obtained by a straightforward application of hierarchical checkpointing. The time-fusible partially retroactive priority queue described in Section 3 meets the prerequisites of Theorem 1 needed to perform the partial-to-full transformation. Consequently we can directly apply this theorem to obtain a fully retroactive priority queue which follows the structure laid out in Section 2. A checkpoint tree contains all retroactive updates ordered by time, and each internal node maintains a time-fusible partially retroactive priority queue that contains the updates within its subtree.

The checkpoint-tree data structure used in this fully retroactive priority queue is shown in Figure 2(a) after 16 retroactive operations have been per-

formed. In this example, the checkpoint tree has 16 leaves each corresponding to a retroactive operation on the priority queue. The time-fusible partially retroactive priority queue data structure described in Section 3 is used to represent the partial checkpoints in a checkpoint tree. Each internal node, $Q_{[a,b)}$, maintains a time-fusible partially retroactive priority queue that contains all retroactive operations in its subtree (i.e. all operations occurring at times $t \in [a, b)$).

The GET-VIEW(t) operation is illustrated in Figure 2(b). A checkpoint representing the priority queue at time $t = 10$ is constructed by combining 3 partial checkpoints from the checkpoint tree. The time-fusible partially retroactive priority queues $Q_{[0,8)}$, $Q_{[8,10)}$, and $Q_{[10,11)}$ that are highlighted in Figure 2 are collected and then merged to obtain obtain a partially retroactive priority queue containing all updates in in the interval $[-\infty, 10]$.

Theorem 4. *There exists a fully retroactive priority queue that supports retroactive updates in $O(\log^3 m)$ amortized time, queries in $O(\log^2 m \log\log m)$, and the operation,* FIND-DELETION-TIME*, in $O(\log^3 m \log\log m)$ time.*

Proof. The time-fusible partially retroactive priority queue described in Section 3 supports retroactive updates in $O(\log m)$ time. Applying Lemma 1 with $A(m) = \log m$ shows that retroactive updates run in $O(\log^3 m)$ amortized time. By Theorem 3, the time to merge $O(\log m)$ time-fusible partially retroactive priority queues is bounded by $O(\log^2 m \log\log m)$. Similarly, the time to query this merged structure is bounded by $O(\log^2 m)$ since the merged priority queue has order $O(\log m)$. Applying Lemma 2 with $T(m) = O(\log^2 m)$ and $Q(m) = O(\log^2 m \log\log m)$ shows that retroactive queries run in $O(\log^2 m \log\log m)$ time. Finally, the FIND-DELETION-TIME(x) operation can be performed via binary search to identify the first time t for which the key x is not in the queue. This involves $O(\log m)$ retroactive queries showing that FIND-DELETION-TIME runs in $O(\log^3 m \log\log m)$ time.

4.2 Faster Retroactive Updates and FIND-DELETION-TIME Queries

The general transformation described in Section 2 maintains balance in the checkpoint tree by reapplying all updates in rebuilt subtrees. As shown in Lemma 6 a checkpoint tree for priority queues can be rebuilt more efficiently.

Lemma 6. *A subtree of the fully retroactive priority queue's checkpoint tree containing m updates can be rebuilt in $O(m \log m)$ time.*

Proof. Consider a node u in the checkpoint tree with children v and w whose subtree contains m updates. The time-fusible priority queue containing all updates in u's subtree can be computed in $O(m)$ time from the 2 time-fusible priority queues associated with v and w. First the FUSE operation outlined in Section 2 is performed to merge v and w. The resulting time-fusible priority queue may represent Q_{now} and Q_{del} using multiple trees, but these trees can be merged in $O(m)$ time. Using this merge procedure, a subtree of the checkpoint tree is rebuilt by first placing all m updates at the leaves of a balanced tree,

and then performing merges from the leaves upward. Each update is involved in $O(\log m)$ merges, so the total time to rebuild the subtree is $O(m \log m)$.

A more efficient implementation of the FIND-DELETION-TIME(k) operation can be obtained by performing a binary search directly on the checkpoint tree. The high-level idea is to perform a binary search for the time of deletion by keeping track of the current number of surviving keys that are less than or equal to k at any particular time. Due to space limitations, this result is stated in Lemma 7 without proof.

Lemma 7. *The* FIND-DELETION-TIME *operation which performs a binary search directly on the checkpoint tree data structure runs in* $O(\log^2 m)$ *time.*

Theorem 5. *The fully retroactive priority queue performs updates in* $O(\log^2 m)$ *amortized time when using a checkpoint tree with the memoized subtree rebuilding procedure, and performs* FIND-DELETION-TIME *operations in* $O(\log^2 m)$ *time.*

Acknowledgments

This research was initiated during the open-problem sessions of the MIT class 6.851: Advanced Data Structures taught by E. Demaine in Spring 2014. We thank Adam Hesterberg, Ofir Nachum, and other members of the class for helpful discussion regarding this problem.

References

1. Demaine, E.D., Iacono, J., Langerman, S.: Retroactive data structures. ACM Transactions on Algorithms (TALG) **3** (2007) 13
2. Giora, Y., Kaplan, H.: Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. ACM Transactions on Algorithms (TALG) **5** (2009) 28
3. Acar, U.A., Blelloch, G.E., Tangwongsan, K.: Non-oblivious retroactive data structures. Technical report (2007)
4. Galperin, I., Rivest, R.L.: Scapegoat trees. In: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms, Society for Industrial and Applied Mathematics (1993) 165–174
5. Frederickson, G.N., Johnson, D.B.: The complexity of selection and ranking in¡ i¿ x¡/i¿+¡ i¿ y¡/i¿ and matrices with sorted columns. Journal of Computer mand System Sciences **24** (1982) 197–208