Journal of Computer Science

Original Research Paper

Fully Retroactive Priority Queues using Persistent Binary Search Trees

[1]JoseWagner de Andrade Junior and [2]Rodrigo Duarte Seabra

[1]*Institute of Systems Engineering and Information Technology, Federal University of Itajuba, Itajuba, Brazil*
[2]*Institute of Mathematics and Computing, Federal University of Itajuba, Itajuba, Brazil*

Article history
Received: 20-04-2020
Revised: 23-06-2020
Accepted: 11-07-2020

Corresponding Author:
Rodrigo Duarte Seabra
Institute of Mathematics and Computing, Federal University of Itajuba, Itajuba, Brazil
Email: rodrigo@unifei.edu.br

Abstract: Classic dynamic data structures maintains itself subject to sequence *S* of operations and answer queries using the latest version of the data structure. Retroactive data structures are those which allow making a modification or a query in any version of this data structure through its timeline. These data structures are used in some geometric problems and in problems related with graphs, such as the minimum path problem in dynamic graphs. This work presents how to implement a data structure to a fully retroactive version of a priority queue through persistent self-balanced binary search trees in polylogarithmic time. We use these data structures to improve the performance merging two versions of partially retroactive priority queues. The empirical analysis showed that the average performance of the proposed algorithm is better in terms of processing times than the other algorithms, despite the high constants in its complexity.

Keywords: Retroactivity, Data Structures

Introduction

Considering the computational evolution and the miniaturization of hardware components, software should be able to support a large volume of data and an expressive number of operations. In some applications, it is necessary to maintain the history of operations performed and a change in one of these operations can create a cascade effect in this historical sequence of events. For example, supposing that one discovered a wrong measure received from a sensor and needs to update this measure given by the sensor. Once the measure from this device changes, all the information previously extracted from the next measures needs to be changed as well. A way to deal with all this different information is performing a rollback on all measures and re-extracting the information about the new measure. This process is sometimes non-optimal. The notion of retroactive data structures was created by (Demaine *et al*., 2007).

In the literature, there are two types of temporal data structures: Persistent and retroactive. In both types, it is possible to perform updates and queries to the past. The difference between these two types of data structures is what happens when an operation is performed. In persistent data structures, a new version of this structure is built, deriving from the modified parts of the operation. As an example of this structure, we have the version control software, such as *git*, which allows creating a new branch in a main timeline of the project, changing only some parts of the entire project.

The other type of data structure is the retroactive one. Retroactive data structure is defined by (Demaine *et al*., 2007) as data structures that efficiently support modifications to the historical sequence of operations performed on the structure. In retroactive data structures, we are interested in studying and optimizing the cascade effect created by changing an operation in the past of the data structure.

In a priority queue, the standard operations are:

- *Push*(*x*): Add a value x to the data structure
- *Pop*(): Delete the minimum value in the data structure
- *GetPeak*(): Return the minimum value in the priority queue

These operations can be easily handed in $O(\lg(n))$ time using binary heaps. However, in the retroactive version of this data structure, we need to be able to execute the following operations:

- *Insert*(*t*, *Push*(*x*)): Add a value x to the data structure at time *t*
- *Insert*(*t*, *Pop*()): Insert a deletion removing the minimum value in the data structure at time *t*

- *Delete*(t, *Push*(x)): Delete the operation *Push*(x) performed at time t
- *Delete*(t, *Pop*()): Delete the operation *Pop*() executed at time t
- *GetPeak*(t): Return the minimum value in the priority queue at time t

In a partial retroactive data structure, the query operations, such as *GetPeak*(t), will always run with $t = \infty$ (i.e., at the present time).

The operations in the retroactive version of the priority queue are a bit harder to handle, once a modification in the past can create a cascade effect, changing the timelines of each element in the data structure.

Demaine *et al.* (2007) proposed retroactive versions, partial and full, to some data structures such as stacks, queues, union-finds and priority queues. In that occasion he proposed a fully retroactive priority queue in $O\left(\lg m \cdot \sqrt{m}\right)$ time per update, where m is the size of the timeline in which the data structure is implemented. Years later, Demaine *et al.* (2015) presented an optimized solution that allows performing the update operations in $O(\lg^2 m)$ time, using a data structure called checkpoint tree. This new approach also supports the operation of determining the time at which an element was deleted from the data structure in $O(\lg^2 m)$ time.

However, we could not find any paper about implementing and testing these data structures. We use the theoretical knowledge presented by Demaine's articles to implement and to test the fully retroactive priority queue and fill this lack of implementations of retroactive data structures. In this article we perform a slight modification in the original algorithm proposed to get the fully retroactive priority queue in poly-logarithmic time using fully persistent self-balanced binary search trees.

Related Works

In computing, a persistent data structure is one that always preserves the previous version of itself when it is modified. This term was introduced by (Driscoll *et al.*, 1989). A data structure is called partially persistent when any version of the data structure can be accessed, but only the newest version can be modified. A data structure is called fully persistent when we are able to modify any version of the data structure. There is also the notion of confluent persistent data structures, which are structures created by merging two different versions of the same data structure. Data structures that are not persistent are called ephemeral (Kaplan, 2018). A practical example of using persistent data structures is the planar point location problem proposed by (Sarnak and Tarjan, 1986). In this problem, we have a set P of n non-intersect polygons $P = \{p_0, p_1, \cdots, p_n\}$ and we need to answer q queries. In each of these queries, we need to answer, given a point $v = (x, y)$, the index of a polygon which contains point v, or answer if the point is not contained in any polygon. Using persistent binary search trees, the algorithm proposed by (Sarnak and Tarjan, 1986) consumes $O(n)$-space and $O(\lg(n))$-time complexity.

The literature considers two types of retroactivity: Partial and full. Partial retroactivity allows the user to know how changes made in the past currently affect the structure. Fully retroactive data structures allow the user to make queries and updates both in the past and in the present (Demaine *et al.*, 2007). There is also a concept of non-oblivious retroactive, introduced by (Tangwongsan and Blelloch, 2007), which are structures that allow the user to know, after an update in the past, the first instant at which this data structure will become inconsistent. Partially retroactive data structures are more efficient and less complex than fully retroactive data structures.

The notion of retroactive data structure helps to solve problems, such as dynamic shortest path problem (Sunita and Garg, 2018) and geometric problems, such as cloning Voronoi diagrams (Dickerson *et al.*, 2010) and nearest neighbor search (Goodrich and Simons, 2011).

Demaine *et al.* (2015) proposed a fully retroactive priority queue that consumes $O(\lg^2 m)$ per update, improving the previous lower bound of $O\left(\sqrt{m} \cdot \lg m\right)$ time per operation. The data structure also supports the operation of determining the time at which an element was deleted from the data structure in $O(\lg^2 m)$.

Chen *et al.* (2018) sets a nearly optimal separation between partially and fully retroactive data structures. The authors used some conjectures to prove that the upper bounds between partially and fully retroactive data structures is $n << \sqrt{m}$, where n is the size of data structure and m is the number of operations in the timeline, by showing a new transformation with multiplicative overhead $n\lg m$. They also proved a lower bound of $\left\{n \cdot \lg m, \sqrt{m}\right\}^{1-o(1)}$.

Henzinger and Wu (2019) sets upper and lower bounds for fully retroactive graph problems, such as graph connectivity, minimum spanning forest and maximum degree. They also proposed an algorithm for incremental fully retroactive connectivity in $\tilde{O}(1)$ time per operation.

Fully Retroactive Priority Queue in Polylogarithmic Time

It is possible to transform a time-fusible data structure, with a logarithmic multiplicative cost, using a technique called hierarchical checkpointing. Two data structures E_1 and E_2 are called time-fusible if they represent the same data structure in consecutive disjoint times.

In other words, let I_{E_1} = [l_1, r_1] be the time range corresponding to E_1 and I_{E_2} = [l_2, r_2] be the time range related to data structure E_2. Thus, these data structures are time-fusible if $r_1 < l_2$ *e* $r_1 + 1 = l_2$. The union of these data structures generates another data structure $E_f = E_1 \cup E_2$, which covers the range I_{E_f} = [l_1, r_2].

With this definition, it is possible to generate a binary tree in the data structure timeline, in which each node represents a continuous time range in this timeline. This transformation was denoted by (Demaine *et al*., 2015) as hierarchical checkpointing

To create a data structure using this technique, the first step consists in building the checkpoint tree - a binary search tree in which each node maintains a partially retroactive data structure containing all the updates made to its sub-trees. This tree is similar to the segment tree.

In a segment tree, each node represents a continuous time range [l, r], starting at time l and ending at time r. Each node from this segment tree will contain a partial retroactive priority queue and two auxiliary sets: Q_{now}, containing the elements inside the priority queue, considering the operations performed between l and r and Q_{del} containing the elements removed by some operation in this temporal range. Sets Q_{now} and Q_{del} are given by the partial retroactive data structure inside a certain node.

If a priority queue is empty when an operation Pop at time t is performed, then this operation will insert a key with infinity value in Q_{del}. That is equivalent to inserting a value ∞ at time t and immediately removing it.

Figure 1 shows the representation of a checkpoint tree. The set $Q_{[l,\ r]}$ represents the priority queue which covers all the operations carried out in the time range [l, r], in the data structure timeline. In this case, the data structure has a timeline of size 16 and a query is being performed in the structure at time 11. The green nodes represent the ones inside the time range in this query and the leaf nodes represent the operations conducted through the timeline. In the leaves, D represents that a delete operation was performed, whereas the numbers represent the insertions. In a checkpoint-tree, the leaf nodes also represent a data structure in a single point, that is, $Q_{[i,i]}$ for all the i inside the timeline range.

Figure 2 shows how each node keeps the information about the operations carried out. Each node contains two sets, Q_{now} and Q_{del}, as aforementioned. Thus, to obtain the range $Q_{[1,11]}$ (equivalent to a fully retroactive priority queue at time 11), we need to obtain $Q_{[1,8]} \cup Q_{[9,10]} \cup Q_{[11,11]}$.

Algorithm to Merge Two Partially Retroactive Priority Queues

Demaine *et al*. (2015), it is possible to merge two partially retroactive priority queues. Consider two time-fusible priority queues Q_1 and Q_2 (that is, with time range [l_1, r_1] and [l_2, r_2] that $r_1+1 = l_2$). We can thus generate a priority queue Q_3 merging Q_1 and Q_2 covering the range [l_1, r_2]:

$$Q_{3,now} = Q_{2,now} \cup \max - A\left\{Q_{1,now} \cup Q_{2,del}\right\}$$
$$Q_{3,del} = Q_{1,del} \cup \min - D\left\{Q_{1,now} \cup Q_{2,delg}\right\}$$

where, $A = |Q_{1,now} \cup Q_{2,del}| - |Q_{2,del}|$, $D = |Q_{2,del}|$ and max-$C\{S\}$ denote the greatest C elements in S, as min-$C\{S\}$ represents the smallest C elements in set S. In the algorithm, these sets are only merged when a query operation is performed.

After defining the sets, it is possible to write Algorithm 1, which returns the union of these partially retroactive priority queues (Demaine *et al*., 2015). In this Algorithm, *getSplitKey*(D,T) returns a value x, in which all the subsets of T should be divided and the number of values smaller or equal to x are equal to D. This function can be implemented in logarithmic time in the size of the set generated by the union of $Q_{1,now}$ and $Q_{2,del}$ using a binary search.

Algorithm 1 Algorithm to merge two partially retroactive priority queues

```
1: procedure Merge($Q_1$, $Q_2$)
2:     $D \leftarrow |Q_{2,del}|$
3:     $T \leftarrow \{Q_{1,now} \cup Q_{2,del}\}$
4:     $x \leftarrow getSplitKey(D,T)$
5:     $Q_{3,now} \leftarrow Q_{2,now} \cup T > x$
6:     $Q_{3,del} \leftarrow Q_{1,del} \cup T \leq x$
7:     return $Q_3$
8: end procedure
```

Given a set T and a key x, a data structure, able to divide this set into two sets $T_1 = T_{\leq x}$ and $T_2 = T_{>x}$, is necessary. It is possible to use a balanced binary search tree to keep the sets Q_{now} and Q_{del}. In a balanced binary search tree, it is possible to divide this tree into two trees as mentioned previously.

Figure 3 depicts the two first nodes merged in a query related at time 11 in the priority queue presented in Figure 1. The blue and red elements respectively correspond to sets Q_{now} and Q_{del} from partial priority queue $Q_{[1,8]}$, while the yellow and purple elements are the elements inside Q_{now} and Q_{del} from partial priority queue $Q_{[9,10]}$. The blue and purple elements are united by operation $\{Q_{[1,8],now} \cup Q_{[9,10],delg}$, being that the smaller $D = |Q_{[9,10],del}|$ are inserted in a new set Q_{del}, while the other ones are inserted in Q_{now}.

The elements inside $Q_{[1,8],del}$ and $Q_{[9,10],now}$ (yellow and red elements, respectively) are not affected by the union operation, since none of the elements inside set

$Q_{[1,8],del}$ can be recovered when these time ranges are merged. Likewise, the elements inside priority queue $Q_{[9,10],now}$ cannot be erased by a deletion operation that occurred prior to its insertion.

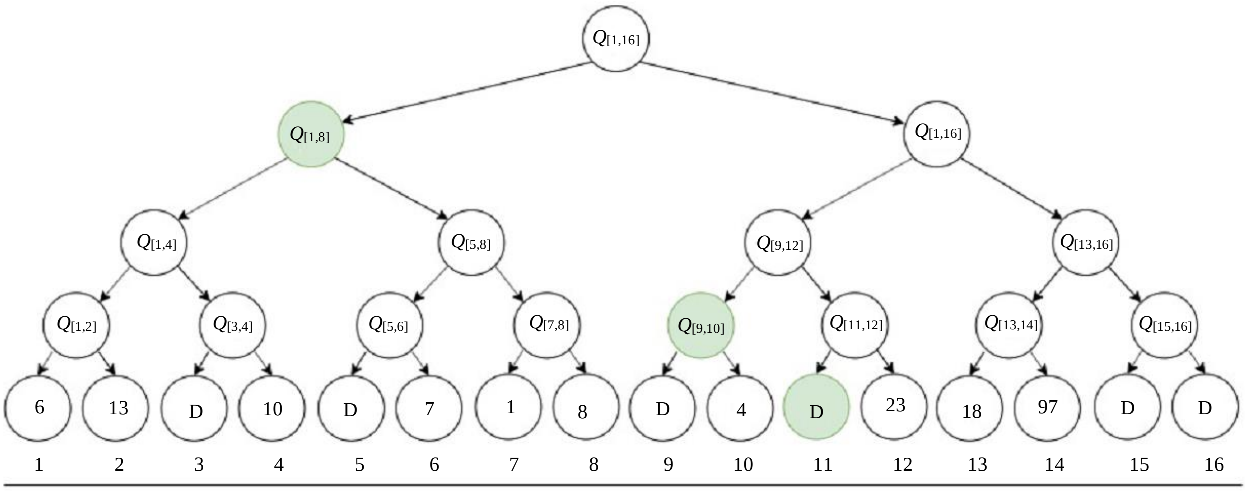

Fig. 1: Example of a checkpoint tree generated from a priority queue. Source: The authors

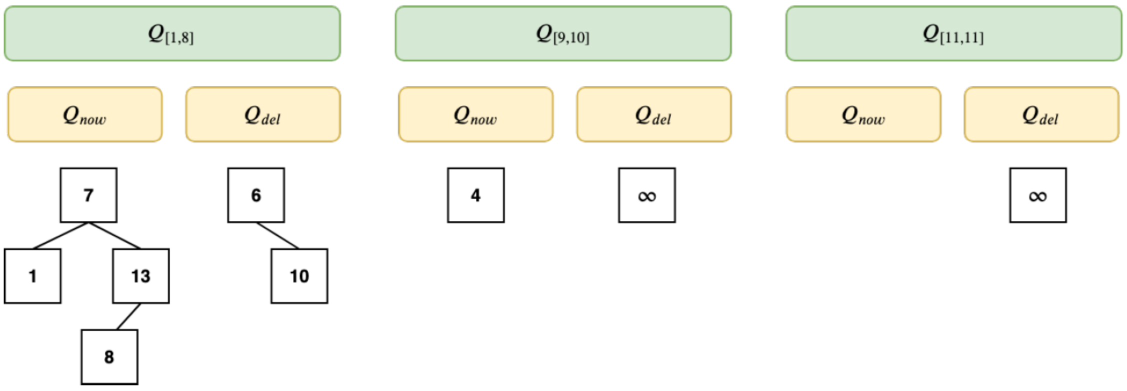

Fig. 2: Query example in a checkpoint tree. Source: The authors

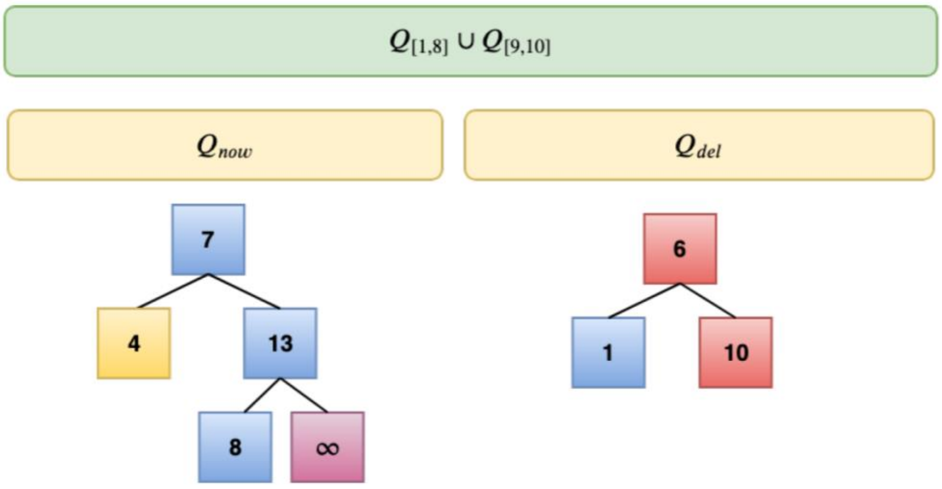

Fig. 3: Example of merging two nodes in a checkpoint tree. Source: The Authors

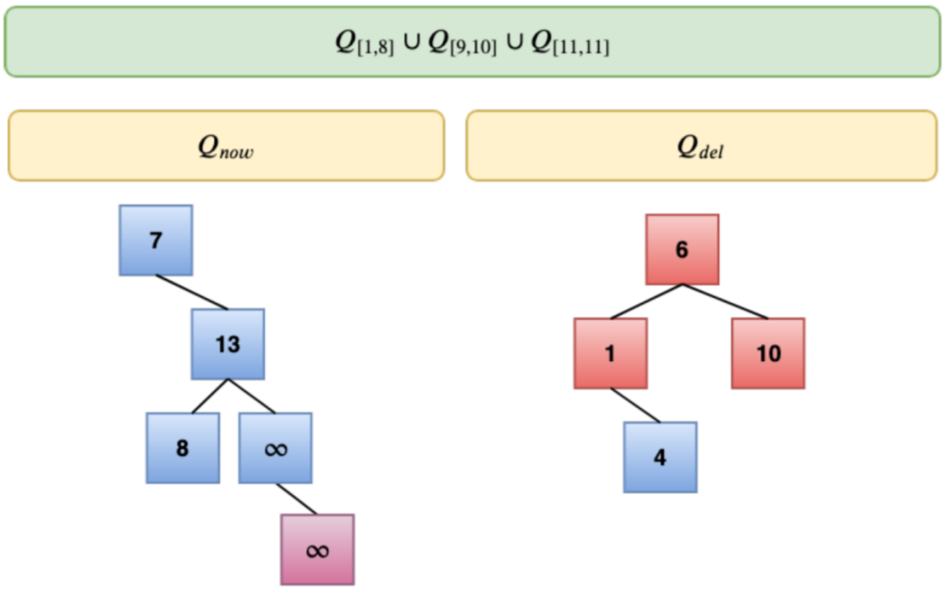

Fig. 4: Example of merging two nodes in a checkpoint tree (continuation). Source: The Authors

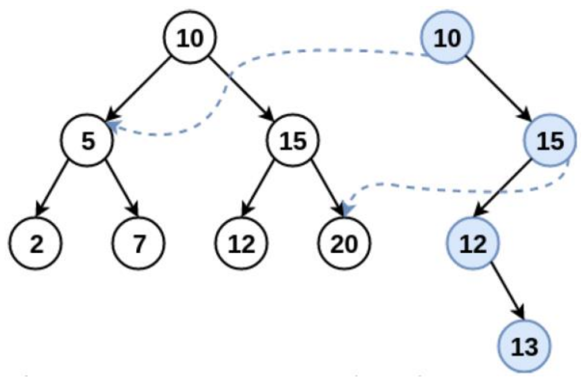

Fig. 5. Persistence example using the path-copying

Figure 4, the same operation is performed, yet considering that $Q_{[1,8]}$ and $Q_{[9,10]}$ were merged by the previous operation. We thus merge $Q_{[1,10]}$ with $Q_{[11,11]}$. Now, the blue and red blocks are the elements inside Q_{now} and Q_{del} from $Q_{[1,10]}$ and the yellow and purple elements are related to partially retroactive priority queue $Q_{[11,11]}$. Thus, it is possible to obtain the minimum element in data structure $Q_{1,11}$ getting the smallest value inside Q_{now} after merging these three nodes from the checkpoint tree.

Using the implementation of a standard binary search tree without any modification for each set, we lose the information about a tree after a division by a value x. A technique in a binary search tree. Source: The Authors. possible solution could be copying the entire tree and, after this operation, performing the division in this new copied tree. However, this solution consumes linear time in the size of the tree. To optimize this solution, we can use a persistent version of a tree, which creates a new version from a tree by only modifying the nodes affected by the division.

Persistent Cartesian Tree

To perform the queries using the approach aforementioned, it is necessary to implement a data structure that keeps the versions of a binary search tree. There are some methods to transform a data structure into its persistent version.

In this implementation, we used a persistent version of the randomized binary search Cartesian tree (Martínez and Roura, 1998). The tree was made persistent using the path-copying technique (Driscoll *et al.*, 1989). In other words, for each node from the root to the modified node, a copy is created and the modifications are made to this copy.

Figure 5 presents the visual representation of an insertion operation in a binary search tree. The example shows the insertion of element 13 in the tree. The operation begins with a standard insertion in a binary search tree. That is, if the x value inserted is lower than the current node; we recursively go to the left child, otherwise, to the right child, repeating this operation until the current node is not empty.

In the example, an insertion in a standard binary search tree would follow path {10, 15, 12} to insert element 13. In the persistent version, this operation follows the same path, but these nodes are copied to create a new version of this tree. When a copy is made, all the attributes of a previous node are copied, including the left and right child of this node. For example, in Fig. 5, node 10 is initially copied and the pointers to the left and right child of this new node point to {5, 15}, respectively (blue dashed edges). After analyzing the node value, the algorithm recursively calls the insertion function to the right, creating a new blue node with value 15 (black edge). Note that, after creating this new node, one of the blue dashed edges was replaced with a black edge by the recursive call. That is, all the blue dashed edges represent connections between distinct versions of the data structure, while the black ones are edges between vertices created in the same version. Using this technique, a new version of the data structure going through the tree height can be created.

Algorithm 2 shows how to implement a node copy *p*. In a copy, all the attributes from this node are duplicated. The function *update*(*p*) updates and returns the node, updating information about the maximum, minimum and the size of the sub-tree rooted in *p*.

Algorithm 2 Function to copy a node

```
pTreapNode copy(pTreapNode p) {
  pTreapNode cpy = new TreapNode(p->key, p->data);
  cpy->l = p->l;
  cpy->r = p->r;
  return update(cpy);
}
```

Algorithm 3 shows the function that implements the split of a binary search tree t, by a value key, generating trees $a = t_{\leq key}$ and $b = t_{>key}$. Therefore, if the current tree is null, this implies that the split function reached the end and the split generated in this state generates two empty trees. If the tree is not empty, the root of this sub-tree will be modified by a split operation and thus, it is necessary to copy this node (line 7).

Algorithm 3 Split operation in a persistent binary search tree

```
1 void split(pTreapNode t, K key,
2             pTreapNode anda, pTreapNode andb) {
3
4    if (!t) {
5       a = b = NULL;
6       return;
7    }
8    pTreapNode aux;
9    t = copy(t);
10   if (key < t->key) {
11      split(t->l, key, a, aux);
12      t->l = aux;
13      b = update(t);
14   }
15   else {
16      split(t->r, key, aux, b);
17      t->r = aux;
18      a = update(t);
19   }
20 }
```

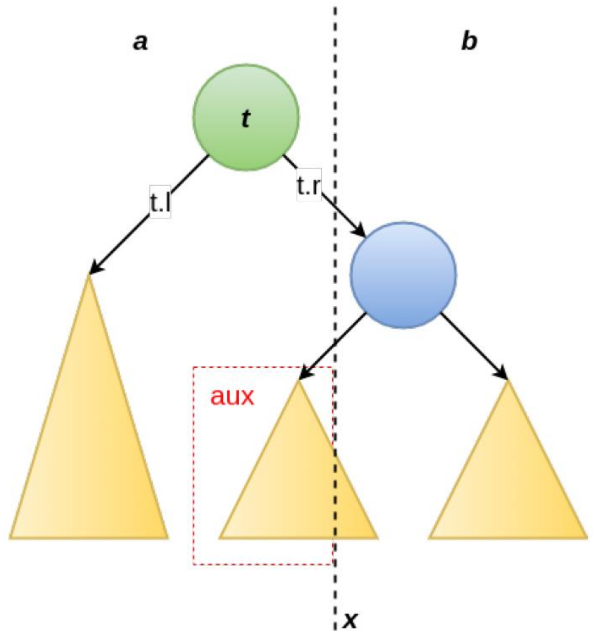

Fig. 6: Example of a split operation by a value *x* in a binary search tree. Source: The Authors

Figure 6 shows the representation of a split operation in a sub-tree rooted at node t by a value x. In the case shown, value *x* is higher than the value in the root of the tree and the figure therefore refers to the operations performed between lines 13 to 16 from Algorithm 3. Initially, the split function is called recursively to the right child of the tree, returning two pointers, *aux* and *b*. The values inside the tree rooted in *aux* are lower than value *x* of the division of the tree; however, higher than the values inside the current tree; thus, *aux* becomes the right child of this tree. The sub-tree rooted in *b* is correct after the recursive call. Finally, the sub-tree rooted in *t* after the split is assigned to *a*; thus, *a* contains all the values lower or equal to *x* while *b* contains all the values higher than *x*. Symmetrically this algorithm is implemented when value *x* is lower than the value in the current sub-tree.

Algorithm 4 shows how to perform an insertion in a randomized binary search tree. In this tree, besides its pair key/value, each element contains, a variable that represents its balancing. This kind of tree is also is commonly called *Treap* (*tree* + *heap* $\rightarrow$ Treap), because it combines the search properties from a binary search tree with the balancing properties from a binary heap. In a binary heap, every child element from a node is lower than this node. This condition should be maintained to keep the heap property. This data structure is called Cartesian tree as well.

In Algorithm 4, the heap condition is kept by the auxiliary variable *y*, which keeps the balancing of the tree through this property. In other words, let (*x*, *y*) be a pair related to a value *x* in the tree and *y* a randomly selected value, we want to keep the heap property in the tree using the *y* values in a binary search tree related to its *x* values.

Algorithm 4 Insertion operation in a persistent Cartesiantree

```
1 pTreapNode insert(pTreapNode t, pTreapNode it) {
2     t = copy(t);
3     if(!t) {
4         t = it;
5     }
6     else if(it->y < t->y) {
7         split(t, it->key, it->l, it->r);
8         t = it;
9     }
10    else {
11        if(it->key < t->key) t->l = insert(t->l, it);
12        else t->r = insert(t->r, it);
13    }
14    return update(t);
15 }
```

Since it is a persistent Cartesian-tree, the first step consists in copying all the nodes in the path between the tree root and the element inserted. In line 3, we have the base case, when the sub-tree root is null and, thus, the sub-tree root is the current element. After this, we need to treat the tree heap property. That is, if the priority of the current inserted element is lower than the priority of the current sub-tree root, this means that the currently inserted element cannot descend further into the tree. Therefore, the sub-tree is divided by the key of the inserted node and the current node is defined as the root of the two trees generated by the division. Otherwise, the insertion of the new element follows the normal operations of a common binary tree.

Algorithm 5 shows the implementation of a function to merge two binary self-balanced search trees, *l* and *r*, in which the highest value from tree *l* is lower than the lowest value from *r*. When a merge operation occurs, there are two possibilities:

- Use tree l as parent from tree *r*
- Use tree r as parent from tree *l*

After defining the merging policies, recursively calls are made until *l* and *r* are not empty. However, the definition of a static policy to merge the sub-trees can generate a completely unbalanced tree.

Algorithm 5 Merge operation in two binary search trees.

```
1 pTreapNode merge(pTreapNode l, pTreapNode r) {
2    if (!l || !r) {
3       return l ? l : r;
4    }
5    int m = getSize(l), n = getSize(r);
6    if(rand() % (m + n) < m) {
7       l = copy(l);
8       l->r = merge(l->r, r);
9       return update(l);
10   }
11   else {
12      r = copy(r);
13      r->l = merge(l, r->l);
14      return update(r);
15   }
16 }
```

In a fully persistent Cartesian tree, besides variable y used to maintain the heap property, the proportional probability of the size of the tree is used to define the policy used in this iteration. Therefore, two different executions of joining the same trees can yield different results, but maintaining an amortized logarithmic height in both cases. Line 6 of Algorithm 5 shows the probabilistic choice of policy, as presented. In the algorithm, the rand function returns a random integer.

Figure 7 shows the union of two trees *L* and *R*, in which the highest value of *L* is lower than x, while the lowest value from *R* is higher than *x*. The blue node is the root resulting from the merging operation, which could be a node from *L* or a node from *R*. The larger the number of nodes in a tree, the greater the expected value of its height is. For this reason, the algorithm is more likely to root the tree with the most nodes.

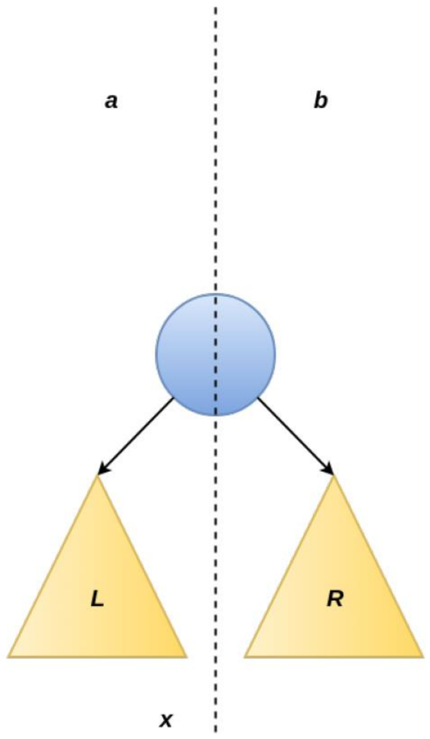

Fig. 7: Example merging of two binary search trees. Source: The Authors

The merging function eventually allows creating a delete function of an element in a Cartesian-tree. Due to the persistent nature of the structure, it is necessary to copy all the way to the removal of the node, which is performed by checking line 3 in Algorithm 6.

Algorithm 6 Delete operation in a persistent Cartesiantree

```
1 pTreapNode erase(pTreapNode t, K key) {
2     t = copy(t);
3     if (t->key == key) {
4         t = merge(t->l, t->r);
5     }
6     else {
7         if(key < t->key) t->l = erase(t->l, key);
8         else t->r = erase(t->r, key);
9     }
10    return update(t);
11 }
```

Empirical Analysis

After the implementation, this data structure was tested under some data sets. In these data sets, the operations are executed consistently. In other words, any operation in the timeline is consistent and can be executed. For example, we do not execute an operation Pop in an empty data structure. The data sets were generated such that every possible operation had the same chance of being executed. These data sets have a fixed temporal timeline of 10^5, i.e., all data structure versions exist in a time interval between 1 and 10^5 and all the operations were chosen randomly.

These tests were executed using an Intel Core i5-4200U CPU@1.60GHz x 4 processor and 8 gigabytes of memory. For measuring the results, we used the gtest tool, to obtain the empirical time complexity and valgrind, to get the memory consumption for each of the implementations.

All the algorithms used in this article can be found at https://github.com/juniorandrade1/Master/blob/master/src/Priority_Queue/ and the data sets can be found at https://github.com/juniorandrade1/Master/tree/master/tests/Datasets/Priority_Queue.

We use this pre-generated data sets to test the performance of three different implementations of a fully retroactive priority queue. The first implementation is the one proposed here, using checkpoint-tree and persistent binary search trees based on (Demaine *et al*., 2015) work. The second is the implementation of an algorithm proposed by (Demaine *et al*., 2007), which used a square-root decomposition technique with the partial retroactive priority queue to perform all updates and queries at time $O\left(\sqrt{m}\cdot\lg(n)\right)$. The third implementation is a brute force algorithm; when a *GetPeak*(t) operation is performed, all the edges added at time $t' < t$ are added to the graph and, later, a standard shortest path algorithm is executed in this graph.

Figure 8 shows the tests performed comparing the performance among three different implementations of the fully retroactive priority queue.

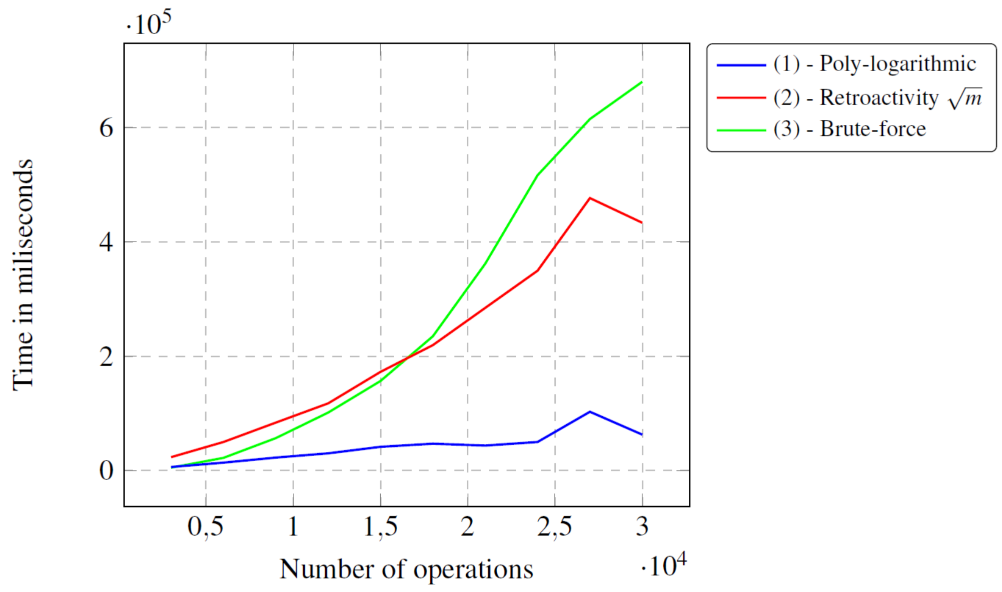

Fig. 8: Performance test using random test cases. Source: The Authors

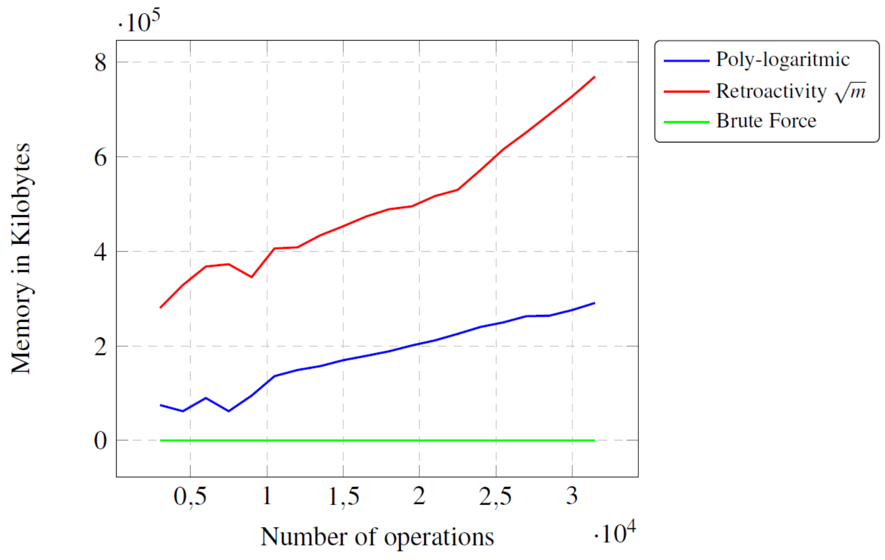

Fig. 9: Memory consumption in a fully retroactive priority queue. Source: The Authors

The brute-force algorithm is faster than the second one in the first 1500 operations and, after that, the second algorithm is faster. This occurs because the constants in the brute-force algorithm are lower than the constants in the second algorithm. Also, the insertion/deletions in the bruteforce algorithm are faster than in the retroactive approach. Conversely, the GetPeak(t) operations is slower than the algorithm that uses the square-root technique.

The first algorithm has a stable performance when the number of operations increase and also performed better than the other two algorithms, on average. This algorithm only presented a worse performance in very few cases, whereby the large implicit constant in the algorithm complexity made the poly-logarithmic algorithm slower as compared to the other ones.

Figure 9 shows the memory consumption when the tests were performed.

The brute force algorithm presented a very small memory consumption compared to the other two algorithms. The brute force algorithm maintains the operations performed ordered, consuming a long time when the queries are executed, but consuming a small amount of memory. The algorithm Retroactivity-$\sqrt{m}$ proposed by Demaine *et al*. (2007) consumes a large amount of memory because, besides storing $\sqrt{m}$ partial retroactive priority queues, when an update operation is performed, the inserted object will be added in $\sqrt{m}$ partial retroactive priority queues, in the worst case. The poly-logarithmic algorithm proposed by (Demaine *et al*., 2015) maintains a checkpoint-tree which, in the worst case, modifies lg (m) partial priority queues. This explains the difference of memory consumption in these two algorithms.

Conclusion

This article presents an implementation to the fully retroactive priority queue using the checkpoint-tree technique proposed by Demaine *et al*. (2015) and fully persistent self-balanced binary search tree. The algorithm implemented in C++ using checkpoint-tree and persistent binary search trees performed better than the algorithm implemented using the square-root technique in terms of time complexity.

Using the persistent self-balanced binary search tree allowed implementing an algorithm to merge two partially retroactive priority queues without losing information about the trees before executing the operations.

In future researches, we intend to measure the influence of using other types of persistent binary search trees when merging two partially retroactive priority queues and also propose some applications in which we can use the poly-logarithmic retroactive priority queue along with the checkpoint tree idea.

Funding Information

This work was supported by the National Council for the Improvement of Higher Education (CAPES).

Author’s Contributions

All the authors have equally contributed in this work.

Ethics

There are no ethical issues associated with this research.

References

Chen, L., Demaine, E. D., Gu, Y., Williams, V. V., Xu, Y. and Yu, Y. (2018). Nearly Optimal Separation Between Partially and Fully Retroactive Data Structures. arXiv preprint arXiv:1804.06932.

Demaine, E. D., Iacono, J. and Langerman, S. (2007). Retroactive data structures. ACM Transactions on Algorithms (TALG), 3(2), 13-es.

Demaine, E. D., Kaler, T., Liu, Q., Sidford, A. and Yedidia, A. (2015, August). Polylogarithmic fully retroactive priority queues via hierarchical checkpointing. In Workshop on Algorithms and Data Structures (pp. 263-275). Springer, Cham.

Dickerson, M. T., Eppstein, D. and Goodrich, M. T. (2010, September). Cloning voronoi diagrams via retroactive data structures. In European Symposium on Algorithms (pp. 362-373). Springer, Berlin, Heidelberg.

Driscoll, J. R., Sarnak, N. and Sleator, D. D. (1989). RE Tar j an. Making data structures persistent. J ournal o f C omputer and S ystem S ciences (JCSS), 38(1), 86-124.

Goodrich, M. T. and Simons, J. A. (2011, December). Fully retroactive approximate range and nearest neighbor searching. In International Symposium on Algorithms and Computation (pp. 292-301). Springer, Berlin, Heidelberg.

Henzinger, M. and Wu, X. (2019). Upper and Lower Bounds for Fully Retroactive Graph Problems. arXiv preprint arXiv:1910.03332.

Kaplan, H. (2018). Persistent data structures. In Handbook of Data Structures and Applications, pages 511–527. Chapman and Hall/CRC.

Martínez, C. and Roura, S. (1998). Randomized binary search trees. Journal of the ACM (JACM), 45(2), 288-323.

Sarnak, N. and Tarjan, R. E. (1986). Planar point location using persistent search trees. Communications of the ACM, 29(7):669–679.

Sunita and Garg, D. (2018). Dynamizing Dijkstra: A solution to dynamic shortest path problem through retroactive priority queue. Journal of King Saud, In press.

Tangwongsan, G. E. and Blelloch, U. A. (2007). Nonoblivious retroactive data structures. Technical report.