Retroactive Priority Queues in Python

Parker Rule

May 18, 2021

1 Introduction and literature review

This project is a Python implementation of retroactive priority queues with an emphasis on readability and data structure reuse. Demaine et al. [DIL07] describe retroactive data structures as a "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." Any data structure can be made retroactive by keeping an event log and replaying events after a change in history; a typical goal in the design of retroactive data structures is to improve over the performance of this naive general transformation. A *partially retroactive* data structure supports updates and queries to the current state ($t = \infty$) and updates to the past; a *fully retroactive* data structure also allows queries in the past. We fully implement a partially retroactive priority queue and partially implement a fully retroactive priority queue.

Demaine et al. [DIL07] give a partially retroactive priority queue that supports **insert** and **delete-min** operations in $O(\log m)$ time, where m is the number of total operations, a significant improvement over the naive $O(m)$ solution. Operations can be deleted, so there are four possible events: **insert**, **delete**(insert), **delete-min**, and **delete**(delete-min). Priority queues are interesting because they are highly nonlocal—one operation can cause cascading effects on the lifetimes of other elements in the priority queue. However, each operation only results in small changes to the current state Q_{now} (at most one element element in and one element out per operation). Demaine et al. make use of *bridges* to give simple update rules for each of the four possible events. (A bridge is a time t in the past where $Q_t \subseteq Q_{\text{now}}$—that is, the elements in the queue at time t are a subset of the current elements.) These updates can be performed in $O(\log m)$ by storing Q_{now}, an event tree (a linked list in the original paper), an updates tree, and a tree of insertion operations in augmented balanced binary search trees. (Each of these trees has size $O(m)$, so the overall data structure has size $O(m)$.) Specifically, we perform queries on the updates tree of the form "what is the last time before t (or first time after t) when the events had the prefix sum x?" These queries are used to find bridges, which correspond to a prefix sum of 0; they can be answered efficiently by augmenting internal nodes with the subtree sum and the minimum and maximum subtree prefix sums. Similarly,

the tree of insertion operations is augmented to keep track of the minimum element in Q_{now} and the maximum element not in Q_{now} at each internal node.

Later work by Demaine et al. [Dem+15] describes a fully retroactive priority queue that supports $O(\text{polylog } m)$ time and $O(m \log m)$ space using *hierarchical checkpointing*. The basic idea is to insert each operation in separate partially retroactive priority queues at the $O(\log m)$ levels of a main tree. (This is somewhat analogous to a segment tree.) We then can construct a view of the priority queue at time t by merging an $O(\log m)$-size set of queues that covers the internal $[0, t]$ (or, more generally, $[t_{\text{start}}, t_{\text{end}}]$). Augmenting each internal node with a partially retroactive priority queue—itself a complicated data structure that contains multiple augmented trees—means that rebalancing the main tree is challenging. However, the tree can easily be rebuilt from the bottom up in $O(m \log m)$ time, so we can use a *scapegoat tree*—a balanced binary search tree that rebuilds unbalanced subtrees all at once, rather than performing local rotations—to achieve updates in $O(\log^2 m)$ amortized time. When rebuilding a tree of size m only takes $O(m)$ time, the scapegoat tree has the same amortized performance guarantees as more familiar balanced binary search trees [GR93]. My implementation uses the scapegoat tree to implement all augmented binary search trees.

Fusing $O(\log m)$ partially retroactive priority queues—or even two such queues—is nontrivial. The fusion algorithm in [Dem+15] relies on several specialized algorithms and data structures. In particular, *weight-balanced B-trees* are used to represent Q_{now} in each partially retroactive priority queue. Q_{del}, which is used to keep track of deleted elements, is similarly stored as a weight-balanced B-tree. These trees, which can be thought of as a generalization of BB-α trees, were developed by Arge and Vitter [AV03] and Bender et al. [BDF05] in the study of cache-oblivious data structure, but they are useful for finding approximate order statistics as well. Unfortunately, this fusion algorithm is destructive, as copying queue elements in an arbitrary time range may take $O(m)$ time.

2 Data structures

I have implemented several data structures relevant to retroactive priority queues. My implementations rely on metaprogramming and advanced Python constructs such as `yield from`; they contain type annotations and docstrings in most places. Most functions are recursive and not optimized for speed.

2.1 Weight-balanced B-tree

The `WBBTree` class (code) is largely based on [AV03] and handles deletions lazily rather than implementing the eager deletion procedure described in [BDF05]. (Additionally, no special attention is paid to memory layout.) Eager deletion, as well as split and concatenate, are left as future work. A cursory analysis suggests that eager deletion is necessary to compute the approximate order

statistics used by the fusion algorithm in [Dem+15] in $O(1)$ time; this algorithm also uses split and concatenate operations.

2.2 Scapegoat tree

The `RangeTree` class (code) is a generic scapegoat tree with key ranges as internal nodes and keys as leaves.[1] We follow some of the implementation details given in *Open Data Structures* [Mor19] but allow a custom α parameter. (The α parameter determines how unbalanced the tree can be, so the choice of α represents a tradeoff between update perfrmance and query performance.) Users of the class must provide their own subtree rebuilding logic (`rebuild_fn`) and can provide a metadata class that augments each node. A user-provided metadata class must implement an abstract base class (`RangeMeta`) that updates node metadata when a key-value pair is inserted into or removed from a node's range. For convenience, we provide a factory (`make_agg_meta`) that generates range tree classes with simple aggregations built in. For instance, the `SumRangeTree` class stores subtree sums as metadata.

2.3 Tree augmentations

We use `RangeTree` to implement a variety of augmented search trees. The `PrefixSumTree` class (code) supports the prefix sum queries necessary for partially retroactive search trees via the `first_node_with_sum` and `last_node_with_sum` methods. Likewise, `InsertTree` (code) supports queries of type "minimum element in Q_{now}" and "maximum element not in Q_{now}." Finally, a basic fully retroactive priority queue (code) stores partially retroactive priority queues in its metadata objects.

2.4 Partially retroactive priority queue

The `PRPriorityQueue` class (code) is a thin wrapper around a collection of weight-balanced B-trees and augmented scapegoat trees, as described above. We implement the bridge-based update operations given in [DIL07].

2.5 Fully retroactive priority queue

The `PriorityQueue` class (code) does not fully implement the fusion algorithm given in [Dem+15], but it does store partially retroactive priority queues in a main tree (an augmented scapegoat tree). We implement an inefficient copy-based fusion function based on the set-theoretic definitions in [Dem+15].

[1] Despite the name, this data structure is unrelated to the range trees of Bentley and Saxe [BS80].

3 Tests

The invariants of weight-balanced B-trees and scapegoat trees are rather complicated. We employ invariant-based testing: the `WBBTree` and `RangeTree` classes both provide `check_invariants` methods which are used in automated tests over dozens of access sequences. Most access sequences are random, but we include several pathological cases (ascending and descending sequences, plus the bit reversal sequence). The invariant tests as written (code) take several minutes to run—there are approximately 750 tests between the two classes—but this performance could be improved by tweaking parameters. The `PrefixSumTree` class is tested similarly (code) and a few manually written tests are included for the `PRPriorityQueue` class (code).

4 Usage

Most data structures contain thorough inline documentation. We include a usage demo of `PRPriorityQueue` below.

```
>>> from retroactive_pq.partial_pq import PRPriorityQueue
>>> q = PRPriorityQueue()
>>> q.insert(1)
>>> q.insert(2)
>>> q
Qnow: 1 2
events:
1.0: insert 1
2.0: insert 2

>>> q.insert(3, t=2.5)
>>> q.delete_min(t=2.25)
>>> q
Qnow: 2 3
events:
1.0: insert 1
2.0: insert 2
2.25: delete min
2.5: insert 3

>>> q.delete_op(2.25)
>>> q
Qnow: 1 2 3
events:
1.0: insert 1
2.0: insert 2
2.5: insert 3
```

5 Challenges and future work

This work could be expanded and improved in a number of ways, including:

- **Implementation of related partially retroactive data structures.** Implementing partially retroactive deques, as described in [DIL07], would require no new augmented data structures.
- **Optimal implementation of fully retroactive priority queues.** This would require modifications to the weight-balanced B-tree algorithm, as described. It would also require an implementation of the linear-time order statistics algorithm given in [RA10]. Though time constraints prevented me from pursuing these modifications, all basic data structures are in place.
- **Performance improvements.** Another 6.851 project implements partially retroactive priority queues using treaps. Though I have not performed careful benchmarks, this implementation is likely more performant, as it uses fewer layers of indirection than mine—my design is oriented around the specific data structures mentioned in [Dem+15].
- **Generalization.** This project could be used as a basis for a collection of integrated retroactive data structures, similar to Chelsea Voss' 6.851 final project.

References

[AV03] Lars Arge and Jeffrey Scott Vitter. "Optimal external memory interval management". In: *SIAM Journal on Computing* 32.6 (2003), pp. 1488–1508.

[BDF05] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. "Cache-oblivious B-trees". In: *SIAM Journal on Computing* 35.2 (2005), pp. 341–358.

[BS80] Jon Louis Bentley and James B Saxe. "Decomposable searching problems I. Static-to-dynamic transformation". In: *Journal of Algorithms* 1.4 (1980), pp. 301–358.

[Dem+15] Erik D Demaine et al. "Polylogarithmic fully retroactive priority queues via hierarchical checkpointing". In: *Workshop on Algorithms and Data Structures*. Springer. 2015, pp. 263–275. URL: https://erikdemaine.org/papers/FullyRetroactive_WADS2015.

[DIL07] Erik D Demaine, John Iacono, and Stefan Langerman. "Retroactive data structures". In: *ACM Transactions on Algorithms (TALG)* 3.2 (2007), 13–es. URL: https://erikdemaine.org/papers/Retroactive_TALG.

[GR93] Igal Galperin and Ronald L Rivest. "Scapegoat Trees." In: *SODA*. Vol. 4. 1993, pp. 165–174. URL: https://people.csail.mit.edu/rivest/pubs/GR93.pdf.

[Mor19] Pat Morin. *Open Data Structures*. 2019. URL: https://opendatastructures.org/newhtml/ods/latex/scapegoat.html.

[RA10] André Rauh and Gonzalo R Arce. "A fast weighted median algorithm based on quickselect". In: *2010 IEEE International Conference on Image Processing*. IEEE. 2010, pp. 105–108.