Balancer — Analysis

Jared Dyreson

January 13, 2022

Contents

Abstract

Current implementations of priority queues in the C++ Standard Template Library do not allow for on demand reordering. Container objects are protected member attributes and cannot explicitly be modified. This is a safety mechanism to not allow the user of the data structure to unintentionally break it. However, since the goal is to bypass this, Balancer is a class that inherits from the base priority queue structure and provides this interface. To still retain the original safety feature contained in the STL; only classes that are friends can request the data structure to reorder. This also ensures that the structure cannot erratically change its state during the runtime of the program.

1 Introduction

This solution was provided by here by a user that suggested to inherit from the base std::priority_queue data structure.

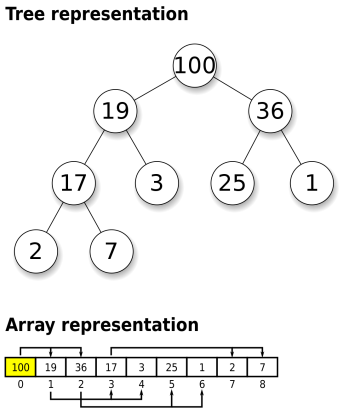

Figure 1: Example of Priority Heap

Priority queues with a predicate function can influence how it's structured. These functions can be tailored to the objects that are stored in the container. In most use cases, node objects should have a time stamp object that indicates how old it is. Older objects are generally assigned higher priority to avoid starvation, along with other criteria provided. These criteria are usually in the form of a predicate function.

2 Use Cases

Various types of programs can benefit from a reordering on demand paradigm. CPU schedulers, fast food preparation and reservation systems could see an efficiency increase by using such a data structure. All of these applications have some aspect of time and dynamic data that can render the tree stale.

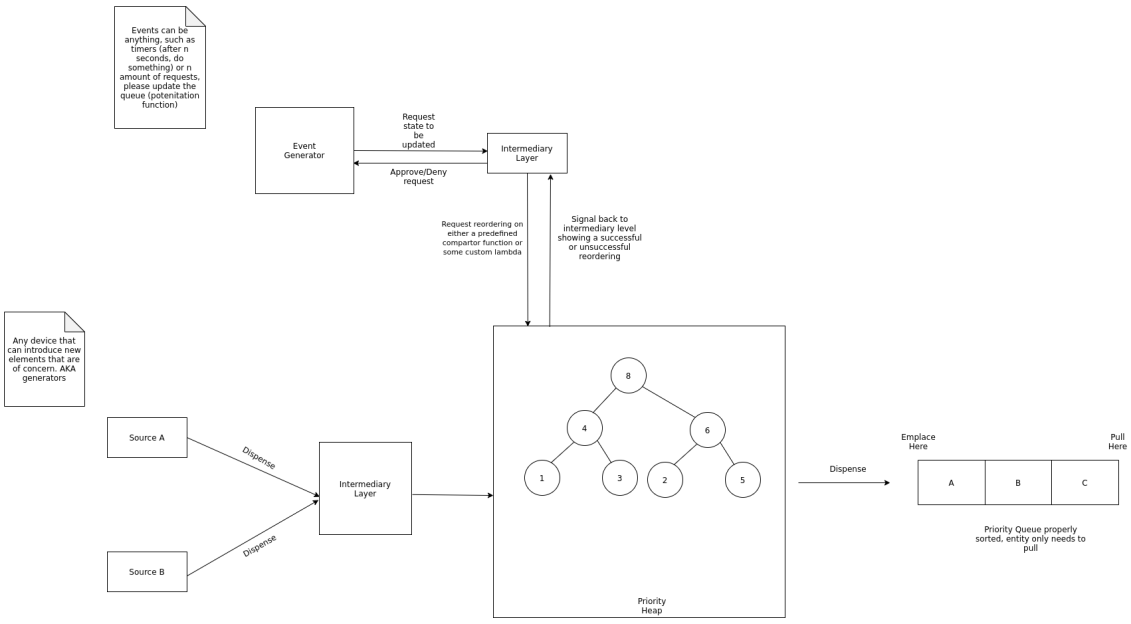

Figure 2: Example system

This sketch shows the newly designed priority heap in action with respect to generalized inputs and outputs. An arbitrary number of requests are spawned, which are then fed into an intermediary level that will then funnel into the queue. Upon insertion, the queue will reorder itself based on it's comparator function. This still retains functionality from the original priority queue in STL, however also checks for the element's presence. Some implementations can use a std::vector, however this checking process is $O(n)$ time, as all the elements would need to be evaluated. We are also operating under the assumption that there is a potential for duplicate elements to be introduced into the system, which can stem from improper use of threads. Please refer to the mathematical and algorithm analysis sections further down to get a more detailed explanation.

3 Events

Events are the only means of updating the state of the structure and are strictly outside influences to the system. Timers are a common event which are outside the confines of the system and can be implemented using `std::thread_sleep`. More advanced events could measure the internal entropy of the system, such as the number of requests given or times the system reorders. Such functions can be expressed using the following equation:

$$y_k = \phi \left(\sum_{j=0}^{m} w_{kj} \cdot x_j \right)$$

y_k denotes the strength of the output given by the inputs along with their respective weights. Weights are used to signal how "important" an input is relative to others and can be generated using another outside algorithm. For example, larger drink orders will hold a higher weight relative to smaller orders from another source. The equation provided is the artificial neuron firing function, and it can be amended to fit this situation. Once a predetermined threshold has been met, the system can then be prompted to reorder. It is important to have a reasonable threshold because the system could reorder too frequently and diminish efficiency.

4 Race Conditions

One important factor to consider is how will the system operate when the heap is reorder. A simple yet effective strategy is to apply a mutex over the resource, locking it from other processes that may wish to access the heap. It is also necessary to note that this is another form of the Banker's Algorithm, wherein multiple processes are vying for resources.

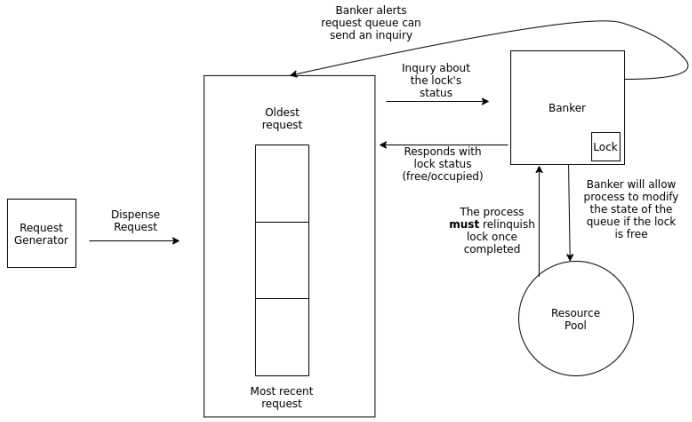

Figure 3: Banker's Algorithm

An external class should manage the state of these processes and shall grant/deny requests. Another approach would to shift the implementation to Rust, as it has a strong concurrency framework built into the language. Further analysis needs to be conducted at a later date.

5 Algorithm and Mathematical Analysis

5.1 Reordering Algorithm — make_heap

5.1.1 Pseudocode

```
function reorder {
      make_heap(underlying_container.begin,
                underlying_container.end,
                comparator)
}
```

5.1.2 Mathematical Analysis

The current implementation of `std::make_heap` operates at $O(n)$ time and the source code can be found here. To construct a binary heap, it can be broken down into two steps:

- Put the nodes, in any order, into a complete binary tree of the right size
- For each node, starting the bottom layer and going upward, run the bubble-down step on that node

At most half of the elements start one layer above that and can move down at least once. At most a quarter of the elements start one layer above that and can move down at least twice. More generally, at most $\frac{n}{2^k}$ of elements can be moved k steps. The upper bound can be defined as this sum:

$$T(n) \leq \sum_{i=0}^{\lceil \log_2 n \rceil} \frac{n\, i}{2^i} = n \implies \sum_{i=0}^{\lceil \log_2 n \rceil} \frac{i}{2^i}$$

Where `n` is the number of elements in the container and `i` is the level inside the heap. There is more to come...

5.2 Proposed Solutions

Currently, there are two different approaches that can be taken:

- Sorting a std::vector — $O(n\ \log(n))$
- Reorder half of a red black tree — $O(\frac{n}{2}\log(\frac{n}{2}))$

5.2.1 Sorting a Vector

Data Structure	Best (Ω)	Average	Worst (Θ)
std::vector	$n\ \log(n)$	$n\ \log(n)$	$n + n\log(n)$

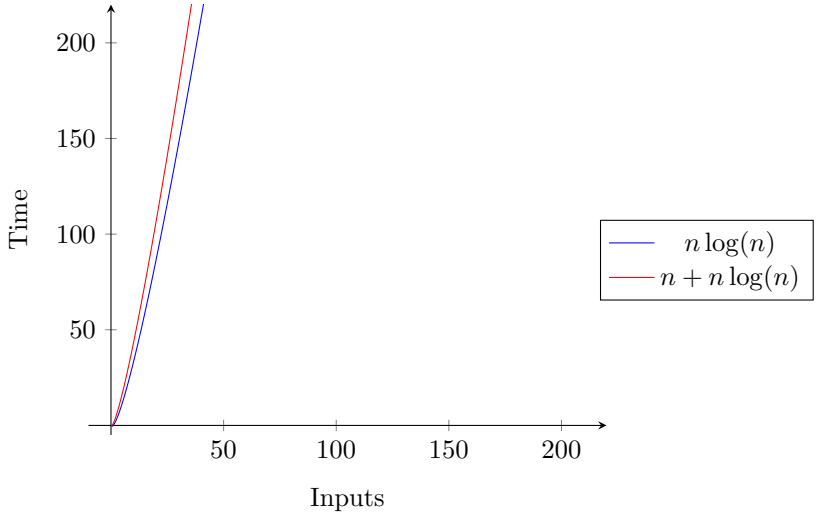

Figure 4: Time Complexity Differences

The first solution is relatively easy to implement, as it wraps around a pre-existing STL data structure[1]. This, however, comes at the cost of sorting which is bounded in linearithmic time. We can attempt to amortize the time complexity by increasing the length of time thread events are allowed to execute. An internal timer can also be embedded in the class to prevent further resorting, akin to the Bankers Algorithm outlined in a previous section. As stated above, thread events can include the measured entropy of the system. By increasing the threshold for the firing function, we can drive down the number of reordering requests. However, this introduces another issue of starvation as some elements in the container may never get to the top of the heap. Emergency restructuring of the container could lead to $O(n + n\log(n))$ time. This is because we are forced to first assess all elements given a set of criteria, re-assign priority and lastly resort the container.

[1] std::vector

5.2.2 Reorder Red Black Tree

The second solution, however, is going to be harder to pull off. In the diagram below, a tree like structure would represent our data, where the left hand side contains less important elements and vice versa for the right.

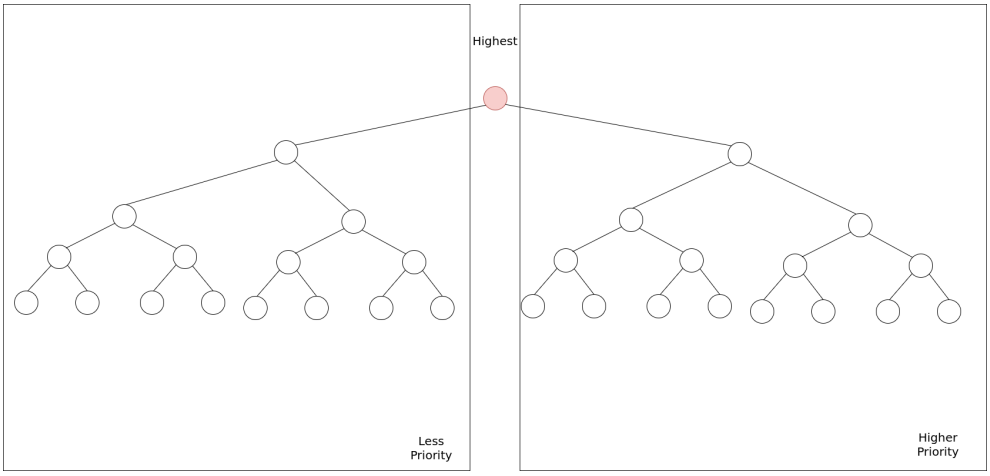

Figure 5: Split Red Black Tree