A Heap Or Priority Queue

William Frid

Fall 2023

Introduction

When developing software one might need to know how to handle different types of priority queues. This can be done in different ways depending on what strengths the queue needs. This report will focus on exploring three types of queues: a queue built with a links list, a binary heap, and an array heap. While a normal queue (linked list) is ordered after priority, the two heaps are tree-like structures that give great advantages in speed as the tree comes with logarithmic time complexities. They are thus more commonly used for acting as priority queues than the linked list queue.

This report will focus on how these three options work and how well they perform to highlight their advantages and disadvantages. Some code snippets are added to give an understanding of how the different functionalities might look in Java 8, and of course, benchmark results with comparisons and discussion.

Lists As Queues

When using lists as queues, one can take two different approaches: Either keep the list ordered by priority or give each node a priority value to be compared. These two different behaviors affect the time complexity of both the algorithms used for adding and removing from the heap. Let's start with **type 1**, which isn't ordered. Because it's unordered, the time complexity becomes $O(1)$ upon adding, and $O(n)$ when removing because it has to compare until it has found the node with the highest priority. Next is **type 2**, which is reversed compared to **type 1**. **Type 2** keeps the list sorting by priority upon add, meaning the add algorithm has the time complexity of $O(n)$ and the remove algorithm has $O(1)$.

To better understand how this works, a snippet was posted below. The snippet displays the method `add()` used in **type 2**. It works in the way that it creates a new node and inserts it as the head if the list is empty. Otherwise, it checks if the new node should replace the head, and if not, it continues to a while loop that runs until the correct spot for the node is

located. This method is very similar to the method `remove()` in list **type 1**. Thus they have the same time complexity; $O(n)$.

```
void add(Integer p, Integer v) {
    Node n = new Node(p, v);
    if (head == null) {
        head = n;
    } else {
        Node i = head;
        if (n.priority.compareTo(i.priority) < 0) {
            n.nxt = i;
            i.prv = n;
            head = n;
            return;
        }
        while (i.nxt != null && i.priority.compareTo(p) < 0)
            i = i.nxt;
        i.nxt = n;
        n.prv = i;
    }
}
```

These two types of structures thus come with separate advantages and disadvantages. Starting with an analysis, we can conclude that **type 1** should be more effective when adding is performed more frequently than removing. Meaning that speed is more important upon adding. **Type 2** on the other hand has fast removals, which can be important when high-priority nodes need to be handled as quickly as possible.

A benchmark was run to test the general speed for these two types of lists in two versions, one that inserted all elements, then removed all elements, and a second one that added and removed one by one so that the list remained short. Looking at Table 1 and 2 we can see that **type 2** performs slightly better in both cases. Meaning that in the average case, **type 2** outperforms **type 1**. But **type 1** might still be preferred in some cases since it adds faster since there's nothing to remove if nothing has been added.

Size	List Type 1 (us)	List Type 2 (us)
10	6	2
20	2	3
40	2	3
80	4	5
160	9	10
320	16	16
640	31	30
1280	70	69
2560	150	150
5120	330	290
10240	740	550

Table 1: Benchmark of inserting all followed by removing all.

Size	List Type 1 (us)	List Type 2 (us)
10	2	1
20	2	2
40	3	3
80	4	4
160	9	10
320	15	15
640	30	30
1280	69	68
2560	150	140
5120	310	260
10240	660	550

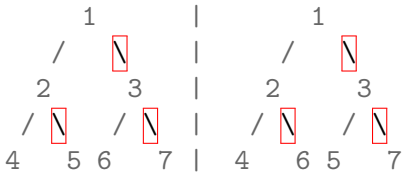
Table 2: Benchmark of removing directly after inserting.

Binary Heap

Another way of handling queues is with binary heaps. These heaps are forms of trees that are also ordered by priority, known as complete binary trees. Meaning that they hold nodes of descending priority downwards, and populate each depth level from left to right. Thanks to this (somewhat sorted) tree structure, it is possible to make use of algorithms similar to binary search, giving use to high speeds and time complexities like $O(log(n))$ for both adding and removing.

The binary heap created for this report is somewhat different from what was instructed as it uses the principle "left leaves first, then right leaves". Thus it doesn't follow the standard binary heap since it isn't considered

complete but still gives the understanding of traditional binary heaps. The binary heap built keeps the branches even at all times. It does this by populating from the left, only the left leaves, and then continues with the right leaves from the left. In addition, upon calling the method `remove()` to remove the root, it also runs a method called `heapify()` which sorts what is needed (sometimes nothing, and other times $log(n)$ elements. The figure below shows a traditional binary heap to the left and the one that was built for this report on the right.



When working with a binary heap and one wants to push the first node further back, there are two options. First first one is to simply remove the node, update its priority, and add it back in. This can however be expensive since it requires more reads, writes, and comparisons. The second option is pushing, where one simply decreases and calls upon a method that can push it into order further down the tree. Now this last step could be done using the method `heapify()` mentioned earlier.

To test this, a benchmark was run for different sizes. It tried each size 10,000 times to find the best run with the least interference and increased the priority value each time with a random number between 10 and 100. Comparing the columns `Push (us)` and `Remove & Add (us)` in Table 3 tells us that the push operation is indeed the faster option, no matter the size. These values will however vary drastically depending on how the priority value changes. If the change in priority in most cases could push the node to the furthest back, then the "remove and add"-approach would be faster.

Size	**Push (us)**	**Remove & Add (us)**	**Ratio**
40	5	7	0.70
80	11	16	0.71
160	24	32	0.75
320	54	72	0.75
640	120	160	0.74
1280	250	340	0.74
2560	550	780	0.70
5120	1300	1790	0.70
10240	3000	4000	0.76

Table 3: Benchmark exploring the advantage of push.

Heap Array

A third option for building a heap is using an array. This type of array doesn't hold the data ordered directly by priority. Instead, it holds that data is ordered as a binary heap (see section "Binary Heap"). Imagine a complete binary tree. copy the contents of each depth level and paste it into the array heap (you could say that you flatten a binary heap into one dimension). Now for navigating the array, one uses four different formulas: $(n-1)/2$ and $(n-2)/2$ to find the parent of a node (depending on if the node has an even or odd index), and $2n+1$ or $2n+2$ to find the children (same goes for odd and even). *Note that the two formulas used for finding parents can in some cases be written as a single one due to rounding done by the programming language.*

The heap structure comes with two main methods: `add()` and `remove()`. The method `add()` simply adds a new node to the end of the array, then uses recursion to push it upwards in the tree into its correct position using a helper method called `swap()`. The method `remove()` works similarly, but in reverse. It returns the root element and replaces it with the last element in the heap. Then it too uses recursion to push it downwards in the tree using `swap()` and the logic to swap with the smaller child node.

Lastly, there's a method called `resizeArr()` which is called upon when needed. It doubles the size of the array heap when it's being added to but is full, and cuts it in half when it's only a quarter full. Below is a code snippet displaying the way the adding was handled.

```
public void add(int item) {
    if (end + 1 > arr.length - 1)
        resizeArr(true);
    arr[++end] = item;
    addHeapify(end);
}
private void addHeapify(int i) {
    if (end == 0)
        return;
    int p;
    if (i % 2 == 0) // Even.
        p = (i - 2) / 2;
    else // Odd.
        p = (i - 1) / 2;
    if (p > -1 && arr[i] < arr[p]) {
        swap(i, p);
        addHeapify(p);
    }
}
```

Analyzing the algorithm, we get that the time complexity for the method `add()` is a combination of its adding action and pushing action. We know that the adding is instant, but the push is logarithmic thanks to the binary structure of this heap. Thus we get the time complexity of $O(log(n))$ as the logarithmic part is dominant compared to the adding step. Next, we have the algorithm used for deleting. Here we can conclude faster since it too has two steps: The first is instant (removal), and the second (pushing) is logarithmic. Thus the removal has time complexity $O(log(n))$. In consolation, we see that this type of structure performs as well when both adding and removing.

To test this type of heap's speed, a benchmark was set up similar to the one run for the binary heap. A benchmark that inserted n many nodes, and removed them all. The test did this for different sizes 20,000 times to find the best results presented in Table 4. If we compare these results to column `Push (us)` in Table 3, we can see a clear similarity between the behavior of time increase. Both the push for the binary heap and adding for the array heap increase similarly thanks to their binary characteristics.

Size	**Time (us)**	**Time/Size**
40	2	44
80	5	49
160	10	56
320	20	63
640	45	70
1280	100	79
2560	220	86
5120	450	88
10240	1000	100
20480	2200	110

Table 4: Benchmark of array heap where n element was added, then removed.

Discussion

Due to the mistake made during the development of the binary heap some benchmark results and comparisons might not have been spot on, but should according to calculations and comparisons (to peer's results) still highlight the different advantages and disadvantages between all these methods. In addition, separate benchmarks should have been run to compare the four different methods individually for the linked list queue, since this data could have been clearly displayed in a linear graph that would better describe the reason for the similar results in Table 1.

Lastly, the array heap could benefit from benchmark results that also

proved the time complexity of the methods. This was not included in the report because of the main purpose of showing how it acts similar to a binary heap, but somewhat faster thanks to it being an array and because of that takes greater advantage of cache.

Conclusion

The conclusion drawn from the code written, benchmarks run, and algorithms analyzed, it's easy to say that there are two groups in this report: the linked list **queue**, and the **heaps** built as both binary and array. The queue is better suited for simpler tasks, such as storing data, while the heap is better when working with complex priority queues or other algorithms where the heap characteristics are an advantage. The type of heap to be used can however vary depending on the specific case as there are max- and min-heaps, plus different ways of constructing them.