# Java PriorityQueue Class

Anders Jacobsen, Dima Karaush

May 7, 2021

**Abstract**

Java's PriorityQueue class has a bottleneck when one needs to update values in the queue. Depending on the use and implementation, Java's PriorityQueue can add serious performance issues when accessing the data not using the poll() method. When updating a value in the queue, performance can be improved by 99% in a "worst-case-scenario" compared to Java's PriorityQueue implementation. Implementing your own version might remove this bottleneck from your software.

## 1 Introduction

This article will discuss Java's "build-in" class `PriorityQueue`. The interest for this topic has grown from an implmentation of a weighted graph in Java, which we found had a possible shortcomming for our use-case. To update a value in the queue, we had to use a linear search (loop) through the queue. That is fine on a small scale, but a complexity of O(n) can make upscaling a difficult task. We believe this can be optimized and that is why this article explores this subject.

We plan to use Peter Sestofts Mark5 benchmarking techniques from his article "Microbenchmarks in Java and C#" [1] to bencnhmark the two PriorityQueue classes. We will do seperate benchmarking with warmup iterations on each of the implementations. Afterwards we will explore the data using a Python Notebook. Then we compare the results to explore our findings.

The entire project and files can be found on GitHub, the repository is located here `https://github.com/Cosby1992/UFO_Exam`.

## 2 Scope

This article aims to compare the performance between Java's PriorityQueue and a PriorityQueue we designed and implemented. The article will only include perfomance comparisson for the task to update an Object within. It will not compare the enqueing or dequeing performance times.

# 3 Problem

## 3.1 Background

Java's PriorityQueue implemented in a heap struckture, this means that there is no way to quickly find an Object within. Therefore it is nessesary to do a linear iteration through the queue. This is made possible by the iterator method inherited in the PriorityQueue class. What we explore is how much faster it would be to use binary search in a sorted array, where the array is representing a queue instead.

## 3.2 Problem Statement

1. How fast is Java's current implmentaion, when the task is to update a value within?
2. How fast is our implementation of PriorityQueue using binary search to find and update a value within?
3. How do the queue implementations compare in performance?

# 4 Analysis

## 4.1 Benchmark Method

This subsection will cover the preperation and execution of the benchmarks. It will also touch the subject of the timings in the two implementations of the PriorityQueue class. Peter Sestofts approach to microbenchmarking in Java [1] is used to benchmark the implementaions. A combination of Mark5 and Mark3 benchmarks with minor changes is used to generate the timing results. To make benchmarks, a Timer class is nessesary. We've designed the a simplest version possible to avoid interference from calculations in the Timer class.

```
1   public class Timer {
2       private long start;
3
4       public void start() {
5           start = System.nanoTime();
6       }
7
8       public long step() {
9           return System.nanoTime() - start;
10      }
11  }
```

Listing 1: Simple Timer class implementation

This implementation enables us to process the times as nano seconds after the benchmarks and also to easily restart the timer.

In addition to the Timer class we have also implemented a TimerTracker class. This class only consists of two lists that can contain the warmup and real benchmark times, together with a method for writing the obtained times to a CSV[1] file. The CSV files will be used to explore the data later.

```
1     private static void benchmarkPriorityQueue(int
      warmupIterations, int iterations, TimeTracker
      tracker) {
2         // Printing removed for simplicity
3         // Warmup
4         for (int i = 0; i < warmupIterations; i++) {
5             tracker.addWarmupTime(pQueueRun());
6         }
7         // Benchmark
8         for (int i = 0; i < iterations; i++) {
9             tracker.addTime(pQueueRun());
10        }
11    }
```

Listing 2: Benchmark iterations

Measurements in the benchmark is done only on the time it takes to update a value in the PriorityQueue. In listing 2 we run a number of warmup iterations before running the benchmark. This is to make sure that Java's JIT[2] compiler has already compiled all the code, before running the benchmarks, as described in Peter Sestofts article[1]. In the listing the method `pQueueRun()` is running the benchmark, this will be described in section 4.3. The `tracker.addTime(long time)` simply adds a time to a list that will later be written to a CSV file.

## 4.2 Update Method

There is no method to retrieve a specific node from the queue. That is why an Iterator is used to run through every node in the queue until a match is found.

```
1     Iterator<Node> it = pQueue.iterator();
2
3     timer.start();
4     while (it.hasNext()) {
5         n = it.next();
6         if (count == QUEUE_LENGTH - 1) {
7             n.verticeTo = 80085;
8             time = timer.step();
9             break;
```

[1] Comma Seperated Values Filestructure
[2] Just In Time

```
        }
        count++;
    }
```

Listing 3: Finding the node

The worst case scenario in a linear search is that our node has the highest weight and thus is positioned in the last index of the queue. This is also the scenario simulated in our benchmark, by updating the node when the check in the code "`if (count == QUEUE_LENGTH - 1)`" is true.

Now a thing to note in the code is the way data is inserted in our PriorityQueue. Since this part was not the focus of the article, the population of data into the queue happens in a linear way, while also moving each element with a lower priority one by one. Java priorityqueue uses a heap which is a lot faster. This results in our code being a lot slower in performance for inserting data.

### 4.3 Benchmark of Java's priorityQueue

The initialization and filling of the queue is descibed in listing 4. This is an extraction of a code partition from the `pQueueRun()` method, which is used to meassure the benchmark times.

```
    PriorityQueue<Node> pQueue = new PriorityQueue<>(
    QUEUE_LENGTH, nodeComp);

    // Filling the queue with Nodes
    for (int i = 0; i < QUEUE_LENGTH; i++) {
        pQueue.add(new Node(i, i + 1, i));
    }
```

Listing 4: Populating the queue

In listing 4 a simple priorityQueue that takes nodes as an object. The queue is filled with nodes, the nodes do not require any special values. However a unique value is added, so that the nodes a distinguishable. The benchmark is running 5000 times resulting in 5000 datapoints for exploration. On table 1 an overview of the datapoints are calculated using a Python Notebook.

| | |
|---|---|
| **count** | 5000 |
| **mean** | 1046789.700000 ns |
| **std** | 116796.066013 ns |
| **min** | 844800.000000 ns |
| **25%** | 973000.000000 ns |
| **50%** | 1030100.000000 ns |
| **75%** | 1097700.000000 ns |
| **max** | 2193600.000000 ns |

Table 1: Key numbers from benchmark

Looking at the mean and standard deviation on table 1 it is visible that the average time for an update is approximately 1.05ms +/- 0.1 ms. The results are visualized using a boxplot in section 4.5.

## 4.4 Benchmark of updatable PriorityQueue

The benchmark of the implmentation of our PriorityQueue can be seen on listing 5.

```
1     private static long pQueueRun() {
2         // Initialization removed for simplicity
3
4         timer.start();
5         Node n = queue.retrieve(new Node(500, 1000,
    QUEUE_LENGTH-1));
6         n.verticeTo = 101;
7         time = timer.step();
8
9         return time;
10    }
```

Listing 5: Benchmark implmentation on our PriorityQueue

As seen listing 5 the initialization is removed for simplicity in the example. So keep in mind that a new queue and timer are initialized every iteration before the benchmark. In the example its visible that we use our `retrieve()` method to get a specific node from the queue. We then mutate a value on the node, and stop the timer. Afterwards the time is returned. Later in the benchmark, the returned time is being saved in the `TimerTracker` witch writes it to a CSV file.

Data from the CSV file have been explored using a Python Notebook. The results can be found on table 2. Be aware that the times are measured in nanoseconds.

| | |
|---|---|
| **Count** | 5000 |
| **Mean** | 409.860000 ns |
| **Std. Dev.** | 754.987935 ns |
| **Min** | 100.000000 ns |
| **25%** | 300.000000 ns |
| **50%** | 300.000000 ns |
| **75%** | 400.000000 ns |
| **Max** | 31800.000000 ns |

Table 2: Key numbers from updated benchmark

| | **Results** | **n = 500000** |
|---|---|---|
| **Linear O(n)** | 1030100.000000 ns | 500000 elements |
| **Binary O(log n)** | 300.000000 ns | 5.69897000434 |
| **% difference** | 99.970877 % | 99.998860 % |

Table 3: Result vs. complexity

The benchmark completed with an average of 409.86 ns +/- 754.987935 ns, this can be seen on table 2. It is clear that much faster and more stable results from the updated algoritm are achieved. In the section 4.5 a visualization of the data as boxplots and a comparisson of the results will be discussed.

## 4.5 Comparisson of PriorityQueues

The medians are used to calculate the percent increase in performance and the result is a 99.97% increase.

These results can be explained by the complexity of both methods. It is known that a linear operation is defined as O(n). A binary search has a complexity of O(log n).

Now if we compare the results and complexities where n = 500000, which is the same size as our queues in the benchmark.

It is visible that our result have increased in performance, with aproximately the same amount as expected when looking at the complexity in accordance to the big O notation on table 3.

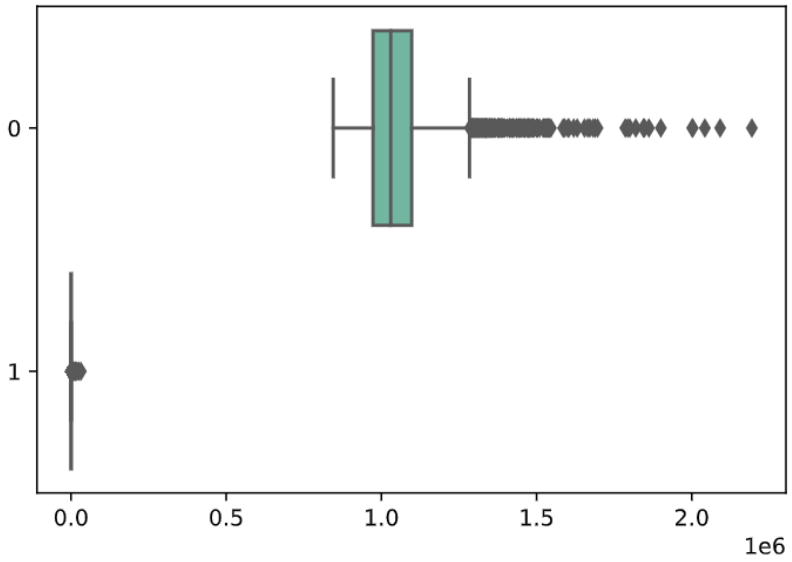


Figure 1: Comparisson of the the Priorityqueues

On figure 1 it is clear that the results is not only faster but also a lot more stable. This is visible from the amount of outliers in the two boxplots. On the machine which was used for benchmarking, Java's PriorityQueue results end up with a median of approximately 1 ms with outliers streching a far as to 2 ms. In comparrison to our implementation with a median of approximately 300 ns +- 754.987935 ns. The updated PriorityQueue reaches much faster and stable results.

# 5 Conclusion

Updating a value already placed in the queue in Java's PriorityQueue is relatively fast, with a median of 1.0301 ms searching for a node in a queue with a length of 500000. Our implementation however is comming in much faster at 300 ns, that is a 99.97% increase in performance. It is not certain that these numbers are precise, since we haven't tested it on various machines, only one. But the performance increase does match the expected result when comparing the two methods complexity using Big O notation. The future of this reasearch could easily be expanded in multiple ways. It could be testet on multiple machines instead of just one. Furthermore, it could be interesting to optimize our PriorityQueue to not insert values in a linear fashion. Hereby making it a real competitor to the Java standard library implementation.

# References

[1] Peter Sestoft. Microbenchmarks in java and c#. *IT University of Copenhagen, Denmark*, pages 2–16, 2015-09-16.