Java PriorityQueue Class

Anders Jacobsen, Dima Karaush

May 6, 2021

Abstract

Java's PriorityQueue class have a bottleneck when you need to update values in the queue. Depending on the use and implementation, Java's PriorityQueue can add serious performance issues when accessing the data using the poll() method. When updating a value in the queue, performance can be improved by more than 50% compared to Java's PriorityQueue implementation. Implementing your own version might remove this bottleneck from your software.

1 Introduction

Why did we choose this subject 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. The shortcomming was that to update a value in the queue, we had to use a linear search (loop) through the queue to update a value within. That is fine on a small scale, but what if need to draw a graph of all the cities in the world? We believe that this has space for optimization and that is why this article explores this subject.

We are going to use a previously developed `Timer` class.

How will we work with the subject

2 Scope

What will be in this article What will not be included in this article

3 Problem

3.1 Background

3.2 Problem Statement

The questions we will try to solve Problemfomulering

4 Analysis

4.1 Benchmark Method

This subsection will cover how we prepared and executed our benchmarks. It will also touch the subject of what we are timing in the two implementations of a PriorityQueue. We've followed Peter Sestofts approach to microbenchmarking in Java [1], we've used a combination of Mark5 and Mark3 benchmarks with a few twists here and there. To make benchmarks, a Timer class is nessesary. We've designed the simplest version possible to avoid interference from calculations in the Timer class.

```java
public class Timer {
    private long start;

    public void start() {
        start = System.nanoTime();
    }

    public long step() {
        return System.nanoTime() - start;
    }
}
```

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. Also a method for writing the optained times to a CSV[1] file. The CSV files will be used to explore the data later.

```java
private static void benchmarkPriorityQueue(int warmupIterations, int iterations, TimeTracker tracker) {
    // Printing removed for simplicity
    // Warmup
    for (int i = 0; i < warmupIterations; i++) {
        tracker.addWarmupTime(pQueueRun());
    }
    // Benchmark
    for (int i = 0; i < iterations; i++) {
        tracker.addTime(pQueueRun());
    }
}
```

Listing 2: Benchmark iterations

[1]Comma Seperated Values Filestructure

Measurements in the benchmark is done only on the time it takes to update a value in the PriorityQueue. Again we follow Peter Sestofts microbenchmarking techniques. In listing 2 we run a number of warmup iterations before running the actual benchmark. This is to fight the battle agains Java's JIT[2] compiler as described in Peter Sestofts article. In the listing the method `pQueueRun()` is running the benchmark, this will be descriped in section 4.2. The `tracker.addTime(long time)` simply adds a time to a list that will later be written to a CSV file.

4.2 Benchmark of Java's priorityQueue

To understand the example that we are using we have to take a look at how we use our `pQueueRun()`.

```
    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 3: Populating the queue

We create a simple priorityQueue that takes nodes as an object. Then we add some nodes that fill up the queue, the nodes do not require any special values. But we do add a unique value to each so that we are able to distinguish the nodes from each other. When were looking at updating a node in Java's priority queue. There is no method to retrieve a specific node from the list. Therefore we use an Iterator to list through every node in our list until we find a match.

```
    Iterator<Node> it = pQueue.iterator();

    timer.start();
    while (it.hasNext()) {
        n = it.next();
        if (count == QUEUE_LENGTH - 1) {
            n.verticeTo = 80085;
            time = timer.step();
            break;
        }
        count++;
    }
```

Listing 4: Finding the node

[2] Just In Time

Notice that we are listing through the queue in a linear way, which is not the most efficient way of searching through a list. The worst case scenario is that our node has the highest weight and thus ends in the end of the list. This is also the scenario that we simulate, by updating the node when the `if (count == QUEUE_LENGTH - 1)` check is true.

As we are running the benchmark and the times are loaded into the CSV file 5000 times. We can get can get an overview of the times that we have saved in the .CSV file. Running the results through a python notebook, we can get a description of the data.

count	5000	data points
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

4.3 Update Method

4.4 Benchmark of updateable PriorityQueue

The way we benchmarked our implmentation of the PriorityQueue can be seen on listing 5.

```
    private static long pQueueRun() {
        // Initialization removed for simplicity

        timer.start();
        Node n = queue.retrieve(new Node(500, 1000,
QUEUE_LENGTH-1));
        n.verticeTo = 101;
        time = timer.step();

        return time;
    }
```

Listing 5: Benchmark implmentation on our PriorityQueue

As seen on the listing we have removed the initializing for simplicity in the exapmle. So keep in mind that a new queue and timer are initialized every iteration in the benchmark. In the example it 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.

We have explored the data from the CSV file using a Python Notebook. This gave us the following inside.

OS	Microsoft Windows 10 Pro
OS Version	10.0.19042 N/A Build 19042
System Type	x64-based PC
Processor(s)	Intel® Core™ i7-10700KF Processor, 16M Cache, up to 5.10 GHz
BIOS Version	American Megatrends Inc. 1.10, 21-05-2020
Total Physical Memory	32.688 MB
Disc(s)	Force Series™ MP510 980GB M.2 SSD (up to 3480MB/sec sequencial read)

4.5 Comparisson of PriorityQueues

5 Conclusion

Answer our questions and possibly introduction

References

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