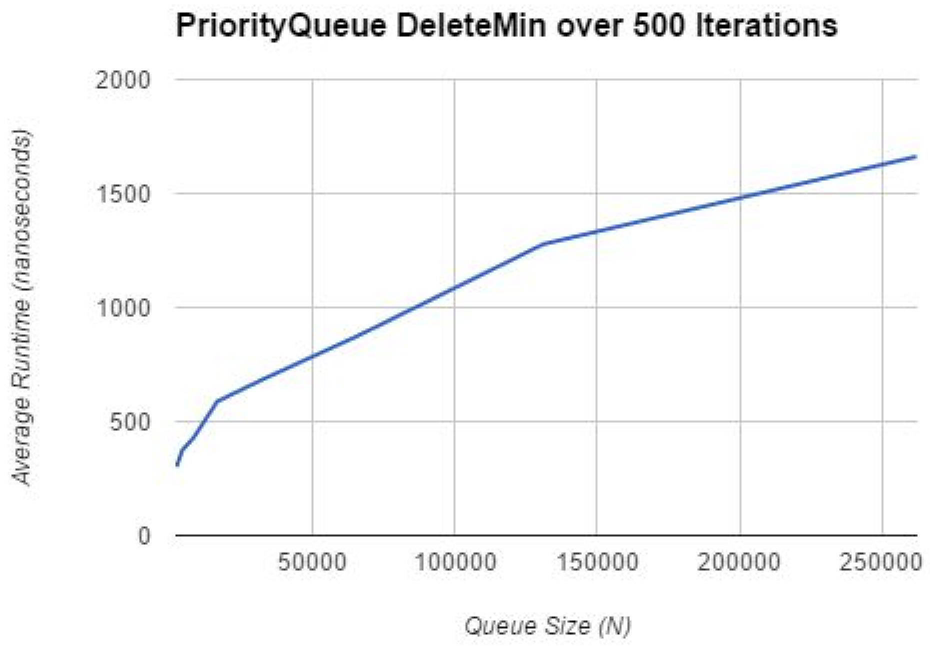
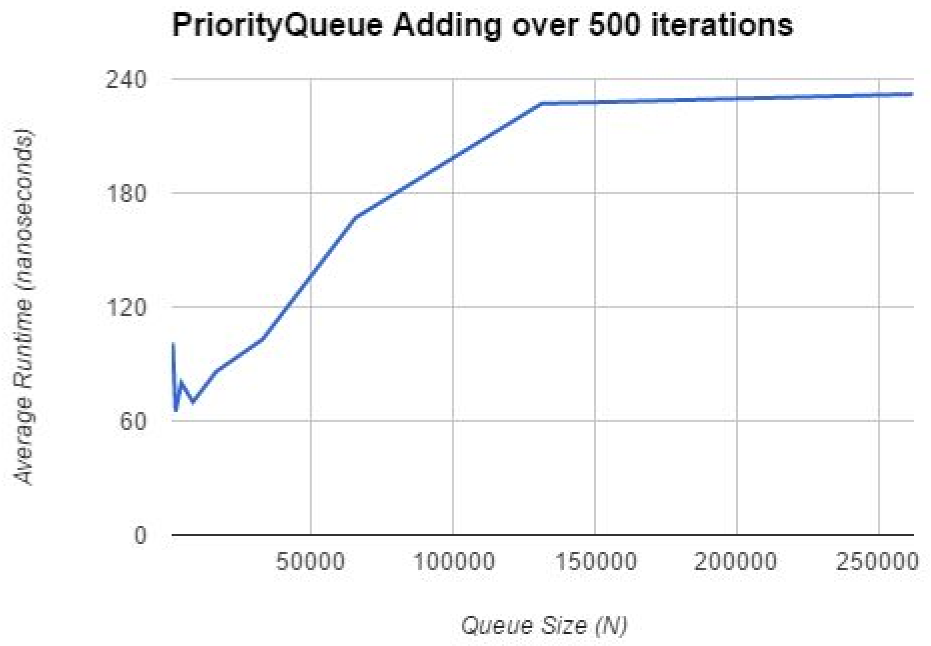
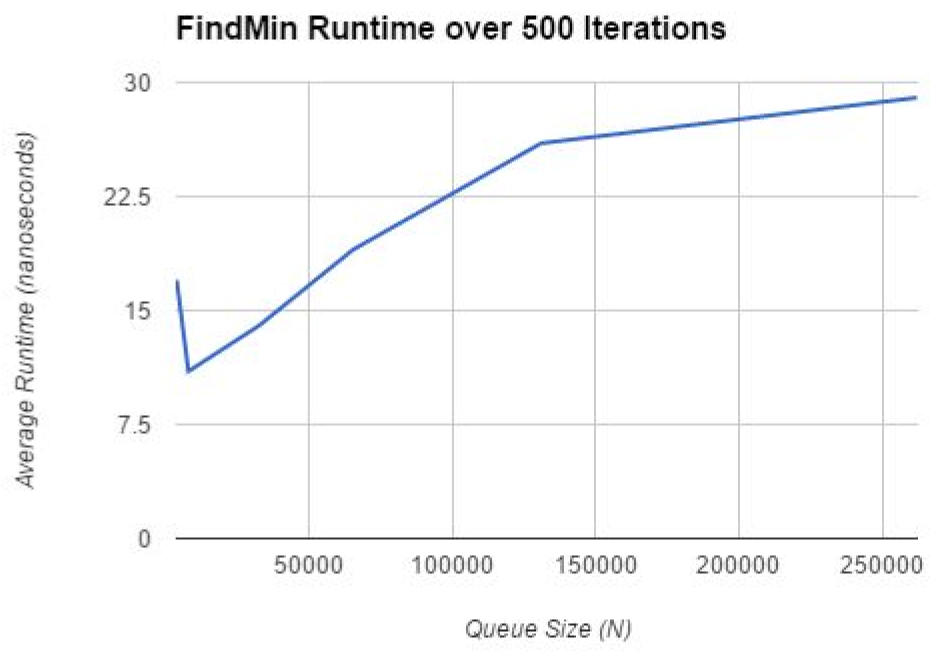
Vladimir Srdic
Assignment 11 Analysis Document
11/16/2016

Design and conduct an experiment to assess the running-time efficiency of your priority queue.







- These experiments were conducted as follows: Lists were made of size N ranging from 2^10 to 2^18. The lists were remade for every iteration. This is costly, but was necessary as doing multiple iterations over the same list yields strange results due to deleteMin removing all of the minimum items, leaving the queue to be full of large numbers only. This then affects the runtime of both add and deleteMin as they need to percolate less and less. During these iterations, the time required to add an item was recorded. After that the minimum item was deleted to retain N. Then the minimum item was found and the time required to do so was recorded. Finally, the minimum item was deleted and the time required to do so was recorded. At the end of each N (2^10, 2^11, etc.) the total time was averaged over the number of iterations (500 in this case) and the averages were plotted as seen above.

What is the cost of each priority queue operation (in Big-O notation)? Does your implementation perform as you expected? (Be sure to explain how you made these determinations.)

- Removing (polling): The big O for Priority Queues when removing the minimum item is O(logN). Whenever an item is removed you must replace it with the last item in the tree and move it down the heap (binary tree) until the structure of the heap is kept where the items above the added item are equal to or smaller than it and the items below are equal to or greater than. In doing so, each swap we do to move the item down the list cuts the amount of possible items by half (because the tree is required to be balanced and full). My Priority Queue accomplishes this as can be seen by the graph, it is definitely a log graph.

- Adding: The big O for Priority Queues when adding is also O(logN). This is due to a similar reason as removing, except going the other way. We start from the bottom and move the item up until the structure of the heap is maintained. A tree has O(logN) levels, so we get the same runtime. My Priority Queue accomplishes this as well, as can be seen by the above graph. It is also quick, as can be seen by the nanoseconds required (capping off at ~240).
- Finding Minimum: The big O for Priority Queues when finding the minimum is O(c). This is because the tree is actually represented by an array where the minimum (top) item is always at the 0 index, so in order to find this we simply return the 0 index value. My Priority Queue accomplishes this, the jump seen is simply due to variance in the JVM and the averaging of those variances.

Briefly describe at least one important application for a priority queue. (You may consider a priority queue implemented using any of the three versions of a binary heap that we have studied: min, max, and min-max)

- Aside from Dijkstra's algorithm and finding the cheapest path in a graph, a priority queue can be used in something like statistical analysis. You could find the top or bottom X items in a set of data. Further, a priority queue could be used in sorting the importance of support tickets that may be submitted to any given company. There would have to be a way to assign a comparable value, potentially a user option or a quick screening by someone who isn't exactly there to resolve them. This way the most important tickets/problems could be addressed first.

How many hours did you spend on this assignment?

- I spent about 5 hours on this assignment.