Assignment 11 – Analysis

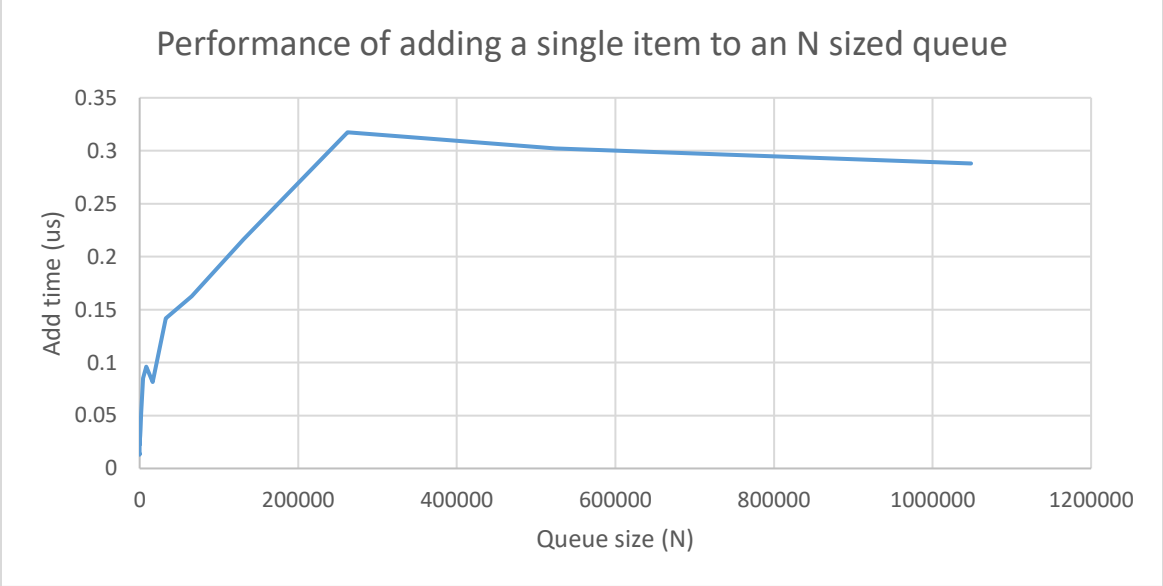
Ben Figlin (u1115949)

Nov 15, 2016

1. **Design and conduct an experiment to assess the running-time efficiency of your priority queue. Carefully describe your experiment, so that anyone reading this document could replicate your results. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plots(s), as well as, your interpretation of the plots is critical.**

   Experiment for assessing the add method performance:
   1. Create an array of size N with random integers between 0 and N.
   2. Add all of the array elements into a new PriorityQueue.
   3. Call the add method one more time with a random integer, measure the time to execute the method.
   4. Repeat #2 for several iterations and calculate the average timing result of the iterations, write the result time and the number N.
   5. Repeat #1, each time incrementing the number of elements: N=N*2, until reaching a sufficient number of samples
   6. Plot the results on a chart.

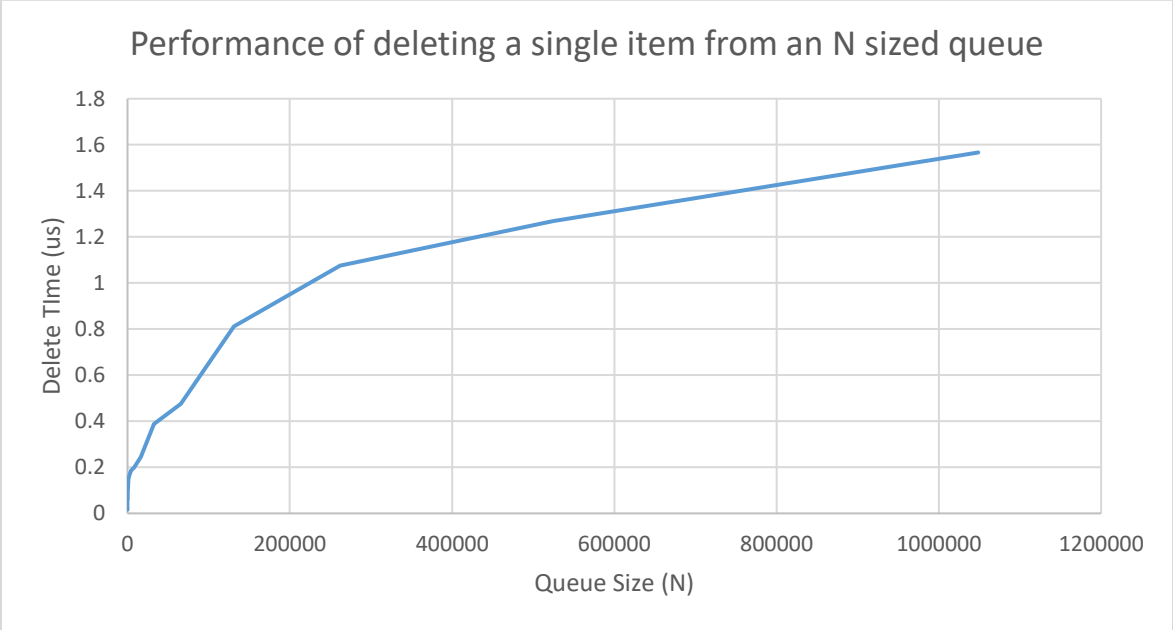
   The following chart represents the results of the experiment. We can see that the addition time barely changes as the queue size increases, so the behavior is very close to constant.



   Experiment for assessing the deleteMin method performance:
   1. Create an array of size N with random integers between 0 and N.
   2. Add all of the array elements into a new PriorityQueue.

3. Call the deleteMin method once, measure the time to execute the method.
4. Repeat #1 for several iterations (to generate a new random array with each iteration) and calculate the average timing result of the iterations, write the result time and the number N.
5. Repeat #1, each time incrementing the number of elements: N=N*2, until reaching a sufficient number of samples.
6. Plot the results on a chart.

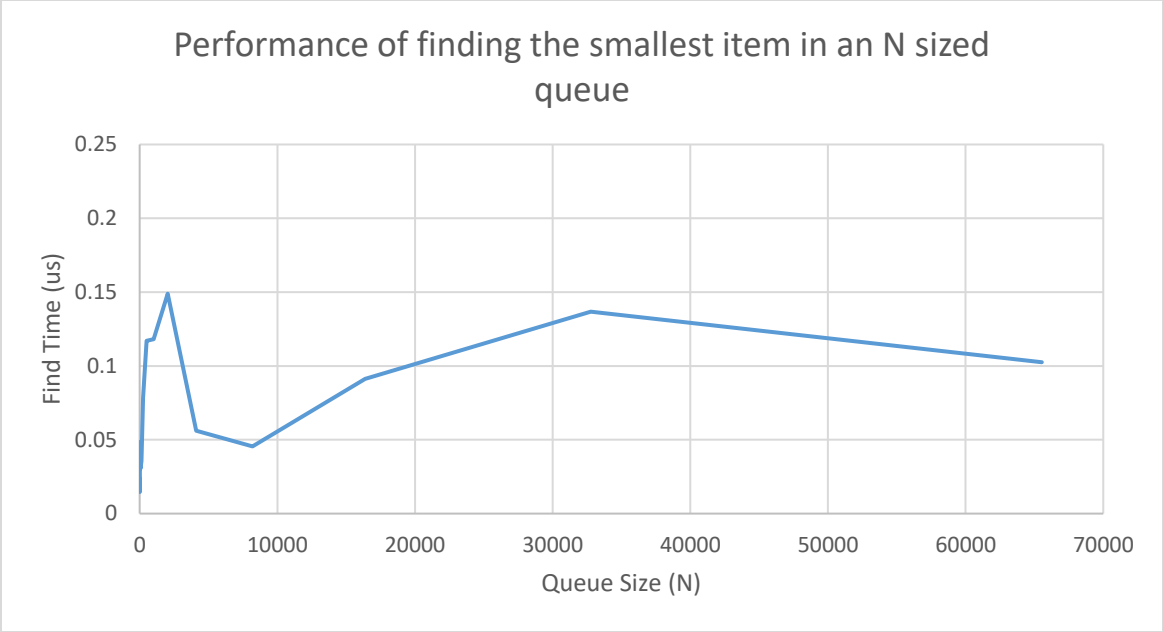
The following chart represents the results of the experiment. We can see that the deletion time slowly increases as the queue size increases, the behavior looks similar to a log(n) behavior.



Experiment for assessing the findMin method performance:

1. Create an array of size N with random integers between 0 and N.
2. Add all of the array elements into a new PriorityQueue.
3. Call the findMin method once, measure the time to execute the method.
4. Repeat #1 for several iterations (to generate a new random array with each iteration) and calculate the average timing result of the iterations, write the result time and the number N.
5. Repeat #1, each time incrementing the number of elements: N=N*2, until reaching a sufficient number of samples.
6. Plot the results on a chart.

The following chart represents the results of the experiment. We can see that the findMin time remains fairly constant as the queue size increases, the behavior can be considered as constant.



2. **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.)**
   - ***add()***
     We can see from the results above that adding an element to a small or a very large queue does very little difference in terms of timing, there is a behavior on the chart that slightly reminds a logarithmic trend, but it seems to stabilize eventually, and might be just caused by the load on the OS or the JVM by previously adding many items (which causes an overall system load, and delaying of other OS threads that may later interfere when we start taking the measurements).
     As the change in time is very slight compared to the queue size, we can define the cost of the add() operation to be in the range of **O(c)**, which is as expected, compared to the theory, where on average there will be only 2-3 location swaps in the tree for each added item, no matter the size of the tree.
   - ***deleteMin()***
     When looking at the results, the behavior of this method looks very much like a logarithmic behavior and the time is constantly increasing, with a slowing down rate. This makes the complexity to be in the order of **O(logN)**, which is expected because when deleting the minimum item, we replace it with one of the largest items (because it is at the end of the tree), so it needs to bubble down a lot, and will settle down on average in the last few levels of the tree, so the larger the tree is, the more time it takes to bubble the item down (with a O(LogN) complexity because it is a binary tree, and each time we go down a level, we eliminate half of the elements).
   - ***findMin()***
     Finding the minimum item is as simple as just returning the value of the root node, which is guaranteed to be the smallest item in the priority queue, per design.
     We can see from the results that the time to find the smallest item fluctuates around a constant time, no matter of the size of the queue. So the complexity in this case is O(c),

which is also expected, because reading the root node data is no dependent on the size of the tree.

3. **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)**
   An obvious use of a priority queue ("obvious", because we used it in class) is in Dijkstra's Algorithm. In that algorithm, we want to enqueue different paths, where the "cheapest" one having the minimum priority value, and is dequeued first when fetching items from the queue.

4. **How many hours did you spend on this assignment?**
   I spent around 5-7 hours on this assignment.