Dan Ruley

u0956834

CS 2420

1/31/2019

Abstract:

This is a documentation of the timing analysis my partner Aric Woodliff and I performed on our SimplePriorityQueue class. Overall, we worked fairly well as a team and I would work with Aric again in the future. We switched roles fairly frequently, perhaps every thirty minutes. I preferred to be in the driver role and I think he did as well. There were times we both wanted to be programming, but we did a reasonable job switching up the driver and navigator roles. I think more time spent with paper and pen before coding will help us in the future.

Experiment One (Find Min and Find Min (total):

Timing breakdown: Twenty iterations starting at a SimplePriorityQueue of N = 100,000 and ending at 2,000,000. Increasing by increments of 100,000. Each iteration, the queues were set up outside of the timing code. Inside the timing code, the findMin() function was called a total of 10,000 times. Before recording the final time, the overhead of the loop was subtracted.

This experiment looked at the runtime of the findMin() method on increasingly large N values. As expected, this found an O(Constant) runtime. This is because findMin() simply accesses the last item in the array and returns it. No matter how large the size of the array, the computer does very little work in fetching the minimum value, which is always at the end of the queue per the implementation. The following chart shows the runtime for findMin(), which is about a 150 nanoseconds for the first iteration, and then next to zero nanoseconds for each subsequent iteration.

Problem Size	findMin (Time in ns)
100000	146.4492
200000	0.079
300000	0.079
400000	0.079
500000	0.079
600000	0.0791
700000	0.0791
800000	0.079
900000	0.0791
1000000	0.0791
1100000	0.079
1200000	0.079
1300000	0.1186
1400000	0.079
1500000	0.079
1600000	0.079
1700000	0.079
1800000	0.079
1900000	0.1185
2000000	0.079

Figure 1: findMin(), including all runs:

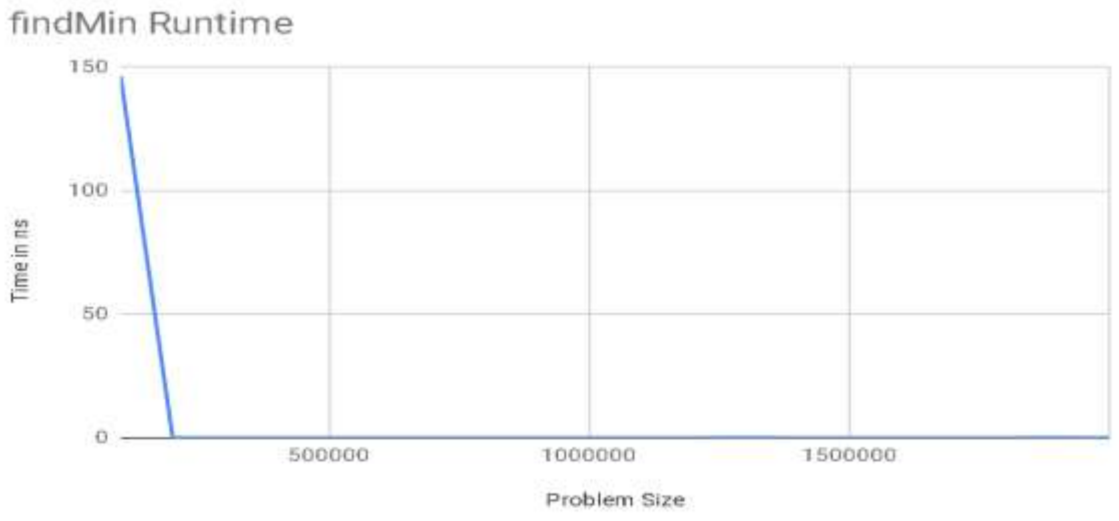
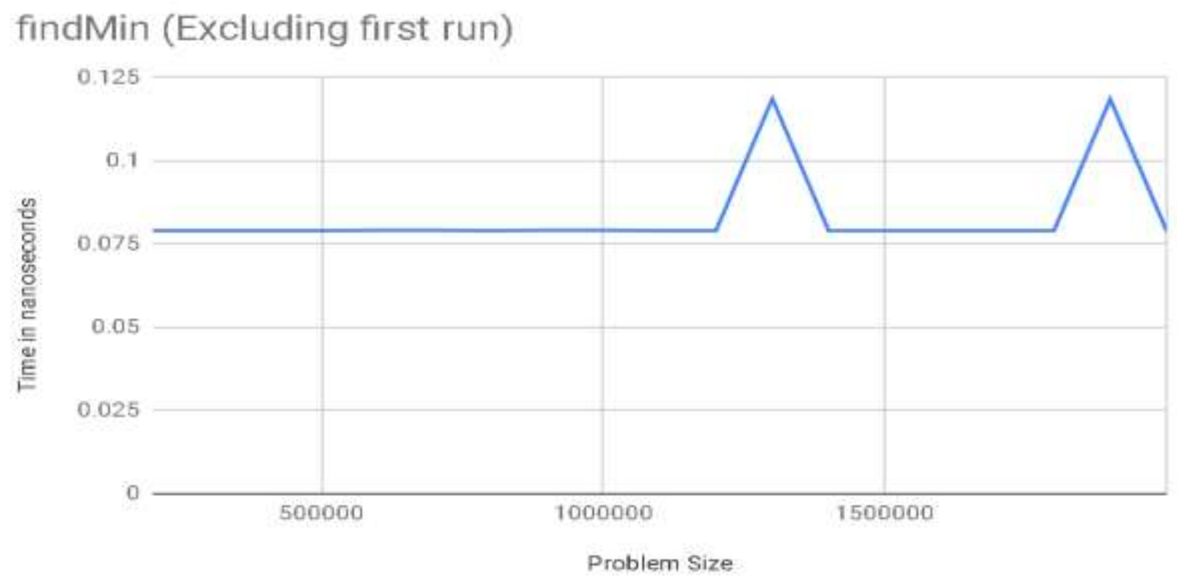


Figure 2: findMin(), excluding the first run:



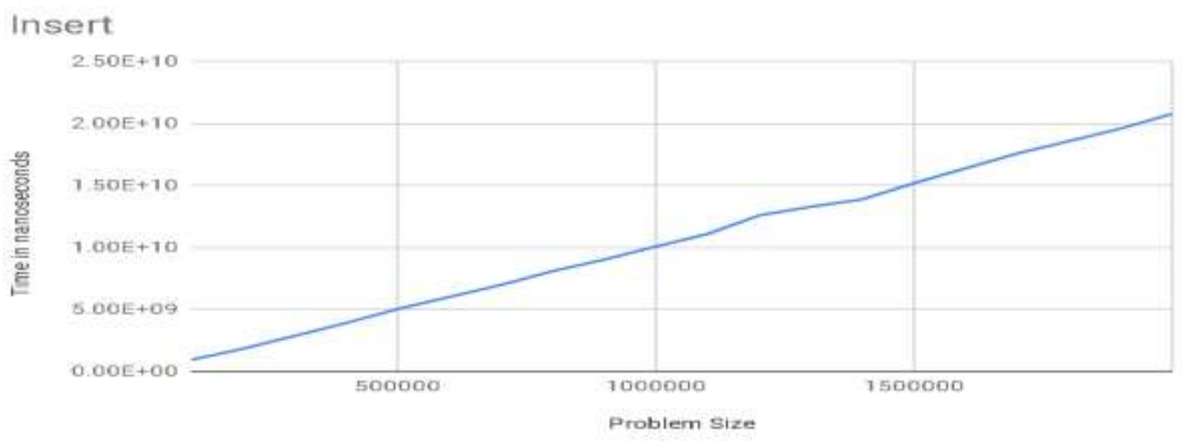
Experiment Two (Insert):

Timing breakdown: Twenty iterations starting at a SimplePriorityQueue of N = 100,000 and ending at 2,000,000. Increasing by increments of 100,000. Each iteration, the queues were set up outside of the timing code. Inside the timing code, the insert function was called a total of 10,000 times, using an average case which was a pseudorandom integer from 0 to N, exclusive. After each insert deleteMin() was called to restore the queue to its N size value. Before recording the final time the overhead of the loop, the deleteMin() method, and the random.nextInt() method were all subtracted.

This experiment looked at the insert method on a queue of increasingly large N. The results are perfectly consistent with the expected O(N) behavior. Consult the following chart to see a visual depiction of the timing of our insert() method.

Problem Size	Insert (Time in ns)
100000	9.54E+08
200000	1.85E+09
300000	2.88E+09
400000	3.92E+09
500000	5.05E+09
600000	6.02E+09
700000	7.00E+09
800000	8.11E+09
900000	9.04E+09
1000000	1.01E+10
1100000	1.11E+10
1200000	1.26E+10
1300000	1.33E+10
1400000	1.39E+10
1500000	1.52E+10
1600000	1.64E+10
1700000	1.76E+10
1800000	1.86E+10
1900000	1.96E+10
2000000	2.08E+10

Figure 3: insert() method:



Conclusion:

Overall, results of our timing experiments were consistent with our expectations. findMin() exhibited it's expected O(Constant) behavior, and insert() exhibited O(N) behavior. The process of performing this analysis caused me to think much deeper about what was actually happening with these algorithms, and greatly increased my understanding of the intricacies in designing efficient algorithms.