Assignment 10: Binary Max Heap

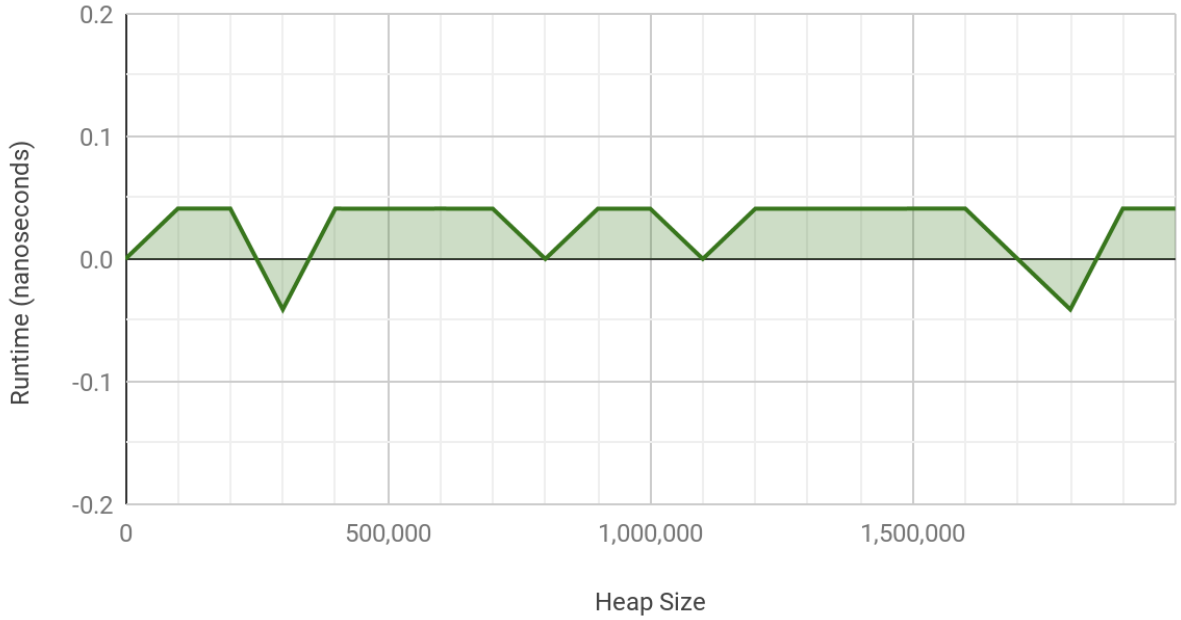
Analysis Document by Qianlang Chen (u1172983)

As in the previous 4 programming assignments, I gladly had Kevin Song as my programming partner in this one again. Kevin did a great job at understanding the problem and coming up with proper solutions, as well as following my commands and giving me ideas. Because of that, I plan to work with him in the final programming project.

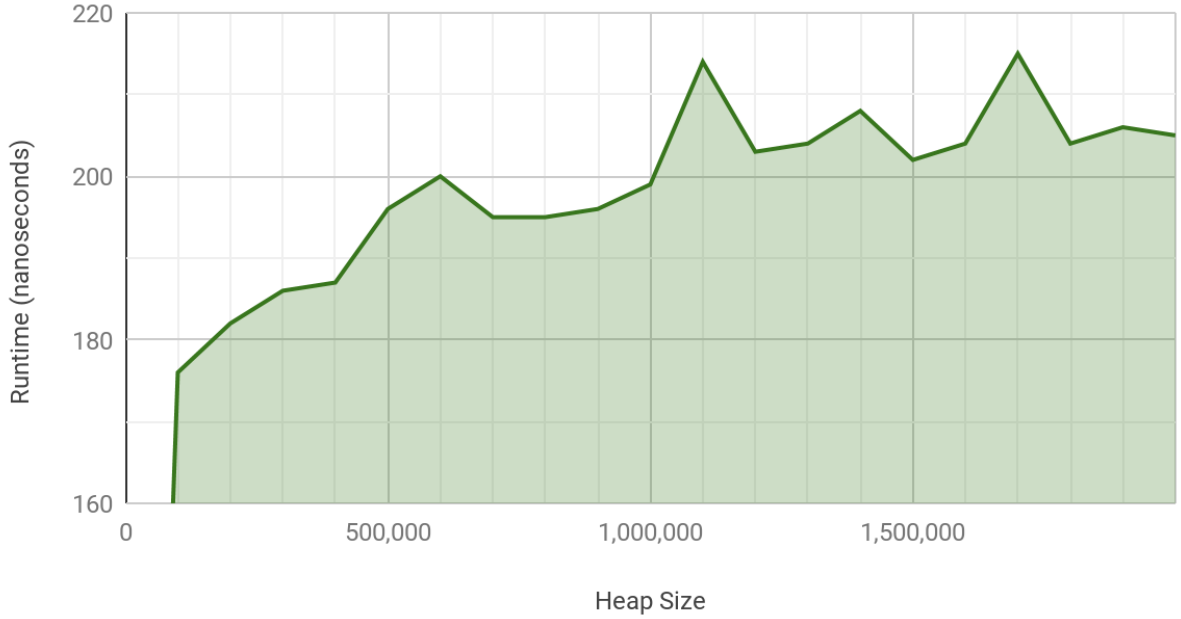
One small question we asked ourselves when designing our *BinaryMaxHeap* was where in the backing array we should use to store the root of the heap. We discussed and chose to store the root at index *1* because this would make our calculations of the indices of children and parent of a node easier. Choosing the index *1* for the root also meant that the index *0* would be wasted as a placeholder, however, in the long run, as the heap size increases, it would not be too wasteful for a single index to be unused.

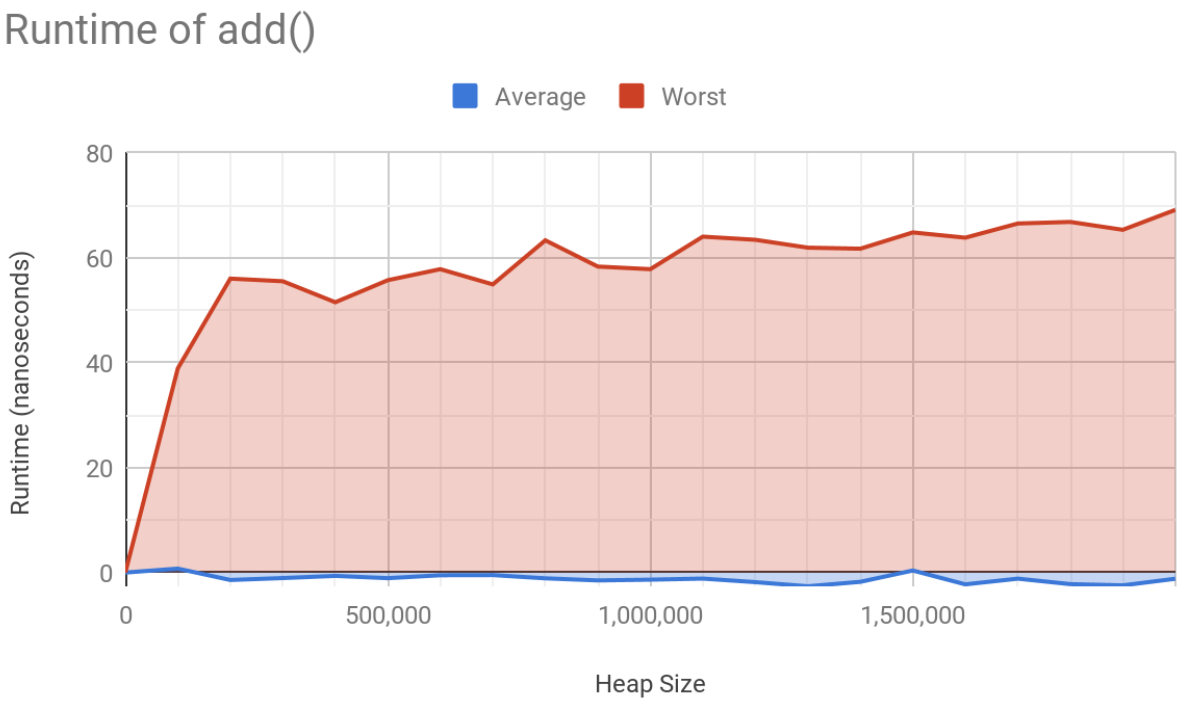
To see and demonstrate that our *BinaryMaxHeap* is efficient at its main 3 methods, namely the *add()*, *peek()*, and *extractMax()* methods, we collected the runtime of those 3 methods with a range of different data sizes. Moreover, to better understand that the *add()* method works better in the average case than our code suggests, we timed the *add()* method twice, with random values being added, compared to always adding the maximum value which we knew resulted in the worst case. Here are our results:

Runtime of peek() on Average



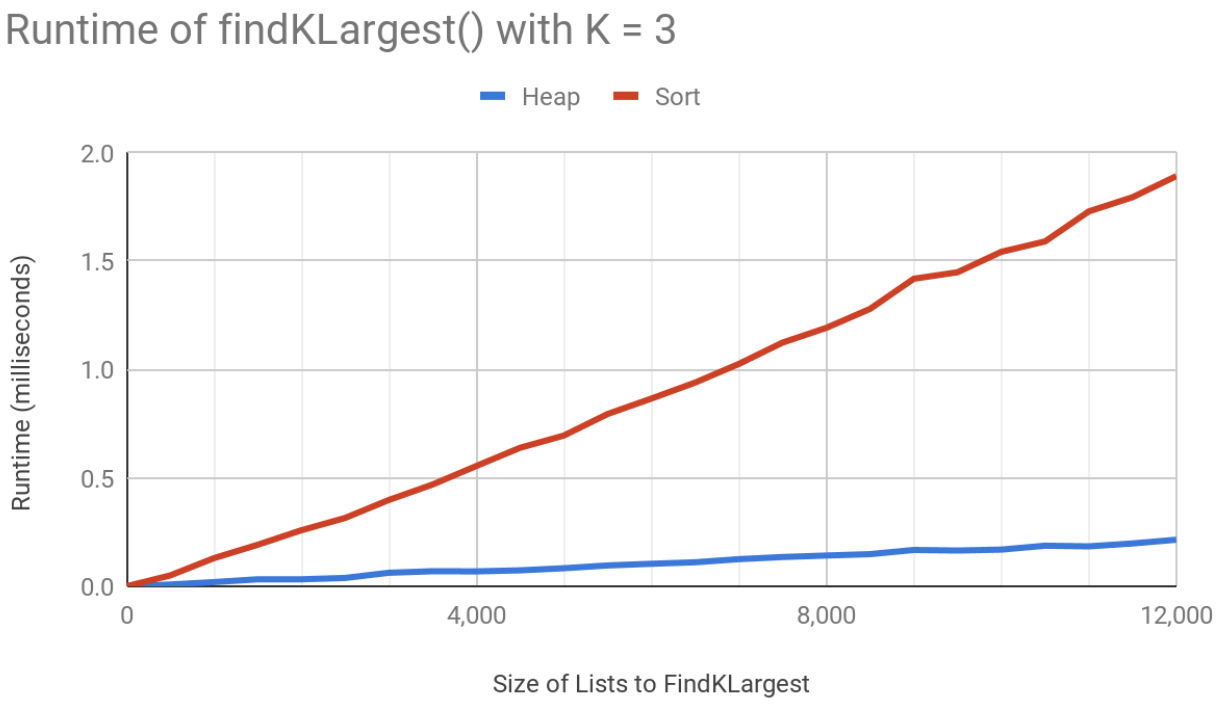
Runtime of extractMax() on Average

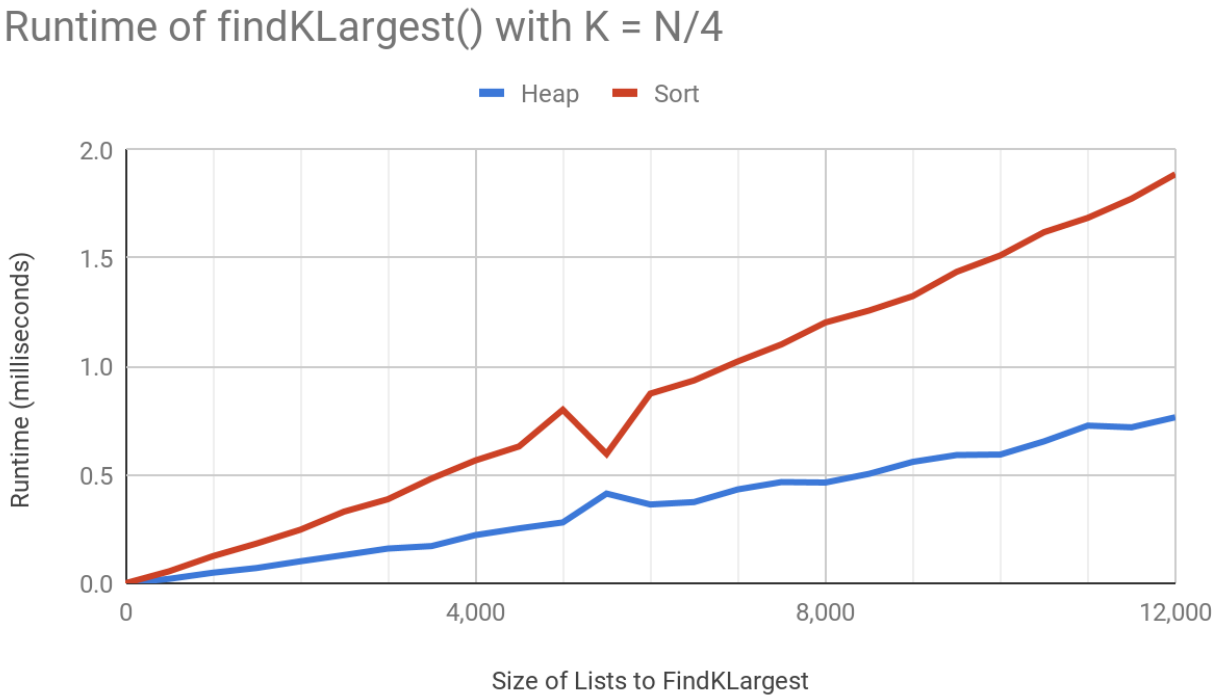


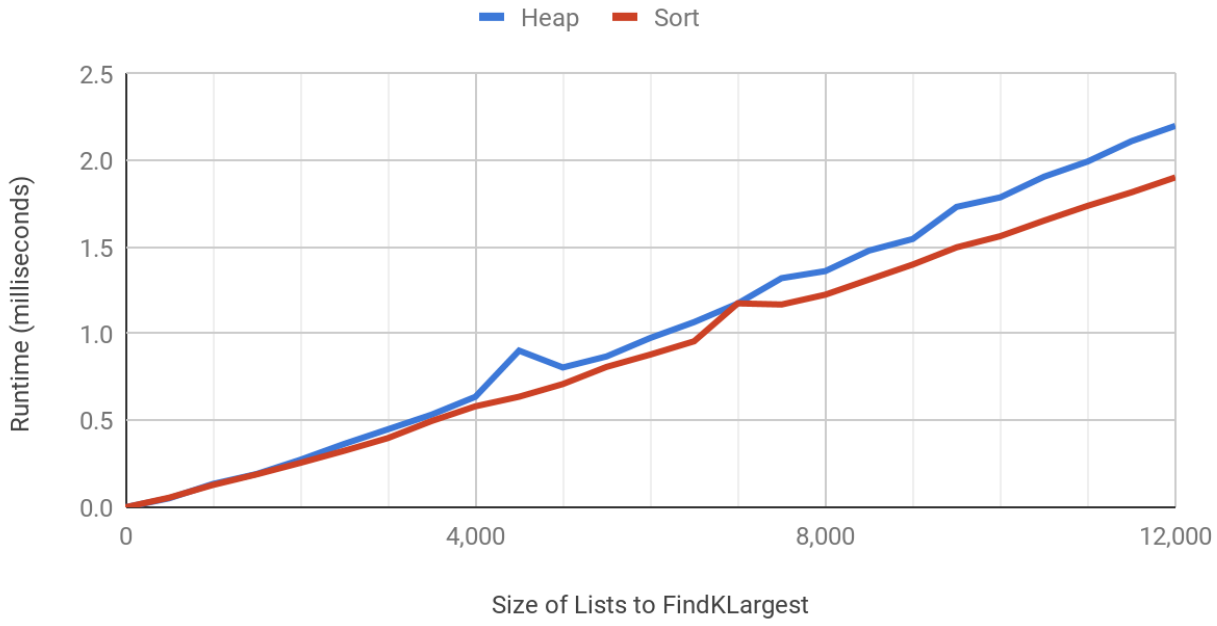


As the graphs show, our *BinaryMaxHeap* follows the theory we have learned and does an excellent job at its 3 basic methods: **constant** runtime for the *peek()* method and the average case of *add()* method and **logarithmic** runtime for the *extractMax()* method and the worst case of *add()* method. These results make sense: by the nature of a max-heap, the maximum value is stored at its root, which can be very easily accessed, giving our *peek()* method a constant runtime. Our *extractMax()* method percolates the new root value downwards, with a high probability that the value would need to get all the way to the last two rows, which performs logarithmically. On the other hand, when our *add()* method percolates the value upwards, chances are that it would not have to go very far from the bottom row, which on average costs constant runtime. In the worst case, however, it is inevitable for the item to get to the root, costing logarithmic runtime. We then used the "check analysis" technique to verify our results, which suggested that our results were correct.

The second part of our programming assignment involved finding a number (k) of maximum values from a given list. We had two approaches to this problem: building a heap from the given list and *extractMax()* k times, or sorting the entire list from maximum to minimum and take the first k elements. Both of these two ways are equally valid, but they provide different runtime behaviors with different k values (the number of maximum values to find). With N being the size of the list to find k maximum values, building a heap costs a runtime of N, and *extractMax()* k times costs k*lg(N). Using the sorting routine instead, Merge Sort seems to be the best choice for sorting a list, costing Nlg(N). When k is small, the heap routine does a better job since linear would dominate in its runtime complexity. When k is close to N, the term $k$lg(N) approaches Nlg(N), which then dominates in the runtime complexity, and sorting routine would do a better job in this case since Merge Sort runs faster in practice than Heap Sort. To see the difference more clearly, we timed and compared the two routines to solving this problem with a range of different list sizes, along with 3 different k values: very small (*3*), moderate (N/4, with N being the list size), and very big (exactly N):




Runtime of findKLargest() with K = N/4
Heap
Sort
2.0
1.5
1.0
0.5
0.0
Runtime (milliseconds)
0
4,000
8,000
12,000
Size of Lists to FindKLargest


Runtime of findKLargest() with K = N
Heap
Sort
2.5
2.0
1.5
1.0
0.5
0.0
Runtime (milliseconds)
0
4,000
8,000
12,000
Size of Lists to FindKLargest

The results match my reasoning above: when k is small (not close to N), the heap routine provides a runtime close to linear and is faster than the sorting routine; when k is close to N, both routines provide an $N\lg(N)$ runtime, and the sorting routine does a better job. We then "checked analysis" and verified that the runtime behaviors of the two routines matched our predictions.