Introduction

The purpose of this assignment was to see the process of building various forms of priority queues, as well as their time and space complexity when performing general operations on them. A heap, unsorted, and sorted priority queue were used in the construction of these data structures.

Theoretical Analysis

For the insert operation:

- Resizing the array takes O(n) time to perform its action, which might domineer the insert function for some types of queues (i.e. unsorted array)
- The unsorted priority queue acted in the fastest time, O(1) when the array doesn't need to be resized, and O(n) when resizing of the array needs to happen. *Average : O (n)*
- The unsorted priority queue took the longest amount of time, with its fastest case being O(n) (occurs when it needs to be put at beginning or end of list) and worst case O(3/2n) if the item needs to be inserted in the middle of the list. This term dominates the resizing of the array. *Average : O (n)*
- For the heap priority queue, it is at best case O(1) if the number is larger than its parent, and worst case O(log2 (n)) if the item needs to be bubbled up to the top. *Average : O(nlog2 (n))*

For the sorting operation:

- For the unsorted array, popping the minimum value always has a complexity of O(n), as a comparison with every variable needs to occur. For the entire operation of popping in sorted order, it will take a complexity of O(n^2) *Average : O (n^2)*
- For the sorted array, popping the minimum value always has a complexity of O(1), as it only needs to pop the first item of the list. For the entire operation of popping ins sorted order, it will take a complexity of O(n) *Average : O (n)*
- ** Note that for both of these priority queue types, if is likely for the array to have to be resized after popping an item, to maintain the order integrity. If considers, it adds additional time complexity by a factor of O(n), making unsorted O(n^3) and sorted O(n^2) to popp every item in sorted order.
- For a priority queue heap, it always pops one item at O(1) complexity. The heapification to maintain the requirements of a heap is at best O(1) due to it only having to make one comparison with the node of depth 2 and being in the right spot. At worst, it is of complexity O(log2 (n)) if the item goes back to the bottom of the priority queue. On average it the complexity will be probaby better represented by O(log2 (n)). If we were to pop every item on the priority queue heap, it would take an average time of O (nlog2 (n)). *Average : O (nlog2 n)*

An unsorted array seems to be best if one is not frequently removing items from the array. If items just need to be inserted and it is not likely that every item will be popped, and unsorted array would be better than a sorted array. A sorted array would be better if one needs to constantly pop items from the priority queue, and every item is likely to be popped.

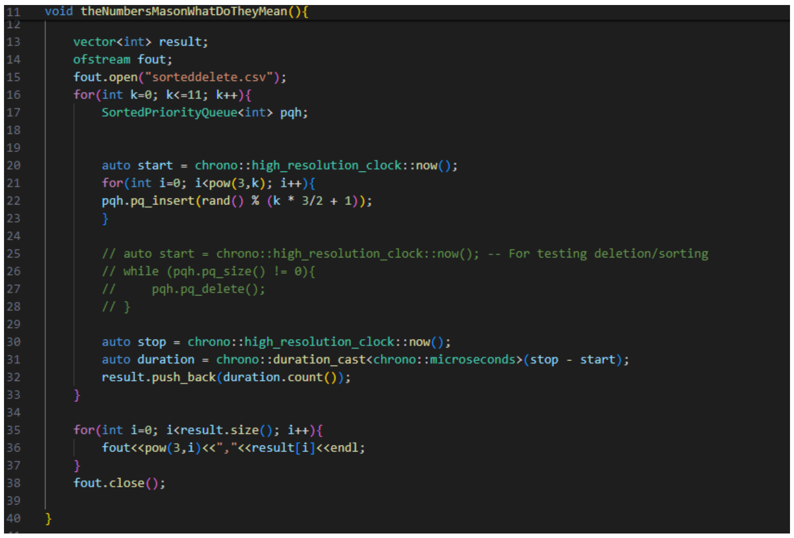
A heap priority queue however, does both the advantages of the unsorted and sorted array priority queue with less downsides (unsorted takes a long time to sort, sorted takes a long time to insert). One can efficiently insert and remove items from the priority queue without either operation intensely bogging down the operation. It would be best to use this organization type for long lists and when operations are being used frequently in general.

Experimental Setup

For the experiment, I borrowed the code from the LE3 assignment which measured the time complexity of vectors, and modified it to track the time complexity of the insert() function and pq_delete() function with size. Note that with the pq_delete() function, deleting all items from the Priority Queue results in a sorted list, so there is validity in testing this function for the priority queue's sorting properties.

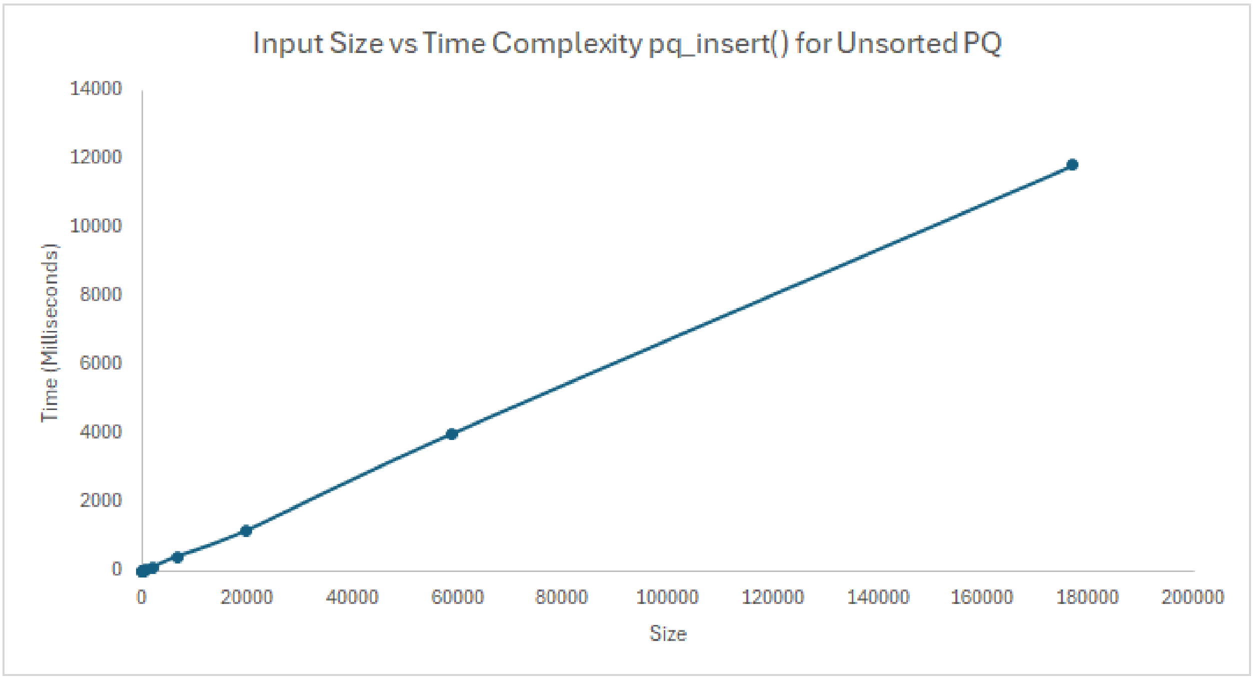
Also note that because of the resizing of the array for the unsorted and sorted priority queues, there is an additional factor of O(n) that accounts for the pq_delete() function. To properly assess, you would have to divide the big O notation of the apparent graph by n to get just the pure functionality of the sorting algorithm.

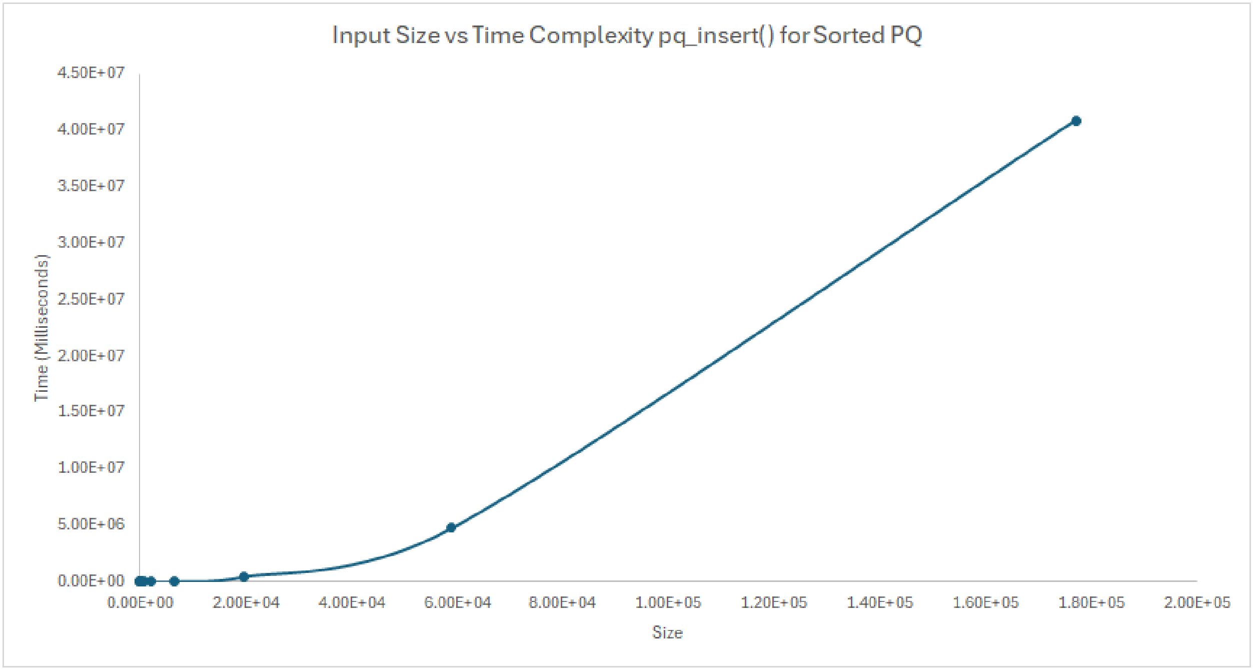
The test inputs were done by using a rand function to generate a number between 1 and 3^(k * 3/2), where n is the size of the priority queue. This is to represent an average, varied case. The max size goes to ~177,000 inputs.

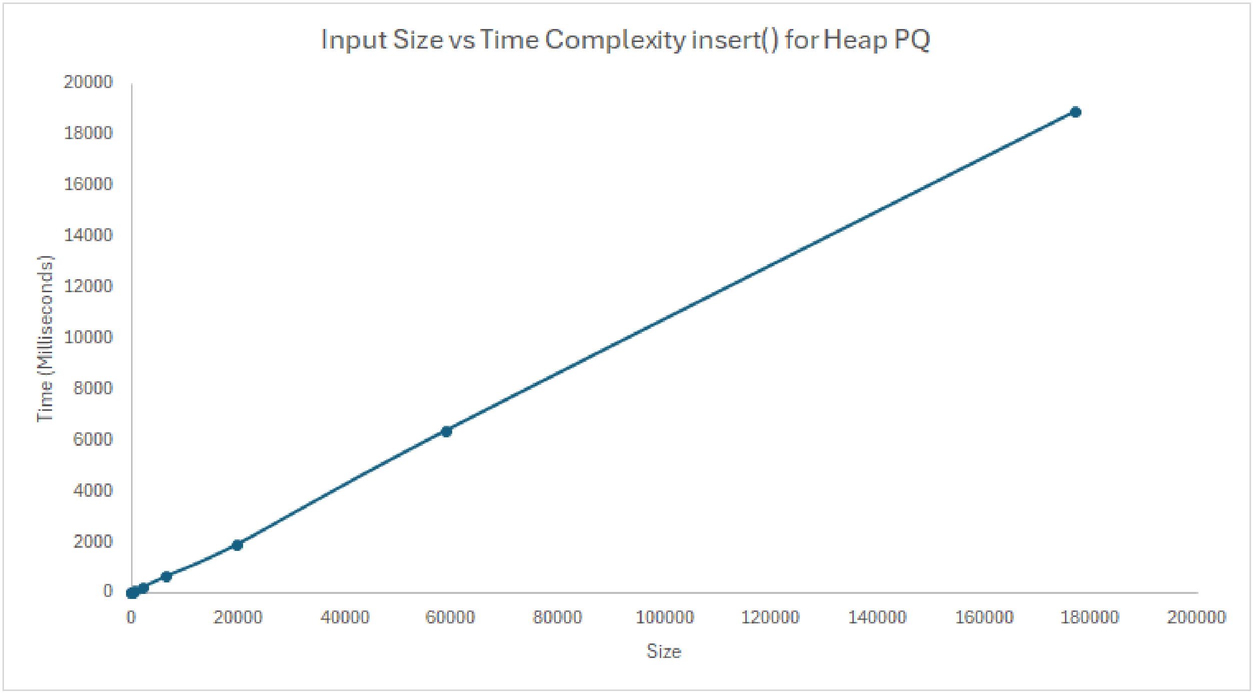


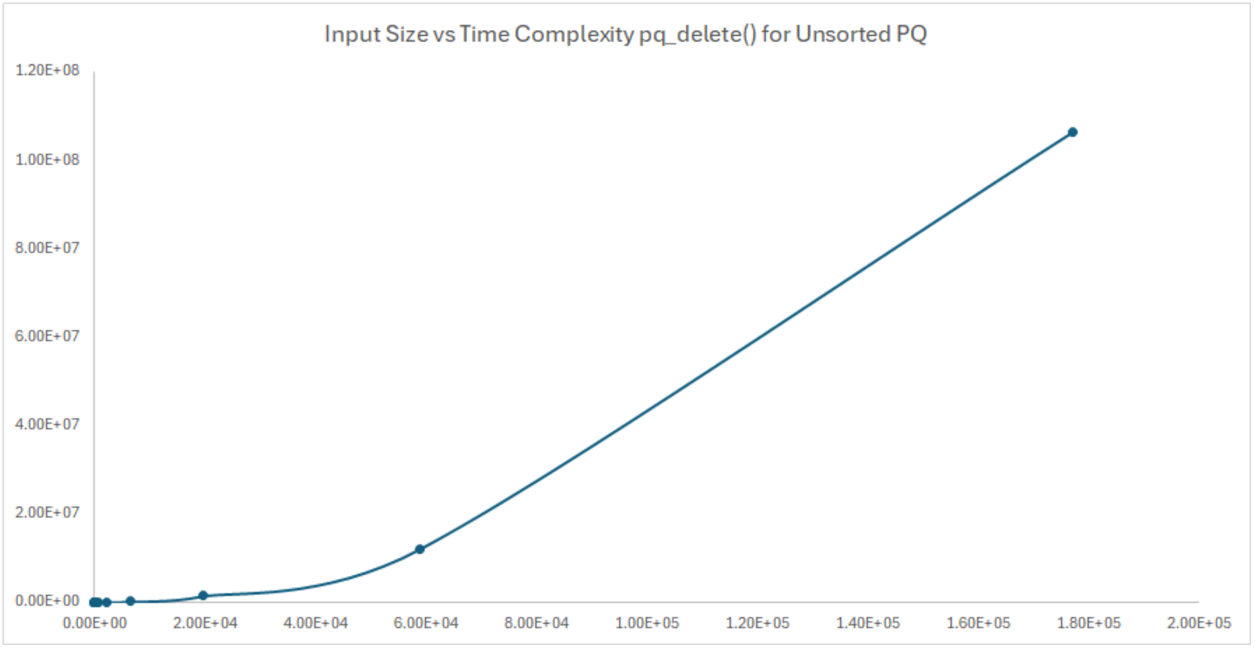
```cpp
void theNumbersMasonWhatDoTheyMean(){

    vector<int> result;
    ofstream fout;
    fout.open("sorteddelete.csv");
    for(int k=0; k<=11; k++){
        SortedPriorityQueue<int> pqh;


        auto start = chrono::high_resolution_clock::now();
        for(int i=0; i<pow(3,k); i++){
        pqh.pq_insert(rand() % (k * 3/2 + 1));
        }

        // auto start = chrono::high_resolution_clock::now(); -- For testing deletion/sorting
        // while (pqh.pq_size() != 0){
        //     pqh.pq_delete();
        // }

        auto stop = chrono::high_resolution_clock::now();
        auto duration = chrono::duration_cast<chrono::microseconds>(stop - start);
        result.push_back(duration.count());
    }

    for(int i=0; i<result.size(); i++){
        fout<<pow(3,i)<<","<<result[i]<<endl;
    }
    fout.close();

}
```

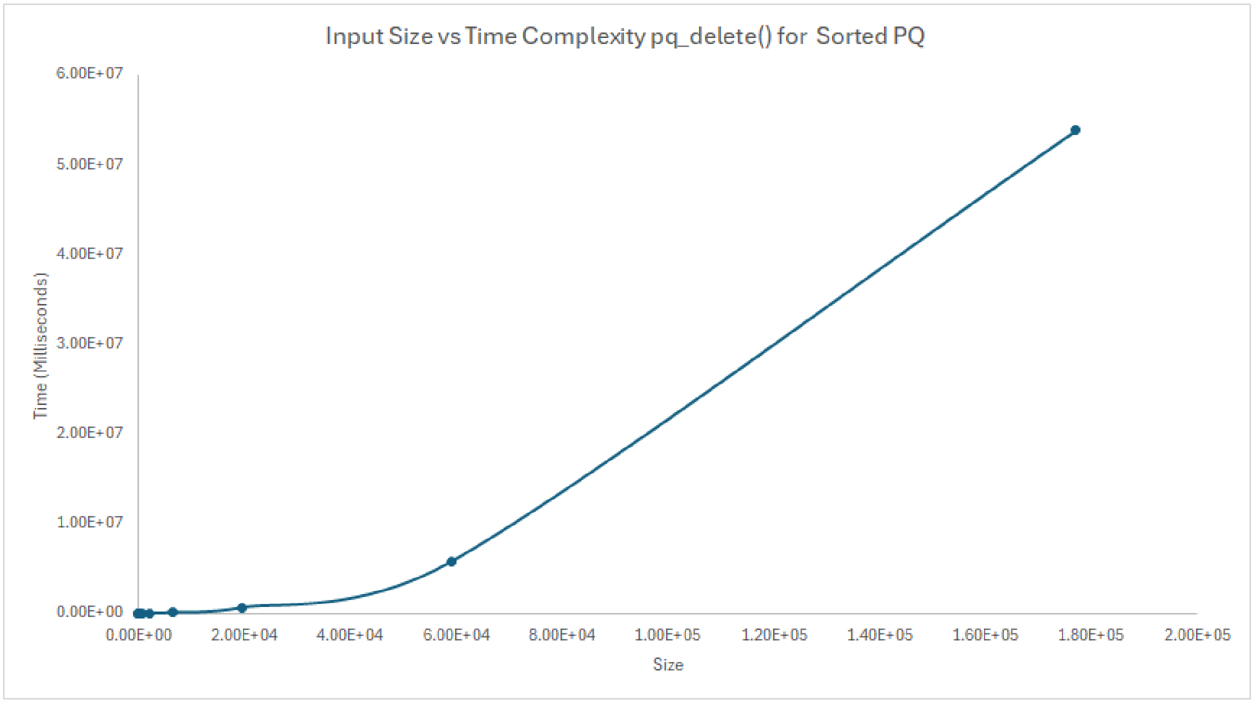
Experimental Results

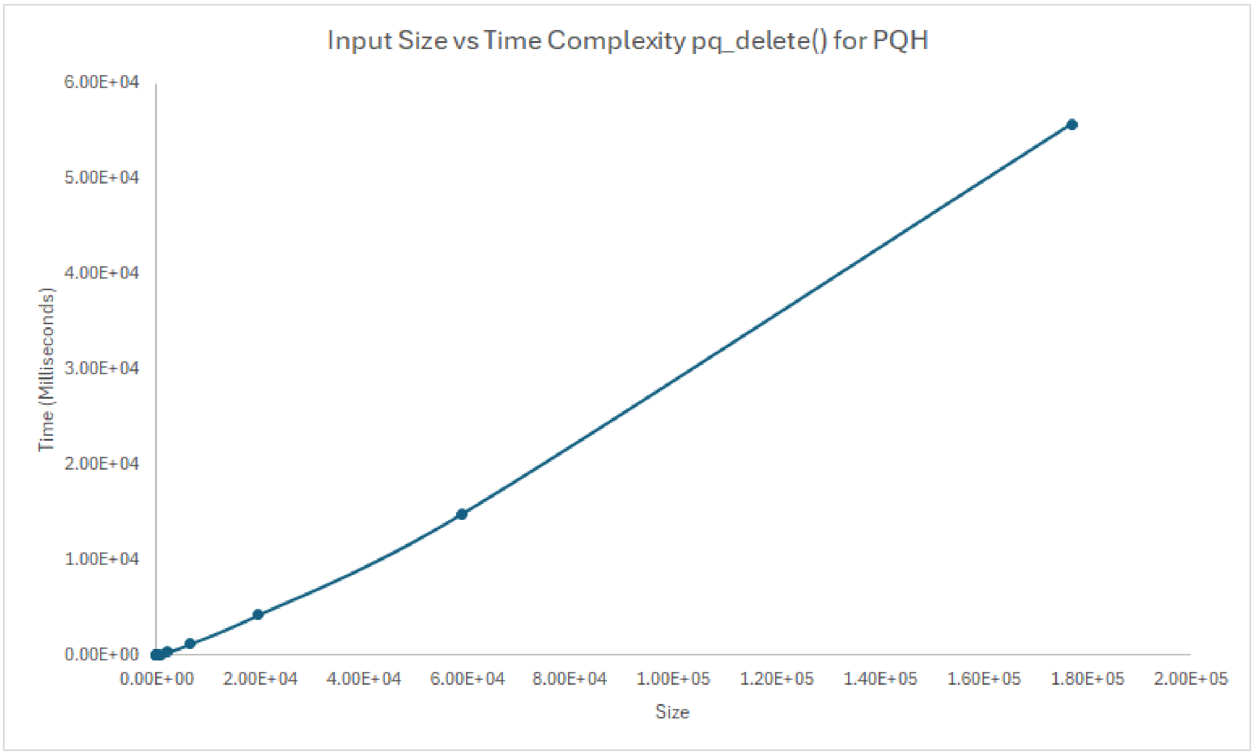





Input Size vs Time Complexity insert() for Heap PQ
Time (Milliseconds)
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
Size


Input Size vs Time Complexity pq_delete() for Unsorted PQ
1.20E+08
1.00E+08
8.00E+07
6.00E+07
4.00E+07
2.00E+07
0.00E+00
0.00E+00
2.00E+04
4.00E+04
6.00E+04
8.00E+04
1.00E+05
1.20E+05
1.40E+05
1.60E+05
1.80E+05
2.00E+05





The priority queue heap performed exceedingly better in the sorting operation, taking countless fewer milliseconds to operate as n grew larger. In terms of insertion, the unsorted priority queue took a little bit less time than the priority queue heap, but such a fact pales in comparison to the deletion operator. The sorted PQ performed on average the worst, taking an excruciating amount of time to perform the insertion operand, and performing worse than the PQH in deletion. Overall, the Heap Priority Queue is more efficient as a data structure.

Overall, it performs similarly, if not very closely, to the theoretical analysis. It is important to note, however, that for the deletion operand, the unsorted and sorted priority queue is resized every time a number is popped, to maintain the integrity of the data structure. Since it is an O(n) operation that is multiplied, there is an additional effect on the time complexity of these graphs. This is important to note because if such a thing wasn’t a factor, it would be likely that the Sorted Priority queue would be faster than the Heap in deletion (smaller big O scaling).