PA3 Report

Introduction:

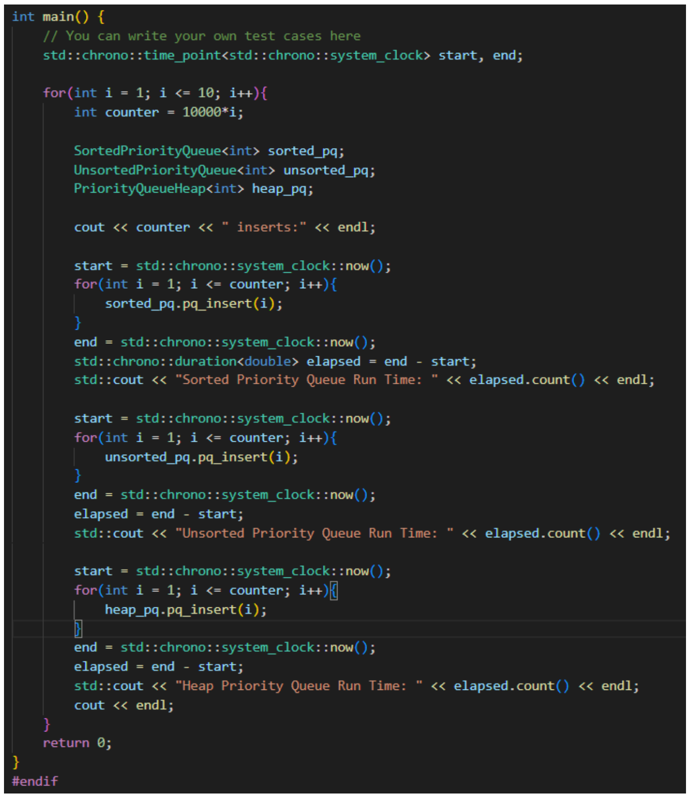
In this Programming Assignment, the aim was to implement a priority queue using 3 different methods. The first method dealt with using an unsorted array for the priority queue that would insert an element without sorting the array within the priority queue data structure and then it would use a search algorithm to find the smallest value in the priority queue to return it whenever the delete function was called and the overall sort algorithm used here is selection sort. The second method used a sorted array where the array was sorted as elements were being inserted into the priority queue and it would just simply return the first value in the array within the priority queue whenever the delete function was called and the overall sort algorithm used here is insertion sort. The final method dealt with implementing the priority queue with a min heap (built using an array) where every child is greater than its parent and this method would return the top node whenever the delete function is called and the overall sort algorithm used here is heap sort. All three methods resize their array by doubling their capacities in the case that they would need to insert an element and the array within them has no space.

Theoretical Analysis:

All three methods have a unique time complexity when it comes to their insert operation, delete min operation, and the overall sorting. The insert operation for the unsorted approach has an average time complexity of O(1), the delete min operation for the same has an average time complexity of O(n), and the overall sorting (selection sort) for this approach has an average time complexity of O(n^2) . The insert operation for the sorted approach has an average time complexity of O(n), the delete min operation for the same has an average time complexity of O(1), and the overall sorting (insertion sort) for this approach has an average time complexity of O(n^2) . The insert operation for the heap approach has an average time complexity of O(log n), the delete min operation for the same has an average time complexity of O(log n), and the overall sorting (heap sort) of this approach has an average time complexity of O(nlog n) . Therefore, from the information above, it can be said that theoretically the heap approach is the fastest and most efficient way to implement a priority queue and this is likely due to the fact that the min heap is stored as a binary tree which makes it easier to insert and fix the heap so that every child is greater than its parent in the heap which also makes it easier to find the min(root node) and swiftly fix the heap after removing the minimum element by just swapping the elements until the rule of the min heap is met. The disadvantage to this approach is the fact that this was the hardest and longest method to implement. The advantage with the unsorted approach is that it can consistently do the the insert operation with a run-time complexity of O(1) as the element would simply be added to the end of the array, but the disadvantage with this approach is that the average run-time complexity of this function is O(n) and this is because the minimum of the priority queue would need to found in the the unsorted array everytime the delete function is called. The advantage of the sorted approach is that it can do the delete operation with an average run-time complexity of O(1) as the first element of the array would simply be removed, but the disadvantage with this approach is that the insert operation has an average runtime complexity of O(n) as the elements need to be sorted as they are inserted into the array in the priority queue. The difficulty of implementation of both, the unsorted and sorted approach, is equally low when compared to the heap approach.

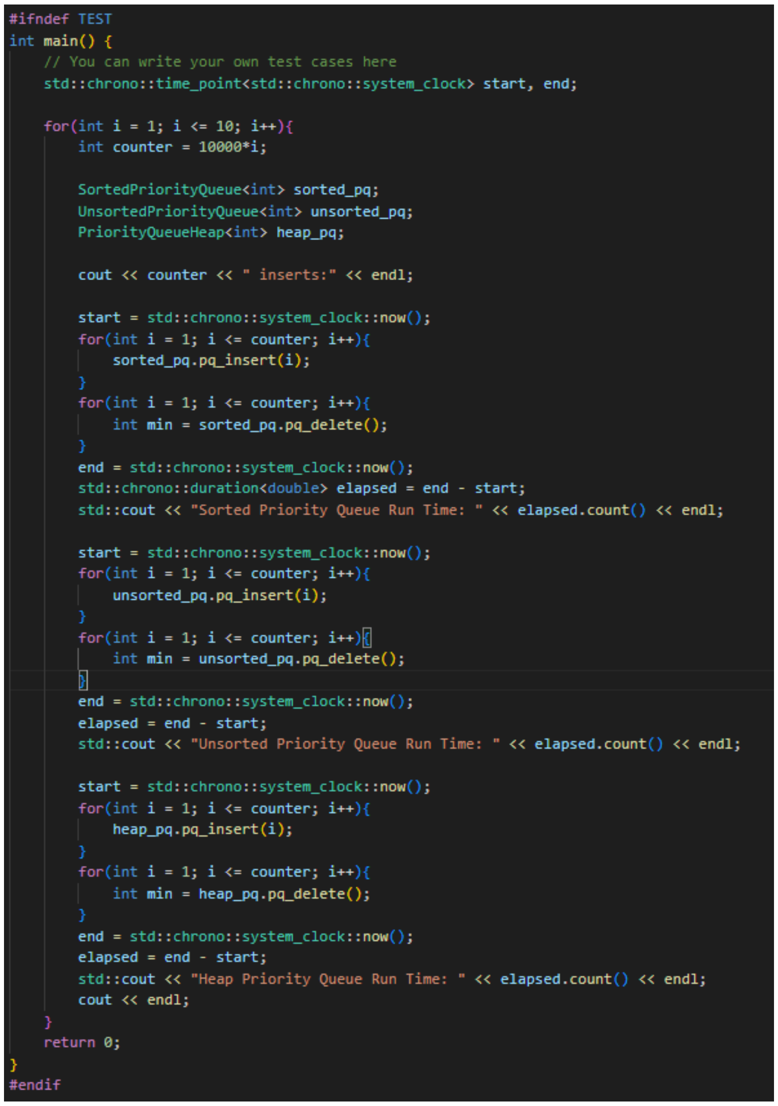
Experimental Setup:
This experiment was conducted in order to find the run-times of the push operation and the total sort for the 3 implementations. The highest number of elements inserted into the priority queue was 100000 and the lowest was zero with an interval of 10000 elements. The input for all three approaches were generated using a for loop. The run-time for the insert operation was recorded first after which the delete operation was added to the existing code and run in order to find the run-time for the total sort for the three implementations. Below is the code that was used to find the run-times:

Code for finding the run-time of the insert operation for all three strategies:



```
int main() {
    // You can write your own test cases here
    std::chrono::time_point<std::chrono::system_clock> start, end;

    for(int i = 1; i <= 10; i++){
        int counter = 10000*i;

        SortedPriorityQueue<int> sorted_pq;
        UnsortedPriorityQueue<int> unsorted_pq;
        PriorityQueueHeap<int> heap_pq;

        cout << counter << " inserts:" << endl;

        start = std::chrono::system_clock::now();
        for(int i = 1; i <= counter; i++){
            sorted_pq.pq_insert(i);
        }
        end = std::chrono::system_clock::now();
        std::chrono::duration<double> elapsed = end - start;
        std::cout << "Sorted Priority Queue Run Time: " << elapsed.count() << endl;

        start = std::chrono::system_clock::now();
        for(int i = 1; i <= counter; i++){
            unsorted_pq.pq_insert(i);
        }
        end = std::chrono::system_clock::now();
        elapsed = end - start;
        std::cout << "Unsorted Priority Queue Run Time: " << elapsed.count() << endl;

        start = std::chrono::system_clock::now();
        for(int i = 1; i <= counter; i++){
            heap_pq.pq_insert(i);
        }
        end = std::chrono::system_clock::now();
        elapsed = end - start;
        std::cout << "Heap Priority Queue Run Time: " << elapsed.count() << endl;
        cout << endl;
    }
    return 0;
}
#endif
```

Code for finding the run-time of the total sort for all three strategies:



```cpp
#ifndef TEST
int main() {
    // You can write your own test cases here
    std::chrono::time_point<std::chrono::system_clock> start, end;

    for(int i = 1; i <= 10; i++){
        int counter = 10000*i;

        SortedPriorityQueue<int> sorted_pq;
        UnsortedPriorityQueue<int> unsorted_pq;
        PriorityQueueHeap<int> heap_pq;

        cout << counter << " inserts:" << endl;

        start = std::chrono::system_clock::now();
        for(int i = 1; i <= counter; i++){
            sorted_pq.pq_insert(i);
        }
        for(int i = 1; i <= counter; i++){
            int min = sorted_pq.pq_delete();
        }
        end = std::chrono::system_clock::now();
        std::chrono::duration<double> elapsed = end - start;
        std::cout << "Sorted Priority Queue Run Time: " << elapsed.count() << endl;

        start = std::chrono::system_clock::now();
        for(int i = 1; i <= counter; i++){
            unsorted_pq.pq_insert(i);
        }
        for(int i = 1; i <= counter; i++){
            int min = unsorted_pq.pq_delete();
        }
        end = std::chrono::system_clock::now();
        elapsed = end - start;
        std::cout << "Unsorted Priority Queue Run Time: " << elapsed.count() << endl;

        start = std::chrono::system_clock::now();
        for(int i = 1; i <= counter; i++){
            heap_pq.pq_insert(i);
        }
        for(int i = 1; i <= counter; i++){
            int min = heap_pq.pq_delete();
        }
        end = std::chrono::system_clock::now();
        elapsed = end - start;
        std::cout << "Heap Priority Queue Run Time: " << elapsed.count() << endl;
        cout << endl;
    }
    return 0;
}
#endif
```

Experimental Results:

Graph for the insert operation for all three functions:

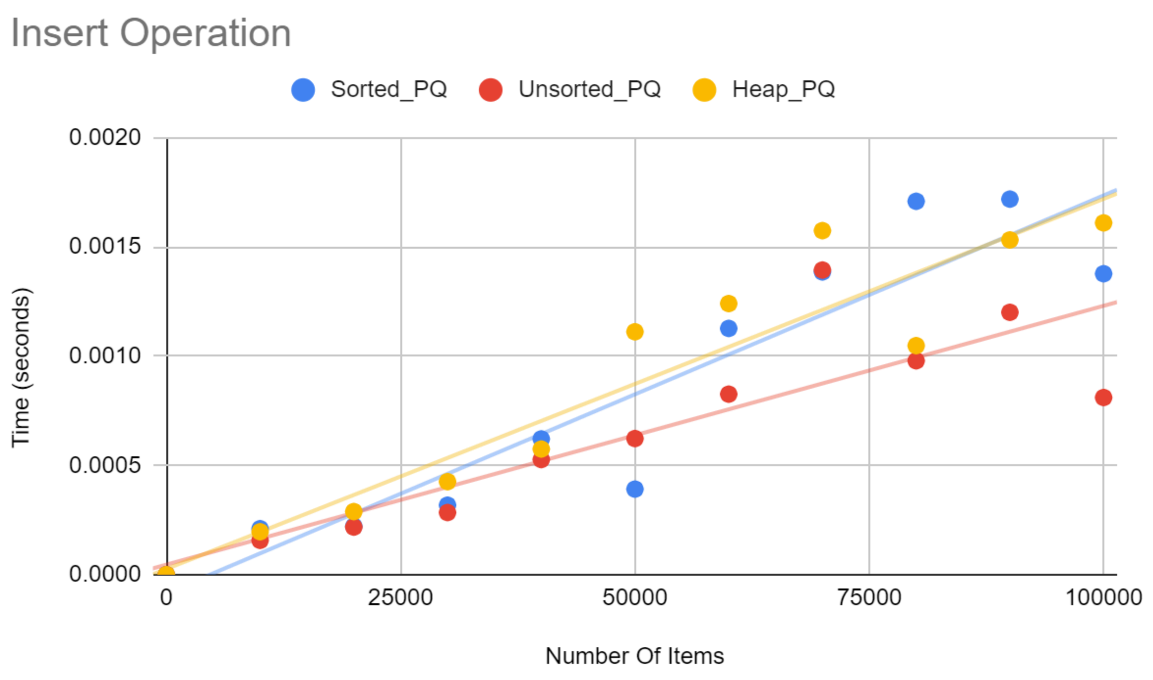
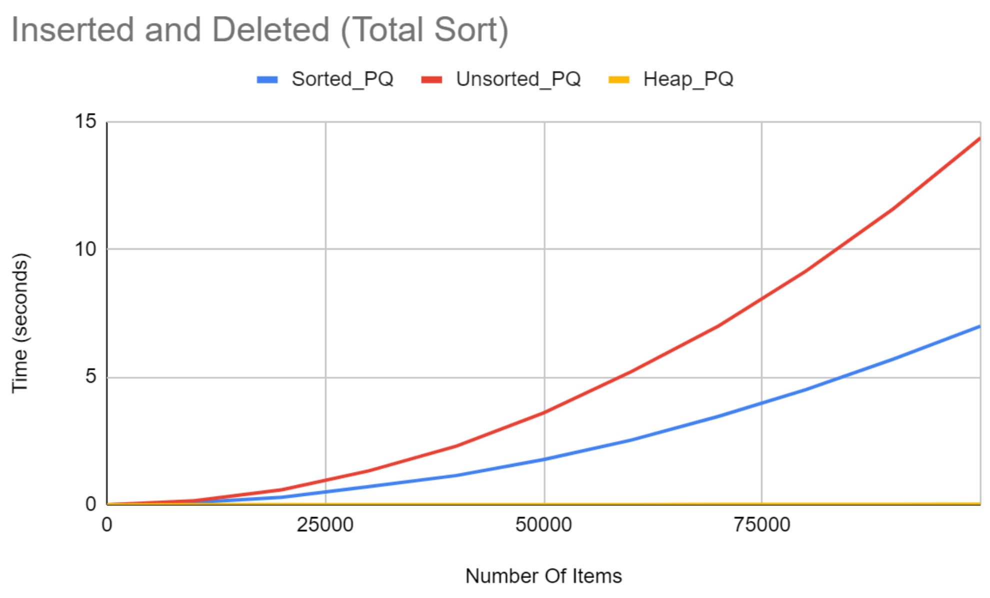


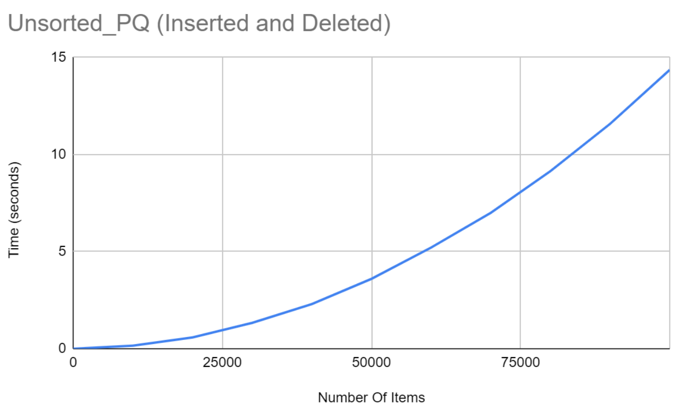
Table for the graph above(Insert Operation):

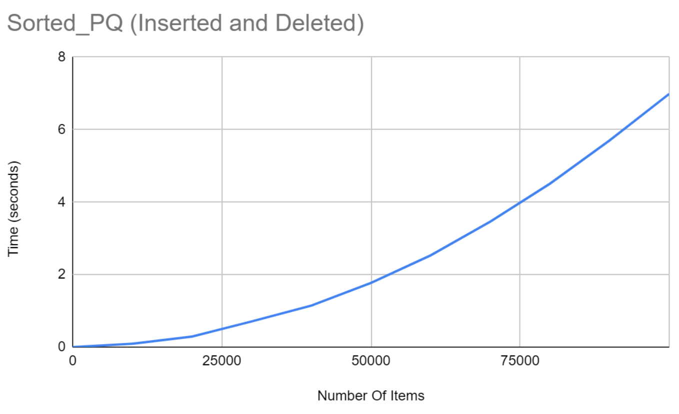
Number Of Items	Sorted_PQ	Unsorted_PQ	Heap_PQ
0	0	0	0
10000	0.00021035	0.000157002	0.000196475
20000	0.000221505	0.000217871	0.000288681
30000	0.000317984	0.000284384	0.000425718
40000	0.000621295	0.000526951	0.000574945
50000	0.000391591	0.000623012	0.00111162
60000	0.00112699	0.000826111	0.00124073
70000	0.00138566	0.00139419	0.00157447
80000	0.00170879	0.000978887	0.00104813
90000	0.00171877	0.00120067	0.00153208
100000	0.00137729	0.000810804	0.00161005

Graph for the inserted and deleted for all three functions:



Individual Graphs For Better Visualization(Inserted and deleted):





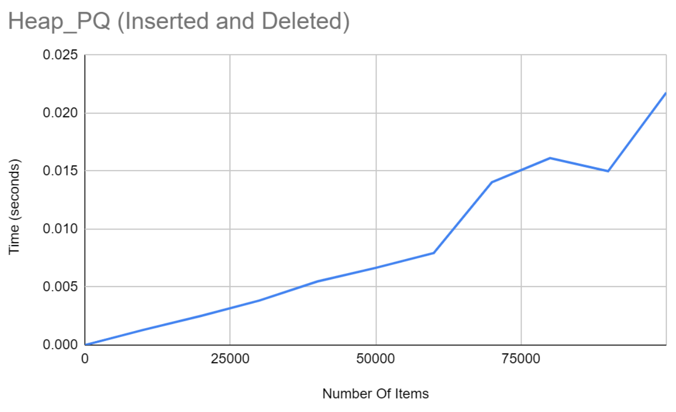


Table for the graph above(Inserted and Deleted):

Number Of Items	Sorted_PQ	Unsorted_PQ	Heap_PQ
0	0	0	0
10000	0.0907411	0.153833	0.00130063
20000	0.289791	0.582879	0.00252096
30000	0.70806	1.3297	0.00383949
40000	1.14304	2.29622	0.00548603
50000	1.76942	3.59698	0.00665318
60000	2.52915	5.20551	0.00793861
70000	3.45921	6.99547	0.0140351
80000	4.50595	9.13974	0.0161212
90000	5.69882	11.58	0.0149918
100000	6.98751	14.3582	0.0217658

<u>Analysis:</u>
To begin with, the insert operation of the unsorted approach was the fastest and this is due to the fact that the element would just simply need to be added to the end of the array within the priority queue since there is no order restriction for this approach. The insert operation for the heap and sorted approach has a higher run-time when compared to the unsorted approach and this is likely due to the fact that both of these have certain order restrictions and hence the insert operation cannot simply insert the element to the end of the array. The results for the insert operation of the unsorted approach aligns correctly with the theoretical run-time complexities as the theoretical average run-time complexity for the insert operation of the unsorted approach was on O(1), but the theoretical run-time complexity for the sorted and heap approach have a discrepancy as the heap approach should theoretically be faster than the sorted approach. The reason for this might be that since the elements were entered in order from smallest to greatest, the insert function of the sorted approach would just insert the element to the back of the array (like the unsorted approach) which might be a reason for this discrepancy with the theoretical complexity. Now, coming to the total sort run-times of each method, the fastest approach was the heap method. This approach out performed the other 2 approaches by a great factor, especially when it came to inserting and deleting a high number of elements. This might be due to the fact that both insert and delete work very effectively together(mentioned in theoretical analysis) in order to create simplicity in extracting the min and sequentially fixing the heap in order to overcome the order restrictions. This result is in agreement with the theoretical

average run-time complexity of the total sort for this approach. The next fastest approach with the lowest run-time for total sort was the sorted approach(insertion sort). This result is not in agreement with the theoretical average run-time complexity of the unsorted (selection sort) and sorted approach as the theoretical average run-time complexity of both the approaches is O(n^2), but the results from the experiment show that the sorted approach is considerably more efficient than the unsorted approach. The reason behind this might be that sorting the array while inserting (especially due to the fact that the input was in ascending order) might take a lower amount of time in order for the delete function to extract the min value and fix the array after removing the min value rather than using a search algorithm in the delete function in order to find the min value.