**Task 1**

A priority queue has its advantages and disadvantages when compared to a standard queue. Priority queues allow for more complicated data to be stored and worked with. Since it inserts and removes based on priority instead of the value itself, this allows for a wider range of more complex uses as opposed to a standard queue. For instance, we can insert and immediately access a value in a priority queue if it has a higher priority than all the other values. On the contrary, a standard queue always puts the value at the end of the line, and accessing that value again will take as many iterations as there are items in the queue. Since the priority queue allows for more these complex interactions, it naturally requires much more complex code. It also has to search the list for the proper location on every insert, so performance on insertions are slower than a queue, especially when dealing with large sets of data. Regardless, neither is necessarily better or worse. Each has their own specific use cases.

**Task 2**

Heaps and binary trees are similar, but each contain their own pros and cons. Since binary trees are always sorted, inserting new values into the proper spot can upset the balance of the tree. This eventually requires a multitude of rebalances throughout a tree’s life. In heaps, this is alleviated since the shape is complete and consistently accounted for in addition or removal from the heap. This full shape allows for the fastest possible processing when accessing elements. However, heaps are also much more complex and messy. Accessing a particular value in a binary tree simply requires searching through based on the order of the leaves’ values, making it easy to check if a value exists. Since heaps are disorganized aside from parents being smaller or larger than their children, accessing a particular element can be difficult and inefficient. Of course, heaps are not primarily used for this purpose. Where heaps shine is in accessing the lowest or highest value depending on if it’s a min or max heap. This is incredibly efficient since the lowest or highest value will always be at the top of the heap. On the contrary, binary trees require searching through each level to find the left-most or right-most item to access the highest or lowest value. In the end, like the priority queue and regular queue, each has their own particular usage.

**Task 3**

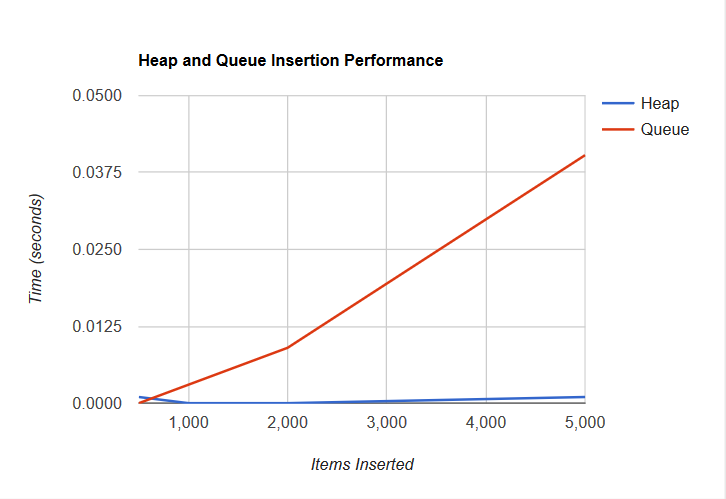


Figure 1 - Heap and Queue Insertion Performance



Figure 2 - Heap and Queue Removal Performance

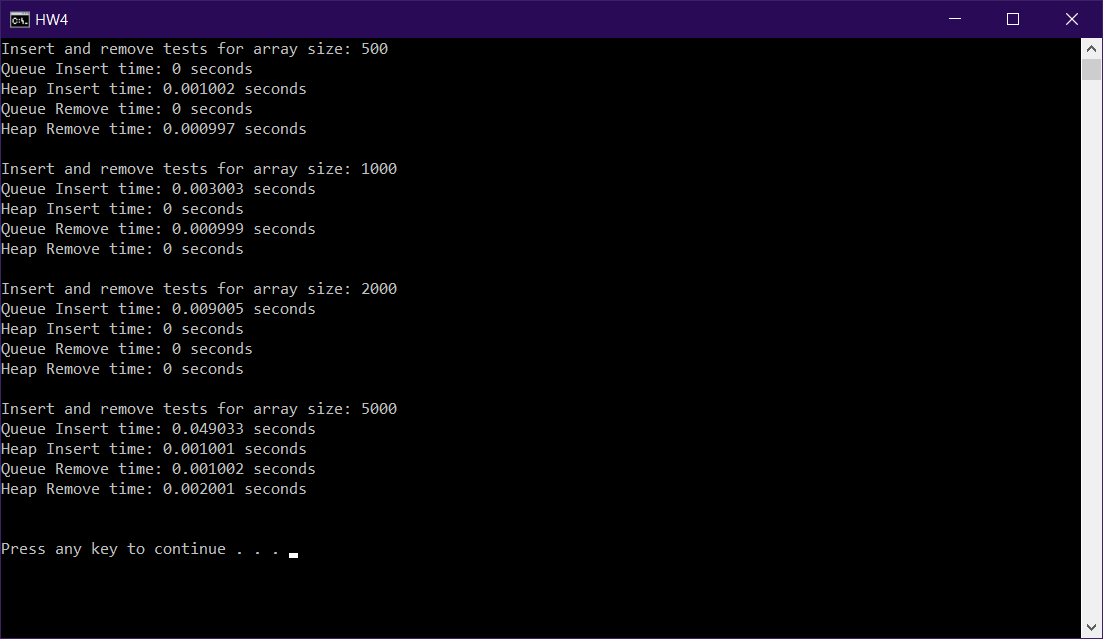


Figure 3 - Raw Insertion and Removal Test Data

Figure 1 displays the results of the insertion tests. The priority queue insertion is O(n) and heap insertion is O(m) where m is the number of levels in the heap. The n in the priority queue insertion increases by one every new item, whereas the m only increments every couple of items since the structure is always complete. As a result, it was hypothesized that the priority queue insertion would be less efficient in dealing with insertion. This would occur in any case unless the number of values in the queue is less than the number of layers in the heap. This specific case can be seen at the very beginning of the Figure 1, but the remainder of the graph drastically holds true to the decreasing efficiency of the queue.

Figure 2 displays the results of the removal tests. Removal from our priority queue is only O(1) since it only requires the first value and is implemented through a linked list. However, removal from the heap is O(m) where m is the number of levels in the heap because it needs to reheap down after each item’s removal. Thus, it can be concluded that the queue would be more efficient in any case because a heap would never have less than one level. Even when a heap doesn’t require reheaping on removal, it still needs to do additional checks to ensure this, so the overall computation time is still larger than that of a priority queue. This is confirmed through the Figure 2 results where the heap consistently takes more time than the queue aside from some data that was too fast to measure.

**Discussion**

In this lab, the topics of priority queues and heaps were explored. These are both important to this course and careers in Engineering since they could be particularly useful in solving different problems. Priority queues allow for the sorting and accessing of data based on priority instead of the actual value, which could allow for operations to be performed or data to be pulled in the order that the user or developer desires rather than the order of their value. Heaps can be useful in situations that require quick accessing of either end of a data set and can even be used to implement priority queues. The performance of these data structures was also analyzed through Task 3. Through this homework assignment, students can easily see when priority queues are more efficient through linked list or heap implementations. If the situation arises where a priority queue is required in a program, students can then look back on the results and consider whether faster removal or insertion would be required so they can use a linked list or heap respectively.

.

**Task 1 and 2 Testing Demonstrations**

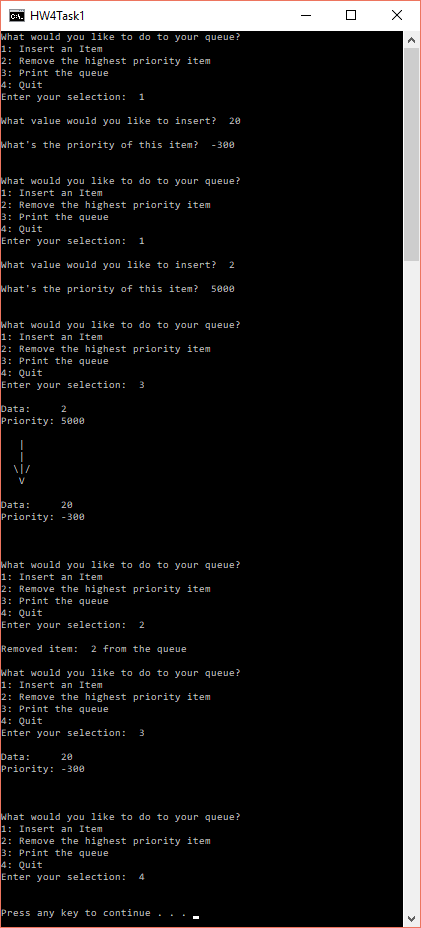


Figure 4 - Task 1 Priority Queue Tests

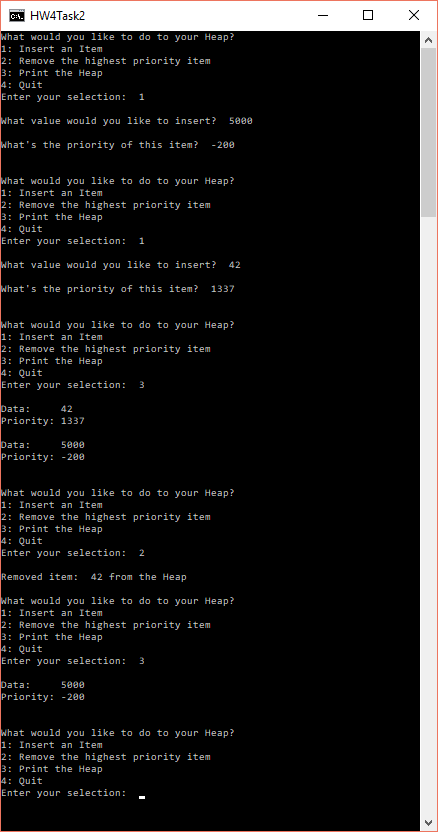


Figure 5 - Task 3 Priority Heap Tests

**Compilation**

HW4Task1.cpp contains the main for Task 1

HW4Task2.cpp contains the main for Task 2

HW4Task3.cpp contains the main for Task 3

The code was compiled using the GNU compiler with standard Windows settings,