Introduction:

The objective of this assignment is to implement the priority queue using both a linked structure and an array. These priority queues are then used to implement the heap sort algorithm. I have implemented this with a linked binary tree, an array, and the heapify or bottom up heap construction algorithm. I have compared these implementations by using the run time of each implementation as the number of items being sorted increases.

Theoretical Analysis:

The PQHeapTree implementation of the priority queue ADT using a linked binary tree structure as well as the PQHeapArray implementation of the priority queue ADT using an array both have 5 functions: insert, removeMin, min, size, and empty. The heapify algorithm used to sort is implemented with the PQHeapArray structure.

I will begin by analyzing the runtime of the PQHeapTree's functions. The insert function just adds another node (constant time) and then bubbles up the value if necessary. The bubbling up portion takes O(log(n)) time due to only having to go up one side of the tree. The min function returns the value at the root of the tree and therefore takes constant time. The removeMin function swaps the value at the root with the value of the last inserted node then removes the last inserted node. After the removal it then bubbles the value at the root down to its proper spot. This takes O(log(n)) time because it will only potentially operate on log(n) nodes due to selecting the smallest child and swapping. The size function take O(1) time due to returning a size value the is incremented and decremented with calls to removeMin and insert. The empty function also takes O(1) time due to returning true if the size is 0 and false otherwise.

The PQHeapArray Implementation has similar runtimes for it's functions. The insert function also runs in O(log(n)) time. This time however instead of swapping the nodes after inserting the value at the bottom of the tree, it swaps the values in the array at the indices designated as parent/child after inserting the value at the next index in the array. This takes less time but has proportionally the same runtime as PQHeapTree. The min function returns the value at the 0^{th} index of the array storing the keys. This takes O(1) time. The removeMin function works in the same fashion as the PQHeapTree implementation's, but swaps the first and last index of the array instead of the nodes. It then swaps the parent/child indices until the array is back into the correct order. This take O(log(n)) time. The size returns a value incremented and decremented with calls to insert and removeMin respectively and therefore runs in O(1) time. The empty function returns true if and only if the size value is 0, therefore also running in O(1) time.

The Heapify Algorithm included in the PQHeapArray implementation takes an array input and returns a PQHeapArray after looping recursively to put the array in properly sorted order. This takes O(n) time.

I implemented the heapsort algorithm 3 different ways. Each way used a different one of the structures above. For both the Array and Tree implementations, I looped through the array input into the function and inserted each value into a heap. I then removed each value and assigned it to the proper index in the array. For the heapify implementation I used the heapify function to sort the array and then removed each value and assigned it to the proper index in the array. For both the array and

tree implementations, the amount of time taken to sort the array was O(n*log(n)). For the heapify implementation, the runtime was O(n*log(n)) but generally took longer than the other 2 implementations.

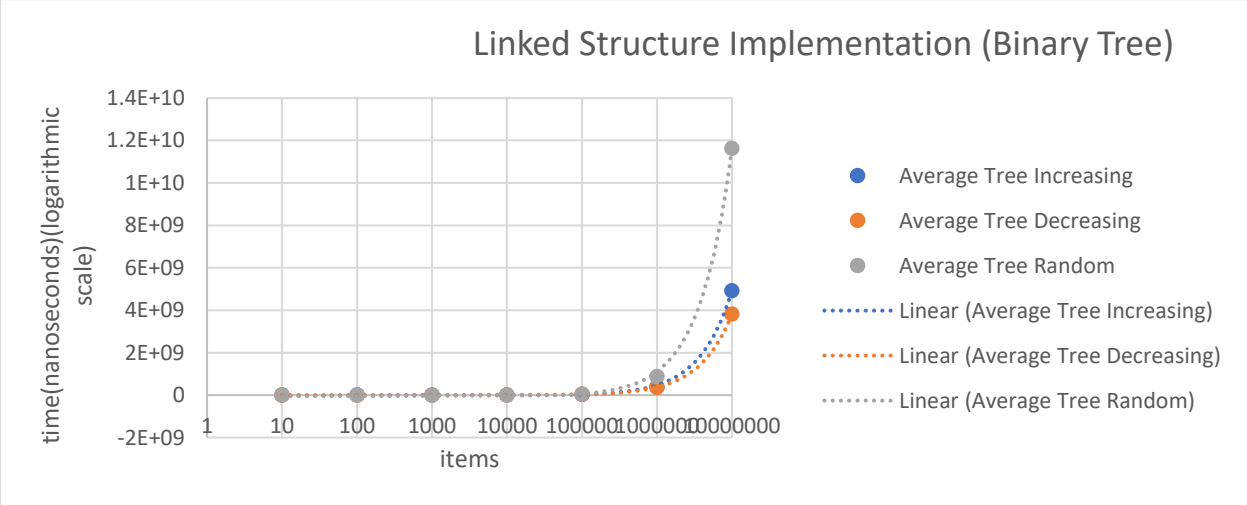
Experimental Setup:

To perform the experiments if this assignment I used a Dell Inspiron 5577 with an Intel Core i7-7700HQ CPU @2.80 GHz. This laptop has 8 GB of RAM and a 64-bit operating system. For the compiler I used Cygwin v3.0.7 and CMake v3.14.5 in combination with the CLion 2019.2.1 IDE. For this program I used C++14.

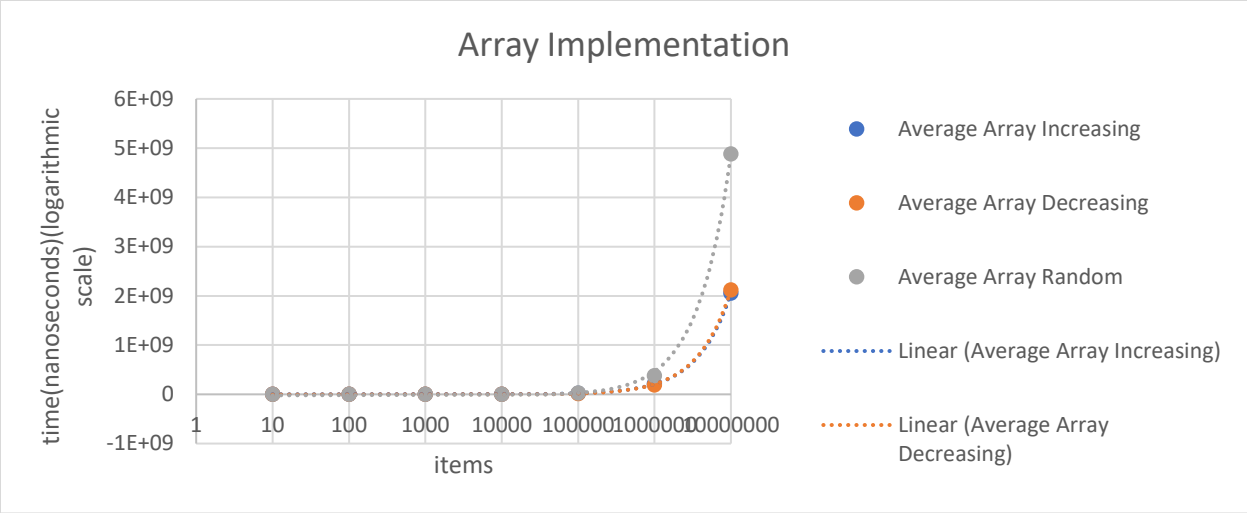
To measure the amount of time to sort n elements I used the high_resolution_clock from the C++ chrono library. I set it to nanoseconds in order to get accurate measurements on the lower test cases. I chose nanoseconds because milliseconds and coarser measurements did not give accurate results below 1000 elements.

I ran the program and recorded results 5 times per implementation per order (increasing,decreasing,random). I temporarily comment out the test for one implementation in order to make sure the results are accurate. I chose to stop the tests at 10000000 elements for both the linked structure and array implementations because the laptop used to test the would freeze up and stop responding at 10000000 elements. For the heapify implemention I stopped at 10000 for the same reason.

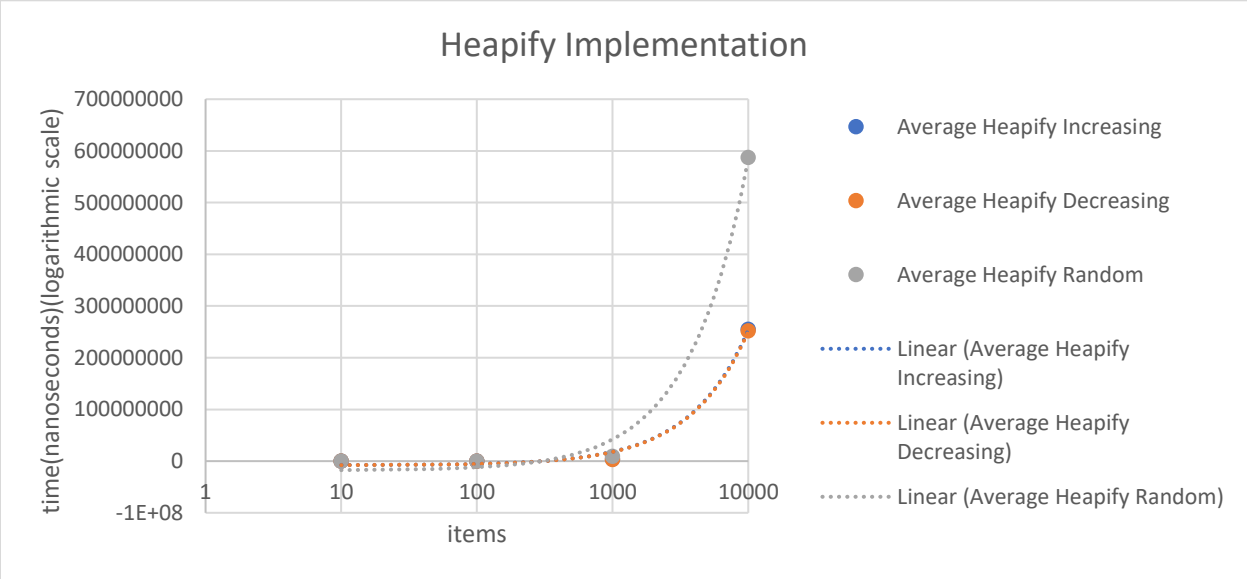
Experimental Results:



This graph shows the runtime of the binary tree implementation of a priority queue being used to sort an array of n items as n is increasing. The amount of time used can be seen here to be proportional to n*log(n) where n is the amount of items to be sorted. For both strictly increasing and strictly decreasing items the amount of time taken is about the same. For randomly selected value between 0 and 99 inclusive, the amount of time is much higher due to having more operations being run in order to follow the heap order property.



This graph shows the runtime of the array implementation of a priority queue being used to sort an array of n items as n is increasing. The amount of time used can be seen here to be proportional to n*log(n) where n is the amount of items to be sorted. For both strictly increasing and strictly decreasing items the amount of time taken is about the same. For randomly selected value between 0 and 99 inclusive, the amount of time is much higher due to having more operations being run in order to follow the heap order property. In all cases, however, the array implementation was faster than the binary tree implementation due to the operations for swapping value being simpler and therefore faster. The space used for this implementation is less efficient and less flexible than the binary tree implementation.



This graph shows the runtime of the array implementation of a priority queue being used to sort an array of n items as n is increasing by using the heapify method. The amount of time used can be seen here to be proportional to n^2 where n is the amount of items to be sorted. This is due to an issue in the code where I copied the values of the array input into the sort function to the items array in the heap array. This causes the program to run slower and not be the expected n*log(n) runtime. This is also the reason for the maximum number of items being 10,000.

For all implementations of the algorithms and structures presented here, the theoretical analysis match up with results except for the heapify sorting algorithm. In practice, the implementation I had ran slower than expected due to a known issue that was not fixed due to time restrictions. This issue has been marked in my code with comment lines.