Christian F. Sousa
CSCI 2270 Final Project

Runtime Analysis of Various Priority Queues

For this project I was tasked with building three different priority queues and evaluating the runtimes for each of them to see which is best for a given data size. But what is a priority queue, and how will the "runtime" be evaluated?

To stay brief a priority queue is like a line of people where each person holds a number. Then the people are ordered by their number either least to greatest or greatest to least depending on what is subjectively determined to be the "highest priority". Runtime programming is defined for the amount of time that it takes a given program to accomplish its task.

In the scenario given I had to create 3 priority queues hold the data for a large number of pregnant women who had gone into labor. It was determined that therefore the priority was least to greatest based on the expected time until the child was born. There was also a secondary condition if the expected time to birth was the same for multiple women. This was the expected amount of time each woman was expected to spend with the doctor. I discussed what the runtime should be with various TA's and we came to the conclusion that the runtime would be the amount of time it took to read in all the information on the women, insert and sort it in the structure and then remove it.

For all three implementations there were a few things I the same. I decided to use the same struct to hold the data for the women, so I did not need to rewrite it for every implementation. I did this by creating a .hpp file that contained the struct and its constructor, then I included the .hpp file in any other file that required the struct. Also, I choose to read in the data from the .csv file the same way. When the program began it immediately opened the file and started reading line by line, (lines deiminated by '\r'), parse the line on the comma character, create an instance of the defined struct and then pass it on to insertion function that would put it into the given data structure.

One of the priority queue implementation evaluated in the project is a structure called a Linked List. In this structure all the nodes (instances of the struct) are connected by pointers and the front and end are defined with pointers within the class. With this structure there are a few things that need to be taken into account when inserting. Here there is no way to jump to a particular node in the structure, the program must start at the beginning or end and iterate backwards or forwards using the points that link the nodes. As the program traverses it compares the value of a node to the node it is trying to insert. If the nod trying to be inserted is greater than the node in the struct, the program goes to the next node. When the node to be inserted is less than the compared node. The program takes the pointers of compared node and the node before the compared node, points them to itself and finally sets its pointers to the compared node and the node before it. There are two edge cases here that are tested for at the very start of the insertion function. The first is if the structure is empty, the second is if the structure has only one node. The only other thing that needs to be accounted for is if when inserting the compared node is the front or end of the structure.
Removing from this structure is very simple, as the front object is the highest priority at all time. Therefore, the removal is to remove the very first node and then update the front pointer to the new first node in the structure. The only edge case here is when the structure is empty.

The next structure is a priority queue made with the Standard Template Library in C++11. With this implementation there is a lot less done by what I have written as this preprogrammed function can add, remove and sort on its own. It used a Vector to store the nodes and reorders the vector every time a node to add or remove. My implementation of this was to parse the data, create the nodes and then pass them to the STL to be inserted. Removal was even simpler as it has a built-in function to remove the highest priority item and return it so that it can be used or deleted.

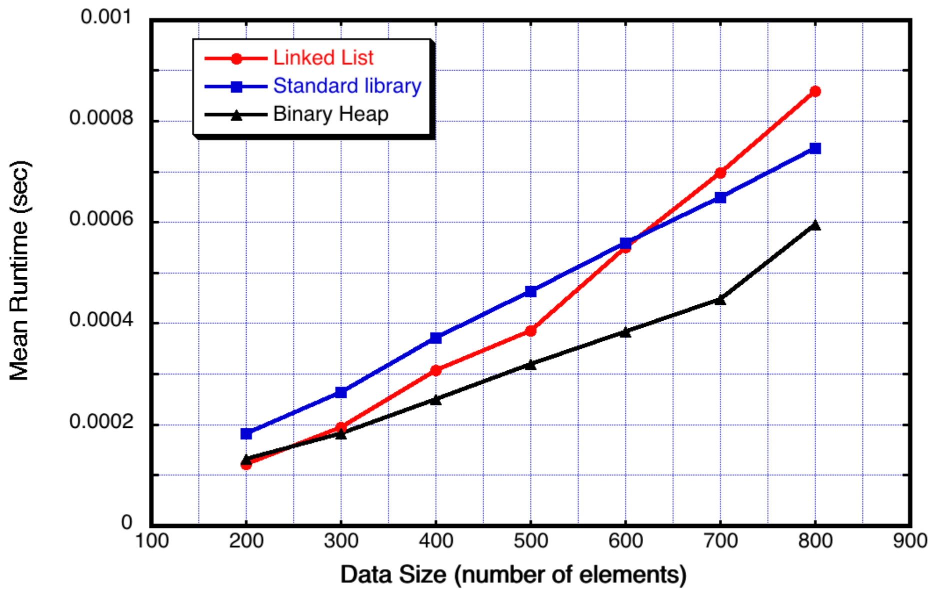
The final priority queue implementation involved a structure called a binary Heap, this structure is vastly complicated and with be explain as simply as possible. It is essentially a binary search tree represented (where the parent has greater priority than its children) in an array. Where instead of parent and child relationships are defined with pointers, they are defined by indices in an array of set length (for our purposes). This makes both insertion and deletion very complex. To make this a little simpler I made various helper functions. The first was a set of three small functions where when given an index the functions could return the indices of the parent or children of the imputed index. The second larger function made was a Boolean function that could directly compare two of the patient structs, it did this by looking at the values in the structs and returning true if the first was greater than the last and false otherwise.

When inserting in a binary heap, the object is inserted at the next available opening in the array and then a function is called to compare the newest object to the ones around it. If the new one is of higher priority than its parent, it is swapped and then the function is repeated at the new index until the parent of the newly inserted is of higher priority than the child.

Removal from the structure is also complicated. The program starts by removing the first thing in the array and then calling a function called heapify to decided what the new highest priority will be, after heapify is completed it returns the node it removed. Heapify is a recursive function that looks at the two children of the index given too it, evaluates them both and moves one of them to the parent position. Then it calls itself again on the parent index. This reshuffles the whole array to preserve the qualities of binary heap such that the parent is always of higher priority than the children.

The data set given was a csv file where each line contained the name of a woman, the time till she gives birth and how long she would need to spend with the doctor. In this case I decided that the highest priority would be those with the shortest time until they gave birth. There is also a secondary condition if two or more women have same time to birth. It is the length of time they are expected to need to be with the doctor. Again, the higher priority here is the one with the smaller values.

DATASIZE	**LINKED LISTS**		**STANDARD LIBRARY**		**BINARY HEAP**	
NUMBER OF ELEMENTS	Mean	STDEV	Mean	STDEV	Mean	STDEV
200	0.000121732	5.18799E-05	0.00018135	5.70955E-05	0.000132566	4.27187E-05
300	0.000195128	6.39091E-05	0.000263584	5.81217E-05	0.00018255	6.61021E-06
400	0.000307358	0.000203903	0.00037066	0.000162954	0.000249344	5.69691E-05
500	0.000386100	8.93555E-05	0.00046381	8.8908E-05	0.000319798	7.3353E-05
600	0.000549370	0.000164821	0.00055925	0.000205076	0.000382638	7.49994E-05
700	0.000697546	0.000208120	0.000649366	8.06961E-05	0.000447364	0.000595648
800	0.000859122	0.000109170	0.000745964	9.03324E-05	0.000595648	0.000442868



The table and graph shown above show all the data I have collected on the three implementations. Data was not taken with 100 elements as I was having issues with runtimes too small to be measured.

The Linked list was the fastest for the first test with 200 elements. As the number of elements increases the runtime increased in a linear fashion. The early success is most likely due to its simplicity in the complexity of each individual action.

The Binary heap quickly overtook the linked list in runtime as while the insertion and deletion algorithms are much more complex they are more efficient as they are called fewer times. As the number of elements increased the complexity also increase but not in an exactly linear fashion. I am not exactly sure what kind of trend this is. However, it would most likely show up with larger testcases.

The Standard Library implementation was slower than both until about halfway through the test. This is most likely due to the fact that it uses a vector which has to include array doubling as things are inserted so on top of the insertion complexity there is the extra runtime for when the arrays (within the vector are being double). There is also an increase in runtime as the number of elements increases, this one also seems very linear which I think Is incorrect and the correct relation would appear with more testcases of larger data sizes