Jon Abrahamson
CSCI 2270 Final Write-up
CA: Andrew Altomare

The Very Very Pregnant People Pagent

In the this project, I was tasked with creating a priority queue that was sort a group of 880 women currently going through labor at the exact same time. The priority was to first be sorted by their “time to delivery”. If there is a tie, then it was to next be sorted by estimated treatment time. In our data set, this would settle all priority ties and accurately set up a queue that moist efficiently ranks which patient should be attended to first. The data was supplied to us in a .csv spreadsheet file. It was given in the order that follows: *Name, Time to Labor, Treatment Time*. So for instance, if there are two patients Shelby, 72, 105 as well as Isabella, 72, 80, then the first number will be prioritized first. In this case, both patients of a time to labor value of 72. So, because it is a tie, then the next number must be evaluated to break the tie. Isabella in this case has a treatment time of 80 where as Shelby has a treatment time of 105. Isabella would then be prioritized and thus placed ahead in the priority queue. The same system is followed until all patients are sorted.

In order to determine the priority, I implemented three different data structures. The first was apart of the basic Standard Library. This is a container that is designated to keeping its first element as the greatest/smallest value it has according to a strict weak ordering criterion. The standard container class for a STL is a vector and this is what I used while implementing the priority queue. The next data structure that I used was a Linked List. This list is created so that the highest priority element is always at the head of the list. The list is then arranged in descending order of elements based on their priority. This allows me to remove the highest priority element in O(1) time. To insert an element we must traverse the list and find the proper position to insert the node so that the overall order of the priority queue is maintained. This makes the push() operation take O(n) time. The final Data Structure that I implemented was a heap. A heap is a way to organize elements in a specific order to obtain quick retrieval. A Min Heap is similar to a priority queue using the STL. The main difference is that in a heap, elements can be added at any time.

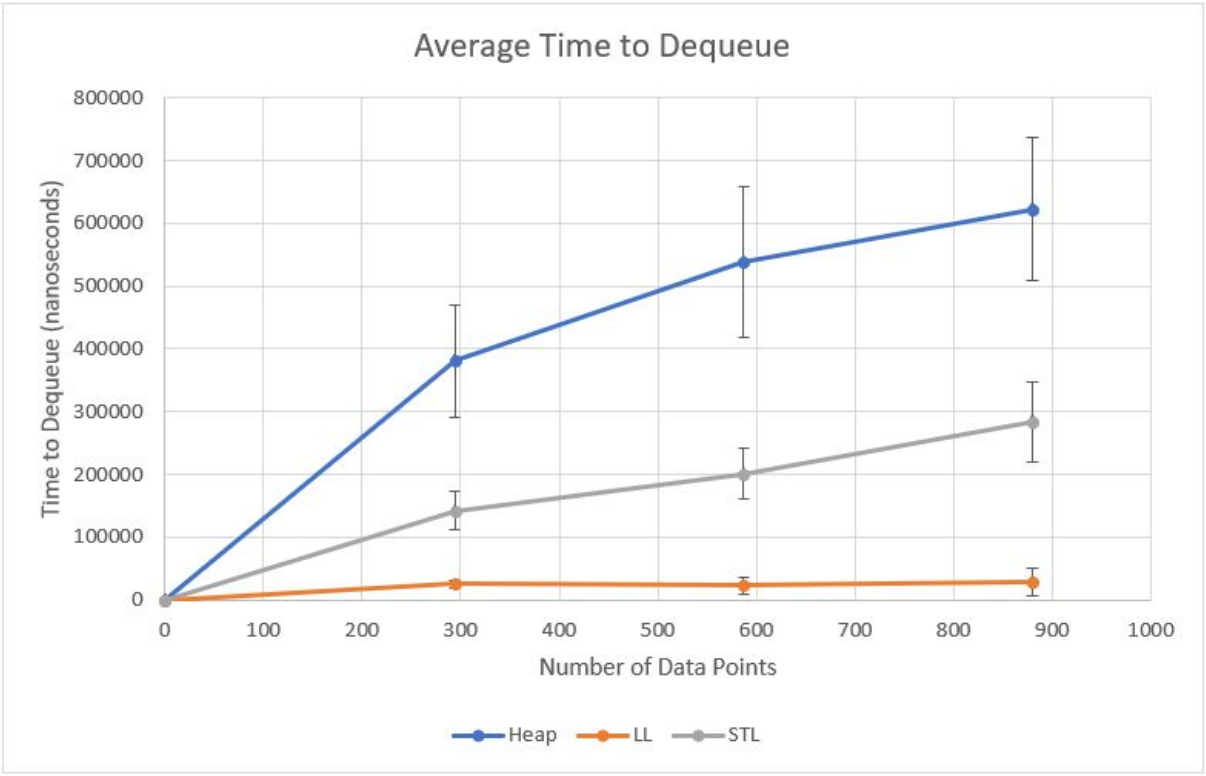
Next, the run time of each structure must be analyzed. In order to do this, I used the <chrono> time library. There are automated high resolution clocks included in the library. Although the project asked for me to measure run time in microseconds, I found that using nanoseconds gave me the best results. A nanosecond is $1x10^{-9}$ seconds and a microsecond is $1x10^{-6}$ seconds. In each case, I created a separate file that contained a main driver function with only the code for a specific data structure. Then, the chrono clocks are used to measure the time it takes to either queue or dequeue the data structure and write this data to a different .csv file. I then implemented a for loop that

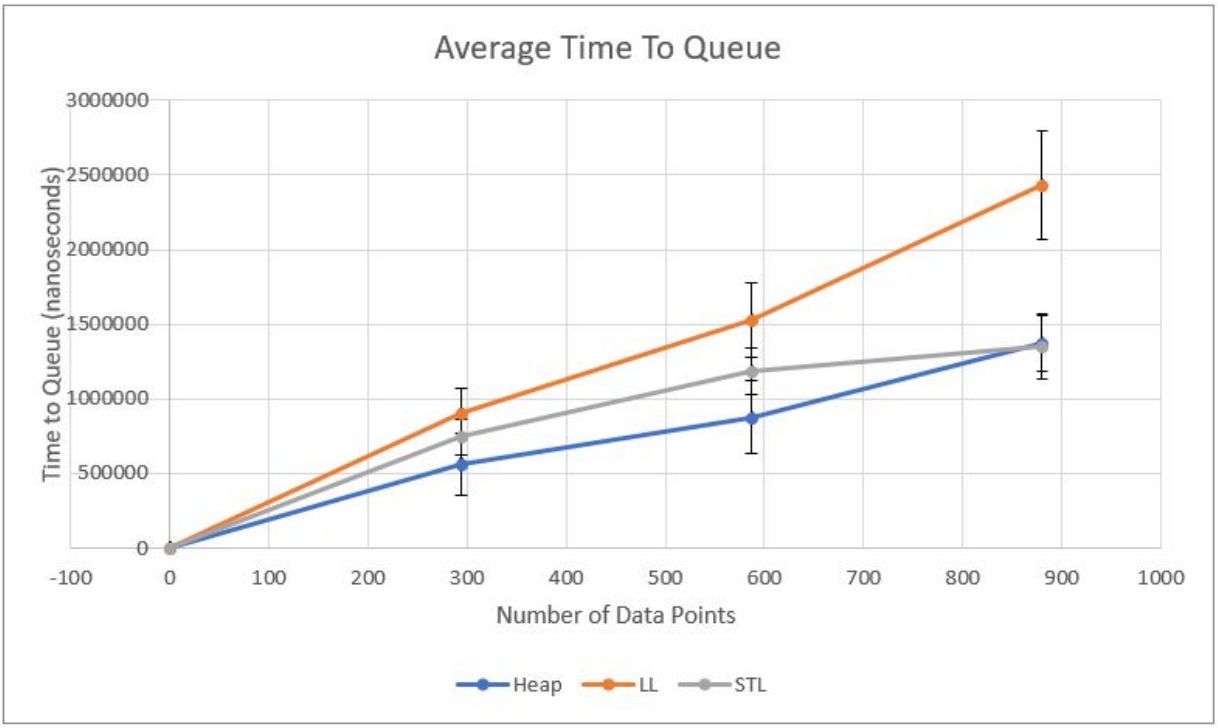
would loop this process 500 times to account for multiple testing to obtain the most accurate data. Another important thing to note is that these tests were also performed with a variety of number of patients or data points. Time it took to process different amounts of patients was also recorded to the files. I wrote this information to a .csv file because it made it far easier to find the average as well as the standard deviations of the 500 test results for each data point.

The results were then collected, compiled, and analyzed. It can be seen that there are very distinct differences from data structure to data structure when it comes to the time it takes for the data to be queued and subsequently dequeued. The data was analyzed in terms of the averages of these times. For the dequeue of the priority queue, the speed in which this action is conducted follows the order from shortest time to longest time: Linked List, STL, then Heap. Linked List far exceeds any of the other data structures in dequeue time because it follows an O(1) delete time. This also explains why no matter how many data points that are used, the time it takes to dequeue was relatively the same. Heap was the worst in this category because its sorting process had a O(n log(n)) time.

Next we look at the times it takes to Queue the priority queue for the different data structures. In this case, Heap was the best. This is because Heaps do not need to be sorted at right away during queue. When my code is ran, when the heap is created, it is not in order. However, when it is dequeued, it is in order. This allows for information to be added whenever and still be able to be in correct priority order when dequeued. Linked List is subsequently the worst. It has an O(1) queue time. The STL library in both the queue and the dequeue are in the middle. It is hard to determine which data structure is the best. Linked Lists are the best for dequeue but the worst for queuing the data. Heaps are the opposite in that the are best for queueing and the worst for dequeue. STL priority queues sit right in the middle of the two.

Additionally, the standard deviations are added to the graphs below to show the range in which much of the data can be found around the general averages of the times. As you can see the standard deviations maintain the accuracy of the data as they do not overlap enough to question whether or not the plots fall in the correct order. The project was a perfect way to sum up the efficiency of various data structures in order to best prioritize large pools of data sets. Going forward, structures can be chosen according to necessity for the ability to quickly ro freely add data when needed, or quickly retrieve data while dequeuing.


Average Time to Dequeue
Time to Dequeue (nanoseconds)
800000
700000
600000
500000
400000
300000
200000
100000
0
0
100
200
300
400
500
600
700
800
900
1000
Number of Data Points
Heap
LL
STL


Average Time To Queue
Time to Queue (nanoseconds)
3000000
2500000
2000000
1500000
1000000
500000
0
-100
0
100
200
300
400
500
600
700
800
900
1000
Number of Data Points
Heap
LL
STL