Ryan Novak
CSCI 2270
Final Analysis
Rhonda Hoenigman
Akshit Arora

The purpose of this project is to compare the change in runtime across different structure implementations. The project uses priority queues to organize patient data in the form of a linked list, STL, and heaps. Each implementation is tested for the time it takes to enqueue and dequeue all the patients. Since computers are fast and only testing one file size once is not enough to get an accurate time difference, each implementation was tested with eight different file sizes, five hundred times each. With this large set of data, an average value and standard deviation can be calculated for each file size allowing for a better analysis of the runtimes.

The three implementations have different styles of enqueueing and dequeuing resulting in different runtimes. To read in the files, a filename is retrieved from the command line and opened. This csv file includes rows with the patients name, priority, and treatment time. Each row is read in then split into the three components and sent to the enqueue function. While reading in patient data is the same for all implementations of the priority queue, each implementation works differently.

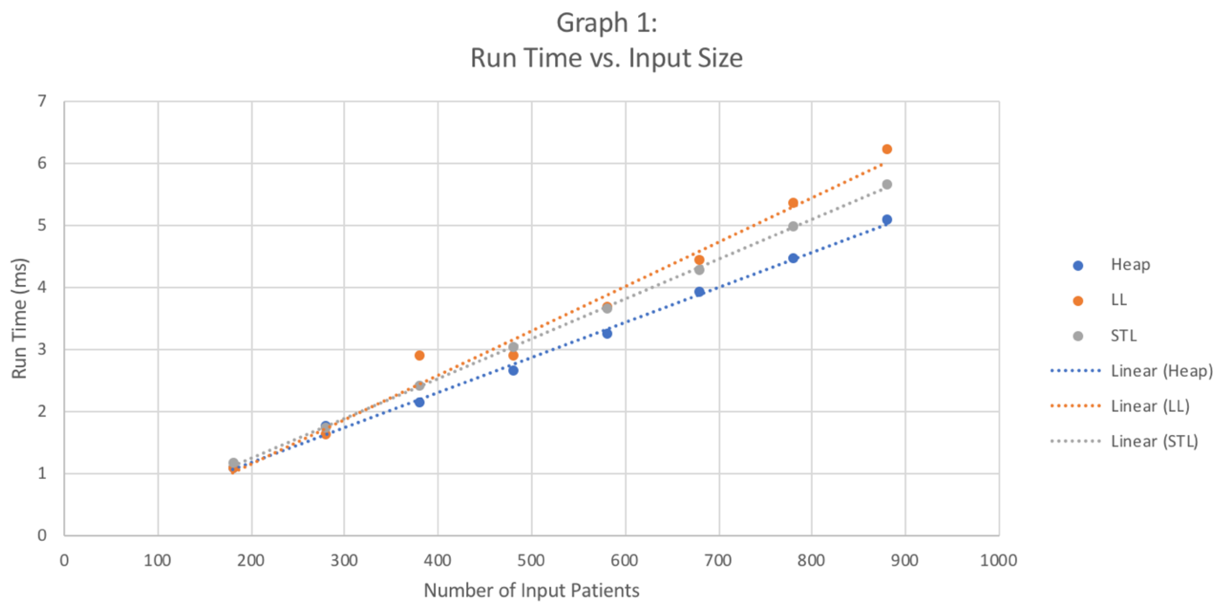
When enqueueing to the linked list, a new patient node is created to store the information read from the csv file. A temporary patient pointer node, initially set to head, is used to find the correct insertion point for the new patient based on priority. The new patients priority is compared to the pointers, moving the pointer back if the priority of the current pointer node is higher than the new patients or inserting the new patient before the pointer if not. If the priorities are equal the treatment times are compared, moving the pointer back if the treatment time of the current pointer is less than the new patients or inserting the new patient before the pointer if not. In the case where both the priority and treatment time are equal the new patient can be inserted before the pointer node. Once all the patients are added, the linked list can be dequeued. The dequeue begins with the head of the linked list by assigning a temporary pointer to the head so the head can be moved back. The temporary patients information is then printed and the node is deleted and the pointers are reassigned.

When enqueueing to the STL, a vector accepting STLqueue structures as the type can be used to store the patients data. The new patient is pushed into the vector and sorted using the compare structure. The compare structure uses the same sorting procedure as with the linked list and moves the new patient until it reaches the appropriate spot. To dequeue the STL, a while loop goes through each index popping the current highest priority patient until the vector is empty.

When enqueueing to the heap, the patient data is read in the same as the other two implementations. The new patient is added to the heap using the push_back function. This will push all the existing patients back an index and add the new patient at the front. When I tried to implement a minHeapify function, patients were not being swapped so I got output identical to the input file. Since my heapify was not working, I used #include <algorithm> to use the make_heap() function to heapify my patients once all the patients were read in and stored. This allowed me to get more accurate data for heaps than reading in input and outputting it in the same order. Using a comparison structure called with the make heap function allowed me to sort the patients by priority and treatment time, as with the other implementations. When heapified, the array is not in order from the zero index to the tail index. The parent of an index can be found at the index divided by two, the right child is the index multiplied by two plus one, and the left child is the index multiplied by two. To dequeue the heap the sort function included with the algorithm library reordered my heap and allowed it to read each index left to right, popping the patient and displaying their information.

The patient data being read in was from a provided csv file including information for 880 different patients. The read in process skipped the first row containing a description of what each column had. The first column being the patients name, the second is their priority, and the third is the treatment time. Each row was then read in one at a time. When the row is read in, it becomes a string with each element separated with a comma. The string needed to be broken into the three components and stored as a new patient using getlines on the string until a comma delimiter.

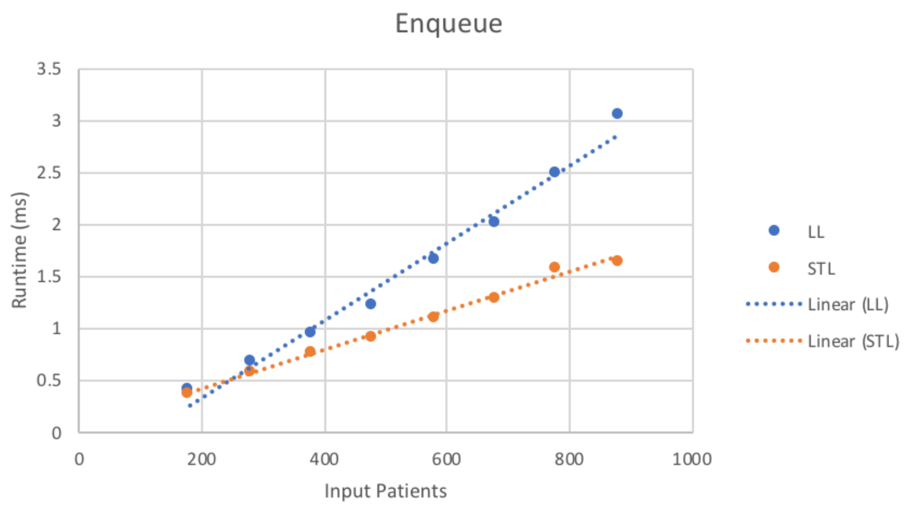
By observing **Graph 1**, the results of the total runtimes can be seen visually.

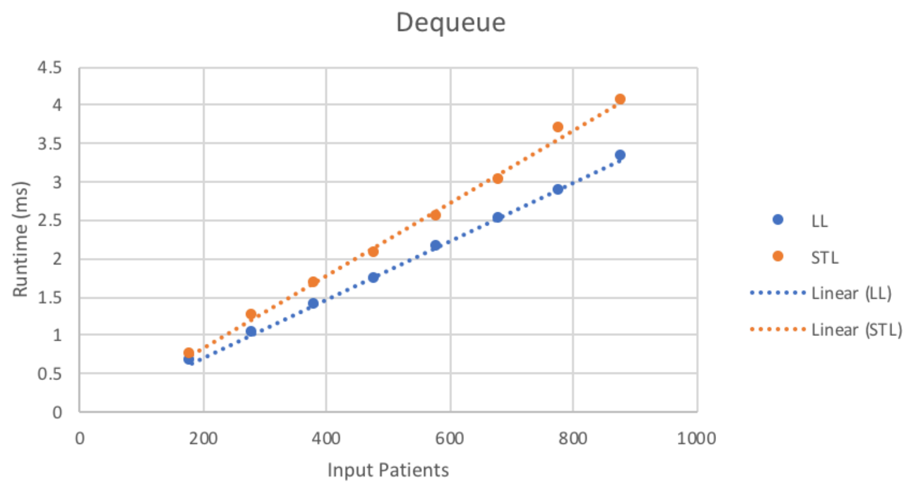


Using my implementations, heaps ran the fastest with an average runtime of 5.1 ± 0.7 ms for 880 patients. Since I used library functions to aid in the heapifying and dequeuing of the heap, my result times could have been affected. If I had dequeued by popping through the array using the parent, right child, and left child indexes, more time could have been required. The second fastest is the STL priority queue with an average runtime of 5.7 ± 0.8 ms for 880 patients. The slowest implementation is the linked list with an average runtime of 6.2 ± 0.7 ms for 880 patients.

As the graph shows, when a very small amount of input is processed there is not any noticeable run time difference between the implementations. As more input is passed to the implementations, it can be seen that they do not perform the same task at equal runtimes. This is important to keep in mind when working with large amounts of data because choosing certain implementations over another can decrease the runtime of a program. When learning about the different implementations of data structures, I did not put much thought into how the difference in runtimes between implementations could vary. This project helped me realize that thinking through what implementations I use in programs can decrease their runtime and increase their efficiency.

While looking at total runtimes can show how long it takes to complete both enqueueing and dequeuing from a priority queue, looking at the components separately can give more insight to the runtime of different implementations. By observing the two graphs below, it can be seen that while total runtimes say that linked lists are slower than STLs, linked list can dequeue faster than STL. To enqueue an entire linked list will take longer than enqueueing an STL but when they are dequeued, the linked list will finish first. Linked lists will enqueue slower than STL because every time a patient is added, the enqueue function will go through the entire linked list to find the correct insertion point. Linked lists dequeue faster because they start at the head and dequeue from left to right, resulting in a quick dequeue that does not require sorting.


Enqueue
Runtime (ms)
Input Patients
LL
STL
Linear (LL)
Linear (STL)


Dequeue
Runtime (ms)
Input Patients
LL
STL
Linear (LL)
Linear (STL)