Jared Gorthy

## Purpose:

The purpose of this project is to create several different implementations to solve one single problem. This is something we'll have to do in industry. Often times, the first data structure that comes to mind, isn't the most efficient. Our employers will want us to create several different solutions to a problem, and then use whichever is most efficient. In this case, we are creating a priority queue for admitted hospital patients. We first determine their priority, setting the lowest number priority as the patient that is admitted first. If they have the same priority, we then compare treatment times, and assign the lowest treatment time first. We use three different data structures to create this queue, and then determine which is the most efficient. This teaches us how to time our functions and compare runtimes!

## Procedure:

The three data structures we're using are a Linked List, a minheap, and an implementation from the STL. Note that for every function, I begin by reading in the .csv file and creating nodes for each of the patients. For my linked list, I used a singly linked list. In hindsight, I may have wanted to use a doubly linked list, however I was able to get it to work. Some of my helper functions included an enqueue, dequeue, and printQueue. When I enqueue a node, I insert the node by comparing the next and previous nodes. I have special cases set aside for when the queue is empty, when it only consists of a head, and when the queue is full. The dequeue takes off of the head (therefore the head has the highest priority). For the minheap, I had helper functions swap, Heapify, pop, push, and of course the constructors and destructors for my class. I used my push function to insert the nodes, based off of the code given in lecture. Then to dequeue, I used my pop function. After the "root" node is popped off, we have to call Heapify to restore the tree qualities. This Heapify and my push function call a swap function, which can be used to switch the places of two nodes in the array. The STL implementation was pretty straightforward, except for the bool operator in the class. Using the code given in class, I was able to use it to create the proper output. The functions in this library take care of almost everything. For the comparison of times, I kept my units in seconds!

## Data:

The data in this set is a struct with three components, a name, a priority, and a treatment time (string, int, int). We prioritize priority first, where the lowest integer has the highest priority. If two patients have the same priority, then we look at the treatment times. The lower the treatment times, the higher the priority! If two patients are exactly the same, then we don't care.

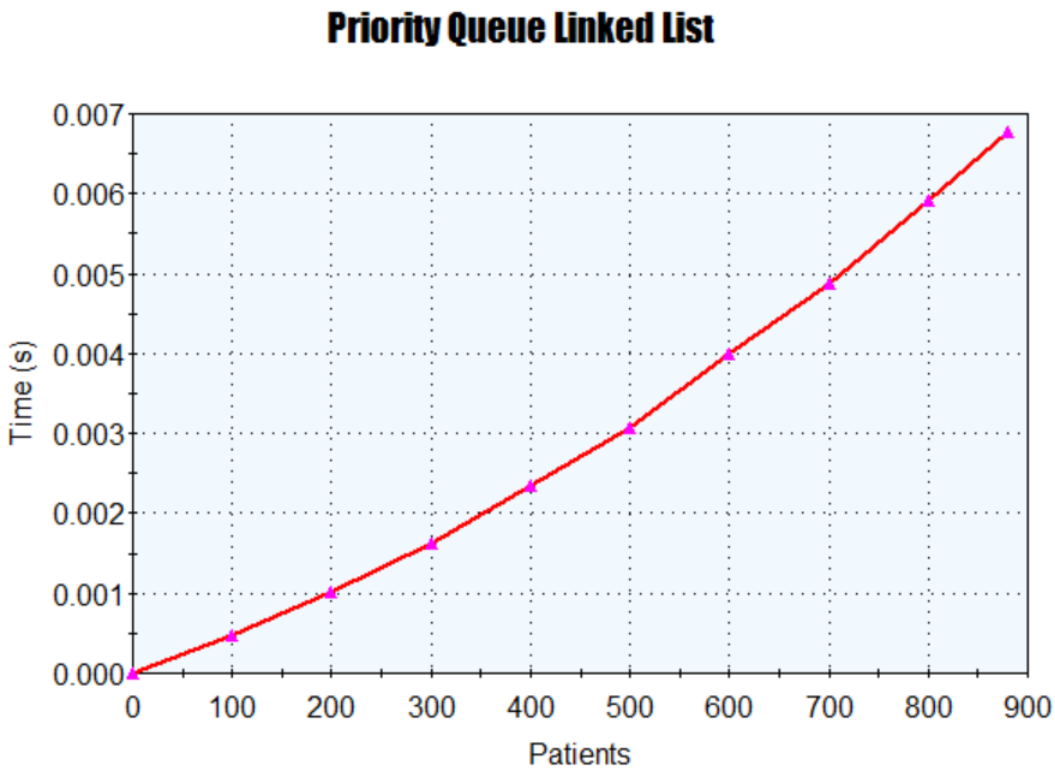
## Results:

The implementation that took the shortest time was Linked List, and then Heap and STL are nearly identical.

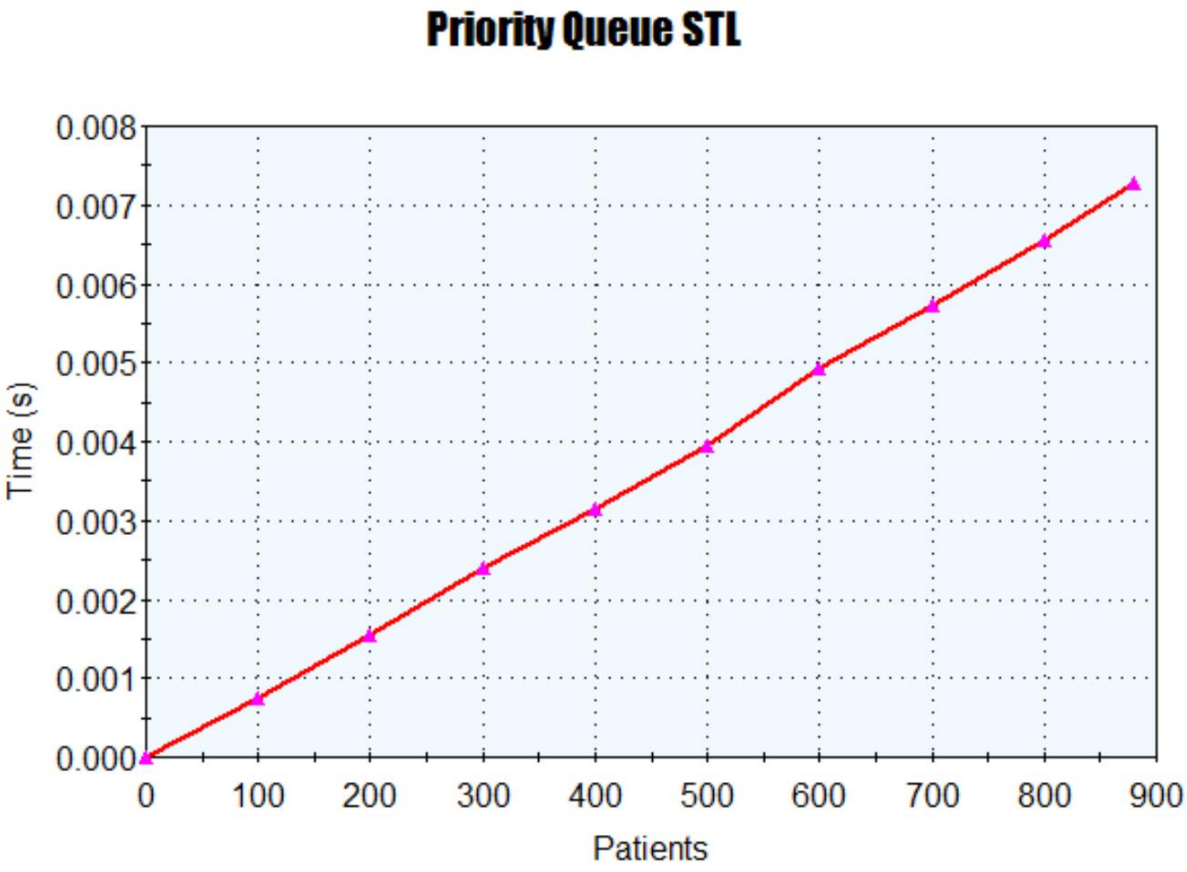
LinkedList: Mean is 0.00678 seconds, with a standard deviation of .00021 seconds
Heap: Mean is 0.00728304 seconds, with a standard deviation of .00019 seconds
STL: Mean is 0.00728304 seconds, with a standard deviation of .00027 seconds

We can also note the complexity of each implementation. For Heap enqueue and dequeue, the complexity is O(nlog(n)). Linked List enqueue is O(1), and dequeue is O(n). For the STL, it is somewhere in between, although I'm not completely sure where exactly. I came to the conclusion that Linked List is most efficient for smaller data sets, such as the ones we have in our project. Although, for smaller data sets, I would recommend using Heap or STL.

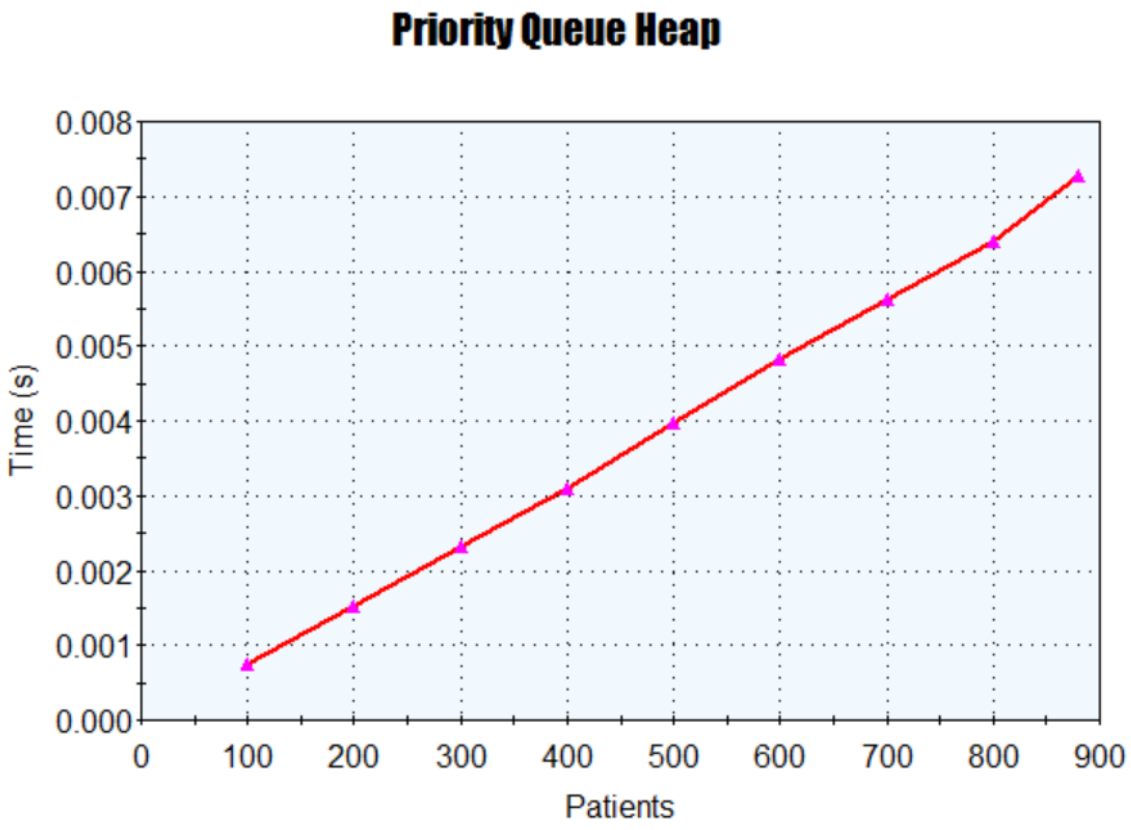
| Patients | Time (s) |
|---|---|
| 880 | 0.00678 |
| 800 | 0.0059156 |
| 700 | 0.00487382 |
| 600 | 0.0039994 |
| 500 | 0.00307558 |
| 400 | 0.00234436 |
| 300 | 0.00162565 |
| 200 | 0.00101434 |
| 100 | 0.00047637 |



| Patients | Time (s) |
|---|---|
| 880 | 0.00728612 |
| 800 | 0.00655046 |
| 700 | 0.00572358 |
| 600 | 0.00493334 |
| 500 | 0.00393748 |
| 400 | 0.00315912 |
| 300 | 0.00238996 |
| 200 | 0.00156006 |
| 100 | 0.00074896 |



| Patients | Heap (s) |
|---|---|
| 880 | 0.00728304 |
| 800 | 0.00640502 |
| 700 | 0.00562566 |
| 600 | 0.00482784 |
| 500 | 0.0039664 |
| 400 | 0.00309544 |
| 300 | 0.0023229 |
| 200 | 0.00152774 |
| 100 | 0.0007402 |



| Patients | Linked List (s) | STL (s) | Heap (s) |
|---|---|---|---|
| 880 | 0.00678 | 0.00728612 | 0.00728304 |
| 800 | 0.0059156 | 0.00655046 | 0.00640502 |
| 700 | 0.00487382 | 0.00572358 | 0.00562566 |
| 600 | 0.0039994 | 0.00493334 | 0.00482784 |
| 500 | 0.00307558 | 0.00393748 | 0.0039664 |
| 400 | 0.00234436 | 0.00315912 | 0.00309544 |
| 300 | 0.001625646 | 0.00238996 | 0.0023229 |
| 200 | 0.00101434 | 0.001560064 | 0.00152774 |
| 100 | 0.000476372 | 0.00074896 | 0.0007402 |