The purpose of this project was to understand different priority queue implementations to store the same data and compare their runtime when enqueuing and dequeuing elements. The data in this project was a list of hospital patients with a name, priority, and operation time. The priority of the patient represents the time until the patient goes into labor. The patients were ordered to have lowest priority first, and then ordered by least operation time if the patients had the same priority. The three implementations were a binary heap, linked list, and the c++ standard template library priority queue.

The binary heap priority queue uses an array to store the data while ordering the data in minimum heap property; every parent node is smaller than its child node. The binary heap insert function uses a swap helper function to keep swapping the inserted value with its greater child until the inserted value satisfies the min heap property. The dequeue function removes the root element because it is always the minimum and replaces it with the last element in the heap to keep the tree complete. The heap is then “heapified” or reordered to ensure the root is still the minimum value and the heap follows the min heap property.

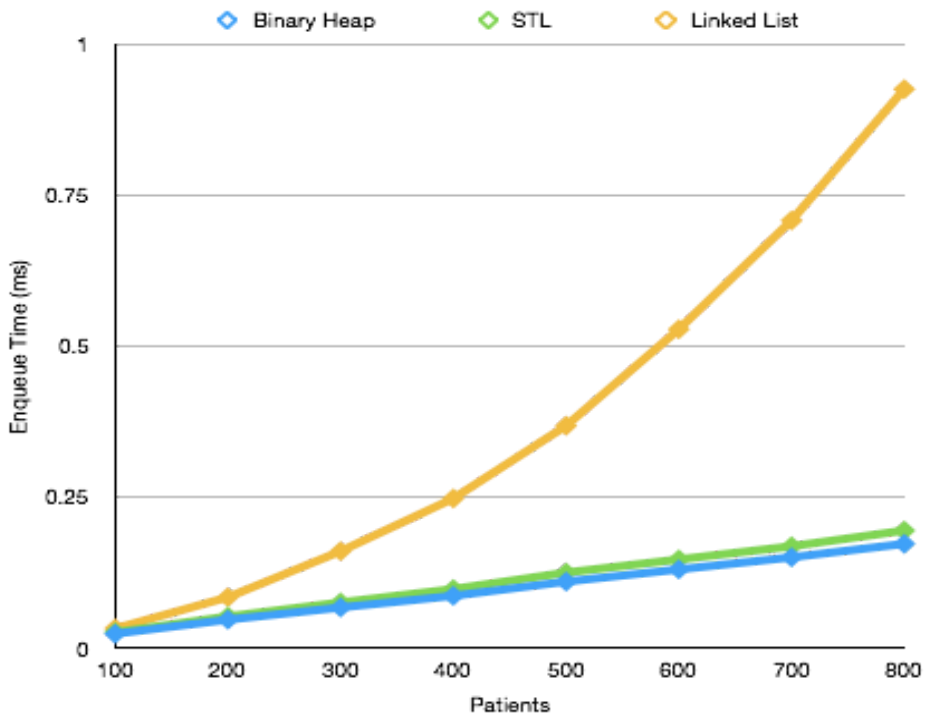
The linked list priority queue uses a patient node structure that contains the address to the next priority element. The insert function traverses the list linearly until the inserted element is correctly ordered by its priority. The delete minimum function is very simple because the minimum is always stored at the head of the list so the function reassigns the head of the list to the second element and deletes the first element.

The standard template priority queue was implemented by creating a comparator struct and vector of the elements being prioritized. We do not necessarily know how the queue works because its source code is not available, but we can compare its runtime to the other implementations of the priority queue.

To test each implantation, the data was read into a array for quick access. A random section of the data with a specific size (100, 200… 800) was then copied into a new array. The different sized and random arrays were then enqueued and dequeued with the three priority queues. To calculate the runtime for enqueuing and dequeuing, the time in milliseconds was recorded before and after the loop with the function call and then stored in a matrix. The mean and standard deviations were then calculated for each queue and respective functions.

The charts and graphs for the mean enqueue and dequeue are given below:

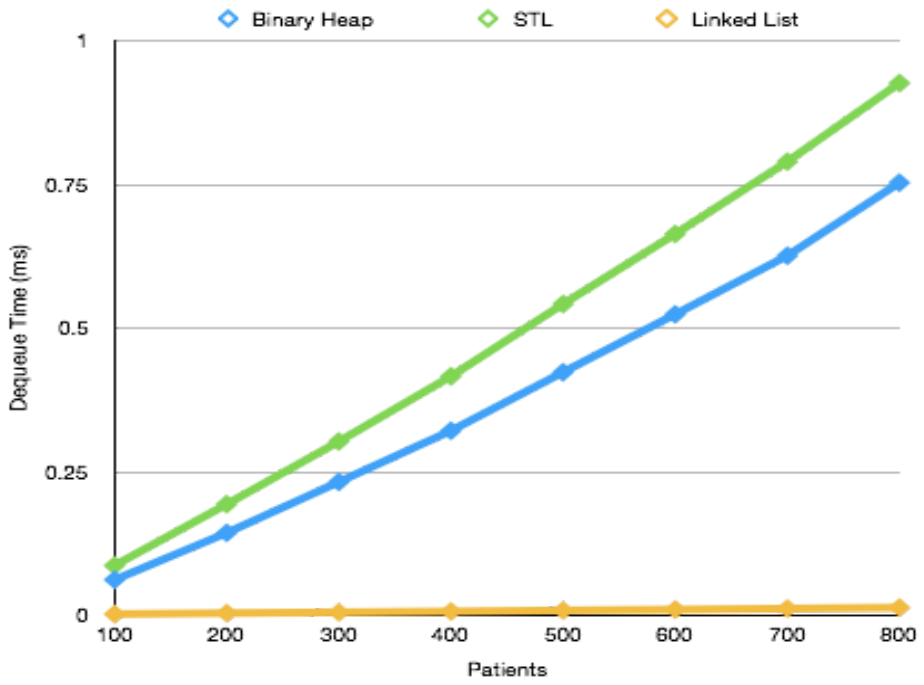
| | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 |
|---|---|---|---|---|---|---|---|---|
| Binary Heap | 0.0247 | 0.0479875 | 0.068106 | 0.0873744 | 0.110931 | 0.131088 | 0.151074 | 0.173358 |
| STL | 0.027212 | 0.0532965 | 0.0764466 | 0.0985369 | 0.125835 | 0.14758 | 0.169313 | 0.195296 |
| Linked List | 0.033414 | 0.0850367 | 0.16115 | 0.24837 | 0.36856 | 0.528127 | 0.707913 | 0.925047 |




For enqueuing a given number of elements, binary heap out performed both the STL and linked list priority queues. As the number of elements being enqueued increased, the binary heap continued to distance itself from the other two queues. The linked list was clearly the slowest and

was limited by the number of elements because the queue has to linearly traverse through the elements. The standard deviation for enqueuing the binary heap, STL, and linked list tested at 800 elements were 0.0270653, 0.0344443, and 0.123068 respectively.

| | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 |
|---|---|---|---|---|---|---|---|---|
| Binary Heap | 0.0620421 | 0.143686 | 0.232271 | 0.321426 | 0.423305 | 0.524132 | 0.626061 | 0.753142 |
| STL | 0.086928 | 0.193512 | 0.302721 | 0.416252 | 0.54199 | 0.66388 | 0.789826 | 0.926602 |
| Linked List | 0.002302 | 0.0040526 | 0.00541012 | 0.00693883 | 0.00867985 | 0.0102234 | 0.0119005 | 0.0137038 |




As for dequeuing, the linked list becomes much faster than both of the other queues because the minimum is always stored at the beginning of the list and does not have to be reordered after dequeuing. The binary heap is slowed down by the necessary heapify method because the heap typically has to be reordered after the removal of the root node. The standard

deviation for dequeuing the binary heap, STL, and linked list tested at 800 elements were 0.763518, 0.976262, and 0.0460677 respectively.

If I were in the situation stated in the project, I would personally implement the STL priority queue because it took the least amount of time to set up even though the binary heap was the fastest to enqueue and the linked list was fastest to dequeue. If I were implementing a larger scale project with a much greater number of elements, I would chose the implementation with the linked list because it is faster overall for both enqueuing and dequeuing added together.