Priority Queues: An analysis of efficiency using Linked List, STL, and Binary Heap implementations

By Jacob Christiansen
University of Colorado at Boulder
CSCI 2270, Spring 2018

Purpose:

The purpose of this project was to analyze the efficiencies when using a minimum priority queue with three different implementations. Based on these analyses, we can determine which method would be most efficient, depending on how many objects that needed to be enqueued at a given time.

Procedure:

The three implementations that were used included Singly Linked List, Minimum Binary Heap, and the C++ Standard Template Library's built in priority queue functionality. For each implementation, 880 Patients (which is the name of the Node structs being used) needed to be enqueued, sorted, and dequeued. The priority queues that were required to be created were special because they consisted not of only integers, but rather "*Patient*" structs which had [at least] three variables stored within each, *name*, *priority*, and *treatment*. The nature of which the minimum priority queues needed to store data, was lowest *priority* first, and in cases of equilivlant priorities, lowest *treatment* first. There were no cases of equilivlant *priority* and *treatment*. In the case of this project, lower *priority / treatment* meant that a *Patient* would be closer to the top of the queue, and would be dequeued before others with higher variable values.

The processes of enqueuing and dequeuing *Patients* was different for each implementation. However, all functioned in the same format. This format is as follows:

Enqueuing:

- Read in line of data from a file.
- Build a new *Patient*.
- Send said *Patient* to the *enqueue()* function.
- Sort the *Patient* into its correct position in the queue using *enqueue()*.
- Repeat until there is no more data to read in.

Dequeuing

- Run the *dequeue()* function.
- *dequeue()* removes the *Patient* at the top of the queue.
- *dequeue()* adjusts the queue to compensate for a removed *Patient*.
- If dequeuing all, repeat this process until the queue is empty.

The part that differs for each implementation, is the sorting in the *enqueue()* function and the adjusting in the *dequeue()* function.

For Linked Lists, *enqueue()* sorts by checking if a new *head* or *tail* need to be created, or, searching the queue from *head* to *tail* and adding a *Patient* if a matching *priority* doesn't exist, or, if there is a matching *priority*, search through all matching priorities until a proper position can be determined from the *treatment*. As for *dequeue()*, the *Patient* after *head* becomes the new *head*, and the old *head* is removed.

For STL, *enqueue()* sorts using another struct called *Compare*. *Compare* has a built in *bool operator()* that compares *priority* and *treatment* between two *Patients*, and returns either true or false based on it's comparison. This *Compare* is simply built into the STL *priority_queue* after it was created as follows: *priority_queue<Patient*, *vector<Patient>*, *Compare>* [name]. After this, enqueuing is as simple as just running a [name].*push(Patient)* function. As for *dequeue()*, nothing special is required beyond running a [name].*pop()* function.

For Binary Heaps, *enqueue()* sorts similarly to STL, however the *Compare* struct's code instead is implemented directly into the *Patient* struct in the header file, so you can directly compare two *Patients* using bool operators when sorting. Inside the *enqueue()* function, it addresses the current size of the heap, and inserts the *Patient* at the bottom of the heap, sifting it up until it's in its correct position. As for *dequeue()*, the *root*, or *Patient* at index 1, of the heap is removed, and *dequeue()* calls upon another function called *heapify()*. *heapify()* handles the resorting of the heap, similarly to how *enqueue()* uses sifting, but instead uses the *leftChild* and *rightChild* algorithms, 2*i, 2*i +1, to identify positions in the heap array. *dequeue()* also addresses the current size of the heap.

When doing runtime analysis, these functions enqueuing and dequeuing of *x* number of *Patients* was measured in seconds.

Data:

The enqueuing and dequeuing processes for each implementation were performed with different amounts of *Patients* in 100 increments, up until 800, with an additional amount of 880. At each of these differing amounts, the test was run 500 times, and the average time between all 500 was taken, in seconds. Examples of how data was recorded are included in the following section.

Results:

The following is the recorded dataset from the runtime analysis:

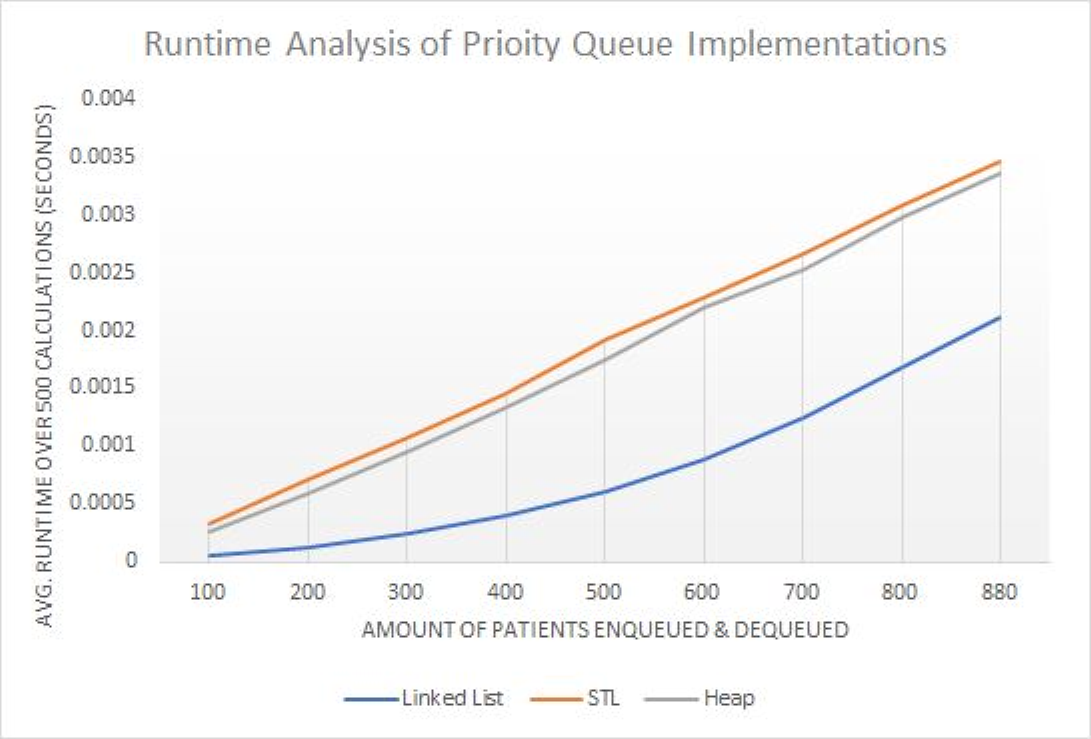
Amount of Patients Enqueued/Dequeued

	100	200	300	400	500	600	700	800	880
Linked List	0.000054 seconds	0.000120 seconds	0.000246 seconds	0.000408 seconds	0.000610 seconds	0.000890 seconds	0.001246 seconds	0.001676 seconds	0.002116 seconds
STL	0.000334 seconds	0.000706 seconds	0.001068 seconds	0.001458 seconds	0.001924 seconds	0.002278 seconds	0.002672 seconds	0.003076 seconds	0.003456 seconds
Heap	0.000266 seconds	0.000590 seconds	0.000948 seconds	0.001338 seconds	0.001748 seconds	0.002196 seconds	0.002526 seconds	0.002978 seconds	0.003354 seconds

Implementation

From left to right, is the average runtime for 500 tests on *x* number of *Patients* enqueued and dequeued, while from top to bottom is the different implementations used.

The following is a graph of the above data:



When analysing the runtimes of these three implementations, output menus were written to help with testing. This allowed the user to enter in the amount of *Patients* they wanted to enqueue and dequeue, and then the program would run that test 500 times, average all of those times, and then output the average in seconds.

Finally, we can determine based on the dataset that STL and Heap priority queues are linear with increasing amounts of *Patients*, while Linked List priority queues are exponential. With a high enough number of *Patients*, a Linked List would be the least efficient, while Heap would be the quickest. But for smaller amounts of *Patients,* Linked List will be the most efficient.

These programs were written by Jacob Christiansen, student at the University of Colorado at Boulder, using the Sublime IDE and the C++ programming language, and the tests were performed on a Razer™ Blade with a Intel® Core™ i7-7700HQ @ 2.80GHz CPU, 16GB of DDR4 RAM @ 2,400MHz, 512GB SSD (PCIe M.2), running Windows® 10 Home 64-bit.