Final Project

Muntaha Pasha

**Purpose**

The purpose of this project was to run time analysis on 3 different data structures to determine in what cases one was better in building, and in what cases one was better at deleting. It was a comparison to see what data structure was better for what kinds of data by using the mean run times for building and deleting different sizes of data. By using 3 different implementations on a variety of data sets and recording average run times, I was able to determine which implementation worked better on small data sets, which implementation worked better on large data sets, and what implementations worked somewhere in between the two. To me, the purpose really was to have sufficient data in order to prove a hypothesis about the implementations of 3 different data structures. It was to be able to back up the notes we read in lecture where we were told heaps worked best with this kind of data, or linked lists worked best with other kinds of data. Doing this project was a way to give proof to those ideas, and it was a way to gain more knowledge on the performance and Big Oh Notation of the 3 implementations.

**Procedure**

The data structures used for this project were a binary min heap, a linked list implementation of a priority queue, and a priority queue implemented from the standard template library.

* Understanding the min heap 🡪 The min heap is an O(log\_n), which means that it is the fastest of all other big oh notations to build. It stores all data in the form similar to a binary tree, where it stores the parent, and children nodes, with the root being the smallest value. Off the root will be values greater than it, and the children will have bigger values than their parents. You can not traverse/build a min heap in order necessarily, rather you can only delete in order. When you are deleting, you call min-heappify, and as you delete a value, it will print from least to greatest (because it’s a min heap, max heap would work the opposite way). It’s important to remember that a min heap can not be built in order, and it can only be deleted from in order.
* Understanding the Linked List Implementation of priority queue 🡪 A linked list is an O(n) for building, because when you are building a linked list, you are building it linearly. For example, if you give it 10 arguments, it will take 10 times longer to build than if you just pass in 1 argument. The head of the linked list will be the smallest value, because we are putting the minimum at the front, and then it will delete in order of FIFO, with a complexity of O(1). First in, first out. Deletion is quick and simple in a linked list.
* Understanding the priority queue STL implementation 🡪 It uses a O(log\_n) build, and follows a queue-implementation taken from the standard template library. It processes the priorities first, and orders them in our case from least to greatest. They can be implemented with any other structures, (as we did with binary heaps and linked lists) and this one works the way a queue does. The highest priority patient will be the first one out.

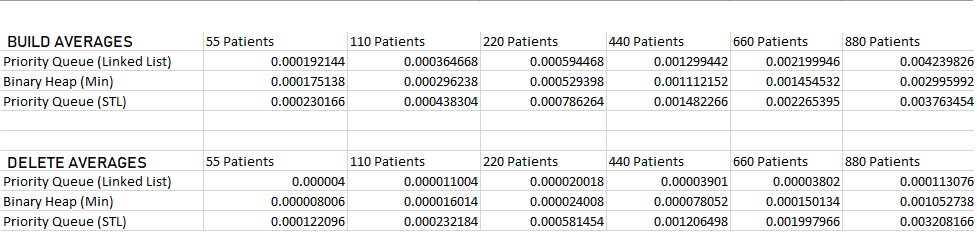
In order to measure the run times, I used chrono, which does the time in milliseconds. I was unable to use clocks because of my operating system, but chronos got the correct precision that I needed. Along with that, I used files of different sizes to compare run times. I used the regular 880 patients given, but then I depleted some each time, performing tests on 660 patients, then 440, then 220, 110, and finally 55 patients. I got build and delete mean times for each of the data structures, and then charted them, showing a very interesting trend. By using different data sizes, I was able to observe which implementation worked best where.

**Data**

The data used in this project was a list of 880 pregnant women who all want to be admitted to a hospital because their due dates to have their babies are coming up. They are given names, priority numbers, and treatment numbers. The priority number is the time until the baby is born (in minutes), and the treatment time is the approximate time the patient will be under the care and supervision of a doctor (also in minutes). Patients with lower priority numbers means that they are due much sooner, and therefore will be at the top of the list. The first thing checked is the priority, in all cases. If two women have the same priority number, then the treatment time is considered. If two women have a priority number of, say 45. If one woman’s treatment time is 20 and the other’s is 50, then the one with 20 will be listed first. This is how ties are broken incase any women have the same priority.

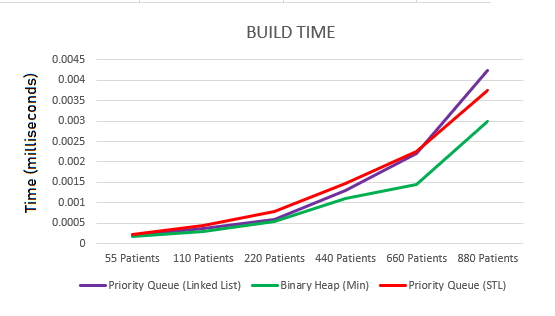
**Results**

For the data, I calculated average build and delete times, by running it 500 times for accurate time results. I outputted my data to an excel file from the code, and then created graphs based on the data I got for my different number of patients. Each implementation handled the building quite differently.

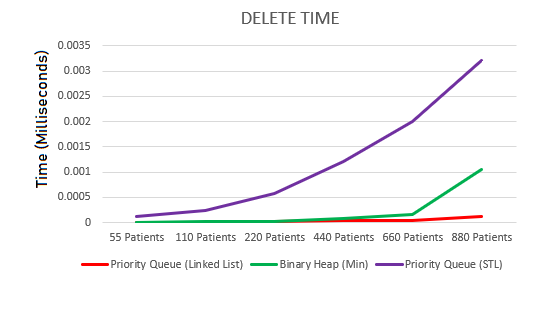


This chart shows the average times (in milliseconds) for each priority queue implementation in building and deleting a set number of patients. As the number of patients decreases, there are more and more zero’s in the times, indicating that they are performing way faster as the number goes down. I was actually given a lot more zero’s in my actual run times too, because the computer was performing in precision up to nanoseconds. For the sake of this project, I didn’t use a nanosecond converter, but that’s an option I would like to explore later on!

This first graph is made from build times of all 3 implementations and for all my different data sizes.



As I had expected, a binary heap ran the fastest build time throughout the entire test. No matter what the data size was, from 50 to a couple hundred, it performed the best out of all the others. Because an O(log\_n) complexity is a lot faster than a build of O(n) or the other ones. STL performed the next fastest, although I noticed on less data it seemed to be very close to the linked list’s performance. Given a bigger data set, and the big Oh notation however, it seemed very reasonable that the STL would perform better than the linked list because it is an O(log\_n), which is one of the fastest and more efficient big oh notations.



The fastest delete was by far the linked list. Because it’s big oh notation is an O(1), deletion is an extremely easy process. Same with the binary heap, which came second with deletion with a O(log\_n). Both of these are relatively much faster compared with the STL, which took a long time to remove any given number of patients, because of the given process time. With linked list, you’re simply removing with an O(1). It’s extremely fast, because it’s just going one by one and deleting. Same with binary heap, with the O(log\_n) it’s just removing things (in sorted order) very quickly because of the way the data is stored. However, with the STL the deletion process takes way longer than both of the other two.