Calvin Zikakis
4 May 2018
Recitation Section 201

Final Project – Priority Queue

Purpose:

The principle idea behind this final project was to test to see which implementation of a priority queue is the fastest. The project consists of 3 separate implementations of a priority queue. We were given a objective to create a priority queue for pregnant women about to go into labor at a beauty pageant and in-order to decide how each different data type preforms, we were told to time different implementations.

Procedure and Data:

All implantations share the same priority ranking system. First the main priority is tested, if the priority doesn't already exist in the queue then we simply add the new one in at the right location. If there are 2 people with identical priority, then we move to the secondary priority and test those. In this case, treatment time was tested as the secondary priority and compared when ever the first priorities were equal. Whom ever had the lowest treatment time would go before a person of the same priority.

My main consisted of a while loop with a timing aspect. The while loop looped over 3 sections of functions dedicated to starting a timer, building an implantation, dequeuing an implantation, then stopping and storing the time before moving onto the next implementation to repeat the same thing. This main while loop allowed me to run lots of iterations of building, queuing, and dequeuing each implementation over and over again. To time each implantation, I used the "chrono", "ratio", and "ctime" which all enabled me to double floating points of how long each implementation took to run. These timing values would then be exported to a CSV file allowing me to easily import the data to excel to calculate averages, standard deviations, and create graphs. Below we will loop at how each implantation worked.

Implementation 1: A STL Priority Queue

For this implementation we were tasked with creating a priority queue by using a built in library. This implementation was by far the easiest to create. To create this data structure, all I had to do was import queue and create a new priority queue. Then this was a matter of reading out of the text file and then pushing each person into the queue. The queue would automatically identify which had a lower priority and store them in that order. If 2 priorities were equal, I wrote a compare struct that was passed in as an argument to the priority queue. The priority queue could then automatically check to see which one of the treatment times was lower when the 2 priorities were the same.

Implementation 2: A Linked List Priority Queue

This implantation used a linked list data structure to store the queue. I decided on using a doubly-linked list for easier navigation through the list. This list consisted of nodes which contained the patient name, priority, treatment time, and pointers to the next and previous nodes. The class file contained a pointer to the head of the linked list for easy popping of data. In-order to make the logic of this implantation easy, I created a helper function that would compare 2 patients and return number values depending on if the new patient belongs before the compared one. This massively reduced the

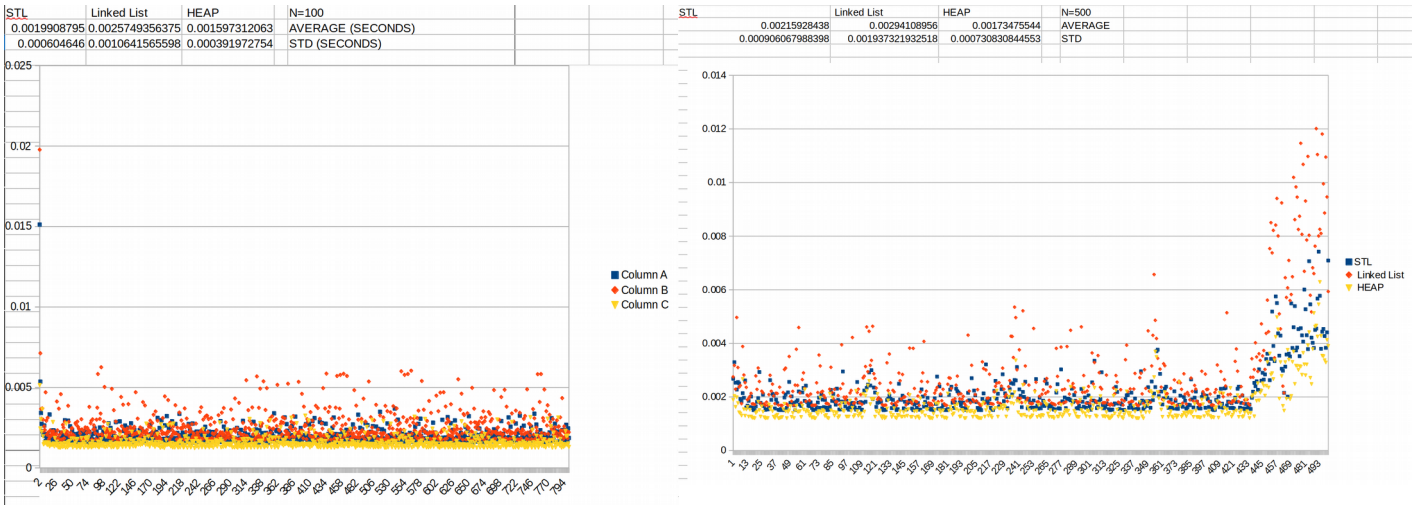
size of the push function making this code easier to follow and understand. The push function is the heart of the priority queue, it takes a name, priority, and treatment time which it creates a node with. It is responsible for adding a new patient into the list in the right spot using the compare function. The pop function simply removes the head of the linked list because the list is already in order and top simply returns the head of the linked list.

Implementation 3: A Binary Heap Priority Queue

The final implantation makes use of a binary heap to store the priority. In this case a min-heap was used to always make sure the item with the lowest priority score is the root of the heap. The improvement of the binary heap over the linked list is it supports insert and delete functions in O(log(n)). This was the most complicated algorithm because the binary heap needs to be able to balance it self in a way. The main functions used in the binary heap were swap (swaps 2 nodes), pop (pops the highest priority item off the queue and returns it), push (adds a new node to the heap), and minHeapify (rearranges the heap to maintain properties of the heap).

All run times of this project were measured in the same method. Each measured time consisted of reading from a text file, constructing the queue, queuing every item, popping every item, and finally closing the text file before moving to the next implementation.

Results:
(Blue = STL, Red = Linked List, Yellow = Heap)



As the data above shows for 800 iterations, the heap had the lowest average speed at 0.0015973 seconds with a standard deviation of 0.0003919 seconds. The STL implementation followed behind with an average of 0.0019908 seconds and a standard deviation of 0.000604 seconds. The linked list was terrible with a time of 0.00257 seconds and a standard deviation of 0.001064 seconds. This data was all conclusive with my tests of n=50, n=100, n=200, n=300, n=400, n=500, n=600, n=700, and n=800. As you can see in the figure on the right of n=500, there was some sort of systematic error when running the process on 500 lines that can be seen by the best fit line of the data increasing off to the right. In conclusion, the heap was the fastest data type only slightly ahead of the STL. This is due to the heap having a larger big o value than the linked list.