CSCI 2270 Final Project Write-up
Bryce DesBrisay
Prof. Hoenigman
TA Rohit Mehra
Interview at 3pm on Friday, May 4th, 2018

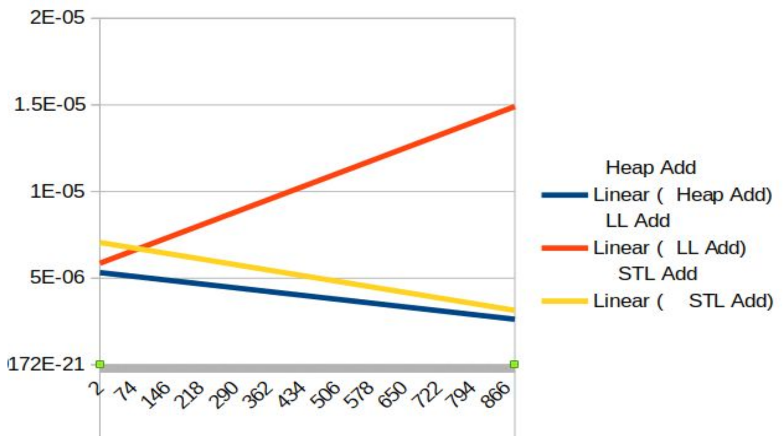
I am the only programmer at the hospital that I work at and am tasked with building a priority queue to help identify which patients are high priority and need to be helped first. In an unexpected turn of events all the women at the hospital go into labor and need to be sorted immediately by the time until they start giving birth and if needed, also by the total time for treatment. I need to make a priority queue in three different ways and then compare them to each other. Once each method is implemented, I am to evaluate the runtime of both adding and removing items from the priority queue.

The data structures that I will implement as a priority queue are a minimum binary heap, a linked list, and the built in Standard Template Library version of a priority queue. A binary heap is essentially a tree that can be represented by an array. This is because each child of a certain node occupies a position in the array relative to its parents position. A binary heap is either a min heap or a max heap, but since the values used to decide the order are time sensitive, I used a min heap. All this means is that the root of the binary heap is the minimum value of all the values in the heap. Another general property of a min heap is that each child is of greater value than its parent and a lesser value than its children, making it easy to restructure the heap after the minimum value has been removed. The function used to restructure the heap after any change is called MinHeapify in my program. The MinHeapify function takes an index, typically 0 to start, and then checks if any of its children are smaller, and if one of them is then it swaps the two and then recursively calls itself to make any further corrections in the rest of the tree. To remove the smallest item from the heap, you just need to remove the head of the heap and then restructure it with MinHeapify. Adding a node uses the same general principle of the MinHeapify function in the way that it inserts the new node at the bottom of the heap but then needs to correct any leg of the tree that doesn't follow the rules of a min heap. The next implementation used is a linked list. A linked list consists of nodes containing a variable linking itself to the the next node in the list. Adding a node to the list is relatively easy compared to the heap. All you have to do it run through the list until you find a node with a lower priority than the new node and then insert it before that node. Deleting the minimum node is extremely easy because all you have to do is set the head equal to its next value. The last implementation I used is the built in STL priority queue. This one is really easy to use because it has built in functions to add and delete items; all you need to do is either pq.push(item) to add and item or pq.pop() to remove the most urgent item. Although the built in priority queue is a max queue by default I needed to make my own compare function for it to use which turns it into a min queue. In order to measure the different runtimes of each part of the data structures I will use the clock library and measure the seconds it takes to complete each step. I will then pipe all this timed data into a .csv file and create a graph with it in excel (graph at the end).

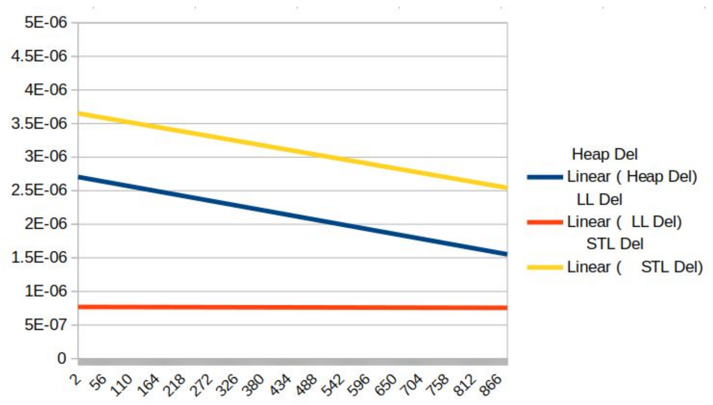
The data being used in this project is a .csv file containing a list of patients names, their time till birth, and the treatment time for each one. As each line of the file is read into my program I add each patient as an item to the priority queues. The metric of priority I use is the time till the patients baby is born, the sooner it is, the higher priority. If there is ever a tie in the time till birth then the tie breaker is the treatment time, a shorter treatment time is higher priority so that the hospital can reallocate its doctors to new patients faster.

As the number of items in the heap increased, the time of addition of a new item decreased, meaning it got quicker to add an item when there were more items already in the heap (blue line in addition chart). As the number of items increased in the linked list, adding an item took longer than it did with less. This can be seen as the red line in the addition chart where it slopes upwards with the number of items. Similar to the binary heap, as the number of items increases in the stl priority queue, the time to add another item decreases and is shown by the yellow line in the addition chart. Out of all three of the insertion functions for the priority queues, the binary heap was the fastest to start and to end (avg: 4E-6, sd: 1E-6), with the stl priority queue following closely (avg: 5E-6, sd: 6E-6), and the linked list starting out near the others but then drifting away and taking longer than the others when faced with more items (avg: 1E-5, sd: 7E-6). For the deletion functions, all three ways except the linked list seemed to get quicker as they deleted more items, thus making the queue smaller and speeding up the process. The linked list deletion was different only because it is constant. This is because the smallest item in the linked list is always the first, so all you need to do it remove the head of the list. Because of this, the linked list was the fastest implementation for deleting an item with an average time of 7E-7 with a standard deviation of 4E-7. Second was the binary heap averaging 2E-6 with a standard deviation of 8E-7 and a close third was the stl priority queue averaging 3E-6 with a standard deviation of 1E-6.

Addition Chart:



Deletion Chart (next page):


5E-06
4.5E-06
4E-06
3.5E-06
3E-06
2.5E-06
2E-06
1.5E-06
1E-06
5E-07
0
2
56
110
164
218
272
326
380
434
488
542
596
650
704
758
812
866
Heap Del
Linear (Heap Del)
LL Del
Linear (LL Del)
STL Del
Linear (STL Del)