Munkhsanaa Otgonbayar

Purpose

The Purpose of our project was to implement three data structures. The standard library version of priority queue, linked lists, and also min heaps. We were given a csv file with inputs that we had to categorized by the priority of the patient and if the priorities were the same we had to compare the treatment times of the patients and categorize the patients with the highest priority first and quickest treatment time first. After we built our data structures we compared the runtimes in which we enqueued and dequeue the data. We compared which one of the data structures were quicker. We know that the STL priority queue was the fastest. The insert has an complexity of O(N) and to sort them we have a complexity of O(N log N). For the heap we know that building a heap is O(N) and the worst case for heapify is O(log n) therefore a total time of O(n log n). For a linked list, doubly-linked list in specific, the time complexity is O(n). The task for us was to implement a priority queue with a linked list and a heap. We had the task to see if the implementations were really what the time complexities said they were.

Procedure

We are using priority queues from the STL and implementing linked lists to act like a priority queue and also min heap. We will be using milliseconds to determine the difference between the implementations. The STL version of priority queues are implemented just like a regular queue but in the main.cpp file I included a bool operator that compared the data that was given and rightfully compared the priorities with each other and if they were equal the function would compare the treatments. The min-heap function works as though it is a binary tree represented as an array. When we enqueue the data into a heap we insert the element to the last

place of the array and bubble sort it until it gets to the correct position. For dequeue we would remove the root and instert the largest value to the root and compare new root with the right and left children until it gets to the correct position. For linked list every time a patient was added I traversed through the linked list until I found where it had to go. I had cases in which the added element had to be before the head, somewhere in the middle or at the end.

Data

So we were given a data set that were patients in a hospital about to give birth. The patients all had a name, priority, and treatment time. We first compared the priorities of the patients and compared which of the patients had a smaller priority number and a smaller treatment time.

Result

The data for my project was not the same as the time complexities they were said to have. For my implementation of the LL version of the priority queue the enqueue, dequeue and implementation was faster than the other two structs. The heap was quicker than the STL priority queue and the STL priority queue was surprisingly the slowest. The reason why I was surprised was because when we saw an implementation in class the priority queues were the quickest and the Linked lists were the slowest. What may have caused it is that when we want to add an element into the middle STL Priority queue, we have to shift every element after the new item we added into the queue. Compared that to how the linked list adds the item, by just changing the pointers of the next and prev nodes, the linked list was much more quicker.

Munkhsanaa Otgonbayar