Jolie Klefeker
CSCI 2270: Final Project
TA: Abhidip Bhattacharyya

Priority Queue Implementation and Analysis

The purpose of this assignment is to compare the performance, in terms of runtime, of three different implementations of a priority queue: heap, standard library, and linked list. The priority queue being built for this assignment is used for medical triage in the theoretical situation of 880 pregnant women all going into labor at the same time. The priority queue must compare time until labor and, when labor times are equal, treatment time, and place these women into an queue of which the priority is defined by lowest time until labor, when needed, lowest treatment time.

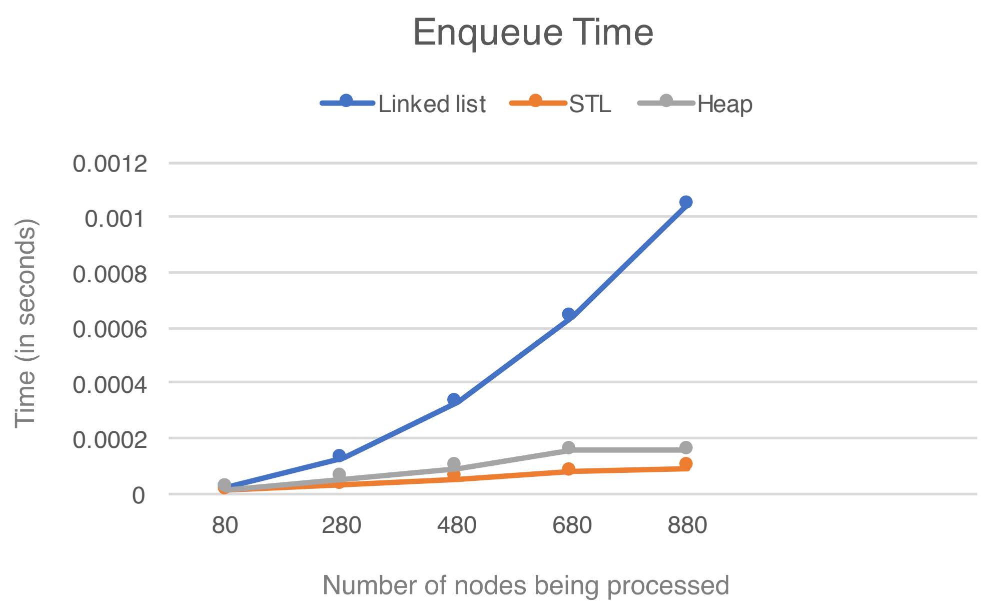
The heap implementation of a priority queue implements a Min Heap of structs, for this assignment the structs contain the name, time until labor, and treatment time for each woman. A Min Heap is a binary tree that is both complete, meaning that every level of the tree is filled with the exception of the bottom or last level of the tree. And, that the root of the tree is the minimum value of the tree. A heap is typically represented as an array and its tree structure comes from three equations that can calculate the index of the parent, left, and right nodes from the index of the node you are currently working with. As nodes are added to the heap, one must iterate through the existing heap nodes and check if the node that is being pushed's labor time value is greater than or equal to the value of its parent. If that is true one must then check if the values are equal, if they are equal one must then compare the treatment time values and accordingly insert the node into heap, either before or after the parent, by calling the swap function, if before. If the node being pushed's labor time value is greater than the parent's then it will be inserted before the parent by calling the swap function.

The standard library implementation of a priority queue uses a heap as well, it already has the push, pop, and other functions defined, and can be implemented as a max or min heap. In the case of a min heap, one has to provide the declaration of the min heap with a vector of nodes and well as a comparator node or class that returns the greater of any two node priority values. In the case of this assignment the comparator class compares the labor time values and returns the greater, except when the labor time values are equal. In this case additional if/else statements are implemented to compare the treatment time values and returns the greater of those two.

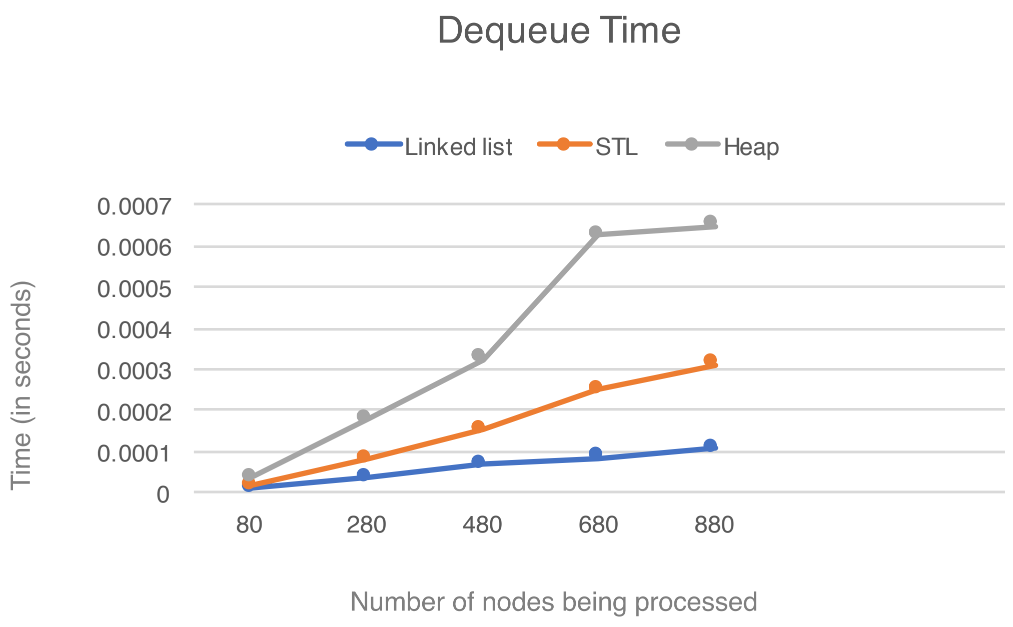
The linked list implementation of a priority queue can be implemented as a basic singly linked list. Each node in the list represents one woman and contains her priority times. Nodes can be added to list and linked together through pointers. As they are added to the list, one must iterate through the existing list and insert the new node at the appropriate place, similar to the comparison process as described in the heap implementation, by comparing the labor time values and if they are equal then additionally comparing the treatment time values.

Results

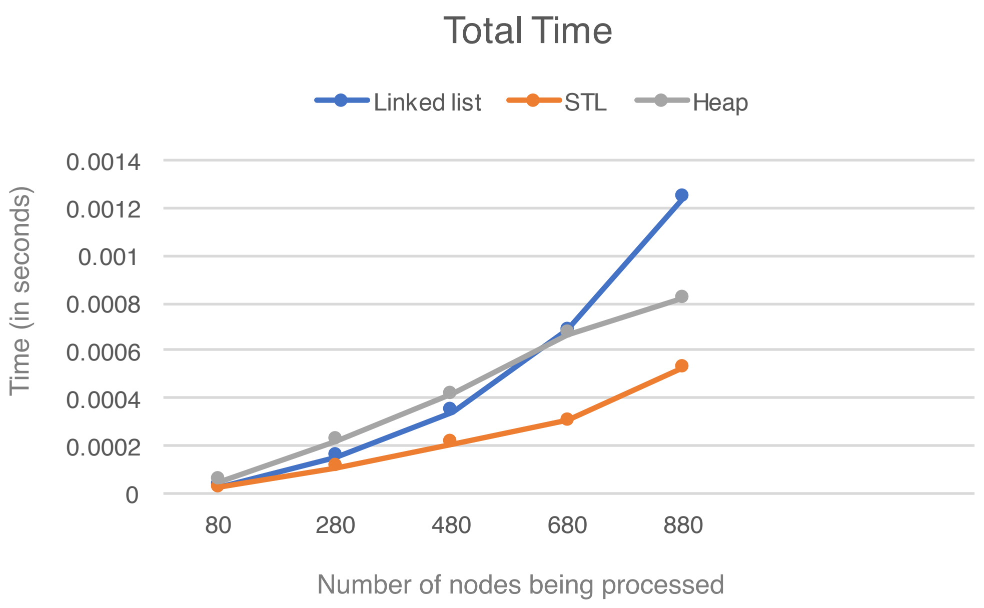
To analyze runtime, the C++ <time.h> library was used to measure the amount of time that it took for each implementation enqueue and dequeue all nodes. By using the clock function included in the time library, a beginning time was set before a for loop that iterated through an array of nodes and either pushed or popped all of the provided nodes. After the for loop was completed the end time was also measured using the clock function. The beginning time was then subtracted from the start time to calculate how long it took for that implementation to push or pop all nodes. In order to gain a more accurate insight into runtime, this process was repeated five hundred times for both push and pop (enqueue and dequeue) for each of the three implementations being compared. The amount of time it took to complete each push or pop was summed and after the process had happened five hundred times the total time sum was divided by 500 and then divided by "CLOCKS_PER_SEC" to convert times into seconds. The below graphs are the results from the runtime analysis described above.



The above graph compares the enqueue time for the three implementations completed during this project. The linked list implementation has the biggest change in time over amount of data. This is due to the structure of a linked list. In order to sort the linked list by priority, every time a node is added, or enqueued, the function has to iterate through the entirety of the existing linked list and compare the node being added with every previous node. As the list gets longer so does the amount of time it takes to enqueue a new node. For the heap and STL heap implementations the amount of time needed to enqueue a new node does not change much over time, there is only a slight increase. Because of a heap's tree like structure, the function is able to eliminate much larger chunks of data at a time during the comparison process.



This graph compares the dequeue times of the three different implementations used for this project, also please note the difference in scale. The heap and STL implementations take more time for their deletions in general, and grow faster in relation to the number of nodes. The dequeue algorithm for both of these is slightly more complicated than the insert because in addition to comparing and swapping, the dequeue method has to maintain the min heap structure of the tree by calling the min heapify function which orders the heap based off of calculated index values for parent and children. Because of this the dequeuing process takes more time. The linked list dequeue takes far less time than its enqueue function because no comparison is necessary, it simply iterates through the list through the pointers, deletes and then resets the pointers to accommodate for the delete.



As expected the linked list implementation took the most time with 880 nodes because its enqueuing method is relatively inefficient due to its need to iterate through the whole list in order to add. The heap took slightly longer than the STL heap, I assume that this is because the library was created by a professional and well thought through so, there was most likely care taken “under the hood” of the library to speed things along. The heap implementation because it was coded by me, an undergraduate, was likely clunky in some ways. But it would make sense that both heap implementations are more efficient than the Linked List because of their tree-like structure, this allows them, to “divide and conquer” eliminating larger chunks of data at a time when comparing.