Margaux McFarland
Final Project
5/6/18

Purpose

Because of the extreme amount of pregnant people being admitted into the hospital at the same time, a priority queue needs to be programmed in order to fairly process all these patients based on time until birth and treatment time. The purpose of this project was to determine and compare the different run times between different implementations of a priority queue. More specifically, the run times for building and deleting from three different implementations, a linked list priority queue, a heap priority queue, and a Standard Template Library queue, were compared.

Procedure

A CSV file of patient names, priorities, and treatment times was read into the program and then added to a linked list priority queue, a heap priority queue, and a Standard Template Library. Run times in milliseconds for building each priority queue were evaluated 500 times. Then patients were deleted from each priority queue and the run times in milliseconds of each deletion were evaluated 500 times as well. The average runtimes for each implementation build and delete were calculated as well the standard deviations.

The linked list priority queue was built by creating a node with the information from the CSV file and traversing the linked list until a node with a higher priority, or higher time until birth, was found or a node with a higher treatment time was found. Then, the new node was added before the node already in the queue. To delete from the linked list priority queue, the head pointer was changed to the head's next node because the node at the front of the list has the lowest priority, or lowest time until birth.

The heap priority queue was built with an array. A patient was pushed to the first open spot in the array and then traversed from the end of the array to the front to put the patient in its correct spot based on priority or treatment time. If the parent's priority is greater than the child's or the parent's treatment time is greater than the child's, in the case where their times until birth are equal, then the two patients are swapped in the array. This continues until the patient is in the correct spot. To pop a patient from the queue. The root, or the first element in the array, is popped and replaced with the last element in the array. Then the heap is heapified. The smallest element is found in the heap and then swapped with the root until the root becomes the smallest element.

The STL priority queue was implemented by including queue from C++ STL. A priority queue of patients was created. The compare constructor was overridden to compare priorities and treatment times if the priorities are equal. Lower priorities and lower treatment times return true which means they will be put before patients with higher priorities or higher treatment times in the queue. Then to add to the priority queue, push() was called and to delete from the priority queue, pop() was called.

Each time a priority queue was built or deleted from, runtimes were found and added to a two dimensional array. There are six elements , one for each implementation build or delete functions, hold an array of 500 runtimes. These runtimes are averaged and also used to calculate the standard deviation.

Data

Data was given in an CSV file of 881 rows including the row labeling the columns. There were 880 patient names, priorities, and treatment times. The priority of each patient was their time until giving birth, so a lower priority was prioritized over a higher priority because it meant the patient would be giving birth sooner. Treatment times were comparing when two patients' priorities were the same, and if one patient's treatment time was smaller than the other, they were prioritized and processed first.

Results

Building a linked list priority queue performed linearly as shown from the blue time in Figure 1.1. The worst case is order O(n) where n is the length of the linked list because the program could traverse the entire linked list if the node being added has the highest priority.

Building a heap priority queue performed linearly as shown from the orange line in Figure 1.1. The runtime for building the heap should be O(logn) because the patient is continually compared to its parent which is i/2. Only half of each subtree will be traversed instead of the entire tree.

Building the STL priority queue performed linearly as shown from the grey line in Figure 1.1. The worst case is order O(n) where n is the size of the vector of patients because the program could have to traverse the entire vector to put a patient with a priority lower than the rest in the right spot.

Deleting from all implementations, as shown in Figure 1.2, is constant, order O(1) because the lists are already ordered and the first element is always popped. With the priority queue heap, however, the heapify function is also performed which could make the runtime O(logn), logn is the height of the tree, because the program could continually go through half of each subtree, swapping each patient.

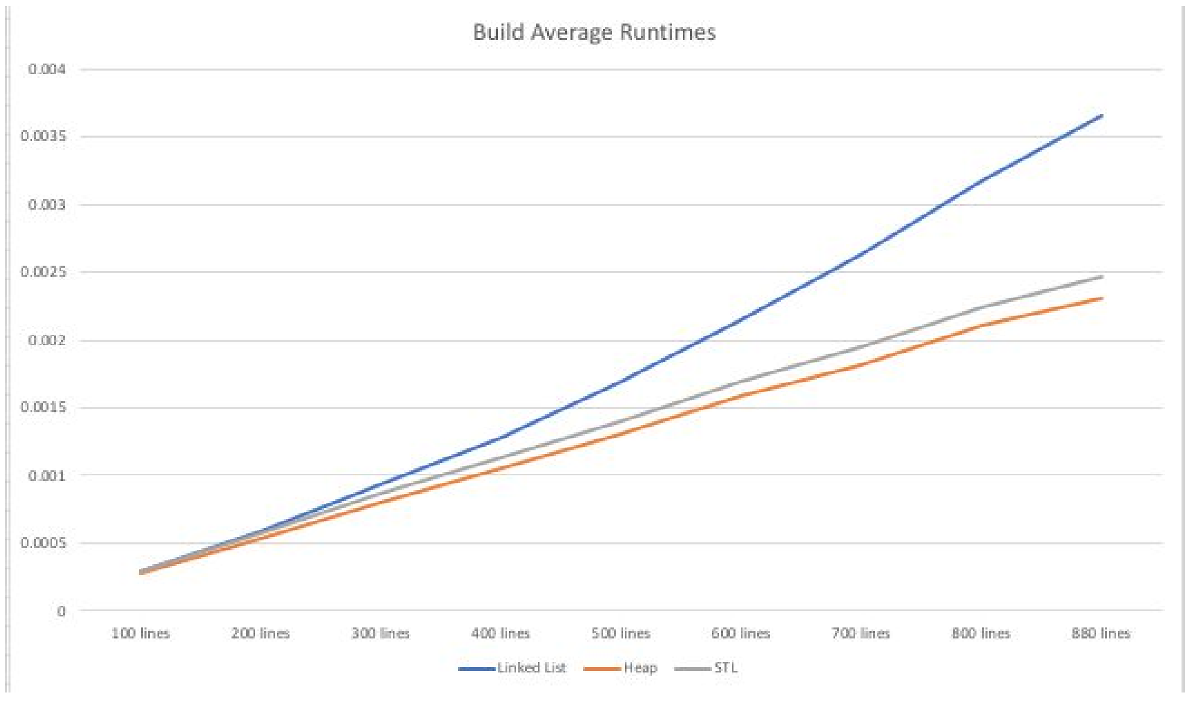

Figure 1.1. This plot shows how the runtimes for building a priority queue linked list, priority queue heap, and STL priority queue change when more lines are read in from the file.

Figure 1.2. This plot shows how the runtimes for deleting from a priority queue linked list, priority queue heap, and STL priority queue change when more lines are read in from the file.

Figure 1.3. This table shows the standard deviations for all the runtimes for building each implementation.

Figure 1.4. This table shows the standard deviations for all the runtimes for deleting elements from each implementation.

Conclusion

Overall, the heap priority queue was the most efficient in building the priority queue, and the linked list was the least efficient. The linked list, however, was the most efficient when deleting elements than the heap, the heap being the least efficient in terms of deleting elements.