CSCI 2270 Final Project Write-Up

Travis Torline

**Purpose:** The purpose of this project was to implement a priority queue in C++ using three different data structures and analyze the differences in runtime between all three implementations. Runtime analysis was performed on adding items to and deleting items from the three different priority queue implementations with different data sizes to understand which implementation is best built to be a priority queue.

**Procedure:** The three data structures being used to build a priority queue are as follows:

1. *Linked List:* A linked list is a data structure of dynamically allocated nodes in which each node contains information as well as a pointer to the next node in the list. In my implementation of the linked list, each node contains a name, a priority, and a treatment time as well as a pointer to their next node. The linked list class has functions *push, pop, and printAll.* The *push* function adds nodes to the priority queue in their correct position by traversing the list, the *pop* function deletes the element at the head of the linked list, and the *printAll* function goes from the head to the tail of the list and prints the values of the variables stored in each node.
2. *Standard Template Library Priority Queue:* In C++, certain data structures have already been built and can be included in the header of a user’s file. The STL priority queue is implemented as a maxheap of integers but can be overridden to take user-defined classes and create a minheap rather than a maxheap. I’ve done just that in order to implement a minheap in my program. All the data structure’s functions were pre-built for me to use.
3. *Minheap:* A minheap is a priority queue that resembles a binary tree but is implemented using an array. In the array starting at index 1, each parent at position *i* has a left child at *2i* and a right child at *2i + 1*. Each parent must have a smaller priority than its children, hence why this data structure is called a **min**heap. My minheap class has functions *push, pop, minHeapify, deleteHeap,* and *isEmpty,* as well as member variables *capacity* and *currentSize*. The *push* function adds elements to the priority queue in correct order, the *pop* function removes and deletes the element at index 1 of the array while utilizing *minHeapify* to correctly order the new array, the *deleteHeap* function starts at index 1 of the array and deletes all items, and the *isEmpty* function checks to see if the *currentSize* of the heap is 0.

To measure the runtime differences between these three implementations, I’ve utilized *clock* from the <ctime> library. The clock starts when the priority queue begins building/deleting and ends when the process is finished. It then calculates the difference between the times and divides by 1000 to get data in milliseconds.

To ease the process of collecting large sets of data, I’ve designed my program to run the process of building/deleting these priority queues many times in a row and then store each individual time in a vector. The vectors can be passed through functions to find the mean and standard deviations of the vectors of times. These functions were implemented using the <cmath> library.

**Data:** My program was built to read in a CSV file from the command line. A CSV – “comma separated value” - file has all its data separated by commas with a newline character at the end of every row of data.

In the case of the data being read for this project, the CSV file was full of patients, each with a name, a priority value, and a treatment time. The priority queue was built based off the priority and treatment values of each patient. Patients with a smaller priority value had a greater priority, and thus were put more towards the beginning of the priority queue. In the case that two patients each had the same priority value, the patient with the smaller treatment time received priority and was put more towards the beginning of the priority queue.

**Results:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| LINKED LIST: |  |  |  |  |  |
|  |  | **Building** |  | **Deleting** |  |
| Data Size | Trials | Mean(ms) | Standard Deviation(ms) | Mean(ms) | Standard Deviation(ms) |
| 100 | 70 | 0.04347 | 0.00499 | 0.00404 | 0.00043 |
| 200 | 70 | 0.12704 | 0.00845 | 0.00771 | 0.00056 |
| 300 | 70 | 0.21238 | 0.08684 | 0.00957 | 0.00567 |
| 400 | 70 | 0.44232 | 0.03223 | 0.0184 | 0.01453 |
| 500 | 70 | 0.70231 | 0.0667 | 0.01944 | 0.00303 |
| 600 | 70 | 1.10987 | 0.0854 | 0.02295 | 0.00195 |
| 700 | 70 | 1.66644 | 0.14019 | 0.02905 | 0.00924 |
| 800 | 70 | 2.33959 | 0.19964 | 0.03112 | 0.00341 |
| 880 | 70 | 2.80974 | 0.19401 | 0.03542 | 0.00775 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| STL: |  |  |  |  |  |
|  |  | **Building** |  | **Deleting** |  |
| Data Size | Trials | Mean(ms) | Standard Deviation(ms) | Mean(ms) | Standard Deviation(ms) |
| 100 | 70 | 0.04715 | 0.00558 | 0.09238 | 0.00418 |
| 200 | 70 | 0.09504 | 0.00932 | 0.20641 | 0.01312 |
| 300 | 70 | 0.14744 | 0.02697 | 0.34193 | 0.04018 |
| 400 | 70 | 0.18835 | 0.01704 | 0.47304 | 0.03304 |
| 500 | 70 | 0.23855 | 0.02507 | 0.61045 | 0.0324 |
| 600 | 70 | 0.28535 | 0.02851 | 0.76844 | 0.05132 |
| 700 | 70 | 0.33929 | 0.03105 | 0.91338 | 0.06104 |
| 800 | 70 | 0.37852 | 0.02616 | 1.06 | 0.05752 |
| 880 | 70 | 0.41381 | 0.02759 | 1.18259 | 0.05491 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| MINHEAP: |  |  |  |  |  |
|  |  | **Building** |  | **Deleting** |  |
| Data Size | Trials | Mean(ms) | Standard Deviation(ms) | Mean (ms) | Standard Deviation(ms) |
| 100 | 70 | 0.0168 | 0.00469 | 0.02575 | 0.02083 |
| 200 | 70 | 0.03208 | 0.00941 | 0.05087 | 0.02020 |
| 300 | 70 | 0.05082 | 0.01792 | 0.09848 | 0.00462 |
| 400 | 70 | 0.06657 | 0.02132 | 0.14815 | 0.02004 |
| 500 | 70 | 0.08465 | 0.01834 | 0.1884 | 0.01758 |
| 600 | 70 | 0.10621 | 0.04056 | 0.22497 | 0.04539 |
| 700 | 70 | 0.10985 | 0.02034 | 0.23731 | 0.08933 |
| 800 | 70 | 0.13265 | 0.02617 | 0.27942 | 0.07349 |
| 880 | 70 | 0.142857 | 0.01823 | 0.29208 | 0.08176 |

**Data Analysis:**

1. *Building:* It’s very noticeable that the Linked List priority queue takes much more time to build than the STL priority queue and Minheap priority queue as the data size increases. In fact, it appears to have an exponential complexity, or O(n2). This is because the Linked List implementation has the possibility of going through the entire linked list to check priorities *and* going through the entire linked list to check treatments. Both the STL and Minheap priority queues run much faster with complexities that appear to be linear, or O(n). This is because, by being ordered as a tree, certain nodes can be disregarded at every level so that not every node need be checked. However, there is again the possibility of *both* priorities and treatments needing to be checked. Between the two, the Minheap priority queue runs faster than the STL priority queue.
2. *Deleting:* Surprisingly, the STL priority queue takes much more time to delete than both the Linked List and Minheap priority queues. It appears to have a linear complexity, or O(n). The Linked List and Minheap priority queues delete much faster, and appear to also have linear complexities, or O(n). It is understandable why the STL and Minheap take longer than the linked list; since they might have to re-organize nodes every time an element is de-queued, the deleting process will take longer than just deleting the head of the linked list every time an element is dequeued from the Linked List priority queue.