Taylor Ellis

Prof. Hoenigman

CSCI 2270

6, May 2018

**Final Project Write-up**

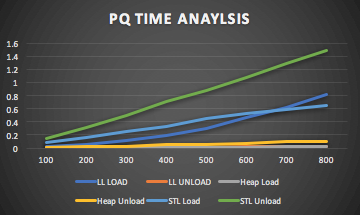
The purpose of this project is to explore the efficiency of various implementations of priority queues, and ultimately we want the most efficient implementation to use to process our hoard of pregnant women. Within this project we completed 3 implementations. These implementations consisted of a heap priority queue, linked list priority queue, and a priority queue implemented using the standard template library (STL primary queue).

The linked list implementation involves adding elements to the LL by searching for its correct position within the LL. Removing elements involve deleting that element and re-assigning its head. The min heap implementation involves “bubbling” the elements into its correct position when adding (push) an element. Popping or removing an item from a min heap simply involves removing the top value, or heap[1]. The STL implementation was fairly simple, and just involved calling the respective .pop() and .push() functions.

The data set used for this project consist of a string and two integers for each data point. The string represented the patient's name. The first int represented their time until delivery, and is what determined their priority within the queue. The lower the number, the higher the priority (less time until they are due). The second integer represented the time the patient would need to spend with the doctor. This data field served as the tie-breaker for when the women had the same delivery time, and the smaller the treatment time the higher the priority.

**Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **LLSize** | **Avg Load Time (ms)** | **STDEV (ms)** | **Avg Unload Time (ms)** | **STDEV(ms)** |
| 100 | 0.01804 | 0.133247 | 0.004008 | 0.0632452 |
| 200 | 0.056122 | 0.230422 | 0.006014 | 0.0774074 |
| 300 | 0.114246 | 0.318498 | 0.010022 | 0.0997177 |
| 400 | 0.190412 | 0.393151 | 0.012024 | 0.109103 |
| 500 | 0.30466 | 0.460982 | 0.01403 | 0.117742 |
| 600 | 0.458994 | 0.499315 | 0.02004 | 0.14028 |
| 700 | 0.625355 | 0.485431 | 0.022048 | 0.147003 |
| 800 | 0.817775 | 0.388326 | 0.02405 | 0.153367 |
|  |  |  |  |  |
| **Heap Size** | **Avg Load Time (ms)** | **STDEV(ms)** | **Avg Unload Time (ms)** | **STDEV (ms)** |
| 100 | 0.00301 | 0.0387422 | 0.007024 | 0.0589467 |
| 200 | 0.005016 | 0.0499085 | 0.018062 | 0.0934658 |
| 300 | 0.00602 | 0.0546241 | 0.029102 | 0.117283 |
| 400 | 0.00802 | 0.0628948 | 0.048162 | 0.147793 |
| 500 | 0.013044 | 0.079837 | 0.061216 | 0.164222 |
| 600 | 0.014048 | 0.0827693 | 0.075254 | 0.17914 |
| 700 | 0.01806 | 0.0934555 | 0.093318 | 0.195219 |
| 800 | 0.019062 | 0.0959101 | 0.105358 | 0.204349 |
|  |  |  |  |  |
| **STL Size** | **Average Load Time (ms)** | **STDEV (ms)** | **Average Unload Time (ms)** | **STDEV (ms)** |
| 100 | 0.082282 | 0.185774 | 0.143492 | 0.226722 |
| 200 | 0.167564 | 0.236617 | 0.306032 | 0.2447 |
| 300 | 0.258878 | 0.250723 | 0.489658 | 0.0767861 |
| 400 | 0.329116 | 0.238329 | 0.707399 | 0.24676 |
| 500 | 0.45755 | 0.14213 | 0.87898 | 0.216665 |
| 600 | 0.522781 | 0.100651 | 1.07265 | 0.173038 |
| 700 | 0.589 | 0.190216 | 1.28132 | 0.249398 |
| 800 | 0.655222 | 0.23119 | 1.48202 | 0.114264 |



As seen above in the data, the min-heap loading implementation is the fastest of the three, while the linked list PQ implementation has the quickest unload time. However the LL loading time seems to exponentially grow as the linked list does. On the other hand the STL implementation seems to be O(n) in complexity. There is a high standard of dev for the lower data sizes used, as expected since for the smaller data sets.