Alex Barclay

CSCI 2270

02 May 2018

Sreesha Nath

Final Project

The purpose of this project is to compare and contrast three data structures insofar as they are to resolve a situation which requires the creation of a queue which is sorted on the bases of two levels of priority. In this specific case, it is a matter of pregnant women being added to a queue, sorted on the basis of time to birth; in the case of comparable times to birth, members of the queue are further prioritized on the basis of their required treatment time. Given the nature of these circumstances, a minimum priority queue will be required, as the lowest number represents the highest priority. Assuming that there are no flaws or issues with each implementation, the question is which of the three data structures will perform fastest. The three data structures compared for this purpose are the linked list, binary heap, and STL priority queue.

When comparing the rates of performance among a linked list implementation, a binary heap implementation, and an STL priority queue implementation, it may be initially unclear as to which is the best. A linked list—singly, for the sake of this implementation—has a theoretical complexity of O(1) for both insertion and deletion, with a worst-case space complexity of O(n). A binary heap, comparatively, has a complexity of O(log n) for its insertion and deletion, and is built in O(n) time, but also requires a method of organization, commonly called “heapify,” which has an average complexity of n log(n). Thus far, it would appear that the linked list has the upper hand in terms of insertion and deletion, while also having roughly the same space complexity. An STL priority queue which uses a heap also has a complexity of O(log n) for insertion and deletion, which means that all three methods should be somewhat comparable to each other. But the results do not bear out that way.

After five thousand iterations, it becomes obvious that the linked list exhibits the lowest efficiency, at an average run-time of 0.00141s. For reference, the binary heap exhibits an average run-time of 0.00059s, and the STL priority queue exhibits an average run-time of 0.00054s. The reason for this is that the linked list must iterate through the entirety of the list to add something at the end, which would be significantly improved to O(1) for a doubly linked list. Specifically, the enqueue function utilized in this instance sorts as it adds, which adds an addition to the runtime operation as the new node is compared to each in turn, stopping when it finds one with lower priority. Prior to the process of iterating through the list, it is first necessary to check if the list’s head exists, as well as if the head should be replaced by the new addition. Following this, the function creates a second pointer to serve as the previous node; when iterating through the list, if a node of lower priority than the new node is found, the lower-priority node’s previous pointer is changed to indicate the new node, while the new node’s next pointer is changed to indicate the lower-priority node. This comparison is made in light of the parameters as described above—time to birth, then treatment time. The dequeue function is better than enqueue in terms of worst-case complexity, as it is essentially equivalent to a “pop” function—the head of the list’s contents are copied for return, then the node itself is deleted while its successor node becomes the new head. This grants it a run-time of O(1), because no iterative searching is required.

The binary heap is slightly different. Because the overall size is also tracked, the enqueue has worst-case complexity of O(1), because any additions are placed at the index accorded by the size. Prior to this, however, the original heap is copied over to a newly-created one, which is larger by one index. The dequeue functions similarly to its implementation for linked lists, except that the size must also be managed. There are additional helper functions used for the purposes of navigating the array implementation of the binary heap.

There isn’t much to be said for the STL implementation, as no functions are required for it.

The comparative results are listed below, at varying levels of iterations. Because iterations were the source of alterations, they are listed as the independent variable.