Programming Assignment 3 Written Report

***Introduction***

This programming assignment tests knowledge and implementation of three different priority queues: two based on a list (one sorted, one unsorted), and one using array-based heap implementation. These three are unique in the ways they insert and remove random numbers, and each of these will be tested and graphed to document their relative efficiencies. Overall, the objective is to see which of the previously listed implementations runs the fastest, and describe ways one could use them in everyday coding to boost performance.

***Theoretical Analysis***

The insert() operation, as used in a priority queue, simply adds a new element to an existing list or tree. The sorted queue navigates through the existing list, finds the position where the new element fits, and then inserts it there, all while keeping the integrity of the sorted list. The unsorted queue’s insert operates faster, for it simply places the new element at the front (or back) of the list and doesn’t need to maintain any sort of order. Finally, the heap implementation uses the heap-sorting algorithm to locate the position of the new element, and inserts it there. The heap implementation’s insert function runs in O(log n) time, where n is the number of elements in the queue. This is because the heap is a balanced binary tree with height log(n), and must maintain the integrity of this tree to operate properly. Since we can insert items at the beginning or end of the list, the unsorted queue can insert items in O(1) time. With a sorted list, we have to find the best position to insert the new element, taking it O(n) time to complete.

When sorting any list or queue, the removeMin() function is necessary to output the numbers into a sorted list. The implementations of the sorted and unsorted (list-based) queues can tackle this with relative simplicity, while the heap is a little more complicated. Since the sorted queue is already in order, it can simply remove the first element in the list, running in O(1) time. On the other hand, the unsorted needs to cycle through the list and compare each element to find the smallest one, running in O(n) time. This is sort of a pay-me-now, pay-me-later situation, for the list-based implementations either takes O(1) or O(n) to insert an element, and the other to remove. Ideally, these two will run in pretty similar time, because they are just opposites of one another. The heap implementation, however, takes equal time to insert() and removeMin(), but they both run in slightly less time (on average) than the list-based queues. As the heap’s insert runs in O(log n) time, so does the removeMin, since the tree has height log(n). Theoretically, when sorting a series of numbers (using Insert and removeMin), the heap should run in the fastest time. This is because it adds two functions of O(log n) time together, versus the list implementation that takes O(1) and O(n) to complete. However, the experimental data puts this to the test, and is discussed in paragraphs to come.

***Experimental Setup***

I ran this test on my MacBook Pro (v. 10.10.2) using TextWrangler and the built-in G++ (Linux) compiler. The first of these tests that I ran was the Unsorted Priority Queue, where the size of the numbers.txt file was only 10. As expected, the terminal took on a file of this size with relative ease, finishing in merely hundredths of a second. As explained earlier, Unsorted Priority Queue would insert a new element at the beginning of the list on O(1) time, and would take the majority of its time executing the removeMin in O(n) time. Theoretically, we decided that this should be faster than the Sorted PQ, for the most efficient algorithm would take the least time inserting elements instead of removing them. After testing with 10, 100, and 1000 input numbers, it was finally decided to use 100,000 numbers in the input file. This size was primarily set to insert and remove a reasonable amount of data, while still maintaining a runtime of under a minute.

Sorted Priority Queue was the second algorithm to be tested. Similarly, the input sizes tested were 10, 100, and 1000, and eventually set at 100,000. The Sorted Priority Queue would take the longest inserting new elements in O(n) time, while removing the minimum value would only take one cycle through the data in O(1). As discussed earlier, this algorithm would probably be the least efficient of those tested, primarily because it takes so long to insert each individual element in the constantly growing list. With this in mind, the experiment was set to run with enough numbers to differentiate the different algorithms, without spending too long actually running them.

The Heap Priority Queue was the more abstract way of tackling this assignment, but running in O(log n) time, was theoretically the most efficient. With this in mind, the run time of the Heap Priority Queue really had no role in determining the input size, for it would run in less time than the others, regardless. The experimental results that follow help to support this data.

***Experimental Results***

As discussed above, the Heap Priority Queue proved to be (by far) the most efficient of the algorithms tested, followed by Unsorted, and finally Sorted Priority Queue. Since both Unsorted and Sorted were list-based queues, implementing an array-based queue was almost guaranteed to be more efficient, as was the Heap.

As the fastest to run, the Heap Priority Queue’s array-based system of up heaping and down heaping successfully inserted and removed 100,000 values in less than half of a second. Using this algorithm, the input size had little to do with runtime, for the up and down heaping took far less time than the linear sorting characteristic of the other 2. The input size definitely affected the runtime in the list-based Sorted and Unsorted Priority Queues, for the larger the input, the more distinct the difference between the two became. With the tested input size of 100,000, the Unsorted queue averaged around 0:32 in each of 10 tests, and the same tests produced an average of 0:43 using the Sorted queue algorithm.

Overall, the theoretical results matched up with the experimental, with the exception of the differences between Sorted and Unsorted. This experiment illustrated the real difference in runtimes of the Insert() and removeMin() functions, for the two algorithms that were supposed opposites of one another proved otherwise as a result of these tests. As introduced, the purpose of this assignment was to test students’ knowledge and ability to prove the differences between three fundamental ADT’s, by comparing them to one another in efficiency and runtime. After rigorous testing and documentation, it was proven that the Heap Priority Queue was prominently the most efficient on this large scale.