Yuri Castro
CSCE 221H-200
Data Structures and Algorithms
March 21, 2018

Programming Assignment 3 – Priority Queues and Heaps

Introduction

The objective of this assignment is to implement three different types of priority queues. With these three different implementations, we are to compare the times of inserting and removing *n* random numbers. The three type of implementation follows: 1) a list-based implementation of an unsorted priority queue, 2) another list-based implementation of a sorted priority queue and, 3) an array-based implementation of a heap priority queue. By the end of testing all three types of priority queues, we can see how the choice in type of implementation may affect performance.

Theoretical Analysis

Here, I will provide an analysis on the complexity for each of the implementations:

1) With an unsorted priority queue, the given list of numbers is inserted into the priority queue as it without sorting them. However, when calling removeMin() to remove the items from the priority queue, the code then iterates through the list each time searching for the smallest value to remove.
    a. The time complexity to add in items to the priority queue is O(n) for each *n* items in the list. Also, there is no sorting at this stage yet.
    b. The time complexity to remove items from the priority queue is O(n^2). This is because when removeMin() is called, it will iterate through the list reach time to extract the smallest value – O(n). The additional O(n) comes from calling removeMin() for each *n* elements in the priority queue—O(n).

    Therefore, the overall time complexity for the unsorted priority queue is O(n^2) in inserting and removing elements.

2) With a sorted priority queue, the given list of numbers is sorted as it is inserted into the list. So when calling removeMin() to remove the items from the priority queue, the code simple removes the first element in the queue.
    a. Inserting an element into the priority queue will have a complexity of O(n^2). This is because inserting the item in the list in a sorted matter will have O(n) in iterating through the list, finding the place to insert the element. And this is executed for each *n* number of elements in the list – O(n).
    b. The time complexity to remove items from the priority queue is O(1) for each removal from the front of the queue. So for each *n* number of elements, it will have a complexity of O(n).

    Therefore, the overall time complexity for the sorted priority queue is O(n^2) in inserting and removing elements.

3) In using a heap to implement a priority queue, the complexity for inserting an element and removing the smallest element is much different than using a list.
    a. Inserting an element into a heap will have a complexity of O(log n). This is because in inserting an element into the heap, we add it to the end of the heap tree. From there, it is needed to check is the heap properties are held still, and heapify if needed which takes O(log n).
    b. Removing the smallest element is also O(log n). this function in a heap swaps the current root to the last leaf of the heap and then heapify as needed which is O(log n)

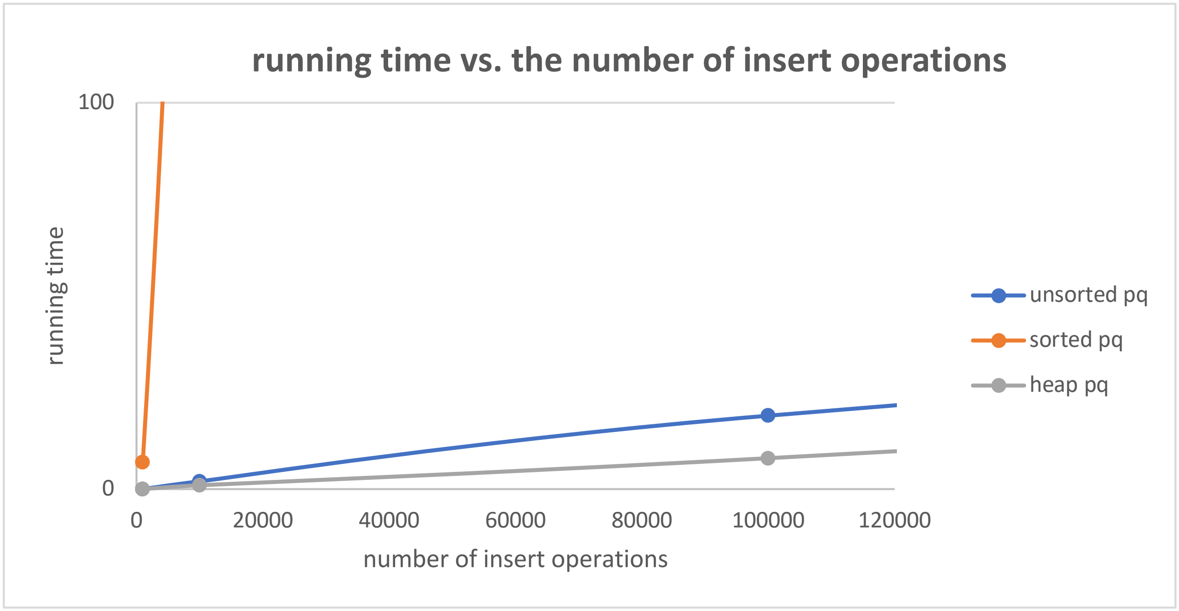
Overall, it will take O(log n) to insert and removing *n* number of elements in the heap priority queue.

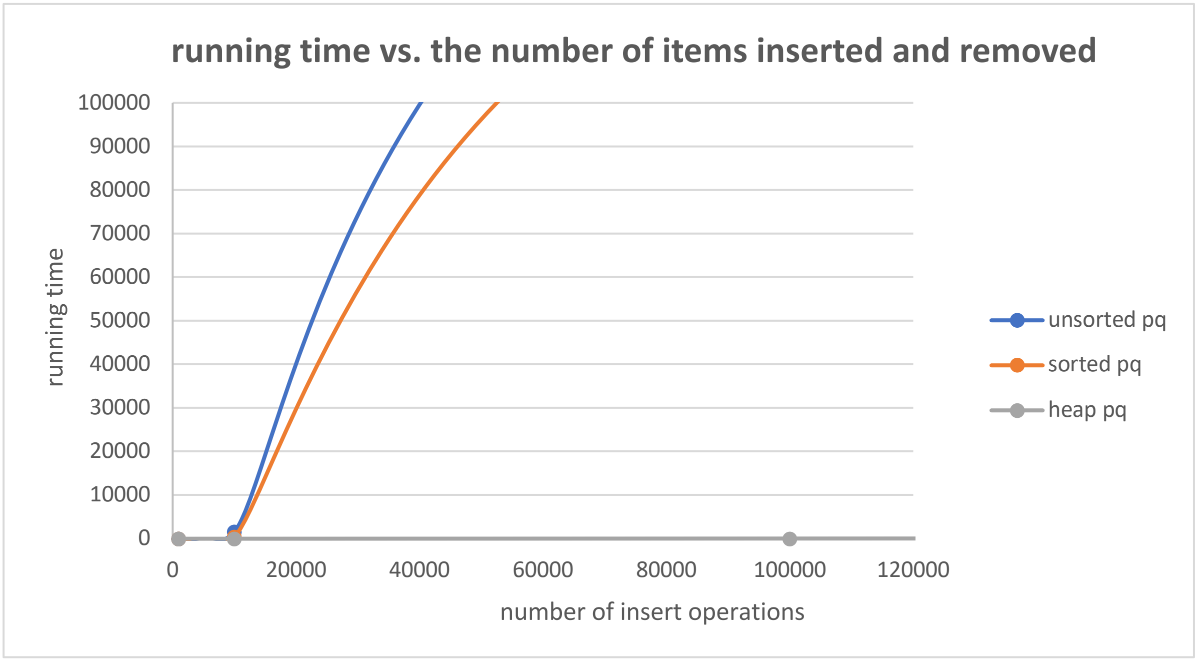
Experimental Setup

In testing the three different types of implementations, I generated several sets of random numbers and ran each set of numbers through the implementations and timed them in intervals.
For the test inputs, I just generated a random number from 1 through 10 for the wanted *n* about of inputs to test. Tnumber of elements in each test set are: 10^6, 10^5, 10^4, and 10^3. This way, we get a good comparison in time runs as the input grows greatly. I used integer data values so it will be easier to compare between values unlike doubles and such. Additionally, I tested each different amount of inputs three times and averaged it.

Machine Specification: macOS 2015 with 2.9 GHz Intel Core i5 processor and 8 GB 1867 MHz DDR3 memory.

Experimental Results





The graphs could not fully show the large differences between the different priority queues. But as I was testing the implementations of the priority queues, there was a big difference in the amount of time it took. The unsorted queue took a while when removing elements. The sorted queue took a while inserting the elements. And the heap took a much shorter amount of time. Overall, it can be concluded that the heap priority queue performs the best. It gradually rises in running time based on input. On the other hand, the two lists-based implementations (unsorted and sorted) have large running times based on input. With the unsorted list, it took a while to remove the elements in order. And with the sorted list, it took a while to properly place the elements in correct order.

The timing of the code execution may be inaccurate in the timer not being able to pick up the smallest times. Additionally, it is not sure if my code is completely correct in the desired time complexities.