CSCE 221 Cover Page

Programming Assignment #3

Due Date: Wednesday October 30, 11:59pm

Submit this cover page along with your report

First Name: Vidhur Last Name: Potluri UIN: 626007235

**Any assignment turned in without a fully completed coverpage will receive ZERO POINTS.**

Please list all below all sources (people, books, webpages, etc) consulted regarding this assignment:

CSCE 221 Students Other People Printed Material Web Material (URL) Other

1. 1. 1. Course materials 1. <http://www.cplusplus.com/reference/vector/vector/> 1.

2. 2. 2. 2. 2.

3. 3. 3. 3. 3.

4. 4. 4. 4. 4.

5. 5. 5. 5. 5.

Recall that University Regulations, Section 42, define scholastic dishonesty to include acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. Disciplinary actions range from grade penalties to expulsion. Please consult the Aggie Honor System Office for additional information regarding academic misconduct – it is your responsibility to understand what constitutes academic misconduct and to ensure that you do not commit it.

I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received nor given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment.

Today’s Date: 11/1/2019

Printed Name (in lieu of a signature): Vidhur Potluri

**CSCE 221 505**

**Programming Assignment #3**

**Report**

Vidhur Potluri

UIN: 626007235

**Introduction**

The aim of the assignment is to determine the run time complexity and efficiency of the different implementations of the Priority Queue. The assignment includes inserting random numbers on to three different implementations of the priority queue – an unsorted priority queue that doesn’t sort its items and traverses through the list to find the minimum value (UnsortedPQ), a sorted priority queue that sorts the queue as items are inserted (SortedPQ), and a heap priority queue (HeapPQ) – and comparing the times it takes to insert and remove minimum value items from each implementation.

**Theoretical Analysis**

All methods but insertItem() and removeMin() have a runtime complexity of O(1) for any of the three implementations. The only way to insert an item into the priority queue is using the insertItem function for all three implementations of the priority queue. The unsorted priority queue has a runtime complexity of *O(1)* for a single insertItem operation since it involves executing a simple pushback function. Therefore, its average runtime complexity to insert n items is . Since the unsorted priority queue doesn’t sort its elements but instead searches for the item with the least value, each call for the removeMin operation results in the complete queue being traversed, leading to a runtime complexity of O(n).

The sorted priority queue sorts the queue as elements are inserted into it, and therefore has a runtime complexity of O(n) for its insert operation. The removeMin() function simply erases the first item in the already sorted queue and therefore, has a runtime complexity of O(1). Since both the sorted and unsorted priority queue have O(1) and O(n) complexities for their operations, their total runtime complexities for the sorting of n elements of both implementations of the priority queue is . Since both implementations have the same average complexity, a decision about which implementation should be used would be made based on whether the user prefers a linear search or sort over the other. Both implementations aren’t very efficient because of their high runtime complexities.

The heap implementation of a priority queue with n elements is an array that resembles and functions as a binary tree with height *log(n)*. Sorting in the form of downheap and upheap happens each tine an item is inserted into the queue. Every time the heap is updated, the heap needs to be re-ordered. Upheap and downheap help with this reordering after insertItem and removeMin operations respectively. Since a heap has a height upheap and downheap run in time. The heap-sort, therefore, sorts n elements and runs in time, and can therefore be considered to be a more efficient alternative to the list based implementations of a priority queue.

**Experimental setup**

1. *Machine specifications:*

Computer model: HP Spectre x360

OS: Windows 10 Home

Processor: Intel® Core™ i5-8250U CPU @ 1.60 GHz 1.80 GHz

Installed Memory: 8.00 GB (7.84 GB Usable)

1. Procedure and testing: The input numbers for the different queues were generated randomly using the rand() function from the ‘stdlib.h’ library. The functions for each of the classes I created were derived from the PriorityQueue.h template provided to us. I pushed the same number of random numbers on to all of the different implementations to facilitate run time complexity evaluation and comparison. Each of the priority queues was tested twice to make sure the results are approximately consistent. I’ve shown the times for only 1 trial for easier comparison between implementations. The following are the classes I created, and the methods I used to test their ability to handle inputs of different sizes.
   * 1. UnsortedPQ: The UnsortedPQ class used the STL library’s list class to create a queue for the items. Using the library allowed me to use its basic functions without me having to create my own. The insertItem() function simply pushes the item to the end of the list, and there is no sorting involved. The removeMin() function creates a variable ‘min’ that is compared to every item and is replaced by an item that has a lower value. Once the function traverses through the whole list, ‘min’ is removed. I tested it by inserting and removing 100,000 randomly generated numbers.
     2. SortedPQ: The SortedPQ class also used the STL library’s list class to create a queue for the items. The queue is sorted as items are inserted. So, the removeMin() function removes the first item from the list. I tested it by inserting and removing 100,000 randomly generated numbers.
     3. HeapPQ: The HeapPQ class uses the vector library and its functions to generate a heap priority queue. The insertItem() function sorts the array to where it resembles a heap using the upheap technique. The removeMin() function removes the minimum item at the top of the heap and then uses downheap to sort the array. I tested it by inserting 100,000 randomly generated numbers and removing them.

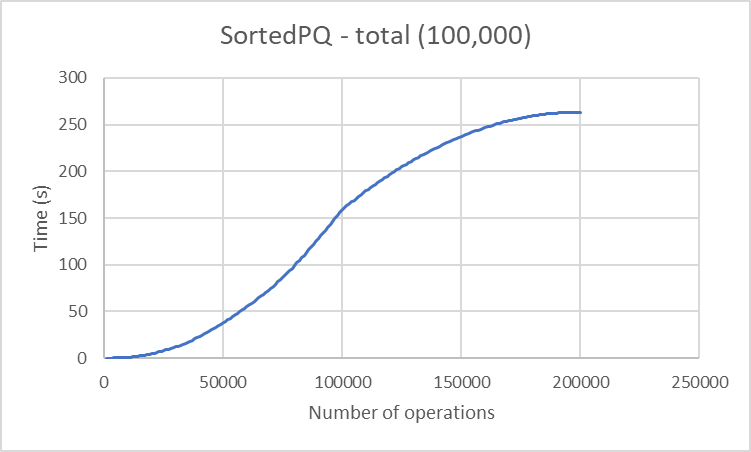
**Experimental results**

The following are the runtimes of the insertItem and removeMin operations for the different Implementations of the priority queue.

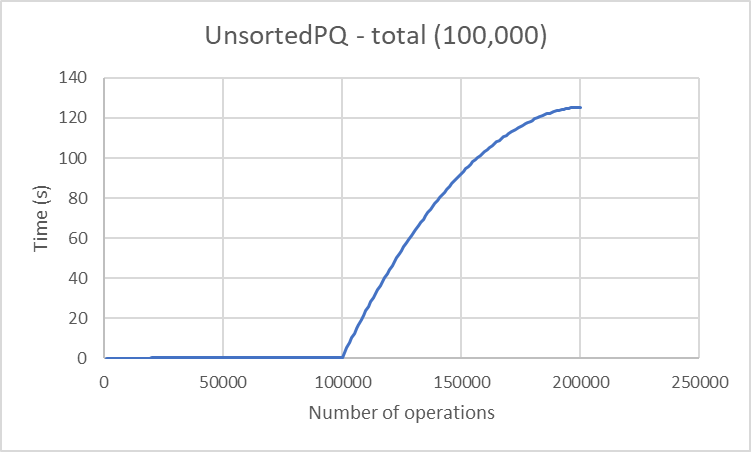
**Fig 1 – UnsortedPQ with 100,000 operations**

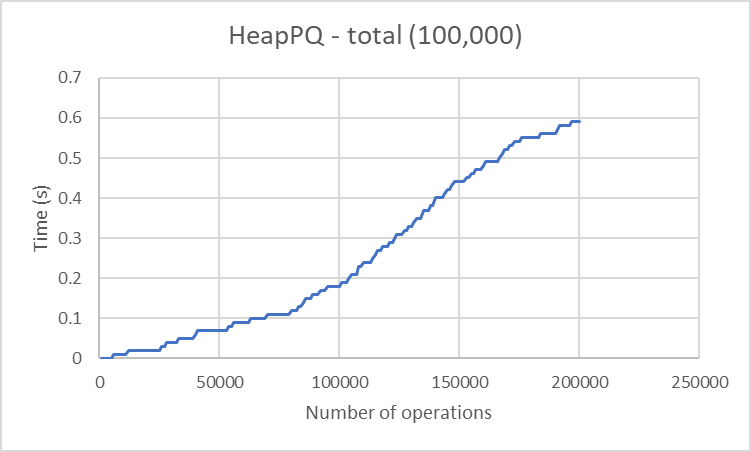
**Fig 2 – SortedPQ with 100,000 operations**

**Fig 3 – HeapPQ with 100,000 operations**



**Fig 4 – Total time – Sorted PQ (remove min() after 100000 operations)**

**  
Fig 5 – Total time – Unsorted PQ (remove min() after 100000 operations)**

**  
Fig 5 – Total time – Heap PQ (remove min() after 100000 operations)**

(Note: The runtime of my sorted list’s remove function does not agree with the runtime complexity that I derived in the theoretical analysis. The curve should have been approximately flat once the process of removing the items began because of a runtime complexity of Instead, it appears to have a runtime complexity of .

Discussion**:** The sorted and unsorted priority queue are significantly slower than the heap-based implementation as predicted in the theoretical analysis. However, the sorted and unsorted priority queues have very similar sorting times. The sorted list would be a better option if the user needs to return not just the minimum value from the list but also the order of the elements in the queue.

I believe the slight discrepancies observed between the trials and across the different implementations might be because of the changing frequency of the processor of the computer. Although the heap-based priority queue proved to be more efficient, the needs of a user (client) and their resources (processor, power, money, etc) budget might lead the user to prefer a different implementation of the priority queue. The heap makes the order of the list a little difficult to understand because of its binary tree orientation. However, the heap implementation is the ideal choice and most efficient to an end user.