CSCE 221 Cover Page

Programming Assignment #3

Due Date: Wednesday October 30, 11:59pm

Submit this cover page along with your report

First Name: Khoa Last Name: Diep UIN: 926005094

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.	1. cplusplus.com	1. 221 notes
2.	2.	2.	2. stackoverflow.com	2. 121 notes
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): Khoa Diep

Introduction:

The objective of this programming assignment is to implement a Priority Queue, an abstract data type, using three different implementations of the Priority Queue. A priority queue is a queue where each element has a priority and will remove elements based on that priority. For this project, the smaller the element, the higher the priority. Thus the minimum value will have the highest priority and will be the first one to be removed. The first implementation UnsortedPQ, which is a priority queue where the element, while inserted, will be unsorted in terms of priority, but when removed will search the priority queue for the element with the most priority, the minimum value, and remove it from the priority queue. The second implementation is SortedPQ, which is a priority queue where the element, while inserted, will be sorted in terms of priority. When removed, the first element will be removed, which holds the minimum value. Both UnsortedPQ and SortedPQ will be implemented using lists. The third implementation is HeapPQ, which is an array-based implementation of a heap. A heap is a complete binary tree data structure where the left child and right child will always be less than its parent for max-heap and be more than its parent for min-heap.

Theoretical Analysis:

Each implementation has a different performance time. During the insertion stage, UnsortedPQ only has to add a new node, which holds that new data, to the end of the linked list and update the tail pointer to point to the new element. We then iterate s(which represents the size of the priority queue). However, when removing the minimum value, the function has to traverse through the entire list to find the minimum data point and then remove the element, without affecting the priority queue. If the minimum is at the head or the tail, we will need to update the head or the tail pointer to the next element or the previous element respectively. If the minimum is in the middle of the PQ, we will have to update to data's previous element to the data's next element and vice versa. We then decrement s.

Let n represent the size of UnsortedPQ. While inserting we will only have to create a new node holding the new data element and updating the tail pointer. Regardless of n, inserting will always be the same, with time complexity of O(1). Thus inserting n elements will be n*O(1) = O(n), we can conclude that the time complexity of inserting into UnsortedPQ is O(n) for the average case. This is also the best-case and the worst-case since there will be the only way to insert into UnsortedPQ. While removing, we will have to traverse the entire list to find the min using binary search. Thus the time complexity of removeMin() is O(n). Thus removing n elements, it will be n*O(n) or O(n^2). We can conclude that the average time complexity of removing n element will be O(n^2), which is also the worst case and best case scenario with the same logic as the insertion method.

In conclusion -
The time complexity of inserting n items in UnsortedPQ is O(n) in the average case scenario
The time complexity of removing n items in UnsortedPQ is O(n^2) in the average case scenario

While inserting into SortedPQ, we will have to sort as we insert. If SortedPQ is empty, then the head and the tail pointer will point to the new node that holds the new data element. Otherwise, the function will traverse the list from the head to the tail until the data being inserted is greater than the current node's data or reach the end of the list. We will have to update to the pointers, based on where the current node of the list is. If the current node at the beginning or end of the list, the head and tail pointers will have to be updated. Otherwise, the node will be in the middle of the list and will have to update the pointer of the current node and the current's next's node.

Let n represent the size of SortedPQ. While inserting into SortedPQ, the sorting algorithm will have a time complexity of O(n) during the average case. This is because we will most likely insert a node in the middle of the list. The worst-case scenario is inserting at the end of the list (which would be when the data element being inserted will be greater than all of the data elements in the list), which is still O(n). The best-case scenario would be inserting it at the beginning of the list (which would be when the data element being inserted will be less than all of the data elements in the list), which is O(1). The time complexity of inserting n items will be n*O(n) or O(n^2). While removing into SortedPQ, the function removes the head, removes its pointers, and returns the head's data. We can conclude that removeMin() will have a time complexity of O(1) on the average case, which is also the best case and worst case. Thus removing n items will have a time complexity of n*O(n) or O(n^2).

In conclusion -
The time complexity of inserting n items in SortedPQ is O(n^2) in the average case scenario
The time complexity of removing n items in SortedPQ is O(n) in the average case scenario

While inserting into HeapPQ, we will have to build a min-heap as we insert using arrays. For HeapPQ, we will use a doubling dynamic array similar to PA1. We will insert the data into the end of the array and then perform upheap on the new element. Upheap for a min-heap functions as such. We check the parent of the data, if the parent is less than the data, then we will swap the two. We will repeat this process until the parent is not less than the data, or we hit the root of the heap. While removing the minimum value, we store the first element of the heap on the stack (minimum value. We then replace the first element of the heap with the last element. We then perform downheap on the root, which now holds the previous last element. Downheap of min-heap works as such. We check the left child and the right child of the node. If the left child or the right child is greater than the node's data then we will swap whichever one if smaller than the other (if they are equal then swap the left child with the parent). We will repeat this process until if the left child or the right child is greater than the node's data, or we reach the end of the heap.

Let n represent the size of HeapPQ and let h represent the height of the tree, where h = log(n) since a heap is a complete binary tree. While performing upheap while inserting, the worst-case scenario would be if the data inserted will have to traverse to the root, the time complexity will be O(log n), which is the height of the tree. The best-case scenario would if we don't need to perform upheap, which will be when the parent of the data is less than data which is O(1). Thus inserting n items will have a time complexity of n*O(log n) or O(n log(n)). The average case for upheap will be O(log n), where we will stop upheap in the middle in heap. The time complexity of removal is proportional to the downheap. While performing downheap while removing, the worst-case scenario would if the root would have to traverse to the bottom of the heap, with the time complexity of O(log n). The best-case scenario would be if downheap is not called (when all the elements in the heap are the same). The average case-scenario will be when O(log n) where we will stop downheap in the middle in heap. Thus removing n items will have a time complexity of n*O(log n) or O(n log(n)).

In conclusion -
The time complexity of inserting n items in HeapPQ is O(n^2) in the average case scenario
The time complexity of removing n items in HeapPQ is O(n) in the average case scenario

The advantages of UnsortedPQ will be if we insert a lot of data but rarely remove it, the time complexity of inserting is O(1), but removing is O(n). This means that if inserting n items will have a time complexity of O(n), and say we only need to remove two items from the priority queue, then the time complexity is O(n) + 2O(n) = 3O(n). This would be much faster than sorting n items and then removing them. However, the disadvantage is that when the number of removals approaches n, the time complexity removal is O(n^2), which is very slow.

The advantage of SortedPQ will be if we do not care about the complexity of inserting an item. Let arbitrary say that inserting into SortedPQ will be done ahead of time, we then only care about the complexity of the removal of n items will be O(n). Another advantage is if we insert data that is already sorted so that the time complexity of inserting will the best-case scenario for every item, thus the time complexity will be O(1). The disadvantage will obviously be when we do care about the complexity of the inserting.

The advantage of HeapPQ will be that inserting and removing n data elements together will be far faster than both UnsortedPQ and SortedPQ. However, inserting n data elements will be slower than UnsortedPQ and removing n data elements will be slower than SortedPQ

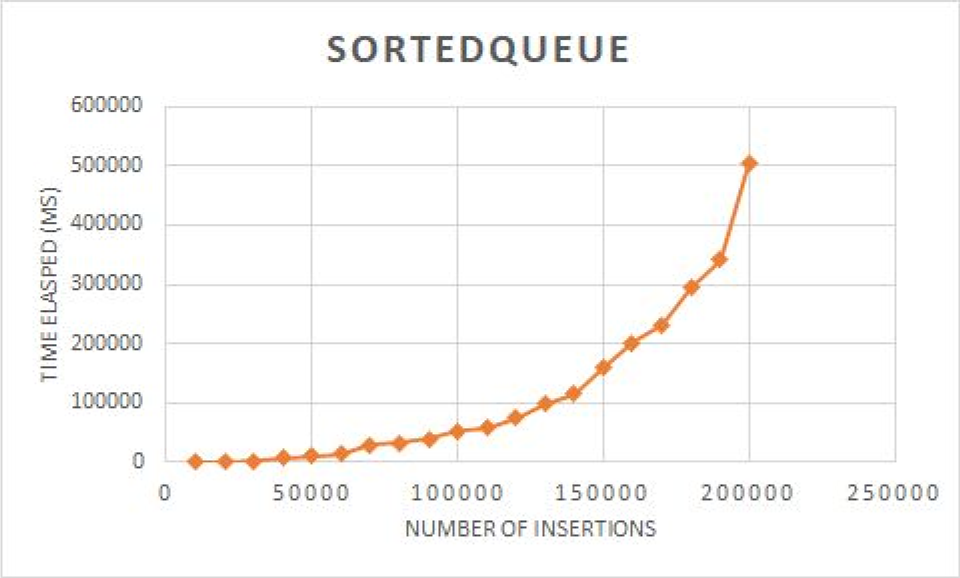
Experimental Setup:

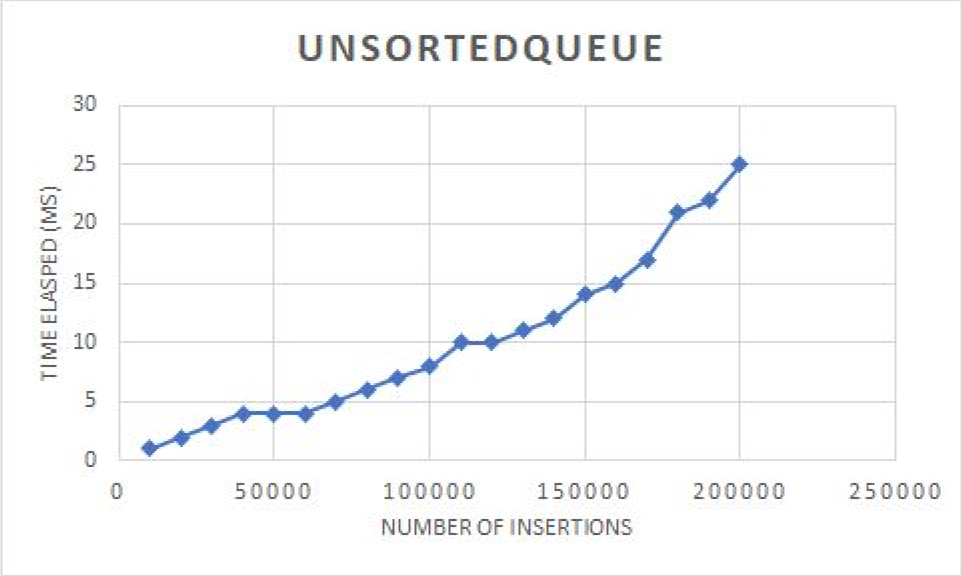
I tested on my laptop, which has an Intel Core i7 processor, 16 GB of RAM, and operating on Windows 10. An IDE called Eclipse is used to write, compile and run the project. In order to test the functions, I created three Priority queues of each type. For all three, I used a for loop to insert and remove, test the size() function and minValue() in main.cpp. I also created a print() function which just prints the linked list from the head to tail for UnsortedPQ and SortedPQ, and prints the array for HeapPQ to check the correctness of the priority queue. I also implemented the number.txt method stated in the instructions. I used <chrono> in the standard library for timing, the amount of time with n number of inserts/removes.

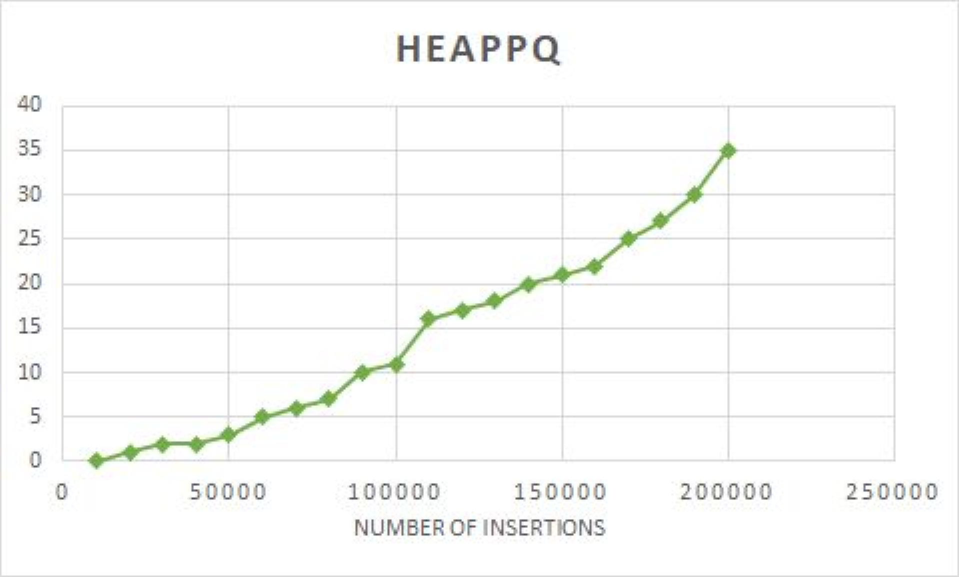
The test inputs used were 10,000 to 200,000 random numbers being inserted and then removed in increments of 10,000 for each implemented priority queue. I started at 10,000 since we can see the difference in speed between SortedPQ and UnsortedPQ since it was a large enough number so while inserting random numbers then deleting those numbers, there will be a smaller chance of lucky cases for UnsortedPQ. I repeated this experiment three times for each data structure.

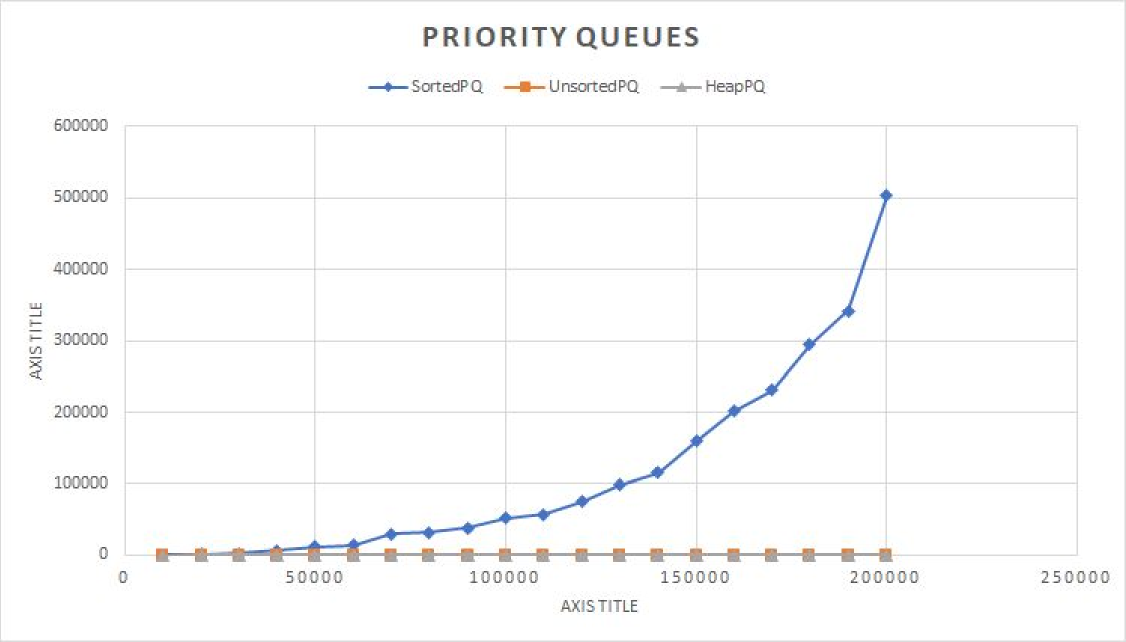
Experimental Results:

Insertion Graphs

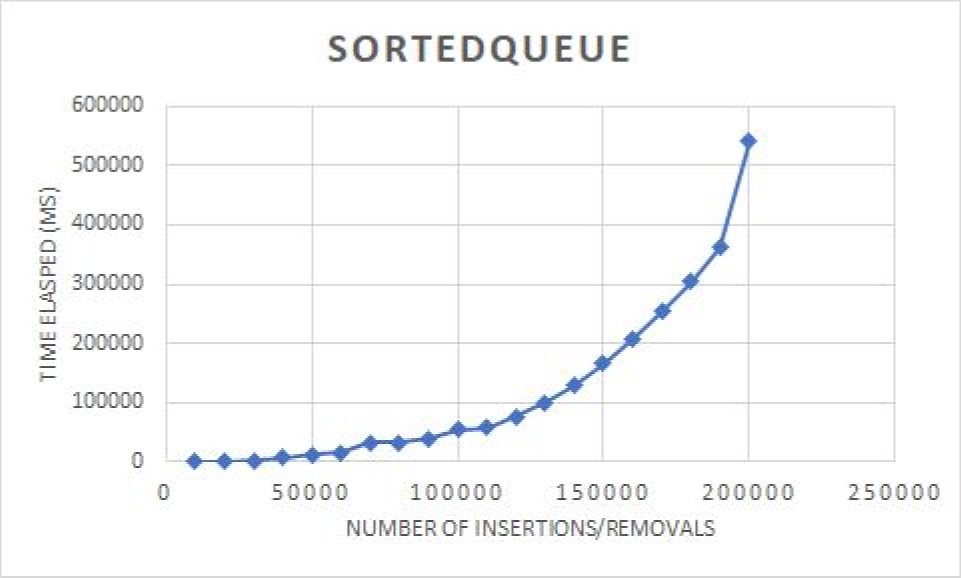


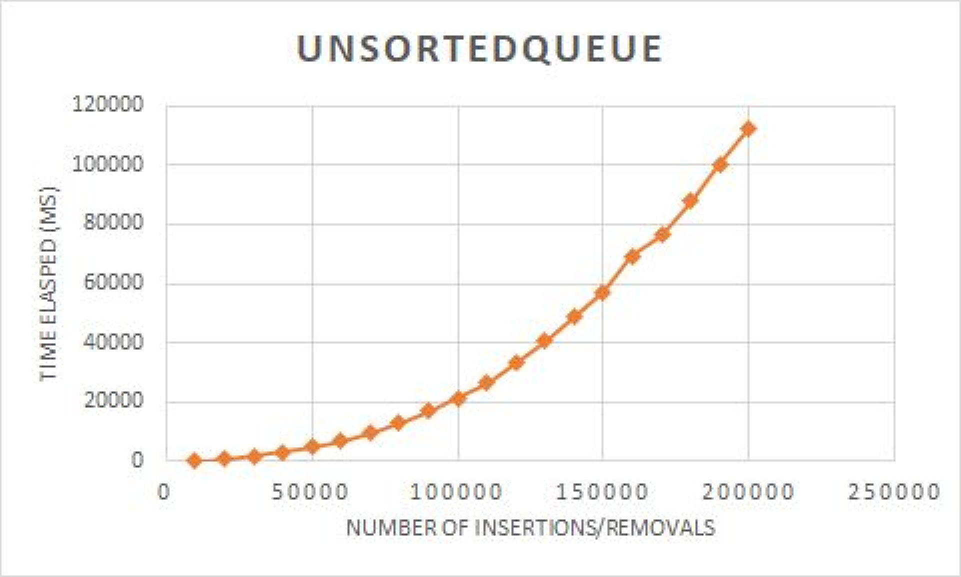


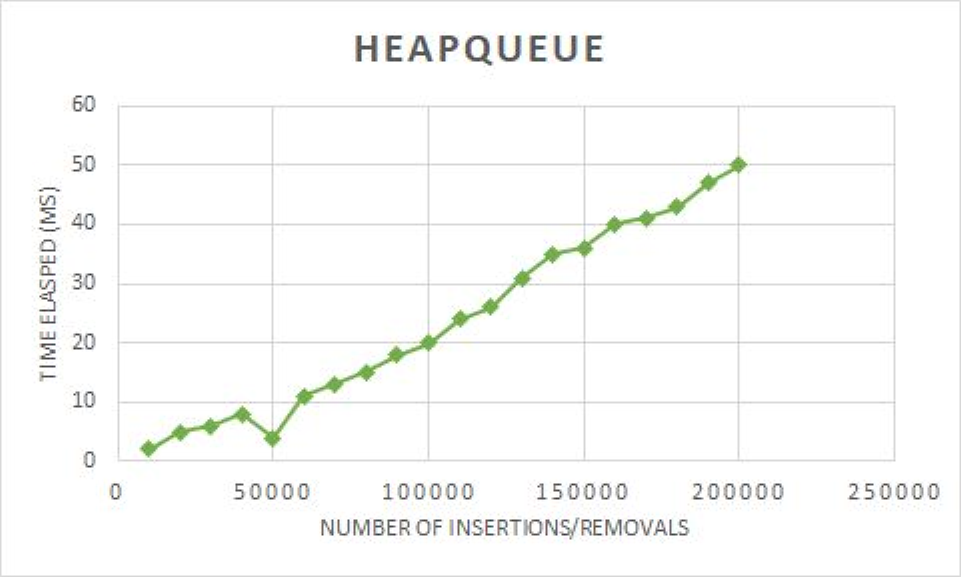

HEAPPQ
40
35
30
25
20
15
10
5
0
0
50000
100000
150000
200000
250000
NUMBER OF INSERTIONS

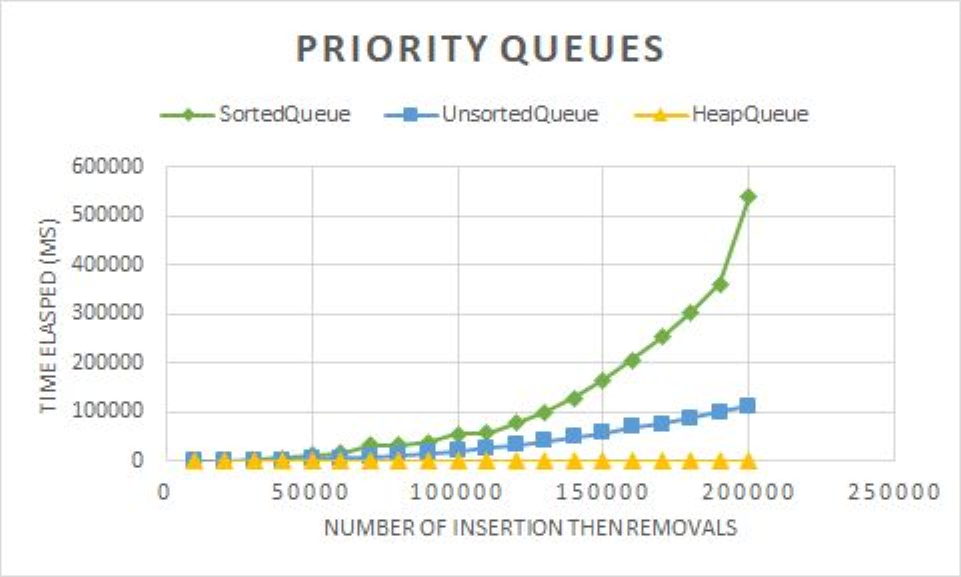

PRIORITY QUEUES
SortedPQ
UnsortedPQ
HeapPQ
600000
500000
400000
300000
200000
100000
0
AXIS TITLE
0
50000
100000
150000
200000
250000
AXIS TITLE

Insertion/Removal Graphs






HEAPQUEUE
60
50
40
30
20
10
0
TIME ELASPED (MS)
0
50000
100000
150000
200000
250000
NUMBER OF INSERTIONS/REMOVALS


PRIORITY QUEUES
SortedQueue
UnsortedQueue
HeapQueue
600000
500000
400000
300000
200000
100000
0
TIME ELASPED (MS)
0
50000
100000
150000
200000
250000
NUMBER OF INSERTION THEN REMOVALS

Results:

While inserting, the UnsortedPQ performed the best in general, which supported my theoretical analysis. This does not depend on input and the graphs support the statement. This is because the UnsortedPQ does not need to perform the upheap algorithm while inserting, as in HeapPQ, nor sort the list as in SortedPQ. While inserting/removing the HeapPQ performed the best in general, which supported my theoretical analysis. This does not depend on input and the graphs support the statement. This is because UnsortPQ has to search, and SortPQ has to sort, which costs a lot of time. In general, HeapPQ is far less costly in terms of time elapsed for both UnsortedPQ and SortedPQ.