REPORT: PA 3 – Priority Queue

Muhammad Taha Haqqani
Section 505

CSCE 221 Cover Page
Programming Assignment #3
Due Date: Wednesday October 30, 11:59pm
Submit this cover page along with your report

First Name: Muhammad Taha   Last Name: Haqqani   UIN: 426004967

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. Stack Overflow	1.
2.	2.	2.	2. geeksForGeeks	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: 11/4/2019

Printed Name (in lieu of a signature): Muhammad Taha Haqqani

Introduction

- Build three different implementations of a Priority Queue ADT in which two of them are based on a list and the third one is array-based
- Test each of the implementations with a list of numbers from a file to check if they sort the numbers and print them in increasing order
- Compare the insertion time, removal time, and sorting time (insertion + removal) for each version of the Priority Queue and discuss the performance in terms of running time, asymptotic analysis, and amortized cost
- Understand the efficiency and use of each implementation and make conclusions based on experimental results

Background

A Priority Queue is a queue in which each element has a priority associated with it. The only difference between a queue and a priority queue is that instead of being "first in first out", values come out based on their priority. An element with a higher priority is dequeued before an element with a lower priority. Typically, every priority queue object has a key associated with it which is used to rank (prioritize) elements. If two elements have the same priority, then the elements are dequeued according to their order. Important operations in a priority queue are insertion, removals, minimum, and maximum. There are two main ways to implement a priority queue: arrays and heaps. There are many applications of a priority queue such as sorting, scheduling systems, traffic lights and anywhere where priority is involved.

Theoretical Analysis

For the priority queue, the insertion and removal operations are being analyzed as their time complexities differ for each implementation. The three different implementations of the PQ lead to three different sorting algorithms which will allow us to compare how each implementation performs in this area. For this experiment, we are only focusing on the insert and remove operations and we will always remove the minimum value in the queue.

The list implementations (Sorted and Unsorted PQ) are implemented using a list which is basically a sequence container that allows insert and remove operations to take place anywhere within the sequence. The list implementation is a naïve approach and is easy to implement since lists allow constant time for insert and erase operations. The heap PQ is a better alternative as we will discuss in the sections below.

<u>Unsorted PQ:</u>

In the Unsorted PQ, elements are added arbitrarily into the list. This is a simple push operation which takes constant time and therefore, for inserting n elements it should only have O(n) time complexity and O(1) amortized time complexity. However, as the name suggests, the unsorted list must be sorted to remove the minimum value in the queue and this takes O(n) for each element. This is because for each remove operation, every element is compared to find the minimum value in the queue and for n number of elements, it will take a total of n comparisons. This means that for n elements, the worst-case running time complexity for n removals will be $O(n^2)$. This provides us a version of selection sort because elements are sorted after they are inserted.

Selection Sort:

The reason why we obtain selection sort from unsorted PQ is because if we insert and remove elements from this PQ, elements are removed in sorted order by repeatedly finding the minimum element. The worst-case happens when we are sorting an array of reversely sorted elements which means every removal will require comparisons to be made with every other element in the queue. The whole sorting process will have a worst case time complexity of $O(n + n^2)$ time which is basically $O(n^2)$ and the amortized time complexity for selection sort will be O(n). The best-case scenario would be when the elements are inserted in sorted order already and in this case the best-time complexity for selection sort will be O(n) as only one comparison needs to be made for each remove operation. The average-case time complexity will also be $O(n^2)$

Advantages:

- Insertions are easy and take constant time
- Easy to implement as list insertions and deletions can happen both at front or back which takes constant time
- Removals in the list implementations do not require moving of elements which is more efficient compared to arrays
- Selection sort is an in-place sorting algorithm which means that no additional temporary storage is required to hold the original list

Disadvantages:

- It has the worst running time complexity O(n) compared to the other two for removing an element
- Worst-case and average-case time complexity for selection sort is $O(n^2)$ which is not the great for sorting large data size
- List implementations require more space as they are non-contiguous and need storage for pointers
- Traversing the list is only done from start to end and direct access to an element cannot be done without traversing the list
- It takes linear time to get the minimum value in the queue

Sorted PQ:

The sorted PQ works by inserting every element in order and then removing elements from the front or back in sorted order. So basically, you pay the cost first during insertions instead of at the end during removals as we do in unsorted PQ. This means that the worst-case time complexity for each insertion operation is O(n) as we need to compare the element to be inserted with every element in the queue. The removal, however, is done in constant time since we are just removing an element either from the front or back depending on the implementation and whether we want to remove the minimum or maximum priority.

Insertion Sort:

Sorting with this implementation will require $O(n^2)$ in the worst case when the elements to be sorted are reversely sorted because every element will require n comparison. For the best-case scenario, when the array to be sorted is already sorted, the time complexity for sorting is O(n) since we only need each element to be compared once. For average-case, however, is still $O(n^2)$ which causes the amortized time complexity to be O(n) for sorting n elements.

Advantages:

- Removals are easy and take constant time
- Easy to implement as list insertions and deletions can happen both at front or back which takes constant time
- Removals in the list implementations do not require moving of elements which is more efficient compared to arrays
- Insertion sort is an in-place algorithm so the space requirement is minimal

- Getting the minimum value only takes constant time

Disadvantages:
- It has the worst running time complexity of O(n) compared to the other two for inserting an element
- Worst-case and average-case time complexity for insertion sort is O(n^2) which is not the great for sorting large data size
- List implementations require more space as they are non-contiguous and need storage for pointers
- Traversing the list is only done from start to end and direct access to an element cannot be done without traversing the list

Heap PQ:

A Heap PQ is a basically a binary tree implementation of the priority queue in which every node has either 0, 1 or 2 children. Since we are making a min Heap, the root element is going to be the smallest element in the tree which means that every parent node must be larger than or equal to its child nodes. Every node has a left child and right child and an array implementation means that we can easily access every node of the tree in constant time. This is the heap order property which is maintained by up-heaping during insertions and down-heaping during removals. In order for the heap to work efficiently, it is important to make sure that the heap order is always maintained and the tree is always a complete binary tree which means that nodes are filled from left to right and each level has all its nodes filled except the bottom level of the tree. This allows us to take advantage of the logarithmic performance of the heap.

Using an array for the heap allows us to efficiently traverse through the tree because we have access to parent node, left node and right node for each node in the tree. Insertions are always done at the outermost part of the tree which is basically the end of the array. If the insertion of an element violates the heap order property, then we perform upheap in which the inserted node is exchanged with its parent node and we continue this until every parent node in the tree is greater than or equal to its left and right child. Since the heap is a balanced tree, the height of the tree is log n where n is the number of nodes in the tree. This means that the worst-case for insert will happen when we need to visit every level of the tree to maintain heap property and this will therefore require O(log n) time complexity. For n insertions this will require a total running time complexity of O(n log n) and an amortized time of O(log n). Removals in the heap are done at the root and the outermost leaf of the tree becomes the new root. If the heap order property is violated, we perform down-heap in which the parent node is compared with its children and is replaces with the smaller child. This will again require O(log n) time in the worst-case scenario when every node must be visited. Therefore, the total running time for inserting and removing is same for insertions are removals in the worst-case. The best-case scenario for insertions and removals happen when there is no violation of the heap order and this will only require O(n) time for n operations.

Heap Sort:

Heap sort is much more efficient compared to the previous two PQ implementations. This is because the running time for insert and remove are much faster. For sorting an array of n numbers, the heap sort only takes O(n log n) time because inserts take O(n log n) time and removals take O(n log n) time for n numbers and the total time complexity becomes O(2n log n) which is just O(n log n). This will be same for average-case and best-case unless if we allow duplicates in which elements with same value are added and that will result in a running time complexity of O(n) . The amortized time complexity for sorting n numbers for worst-case would then result in O(log n).

Advantages:
- Best running time complexity for sorting compared to the other two implementations
- The guaranteed logarithmic performance of heap sort is great to use where time is critical and data is large

- Arrays are contiguous in memory which makes it perfect for accessing individual indexes and memory usage is minimal

Disadvantages:

- It is an unstable sort which means that the order of elements changes since rearrangement becomes necessary whenever the heap order is violated
- Difficult to implement because all edge cases must be taken into account

	Insert	Remove	PQ Sort
Unsorted PQ	O(1)	Best case: O(1) Worst case: O(n)	Selection Sort O(n^2)
Sorted PQ	Best case: O(1) Worst case: O(n)	O(1)	Insertion Sort O(n^2)
Heap PQ	Best case: O(n) Worst case: O(log n)	Best case: O(n) Worst case: O(log n)	Heap Sort Worst: O(n log n) Best: O(n) (duplicates)

Experimental Setup.

1. **Machine specification**

- HP ENVY Notebook
    - Processor: Intel(R) Core (TM) i7-7500U-CPU @2.70 GHz 2.90 GHz
    - RAM: 8.00 GB
    - System Type: 64-bit Operating System

2. **Implementation:**

   A few improvements and additions were made to improve the performance and efficiency of each version of the PQ. The list STL was used for unsorted and sorted PQ for simplification purposes as inbuilt functions were used from the STL library such as pop_back() and push_front(). The STL list is a doubly linked list implementation built so that we can easily use it without having to implement the whole doubly linked list class. The choice of STL list allowed us to add and remove elements from the front and the back of the list which made the code more concise and easier to follow through.

   For the Heap PQ, a vector was used instead of a primitive array. Vector can grow and shrink dynamically and this was useful rather than manually resizing the array every time it's filled. Vectors also have many helpful inbuilt functions such as push_back() which helped ease the implementation process. The best part of vectors is that we can declare vectors without mentioning the size unlike arrays. This allowed us to have less memory usage since we only dealt with memory that was being used to store elements.

   A few additions:

- Print() function in the Heap PQ to visualize the array and check if the heap is working properly
- Private helper functions such as getRightChild() and hasParent() to avoid code clutter and make it easier to understand rather than using formulas for every conditional statement
- Main was used to test every implementation's function and ensure that it is working properly

3. Input Generation and Testing

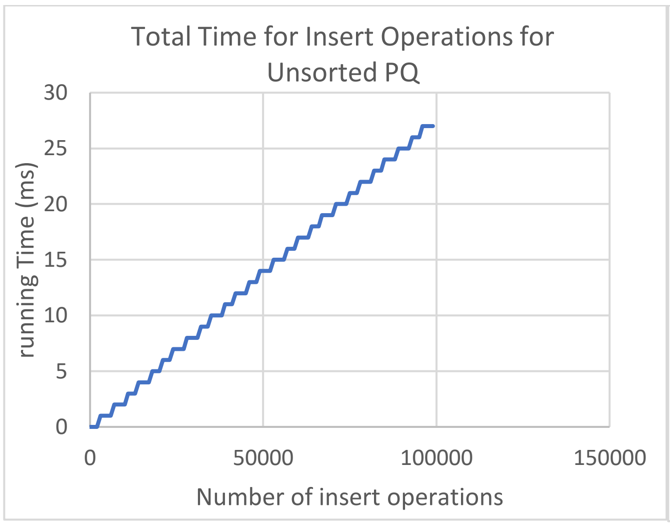
Three different input testing was done for each version of the PQ. To effectively measure the time complexity for insert, removals and sorting (inserts + removals), these functions were tested separately for each implementation. For each implementation:

- 100k random numbers were inserted and time was measured after every 1000 inserts which allowed us to measure time at 100 different intervals
- 100k random numbers were removed and time was measured for each 1000 removal operations
- Sorting was done on initially 100 random numbers and the input size was incremented by 100 after every sorting cycle to measure the time taken to sort for increasing size of input
- 3 repetitions were made for every testing unit and an average was taken, removing any anomalous or abnormal values
- time was measured by using the chrono library in which time was recorded in milliseconds
- testing was done in the same environment making sure the same editor, compiler and programs are used to avoid any discrepancies during testing

Experimental Results

For each implementation, time was measured for insertions, removals and sorting 100k numbers. Graphs were made in excel for running time against the number of elements inserted/removed and then a graph for amortized analysis to compare the results for each implementation.

<u>Unsorted PQ:</u>



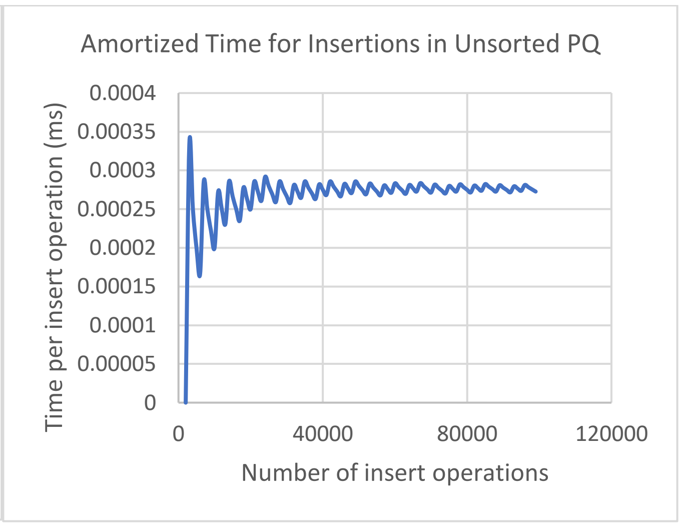


Fig: Analysis on Insert for Unsorted PQ

The experiments supported our theoretical analysis for insert operation in unsorted PQ. We obtained linear time for inserting 100k operations and constant amortized time. For each insert we can conclude that the time complexity is O(1) and for n push operations it will be O(n)

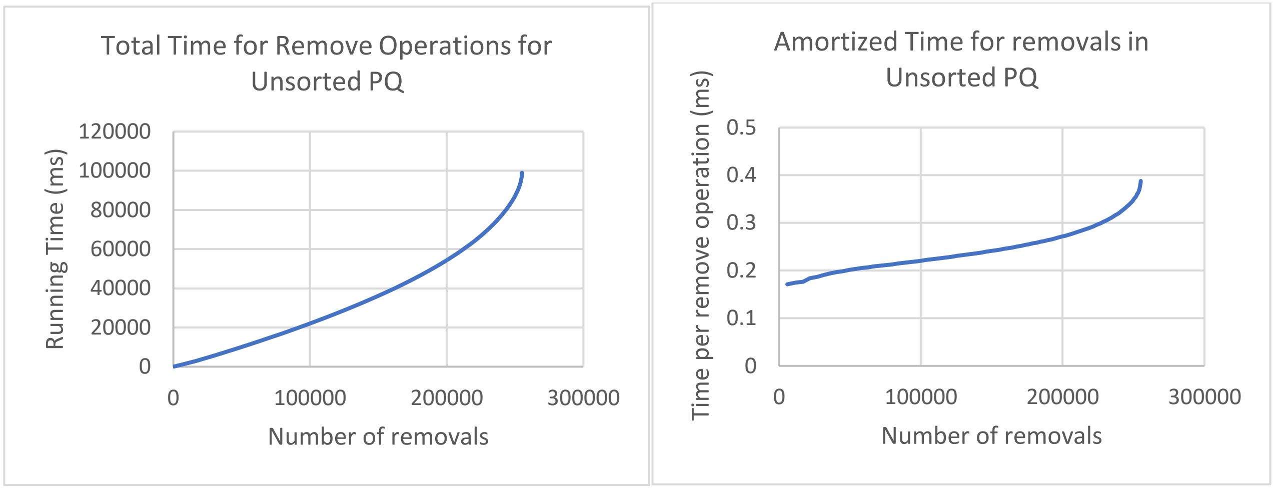


Fig: Analysis on Remove for Unsorted PQ

The experiments supported our theoretical analysis for remove operation in unsorted PQ. We obtained quadratic time for removing 100k operations and linear amortized time. For each removal we can conclude that the time complexity is O(n) and for n remove operations it will be O(n^2). The slight tilt at the end of the graph can be explained by the fact that the values to be removed can be deeper in the list and would take longer since the size of the list is incredibly large.

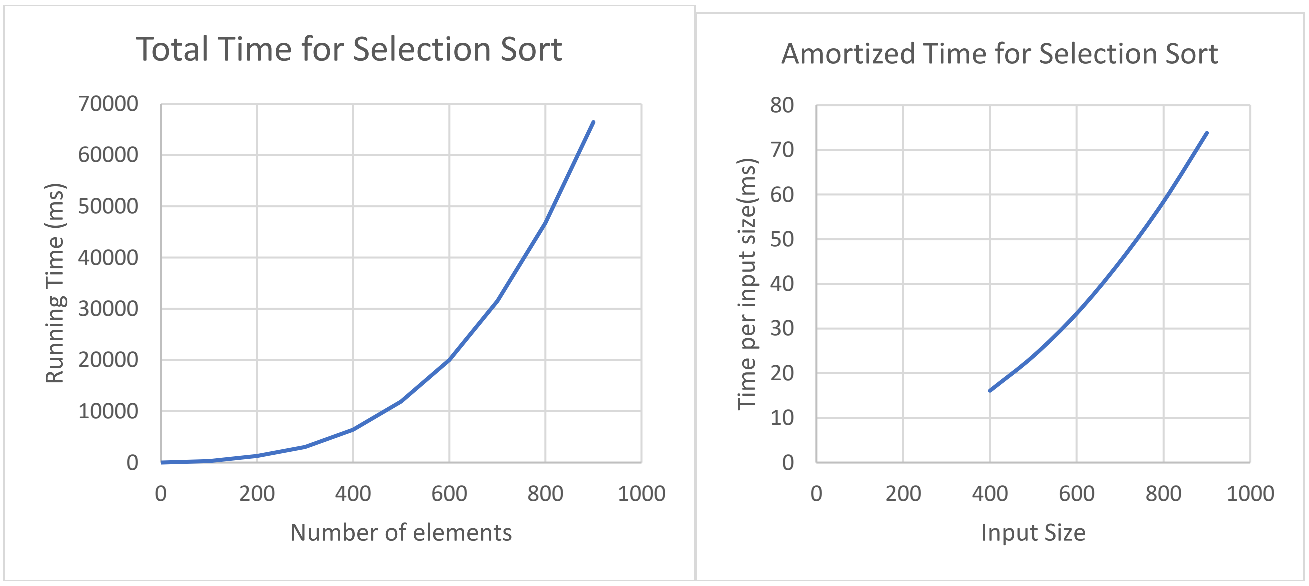
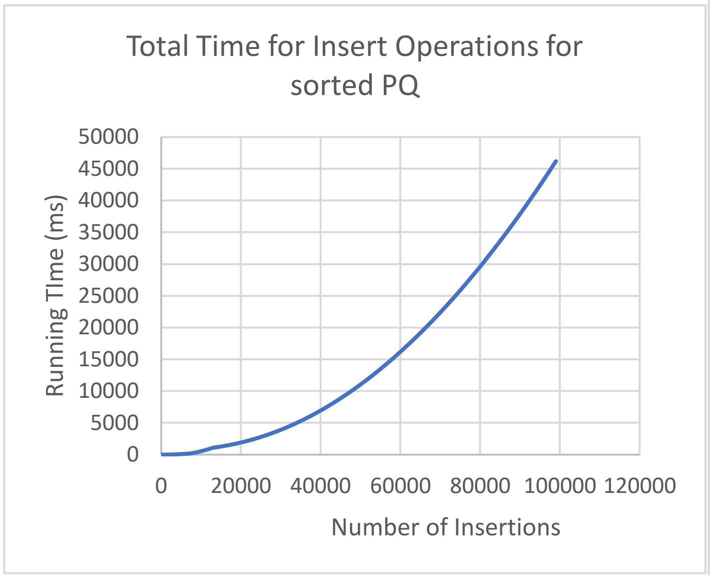


Fig: Analysis on Selection sort using Unsorted PQ

As expected, the running time complexity for sorting using Unsorted PQ was O(n^2) and the amortized time is O(n) for each input size of n. We obtained these graphs by sorting 100 random numbers and increasing the input by 100 for each sorting cycle.

Sorted PQ:



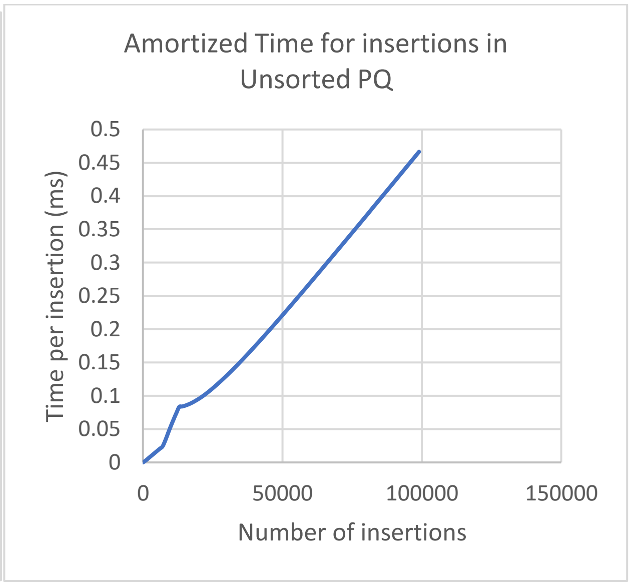
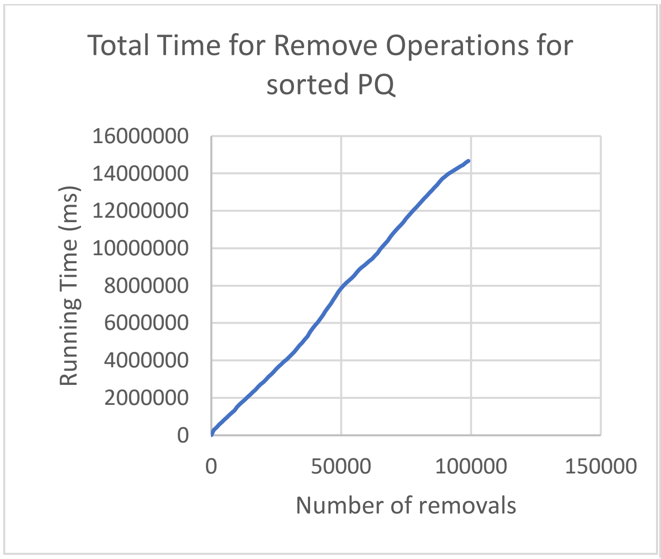


Fig: Analysis for Insert operation in Sorted PQ

For sorted PQ, we can see that the insert operation has a quadratic graph which agrees with our theoretical analysis that the time complexity for n insertions is O(n^2). The amortized cost for each insert operation is O(n) which is also what we expected from unsorted PQ. The slight changes in the beginning of the graph for amortized time is due to insertions of completely random numbers which can take slightly long if the number to be inserted is near the end of the list.



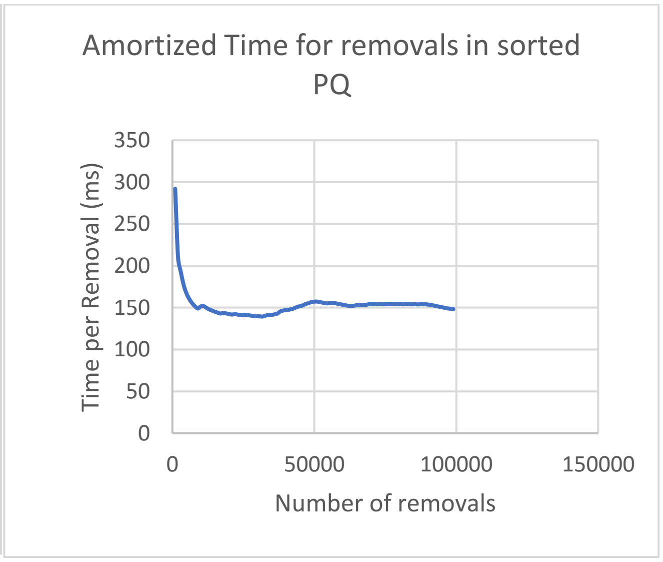


Fig: Analysis for Remove operation in Sorted PQ

The experiments supported our theoretical analysis for remove operation in sorted PQ. We obtained linear time for removing 100k operations and constant amortized time. For each removal we can conclude that the time complexity is O(1) and for n remove operations it will be O(n).

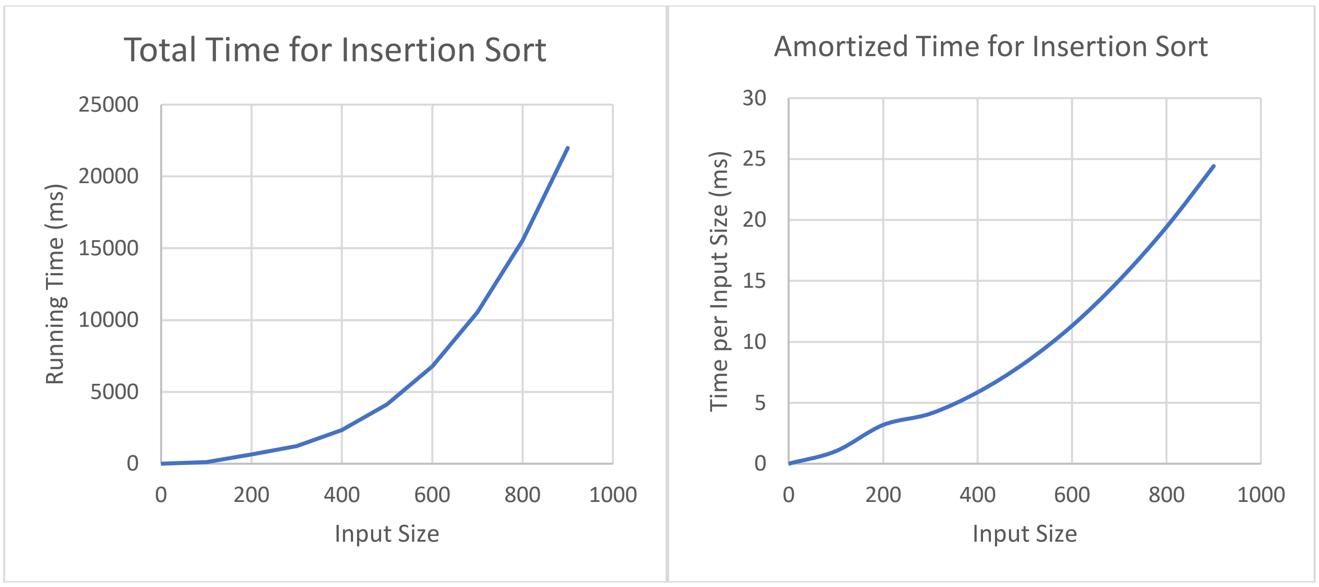
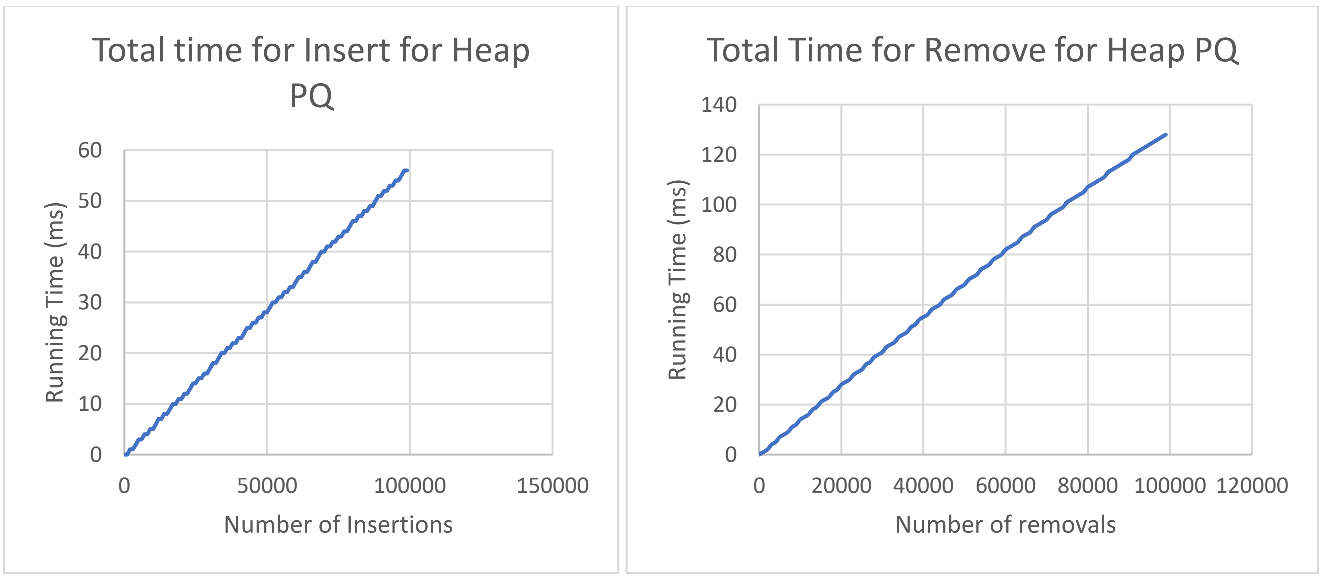


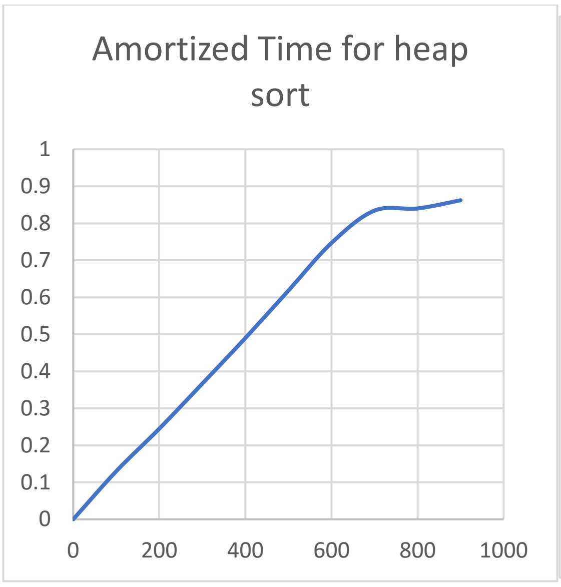
Fig: Analysis on insertion sort using sorted PQ

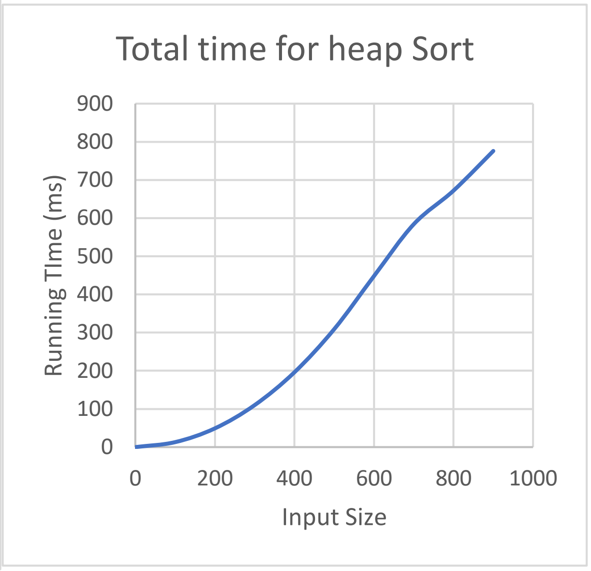
The running time complexity for insertion sort was clearly as we expected it to be. For sorting 100k numbers, the sorting time was quadratic just like selection sort. This time complexity of O(n^2) was also supported by the linear graph of the amortized time complexity for unsorted PQ. The graph for amortized time complexity for insertion sort was close to quadratic because the total time for sorting is O(n^2 +n) which is close to O(n^2).

Heap PQ:



The running time complexity for 100k insertions and removals for Heap PQ is O(n log n) which is a graph close to linear graph. We obtained results that supported our theoretical analysis. There is no need to compare the amortized cost because it will essentially be O(log n) and it will be close to the graph we obtained.





The heap sort was expected to run in O(n log n) time complexity. The graph for the total running time of heap sort clearly shows that is logarithmic and supports our theoretical analysis. The amortized time complexity was also what we expected to see which is O(log n).

Conclusion

Based on the results of our experiment, we can conclude that heap PQ was the most efficient implementation out of the three. The Heap sort algorithm had the fastest running time complexity and it was very efficient and consistent. Although selection sort and insertion sort did not provide the best time complexity, they are still more practical for small data sets because they will occupy less space. If the input size is large, the Heap PQ is the most practical implementation to use simply because it is faster and more efficient. The list implementations were easier to implement than the heap PQ because there were more edge cases in the heap PQ. The slight differences in the theoretical analysis and experiment were due to the fact that random numbers were being inserted which meant that there was a combination of worst-case, best-case and average-case.

Improvements

- Bottom up construction for heap:

A heap can be built in linear time from an arbitrarily sorted array. This can be done by swapping items, ending up with an algorithm requiring at most $kn+c$ swaps, where n is the number of items in the array and k and c are small constants.
This algorithm starts by examining the last internal node of the tree and upheaping if the chidren of the node are smaller than the node. This is repeated for every other internal node until we reach the root.

- Fibonacci Heap

A Fibonacci heap is a collection of trees in which trees can have any shape and all tree can be single nodes too. A pointer is maintained to a minimum value and roots are connected using a circular doubly linked list. The implementation is a bit complicated but the Fibonacci heap has better time complexity than a binary heap.

- Experimenting manually

One of the reasons why there were some discrepancies was because of using random insertions. If we make the best-case, worst-case and average case sets ourselves, it will be much more efficient to prove our theoretical analysis. This is because random values do not allow us to know exactly what scenario is represented. For the next experiment it would be better to validate the results first using sets of values that will give us information about the input such as size and best-case/worst-case.