# Assignment 4: Heap Data Structures: Implementation, Analysis, and Applications

## Heapsort Implementation and Analysis

### 1. Implementation

### 2. Analysis of Implementation

Time Complexity:

Worst, Average, and Best Case: The time complexity of Heapsort is O(n log n) for all the cases.

Building the max-heap takes O(n) time because each level of nodes requires progressively less work. Once the heap is built, each extraction operation (swapping the root with the last element and heapifying) takes O(log n). With n extractions, the overall complexity remains O(n log n)

Space Complexity:

Heapsort is an in-place algorithm, meaning it does not require additional storage for the array.

The space complexity is O(1) for auxiliary data, making it efficient in terms of memory usage.

There is minimal overhead aside from the array itself, as Heapsort uses a fixed amount of memory for pointers and indices during heapification.

### 3. Comparison

Heapsort Time:

Array with sorted elements: 0.00871 seconds

Array with Reverse sorted elements: 0.00298 seconds

Random Array: 0.00366 seconds

Randomized QuickSort:

Array with sorted elements: 0.00827 seconds

Array with Reverse sorted elements: 0.01770 seconds

Random Array: 0.00330 seconds

Comparison with Quicksort

Quicksort:

Best and Average Case: O(n log n), Worst Case: O(n^2) if the pivot selection is poor (e.g., already sorted arrays without randomization).

Memory: In-place but not stable, with an average space complexity of O(log n) due to recursion.

Practical Results: Often faster than Heapsort for average-case inputs due to lower constant factors and optimized partitioning.

Heapsort:

Time Complexity: O(n log n) for all cases.

Memory: O(1) auxiliary space, as it is in-place.

Practical Results: Heapsort is typically slower than Quicksort on average due to higher constant factors in O(n log n)

Observed Results and Theoretical Analysis

Empirical tests often reveal that extremely close results on tests using 760 elements arrays, with the exception of Array with Reverse sorted elements which presents a struggle for the Randomized QuickSort algorithm compared to Heapsort.

## Priority Queue Implementation and Applications

## Part A: Priority Queue Implementation

### 1. Data Structure:

### 2. Core Operations:

In a real-world application like a task scheduling system, tasks with the highest urgency (lowest priority values) need immediate processing. A min-heap is particularly effective here as it allows us to retrieve the most urgent task in O(log n) time while efficiently handling insertions and priority adjustments. Maintaining an appropriate priority order ensures that critical tasks are processed on time, enhancing system reliability and response time for high-priority demands.

To implement a priority queue, we will use a binary heap represented by an array. The array based heap is useful because it provides an efficient way to perform heap operations while maintaining contiguous memory allocation, which simplifies insertion and deletion. Additionally, the array-based structure provides an implicit tree structure, where each element's children can be accessed in constant time.

For this implementation, a min-heap will be used, meaning tasks with the lowest priority values meaning the most urgent tasks, will be served first. This is typical for scheduling algorithms in operating systems and real time applications.

The task class to store the task’s ID, priority, arrival time, deadline.

The priority queue will have the following core operations: insert, extract\_min, decrease\_key, and is\_empty.

Insert Operation

This operation adds a new Task to the heap while maintaining the heap property. The task is added to the end of the array, then "bubbled up" to its correct position.

Time Complexity: O(log n), as it requires adjusting the heap after insertion.

Extract Min Operation

This operation removes and returns the task with the lowest priority from the heap (the root). The last element is moved to the root, and the heap is restructured by "bubbling down" this element.

Time Complexity: O(log n), as it requires adjusting the heap to restore the min-heap property.

Decrease Key Operation

This operation changes the priority of a given task and repositions it accordingly within the heap to maintain the heap property.

Time Complexity: O(log n), as it may need to adjust the heap by moving the element up or down.

Is Empty Operation

A simple check to see if the heap contains any tasks.

Time Complexity: O(1), as it only checks the length of the array.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). The MIT Press.

Weiss, M. A. (2011). *Data Structures and Algorithm Analysis in Java* (3rd ed.). Pearson.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.