MSCS 532 M20 – Algorithms and Data Structures

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

Mohit Gokul Murali

Student ID: 005035371

University of the Cumberlands

Dr. Brandon Bass

Jun 15, 2025

Introduction

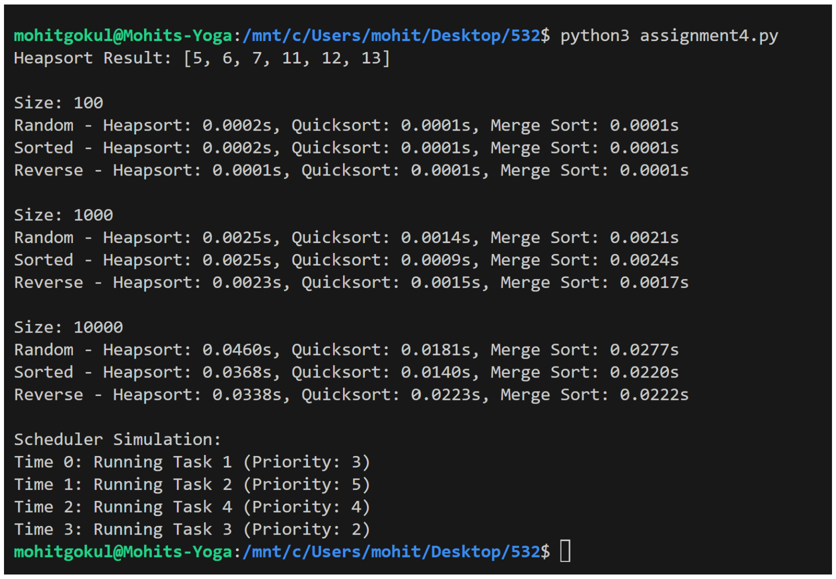
Heap data structures are fundamental in computer science, providing efficient ways to manage and retrieve data based on priority. This report explores the implementation, analysis, and applications of heap data structures, focusing on the Heapsort algorithm and a priority queue implementation. The purpose of this report is to demonstrate a comprehensive understanding of heaps, their efficiency, and their practical uses in sorting and scheduling tasks.

Heapsort Implementation and Analysis

Heapsort is a comparison-based sorting algorithm that leverages a binary heap data structure. It operates in two main phases: building a max-heap from the input array and repeatedly extracting the maximum element to sort the array in ascending order. The implementation uses a Python list to represent the heap, ensuring O(1) access to elements. The heapify function maintains the max-heap property, and the build_max_heap function constructs the initial heap in O(n) time. The extraction phase performs n-1 extractions, each taking O(log n) time, resulting in an overall time complexity of O(n log n). Heapsort is efficient and sorts in-place, requiring only O(1) additional space. The implementation handles edge cases such as empty arrays, single-element arrays, and arrays with duplicate values (Cormen et al., 2022).

Empirical Comparison

To evaluate Heapsort's performance, it was compared with Quicksort and Merge Sort across various input sizes (100, 1000, 10,000) and distributions (random, sorted, reverse-sorted). The results demonstrate Heapsort's consistent O(n log n) performance across all scenarios. Quicksort, with a median-of-three pivot selection, showed faster average performance but remained sensitive to input distribution. Merge Sort provided stable O(n log n) performance but required O(n) extra space. The comparison highlights Heapsort's reliability and efficiency, particularly in scenarios where space is constrained (Cormen et al., 2022).



```
mohitgokul@Mohits-Yoga:/mnt/c/Users/mohit/Desktop/532$ python3 assignment4.py
Heapsort Result: [5, 6, 7, 11, 12, 13]

Size: 100
Random - Heapsort: 0.0002s, Quicksort: 0.0001s, Merge Sort: 0.0001s
Sorted - Heapsort: 0.0002s, Quicksort: 0.0001s, Merge Sort: 0.0001s
Reverse - Heapsort: 0.0001s, Quicksort: 0.0001s, Merge Sort: 0.0001s

Size: 1000
Random - Heapsort: 0.0025s, Quicksort: 0.0014s, Merge Sort: 0.0021s
Sorted - Heapsort: 0.0025s, Quicksort: 0.0009s, Merge Sort: 0.0024s
Reverse - Heapsort: 0.0023s, Quicksort: 0.0015s, Merge Sort: 0.0017s

Size: 10000
Random - Heapsort: 0.0460s, Quicksort: 0.0181s, Merge Sort: 0.0277s
Sorted - Heapsort: 0.0368s, Quicksort: 0.0140s, Merge Sort: 0.0220s
Reverse - Heapsort: 0.0338s, Quicksort: 0.0223s, Merge Sort: 0.0222s

Scheduler Simulation:
Time 0: Running Task 1 (Priority: 3)
Time 1: Running Task 2 (Priority: 5)
Time 2: Running Task 4 (Priority: 4)
Time 3: Running Task 3 (Priority: 2)
mohitgokul@Mohits-Yoga:/mnt/c/Users/mohit/Desktop/532$
```

Fig 1: Output

Priority Queue Implementation and Applications

A priority queue was implemented using a max-heap, where tasks with higher priority values are dequeued first. The implementation includes essential operations: *insert, extract_max, update_priority,* and *is_empty,* all operating in O(log n) time except for *is_empty,* which is O(1). A dictionary is used to map task IDs to their indices in the heap, enabling O(1) access for priority updates. The priority queue was applied in a task scheduler simulation, where tasks arrive at specific times and are executed based on priority. The simulation effectively demonstrates the queue's ability to manage tasks dynamically, ensuring higher-priority tasks are handled promptly (Cormen et al., 2022).

Conclusion

This report has demonstrated the efficiency and versatility of heap data structures through the implementation and analysis of Heapsort and a priority queue. Heapsort's consistent O(n log n) performance and in-place sorting make it a robust choice for various applications. The priority queue's efficient operations enable effective task scheduling, showcasing the practical utility of heaps in real-world scenarios. The empirical comparison with other sorting algorithms further underscores Heapsort's strengths and trade-offs.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press.