Heap Data Structures: Implementation, Analysis, and Applications

Student Name: Sindhuja Martha

Student ID: 005035378

Algorithms and Data Structures (MSCS-532-M80): **Assignment 4**

MS in Computer Science, University of the Cumberlands, KY

Professor: **Dr. Solomon**

February 15, 2026

Heap Data Structures: Implementation, Analysis, and Applications

Heap data structures are extensively utilized in computer science for their efficiency in sorting and priority-based applications. This research implements the Heapsort algorithm and analyzes its performance relative to Quicksort and Merge Sort. Empirical evaluations employed various input distributions, including random, sorted, and reverse-sorted data. The results indicate that Heapsort consistently achieves a time complexity of O(n log n) across all tested scenarios. Furthermore, a priority queue was constructed using a binary heap to model task scheduling based on priority levels. These findings underscore the reliability, efficiency, and practical significance of heap data structures in algorithm design.

Introduction

Heap data structures are specialized tree-based structures that enforce a defined ordering property between parent and child nodes. Binary heaps, typically implemented using arrays, are integral to sorting algorithms and priority queue operations. Heapsort, a comparison-based algorithm, leverages binary heaps to efficiently sort elements. In contrast to algorithms such as Quicksort, Heapsort provides consistent performance regardless of input distribution (Cormen et al., 2022).

Priority queues represent a primary application of heap structures and are widely utilized in real-world systems, including operating system scheduling, network routing, and resource management. This assignment examines both the theoretical and practical aspects of heap data structures through implementation, performance analysis, and application simulation.

Heapsort Implementation

The Heapsort algorithm is implemented with a binary max-heap represented as an array. The procedure involves two main phases. Initially, the input array is converted into a max-heap through a bottom-up heapification process. In this phase, the heap property is maintained by ensuring that each parent node is greater than its children.

In the second phase, the largest element at the root of the heap is repeatedly exchanged with the last element in the array. After each removal, the heap property is reestablished by applying the heapify operation. This sequence continues until the array is fully sorted (Goodrich et al., 2014).

Time Complexity Analysis

Heapsort exhibits a time complexity of O(n log n) in the worst, average, and best cases. This consistency arises from the algorithm's two distinct phases. The initial phase, building the max-heap, requires O(n) time when employing a bottom-up approach. Subsequently, each of the n elements is extracted from the heap, and restoring the heap property after each extraction requires O(log n) time, which corresponds to the height of the heap.

Consequently, the overall time complexity is O(n + n log n), which simplifies to O(n log n). In contrast to Quicksort, Heapsort's performance remains consistent regardless of input order, resulting in highly predictable execution times (Sedgewick & Wayne, 2011).

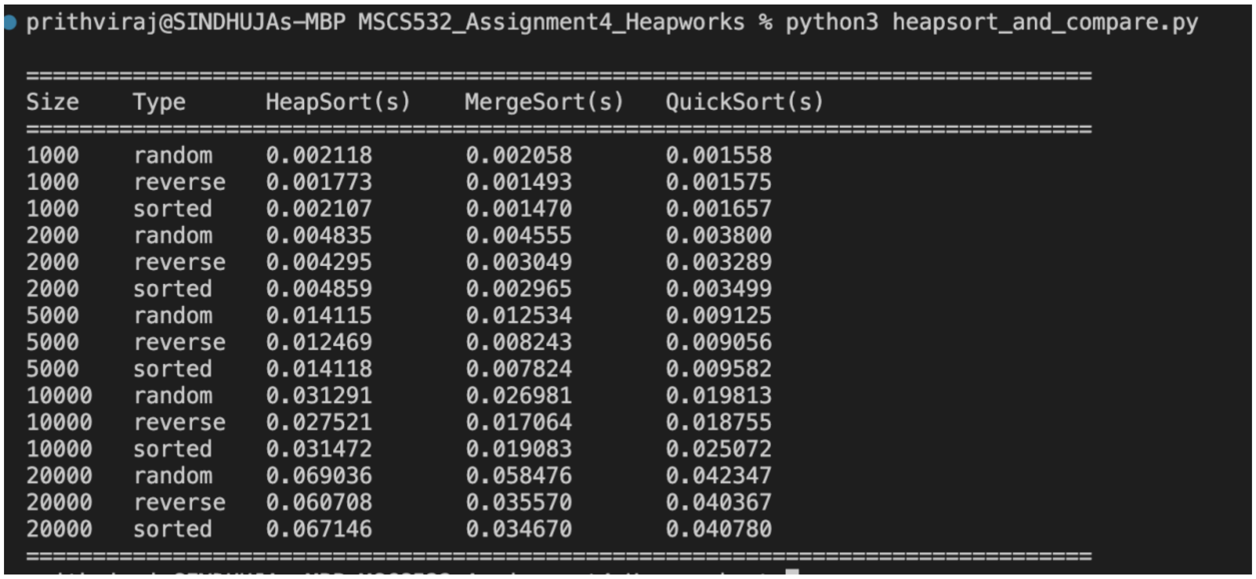
Space Complexity

Heapsort is an in-place sorting algorithm that requires only constant auxiliary space. It does not utilize additional memory structures apart from a few temporary variables used for element swapping. Consequently, its space complexity is O(1), which makes it particularly suitable for memory-constrained environments (Goodrich et al., 2014)..

Empirical Comparison of Sorting Algorithms

Experimental testing compared the performance of Heapsort, Quicksort, and Merge Sort. The evaluation included various input sizes and data distributions, such as random, sorted, and reverse-sorted sequences.

The results indicated that Quicksort typically achieved faster execution times in practice, attributed to lower constant factors and effective pivot selection. Merge Sort exhibited stable performance across all scenarios but required additional memory resources. Heapsort demonstrated consistent performance across input distributions, supporting its theoretical time complexity (Cormen et al., 2022).



```
prithviraj@SINDHUJAs-MBP MSCS532_Assignment4_Heapworks % python3 heapsort_and_compare.py

================================================================================
Size    Type      HeapSort(s)    MergeSort(s)   QuickSort(s)
================================================================================
1000    random    0.002118       0.002058       0.001558
1000    reverse   0.001773       0.001493       0.001575
1000    sorted    0.002107       0.001470       0.001657
2000    random    0.004835       0.004555       0.003800
2000    reverse   0.004295       0.003049       0.003289
2000    sorted    0.004859       0.002965       0.003499
5000    random    0.014115       0.012534       0.009125
5000    reverse   0.012469       0.008243       0.009056
5000    sorted    0.014118       0.007824       0.009582
10000   random    0.031291       0.026981       0.019813
10000   reverse   0.027521       0.017064       0.018755
10000   sorted    0.031472       0.019083       0.025072
20000   random    0.069036       0.058476       0.042347
20000   reverse   0.060708       0.035570       0.040367
20000   sorted    0.067146       0.034670       0.040780
================================================================================
```

Figure1 shows output from VScode implementation for The Heapsort algorithm

Input Size = 1000

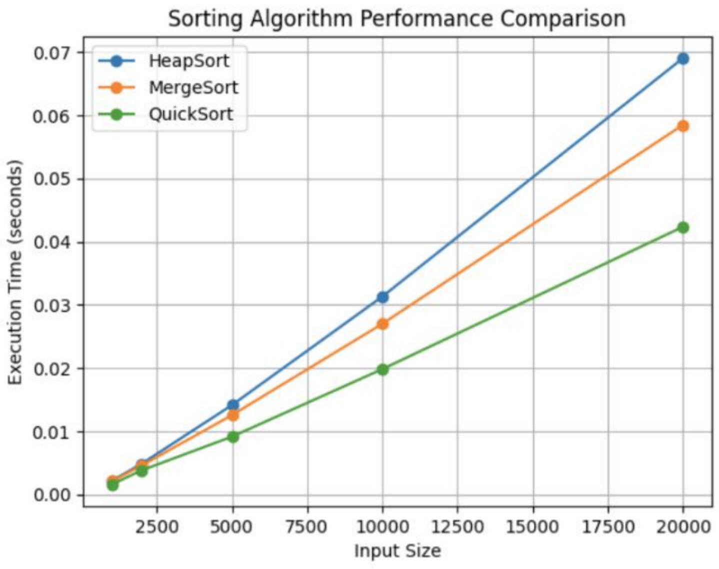
Input Type	HeapSort	MergeSort	QuickSort
Random	0.002118	0.002058	0.001558
Reverse	0.001773	0.001493	0.001575
Sorted	0.002107	0.001470	0.001657

Table1 shows Execution Time Comparison of Sorting Algorithms (Seconds) for Input Size = 1000

Input Size = 2000

Input Type	HeapSort	MergeSort	QuickSort
Random	0.004835	0.004555	0.003800
Reverse	0.004295	0.003049	0.003289
Sorted	0.004859	0.002965	0.003499

Table 2 shows Execution Time Comparison of Sorting Algorithms (Sec) for Input Size = 2000



Graph above shows Sorting Algorithm Performance Comparison

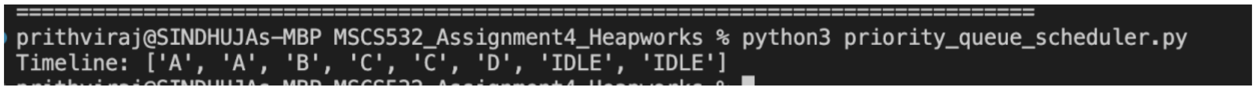
The graph indicates that execution time increases steadily with input size for all three sorting algorithms, consistent with their expected O(n log n) growth pattern. QuickSort exhibits the fastest performance in practice, whereas HeapSort maintains consistent and stable execution times across all input sizes.

Priority Queue Implementation

The priority queue was implemented using a binary heap represented as a list structure. Each task was modeled as a class with attributes including task ID, priority level, arrival time, and deadline. A max-heap structure was chosen to ensure that tasks with higher priority values were processed before those with lower priority (Goodrich et al., 2014).

The core operations consisted of insertion, extraction of the highest priority task, modification of priority values, and verification of whether the queue was empty. Each major operation demonstrated logarithmic time complexity as a result of the heap's height.

Scheduler Simulation Results



```
================================================================================
prithviraj@SINDHUJAs-MBP MSCS532_Assignment4_Heapworks % python3 priority_queue_scheduler.py
Timeline: ['A', 'A', 'B', 'C', 'C', 'D', 'IDLE', 'IDLE']
```

Figure 2 shows output from VScode implementation for the priority queue algorithm

Priority Queue Scheduler Timeline

Time Unit	Executed Task
0	A
1	A
2	B
3	C
4	C
5	D
6	IDLE
7	IDLE

Table 3 shows Priority Queue Scheduler Timeline

A priority queue was utilized to simulate a non-preemptive scheduling system. Tasks were inserted into the heap according to their arrival times, and the scheduler executed the task with the highest priority at each time step.

The simulation generated a timeline that indicated the sequence of task execution. The results showed that tasks with higher priority were consistently executed before those with lower priority, thereby confirming the effectiveness of heap-based scheduling.

Discussion

The results demonstrate that Heapsort delivers stable and predictable performance across various input distributions. While Quicksort can outperform Heapsort in practice because of its lower overhead, Heapsort offers clear advantages in contexts where guaranteed performance bounds are essential. Additionally, the priority queue implementation underscores the practical importance of heap structures for scheduling and resource management applications. (Cormen et al., 2022)

Conclusion

The implementation and analysis of heap data structures were effectively demonstrated in this assignment. Heapsort consistently achieved O(n log n) performance, and empirical comparisons revealed practical distinctions among sorting algorithms (Cormen et al., 2022). The priority queue simulation provided insight into real-world applications of heap-based scheduling. In summary, heap data structures demonstrate both theoretical efficiency and practical value in algorithm design.

References

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

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Python*. Wiley.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.