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

Gregory W. Renteria Villar

Student ID: 005039058

School of Computer and Information Sciences, University of the Cumberlands

MSCS-532-M20: Algorithms and Data Structures

Dr. Brandon Bass

June 15, 2025

Heap Data Structures: Implementation, Analysis, and Applications

Heapsort Implementation and Analysis

Heapsort is a comparison-based sorting algorithm that builds a binary heap (typically a max-heap for ascending sort) and repeatedly extracts the maximum element to sort the array. The time complexity of Heapsort in the worst, average, and best cases is all $O(n \log n)$. This consistent performance arises from the algorithm's two main phases:

1. Heap Construction: This phase transforms the input array into a max heap. Building a heap from an unsorted array takes $O(n)$ time, not $O(n \log n)$ as might be assumed. This is because the time to heapify nodes closer to the leaves is much smaller, and the aggregated work across all nodes can be bound by a linear sum. Specifically, the cost is: $\sum_{i=0}^{\log n} \frac{n}{2^i} \cdot i = O(n)$. Hence, this phase is linear in time complexity.
2. Sorting (Extraction): In the second phase, the root (maximum element) is repeatedly swapped with the last element of the heap, the heap size is reduced, and the root is heapified to maintain the max heap property. This heapify operation takes $O(n \log n)$ time and is performed n times, resulting in an overall time of $O(n \log n)$ for this phase.

Since the heap structure and the number of operations remain the same regardless of the initial arrangement of the elements (already sorted, reverse sorted, or random), the time complexity remains $O(n \log n)$ in the worst, average, and best cases. Unlike Quicksort, which depends on the partition quality and can degrade to $O(n^2)$, Heapsort is more stable in performance.

Heapsort is an in-place sorting algorithm and does not require any additional memory for arrays or recursion stacks, unlike merge sort or quicksort. It maintains the heap structure within the original array. Thus, the space complexity is $O(1)$, aside from a small constant amount of

additional space for index variables and swaps.

The only overhead involved is the non-recursive but iterative heapify process, which may introduce slightly more instruction-level overhead compared to recursive sorting algorithms. However, this is generally negligible in terms of both space and runtime, especially in large scale sorting tasks where $O(n \log n)$ performance and $O(1)$ space and highly advantageous.

Empirical Comparison of Heapsort, Quicksort, and Merge Sort

To empirically compare the performance of Heapsort with Quicksort and Merge Sort, we measured their execution time across various input sizes (e.g. 1,000 to 1,000,000 elements) and input distributions: random, already sorted, and reverse sorted arrays. The implementation were standard, in-place Quicksort using median-of-three pivot selection, top-down recursive Merge Sort and in-place Heapsort.

For random inputs, Quicksort consistently outperformed both Merge Sort and Heapsort. This is expected, as Quicksort has excellent average-case performance $O(n \log n)$ and benefits from good cache locality. Merge Sort and Heapsort were close in performance, with Merge Sort slightly faster due to its consistent divide-and-conquer behavior and efficient merge step. Heapsort, while still $O(n \log n)$, had more overhead from repeated heapify operations and less efficient memory access patterns, making it slower.

On already sorted data, Quicksort's performance deteriorated when not using a randomized or median pivot (as expected), often degrading toward $O(n^2)$. However, using a median of three pivot or random pivot selection restored its performance to $O(n \log n)$. Merge Sort maintained steady performance regardless of input distribution because it doesn't rely on element order. Heapsort also performed consistently across input types, confirming its insensitivity to input order a major theoretical strength.

With reverse-sorted arrays, similar trends were observed. Merge Sort remained robust and fast, Quicksort needed pivot optimization to avoid performance drops, and Heapsort showed uniform execution time close to Merge Sort but still behind in practice.

Priority Queue Implementation and Algorithms

Data Structure Justification

The chosen data structure for implementing the Priority Queue is a binary heap, specifically managed via a dynamic array (Python list). This is a well-suited choice due to a balance between ease of implementation, theoretical efficiency, and practical performance.

Binary heaps provide logarithmic time complexity for essential operations like insertion $O(n \log n)$, extraction of the highest or lowest priority element $O(n \log n)$, and priority updates (increase/decrease key, $O(n \log n)$). Their array-based representation eliminates the need for explicit pointer management, making implementation straightforward while still supporting efficient indexing for parent and child nodes. This enables compact and fast memory access compared to more complex structures like Fibonacci heaps, which offer better theoretical bounds for some operations but are significantly harder to implement and maintain.

Additionally, binary heaps offer predictable performance, which is important in scheduling and real-time systems. The constant factors in the time complexity are small, and no auxiliary data structures (e.g., auxiliary queues or trees) are needed to maintain heap order, contributing to simplicity and reliability.

Justification for Using Max Heap

In the context of the implemented scheduling algorithm, we chose a max heap. This decision is driven by the need to always retrieve the task with the highest priority value. In many real-world systems, such as real-time schedulers or priority-based CPU scheduling, tasks with

higher priority values should be executed before lower-priority ones. Thus, max-heap naturally supports this policy by ensuring that the task with the maximum priority resides at the root and can be extracted in $O(\log n)$ time.

If the scheduling algorithm had favored tasks with the lowest priority values (e.g., for shortest-job-first scheduling, earliest deadline first, or minimizing cost), min-heap would be more appropriate. But in our case, where priority is directly interpreted as "importance" or "urgency," using a max-heap aligns directly with the scheduling goal, always servicing the most critical task next.

Time Complexity Analysis of Core Priority Queue Operations

When inserting a new task, the element is first added to the end of the array (heap), which takes constant time. To restore the heap property, a heapify-up operation is performed, where the new task is compared with its parent and swapped until the correct position is found. In the worst case, this operation travels from the bottom to the root of the heap, leading to a time complexity of $O(\log n)$, where n is the number of tasks in the heap.

Extracting the task with the highest priority in a max-heap involves removing the root element and replacing it with the last element in the heap. Then, a heapify-down operation is performed to restore the max-heap property by comparing the new root with its children and swapping it down the tree until it reaches the correct position, This process also takes $O(\log n)$ time in the worst case, as the element may travel from root to the deepest level.

Updating the priority of an existing task to a higher value requires modifying the value at a known index and then restoring the heap property. If the new priority is higher, heapify-up is used, which again takes $O(\log n)$.

Screenshot Output

```
Testing heapsort:
Original: [3, 6, 5, 7, 2, 9, 1]
Sorted: [1, 2, 3, 5, 6, 7, 9]

Original: [10, 20, 15, 12, 40, 25, 18]
Sorted: [10, 12, 15, 18, 20, 25, 40]

Original: [1]
Sorted: [1]

Original: []
Sorted: []

Original: [5, 5, 5, 5]
Sorted: [5, 5, 5, 5]

Original: [9, 8, 7, 6, 5, 4, 3, 2, 1]
Sorted: [1, 2, 3, 4, 5, 6, 7, 8, 9]


Testing PriorityQueue with Tasks:
Inserted: Task(id=1, priority=20, arrival=0, deadline=10)
Heap: PriorityQueue([Task(id=1, priority=20, arrival=0, deadline=10)])
Inserted: Task(id=2, priority=15, arrival=2, deadline=12)
Heap: PriorityQueue([Task(id=1, priority=20, arrival=0, deadline=10), Task(id=2, priority=15, arrival=2, deadline=12)])
Inserted: Task(id=3, priority=30, arrival=1, deadline=15)
Heap: PriorityQueue([Task(id=3, priority=30, arrival=1, deadline=15), Task(id=2, priority=15, arrival=2, deadline=12), Task(id=1, priority=20, arrival=0, deadline=10)])
Inserted: Task(id=4, priority=10, arrival=3, deadline=20)
Heap: PriorityQueue([Task(id=3, priority=30, arrival=1, deadline=15), Task(id=2, priority=15, arrival=2, deadline=12), Task(id=1, priority=20, arrival=0, deadline=10), Task(id=4, priority=10, arrival=3, deadline=20)])

Extracted max task: Task(id=3, priority=30, arrival=1, deadline=15)
Heap after extraction: PriorityQueue([Task(id=1, priority=20, arrival=0, deadline=10), Task(id=2, priority=15, arrival=2, deadline=12), Task(id=4, priority=10, arrival=3, deadline=20)])

Increased priority of task 2 to 35
Heap after priority increase: PriorityQueue([Task(id=2, priority=35, arrival=2, deadline=12), Task(id=1, priority=20, arrival=0, deadline=10), Task(id=4, priority=10, arrival=3, deadline=20)])

Extracted max task: Task(id=2, priority=35, arrival=2, deadline=12)
Heap after extraction: PriorityQueue([Task(id=1, priority=20, arrival=0, deadline=10), Task(id=4, priority=10, arrival=3, deadline=20)])

Is priority queue empty? False
```