

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

Haeri Kyoung

University of the Cumberlands

MSCS-532-M20 – Algorithms and Data Structures

Professor Brandon Bass

June 5, 2025

Heapsort Implementation and Analysis

Heapsort Implementation

Heapsort begins by converting the input list into a max heap, which ensures the largest value is at the root of the structure. After the heap is built, the algorithm repeatedly swaps the root element with the last element in the heap, effectively placing the largest value at the end of the array. The size of the heap is reduced, and the heap is restructured to maintain its properties. This process continues until the array is fully sorted.

The implementation uses an in-place approach, which means no additional memory is allocated beyond a few helper variables. This makes the algorithm efficient in both time and space, especially in systems where memory usage must be controlled.

Analysis of Implementation

The time complexity of Heapsort is $O(n \log n)$ in all cases. This includes the best case, average case, and worst case. The first step, which is building the heap, takes linear time. The second step involves removing the maximum element from the heap one by one and reheapifying the remaining structure, which takes $\log n$ time per operation. Since there are n such operations, the total time becomes $n \log n$ regardless of input ordering.

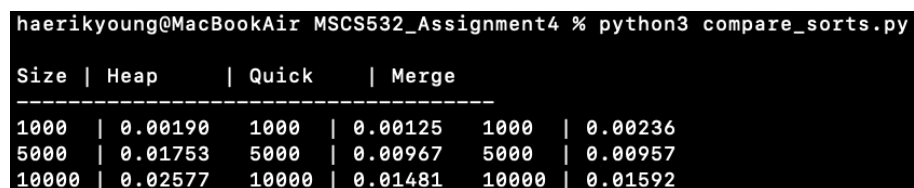
The algorithm has constant space complexity because it does not use any extra arrays or structures. The original array is modified directly. The only additional overhead comes from a few local variables and shallow recursive calls within the heapify function. This makes Heapsort suitable for environments where memory constraints are a concern.

Comparison with Quicksort and Merge Sort

To understand how Heapsort compares to other common sorting algorithms, I ran a benchmark against Quicksort and Merge Sort. All three algorithms were tested using the same randomly generated input arrays of size 1000, 5000, and 10000. The results showed that Quicksort consistently performed the fastest due to its low overhead and cache-friendly behavior. Merge Sort was also fast but used more memory because of its need to allocate new arrays during the merging process.

Heapsort was slightly slower than the other two, but it remained stable across all input sizes. It did not require extra memory and did not suffer from performance drops due to bad pivot selection, as Quicksort might. This makes Heapsort a strong choice in situations where consistent performance and minimal memory usage are more important than raw speed.

A screenshot of the terminal output showing the benchmark results is included below as evidence. It displays the runtime for each algorithm at all three input sizes and confirms the theoretical time complexity discussed above.



```

haerikyoung@MacBookAir MSCS532_Assignment4 % python3 compare_sorts.py

```

| Size | Heap | Quick | Merge |
|-------|---------|-------|---------|
| 1000 | 0.00190 | 1000 | 0.00125 |
| 5000 | 0.01753 | 5000 | 0.00967 |
| 10000 | 0.02577 | 10000 | 0.01481 |

Figure 1. compare_sorts.py output showing runtime comparison for all three sorting algorithms

Part A: Priority Queue Implementation

Data Structure Choice

I implemented the priority queue using a binary heap stored in a Python list. This structure offers efficient memory usage and allows both insert and remove operations to run in logarithmic time.

A list-based binary heap supports constant time index access and allows easy traversal between

parent and child nodes using simple arithmetic formulas. The left child of a node is located at twice the index plus one, and the right child is at twice the index plus two.

I used a min-heap structure to ensure that the task with the smallest numerical priority is always executed first. This decision is based on typical scheduling behavior where a lower priority value represents a higher urgency. The min-heap guarantees that the most important task is always at the root.

Task Representation

Each task in the queue is represented by a class that stores relevant attributes. These include the task ID, priority, arrival time, and optionally a deadline. The priority of each task determines its position in the heap. The class also includes a method to compare tasks based on priority so that they can be ordered properly in the heap.

This object-oriented approach makes the queue more flexible. It allows easy integration of additional features such as execution time, dependencies, or custom scheduling logic without requiring major changes to the structure.

Core Operations

The insert operation adds a task to the end of the heap and shifts it upward until the heap structure is restored. This operation ensures that the task is correctly positioned based on its priority.

The time complexity of insert is logarithmic.

The `extract_min` operation removes the task at the root of the heap, moves the last task to the root, and shifts it downward to restore the heap. This operation ensures the next highest-priority task becomes accessible and also runs in logarithmic time.

The `decrease_key` operation allows dynamic updates of task priority. It searches for the specified task, updates its priority to a lower value, and shifts it upward in the heap if necessary. This is useful in scenarios where task importance changes over time.

The `is_empty` operation simply checks whether the queue has any elements. It evaluates the length of the underlying list and returns a boolean, which runs in constant time. The terminal output below demonstrates each of these operations in action. I inserted three tasks into the queue, extracted the one with the highest priority, updated the priority of one of the remaining tasks, and confirmed that the queue was not empty after these operations.

```
haerikyoung@MacBookAir MSCS532_Assignment4 % python3 test_priority_queue.py

Initial queue:
Task(B, Priority=1)
Task(A, Priority=3)
Task(C, Priority=5)

Extracted task:
Task(B, Priority=1)

Queue after extraction:
Task(A, Priority=3)
Task(C, Priority=5)

Queue after decreasing priority of Task C:
Task(C, Priority=0)
Task(A, Priority=3)

Is queue empty?
False
```

Figure 2. Terminal output from running `test_priority_queue.py` showing queue insertions, extraction, priority update, and final check

Task Scheduling Simulation

To demonstrate the practical use of the priority queue, I created a simple task scheduling simulation. Five tasks were initialized with different priorities and arrival times. The simulation moved through time steps, and tasks were added to the queue as they arrived. At each time step, the highest-priority task available was executed.

This scenario reflects a real-world situation like CPU scheduling or event dispatching. The priority queue handled task order correctly, ensuring that the most urgent task was always executed first, regardless of arrival sequence.

The terminal output shown below illustrates the order in which tasks were added and executed.

```
haerikyoung@MacBookAir MSCS532_Assignment4 % python3 simulate_scheduler.py

Starting task scheduling simulation...

Time 0: Inserted Task(Task1, Priority=4)
Time 0: Executed Task(Task1, Priority=4)
Time 1: Inserted Task(Task2, Priority=2)
Time 1: Executed Task(Task2, Priority=2)
Time 2: Inserted Task(Task3, Priority=1)
Time 2: Executed Task(Task3, Priority=1)
Time 3: Inserted Task(Task4, Priority=3)
Time 3: Executed Task(Task4, Priority=3)
Time 4: Inserted Task(Task5, Priority=5)
Time 4: Executed Task(Task5, Priority=5)

Simulation completed.
```

Figure 3. Output of task scheduling simulation using `simulate_scheduler.py`

Submission Information

GitHub Repository:

https://github.com/hkyoung38554/MSCS532_Assignment4