Assignment 4

MSCS-532-M80

Bibek Itani

Oct 5, 2025

Dr. Satish Penmatsa

This report presents the implementation and analysis of Heap Data Structures through two primary applications:

Heapsort for efficient in-place sorting.

Priority Queue for managing tasks based on priority.

We explore the theoretical and empirical performance of heaps and compare Heapsort with other classic sorting algorithms such as Quicksort and Merge Sort. The second part demonstrates a priority-based task scheduler using a binary heap.

Implementation:

<https://github.com/bibekitani-git/HeapSort.git>

Time Complexity Analysis

| Case | Time Complexity | Explanation |
| --- | --- | --- |
| Worst Case | O(n log n) | Each heapify operation takes O(log n) and is done n times. |
| Average Case | O(n log n) | Same operations regardless of input distribution. |
| Best Case | O(n log n) | Still builds heap and extracts max repeatedly. |

* Build Heap: O(n) using bottom-up approach.
* Extraction Phase: O(n log n) (n times heapify).
* Total: O(n log n) in all cases.

3. Space Complexity

- In-place algorithm: O(1) auxiliary space.

- Heap is represented using an array.

- Recursive calls in heapify: Could use O(log n) stack space, depending on implementation.

4. Comparison with Other Sorting Algorithms

Test Setup

* Algorithms: Heapsort, Quicksort (Python's sorted()), Merge Sort
* Input sizes: [1,000, 5,000, 10,000, 50,000]
* Distributions: Random, Sorted, Reverse Sorted
* Environment: Python 3.x, timeit module

Results (example table)

| Input Size | Distribution | Heapsort (ms) | Quicksort (ms) | Merge Sort (ms) |
| --- | --- | --- | --- | --- |
| 1,000 | Random | 4.5 | 3.9 | 4.2 |
| 10,000 | Sorted | 57.3 | 6.2 | 41.1 |
| 10,000 | Reverse | 60.1 | 9.7 | 42.0 |

Observations

* Heapsort is consistent across input distributions due to O(n log n) guarantee.
* Quicksort is usually faster on average but can degrade to O(n²) in worst-case.
* Merge Sort is stable and reliable, with O(n log n) time and O(n) space.

Part II: Priority Queue Implementation & Applications

Part A: Data Structure & Design

1. Heap Choice

Data structure: Python list

Justification: Array allows simple index calculations for parent/child, efficient insert/extract operations.

2. Task Class

class Task:

def \_\_init\_\_(self, task\_id, priority, arrival\_time, deadline):

self.task\_id = task\_id

self.priority = priority

self.arrival\_time = arrival\_time

self.deadline = deadline

def \_\_lt\_\_(self, other):

return self.priority < other.priority # For min-heap

Uses min-heap for earliest-deadline-first scheduling.

Part B: Core Priority Queue Operations

1. Insert Task

def insert(heap, task):

heap.append(task)

i = len(heap) - 1

while i > 0 and heap[i] < heap[(i - 1) // 2]:

heap[i], heap[(i - 1) // 2] = heap[(i - 1) // 2], heap[i]

i = (i - 1) // 2

Time Complexity: O(log n)

2. Extract Min

def extract\_min(heap):

if len(heap) == 0:

return None

root = heap[0]

heap[0] = heap[-1]

heap.pop()

heapify(heap, len(heap), 0)

return root

Time Complexity: O(log n)

3. Decrease Key (Change Priority)

def decrease\_key(heap, i, new\_priority):

heap[i].priority = new\_priority

while i > 0 and heap[i] < heap[(i - 1) // 2]:

heap[i], heap[(i - 1) // 2] = heap[(i - 1) // 2], heap[i]

i = (i - 1) // 2

Time Complexity: O(log n)

4. Is Empty

def is\_empty(heap):

return len(heap) == 0

Time Complexity: O(1)

🧪 Simulation and Results

Use Case: Task Scheduling

We simulated a scheduler where:

* Tasks arrive with varying priorities and deadlines.
* Scheduler uses a min-heap to prioritize tasks with the earliest deadline.

Metrics Collected

* Average waiting time
* Number of missed deadlines
* Throughput (tasks completed per unit time)

Observations

* Min-heap efficiently retrieved the highest-priority task.
* All core operations performed well on datasets up to 10,000 tasks.
* Scheduler performed better with dynamic priority updates via decrease\_key.

Summary of Findings

| Aspect | Heapsort | Priority Queue |
| --- | --- | --- |
| Time Complexity | O(n log n) | O(log n) per operation |
| Space Complexity | O(1) | O(n) for heap array |
| Input Sensitivity | Not input sensitive | Dynamic task insertion |
| Real-world Utility | Efficient sorter | Task scheduling, OS, etc. |

How to Run:

python heapsort.py

python priority\_queue.py

Dependencies: None beyond Python 3

Summary of Findings: Heapsort is robust and consistent; priority queues are ideal for scheduling tasks by priority or deadline.