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

Name: Murali Krishna Chintha

Student ID: 005034546

Course Title: Algorithms and Data Structures (MSCS-532-B01)

Heapsort Implementation and Analysis

1. Implementation

Python Code for Heapsort:

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[l] > arr[largest]:
        largest = l

    if r < n and arr[r] > arr[largest]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapsort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr
```

The Heapsort algorithm was implemented using a Max-Heap data structure built within a Python list. The key steps include building the heap from the array, repeatedly swapping the root (maximum element) with the last element, and reducing the heap size while restoring the heap property.

2. Time and Space Complexity Analysis

Time Complexity:
- Worst-case: O(n log n)
- Average-case: O(n log n)
- Best-case: O(n log n)

All cases result in O(n log n) time complexity due to heap construction in O(n) and repeated heapify calls on log(n) levels.
Space Complexity: O(1) (in-place algorithm, no auxiliary arrays used).

3. Empirical Comparison

Heapsort was compared with Quicksort and Merge Sort across varying sizes of arrays (sorted, reverse, random). While Quicksort often performs faster due to better cache locality, Heapsort provides consistent performance and better worst-case guarantees.

Priority Queue Implementation and Applications

Part A: Priority Queue Design

Python Code for Priority Queue Implementation:

```
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

class PriorityQueue:
    def __init__(self):
        self.heap = []

    def is_empty(self):
        return len(self.heap) == 0

    def insert(self, task):
        self.heap.append(task)
        self._sift_up(len(self.heap) - 1)

    def extract_min(self):
        if self.is_empty():
            return None
        self._swap(0, len(self.heap) - 1)
```

```
        min_task = self.heap.pop()
        self._heapify(0)
        return min_task

    def decrease_key(self, index, new_priority):
        self.heap[index].priority = new_priority
        self._sift_up(index)

    def _sift_up(self, idx):
        parent = (idx - 1) // 2
        while idx > 0 and self.heap[idx] < self.heap[parent]:
            self._swap(idx, parent)
            idx = parent
            parent = (idx - 1) // 2

    def _heapify(self, idx):
        smallest = idx
        left = 2 * idx + 1
        right = 2 * idx + 2

        if left < len(self.heap) and self.heap[left] < self.heap[smallest]:
            smallest = left
        if right < len(self.heap) and self.heap[right] < self.heap[smallest]:
            smallest = right
        if smallest != idx:
            self._swap(idx, smallest)
            self._heapify(smallest)

    def _swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
```

A min-heap was used to implement the priority queue, ideal for task scheduling based on priority. The Python list represents the heap, allowing efficient insertions and deletions.

Each task is modeled using a Task class containing task ID, priority, arrival time, and deadline.

Core Operations

- insert(task): O(log n) – Inserts a task and restores heap property via sift-up.
- extract_min(): O(log n) – Removes the task with the lowest priority and re-heapifies.
- decrease_key(): O(log n) – Updates priority and performs sift-up to maintain the heap.
- is_empty(): O(1) – Checks if the heap is empty.

Conclusion

This assignment provided a comprehensive understanding of heap-based algorithms and their real-world applications in sorting and priority scheduling. The implementations demonstrate efficient time complexities and validate theoretical expectations.