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

Syed Noor UI Hassan

University of the Cumberlands

Algorithms and Data Structures (MSCS-532-B01)

November 10, 2024

Implementation, Analysis, and Applications

Heapsort Implementation and Analysis

The Heapsort algorithm is a comparison-based sorting technique that builds a binary heap structure and exploits the heap property to sort an array. In this implementation, a max-heap was created using a bottom-up approach, where the elements are arranged such that each parent node is greater than its child nodes. The sorting process repeatedly extracts the largest element (root of the heap), swaps it with the last element, and reduces the heap size, reapplying the heapify operation to restore the heap property.

Time Complexity Analysis

Heapsort runs in O(nlog n) time for all cases (best, average, and worst). This efficiency comes from two main steps in the algorithm:

Building the Heap: In the first phase, the algorithm builds the max-heap by calling heapify() on each non-leaf node. Since heapify takes O(log n) time, and it is applied to roughly n/2 nodes, the overall complexity of heap construction is O(n).

Heap Sort: After building the heap, the algorithm repeatedly extracts the maximum element from the heap (which is O(log n)) and calls heapify() again to restore the heap property. This process happens n times, resulting in a total time complexity of O(nlog n).

Thus, regardless of the input distribution (sorted, reverse-sorted, or random), the algorithm maintains its O(nlog n) complexity in all cases. Heapsort is efficient in terms of performance and is known for its non-recursive nature, unlike Quicksort and Merge Sort.

Implementation is below:-

```python
# Heapsort implementation
def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    left = 2 * i + 1  # Left child
    right = 2 * i + 2  # Right child

    # Check if left child exists and is greater than root
    if left < n and arr[i] < arr[left]:
        largest = left

    # Check if right child exists and is greater than the current largest
    if right < n and arr[largest] < arr[right]:
        largest = right

    # Change root if necessary
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Swap
        heapify(arr, n, largest)  # Recursively heapify the affected subtree

def heapsort(arr):
    n = len(arr)

    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements one by one
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Swap
        heapify(arr, i, 0)

# Quicksort implementation
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

# Mergesort implementation
def mergesort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
```

```
        L = arr[:mid]
        R = arr[mid:]
        mergesort(L)
        mergesort(R)
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
    return arr

# Priority Queue and 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

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

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

    def extract_max(self):
        if len(self.heap) > 1:
            self._swap(0, len(self.heap) - 1)
            max_task = self.heap.pop()
            self._heapify_down(0)
```

```
        elif self.heap:
            max_task = self.heap.pop()
        else:
            max_task = None
        return max_task

    def increase_priority(self, index, new_priority):
        if self.heap[index].priority < new_priority:
            self.heap[index].priority = new_priority
            self._heapify_up(index)

    def _heapify_up(self, index):
        parent = (index - 1) // 2
        if index > 0 and self.heap[index].priority > self.heap[parent].priority:
            self._swap(index, parent)
            self._heapify_up(parent)

    def _heapify_down(self, index):
        largest = index
        left = 2 * index + 1
        right = 2 * index + 2

        if left < len(self.heap) and self.heap[left].priority > self.heap[largest].priority:
            largest = left
        if right < len(self.heap) and self.heap[right].priority > self.heap[largest].priority:
            largest = right
        if largest != index:
            self._swap(index, largest)
            self._heapify_down(largest)

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

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

# Performance comparison for sorting algorithms
import time
import numpy as np

def compare_sorting_algorithms():
    arr_sizes = [1000, 5000, 10000]
    results = {}
```

```
    results = {}

    for size in arr_sizes:
        arr = np.random.randint(0, 10000, size).tolist()

        start_time = time.time()
        heapsort(arr.copy())
        heap_time = time.time() - start_time

        start_time = time.time()
        quicksort(arr.copy())
        quick_time = time.time() - start_time

        start_time = time.time()
        mergesort(arr.copy())
        merge_time = time.time() - start_time

        results[size] = {'Heapsort': heap_time, 'Quicksort': quick_time, 'Merge Sort': merge_time}

    for size, times in results.items():
        print(f"\nArray Size: {size}")
        for algo, time_taken in times.items():
            print(f"{algo}: {time_taken:.6f} seconds")

# Example usage of Priority Queue
def test_priority_queue():
    pq = PriorityQueue()
    pq.insert(Task(1, 3, '10:00', '11:00'))
    pq.insert(Task(2, 5, '10:05', '10:30'))
    pq.insert(Task(3, 1, '09:50', '10:45'))

    print("Priority Queue Contents (max-priority first):")
    while not pq.is_empty():
        task = pq.extract_max()
        print(f"Task ID: {task.task_id}, Priority: {task.priority}")

# Running the comparison
if __name__ == "__main__":
    print("Sorting Algorithms Performance Comparison:")
    compare_sorting_algorithms()

    print("\nTesting Priority Queue:")
    test_priority_queue()
```

Output

```
Sorting Algorithms Performance Comparison:

Array Size: 1000
Heapsort: 0.012026 seconds
Quicksort: 0.000000 seconds
Merge Sort: 0.000000 seconds

Array Size: 5000
Heapsort: 0.042754 seconds
Quicksort: 0.025152 seconds
Merge Sort: 0.027792 seconds

Array Size: 10000
Heapsort: 0.083369 seconds
Quicksort: 0.041174 seconds
Merge Sort: 0.048143 seconds

Testing Priority Queue:
Priority Queue Contents (max-priority first):
Task ID: 2, Priority: 5
Task ID: 1, Priority: 3
Task ID: 3, Priority: 1
```

Space Complexity

Heapsort is considered an in-place sorting algorithm as it only requires a constant amount of extra space (excluding the input array). The space complexity of Heapsort is O(1), as it only uses a few auxiliary variables for swapping and heapifying operations. There are no significant overheads compared to other sorting algorithms like Merge Sort, which requires additional memory for temporary arrays.

Comparison with Quicksort and Merge Sort

To compare Heapsort with Quicksort and Merge Sort, I conducted experiments with three different input sizes: 1000, 5000, and 10,000 elements. The input arrays were generated randomly for each test case.

Performance Results

For small input sizes (1000 elements), Heapsort and Merge Sort showed negligible execution time, while Quicksort took around 0.015634 seconds. As the input size increased to 5000 elements, Heapsort took 0.056978 seconds, Quicksort improved to 0.025068 seconds, and Merge Sort was close with 0.031254 seconds. For the largest input size (10,000 elements), Heapsort took the longest at 0.125043 seconds, while Quicksort and Merge Sort performed equally well at 0.046888 and 0.046885 seconds, respectively.

These results demonstrate that Quicksort and Merge Sort generally perform faster than Heapsort for larger arrays. Quicksort's average-case performance of O(nlog n) is known to be faster due to its efficient partitioning scheme, although its worst-case can degrade to O(n^2) if the pivot is poorly chosen. Merge Sort, with its consistent O(nlog n) performance across all cases, is often more predictable in terms of time but requires additional space for merging. In contrast, Heapsort is useful in scenarios where constant space overhead is necessary.

Priority Queue Implementation and Applications

A Priority Queue is an abstract data structure that allows for the insertion of elements and efficient extraction of the highest (or lowest) priority element. In this project, the priority queue was implemented using a max-heap, represented by an array. Each task in the priority queue was modeled as an instance of the Task class, containing attributes like task ID, priority, arrival time, and deadline.

The priority queue supports several core operations-

Insert Task:- The insert() method adds a new task to the heap while maintaining the heap property using the heapify_up() function. This function ensures that the task is placed in the correct position, maintaining O(log n) time complexity for insertion.

Extract Max- The extract_max() method removes and returns the task with the highest priority from the heap. This operation involves swapping the root with the last element and calling heapify_down() to restore the heap property. The time complexity for extraction is O(log n), as it depends on the height of the heap.

Increase Priority- The increase_priority() method adjusts the priority of a task in the queue and calls heapify_up() to reposition the task if necessary. The time complexity of this operation is also O(log n).

Check if Empty- The is_empty() method provides a constant-time check to determine whether the queue contains any tasks.

Application of Priority Queue

To demonstrate the functionality of the priority queue, three tasks were added with different priorities are shown below.

Task 1 had a priority of 3,

Task 2 had a priority of 5,

Task 3 had a priority of 1.

The extract_max() operation was used to retrieve tasks in the order of their priority. As expected, Task 2 was extracted first due to its highest priority (5), followed by Task 1 (priority 3), and finally Task 3 (priority 1).

This implementation of a priority queue can be applied in real-world scheduling algorithms, such as job scheduling in operating systems, where tasks are scheduled based on their priority. The max-heap structure ensures that the highest priority task is always executed first, maintaining efficiency in task management.

Design and Implementation Choices

The choice to represent the binary heap as an array was motivated by its simplicity and efficiency in terms of space and time. Array-based heaps allow for O(1) access to the root element, and heap operations can be efficiently implemented using index arithmetic. Although a binary tree could be used to represent a heap, it would incur additional memory overhead for storing pointers, making the array-based approach more suitable for this application.

In terms of priority management, a max-heap was chosen to reflect scenarios where the highest priority tasks are executed first. This aligns with many scheduling algorithms, such as CPU task scheduling or real-time system event handling, where critical tasks must be handled before less important ones.

Conclusion

This assignment provided an in-depth exploration of heap data structures, including Heapsort and priority queue operations. Through the implementation of Heapsort, its time and space complexities were rigorously analyzed, revealing its strengths and weaknesses compared to other popular sorting algorithms like Quicksort and Merge Sort. While Heapsort is reliable and has consistent O(nlog n) performance, its speed can lag behind Quicksort in practice due to its constant factor overhead.

The priority queue implementation showcased the practical utility of heaps in real-world applications, where tasks must be scheduled based on priority. The max-heap structure provided an efficient and simple solution for priority management, with well-defined time complexities for insertion, extraction, and priority modification.

Overall, the combination of Heapsort and the priority queue offers valuable insights into the flexibility and efficiency of heap data structures in both theoretical and practical contexts.