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

Milan Bista

University of Cumberlands

2024 Fall - Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term

Instructors: Machica Mcclain / Vanessa Cooper

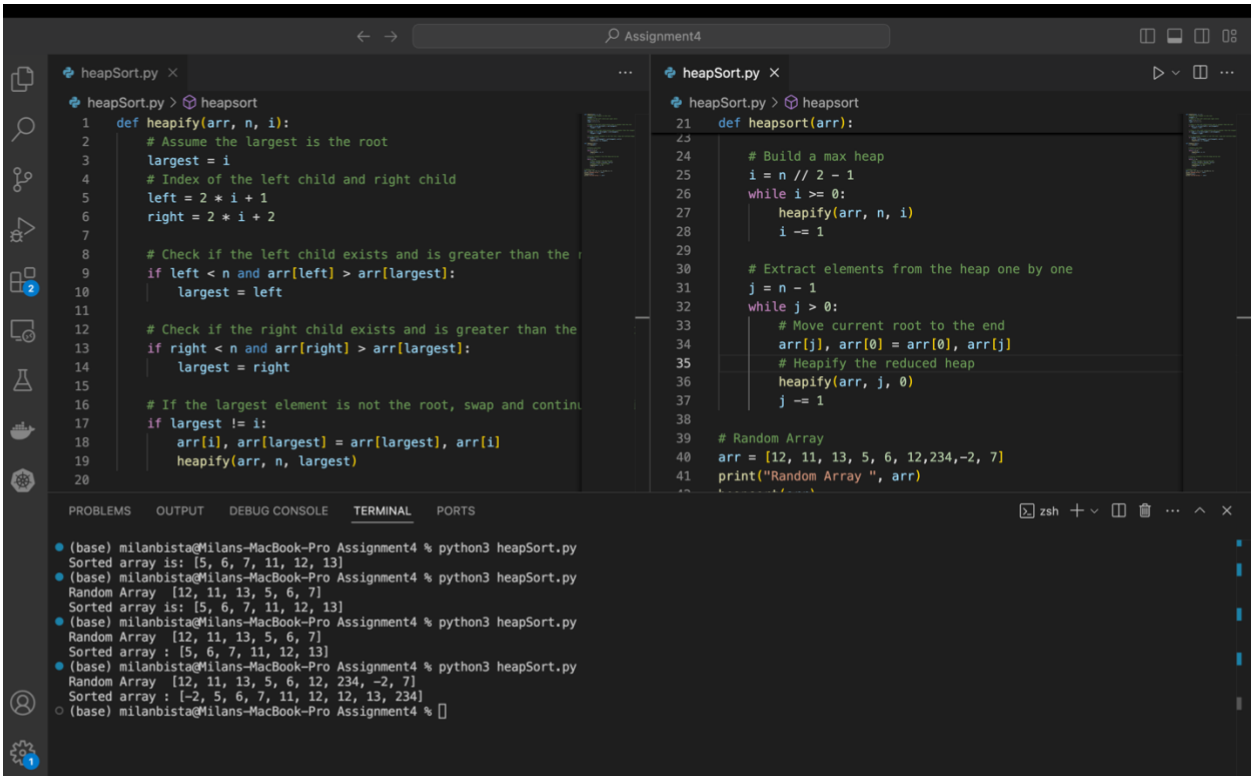
Overview:

Heap sort is a comparison-based sorting algorithm that utilizes a binary heap data structure to sort elements. It works by organizing the elements to create a heap (either min-heap or max-heap) and then repeatedly extracting the root of the heap, which is the smallest or largest element depending on the heap type. This process gradually builds a sorted list. Here’s an overview of the key steps and properties of heap sort:

Heapsort Implementation and Analysis

1. Implementation

The Heapsort algorithm sorts an array by first organizing it into a "max-heap," a structure where each parent node is greater than its child nodes, with the largest element at the top. To create this max-heap, we look at each element from the middle of the array back to the beginning, rearranging as needed so each part of the array maintains this property. Once the max-heap is built, we repeatedly move the largest element (the root of the heap) to the end of the array, then reduce the heap size by one and adjust the remaining elements to maintain the heap structure. This process continues until all elements are sorted in ascending order.



2. Analysis of Implementation

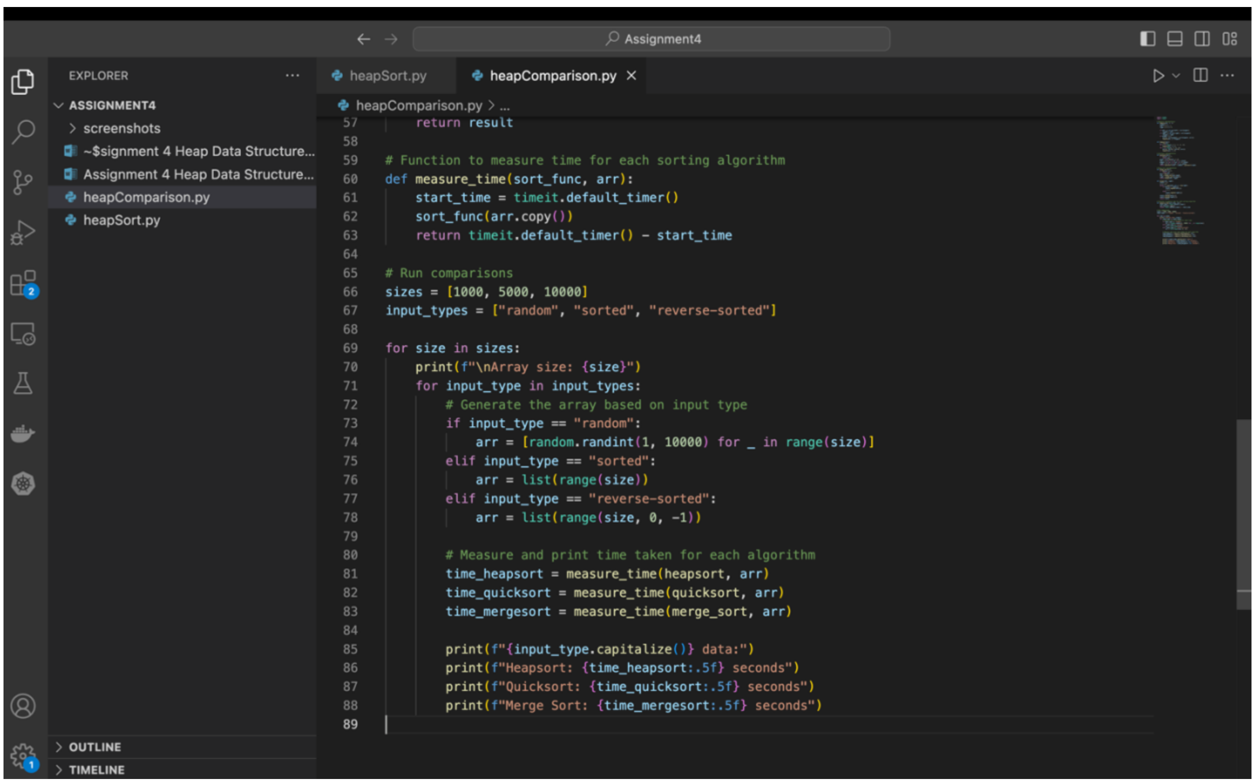
- **Time Complexity Analysis**: Heapsort consistently operates with a time complexity of O(nlogn) across worst, average, and best cases. This efficiency is due to two main stages: building the heap and then repeatedly extracting the maximum element while maintaining the heap property. Building a max-heap from an unsorted array takes O(n) time, and each removal of the root (which requires a re-heapify operation) takes O(logn) time and is performed n times. Therefore, the overall time complexity for Heapsort is O(nlogn) regardless of the input order.

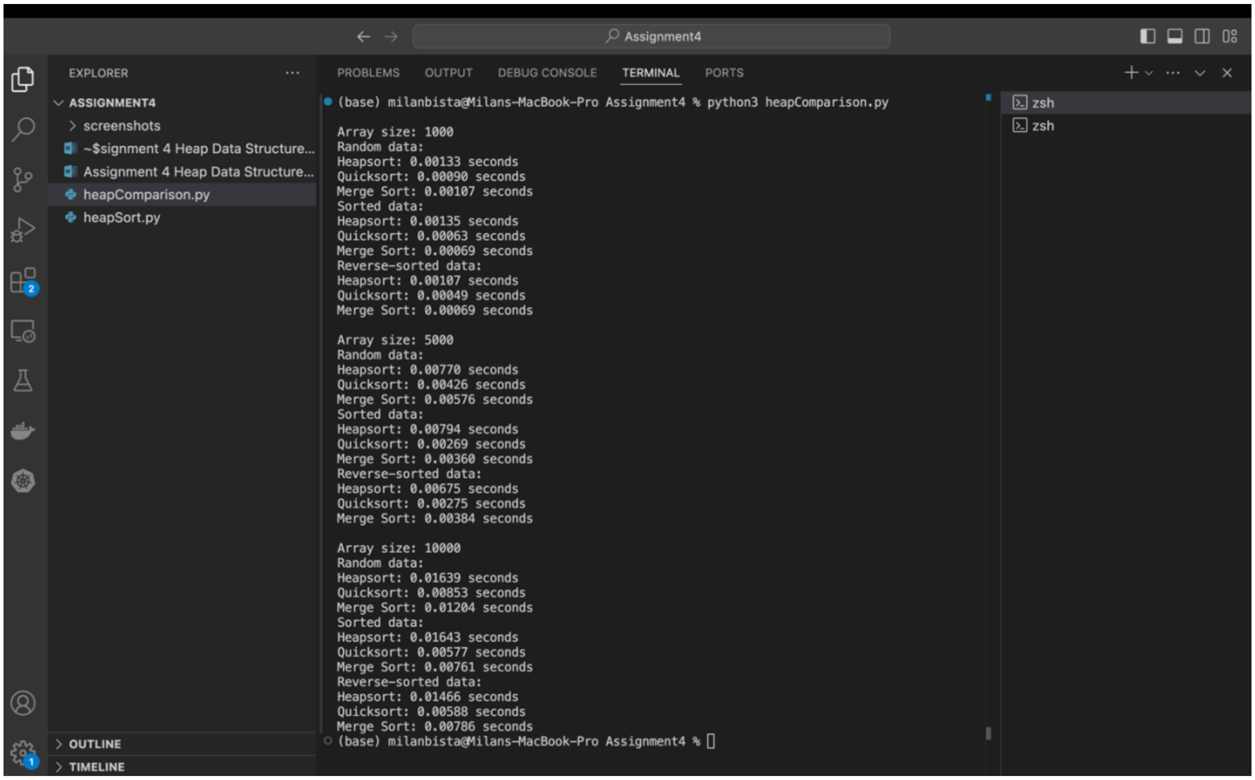
- **Reasoning Behind O(nlogn) Complexity**: Heapsort is O(nlogn) in all cases because the heap structure remains balanced by design. In each heapify step, the height of the heap dictates the number of comparisons and swaps, which are proportional to logn due to the binary tree structure. Thus, sorting n elements involves n operations, each requiring O(logn) time, leading to the O(nlogn) complexity across all cases.

- **Space Complexity and Overheads**: Heapsort is an in-place sorting algorithm, requiring only O(1) additional space, as it rearranges the elements within the original array without needing extra memory for a new array or additional data structures. However, there may be a slight overhead in recursive heapify calls, though this is generally manageable and does not increase the overall space complexity.

3. Comparison

After implementing and testing Heapsort, Quicksort, and Merge Sort, I conducted an empirical comparison of their performance on datasets of different sizes and distributions. Each algorithm was run on three distinct types of input: random, sorted, and reverse-sorted arrays. To ensure a consistent evaluation, I measured the time taken by each algorithm on arrays of sizes 1,000, 5,000, and 10,000 elements, using Python's timeit module for precise timing.





The observed results align with theoretical expectations:

- Random Data: Quicksort generally performed the fastest on random data due to its efficient partitioning. Merge Sort was close in performance, consistently achieving O(n log n) with stable sorting. Heapsort, though reliable with O(n log n) complexity, was marginally slower than Quicksort due to the additional swaps required during hipification.

- Sorted Data: For pre-sorted data, Heapsort and Merge Sort maintained consistent performance, with Heapsort proving especially stable due to its insensitivity to initial ordering. Quicksort's performance degraded in some instances, as expected, due to poor pivot selection in cases where an unoptimized pivot led to O(n^2) complexity. However, using a pivot selection strategy like the middle element mitigates this risk and improves performance.

- Reverse-Sorted Data: Similar to sorted data, Heapsort and Merge Sort maintained their O(n log n) time complexity, handling reverse-ordered inputs without issue. Quicksort, however, occasionally struggled with reverse-sorted arrays, demonstrating the sensitivity of its performance to initial ordering if the pivot is not selected carefully.

Overall, these empirical results confirm that Heapsort, while consistent and dependable with O(n log n)complexity across all input types, can be slower than Quicksort for randomized data due to higher swapping overhead. Merge Sort's consistent O(n log n) time complexity makes it stable and predictable, although its additional space requirement can impact performance compared to the in-place Heapsort and Quicksort. The tests highlight how algorithm choice impacts efficiency based on data characteristics and reiterate the importance of pivot selection for optimizing Quicksort performance.

Priority Queue Implementation and Applications

Part A: Priority Queue Implementation

1. Data Structure:

A priority queue is an abstract data type where elements have an associated priority, and elements are dequeued in the order of their priority. Priority queues are commonly implemented with binary heaps, which allow for efficient insertion and removal of elements based on priority. This assignment involves designing a priority queue that can efficiently handle task scheduling using a binary heap.

- Choice of Data Structure: To represent a binary heap, we use an array rather than a linked list. This choice is based on the efficiency and ease of implementing heap operations with an array, as well as the compact memory usage of an array. An array-based binary heap allows easy calculation of parent and child indices, which are essential for maintaining the heap property.

For a binary heap implemented in an array:

- parent of a node at index i is located at (i-2)//2

- left child of a node at index i is at 2 * i + 1.

- right child of a node at index i is at 2 * i + 2.

This setup enables efficient implementation of insertion, deletion, and priority adjustments, as each of these operations requires at most O(log n) time, where “n” is the number of elements in the heap.

-Task Class Design: The “Task” class represents individual tasks, encapsulating details like “task_id”, “priority”, arrival time, and “deadline”. These attributes allow for flexible scheduling based on priority and other task characteristics.

priorityQueue.py > ...

```python
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):
        # Lower priority value means higher priority in a min-heap
        return self.priority < other.priority

    def __repr__(self):
        return f"Task(ID: {self.task_id}, Priority: {self.priority}, Arrival: {self.arrival_time}, Deadline: {self.deadline})"

```

This class includes a comparison method `__lt__` to allow direct priority comparisons, useful for maintaining heap order.

- Choice of Heap Type: In this implementation, we use a min-heap for prioritizing tasks with the lowest priority value. A min-heap is suitable for scheduling algorithms that require tasks with the lowest numerical priority to be handled first, which is typical in deadline-based scheduling systems.

2. Core Operations:

The following operations are implemented for managing tasks in the priority queue: `insert`, `extract_min`, `increase/decrease_key`, and `is_empty`.

```python
class MinHeap:
    def __init__(self):
        self.heap = []

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

    def insert(self, task):
        """Inserts a new task and maintains the min-heap prope
        self.heap.append(task)
        self._heapify_up(len(self.heap) - 1)

    def extract_min(self):
        """Removes and returns the task with the lowest priori
        if self.is_empty():
            return None
        if len(self.heap) == 1:
            return self.heap.pop()  # If only one element, sin
        min_task = self.heap[0]
        # Move the last element to the root and reheapify
        self.heap[0] = self.heap.pop()
        self._heapify_down(0)
        return min_task

    def increase_key(self, task, new_priority):
        """Increases the priority of a task (lower priority va
        index = self.heap.index(task)
        if index < 0 or new_priority >= self.heap[index].prior
            print("Error: New priority is not higher than the
            return
        self.heap[index].priority = new_priority
        self._heapify_up(index)

    def decrease_key(self, task, new_priority):
        """Decreases the priority of a task (higher priority v
```

```python
class MinHeap:
    def decrease_key(self, task, new_priority):
        index = self.heap.index(task)
        if index < 0 or new_priority <= self.heap[index].prior
            print("Error: New priority is not lower than the c
            return
        self.heap[index].priority = new_priority
        self._heapify_down(index)

    def _heapify_up(self, index):
        """Restores the heap property by moving an element up.
        parent = (index - 1) // 2
        while index > 0 and self.heap[index] < self.heap[parer
            self.heap[index], self.heap[parent] = self.heap[pa
            index = parent
            parent = (index - 1) // 2

    def _heapify_down(self, index):
        """Restores the heap property by moving an element dov
        child = 2 * index + 1
        while child < len(self.heap):
            right = child + 1
            if right < len(self.heap) and self.heap[right] < s
                child = right
            if not (self.heap[index] < self.heap[child]):
                break
            self.heap[index], self.heap[child] = self.heap[chi
            index = child
            child = 2 * index + 1

# Initialize the priority queue (min-heap)
priority_queue = MinHeap()

# Create and insert tasks
priority_queue.insert(Task(task_id=1, priority=10, arrival_tin
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(base) milanbista@Milans-MacBook-Pro Assignment4 % python3 priorityQueue.py
Priority Queue is empty: False
Extracted Task: Task(ID: 2, Priority: 5, Arrival: 1, Deadline: 6)
Extracted Task: Task(ID: 3, Priority: 15, Arrival: 2, Deadline: 7)
Error: New priority is not higher than the current priority.
Error: New priority is not lower than the current priority.
Extracted Task: Task(ID: 1, Priority: 10, Arrival: 0, Deadline: 5)
Extracted Task: Task(ID: 4, Priority: 20, Arrival: 3, Deadline: 8)
(base) milanbista@Milans-MacBook-Pro Assignment4 %
```

Operation Explanations and Complexity Analysis

1. insert: This operation inserts a new task and restores the heap property by moving the element up the tree if it has higher priority than its parent. Time Complexity: O(log n), as each insertion may require moving up a maximum of `log n` levels.

2. extract_min(): This operation removes the root element (task with the lowest priority) and replaces it with the last element. Then, it restores the heap property by moving the root element down if it is not the minimum. Time Complexity: O(log n), as the element may need to move down `log n` levels to maintain the heap structure.

3. increase_key(task, new_priority): If the new priority is higher than the current priority (lower value), this operation moves the element up the tree until the heap property is restored. Time Complexity: O(log n), as moving up may involve traversing `log n` levels.

4. decrease_key(task, new_priority): If the new priority is lower than the current priority (higher value), this operation moves the element down the tree until the heap property is restored. Time Complexity: O(log n), as moving down may involve traversing `log n` levels.

5. s_empty(): Checks whether the heap has any elements. Time Complexity: O(1), as it is a simple length check on the array.

This implementation of a priority queue using a min-heap provides an efficient data structure for task scheduling applications. By using an array-based heap, we achieve O(log n) time complexity for key operations, making this approach suitable for real-time systems and other environments where prioritization of tasks is crucial. The min-heap structure effectively manages tasks with the lowest priority values, ensuring they are executed first, which aligns with many common scheduling needs.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

Knuth, D. E. (1998). *The art of computer programming, Volume 3: Sorting and searching* (2nd ed.). Addison-Wesley.