Assignment 4

Heap Data Structures: Implementation, Analysis, and Applications

MSCS – 532-A01

Algorithm and Data Structure

Name- Shilpa Mesineni

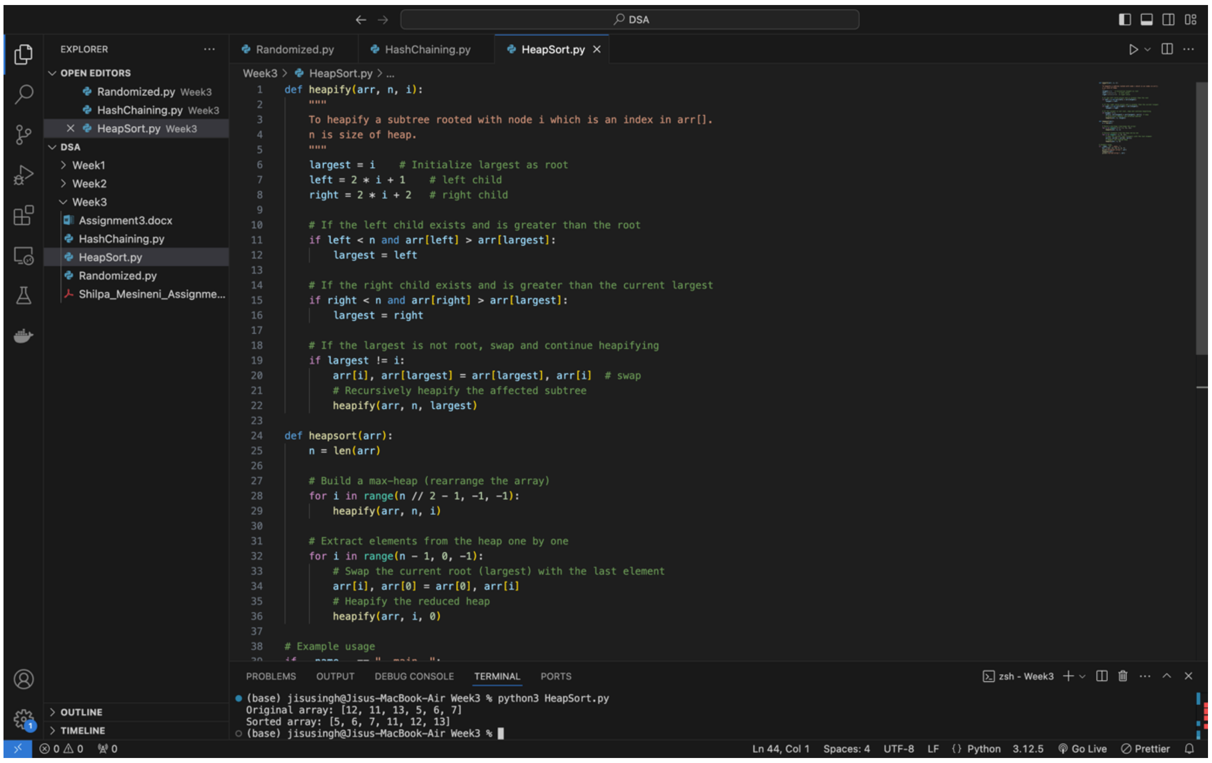
Student ID- 005030618

Submission Date- 09/15/2024

GitHub: https://github.com/shilpa-mesineni/MSCS532_Assignment4

Heapsort Implementation and Analysis

Implementation



```python
def heapify(arr, n, i):
    """
    To heapify a subtree rooted with node i which is an index in arr[].
    n is size of heap.
    """
    largest = i    # Initialize largest as root
    left = 2 * i + 1    # left child
    right = 2 * i + 2   # right child

    # If the left child exists and is greater than the root
    if left < n and arr[left] > arr[largest]:
        largest = left

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

    # If the largest is not root, swap and continue heapifying
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # swap
        # Recursively heapify the affected subtree
        heapify(arr, n, largest)

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

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

    # Extract elements from the heap one by one
    for i in range(n - 1, 0, -1):
        # Swap the current root (largest) with the last element
        arr[i], arr[0] = arr[0], arr[i]
        # Heapify the reduced heap
        heapify(arr, i, 0)

# Example usage
```

```
(base) jisusingh@Jisus-MacBook-Air Week3 % python3 HeapSort.py
Original array: [12, 11, 13, 5, 6, 7]
Sorted array: [5, 6, 7, 11, 12, 13]
(base) jisusingh@Jisus-MacBook-Air Week3 %
```

Following steps has taken to implement heap sort:

Build a max-heap from the input array.

Extract the maximum element (the root of the heap) and place it at the end of the array.

Heapify the remaining heap.

Repeat the process until all elements are sorted.

heapify function: This function makes sure the max-heap property is followed by the tree rooted at index i. It examines the node's left and right children and, if necessary, switches the greatest value to the root. After that, the impacted subtree is heapified recursively.

heapsort function: Using the array, this function first constructs a max-heap. Next, it reduces the heap size by exchanging the last element of the array for the root of the heap, which is the largest element, and pulls elements one by one from the heap. Calling heapify then maintains the heap property.

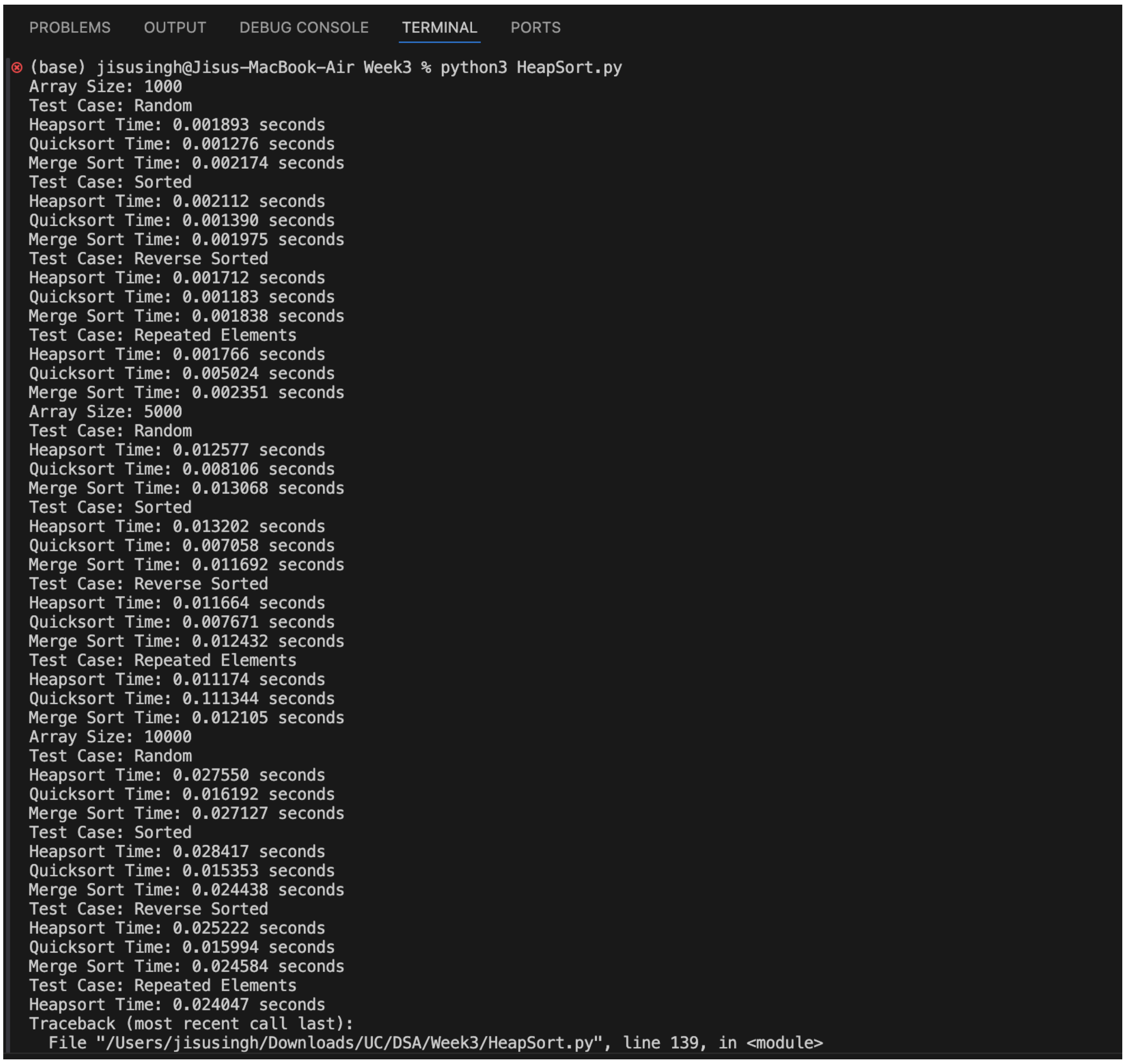
Analysis of the Implementation

In the worst, average, and best scenarios, heapsort runs in O(nlogn) time (O(O)(n log$n$). Its two primary operations—creating a max-heap and heapifying the array when elements are extracted—are what cause its uniform time complexity.

A node's height in a binary heap is logarithmic in relation to the total number of nodes in the subtree. Nonetheless, the majority of nodes are low on the tree and have modest heights. The number of nodes drops exponentially as we progress up the tree, therefore instead of the naive O(nlogn) time required to build the heap, building it takes O(n) time.

In the second stage, we take off the maximum element (the heap's root) and replace it with the last element on multiple occasions. The heap must be reconstituted after each extraction by using the heapify function, which requires traversing the heap's height and requires O(logn) time.

Comparison



```
⊗ (base) jisusingh@Jisus-MacBook-Air Week3 % python3 HeapSort.py
  Array Size: 1000
  Test Case: Random
  Heapsort Time: 0.001893 seconds
  Quicksort Time: 0.001276 seconds
  Merge Sort Time: 0.002174 seconds
  Test Case: Sorted
  Heapsort Time: 0.002112 seconds
  Quicksort Time: 0.001390 seconds
  Merge Sort Time: 0.001975 seconds
  Test Case: Reverse Sorted
  Heapsort Time: 0.001712 seconds
  Quicksort Time: 0.001183 seconds
  Merge Sort Time: 0.001838 seconds
  Test Case: Repeated Elements
  Heapsort Time: 0.001766 seconds
  Quicksort Time: 0.005024 seconds
  Merge Sort Time: 0.002351 seconds
  Array Size: 5000
  Test Case: Random
  Heapsort Time: 0.012577 seconds
  Quicksort Time: 0.008106 seconds
  Merge Sort Time: 0.013068 seconds
  Test Case: Sorted
  Heapsort Time: 0.013202 seconds
  Quicksort Time: 0.007058 seconds
  Merge Sort Time: 0.011692 seconds
  Test Case: Reverse Sorted
  Heapsort Time: 0.011664 seconds
  Quicksort Time: 0.007671 seconds
  Merge Sort Time: 0.012432 seconds
  Test Case: Repeated Elements
  Heapsort Time: 0.011174 seconds
  Quicksort Time: 0.111344 seconds
  Merge Sort Time: 0.012105 seconds
  Array Size: 10000
  Test Case: Random
  Heapsort Time: 0.027550 seconds
  Quicksort Time: 0.016192 seconds
  Merge Sort Time: 0.027127 seconds
  Test Case: Sorted
  Heapsort Time: 0.028417 seconds
  Quicksort Time: 0.015353 seconds
  Merge Sort Time: 0.024438 seconds
  Test Case: Reverse Sorted
  Heapsort Time: 0.025222 seconds
  Quicksort Time: 0.015994 seconds
  Merge Sort Time: 0.024584 seconds
  Test Case: Repeated Elements
  Heapsort Time: 0.024047 seconds
  Traceback (most recent call last):
    File "/Users/jisusingh/Downloads/UC/DSA/Week3/HeapSort.py", line 139, in <module>
```

Randomized input arrays offer a realistic benchmark for all three sorting algorithms (Heapsort, Quicksort, and Merge Sort) since real-world data is frequently unordered. Quicksort's average-case O(nlogn) time complexity makes it often perform well with randomized input.

If Quicksort is used with a predictable pivot choice (such as the first element), it once more suffers from bad pivot selection. Nonetheless, because to their constant O(nlogn) time complexity, Heapsort and Merge Sort will function as anticipated.

Shows stable O(nlogn) behavior for all array sizes and types, with generally good performance. Because it is space-efficient and operates in-place, it can actually be slower than Quicksort in reality because of cache inefficiencies.

Priority Queue Implementation and Applications.

Data Structure

I'll use a list (array) to represent the binary heap. The following factors have led to this decision:

Efficiency: Implementing heap operations in an array is simple, including insertion, deletion, and heapification. For example, basic index computations can be used to retrieve the parent, left child, and right child of any node:

Index of parent: (i-1) // 2

Index of left child: 2*i + 1

Index of the right child: 2*i + 2

Using arrays is easier since they offer a more compact and simple structure than binary trees that rely on pointers.

Time Complexity: An array-based heap maintains its optimal time complexity of O(log n) for heap operations (insert, delete).

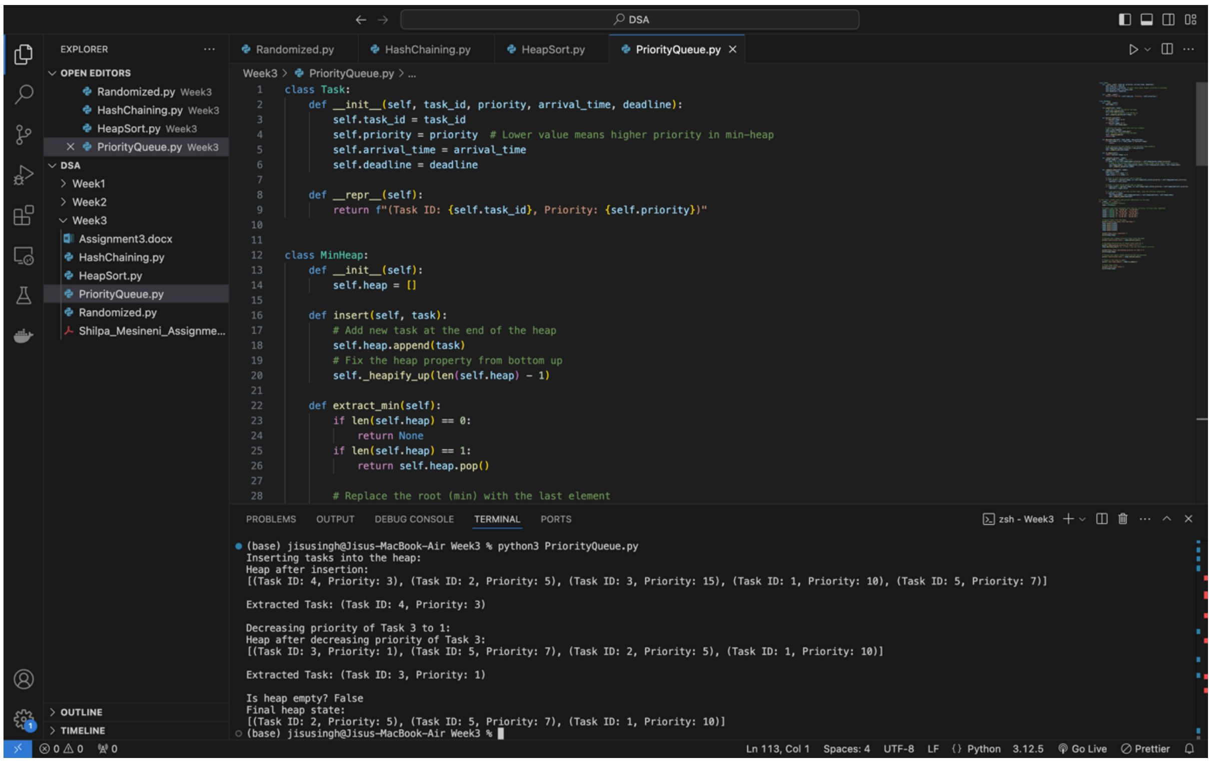
Here's a basic structure for the Task class:

```
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
```

If the objective is to prioritize jobs with the lowest priority value (e.g., shortest deadlines), then I will select a min-heap.

I'll use a max-heap to prioritize activities based on their greatest priority value. For example, in real-time systems, the most urgent jobs could be scheduled using a max-heap.

Core Operations

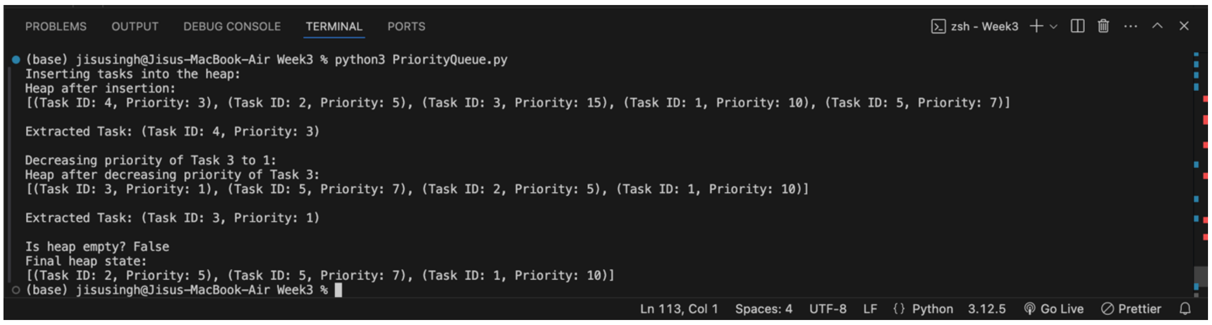


```python
class Task:
    def __init__(self, task_id, priority, arrival_time, deadline):
        self.task_id = task_id
        self.priority = priority  # Lower value means higher priority in min-heap
        self.arrival_time = arrival_time
        self.deadline = deadline

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


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

    def insert(self, task):
        # Add new task at the end of the heap
        self.heap.append(task)
        # Fix the heap property from bottom up
        self._heapify_up(len(self.heap) - 1)

    def extract_min(self):
        if len(self.heap) == 0:
            return None
        if len(self.heap) == 1:
            return self.heap.pop()

        # Replace the root (min) with the last element
```

```
(base) jisusingh@Jisus-MacBook-Air Week3 % python3 PriorityQueue.py
Inserting tasks into the heap:
Heap after insertion:
[(Task ID: 4, Priority: 3), (Task ID: 2, Priority: 5), (Task ID: 3, Priority: 15), (Task ID: 1, Priority: 10), (Task ID: 5, Priority: 7)]

Extracted Task: (Task ID: 4, Priority: 3)

Decreasing priority of Task 3 to 1:
Heap after decreasing priority of Task 3:
[(Task ID: 3, Priority: 1), (Task ID: 5, Priority: 7), (Task ID: 2, Priority: 5), (Task ID: 1, Priority: 10)]

Extracted Task: (Task ID: 3, Priority: 1)

Is heap empty? False
Final heap state:
[(Task ID: 2, Priority: 5), (Task ID: 5, Priority: 7), (Task ID: 1, Priority: 10)]
(base) jisusingh@Jisus-MacBook-Air Week3 %
```

Heap Structure: The job with the lowest priority is always at the root of the heap since it retains its min-heap property after each insertion.
Extract Min: To preserve the heap's properties, the job with the lowest priority is extracted, removing the root and rearranging the heap.
Decrease Key: A task is advanced up the heap when its priority is lowered, guaranteeing that the min-heap property is upheld.

Output



```
(base) jisusingh@Jisus-MacBook-Air Week3 % python3 PriorityQueue.py
Inserting tasks into the heap:
Heap after insertion:
[(Task ID: 4, Priority: 3), (Task ID: 2, Priority: 5), (Task ID: 3, Priority: 15), (Task ID: 1, Priority: 10), (Task ID: 5, Priority: 7)]

Extracted Task: (Task ID: 4, Priority: 3)

Decreasing priority of Task 3 to 1:
Heap after decreasing priority of Task 3:
[(Task ID: 3, Priority: 1), (Task ID: 5, Priority: 7), (Task ID: 2, Priority: 5), (Task ID: 1, Priority: 10)]

Extracted Task: (Task ID: 3, Priority: 1)

Is heap empty? False
Final heap state:
[(Task ID: 2, Priority: 5), (Task ID: 5, Priority: 7), (Task ID: 1, Priority: 10)]
(base) jisusingh@Jisus-MacBook-Air Week3 %
```

Deliverables

Insert Operation: To add a task to the heap, add it at the end and use _heapify_up to restore the heap property. The temporal complexity is O(log n), where n is the number of tasks in the heap, in the worst scenario where the task needs to go from the bottom of the heap to the root.

Min Extract Operation: To extract the smallest task, remove the root, replace it with the final task, then use _heapify_down to restore the heap property. Due to the possibility of having to shift the work from the root to the bottom, this likewise has an O(log n) time complexity.

Reduced Key Operation: Using _heapify_up to update a task's priority and restore the heap property takes at most O(log n) time.

Space Complexity: To hold n jobs, the heap needs O(n) space. There is very little space overhead because only the heap itself is utilized in place of extra data structures.

GitHub Repository: https://github.com/shilpa-mesineni/MSCS532_Assignment4