Assignment 4

Prashanna Acharya

Algorithms and Data Structures (MSCS-532-M80)

University of the Cumberlands

GitHub Link: https://github.com/prashcodes10

Heap Sort

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to sort elements efficiently. It works by first converting the input list into a max-heap, where the largest element is at the root. Then, it repeatedly swaps the root with the last element, removes it from the heap, and re-heapifies the remaining elements. This process continues until all elements are sorted. Heap Sort is known for its consistent O(n log n) time complexity and its in-place sorting nature, meaning it doesn't require extra memory, making it both efficient and memory-friendly.

C: > Users > prash > Desktop > Masters > Fall 2025 > Algorithm and Data Structures > Assignment 4 > heap.py > heap_sort

```python
#Heap_Sort

def maintain_heap(tree, size, root_index):

    max_index = root_index         # Assuming root is the largest
    left_child = 2 * root_index + 1   # Calculating left child index
    right_child = 2 * root_index + 2  # Calculating right child index

    # If left child exists and is greater than root
    if left_child < size and tree[left_child] > tree[max_index]:
        max_index = left_child

    # If right child exists and is greater than current largest
    if right_child < size and tree[right_child] > tree[max_index]:
        max_index = right_child

    # If root is not the largest, swap with the largest child
    if max_index != root_index:
        tree[root_index], tree[max_index] = tree[max_index], tree[root_index]
        # Recursively heapify the affected subtree
        maintain_heap(tree, size, max_index)


def heap_sort(data):

    length = len(data)

    # Building a max heap from the input list
    for i in range(length // 2 - 1, -1, -1):
        maintain_heap(data, length, i)

    # Repeatedly extracting the maximum and heapify the remaining elements
    for i in range(length - 1, 0, -1):
        # Move current max (root) to the end
        data[0], data[i] = data[i], data[0]
        # Restore max-heap structure on the reduced heap
        maintain_heap(data, i, 0)

    return data
```

The above code implements the Heap Sort algorithm in Python to sort a list of numbers in ascending order. It uses a helper function called maintain_heap to ensure that a given portion of the list maintains the max-heap property, where each parent node is greater than or equal to its children. First, the heap_sort function builds a max-heap from the unsorted list by calling maintain_heap on all non-leaf nodes. Then, it repeatedly swaps the root of the heap (the maximum element) with the last element of the heap, reduces the heap size, and calls maintain_heap again to restore the heap structure. This process continues until the entire list is sorted. The sorted list is returned at the end. The if __name__ == "__main__": block demonstrates the algorithm using a sample list.

Analysis of Implementation

Time Complexity Analysis of Heap Sort

Heap Sort has a consistent time complexity of O(n log n) in the worst, average, and best cases. This uniformity arises from the algorithm's two main phases: heap construction and heap extraction. During the first phase, the algorithm builds a max-heap from the unsorted list by calling the maintain_heap function (heapify) on all non-leaf nodes in a bottom-up manner. Although it might appear that this would require O(n log n) operations, this phase runs in O(n) time because most of the heapify calls are on nodes near the bottom of the tree, which take significantly less time to process.

The second phase of Heap Sort involves removing the maximum element (root) and placing it at the end of the list, then calling maintain_heap again to restore the heap structure. This step is repeated for each element in the array, and each heapify call takes up to O(log n) time because it may traverse from the root to the leaf of the heap (which has height log n). Since this operation is done n times, the total complexity for this phase is O(n log n). Therefore, while the heap-building step is O(n), it is dominated by the O(n log n) extraction phase, resulting in an overall time complexity of O(n log n) in all cases.

The reason Heap Sort doesn't have a better best-case time than O(n log n) is because it always performs the same extraction and re-heapifying steps regardless of the input's initial order, whether sorted, reversed, or random. Unlike algorithms like quicksort, which can have better or worse performance based on the pivot choice and input distribution, heap sort follows a fixed structure.

Space Complexity and Overheads

Heap Sort is an in-place sorting algorithm, meaning it doesn't require any significant additional memory. The space complexity is O(1), and only a constant amount of extra space is used for temporary variable swaps and loop control, aside from the input array itself. There is no need for auxiliary arrays, which makes it more memory-efficient than algorithms like Merge Sort, which uses O(n) extra space.

As for overheads, the algorithm makes frequent swaps, which can be relatively expensive on systems with high memory latency or when sorting large data types. Heap Sort is also not a stable sort, meaning it doesn't guarantee the preservation of the relative order of equal elements. This can be a drawback in applications where stability is important.

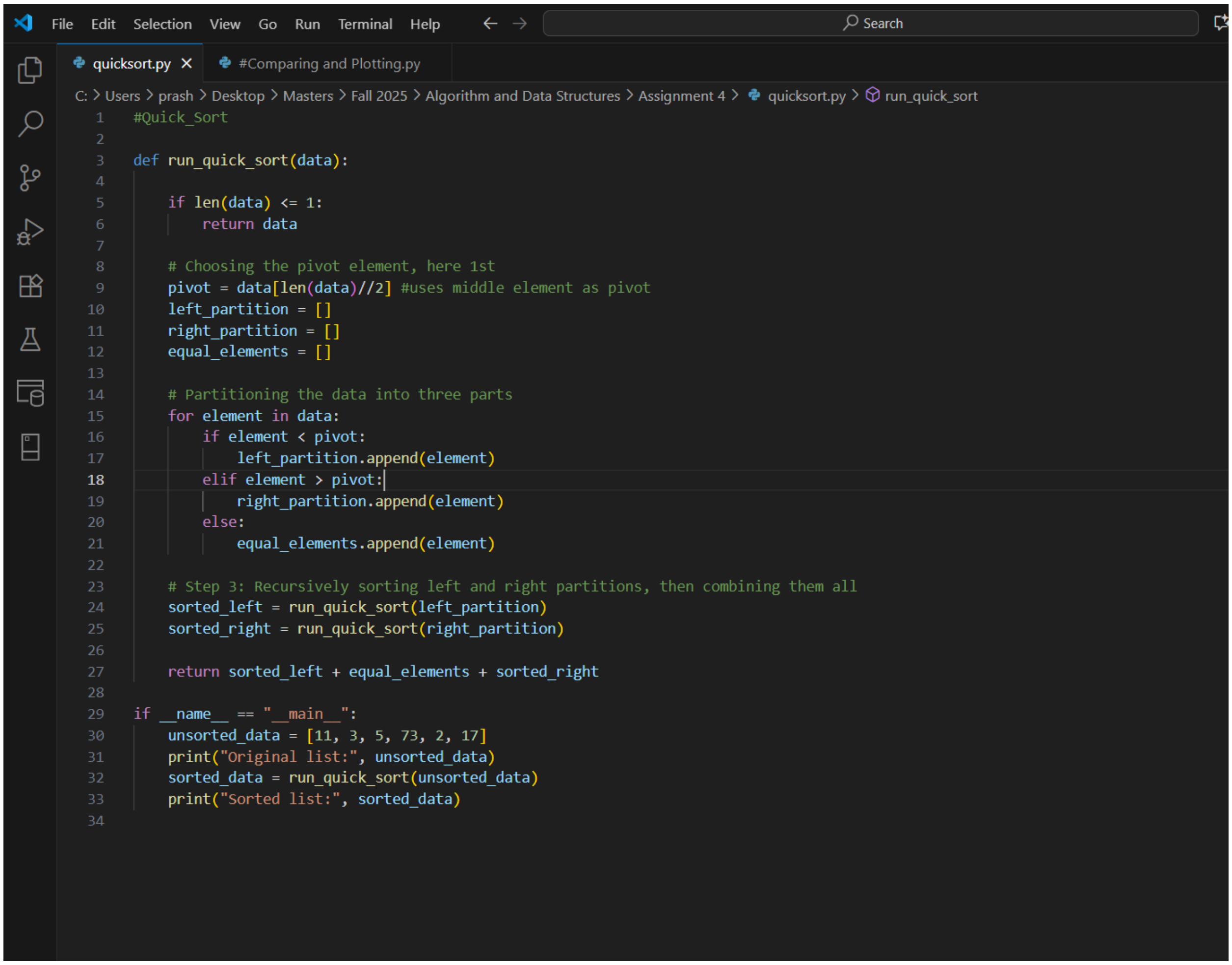
In summary, Heap Sort is a reliable and space-efficient algorithm with a predictable O(n log n) time complexity across all input types. However, it may be outperformed in practice by other algorithms like quicksort or timsort (used in Python's built-in sort) due to lower constant factors and better cache performance.

Runtime Comparisons of Heapsort, Quick Sort, and Merge Sort

Quick Sort

The Quick Sort algorithm is a highly efficient, comparison-based sorting method that follows the divide-and-conquer strategy. It works by selecting a pivot element from the list and partitioning the remaining elements into two sub-lists: those less than the pivot and those greater than the pivot. These sublists are then recursively sorted using the same approach. Once the sublists are sorted, they are combined with the pivot to form the final sorted list. Quick Sort performs well on average with a time complexity of *O(n log n)*, but its worst-case time complexity is $O(n^2)$, which can occur if poor pivot choices lead to highly unbalanced partitions.

Despite that, its speed and in-place nature make it one of the most commonly used sorting algorithms.



Merge Sort

Likewise, Merge Sort is another reliable and efficient sorting algorithm based on the divide-and-conquer principle. It works by recursively dividing the input list into two halves until each sublist contains only one element, which is naturally sorted. Then, it repeatedly merges these sorted sublists back together in the correct order to produce a single sorted list. Unlike Quick Sort, Merge Sort guarantees a consistent time complexity of *O(n log n)* in the best, worst, and average cases, making it very predictable in performance. However, it requires additional space for the merging process, so its space complexity is *O(n)*. Because it maintains the original order of equal elements, Merge Sort is also a stable sorting algorithm, which is useful in scenarios where order matters.

C: > Users > prash > Desktop > Masters > Fall 2025 > Algorithm and Data Structures > Assignment 4 > mergesort.py > ...

```python
#Merge_Sort

def run_merge_sort(data):

    if len(data) <= 1:
        return data

    # Splitting the list into two halves
    middle = len(data) // 2
    left_partition = data[:middle]
    right_partition = data[middle:]

    #Recursively s (function) def run_merge_sort(data) -> Any | list
    sorted_left = run_merge_sort(left_partition)
    sorted_right = run_merge_sort(right_partition)

    #Merging the sorted halves into one sorted list
    return combine(sorted_left, sorted_right)


def combine(left_list, right_list):

    result = []  # Final merged result
    left_index = 0
    right_index = 0

    #Comparing elements from both lists and adding the smaller one
    while left_index < len(left_list) and right_index < len(right_list):
        if left_list[left_index] < right_list[right_index]:
            result.append(left_list[left_index])
            left_index += 1
        else:
            result.append(right_list[right_index])
            right_index += 1

    #Changing any remaining elements from left_list
    while left_index < len(left_list):
        result.append(left_list[left_index])
        left_index += 1

    #Changing any remaining elements from right_list
    while right_index < len(right_list):
        result.append(right_list[right_index])
        right_index += 1
```

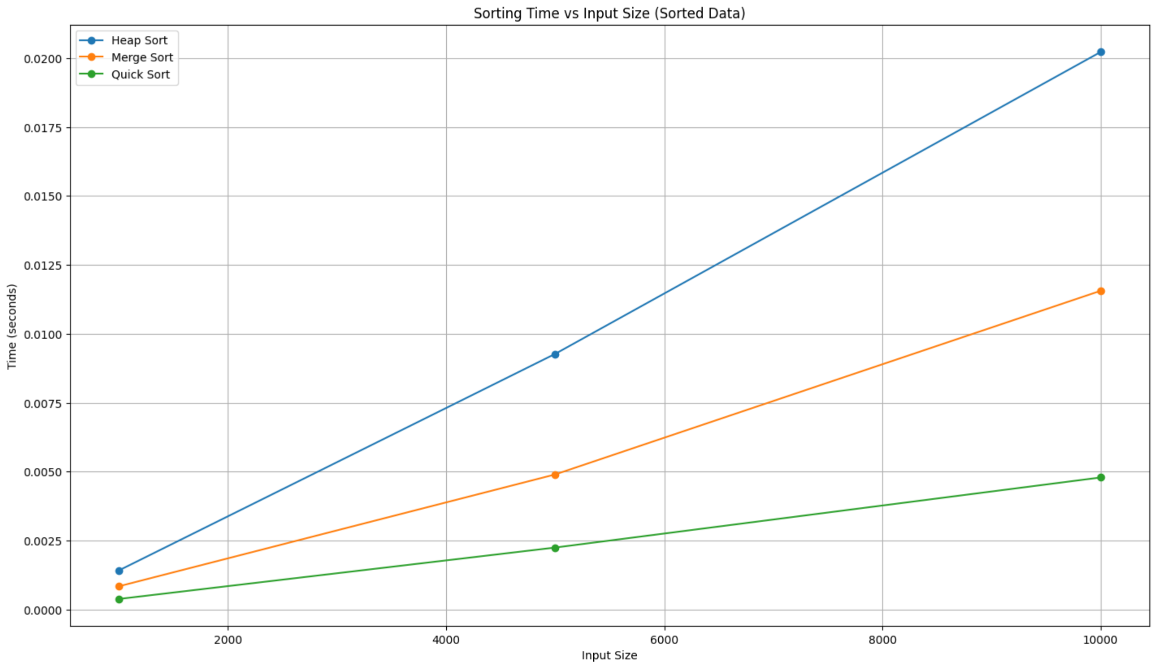
Code Snippet to Compare these three Algorithms

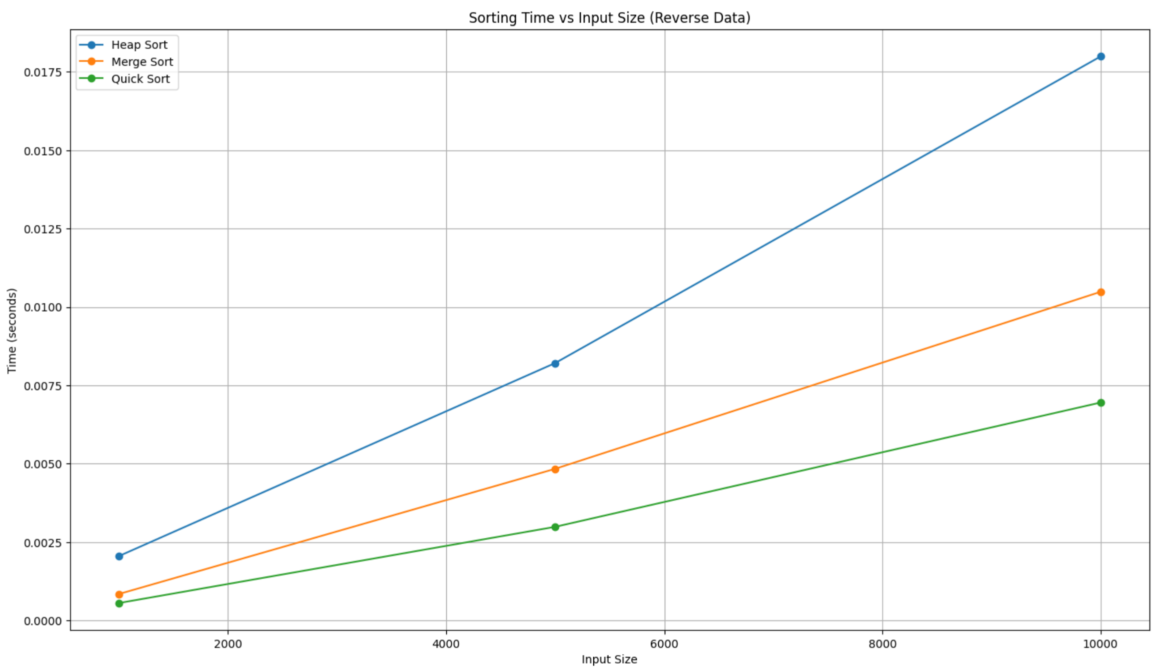
```python
#Comparing and Plotting
import time
import random
import matplotlib.pyplot as plt

# ----------------------------------------
# Importing the sorting algorithms
# ----------------------------------------

from heap import heap_sort
from quicksort import run_quick_sort
from mergesort import run_merge_sort


def generate_input(size, distribution):

    if distribution == 'sorted':
        return list(range(size))
    elif distribution == 'reverse':
        return list(range(size, 0, -1))
    elif distribution == 'random':
        lst = list(range(size))
        random.shuffle(lst)
        return lst
    else:
        raise ValueError("Unknown distribution type")


def time_sorting_algorithm(sort_func, data):

    start_time = time.perf_counter()
    sort_func(data.copy())
    end_time = time.perf_counter()
    return end_time - start_time


def run_benchmarks():
    sizes = [1000, 5000, 10000]  # Different input sizes to test
    distributions = ['sorted', 'reverse', 'random']
    sorting_algorithms = {
        'Heap Sort': heap_sort,
        'Merge Sort': run_merge_sort,
        'Quick Sort': run_quick_sort
    }
```

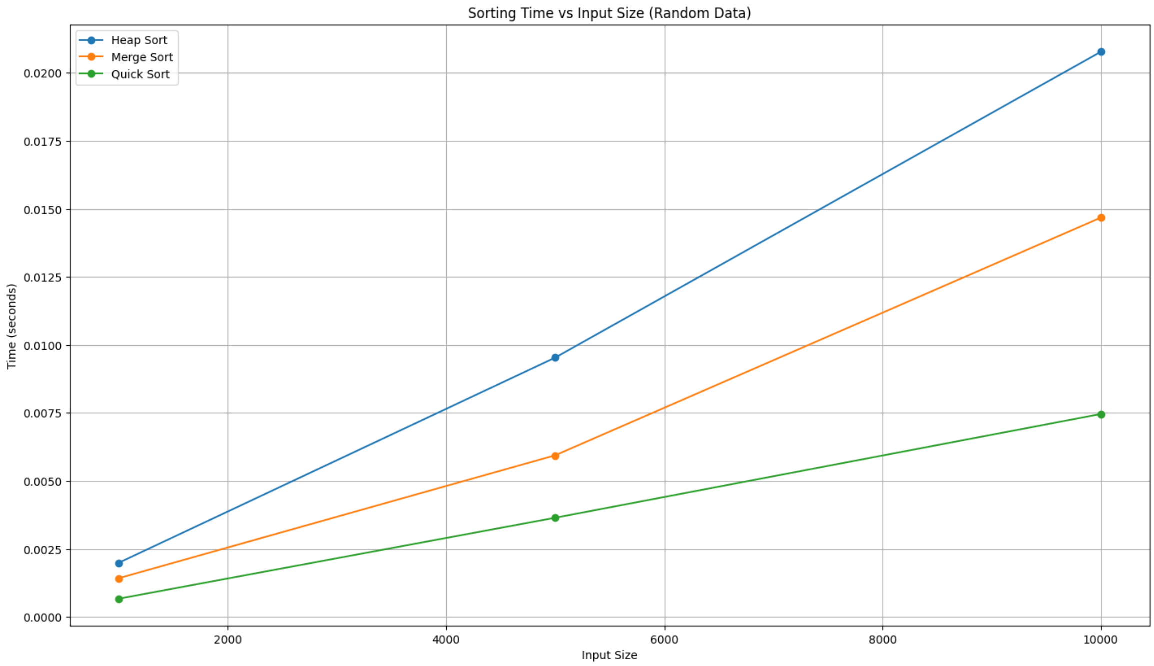
```python
    }


    results = {dist: {name: [] for name in sorting_algorithms} for dist in distributions}

    for dist in distributions:
        for size in sizes:
            data = generate_input(size, dist)
            for name, func in sorting_algorithms.items():
                elapsed = time_sorting_algorithm(func, data)
                results[dist][name].append(elapsed)
                print(f"Size {size}, {dist.capitalize()}, {name}: {elapsed:.6f} sec")

    return sizes, results


def plot_results(sizes, results):

    for dist, algorithms in results.items():
        plt.figure(figsize=(8, 5))
        for name, times in algorithms.items():
            plt.plot(sizes, times, marker='o', label=name)
        plt.title(f"Sorting Time vs Input Size ({dist.capitalize()} Data)")
        plt.xlabel("Input Size")
        plt.ylabel("Time (seconds)")
        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.show()


if __name__ == "__main__":
    sizes, benchmark_results = run_benchmarks()
    plot_results(sizes, benchmark_results)
```

This script showcases the performance of Heap Sort, Merge Sort, and Quick Sort on lists of varying sizes and distributions (sorted, reverse, and random). It generates input data, times how long each algorithm takes to sort the lists and stores the results. After running the tests, it plots the timing results using line graphs to visually compare algorithm performance. The code runs all benchmarks when executed and displays both the timing output and performance plots.

Results and Graphs


Sorting Time vs Input Size (Sorted Data)
Heap Sort
Merge Sort
Quick Sort
Time (seconds)
0.0000
0.0025
0.0050
0.0075
0.0100
0.0125
0.0150
0.0175
0.0200
2000
4000
6000
8000
10000
Input Size


Sorting Time vs Input Size (Reverse Data)
Heap Sort
Merge Sort
Quick Sort
Time (seconds)
0.0000
0.0025
0.0050
0.0075
0.0100
0.0125
0.0150
0.0175
2000
4000
6000
8000
10000
Input Size



Analysis:

The three graphs provided compare the running times of Heap Sort, Merge Sort, and Quick Sort on datasets of varying order: Sorted Data, Reverse Data, and Random Data. These analyses helps in understanding the performance of each sorting algorithm in different data scenarios.

In the Sorted Data scenario, all three algorithms show an increase in running time as the input size grows. Heap Sort's time increases steadily in a linear fashion, suggesting that its performance is relatively unaffected by the initial order of the data. Merge Sort behaves similarly, with a slight edge in performance compared to Heap Sort, as it shows a slightly slower increase in time. However, Quick Sort stands out as the fastest among the three. Its time complexity is $O(n\log\ n)$, which generally leads to faster performance than both heap sort and merge sort, especially on sorted data.

When applied to Reverse Data, Heap Sort and Merge Sort exhibit similar behaviors to those seen with sorted data, with both showing a linear increase in running time. However, Quick Sort's performance degrades in this scenario. This is because Quick Sort may encounter its worst-case time complexity of $O(n^2)$if it consistently picks poor pivot elements (which can happen with reverse-sorted data), leading to a noticeable increase in time. Therefore, while Merge Sort and Heap Sort maintain stable and predictable performance, Quick Sort's efficiency is heavily impacted by the reverse order of the input.

In the Random Data scenario, the performance of all three algorithms follows a similar pattern to that seen in the sorted data case, but with some variations. Heap Sort's running time increases steadily, slightly higher than when the data is sorted but still showing a linear growth pattern. Merge Sort performs similarly, exhibiting a stable linear increase in time, unaffected by the data's order. Quick Sort, on the other hand, performs the best in this scenario. As expected, Quick Sort's time complexity follows an $O(n\log\ n)$pattern, making it the most efficient algorithm for random data. This is because Quick Sort typically avoids the worst-case scenario when working with random datasets.

In summary, Heap Sort and Merge Sort show relatively consistent performance across all three types of data, both demonstrating linear growth in time complexity. While Heap Sort is slightly faster than Merge Sort on sorted and reverse data, both algorithms are stable and predictable, regardless of the data order. Quick Sort, however, excels on random data, displaying its characteristic $O(n\log\ n)$performance. However, its performance degrades significantly when working with reverse-sorted data due to the worst-case $O(n^2)$time complexity. The overall takeaway is that Quick Sort is ideal when working with random or mostly sorted data, Merge

Sort is a reliable and stable choice for all input types, and Heap Sort, while consistent, is generally slower than Quick Sort in random or sorted data scenarios.

Priority Queue Implementation

A binary heap is utilized for implementing a priority queue due to its efficiency and simplicity. It can be represented using an array (or list) and offers optimal performance for the key operations of a priority queue. The binary heap allows for fast insertion and removal, which is essential for managing tasks according to their priority.

In this code, the Task class stores key details for each task, including a unique ID, priority level (with higher priority processed first), arrival time, and deadline. These attributes enable effective management and scheduling of tasks with varying urgency, arrival times, and deadlines. A max heap was chosen because tasks with higher priority must be processed first. This ensures the highest-priority task is always at first, enabling efficient extraction.

quicksort.py | #Comparing and Plotting.py | priorityqueue.py

C: > Users > prash > Desktop > Masters > Fall 2025 > Algorithm and Data Structures > Assignment 4 > priorityqueue.py > ...

```python
#Priority_Queue

class Job:
    def __init__(self, jid, priority, start_time, due_time):
        self.jid = jid
        self.priority = priority
        self.start_time = start_time
        self.due_time = due_time

    def __repr__(self):
        # A readable string representation of the job is returned
        return f"[Job #{self.jid} | Priority: {self.priority} | Start: {self.start_time} | Due: {self.due_time}]"


class MaxPriorityQueue:
    def __init__(self):
        # The internal list to store the heap is initialized
        self.queue = []

    def insert(self, job):
        # The job is checked before insertion to avoid None values
        if job is None:
            print("Invalid job. Cannot insert.")
            return

        self.queue.append(job)
        i = len(self.queue) - 1

        while i > 0:
            parent = (i - 1) // 2
            if self.queue[i].priority > self.queue[parent].priority:
                self.queue[i], self.queue[parent] = self.queue[parent], self.queue[i]
                i = parent
            else:
                break

    def extract_max(self):
        # If the queue is empty, a message is shown
        if not self.queue:
            print("Queue is empty.")
            return None
```

```python
        # The heap property is restored by bubbling down
        self._heapify(0)
        return top

    def _heapify(self, i):
        # The children indices are computed
        left = 2 * i + 1
        right = 2 * i + 2
        largest = i

        # The index of the largest value is determined
        if left < len(self.queue) and self.queue[left].priority > self.queue[largest].priority:
            largest = left
        if right < len(self.queue) and self.queue[right].priority > self.queue[largest].priority:
            largest = right

        # A swap is performed if the heap property is violated
        if largest != i:
            self.queue[i], self.queue[largest] = self.queue[largest], self.queue[i]
            # The process is repeated recursively for the affected subtree
            self._heapify(largest)

    def update_priority(self, job, new_priority):
        # The job is validated before proceeding
        if job is None:
            print("Cannot update priority for a null job.")
            return

        # The priority is updated to the new value
        job.priority = new_priority
        i = self.queue.index(job)

        # The job is bubbled up to maintain the heap structure
        while i > 0:
            parent = (i - 1) // 2
            if self.queue[i].priority > self.queue[parent].priority:
                self.queue[i], self.queue[parent] = self.queue[parent], self.queue[i]
                i = parent
            else:
                break

    def is_empty(self):
        # A boolean indicating whether the queue is empty is returned
        return not self.queue
```

Insert Operation (O(log n))

When a task is inserted, it is first placed at the end of the heap. To maintain the heap property, a heapify-up operation is performed, which may move the task up toward the root. Since this process traverses the height of the heap, it takes logarithmic time in the worst case.

Extract Max Operation (O(log n))

To extract the highest-priority task, the root node is replaced with the last node in the heap, followed by a heapify-down operation. This re-establishes the max-heap structure by comparing and swapping nodes down the tree, which also takes logarithmic time in the worst case.

Insert Key Operation (O(log n))

When a task's priority is increased, the key is updated, and the heapify-up operation is used to restore the max-heap property. As the task may need to move from a leaf node up to the root, the time complexity remains logarithmic.

Empty Operation: $\boldsymbol{O(1)}$

It simply checks the heap length, which takes constant time.

References

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

Weiss, M. A. (2014). *Data structures and algorithm analysis in C++* (4th ed.). Pearson.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Java* (6th ed.). Wiley.