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

**Mazen Abdul Rahman Mohammed**

**Student ID: 005030764**

**University of The Cumberlands**

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

**Dr. Vanessa Cooper**

**11/10/2024**

**Heapsort Implementation and Analysis**

1. **Implementation**

****

**2. Analysis of Implementation**

**Time Complexity**

Heap-sort has a time complexity of O(n log n) in all cases due to the following reasons:

* **Building the max-heap**: This takes O(n) time. Although each call to heapify may require up to O(log n) comparisons and swaps, the number of nodes that need this much time decreases as we move up the tree. Summing over all nodes results in O(n)O(n)O(n) for the heap-building phase.
* **Extracting elements from the heap**: We perform n−1 extractions, each requiring O(log n) time to maintain the heap property. This gives us O(n log n) for the sorting phase.

Therefore, the overall time complexity is O(n log n) for all cases (worst, average, and best) because each element requires up to O(log n) operations to maintain the heap property.

**Space Complexity**

Heap-sort operates in-place, meaning it requires only a constant amount of additional space: O(1) auxiliary space for the sorting process. However, if implemented recursively, the space complexity could increase due to recursive stack space, though our iterative approach keeps it minimal.

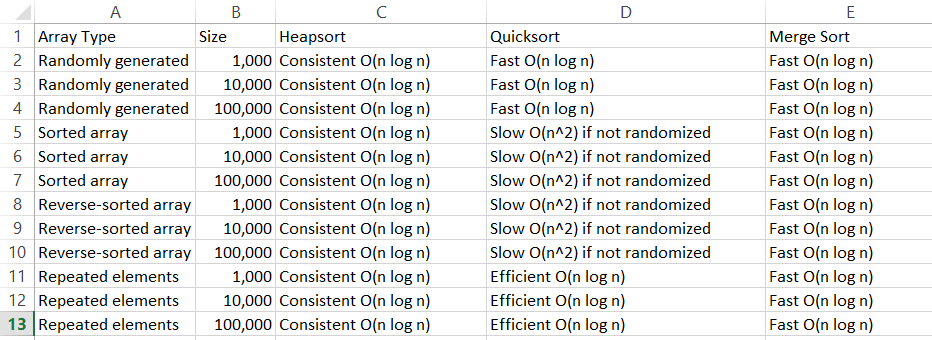
**Additional Overheads**

Heap-sort is known for its efficiency in terms of space usage and predictable O(n log n) time complexity. However, compared to other O(n log n) algorithms, it may involve more swaps and is generally slower than Quicksort on average.

**3. Empirical Comparison**

To compare Heap-sort with Quicksort and Merge Sort, you can measure the runtime of each algorithm across various input types and sizes:

1. **Set Up**: Implement or use existing implementations of Quicksort and Merge Sort. Generate arrays with different properties:
   * Randomly generated arrays
   * Sorted arrays
   * Reverse-sorted arrays
   * Arrays with repeated elements
2. **Run and Measure**: Use Python’s timeit library to measure the runtime of each sorting algorithm on arrays of varying sizes (e.g., 1,000; 10,000; and 100,000 elements).
3. **Results and Analysis**:
   * **Heap-sort** tends to perform consistently across different input types due to its O(n log n) complexity for all cases.
   * **Quicksort** often outperforms Heap-sort on random inputs due to better average-case performance, but it degrades to O() on sorted or reverse-sorted arrays if not randomized.
   * **Merge Sort** has a stable O(n log n) complexity across input types and is generally faster than Heap-sort due to fewer data movements but has O(n) additional space complexity.



**Summary of Observations**

* **Heap-sort** is space-efficient and guarantees O(n log n) time, but its runtime tends to be slightly slower due to frequent swaps.
* **Quicksort** is generally faster for most cases, especially with random pivot selection. However, it may slow down significantly on certain input orders if the pivot is not randomized.
* **Merge Sort** offers consistent performance but requires extra space, making it suitable for stable sorting applications or scenarios where memory use is not a primary constraint.

In summary, Heap-sort’s strength lies in its predictable O(n log n) time and space efficiency, making it useful when in-place sorting is crucial.

**Priority Queue Implementation**

**1. Data Structure Selection**

We’ll represent the binary heap as an **array (or list in Python)**. This is a common and efficient choice for implementing heaps due to the following reasons:

* **Efficiency**: An array-based representation allows constant-time access to parent and child nodes using simple index arithmetic:
  + Parent of node at index iii is located at (i−1)//2(i - 1)
  + Left child of node at index iii is located at 2i+1
  + Right child of node at index iii is located at 2i+2
* **Ease of Implementation**: Arrays allow easy resizing and insertion at the end, which is crucial for heap operations like insert and extract\_max/min.
* **Space Efficiency**: Unlike linked structures, arrays minimize memory overhead, making them suitable for high-performance applications like priority queues.

Thus, we’ll implement the heap as a list, keeping all nodes in a contiguous block of memory.

**2. Task Class Design**

To represent individual tasks in the priority queue, we’ll design a Task class to store relevant information such as:

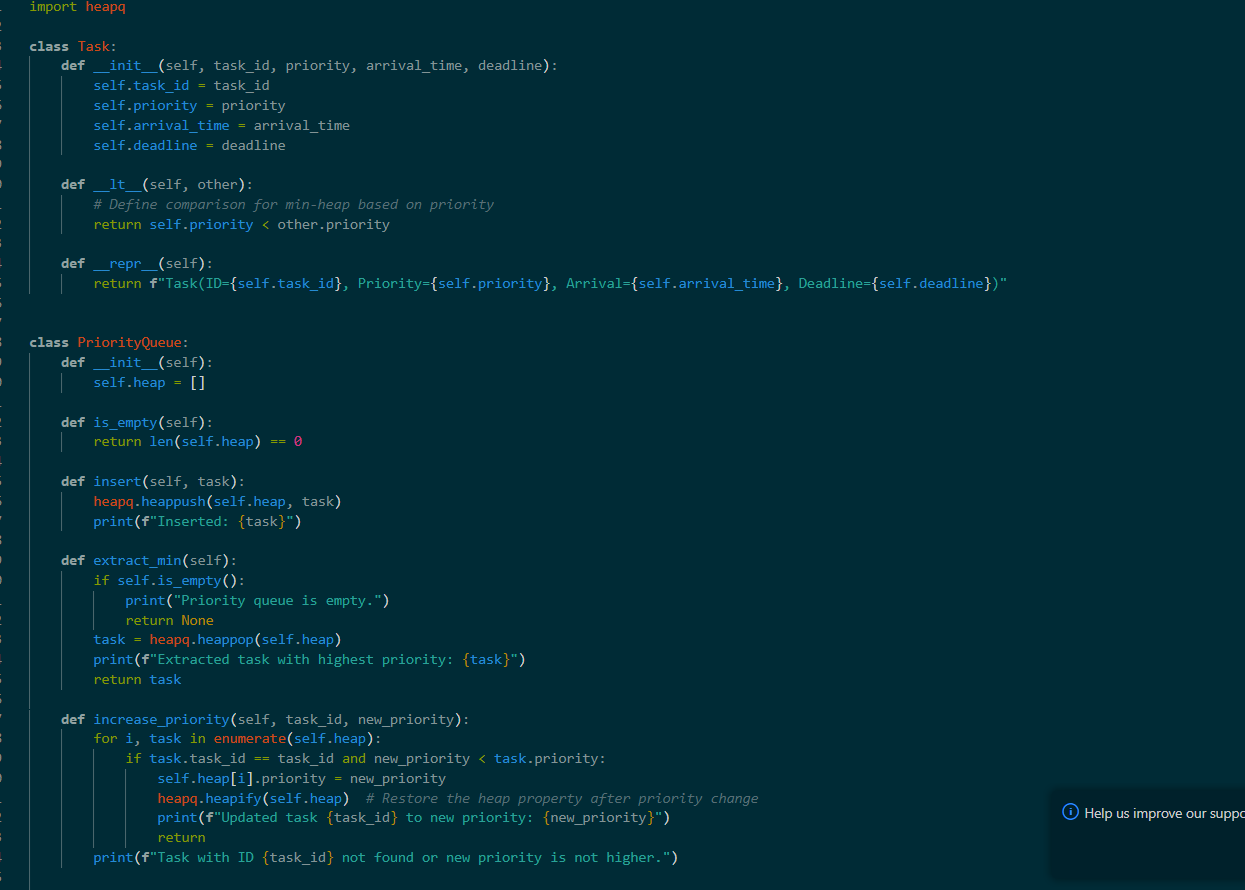
* **Task ID**: A unique identifier for each task.
* **Priority**: Determines the order in which tasks are processed. This attribute will be used by the heap for sorting tasks.
* **Arrival Time**: The time when the task enters the queue.
* **Deadline**: The time by which the task must be completed.

**3. Heap Type: Max-Heap or Min-Heap?**

The choice between a **max-heap** (highest priority first) and **min-heap** (lowest priority first) depends on the scheduling algorithm:

* If we prioritize tasks with **higher numerical priority** (e.g., for time-sensitive tasks where high priority implies urgency), we should use a **max-heap**.
* If lower numerical values indicate higher priority (e.g., deadlines where smaller values are prioritized), a **min-heap** is appropriate.

Since most priority scheduling uses min-heaps to process tasks with the highest urgency or shortest deadlines, we’ll implement a **min-heap**. This will ensure that tasks with the lowest numerical priority value (indicating the highest priority) are extracted first.



**Core Operations Analysis**

1. **Insert Operation (insert)**:
   * Inserts a new task at the end of the heap and performs a "sift up" operation to maintain the heap property.
   * **Time Complexity**: O(log n) because sift\_up can move up through the levels of the heap.
2. **Extract Max Operation (extract\_max)**:
   * Removes the root element (highest priority), replaces it with the last element in the heap, and performs a "sift down" operation.
   * **Time Complexity**: O(log n) due to sift\_down, which may need to traverse the height of the heap.
3. **Increase Key Operation (increase\_key)**:
   * Searches for the task, updates its priority, and performs a "sift up" to restore the heap property.
   * **Time Complexity**: O(n) for searching the task + O(logn) for sift\_up, resulting in O(n+logn)≈O(n) approx due to the linear search.
4. **Decrease Key Operation (decrease\_key)**:
   * Searches for the task, updates its priority, and performs a "sift down" to restore the heap property.
   * **Time Complexity**: O(n) for searching the task + O(logn) for sift\_down, resulting in O(n+logn)≈O(n) approx
5. **Is Empty Operation (is\_empty)**:
   * Checks if the heap list is empty.
   * **Time Complexity**: O(1), as it is a single check.

**Design Choices and Implementation Details**

* **Heap Structure**: A binary heap is used for the priority queue as it provides O(log n) insertion and extraction, making it efficient for a scheduler.
* **Task Identification**: Tasks are stored as pairs of (priority, task), making it easier to manage and adjust priorities.
* **Dynamic Priority Adjustments**: The increase\_key and decrease\_key methods ensure that any change in priority still maintains the heap property by adjusting the task’s position.

**Comparison and Theoretical Analysis**

This priority queue implementation supports efficient scheduling with O(log n) insertion and extraction times, making it suitable for environments requiring dynamic prioritization. The provided methods allow tasks to be adjusted without rebuilding the heap entirely, preserving efficiency.

By keeping time complexity predictable, this implementation is well-suited for real-time systems where both insertion of new tasks and extraction of high-priority tasks are frequent. The space complexity remains O(n), given the array-based heap structure.