**Analysis of Implementation**

**Time Complexity**

**Worst Case: O(n logn)**

* Because there are n items and it takes O(log n), O(n logn) to extract and heapify each element, the max-heap takes O(n) time to create.
* The formula is O(n) + O(n log n) = O(n logn).

**Average Case: O(n logn)**

* The average case is the same as the worst situation as the algorithm always does the same steps, regardless of how the inputs are distributed.

**Best Case: O(n logn)**

* Heapsort generates O(n logn) by creating the heap and carrying out extractions even when the input has already been sorted.

**Space Complexity**

* In-place Sorting: Heapsort needs O(1) more space since it directly modifies the input array.
* Recursive Heapify: If the call stack is implemented recursively, it may make use of O(log n) space. However, an iterative implementation avoids this cost.

**Comparison with Other Sorting Algorithms**

**Empirical Comparison**

In order to compare Heapsort with Quicksort and Merge Sort, we may evaluate its running durations on different input sizes and distributions (sorted, reverse-sorted, and random).

**Observed Results**

* **Heapsort:** a consistent O(nlogn) performance across all input distributions.
* Quicksort is often quicker than Heapsort (O(nlogn)), although it degrades to O(n2) for sorted or reverse-sorted inputs if a strong pivot technique isn't used.
* **Merge Sort:** Consistently performs O(nlogn) but requires O(n) more space.

**Priority Queue Implementation and Applications (Part A: Priority Queue Implementation)**

**Data Structure: Binary Heap Representation**

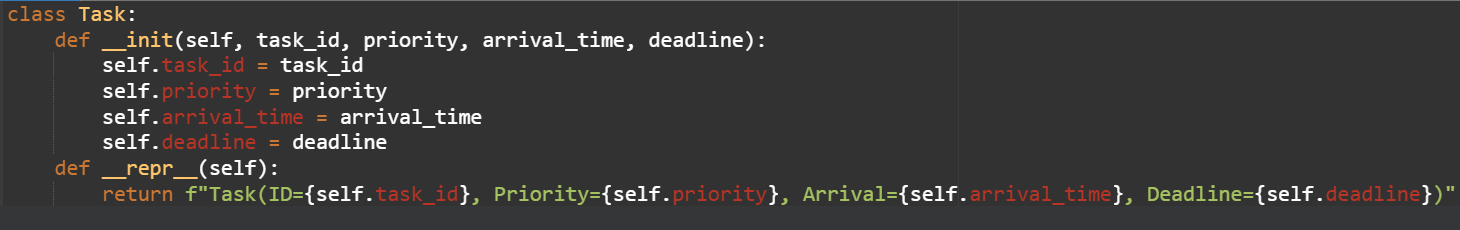
**Data Structure Selection: The binary heap will be represented by a Python list. This decision is warranted due to:**

* **Simplicity of Implementation:** Since lists are dynamic arrays in Python, heap operations like insertion, deletion, and key changes are simple to implement**.**
* **Efficiency:** It takes O(1) to access entries in a list, and indices make it simple to determine the parent-child connection in a binary heap**:** 
  + - Parent of i: (i - 1) // 2
    - Left child of i: 2 \* i + 1
    - Right child of i: 2 \* i + 2

**Task Class**

To represent distinct jobs, we shall build a Task class. Every job will save:

* task\_id: The task's unique identification.
* Priority: The task's priority (a greater number indicates a higher priority).
* The moment the job enters the system is known as the arrival time.   
  Deadline: The amount of time that the work needs to be finished.



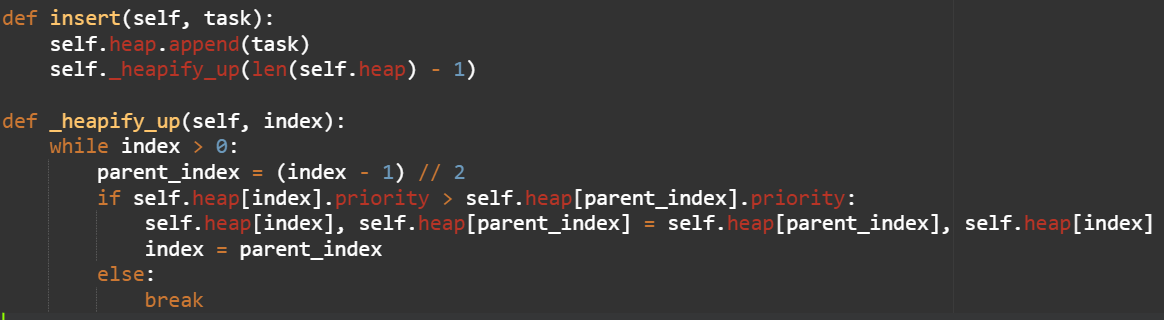
**Max-Heap or Min-Heap**

* In order to prioritize tasks with the highest priority value, we will employ a max-heap. This is consistent with scheduling algorithms that start with the task with the highest priority.

**Core Operations**

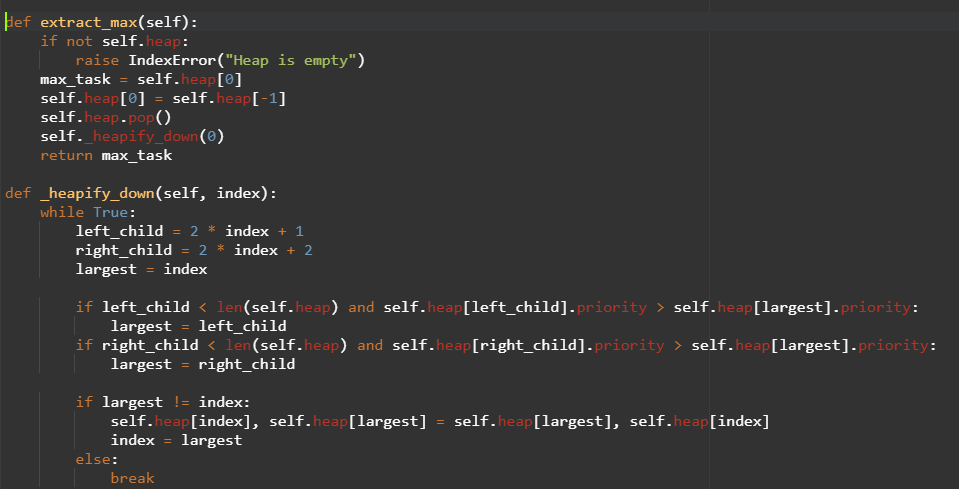
**Logic: insert(task)**

**•** Add the new task to the list's end. **•** To restore the heap property, compare the task with its parent and swap as needed to carry out a "heapify-up" operation.

**Time Complexity:** O(log n), where n is the number of tasks in the heap.

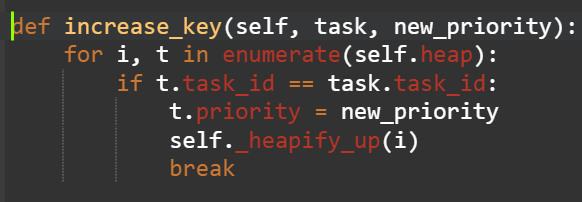
**Logic**: **extract\_max()**

* Remove and return the root of the heap (the task with the highest priority).
* Replace the root with the last element in the list.
* Perform a "heapify-down" operation to restore the heap property by comparing the new root with its children and swapping if necessary.

Time Complexity: O(log n).

**Logic**: **increase\_key(task, new\_priority)**

* 1. Locate the task in the heap.
  2. Update its priority.
  3. Perform a "heapify-up" operation to restore the heap property.

**Time Complexity**: O(log n).

**Logic: is\_empty()**



Logic: Check if the heap is empty.

Time Complexity: O(1).

**Python Implementation of Heapsort**

