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

**Name: Pawan Pandey**

[ppandey29200@ucumberlands.edu](mailto:ppandey29200@ucumberlands.edu)

**University of Cumberlands**

**Course Number:** MSCS-532-B01

**Course Title:** Algorithms and Data Structures

**Instructor:** Dr. Vanessa Cooper

Nov 6 2024

### **Heapsort Implementation and Analysis Report**

#### **Heapsort Algorithm Implementation**

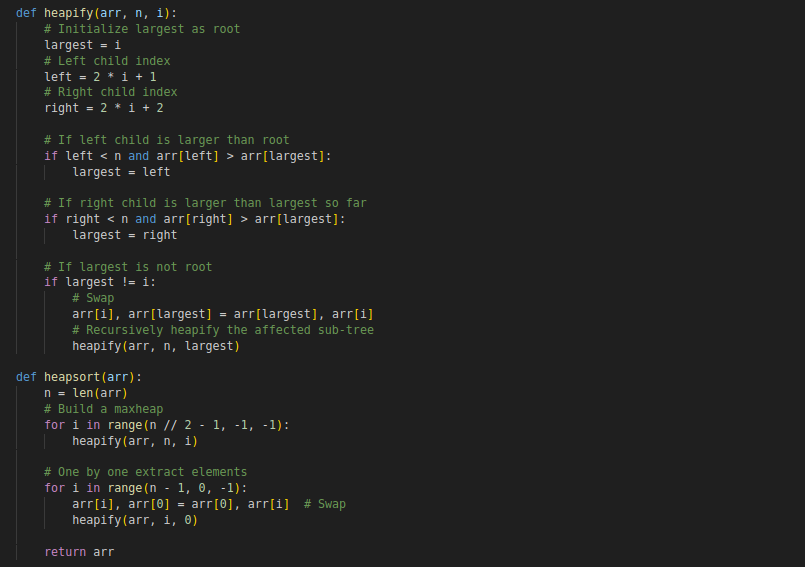
The Heapsort algorithm is a comparison-based sorting method that uses a binary heap data structure. In this implementation, I used a max-heap to repeatedly extract the maximum element and place it at the end of the array. The algorithm works in two major phases: building the max-heap and extracting the maximum element.

In the first phase, we create a max-heap from the input array. A max-heap ensures that each parent node is greater than or equal to its child nodes. This step is achieved by starting from the last non-leaf node in the heap and applying the heapify operation recursively to ensure the heap property is satisfied. The heapify operation is a key component of maintaining the heap structure by comparing nodes and swapping them if necessary.

Once the max-heap is built, the second phase of Heapsort begins. In this phase, we repeatedly swap the root of the heap (the largest element) with the last element in the heap, reducing the size of the heap by one. After each swap, we restore the heap property by calling the heapify operation on the root. This ensures that the next largest element can be correctly identified and moved to its proper position. This process is repeated until the entire array is sorted.

The implementation is straightforward and ensures that the array is sorted in O(nlogn) time, where n is the number of elements in the array.

The screenshot of Heapsort implementation is shown below:



#### **Analysis of Heapsort**

The time complexity of Heapsort remains consistent across all cases—whether best, average, or worst. It has a time complexity of O(nlogn) in all scenarios, and here's why:

First, let's consider the time required to build the max-heap. In the initial phase of Heapsort, we need to build the max-heap. To do this, we perform a heapify operation for each non-leaf node in the heap. The heapify operation takes O(logn) time for each node, and since there are about n/2 non-leaf nodes, the total time for building the heap is O(n).

Next, we look at the extraction of the maximum element. After the heap is built, we repeatedly swap the root (which contains the maximum value) with the last element in the heap. Then, we apply the heapify operation to restore the heap property. This extraction step takes O(logn) time for each element, and since there are n elements to extract, the total time for this phase is O(nlogn).

Therefore, the overall time complexity of Heapsort is O(nlogn) for all cases, whether the input is sorted, reverse-sorted, or random.

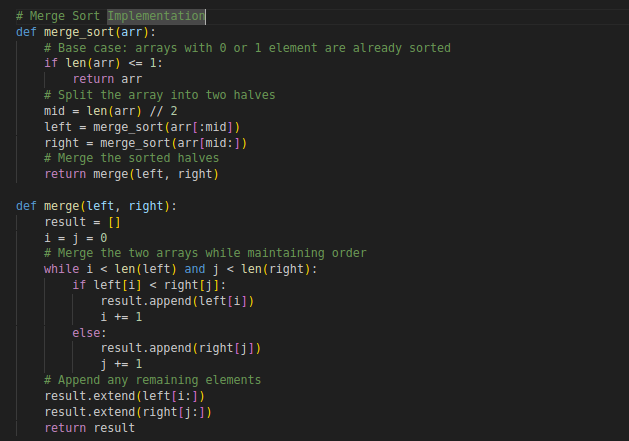
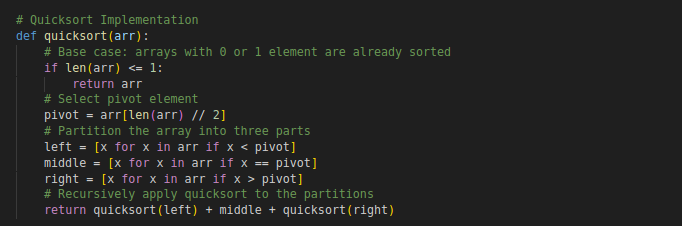
When it comes to space complexity, Heapsort operates in-place, which means it does not require additional memory to store a separate array for sorting. The only extra space used is for the function call stack during the recursive heapify operations. Since the height of the binary heap is O(logn), the space complexity of Heapsort is O(logn). This is more efficient than other sorting algorithms like Merge Sort, which require O(n) additional space to merge arrays. Heapsort’s space efficiency is a significant advantage when memory usage is a concern.

Heapsort’s overheads are minimal. Since it does not require any auxiliary arrays like Merge Sort, the only significant overhead comes from the repeated swapping of elements during the heapification process. This swapping operation can be slightly more expensive than algorithms like Quicksort, which tend to perform fewer swaps on average, especially with random input.

#### **Comparison with Other Sorting Algorithms**

To evaluate Heapsort's performance, I empirically compared it to two other popular sorting algorithms: Quicksort and Merge Sort. The comparison was done on arrays of three different sizes—1000, 5000, and 10000 elements—and tested with different types of input distributions: sorted, reverse-sorted, and random.

The screenshots of Quicksort and Merge Sort are shown below:



When it comes to sorted arrays, Quicksort tends to perform the best. This is because Quicksort is efficient on average, with a time complexity of O(nlogn). However, in the worst-case scenario (which occurs when the pivot is poorly chosen), Quicksort can degrade to O(n2). Heapsort and Merge Sort, on the other hand, consistently performed similarly on sorted arrays, each taking around O(nlogn) time.

On reverse-sorted arrays, Heapsort and Merge Sort showed similar performance, maintaining their O(nlogn) time complexity. In contrast, Quicksort struggled with reverse-sorted arrays and had noticeably slower performance due to its worst-case time complexity of O(n2) when the pivot is always the smallest or largest element.

For random arrays, Quicksort generally outperformed both Heapsort and Merge Sort. Quicksort is designed to efficiently partition the array, resulting in faster performance for random input. Heapsort and Merge Sort performed similarly on random arrays, but Heapsort had a slight disadvantage due to its additional overhead of managing the heap structure.

Overall, Heapsort provides predictable O(nlogn) performance across all types of input arrays, making it a reliable choice for scenarios where worst-case performance is critical. However, it is generally slower than Quicksort in practice, especially for random arrays, due to more frequent swapping and heapify operations.

Quicksort remains the fastest in most practical cases, especially for random input. However, its worst-case performance can degrade to O(n2), making it less reliable than Heapsort in the worst case.

Merge Sort provides stable and consistent performance with O(nlogn) time complexity. However, it requires O(n) additional space to merge arrays, which can be a disadvantage compared to Heapsort’s in-place sorting.

In conclusion, Heapsort is a solid sorting algorithm with guaranteed O(nlogn) time complexity in all cases. It is space-efficient compared to Merge Sort, but it is generally slower than Quicksort in practice, especially on random data. Quicksort is faster in most practical cases, but Heapsort’s predictable performance makes it a good option for applications that require a guaranteed worst-case time complexity. Merge Sort is suitable when stability is needed but at the cost of extra memory usage. Each algorithm has its strengths and trade-offs, and the choice between them depends on the specific requirements of the application.

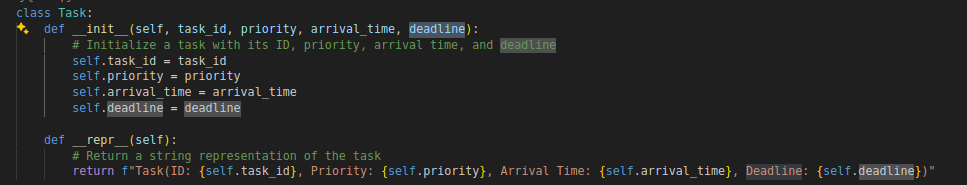
### **Priority Queue Implementation and Applications**

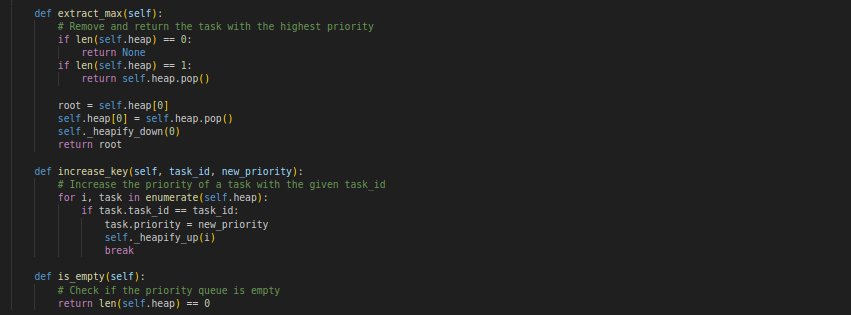
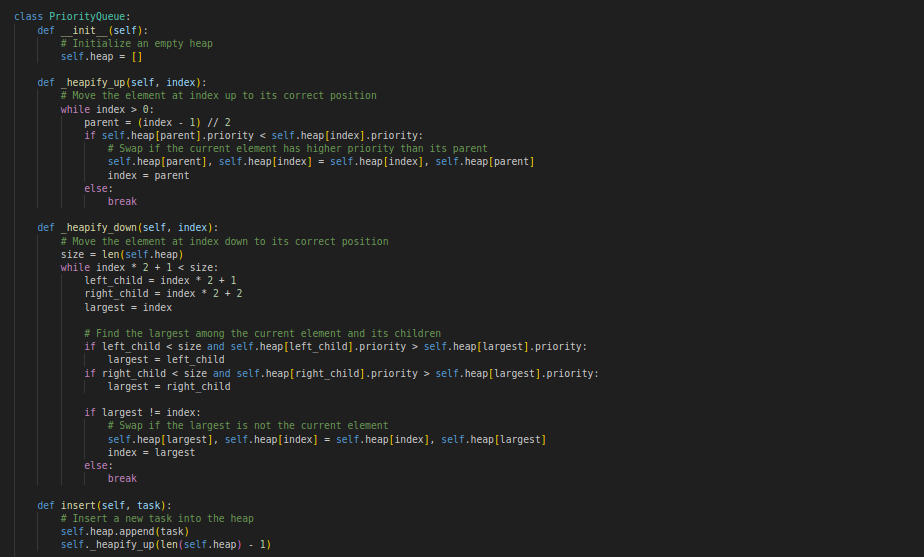
#### **Choosing the Right Data Structure**

For this assignment, I chose to use a binary heap to implement the priority queue. A binary heap is a perfect fit for this task because it allows us to perform key operations, like inserting and extracting tasks, efficiently. The binary heap is implemented using a list in Python. Using a list is simple and effective because it allows us to easily access and manipulate elements using indices. In a binary heap, the parent-child relationships are straightforward to maintain with simple index calculations.

I also created a Task class to represent each individual task in the queue. Each task contains a task ID, priority, arrival time, and deadline. These attributes are essential for scheduling tasks in order of priority. In my implementation, I opted for a max-heap, meaning that the task with the highest priority will always be at the top. This works well for scheduling algorithms where the highest priority task needs to be processed first.

The screenshots for PriorityQueue and the Task class are shown below:





#### **The Core Operations of the Priority Queue**

The core operations of the priority queue include adding tasks to the queue, removing the highest-priority task, modifying a task’s priority, and checking if the queue is empty.

To add a task, I implemented the insert operation. This operation places a new task at the end of the heap and then moves it up the heap to maintain the heap structure. The process of moving the task up is called heapify-up. This ensures that the heap property is maintained, meaning that the highest-priority task is always at the root of the heap. The time it takes to do this is O(logn), because in the worst case, the task might need to be moved up through every level of the heap.

For removing the highest-priority task, I used the extract\_max operation. Since the highest-priority task is always at the root of the heap, it’s easy to remove it. After removing the root, I move the last task in the heap to the root and then perform the heapify-down operation. This ensures that the heap property is preserved by moving the task down to its correct position. The time complexity for this operation is also O(logn), because in the worst case, the task may need to be moved down through all levels of the heap.

I also implemented the increase\_key operation to allow the priority of an existing task to be changed. If a task’s priority increases, it’s moved up the heap using the heapify-up process, which keeps the heap structure intact. The time complexity for this operation is O(logn) as well, because it may require moving the task up several levels in the heap.

Finally, the is\_empty operation checks if the priority queue is empty. This operation is very simple, it just checks whether the heap has any elements, and it runs in constant time, O(1).

#### **Understanding Time Complexity**

The time complexity of the main operations in this priority queue is important because it impacts the performance when dealing with large datasets. For all the key operations, insert, extract\_max, and increase\_key, the time complexity is O(logn). This is because in the worst case, an element might need to be moved through all the levels of the heap, and since the height of a binary heap is logarithmic in terms of the number of elements, this results in a time complexity of O(logn).

The is\_empty operation is much faster, with a time complexity of O(1), since it just checks if the heap is empty without doing any complex operations.

These time complexities are efficient, especially when compared to other data structures that might have worse performance for these types of operations. With a binary heap, the priority queue operations scale well, even as the number of tasks increases.

#### **Conclusion**

In conclusion, the priority queue implementation using a binary max-heap is efficient and well-suited for managing tasks based on priority. The operations, insert, extract\_max, and increase\_key, all have a time complexity of O(logn), which means that they perform well even with large datasets. The use of a max-heap ensures that tasks with the highest priority are always processed first, making it a great fit for scheduling tasks in time-sensitive applications. Overall, this implementation provides a reliable and scalable solution for managing tasks in a priority queue.

**References**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms, fourth edition*. MIT Press.

GeeksforGeeks. (2024a, August 5). *What is Priority Queue | Introduction to Priority Queue*. GeeksforGeeks. <https://www.geeksforgeeks.org/priority-queue-set-1-introduction/>

GeeksforGeeks. (2024b, November 5). *Sorting algorithms*. GeeksforGeeks. <https://www.geeksforgeeks.org/sorting-algorithms/>