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

Anushka Nanaware

STUDENT ID: 005042356

Department of Computer Science

University of the Cumberlands

MSCS532 - M80 Algorithms and Data Structures

Dr. Micahel Solomon

February 15, 2026

GITHUB LINK: https://github.com/ananaware/HeapDataStructures_Assignment4

https://github.com/ananaware

Heap data structures play an important role in algorithm design, especially in sorting and scheduling problems. To satisfy the ordering and structural property, a complete binary tree is needed, and binary heap is such a tree for that. When each parent node is larger than or equal to its children, it is max-heap and when the parent node is less than or equal to the children then its min-heap. Because the tree is complete, it can be efficiently stored using an array instead of pointers. This allows constant-time access to parent and child relationships using index formulas. According to Cormen et al. (2022), to make the heaps more efficient for sorting algorithms like Heapsort and the implementation of priority queues, these structures can be useful.

The goal of this assignment was to implement Heapsort, analyze its theoretical performance, compare it empirically with other sorting algorithms, and design a priority queue using a binary heap. This assignment connects theoretical complexity analysis with real implementation behavior.

Heapsort Implementation

Heapsort is an algorithm that operates in two clear stages and is comparison based which uses max-heap binary for the organization of the elements. First, it converts the input array into a max- heap then forms the heap the maximum element is removed and is then placed at the very end of the array and by repeating this process, the array becomes sorted in ascending order.

The heap is stored in an array, and the following index relationships are used:

- Left child of index $i \rightarrow 2i + 1$
- Right child of index $i \rightarrow 2i + 2$
- Parent of index $i \rightarrow \lfloor (i - 1) / 2 \rfloor$

This representation avoids pointer usage and ensures efficient memory access.

The key operation in Heapsort is heapify. This function ensures that a subtree satisfies the heap property. If a node violates the max-heap condition, it is swapped with its larger child, and the process continues recursively until the structure becomes valid. This operation is essential for both building the heap and maintaining it during extraction.

Time Complexity Analysis of Heapsort

The time complexity of Heapsort can be understood by analyzing its two phases: building the heap and extracting elements.

When building the heap using the bottom-up approach described by Cormen et al. (2022), this is where there is application of the heapify to the root starting from the last non-leaf node. Although a single heapify call can take O(log n) time, most nodes are near the bottom of the tree and require very little adjustment. Because of this, the total time to build the heap is O(n), not O(n log n). This result comes from summing the work done at each level of the tree.

When the building of the heap is done then repetitive extraction of the maximum element is done by the algorithm and each of this extraction:

- reduces the size of the heap.
- root is swapped with the element in the end.
- calls the heapify for restoration of the order.

Since heapify takes O(log n) time and is performed n times, this phase takes O(n log n), this is why the total time complexity of the Heapsort is O(n) + O(n log n) = O(n log n), and the time compexity of the Heapsort has:

- Best case
- Average case
- Worst case

Unlike Quicksort, which can degrade to $O(n^2)$ depending on pivot selection, Heapsort guarantees O(n log n) in all cases (Cormen et al., 2022). This makes it reliable when worst-case guarantees are required.

Space Complexity of Heapsort

Heapsort is considered an in-place algorithm because it uses the original array to store the heap structure and there is no need of any extra arrays that are proportional to the size of the input and that is why O(1) is the auxiliary space complexity is O(1). In the Python implementation, the recursive version of heapify uses stack space up to O(log n). But if we see from the theoretical and conceptual point of view, then in-place algorithm is the right way to treat Heapsort.

Empirical Comparison with Quicksort and Merge Sort

For understanding of the practical behavior , comparison of Heapsort was done with Merge and Quick sort by using sorted data, random data, and reverse-sorted data input distributions.

Multiple input sizes were tested to observe how performance scales. The results showed that all three algorithms followed O(n log n) growth as the input size increased. However, Quicksort and Merge Sort were often faster in practice than Heapsort. This difference can be explained by factors such as cache efficiency and the number of swaps performed.

The implemented version of Quicksort selected the middle element as the pivot. This helped avoid worst-case behavior on sorted inputs. If the first element were always chosen as pivot, sorted arrays would cause Quicksort to degrade to $O(n^2)$, as discussed by Cormen et al. (2022).

Merge Sort demonstrates stable O(n log n) performance across all input types because it always divides the array evenly. However, it requires O(n) additional memory. Heapsort was slightly slower in practice but maintained consistent and predictable performance. Its main strength lies in its guaranteed worst-case time complexity and minimal extra memory usage.

Priority Queue Design and Implementation

With the use of a binary max-heap that is stored in the python list, an implementation of the priority queue was done. This design follows the array-based heap representation described by Cormen et al. (2022). Using a list simplifies implementation because parent and child indices can be calculated directly. To represent every individual task, a Task class was designed that included:

- Optional arrival time
- Task ID
- Optional deadline

- Priority value

This structure makes the implementation suitable for scheduling simulations. A max-heap was selected so that the highest priority task is always extracted first. This matches common scheduling requirements.

Time Complexity of Priority Queue Operations

The insert operation adds a new task to the end of the heap and then performs a heapify-up procedure. In the worst case, the element may move from the leaf level to the root, and that is why O(log n) is the time complexity that we get.

The extract_max operation removes the highest priority task from the root, replaces it with the last element, and performs heapify-down and this also needs O(log n) time.

The increase_key operation first searches for the task by its ID. In the current implementation, this search takes O(n) time because a simple list is used without additional indexing structures. After updating the priority, heapify-up takes O(log n). Therefore, the overall worst-case complexity is O(n). This could be improved by maintaining a hash table that maps task IDs to indices, and the is_empty operation is used to check what the sixe of the heap is and then runs it in s O(1) time.

This assignment provided a complete understanding of heap data structures from both theoretical and practical perspectives. The empirical comparison supported theoretical expectations and highlighted practical performance differences among sorting algorithms. The priority queue implementation demonstrated how heap operations maintain efficient task scheduling with logarithmic time complexity for insertion and extraction. Overall, this

assignment strengthened understanding of algorithm analysis, complexity reasoning, and practical implementation of heap-based systems.

Reference:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press.