**Analysis Report: Min-Heap Data Structure**

**Course:** Design and analysis of Algorithms

**Student:** Mirman Nursaya SE-2438

**Executive Summary**

This report presents a comprehensive analysis and empirical validation of a **Min-Heap** data structure implementation. The theoretical time complexity of the primary operations—insert, decrease-key, and extract-min—is confirmed to be O(logN) in the worst case, and Θ(N) for build-heap. Empirical results, gathered across various input distributions up to N=105, fully validate this complexity, showing a clear linear relationship on a log-log scale. Due to partner non-response, the peer code review section focuses on general, high-impact optimization recommendations relevant to any heap implementation.

**1. Algorithm Overview**

The Min-Heap is a crucial data structure for efficient priority queue implementation. It adheres to the **Heap Property**, where the value of every node is less than or equal to the values of its children. This property ensures that the minimum element is always accessible at the root of the tree.

**Representation:** The Min-Heap is stored as a **complete binary tree**, typically realized using a dynamic array or list. This array-based storage allows for fast index arithmetic to navigate the tree: the parent, left child, and right child of an element at index i are calculated as ⌊(i−1)/2⌋, 2i+1, and 2i+2, respectively.

**Key Operations:** The core functions analyzed include:

* **insert:** Adds a new element, maintaining the Heap Property by "bubbling up."
* **extract-min:** Removes the root (minimum) element and restores the Heap Property via "sifting down."
* **decrease-key:** Modifies an element's value and restores the Heap Property by "bubbling up."

**2. Theoretical Complexity Analysis**

The time complexity of the Min-Heap is directly tied to the **height** (H) of the tree, which is H=⌊log2​N⌋. Any operation requiring an element to traverse the path from the root to a leaf (or vice-versa) results in a time complexity proportional to this height. The analysis uses Big-O (O), Big-Theta (Θ), and Big-Omega (Ω) notations.

| Operation | Worst-Case Time (O) | Best-Case Time (Ω) | Average-Case Time (Θ) | Justification |
| --- | --- | --- | --- | --- |
| **Extract-Min** | O(logN) | Ω(logN) | Θ(logN) | Removing the root and restoring the heap property requires a sequence of swaps down the height of the tree (H=logN). |
| **Insert** | O(logN) | Ω(1) | Θ(logN) | Insertion takes Θ(1) time, but the "bubble up" process may travel up to H levels. The best case occurs when the new element is smaller than its parent. |
| **Decrease-Key** | O(logN) | Ω(1) | Θ(logN) | Similar to insertion, the element is "bubbled up" at most logN levels. |
| **Build-Heap** | O(N) | Ω(N) | Θ(N) | While N/2 calls to Min-Heapify are made, the overall cost is linear. This is proven by summing the complexity contribution of nodes at each level of the tree. |

**3. Empirical Results and Validation**

Empirical testing was performed on the Min-Heap implementation by measuring three key metrics (Comparisons, Swaps, and Time) across input sizes from N=102 to N=105. All metrics were tested against Random, Sorted, and Reverse-Sorted data distributions.

**3.1 Validation of O(NlogN) Complexity**

The primary goal of the empirical validation is to confirm the theoretical O(NlogN) complexity of the dominant operations. When plotting data that follows a function y=c⋅xlogx on a **log-log scale**, the resulting curve should approximate a **straight line**.

**Figure 1: Empirical Validation: Comparisons vs. Input Size (N)**

The plot above shows the number of comparisons made during heap operations. The resulting curves for all three data types are almost perfectly linear on the log-log scale. This **visibly and quantitatively confirms** the expected theoretical complexity of O(NlogN) for the overall time spent on comparisons.

**Figure 2: Empirical Validation: Swaps vs. Input Size (N)**

The number of data swaps made during heap maintenance also follows the same O(NlogN) growth rate, as evidenced by the linear relationship in Figure 2. All three data distributions (Random, Sorted, ReverseSorted) run **parallel** to each other, demonstrating that the fundamental algorithmic efficiency is independent of input order.

**Figure 3: Empirical Validation: Time vs. Input Size (N)**

The final measurement of execution time (nanoseconds) again confirms the pattern. The nearly linear growth on the log-log scale proves the overall scalability of the Min-Heap implementation is firmly bounded by O(NlogN).

**3.2 Analysis of Input Distribution**

While the asymptotic complexity is the same for all inputs, the **constant factors** differ:

* **Reverse-Sorted Data** consistently shows the highest cost (topmost parallel line) because inserting elements requires the maximum number of comparisons and swaps to restore the heap property at every step.
* **Sorted Data** shows the lowest cost (bottommost parallel line) because the heap property is often satisfied immediately, resulting in more Ω(1) "insert" operations.