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

**Course:** Design and analysis of Algorithms

**Student:** Mirman Nursaya SE-2438

**Partner:** Ulan Maksat (Max-Heap Implementation)

**Executive Summary**

This report presents a comprehensive analysis of a Java implementation of the **Max-Heap** data structure, fulfilling the requirements for Assignment 2. The report confirms the theoretical time complexity of O(logN) for core operations and Θ(N) for buildHeap. Empirical results from the Min-Heap (used for validation) strongly align with the predicted O(NlogN) growth. The peer code review highlights the efficient use of **iterative heapify** methods but identifies a critical flaw in the **performance tracking integration**, which compromises the empirical measurement of key metrics like swaps and comparisons. Recommendations for optimization focus on resolving this tracking issue and improving code reusability through Generics.

**1. Algorithm Overview**

The Max-Heap is a priority queue implementation based on the **complete binary tree** structure. It strictly adheres to the **Max-Heap Property**: the value of any node must be greater than or equal to the values of its children. This property ensures that the maximum element is always available at the root (index 0).

**Data Structure:** The implementation utilizes an **array-based** representation (int[] heap), allowing efficient memory management and Θ(1) access to parent and child nodes using standard index arithmetic:

* Parent index: ⌊(i−1)/2⌋
* Left Child index: 2i+1
* Right Child index: 2i+2
* **Key Operations:** The analyzed implementation supports the required operations: extractMax() (retrieves and removes the maximum element), increaseKey(index, newValue) (raises the priority of an element), insert(value), and buildHeap(). It also includes the advanced merge functionality.

**2. Theoretical Complexity Analysis**

The complexity of Max-Heap operations is governed by the **height** (H) of the complete binary tree, H=⌊log2​N⌋. Operations that restore the Max-Heap Property require traversing at most H levels, resulting in logarithmic time complexity.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operation** | **Worst-Case Time (O)** | **Best-Case Time (Ω)** | **Average-Case Time (Θ)** | **Justification** |
| **Extract-Max** | O(logN) | Ω(logN) | Θ(logN) | Requires one element swap (root replacement) followed by a **sift-down** operation, which travels down the height of the heap. |
| **Insert** | O(logN) | Ω(1) | Θ(logN) | Appends the element (Θ(1)) then requires **sift-up** (bubbling up) to restore the property. The path length is bounded by H. |
| **Increase-Key** | O(logN) | Ω(1) | Θ(logN) | Similar to insert, the element is "sifted up" until the Max-Heap property is satisfied. |
| **Build-Heap** | O(N) | Ω(N) | Θ(N) | While N/2 calls to siftDown are made, the total complexity is linear, as proven by summing the work done at each level of the tree. |
| **Merge (using Build-Heap)** | O(N) | Ω(N) | Θ(N) | The implementation merges arrays and then rebuilds the heap, making the operation dominated by the linear time buildHeap step. |

**3. Empirical Results and Validation**

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

Empirical testing was performed on the associated Min-Heap implementation (my implementation) to validate the expected growth function f(N)=O(NlogN). Results were gathered across input sizes from N=102 to N=105 on various data distributions.

s

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

The resulting curves for all metrics show a linear relationship when plotted on a **log-log scale** (see Figure 1). This linearity serves as strong empirical evidence confirming that the overall complexity of running N heap operations (Insertions/Extractions) is bounded by O(NlogN), aligning perfectly with the theoretical analysis.

**3.2 Analysis of Input Distribution**

The constant factor c in the complexity function f(N)=c⋅NlogN clearly varies based on the input data:

1. **Reverse-Sorted Data:** Consistently yields the highest metric counts (topmost parallel line). This occurs because every newly inserted element must travel the full height of the heap, maximizing Swaps and Comparisons.
2. **Sorted Data:** Results in the lowest metric counts. Elements often land in a position that already satisfies the heap property, resulting in many Ω(1) operations and minimizing c.

**4. Peer Code Review & Optimization**

This section analyzes the provided Max-Heap implementation (Java files: MaxHeap.java, BenchmarkRunner.java, PerformanceTracker.java) based on algorithmic efficiency, correctness, and adherence to professional standards.

**4.1 Implementation Strengths**

1. **Iterative Heapify:** The implementation correctly uses **iterative loops** in both siftUp and siftDown. This is a significant strength, as it prevents the **stack overflow** error (StackOverflowError) that can occur with recursive solutions when dealing with very large input sizes (N≥106).
2. **Clean Logic:** The logic for extractMax, insert, and increaseKey is structurally sound, adhering to the standard textbook approach for maintaining the Max-Heap Property.
3. **Utility Operations:** The inclusion of increaseKey, buildHeap, and merge demonstrates a comprehensive understanding of advanced heap usage.

**4.2 Critical Bottleneck and Tracking Flaw**

**CRITICAL ISSUE: Missing Performance Tracker Integration**

The implementation suffers from a critical flaw: the MaxHeap.java class **does not use** the provided PerformanceTracker class to count fundamental operations (Comparisons, Swaps, Array Accesses).

* **Impact:** The BenchmarkRunner only measures **wall time** (System.currentTimeMillis()), which is unreliable due to OS scheduling, garbage collection, and CPU noise. Without tracking internal **Comparisons** and **Swaps** within the siftDown and siftUp loops, the empirical results **cannot validate the theoretical complexity** based on the primary metrics that define O(logN).
* **Recommendation:** The MaxHeap class must be refactored to accept or utilize the PerformanceTracker instance and call tracker.incrementComparisons() wherever an if (heap[parent] < heap[index]) occurs, and tracker.incrementSwaps() inside the swap() method.

**4.3 Optimization Recommendations**

1. **Use of Generics for Reusability:**
   * **Current Issue:** The class is restricted to int primitives (private int[] heap).
   * **Optimization:** Convert MaxHeap to use **Generics** (public class MaxHeap<T extends Comparable<T>>). This allows the heap to store any object type (e.g., custom Priority objects, Strings) and greatly increases the code's reusability and professional quality.
2. **Amortized Capacity Expansion:**
   * The expandCapacity() method, which doubles the array size, is O(N). While this is generally acceptable for a dynamic array (Amortized O(1) over many operations), for a fixed-size dataset scenario, it is more memory-efficient to **pre-size the array** to the maximum expected input size N in the constructor to avoid unnecessary copying operations entirely.
3. **Refactor siftDown Comparison Logic:**
   * In siftDown, three comparisons are performed sequentially to find the largest child (largest = left; then if (right < size && heap[right] > heap[largest])). While correct, this can be slightly streamlined to minimize index checks and direct comparisons within a clean loop structure, though the asymptotic complexity remains O(logN).

**5. Conclusion**

The Max-Heap implementation is **algorithmically sound** and benefits from the crucial choice of **iterative heapify**, ensuring stability at large scale. The theoretical complexity analysis is robust and supported by empirical validation from the Min-Heap data.

However, the implementation is critically flawed in its **empirical measurement design**; the absence of integrated comparison/swap tracking means the code cannot fully prove its O(NlogN) performance beyond unreliable wall time. Rectifying this tracking issue, alongside adopting Generics, is essential for transforming the current implementation into a robust, high-quality component suitable for production use.