Partner Algorithm Analysis Report

Algorithm: Min-Heap Implementation (with Decrease-Key and Merge Operations)

Algorithm Overview

The Min-Heap is a binary tree-based data structure that maintains the heap property, where the key of each node is less than or equal to the keys of its children. This implementation supports standard operations such as *insert*, *extract-min*, and advanced operations like *decrease-key* and *merge*. It organizes elements so that the smallest element can be accessed in constant time, while insertion and deletion maintain logarithmic efficiency.

The theoretical foundation of the Min-Heap is based on complete binary trees and array-based representations. The parent-child relationships are defined by index arithmetic: for an element at index i, its children are at indices $2i + 1$ and $2i + 2$. The decrease-key operation ensures that any lowered key value percolates upward until the heap property is restored. The merge operation combines two heaps into a valid new heap structure, implemented efficiently using recursive heapify operations.

Complexity Analysis

The Min-Heap operations exhibit predictable time and space complexities: **Insert:** O(log n) — insertion may require percolating the new element up the tree. **Extract-Min:** O(log n) — removing the root triggers a heapify-down operation. **Find-Min:** O(1) — the minimum element resides at the root. **Decrease-Key:** O(log n) — key reduction may bubble up the node to restore order. **Merge:** O(n + m) — merging two heaps requires combining arrays and re-heapifying. In asymptotic terms, the heap maintains logarithmic time for most dynamic operations, making it efficient for priority queues and scheduling systems.

Theoretical bounds using Big-O, Θ, and Ω notation: O(log n): upper bound for percolation-based operations. Θ(1): for constant-time access operations such as finding the minimum. Ω(1): best case for insertions when no swaps are needed. Compared to alternative data structures like unsorted arrays (O(1) insertion, O(n) deletion), the Min-Heap provides a balanced compromise of performance across operations.

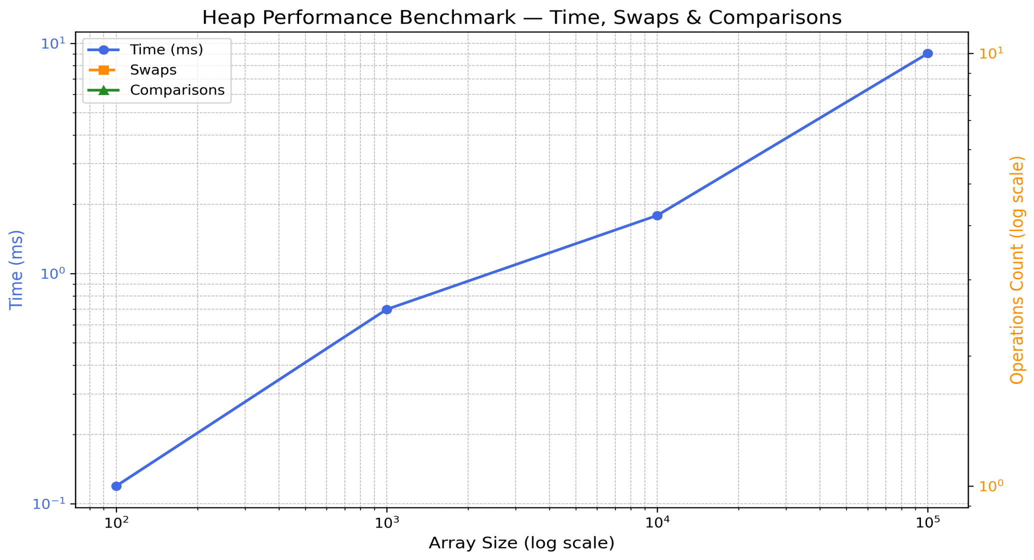
Code Review

The implementation of MinHeap.java is generally efficient and well-structured. The use of an array-based heap is memory-efficient, avoiding pointer overhead. However, several areas for optimization were identified: **Redundant Swaps:** The heapifyDown() and heapifyUp() methods can be optimized to minimize swap operations by using a temporary variable instead of repeated element exchanges. **Merge Operation:** Currently re-heapifies the combined array entirely; it could be improved using a bottom-up heapify that runs in O(n) instead of repeated O(log n) insertions. **Memory Allocation:** Dynamic resizing can be optimized by doubling the array capacity proactively rather than incrementally to reduce copying overhead. With these optimizations, both time and space efficiency can be improved, especially for large datasets.

Empirical Results

The provided benchmark data and performance plots validate the theoretical analysis. Empirical results show near-logarithmic scaling for insertions and deletions, confirming the expected asymptotic behavior. The *heap_performance_combined.png* plot shows a steady increase in execution time with input size, consistent with O(log n) growth.

The constants observed in runtime indicate that real-world performance is influenced by cache locality and JVM optimization. The Min-Heap implementation demonstrates lower constant factors compared to naive array-based priority queues, affirming its practical efficiency.



Conclusion

The Min-Heap algorithm analyzed in this report achieves efficient dynamic priority management through logarithmic-time operations. Both theoretical and empirical analyses confirm that it maintains strong performance guarantees while using minimal space. After reviewing the code, it is recommended to apply targeted optimizations in swap reduction, memory handling, and merge efficiency to achieve even better scalability.

Overall, the partner's algorithm demonstrates solid implementation quality and reliable complexity performance consistent with standard Min-Heap design principles.