Algorithm Analysis Report

SE-2407
Assignment 2 – Algorithmic Analysis and Peer Code Review
Student A: Damir Ushkempir *(*Min-Heap Implementation*)*
Partner (Student B): Aimaut Bolatkhanuly *(*Max-Heap Implementation)

---

1. Algorithm Overview

The **Min-Heap** is a binary tree–based data structure that maintains the heap property:
each parent node is **less than or equal** to its children.
It allows efficient retrieval of the smallest element in **O(1)** time and supports logarithmic-time insertion and deletion.

Key operations implemented:

- **insert(value)** – inserts a new element while maintaining the heap order via upward percolation.
- **extractMin()** – removes the smallest element (root) and restores heap order via downward percolation.
- **decreaseKey(index, newValue)** – decreases a key value and percolates it upward if needed.
- **merge(h1, h2)** – combines two heaps into a single valid heap.
- **buildHeap()** – constructs a valid heap from an unsorted array in linear time.

Additional features:

- Integrated **PerformanceTracker** for operation counting (comparisons, swaps, array accesses, memory allocations).
- Dynamic resizing of the internal array using `Arrays.copyOf()`.
- JMH-based benchmarking and CSV export for performance analysis.

Overall, the implementation is **feature-complete**, modular, and optimized for real empirical measurement.

---

2. Complexity Analysis

Operation	Best Case	Average Case	Worst Case	Explanation
insert()	Θ(1)	Θ(log n)	O(log n)	Element may move up the tree until the root
extractMin()	Θ(1)	Θ(log n)	O(log n)	Root replaced, heapified down log n levels
decreaseKey()	Θ(1)	Θ(log n)	O(log n)	Element may percolate up
buildHeap()	Θ(n)	Θ(n)	O(n)	Bottom-up heapify visits each node ≤ 2 times

Operation	Best Case	Average Case	Worst Case	Explanation
merge()	Θ(n + m)	Θ(n + m)	O(n + m)	Copies both arrays and rebuilds heap
isEmpty() / getSize()	Θ(1)	Θ(1)	Θ(1)	Constant-time operations

2.2 Space Complexity

- **Auxiliary Space: O(1)** — operations performed in-place.
- **Total Space: O(n)** — proportional to the number of elements.
- **Recursion** is avoided, so call stack usage is constant.

2.3 Recurrence Relation

The key recursive structure arises from the heapifyDown() function, which maintains the heap property by comparing a node with its children and potentially swapping it down the tree.

T(n) = T(2n/3) + O(1)

Solving this recurrence yields:

T(n) = O(log n)

This result is consistent with the logarithmic percolation depth expected for heap operations such as heapifyDown() and extractMax().

2.4 Theoretical Comparison with Min-Heap

Operation	MinHeap	MaxHeap	Complexity Match
Insert	O(log n)	O(log n)	✓
Extract	O(log n)	O(log n)	✓
Build Heap	O(n)	O(n)	✓
Merge	O(n + m)	—	✗ (extra feature in MinHeap)

Both share identical asymptotic behavior; only comparison direction differs.
MinHeap has an extra merge operation, giving it a minor functional advantage.

---

3. Code Review & Optimization

Strengths:

- Excellent modular structure and method encapsulation.
- Dynamic resizing allows flexible heap growth.
- Strong exception handling for invalid operations.

- Extensive **JUnit tests** covering edge cases and exceptions.
- Integrated benchmarking (JMH) and CSV data export.
- Uses PerformanceTracker consistently for performance metrics.

Readability:

- Code style consistent and clean.
- Logical variable names, clear loops, and control flow.
- Minor note: inline documentation (JavaDoc comments) could improve clarity.

3.2 Detected Inefficiencies

Area	Observation	Impact
heapifyDown()	Always checks both children even if left child is smaller	Slightly redundant comparisons
swap()	Updates arrayAccesses for both read and write individually	Minor overcount of metrics
merge()	Uses full array copy, then rebuilds heap — could be optimized with direct heapify	O(n+m) remains fine but could reduce constants
PerformanceTracker	Each array access increments counters individually	Adds small overhead in large benchmarks

3.3 Optimization Suggestions

1. **Early Exit in heapifyDown():**
   If left < right, skip unnecessary right check — reduces comparisons.
2. **Optimize swap():**
   Combine tracker increments into one call to prevent inflated access count.
3. **Batch metric updates:**
   Instead of per-access tracking, group updates for lower overhead in empirical runs.
4. **Enhanced merge():**
   Instead of rebuilding, use heapify on combined array directly to reduce rebuild cost.
5. **Add getMin():**
   A simple O(1) access to root could improve interface completeness.

4. Empirical Results

4.1 Experimental Setup

- **Environment:** JMH harness (Java 24)
- **Input sizes:** 1,000; 10,000; 50,000
- **Input types:** random (seeded with 42)
- **Metrics tracked:** time (ns), comparisons, swaps, memory allocations

4.2 Observations

Input Size	Avg Time (ms)	Comparisons	Swaps	Trend
1,000	~2	~11,000	~4,000	n log n
10,000	~25	~140,000	~45,000	n log n
50,000	~130	~700,000	~230,000	n log n

Input Size Avg Time (ms) Comparisons Swaps Trend

- Time grows linearly with n log n.
- insert and extractMin dominate runtime.
- Performance consistent with heap theory.
- Overhead from metric tracking visible but predictable.

4.3 Performance Plots

- **Time vs n** → smooth logarithmic curve
- **Comparisons vs n** → proportional to n log n
- **Swaps vs n** → ~constant factor × n log n
- Confirms analytical model and stability.

5. Conclusion

The **MinHeap implementation by Damir Ushkempir** is:

- Correct, stable, and well-optimized.
- Empirically and theoretically aligned with **O(n log n)** growth.
- Enhanced with useful features like **merge()** and **auto-resizing**.
- Code style is clear, readable, and professional.

Minor recommendations:

- Reduce redundant comparisons in heapifyDown.
- Optimize performance metric tracking to better reflect real runtime.
- Add brief JavaDoc descriptions for public methods.

✅ **Final Verdict:**
A highly functional, well-engineered MinHeap with robust testing, accurate complexity behavior, and good empirical validation.