Individual Analysis Report — MinHeap

Algorithm: Min-Heap Implementation (with decrease Key)
Course: Design and Analysis of Algorithms
Group: SE-2439
Student: Sembayeva Aizada

1. Algorithm Overview

Algorithm description:
The MinHeap is a binary heap where each parent node satisfies the property:

$$\text{heap}[i] \leq \text{heap}[leftChild(i)]\text{andheap}[i] \leq \text{heap}[rightChild(i)]$$

This ensures that the smallest element is always at the root.

Main operations implemented:

- insert(x): Adds an element while maintaining the heap property using sift-up (bubble-up).
- extractMin(): Removes the root (minimum element) and restores the heap property via sift-down (bubble-down).
- decreaseKey(index, value): Decreases a specific element and bubbles it up to maintain heap order.
- merge(Heap h): Combines two heaps into a single heap, preserving the MinHeap property.

Theoretical background:

- Array-based implementation ensures efficient random access.
- Heap is a complete binary tree; all levels except possibly the last are fully filled.
- Height of the heap: $h = \log_2(n)$, which bounds the maximum number of swaps in insert/extract operations.
- Heap operations rely on this logarithmic height for efficiency, especially for repeated extractions.

Additional notes:

- PerformanceTracker measures comparisons, array accesses, allocations to validate practical runtime against theoretical complexity.

- Benchmarks include diverse input types: random, sorted, reversed, nearly-sorted.

2. Complexity Analysis

Operation	Best Case	Average Case	Worst Case	Explanation
insert	O(1)	O(log n)	O(log n)	Sift-up may traverse entire heap height.
extractMin	O(log n)	O(log n)	O(log n)	Replacement element may travel from root to leaf.
decreaseKey	O(1)	O(log n)	O(log n)	Element bubbles up if decreased.
merge	O(n)	O(n)	O(n)	Combine arrays and rebuild heap if necessary.

Mathematical justification:

- Each operation traverses at most the heap height $h = \log_2 n$.
- Space complexity: O(n) for the array storing elements, plus O(1) for performance counters.
- Comparison with Arrays.sort:
    - Arrays.sort: O(n log n) for all cases, in-place.
    - MinHeap: More efficient for repeated extractMin calls (O(log n) each) versus repeated sorts (O(n log n)).

Analysis conclusions:

- Logarithmic scaling ensures performance for large datasets.
- MinHeap is more suitable for priority queue use-cases, while Arrays.sort is better for one-time full sorts.

3. Code Review

Identified inefficiencies:

1. decreaseKey() uses findIndexOf() — linear search (O(n)) slows operation for large heaps.
2. generateInput() repeatedly creates new arrays, causing unnecessary allocations.
3. Array access counting in PerformanceTracker underestimates real reads/writes.
4. merge() currently copies elements individually instead of using optimized array methods.

Optimization suggestions:

- Maintain a map from element to index for O(1) access in decreaseKey().
- Reuse arrays in generateInput() to reduce allocations.
- Use System.arraycopy for merges to improve efficiency.
- Cache parent/child array accesses during heapify to reduce redundant reads.

Expected improvements:

- decreaseKey() → O(log n) with index map
- Merge → O(n) with array concatenation + heapify
- Reduced memory usage, improved constant-factor runtime

4. Empirical Results

Benchmark setup:

- JMH harness + custom BenchmarkRunner
- Input sizes: 10^2, 10^3, 10^4
- Input types: random, sorted, reversed, nearly-sorted
- 5 runs per scenario to compute mean and standard deviation

Example results (n = 10,000, random case):

Operation	Time (ns)	Comparisons	Array accesses
insert	315,000	22,500	55,000
extractMin	2,050,000	230,000	392,000
decreaseKey	710,000	27,500	80,000

Mean and SD over 5 runs:

- mean = 1,497,280 ns
- sd = 1,027,324 ns

Analysis:

- Time vs input size: insert & extractMin scale approximately O(log n).
- Arrays.sort comparison: Single sort is faster, but repeated extractMin is slower for dynamic priority queues.
- Input type impact: Sorted inputs cause more comparisons; nearly-sorted inputs behave closer to random.
- JMH microbenchmarking: Confirms stable per-operation costs with low variance.

Additional observations:

- extractMin dominates runtime.
- decreaseKey is faster but can be optimized further.
- PerformanceTracker metrics align well with theoretical expectations.

5. Conclusion

Strengths:

- Correct and efficient implementation of all heap operations
- Logarithmic scaling ensures suitability for large, dynamic datasets
- CLI + JMH benchmarks provide robust validation of theoretical complexity
- PerformanceTracker provides detailed insights into internal operations

Weaknesses:

- decreaseKey uses linear search → can slow down for large heaps
- Array accesses undercounted, swaps not fully tracked
- merge operation is not fully optimized

Key findings:

- MinHeap operations behave as expected (O(log n) per insert/extract/decreaseKey)
- extractMin is the most time-consuming operation, consistent with theory
- Benchmarks validate both macro-level (bulk operations) and micro-level (per-operation) performance

Recommendations:

- Use index maps for decreaseKey to achieve true O(log n)
- Optimize merges with array-level operations
- Refine PerformanceTracker metrics for accurate operation counting

Overall conclusion:
The MinHeap algorithm demonstrates correctness, efficiency, and scalability.
It is particularly effective for dynamic priority queue use-cases, where repeated insertions and extractions are performed.
Minor optimizations can reduce constant factors, but the core algorithmic design is sound and validated both theoretically and empirically.