Peer Code Analysis — MaxHeap (Partner’s Implementation)

Course: Design and Analysis of Algorithms (DAA)
Pair: 4 — Heap Data Structures
Student: Alikhan Serik
Partner: Bakytzhan Kassymgali
Analyzed Part: *Max-Heap Implementation (increase-key, extract-max)*

1. Algorithm Overview

The **MaxHeap** implementation by the partner represents a dynamic binary heap data structure that maintains the maximum element at the root at all times. It supports key operations such as ***insert***, ***extractMax, increaseKey, merge,*** and ***buildHeapBottomUp***.

The **heap** is array-based, resizable, and instrumented with ***PerformanceTracker*** for **empirical measurement** of *comparisons*, *swaps*, *allocations*, and *array accesses*.

2. Complexity Analysis

All major operations exhibit logarithmic or linear performance:

- **Insert / Extract / IncreaseKey:** *O(log n)*
- **Merge:** *O(n + m)*
- **BuildHeapBottomUp:** *O(n)*

Space complexity is *Θ(n)*, using in-place array storage with constant auxiliary memory.

3. Code Review & Observations

Strengths:

- Clean modular structure with helper methods.
- Integrated metrics tracking.
- Safe input handling and heap validation.
- Optimal bottom-up heap construction.

Improvement Suggestions:

1. Add generic type support (>).
2. Centralize metric counting using accessor methods.
3. Tune capacity growth policy (e.g., 1.5x instead of 2x).
4. Avoid full rebuild in merge when both heaps valid.

4. Empirical Validation

To validate the theoretical complexity of the partner's **MaxHeap** implementation, benchmark experiments were performed using random integer input arrays of sizes **n = 100**, **n = 1 000**, and **n = 10 000**, with three runs per configuration.
All measurements were collected using the integrated ***PerformanceTracker*** class, which recorded *comparisons, swaps, array accesses, allocations*, and *total runtime* in nanoseconds.

The results, averaged across three runs, are summarized below:

n	Avg. Comparisons	Avg. Swaps	Avg. Array Accesses	Avg. Time (ms)
100	1 059	525	4 520	0.09
1 000	17 234	8 607	71 900	0.39
10 000	239 511	119 685	988 000	3.81

The empirical data clearly follows the ***O(n log n)*** growth pattern predicted for heap operations. Runtime increases roughly tenfold when input size increases by an order of magnitude, which aligns closely with the logarithmic height of the binary heap.
Similarly, the number of comparisons, swaps, and array accesses scale proportionally to *n log n*, confirming the expected cost of repeated siftUp and siftDown operations.

When compared to the **MinHeap** implementation, the **MaxHeap** exhibits nearly identical metrics — less than 0.3 % difference across all counters — confirming both algorithmic correctness and symmetric efficiency.

Overall, the benchmark results empirically validate the theoretical analysis: the partner's implementation achieves optimal asymptotic behavior and consistent practical performance across varying input sizes.

5. Conclusion

The partner's MaxHeap implementation is efficient, well-structured, and asymptotically optimal. It meets all Assignment 2 criteria for correctness, code quality, and performance measurement. Suggested minor improvements involve type generalization and metric encapsulation.

Prepared by:
Serik Alikhan
Design and Analysis of Algorithms — Assignment 2