Cross Joint Summary: Heap Data Structures

1. Introduction

This joint report presents a comparative analysis of two complementary heap data structures implemented and analyzed independently by two students: **Student A (MinHeap)** and **Student B (MaxHeap)**.
Both projects were completed within the *Design and Analysis of Algorithms* course (October 2025). The shared objective of this joint analysis is to evaluate, compare, and interpret the theoretical and empirical performance of the **MinHeap** and **MaxHeap** implementations—two closely related algorithms that form the foundation of priority queue operations and heap-based sorting mechanisms.

The purpose of this collaboration is to integrate findings from both individual analyses, emphasizing the similarities and differences in algorithmic design, time and space complexity, empirical benchmarking, code quality, and optimization strategies. The report aims to provide a comprehensive understanding of how opposite heap orientations (minimum vs. maximum) affect computational efficiency, memory usage, and practical performance behavior in Java implementations.

2. Algorithm Overview

Both MinHeap and MaxHeap are **array-based complete binary trees** that maintain a specific ordering property:

- **MinHeap:** each parent node is smaller than or equal to its children, ensuring the smallest element is always at the root.
- **MaxHeap:** each parent node is greater than or equal to its children, ensuring the largest element is at the root.

MinHeap Implementation

Student A's MinHeap includes the following core operations:

- insert(x) — inserts a new element while maintaining heap order via **sift-up (bubble-up)**.
- extractMin() — removes the smallest element (the root) and restores heap order via **sift-down (bubble-down)**.
- decreaseKey(index, value) — decreases a key's value and moves it upward if necessary.
- merge(Heap h) — merges two heaps while preserving heap properties.

Additional features include a **PerformanceTracker** that records comparisons, array accesses, and allocations to measure real-time efficiency. Benchmarks were conducted for random, sorted, reversed, and nearly-sorted input arrays. The implementation emphasizes *correctness, logarithmic scalability,* and *performance validation through empirical testing.*

MaxHeap Implementation

Student B's MaxHeap provides symmetrical functionality:

- insert(value) — inserts an element and restores order via **sift-up**.
- extractMax() — removes the root (maximum element) and performs **sift-down**.
- increaseKey(index, newValue) — increases a value and repositions it upward.

A custom **PerformanceTracker** records comparisons, swaps, allocations, and array accesses. Benchmarking was performed with larger datasets (up to n = 100,000) using both CLI and **JMH microbenchmarks**. The code uses **iterative heapify** to avoid recursion, ensuring stack safety and predictable runtime.

Key Differences and Design Features

Aspect	MinHeap	MaxHeap
Core Property	Smallest element at root	Largest element at root
Key Adjustment	decreaseKey()	increaseKey()
Extra Function	merge() operation	none (focus on stability)
Tracking Metrics	comparisons, array accesses, allocations	comparisons, swaps, array accesses, allocations
Benchmark Scope	$n \leq 10^4$	$n \leq 10^5$
Implementation Style	simple, functional	iterative, benchmark-driven
Optimizations	array reuse, System.arraycopy	cached reads, accurate counters, capacity growth

Both versions correctly preserve the heap invariant and demonstrate algorithmic completeness. The MinHeap emphasizes *functional correctness* and *memory reuse*, while the MaxHeap prioritizes *measurement accuracy* and *fine-grained performance profiling.*

3. Theoretical Complexity Comparison

Both heap types share the same asymptotic complexity for their fundamental operations, differing only in their direction of comparison (min vs. max).
Let **n** be the number of elements and **$h = \lfloor \log_2 n \rfloor$** be the heap height.

Operation	MinHeap Complexity	MaxHeap Complexity	Explanation
insert	Best: O(1) Avg: O(log n) Worst: O(log n)	Best: O(1) Avg: O(log n) Worst: O(log n)	Sift-up through heap height
extract	O(log n) in all cases	O(log n) in all cases	Element travels from root to leaf
decreaseKey / increaseKey	Best: O(1) Avg/Worst: O(log n)	Best: O(1) Avg/Worst: O(log n)	Bubble-up movement
merge (MinHeap only)	O(n)	—	Combines two heaps, rebuilds if needed

Operation	MinHeap Complexity	MaxHeap Complexity	Explanation
space complexity	O(n)	O(n)	Both are array-based complete binary trees

The height-based logarithmic scaling ensures that heap operations remain efficient even for large inputs.
The **BuildHeap** operation (not explicitly tested in MinHeap but derived theoretically) runs in Θ(n), aligning with the bottom-up heap construction principle used in MaxHeap analysis.

Discussion:
Both heaps exhibit identical theoretical behavior. The difference lies primarily in *use case* rather than *complexity*:

- MinHeap is typically used for scheduling and priority queues where smallest values have precedence.
- MaxHeap is used for sorting, median tracking, or top-k element extraction.
  In both, the logarithmic relationship between element movement and heap height confirms theoretical efficiency and predictable scalability.

4. Empirical Performance Results

Benchmark Configuration

Both reports conducted controlled experiments using multiple input sizes and patterns:

Parameter	MinHeap	MaxHeap
Input sizes	$10^2 – 10^4$	$10^2 – 10^5$
Input types	Random, Sorted, Reversed, Nearly-sorted	Random, Sorted, Reversed, Nearly-sorted
Tools	CLI + JMH harness	CLI + JMH microbenchmarks
Metrics	Time (ns), Comparisons, Array Accesses	Time (ns), Comparisons, Swaps, Array Accesses

Observed Results

- For **n = 10,000**, MinHeap's extractMin() averaged ~2.05×10^6 ns with ~392,000 array accesses.
- For **n = 100,000**, MaxHeap's extractMax() required ~1.08×10^7 ns with ~4.9×10^6 array accesses.

Both reports show that extract operations dominate runtime due to full-depth traversal during heapify-down.
In both implementations, per-operation cost increases logarithmically with input size, matching theoretical predictions.

Input distribution effects:

- Sorted inputs produced higher comparison counts due to frequent heap violations.
- Reversed inputs occasionally performed better for small n, as fewer swaps were needed.
- Nearly-sorted inputs consistently yielded intermediate performance for both heaps.

Empirical Agreement with Theory

Plots of time vs. n (from both CLI and JMH benchmarks) demonstrated near-linear behavior on logarithmic scales, confirming that **$T(n) \propto n \cdot \log n$**.
JMH microbenchmarks for MaxHeap showed per-operation times of **≈0.12–0.16 ms/op**, consistent across different input sizes — strong evidence of logarithmic complexity and efficient JVM optimization.

Comparative Insights

- **Performance Dominance:** The MaxHeap achieved greater runtime stability on larger datasets, likely due to iterative (non-recursive) heapify and efficient caching.
- **Resource Usage:** The MinHeap demonstrated slightly lower memory overhead due to its smaller benchmark scope and array reuse optimization.
- **Metric Accuracy:** The MaxHeap analysis revealed more precise measurement tracking (including swaps), whereas the MinHeap undercounted some array accesses.

Overall, both implementations validated theoretical expectations; minor differences were attributed to measurement granularity and JVM behavior rather than algorithmic design.

5. Code Quality and Maintainability

Criterion	MinHeap	MaxHeap
Readability	Clear, modular structure	Clean iterative design
Performance Tracking	Comprehensive but slightly incomplete	Detailed, CSV output, includes swaps
Optimization Coverage	Suggested reuse and System.arraycopy	Focused on caching and realistic allocation
Coding Style	Straightforward academic Java	Professionally structured Java with benchmarking tools
Maintainability	Easy to extend (merge, maps for indices)	Easy to profile and benchmark

The **MinHeap code** excels in clarity and functional completeness but could benefit from better metric accuracy and index mapping for faster `decreaseKey()`.
The **MaxHeap code** demonstrates superior instrumentation and experimental rigor, integrating modern benchmarking practices (CLI + JMH) for reproducible performance evaluation.

Both adhere well to the principles of the *Design and Analysis of Algorithms* course — correctness, asymptotic efficiency, and empirical validation — while demonstrating slightly different strengths in readability versus precision.

6. Joint Conclusion

Both the MinHeap and MaxHeap analyses confirm the correctness, scalability, and efficiency of heap-based data structures. Despite their opposite ordering principles, both exhibit nearly identical theoretical and empirical behavior:

- All primary operations — **insert**, **extract**, and **key adjustment** — operate in **O(log n)** time.
- **extractMin / extractMax** consistently emerge as the most time-consuming operations.
- Empirical benchmarks confirm tight alignment with theoretical asymptotics (**$T(n) \propto n \log n$**) for both heaps.
- Both implementations show excellent space efficiency (**O(n)** in-place storage).

Comparative Insights:

- The MinHeap implementation emphasizes algorithmic clarity and dynamic use-cases (priority queues).
- The MaxHeap implementation provides superior metric precision and benchmark scalability (up to $n = 10^5$).
  Together, they offer a holistic view of heap performance across both logical orientations.

Recommendations:

- Integrate index mapping to optimize decreaseKey() and increaseKey().
- Improve performance metric tracking consistency across both implementations.
- Extend joint experiments to include buildHeap() and heapSort() comparisons for end-to-end performance insights.

Reflection:
Through this collaborative analysis, both participants deepened their understanding of asymptotic analysis, benchmarking methodology, and optimization trade-offs in Java heap implementations. The cross-study demonstrates that while MinHeap and MaxHeap differ only in ordering, their performance profiles and optimization strategies reveal complementary strengths in real-world algorithm design.