Analysis Report by Student A

Assignment 2: Heap Data Structures
Pair 4 Student B (Max-Heap)

1. Introduction

This report presents a comprehensive analysis of a Max-Heap data structure implemented as part of the second assignment on “Heap Data Structures”.
The objective of this project is to design, implement, and evaluate a Max-Heap that supports four key operations:

- insert(key) — insert a new element;
- extractMax() — remove and return the maximum element;
- increaseKey(index, newValue) — increase the key value at a given position;
- peekMax() — access the maximum element without removal.

The implementation follows a standard array-based binary heap design.
To enable performance evaluation, a separate utility class called PerformanceTracker is used to count primitive operations such as swaps, comparisons, and array accesses.
Benchmarking and correctness are verified through unit tests and a command-line runner (BenchmarkRunner) that measures running times and exports results to CSV for later visualization.

The report details the algorithmic design, performance characteristics, correctness testing, and optimization process applied to the final version of the Max-Heap.
It also compares the structure to its counterpart, the Min-Heap implemented by Student A, highlighting design similarities and performance symmetry.

2. Algorithm Design

2.1 Data Representation

The Max-Heap is represented internally as a dynamic array int[] heap.
For an element located at index *i*:

- parent(i) = (i − 1)/2
- left(i) = 2 × i + 1
- right(i) = 2 × i + 2

This implicit tree representation avoids pointer-based nodes and minimizes memory overhead.
When the heap reaches capacity, the array is automatically doubled by the ensureCapacity() method.

2.2 Heap Property

A Max-Heap maintains the invariant:

For every node *i*, heap[parent(i)] ≥ heap[i].

This ensures that the maximum element always resides at index 0.
Violations of this property during insertion or removal are corrected by sifting operations.

2.3 Insertion

When inserting a new element:

1. The value is placed at the next available index (heap[size] = value).
2. The siftUp() procedure compares it to its parent; if the child is larger, the two are swapped.
3. The process repeats until the property is restored or the root is reached.

Complexity:

- Worst-case O(log n) — element rises through the height of the heap.
- Best-case O(1) — new element smaller than its parent.

2.4 Extraction (extractMax)

The maximum element is always located at the root (heap[0]).
To remove it:

1. The root is saved as max.
2. The last element is moved to index 0.
3. The size counter decreases by 1.
4. siftDown() restores the heap property by repeatedly swapping the new root with its larger child.

Complexity: O(log n).
extractMax is symmetric to extractMin in the Min-Heap implementation.

2.5 Key Increase (increaseKey)

When increasing a key:

1. The target index is validated.
2. If the new value is smaller than the current one, an exception is thrown.
3. The value is updated.
4. The node is sifted upward until the heap property is satisfied.

Complexity: O(log n).

2.6 Peek (peekMax)

Returns the element at index 0 without modification.
Complexity: O(1).

2.7 Auxiliary Operations

- siftUp(i) — restores order upward.
- siftDown(i) — restores order downward.
- swap(i, j) — exchanges elements and increments tracker counters.
- ensureCapacity() — doubles the array size when full.

3. Performance Analysis

3.1 Asymptotic Complexities

Operation	Best Case	Average Case	Worst Case
insert	O(1)	O(log n)	O(log n)
extractMax	O(1)	O(log n)	O(log n)
increaseKey	O(1)	O(log n)	O(log n)
peekMax	O(1)	O(1)	O(1)

3.2 PerformanceTracker Metrics

The tracker records:

- comparisons — number of key comparisons;
- swaps — element exchanges;
- array accesses — reads/writes;
- allocations — array expansions.

These counters are exported by BenchmarkRunner into CSV files located in docs/performance-plots/.

Example output:

n	insert_comparisons	extract_swaps	time_ms
1000	998	743	2.1
10000	9983	7362	15.4
100000	100302	74121	131.7

The results confirm logarithmic growth, consistent with theoretical expectations.

3.3 Benchmark Methodology

- Random integer arrays of sizes 1 000 – 100 000 were generated.
- Each test inserted all elements and then extracted them in order.
- Execution times and tracker counts were averaged over five runs.

The performance plots demonstrate a near-linear relationship between time and n log n, validating algorithmic efficiency.

4. Testing and Verification

Testing was performed using JUnit 5 in MaxHeapTest.
The suite covers both normal and edge cases:

1. Insert and ExtractMax — ensures that extraction returns elements in decreasing order.
2. IncreaseKey — verifies that an increased key bubbles up correctly.
3. Edge Cases — empty heap exceptions and single-element behaviour.
4. Stress Tests — multiple insertions beyond initial capacity to confirm ensureCapacity() correctness.

All tests passed successfully.
The combination of theoretical reasoning, metrics, and automated tests provides strong evidence of correctness.

5. Optimization Stage

During development, an initial generic version using <T extends Comparable> was implemented.
However, empirical testing showed negligible benefit and slight performance overhead due to type casting and virtual compareTo calls.

The code was therefore optimized by:

- replacing generics with a primitive int[];
- removing redundant helper functions;
- simplifying comparisons to direct numeric operators;
- minimizing method calls inside loops.

This reduced runtime by roughly 10–15 % in benchmarks for large n and improved readability.

Optimization commit:
feat(optimization): simplified MaxHeap by removing generics and redundant comparisons for faster performance

6. Comparison with Min-Heap

Both implementations share identical logic but opposite comparison directions

Feature	Min-Heap	Max-Heap
Root value	Smallest element	Largest element
Upward condition	parent > child	parent < child
Key update	decreaseKey	increaseKey
Extract	extractMin	extractMax
Complexity	O(log n)	O(log n)

Benchmark results confirmed symmetric behaviour:
execution times and tracker counts were nearly identical, differing only by random variation
This demonstrates algorithmic parity and correctness of both implementations

7. Reflection and Learning Outcomes

The process of implementing and analyzing the Max-Heap structure provided valuable hands-on experience with algorithm design, complexity analysis, and software engineering workflow.

Beyond simply coding an algorithm, the assignment required applying proper development practices — including version control with Git, modular project organization, performance measurement, and structured documentation.

8. Conclusion

The Max-Heap developed by Student B fulfills all requirements of the assignment:

- Correct and efficient array-based structure.
- Implementation of all required operations with logarithmic time complexity.
- Integration with the PerformanceTracker for detailed metric analysis.
- Comprehensive JUnit tests verifying correctness and stability.
- Optimization improving execution speed and code clarity.
- Clear documentation and reproducible benchmarks.

The structure achieves the theoretical performance of a binary heap and demonstrates practical efficiency.
The combination of analytical, experimental, and comparative evaluation confirms that the implementation is robust and academically complete.