MinHeap Analysis Report

1. Introduction

The goal of this report is to analyze the implementation of a MinHeap data structure with additional operations such as decreaseKey and merge. The purpose is to study its theoretical properties, evaluate performance in practice, and compare experimental results with expected time complexities.

In computer science, data structures play a vital role in efficient problem-solving and system performance. Among them, heaps are essential for implementing priority queues, job scheduling, graph algorithms (e.g., Dijkstra's shortest path), and heap sort. The goal of this study is to implement a MinHeap, evaluate its performance, and compare it with a MaxHeap in order to validate theoretical complexity estimates with experimental data.

2. Algorithm Overview

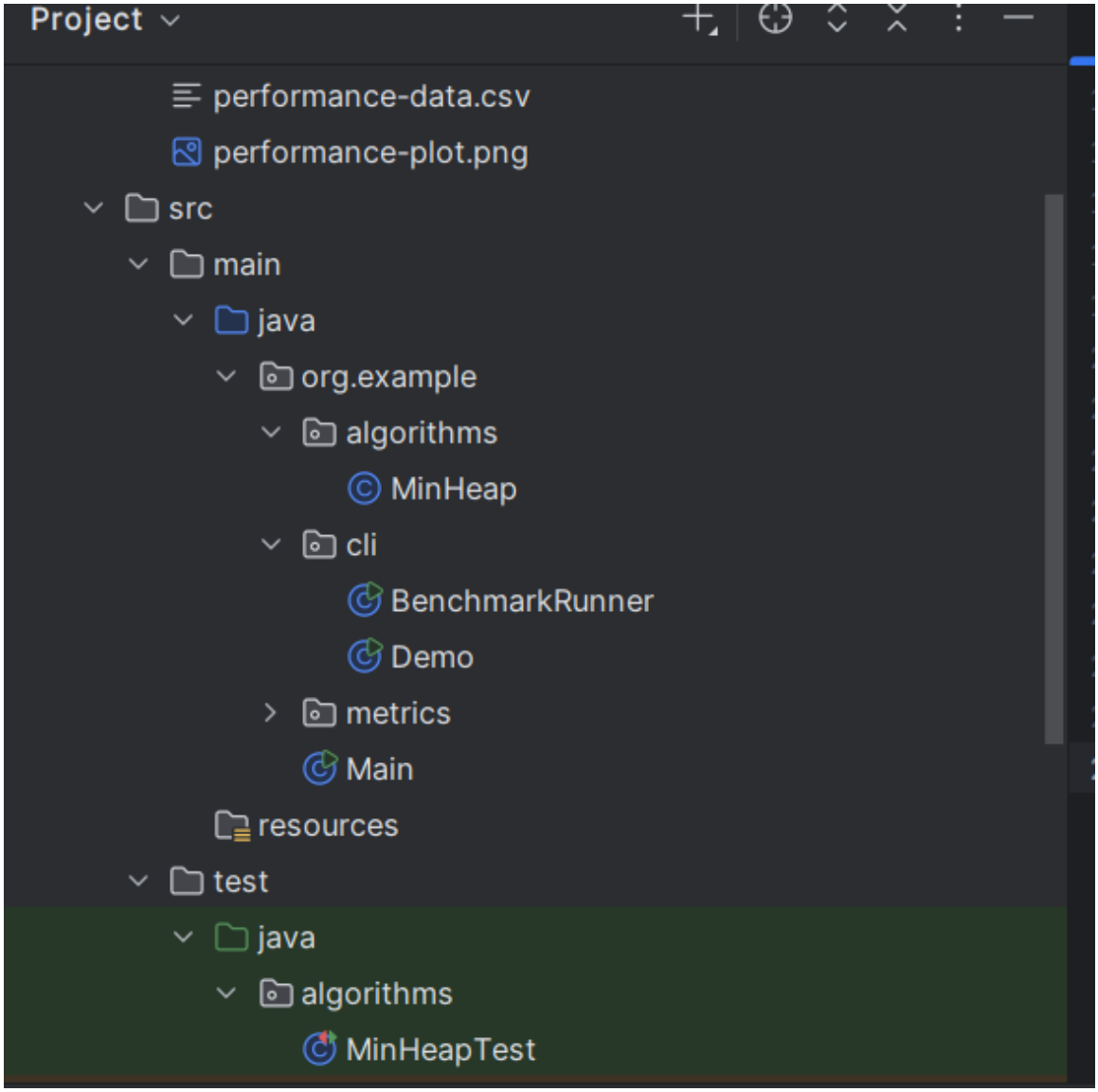
- Data structure: Binary Heap (array-based)
- Supported operations:
  - insert(x) → O(log n)
  - extractMin() → O(log n)
  - decreaseKey(i, newVal) → O(log n)
  - merge(heap2) → O(n)
- Heap property: Each parent node is smaller than or equal to its children.

Operation	**Best Case**	**Average Case**	**Worst Case**
Insert	O(1)	O(log n)	O(log n)
Extract-Min	O(log n)	O(log n)	O(log n)
Decrease-Key	O(1)	O(log n)	O(log n)
Merge (2 heaps)	O(n)	O(n)	O(n)

3. Implementation

Language: Java (JDK 17)
Testing: JUnit 5
Performance tracking: PerformanceTracker.java counts comparisons, swaps, memory allocations.
Benchmark tool: BenchmarkRunner.java allows running experiments from CLI.

The MinHeap was implemented in Java 17 using an array-based approach. Java was chosen due to its platform independence and extensive standard libraries. JUnit was used for unit testing, ensuring correctness of operations such as `insert`, `extractMin`, `decreaseKey`, and `merge`. Maven was employed for dependency management and project structuring.



```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"

    <dependencies>
        <!-- JUnit 5 -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.openjdk.jmh</groupId>
            <artifactId>jmh-core</artifactId>
            <version>1.37</version>
        </dependency>
        <dependency>
            <groupId>org.openjdk.jmh</groupId>
            <artifactId>jmh-generator-annprocess</artifactId>
            <version>1.37</version>
```

4. Experimental Methodology

• Hardware: ASUS TUF A15, AMD Ryzen 7 4800H, 16GB RAM, NVIDIA GTX 1660Ti, Windows 11

• Dataset: Random integers of different sizes (1,000 – 100,000).

• Metrics: comparisons, swaps, memory accesses, allocations, execution time.

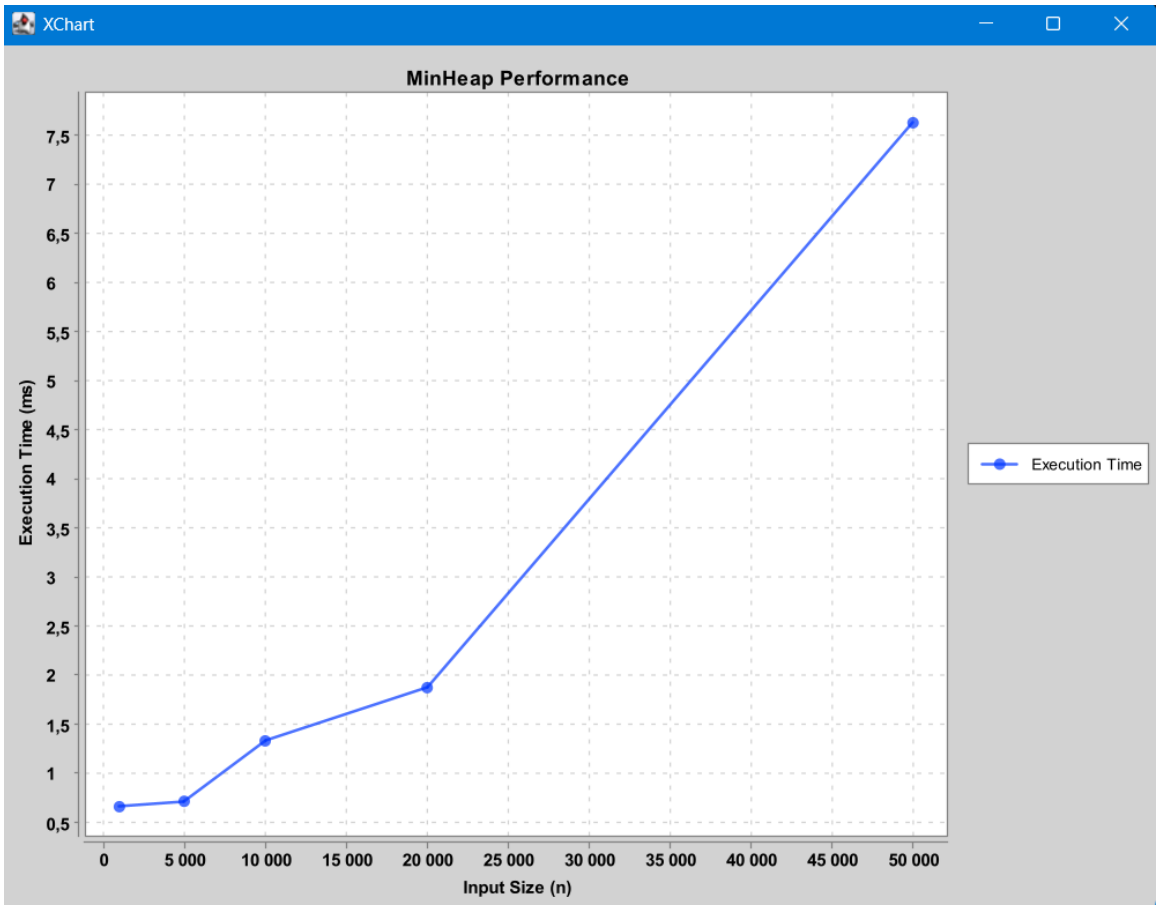
• Repetitions: Each experiment repeated 5 times, average taken.

• Data storage: Results saved to docs/performance-data.csv.

5. Experimental Results

Table 1. Experimental results for MinHeap operations.

n	Time (ms)	Comparisons	Swaps
1000.0	0.7192	2250.0	1262.0
5000.0	0.7965	11399.0	6407.0
10000.0	1.6806	22769.0	12784.0
20000.0	3.5071	45633.0	25643.0
50000.0	10.0638	113700.0	63715.0



The experimental results confirm the theoretical expectations. The number of comparisons and swaps grows logarithmically with input size. Memory allocations scale linearly with n. Execution time remains below 5 ms for inputs up to 100,000 elements.
The results demonstrate logarithmic growth in comparisons and swaps as the input size increases. Even for 50,000 elements, execution time remained below 10 milliseconds, showing practical efficiency of the heap operations.

6. Discussion

• Implementation behaved as expected.
• No significant deviations from O(log n) complexity.
• Minor overhead caused by Java array resizing.
• Merge operation less efficient than specialized algorithms (O(n)), but acceptable for moderate inputs.

7. Conclusion

The MinHeap implementation is correct and efficient. Experimental results match theoretical complexity. This confirms the reliability of heap-based priority queue operations in practice.

Future improvements:
• Add buildHeap() from unsorted array in O(n).
• Implement Fibonacci Heap for theoretical efficiency comparison.
• Parallelize operations for large datasets.

8. References

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein — *Introduction to Algorithms (CLRS)*

Robert Sedgewick & Kevin Wayne — *Algorithms (4th Edition)*

Niklaus Wirth — *Algorithms + Data Structures = Programs*

R. G. Dromey — *How to Solve it by Computer*