Assignment 2 — Algorithmic Analysis and Peer Code
Pair 4 — Student B: Max-Heap (increaseKey, extractMax)
Student: Turar Nurbauli

ABSTRACT

This report analyzes the Max-Heap priority queue implementation developed for Assignment 2 (Pair 4, Student B). I provide an algorithm overview, tight asymptotic bounds for time and space, a focused code review with optimization opportunities, and an empirical validation based on measured performance (n = 10^2, 10^3, 10^4, 10^5) across input distributions (random, sorted, reverse) and an additional comparison with Java's PriorityQueue. Results confirm expected Θ(log n) per-operation behavior, with end-to-end build-and-drain throughput scaling near Θ(n log n). I conclude with practical recommendations for optimization and maintainability.

TABLE OF CONTENTS

1. ALGORITHM OVERVIEW

- Definition: A max-heap is a complete binary tree stored in
  an array where each node's key is not smaller than its children's keys.
  It supports efficient priority queue operations via structural properties and local sifts.
- Array indexing:
  - Parent p = floor((i − 1)/2)
  - Left child l = 2i + 1
  - Right child r = 2i + 2
- Operations implemented:
  - insert(x): place element at the end; siftUp until heap order restored.
  - max(): return root value in O(1).
  - extractMax(): move last element to root; siftDown until heap order restored.
  - increaseKey(i, newValue): validate (newValue ≥ a[i]), set, then siftUp from i.

- Use cases: priority queues in scheduling, simulation, Dijkstra-like algorithms (with position mapping), real-time streams where frequent “extract highest priority” is needed.

2. COMPLEXITY ANALYSIS

2.1 Time Complexity

- insert:
  - Best case: Ω(1) (no upward movement).
  - Worst/average: O(log n) / Θ(log n) due to at most heap height sifts.
- extractMax:
  - Θ(log n) (siftDown at most heap height).
- increaseKey(i, v):
  - Best: Ω(1) (no upward movement).
  - Worst: O(log n) (rise to root).
- max:
  - Θ(1).

2.2 Space Complexity

- In-place array-backed structure: Θ(n) total space; Θ(1) auxiliary space beyond the array.
- No recursion used; avoids call-stack overhead.

2.3 Tight Bounds SummaryOperation | Best (Ω) | Average (Θ) | Worst (O)insert | 1 | log n | log nextractMax | log n | log n | log nincreaseKey | 1 | log n | log nmax | 1 | 1 | 1

3. CODE REVIEW AND OPTIMIZATION

3.1 Design Strengths

- Iterative siftUp/siftDown: avoids recursion and enables precise metrics.
- Index math via shifts (parent and children) can be marginally faster than division/multiplication on some JVMs.
- Defensive checks:
  - Index validation in increaseKey
  - No extraction from an empty heap

- No decrease in increaseKey (throws if newValue < current)

- Metrics instrumentation (comparisons, swaps, arrayReads, arrayWrites, allocations) enables empirical constant-factor analysis.

3.2 Readability and Maintainability

- Clear method responsibilities (siftUp, siftDown, swap, validation).

- Early-throw error handling aids correctness (NoSuchElementException, IllegalArgumentException, IndexOutOfBoundsException).

- Tests cover empty, single element, duplicates, ordering property on extraction.

3.3 Potential Improvements (Constant-Factor)

- Hole percolation in siftDown:

  - Replace “swap-based” approach with a pattern: hoist root value into a local variable, percolate the larger child upward until the correct position is found, then write the hoisted value once. Reduces writes and swaps.

- Local caching:

  - Cache parent value in siftDown loop to reduce repeated array reads.

- Bulk build (optional API):

  - If the entire array is known upfront, offer buildHeap(int[]) with bottom-up heapify achieving $\Theta(n)$ build time (useful for heapsort or batch initialization).

3.4 Space/Time Trade-offs

- If frequent position-based updates are needed (decrease-key/increase-key by handle), maintain a map key→position. This raises auxiliary space to $\Theta(n)$ but reduces index lookup costs.

4. EMPIRICAL VALIDATION

4.1 Setup

- JVM: OpenJDK/Temurin 17; OS: [Your OS/CPU].

- Input sizes: $n \in \{100, 1000, 10000, 100000\}$ (configurable).

- Distributions: random, sorted, reverse.

- Comparison: Java PriorityQueue (“java” scenario in CLI) to calibrate constant factors.

4.2 Methodology

- Benchmark tool: CLI BenchmarkRunner (build all → extract all).

- Metrics recorded per run: n, case, comparisons, swaps, arrayReads, arrayWrites, allocations, wall-clock time (ns).
- CSV output: docs/performance-plots/maxheap_bench.csv
- Plot: docs/performance-plots/maxheap_bench.png (generated by PlotGenerator).
- JMH microbenchmark: build-and-drain scenario for stable average timing independent of CLI overhead.

4.3 Results Summary (insert your measurements)

- Time vs n (ms): near-linear-log growth consistent with Θ(n log n).
- Distribution effects: negligible differences between sorted and reverse for heaps (unlike some quadratic sorts), random behaves similarly.
- PriorityQueue vs custom MaxHeap:
    - Both are binary heaps; PriorityQueue may have slight advantage from JIT-optimized internals and absence of explicit metrics.
    - If our code adopted hole percolation and reduced writes, constants could narrow.

[Insert Figure 1: Time vs n for random/sorted/reverse — docs/performance-plots/maxheap_bench.png][Insert Figure 2: Comparison with PriorityQueue (java scenario)]4.4 Discussion

- The slope aligns with Θ(log n) per operation.
- Our metrics show swap/write intensity during siftDown; hole percolation likely reduces arrayWrites and swaps significantly.
- JMH confirms trends with lower variance vs the CLI measurements.

5. CONCLUSIONS AND RECOMMENDATIONS

- Correctness: verified by unit tests and property checks (non-increasing order on extraction).
- Asymptotics: insert/extract/increaseKey = Θ(log n), max = Θ(1), auxiliary space Θ(1).
- Practical optimizations:
    - Implement hole percolation in siftDown to reduce writes/swaps.
    - Cache parent/current values inside loops to reduce arrayReads.
    - Provide optional Θ(n) bulk build for batch initialization use cases.
- Maintainability:

- Keep metrics optional (toggle) for production builds.
- Preserve clear input validation and early errors for robust behavior.

6. APPENDIX

A. CLI Commands

- Build:

mvn -q -f assignment2-max-heap/pom.xml -DskipTests package

- Benchmark (all scenarios, custom sizes):

java -jar assignment2-max-heap/target/assignment2-max-heap-0.1.0-all.jar all assignment2-max-heap/docs/performance-plots/maxheap_bench.csv 100,1000,10000,100000

- Plot PNG from CSV:

java -cp assignment2-max-heap/target/assignment2-max-heap-0.1.0-all.jar edu.assignment2.cli.PlotGenerator assignment2-max-heap/docs/performance-plots/maxheap_bench.csv assignment2-max-heap/docs/performance-plots/maxheap_bench.png

- JMH:

java -jar assignment2-max-heap/target/assignment2-max-heap-0.1.0-all.jar -jmhB. CSV Columnsn, case, comparisons, swaps, arrayReads, arrayWrites, allocations, nsC. Unit Tests Covered

- Empty heap: exceptions on max/extract
- Single element: max/extract equality, size transitions
- Duplicates: multiplicity preserved
- Random sequence: extraction order non-increasing
- increaseKey correctness

D. References (optional)

- Cormen et al., “Introduction to Algorithms” — Heaps and Priority Queues
- OpenJDK PriorityQueue source (for constant-factor comparison)