MaxHeap Implementation

Comprehensive Code Review & Analysis Report

Course: Design and Analysis of Algorithms

Assignment: Pair Project #4 - Min and Max Heap Analysis

Student: Saparbekov Nurdaulet(Min-Heap Implementation)

Partner: Omarov Ernar(Max-Heap Implementation)

Instructor: Khaimuldin Nursultan

Overall Assessment

The MaxHeap implementation demonstrates correct algorithmic logic for heap operations but contains several critical design flaws that limit its practical usability. The code successfully maintains the max-heap property through proper heapify operations, but architectural decisions prevent it from handling real-world scenarios.

Code Review Findings

2.1 Critical Issues

⚠ CRITICAL #1: Fixed-Size Array (Heap Overflow)

Problem Description

Severity: CRITICAL | Line: 17-19

```
public void insert(int value) {
tracker.incrementMemoryAccesses(); if (size >= heap.length) {
throw new IllegalStateException("Heap overflow"); // ❌ }
heap[size] = value; size++; heapifyUp(size - 1); }
```

Impact:

- Application crashes when inserting more than 1000 elements (default capacity)
- Unusable in production where data size is unknown
- Violates principle of dynamic data structures
- Partner's MinHeap handles unlimited elements, creating API inconsistency

Why This Happens:
The heap array is initialized with fixed size in constructor and never expanded. When size reaches capacity, further insertions fail.

✓ Recommended Fix: Dynamic Array Resizing

```
private void resize() { int[] newHeap = new int[heap.length *
2]; // Doubling strategy System.arraycopy(heap, 0, newHeap, 0,
size); heap = newHeap; } public void insert(int value) { if
(size >= heap.length) { resize(); // ✓ Expand instead of crash }
heap[size] = value; size++; heapifyUp(size - 1); }
```

Complexity Analysis:

- Worst-case single insert: O(n) when resizing occurs
- Amortized cost: O(1) per insert
- Proof: Resize sequence 2→4→8→16→... costs 2+4+8+...+n = 2n total, so 2n/n = O(1) amortized

2.2 Medium Severity Issues

⚠ MEDIUM #1: Primitive Type Only (No Generics)

Problem Description

Severity: MEDIUM | Lines: 5-6

```
private int[] heap; // ❌ Only supports integers public void insert(int value) { ... }
```

Limitations:

- Cannot store String, Double, custom objects
- Partner's MinHeap uses generics: `<T extends Comparable<T>>`
- Reduces reusability and type safety
- Need separate implementations for each type

✓ *Solution: Generic Type Parameter*

```
public class MaxHeap<T extends Comparable<T>> { private T[] heap;
@SuppressWarnings("unchecked") public MaxHeap(int capacity) { heap = (T[]) new
Comparable[capacity]; } public void insert(T value) { // Type-safe insertion } }
```

⚠ MEDIUM #2: Index-Based increaseKey API

Problem Description

Severity: MEDIUM | Line: 32

```
public void increaseKey(int index, int newValue) { // ❌ Requires index if
(index < 0 || index >= size) { throw new IllegalArgumentException("Invalid
index"); } // ... }
```

Usability Issue:

- Client code must track internal array indices
- Breaks encapsulation - internal structure exposed
- Error-prone when heap structure changes

Better API: increaseKey by value, not index

```
// Partner's approach (MinHeap): private HashMap<T, Integer> elementIndexMap;
public void decreaseKey(T oldValue, T newValue) { Integer index =
elementIndexMap.get(oldValue); // O(1) lookup if (index == null) throw new
IllegalArgumentException(); // Update value and heapify }
```

Trade-off Analysis:

Approach	Lookup Time	Space	Usability
By Index (current)	O(1)	O(1)	Poor - client must track indices
Linear Search	O(n)	O(1)	Good - but slow
HashMap (recommended)	O(1) avg	O(n)	Excellent - clean API

⚠ MEDIUM #3: Incomplete Performance Metrics

Problem Description

Severity: MEDIUM | Lines: 17, 24

```
public void insert(int value) { tracker.incrementMemoryAccesses(); // ❌ Only incremented once // But operation has multiple array accesses: heap[size] = value; // +1 access heapifyUp(size - 1); // +multiple accesses }
```

Inaccuracy:

- Counts only start of operation, not all array accesses
- Partner's MinHeap counts every array read/write
- Makes empirical analysis unreliable
- Cannot validate theoretical O(n) predictions

✓ Correct Approach

```
public void insert(int value) { if (size >= heap.length) resize(); tracker.incrementArrayAccesses(); // heap[size] write heap[size] = value; size++; heapifyUp(size - 1); // heapifyUp increments its own accesses } private void heapifyUp(int i) { while (i > 0) { int parent = (i - 1) / 2; tracker.addArrayAccesses(2); // Read child AND parent if (heap[i] > heap[parent]) { swap(i, parent); // Swap tracks its accesses i = parent; } else break; } }
```

2.3 Low Severity Issues

⚠ LOW #1: Missing peekMax() Method

Severity: LOW | Best Practice

Issue: No way to view maximum without extracting it

Standard API includes:

- `peek()` - view root without removing: O(1)
- `extract()` - remove and return root: O(log n)

Quick Fix:

```
public int peekMax() { if (size == 0) throw new IllegalStateException("Heap is empty"); return heap[0]; // O(1) }
```

3. Complexity Analysis

3.1 Theoretical Complexity

Operation	Current Implementation	Expected	Status
insert	O(log n)	O(log n)	✓ Correct
extractMax	O(log n)	O(log n)	✓ Correct
peekMax	N/A (missing)	O(1)	❌ Not implemented
increaseKey	O(log n)*	O(log n)	⚠ Wrong API (by index)

4. Comparison with MinHeap (Partner Implementation)

4.1 Feature Comparison

Feature	MinHeap (Student A)	MaxHeap (Student B)	Assessment
Architecture	Interface → Abstract → Concrete	Standalone class	⚠ **Less maintainable**
Type System	Generic <T extends Comparable<T>>	int only	⚠ **Limited reusability**
Array Sizing	Dynamic (doubling)	Fixed (throws exception)	❌ **Production blocker**
decreaseKey	✓ By value (HashMap)	❌ Not applicable	N/A
increaseKey	❌ Not applicable	✓ By index only	⚠ **Poor API**
merge()	✓ O(n) Floyd's	❌ Missing	❌ **Assignment incomplete**
peekRoot()	✓ O(1)	❌ Missing	ℹ **Minor omission**
Metrics Tracking	✓ Detailed (every access)	⚠ Incomplete (operation-level)	⚠ **Reduces accuracy**
CSV Export	✓ Full support	✓ Via tracker	✓ Adequate

4.2 Code Quality Metrics

Metric	MinHeap	MaxHeap
Lines of Code	~180	~70
Methods	15+	8
Cyclomatic Complexity	Higher (more features)	Lower (simpler)

Conclusion

Final Assessment

Category	Rating	Comment
Correctness	8/10	Core algorithms correct, missing features
Completeness	5/10	peekMax, buildHeap
Robustness	4/10	Fixed-size array is critical flaw
Code Quality	7/10	Clean structure, good separation
Performance	7/10	Correct complexity, incomplete tracking
Overall	**6.2/10**	**REQUIRES REFACTORING**

The MaxHeap implementation demonstrates **correct algorithmic understanding** of heap operations with properly functioning heapifyUp and heapifyDown methods that maintain the max-heap property. The code successfully tracks performance metrics and handles basic insertion and extraction operations.

However, the implementation suffers from **critical architectural flaw**:

1. **Fixed-size array design** prevents handling variable-sized datasets and causes runtime exceptions in production scenarios