Peer Analysis Report: Min-Heap Implementation

Based on the assignment requirements for **Pair 4: Heap Data Structures**, I'll provide a comprehensive analysis of your partner's Min-Heap implementation.

---

1. Algorithm Overview

The implementation provides a **binary min-heap** data structure using an array-based representation. It includes:

- Standard heap operations (insert, extractMin, peekMin)
- Advanced operations (decreaseKey, merge)
- Performance metrics tracking
- Comprehensive testing and benchmarking infrastructure

Theoretical Background: A min-heap maintains the heap property where each parent node is smaller than its children, enabling O(log n) insertions and O(1) minimum access.

---

2. Complexity Analysis

Time Complexity

Operation	Best Case	Average Case	Worst Case	Implemented
insert	Ω(1)	Θ(log n)	O(log n)	✓ Correct
extractMin	Ω(log n)	Θ(log n)	O(log n)	✓ Correct
buildHeap	Ω(n)	Θ(n)	O(n)	✓ Correct
decreaseKey	Ω(1)	Θ(log n)	O(log n)	⚠️ **BROKEN**
merge	Ω(n+m)	Θ(n+m)	O(n+m)	⚠️ **BROKEN**

Mathematical Justification:

Insert: Each insertion requires heapifying up from a leaf. In the worst case, this traverses the full height of the tree.

```
T(n) = O(height) = O(log n)
```

BuildHeap: Bottom-up heapification processes nodes from last non-leaf upward:

```
T(n) = Σ(i=0 to h) [n/2^(i+1)] * i = O(n)
```

This is optimal and correctly implemented.

Space Complexity

- **Auxiliary Space**: Θ(n) for the ArrayList + Θ(n) for the HashMap = **Θ(n) total**
- **In-place optimizations**: None currently implemented (assignment specifies this)

---

3. Critical Code Review & Issues

🔴 CRITICAL: Merge Corrupts Index Map

```
public void merge(MinHeap<T> other) {
    for (T element : other.heap) {
        heap.add(element);
        indexMap.put(element, heap.size() - 1);  // ❌ Overwrites existing entries
    }
    // Rebuild heap
}
```

Problem: If both heaps contain value `42`, the indexMap will only track one position, breaking all index-dependent operations.

Test this yourself:

```
MinHeap<Integer> h1 = new MinHeap<>();
h1.insert(10);
```

```
MinHeap<Integer> h2 = new MinHeap<>();
h2.insert(10);
h1.merge(h2);
// indexMap now has {10 → 1}, but heap has 10 at positions 0 AND 1
h1.decreaseKey(10, 5);  // Which 10 gets decreased? Undefined behavior!
```

⚠️ Performance Bottlenecks

1. Unnecessary HashMap Maintenance

```
private void swap(int i, int j) {
    // ...
    indexMap.put(heap.get(i), i);  // Extra HashMap operations
    indexMap.put(heap.get(j), j);  // on every swap
    arrayAccesses += 4;
}
```

Cost: HashMap operations add ~O(1) amortized overhead but with high constant factors (hashing, collision handling). For a heap that never uses decreaseKey, this is pure waste.

Optimization: Use lazy initialization:

```
private Map<T, Integer> indexMap;  // null by default

public void enableDecreaseKey() {
    indexMap = new HashMap<>();
    // Rebuild map from current heap
}
```

2. Metrics Tracking Overhead

```
if (heap.get(index).compareTo(heap.get(parentIdx)) < 0) {
    comparisons++;  // Branch + memory write on every comparison
    swap(index, parentIdx);
}
```

Impact: Metrics tracking adds ~10-15% overhead. For production use, this should be compile-time optional via a flag.

---

4. Optimization Suggestions

Time Complexity Improvements

A. Remove HashMap if DecreaseKey Unused (Expected 15-20% speedup)

```
public MinHeap(boolean trackIndices) {
    this.heap = new ArrayList<>();
    this.indexMap = trackIndices ? new HashMap<>() : null;
}
```

B. Optimize HeapifyDown (Current implementation has redundant comparisons)

```
private void heapifyDown(int index) {
    while (leftChild(index) < heap.size()) {
        int left = leftChild(index);
        int right = rightChild(index);
        int smallest = index;

        // Current: 2 comparisons per iteration
        // Optimized: Find smallest child first, then compare once
        int smallestChild = left;
        if (right < heap.size() &&
            heap.get(right).compareTo(heap.get(left)) < 0) {
            smallestChild = right;
        }

        if (heap.get(smallestChild).compareTo(heap.get(smallest)) < 0) {
            swap(index, smallestChild);
            index = smallestChild;
        } else {
            break;
        }
```

```
    }
}
```

C. Implement Floyd's Heap Construction (For better cache locality)
Current bottom-up heapify is optimal O(n), but can be improved for modern CPUs:

```
// Process nodes in breadth-first order for better cache behavior
for (int level = height; level >= 0; level--) {
    int start = (1 << level) - 1;
    int end = Math.min((1 << (level + 1)) - 1, heap.size());
    for (int i = start; i < end; i++) {
        heapifyDown(i);
    }
}
```

Space Complexity Improvements

A. Remove Index Map (Saves ~32 bytes per element)

```
Current: 40 + n*8 (ArrayList) + 64 + n*32 (HashMap) ≈ n*40 bytes
Optimized: 40 + n*8 ≈ n*8 bytes
```

B. Use Primitive Array (For Integer heaps)

```
private int[] heap;  // Instead of ArrayList<Integer>
```

Saves 16 bytes per element (Integer object overhead) + ArrayList overhead.

C. Pre-allocate Capacity

```
public MinHeap(int expectedSize) {
    this.heap = new ArrayList<>(expectedSize);  // Avoid resizing
}
```

---

5. Empirical Validation

Issues with Current Benchmarks

1. CSV Export Bug

Your code exports with comma delimiter, but results show semicolon:

```
// Code says:
writer.println("Operation,n,Time_ms,Comparisons,Swaps");
// File shows:
Operation;n;Time_ms;Comparisons;Swaps;Ñòîëáåö1
```

Cause: Likely Excel auto-converting on Windows with regional settings. Use explicit UTF-8 BOM:

```
writer.write('\ufeff');  // UTF-8 BOM
writer.println("Operation,n,Time_ms,Comparisons,Swaps,ArrayAccesses");
```

2. DecreaseKey Benchmark is Deceptive

```
for (int i = 0; i < size; i++) {
    heap.insert(i * 10);  // Unique values only!
}
```

This **hides the duplicate-handling bug**. Real-world data has duplicates.

3. Missing Memory Profiling

```
private long estimateMemoryUsage(int size) {
    return 40 + (size * 8) + 64 + (size * 32);  // Wrong!
}
```

Should use actual memory measurement:

```
Runtime runtime = Runtime.getRuntime();
long before = runtime.totalMemory() - runtime.freeMemory();
// ... create heap ...
```

```
long after = runtime.totalMemory() - runtime.freeMemory();
long used = after - before;
```

Recommended Benchmark Improvements

```
// Test with duplicate-heavy data
for (int i = 0; i < size; i++) {
    heap.insert(rand.nextInt(size / 10));  // ~10% unique values
}

// Test worst-case: reverse-sorted input
for (int i = size; i > 0; i--) {
    heap.insert(i);
}

// Test best-case: already sorted
for (int i = 0; i < size; i++) {
    heap.insert(i);
}
```

---

6. Comparison with Max-Heap

Since you implemented Max-Heap, here's the complexity comparison:

Aspect	Min-Heap (Ali)	Max-Heap (Nurzhan)	Winner
Insert	O(log n)	O(log n)	Tie
Extract	O(log n)	O(log n)	Tie
BuildHeap	O(n) ✓	O(n) ?	?
IncreaseKey	N/A	O(log n) ?	Nurzhan
Space	O(n) with waste	O(n) ?	?

Key Difference: The index map overhead in Min-Heap is unnecessary if you don't use DecreaseKey. Check if your Max-Heap avoids this.

---

7. Code Quality Assessment

Strengths

- Excellent test coverage (25+ unit tests)
- Good separation of concerns (algorithms, metrics, CLI)
- Proper exception handling
- Clean Git workflow (feature branches)

Weaknesses

- **Critical bug**: DecreaseKey fundamentally broken
- **Critical bug**: Merge corrupts index map
- Unnecessary HashMap overhead
- Missing documentation on duplicate handling
- Metrics tracking always enabled (should be optional)

---

8. Actionable Recommendations

Must Fix (Blocking Issues)

1. **Remove or completely rewrite decreaseKey**
    - Current implementation is dangerously broken
    - Either fix with index-based API or remove feature entirely
2. **Fix merge operation**
    - Document that merge only works with disjoint value sets
    - Or implement proper duplicate handling
3. **Fix CSV export encoding**
    - Add UTF-8 BOM
    - Include all metrics columns

Should Fix (Performance)

1. **Make HashMap optional** (15-20% speedup)
2. **Optimize heapifyDown** (reduce redundant comparisons)
3. **Add memory profiling** (actual measurements, not estimates)

Nice to Have

1. Implement primitive int[] version for better performance
2. Add benchmarks with duplicate-heavy data
3. Add flag to disable metrics tracking in production

Conclusion

This is a **solid implementation with critical flaws**. The core heap operations (insert, extractMin, buildHeap) are correct and efficient. However, the advanced operations (decreaseKey, merge) have fundamental design bugs that make them unusable with duplicate elements.