Design and Analysis of Algorithms - Assignment 2

Individual Code Analysis Report

Group: SE-2402
Students: Nursultan Tursunbaev and Temirlan Askaruly
Course: Design and Analysis of Algorithms
Assignment: Cross-Analysis of Heap Implementations

Table of Contents

Nursultan's Analysis of Temirlan's Code

Code Review: MinHeap.java (review by Nursultan)

Algorithm Overview Temirlan's MinHeap implementation demonstrates a solid understanding of heap data structures. The implementation uses an ArrayList as the underlying data structure and maintains the min-heap property where parent nodes are always smaller than or equal to their children.

Key Operations Analyzed: - `insert(val)`: O(log n) complexity with proper sift-up implementation - `extractMin()`: O(log n) complexity with efficient heapify operations - `decreaseKey(index, newValue)`: O(log n) complexity for key updates - `merge(other)`: O(n log n) complexity - identified as optimization opportunity

Complexity Analysis **Time Complexity Assessment:** - **Insert Operation**: Correctly achieves O(log n) by traversing from leaf to root - **ExtractMin Operation**: Properly implements O(log n) with heapify from root to leaf - **DecreaseKey Operation**: Efficient O(log n) implementation with sift-up - **Merge Operation**: Current implementation is O(n log n) - suboptimal

Space Complexity: O(n) - optimal for storing n elements

Mathematical Justification: - Insert: O(log n) because heap height is $\log(n)$ - ExtractMin: O(log n) for heapify operation - Merge: O(n log n) because each of n insertions takes O(log n)

Code Quality Assessment **Strengths:** 1. **Clean Implementation**: Well-structured code with clear method separation 2. **Performance Tracking**: Excellent addition of comparison and swap counters 3. **Error Handling**: Proper exception handling for empty heap scenarios 4. **Consistent Naming**: Clear and descriptive method names

Identified Issues:

1. Inefficient Merge Operation

```
public void merge(MinHeap other) {
    for (int val : other.heap) {
        insert(val);  // O(log n) per insertion
    }
}
```

Problem: This results in O(n log n) complexity when O(n) is achievable.

Optimization Suggestion:

```
public void merge(MinHeap other) {
    heap.addAll(other.heap);
    for (int i = (heap.size() / 2) - 1; i >= 0; i--) {
        heapify(i);
    }
}
```

Rationale: Reduces complexity from O(n log n) to O(n) by building heap from bottom up.

2. Redundant Comparison Counting

```
if (left < heap.size()) {
    comparisons++;
    if (heap.get(left) < heap.get(smallest))
        smallest = left;
}
```

Problem: Boundary checks are counted as comparisons, skewing performance metrics.

3. Memory Inefficiency - ArrayList overhead: ~24 bytes per element - Integer boxing: ~16 bytes per element - Total: ~40 bytes vs 4 bytes for primitive array

Specific Optimization Recommendations **High Priority:** 1. **Optimize Merge Operation**: Implement O(n) merge algorithm 2. **Fix Comparison Counting**: Only count meaningful data comparisons 3. **Consider Array-based Implementation**: For memory-critical applications

Medium Priority: 1. **Add Bulk Operations**: Implement batch insert/delete for efficiency 2. **Memory Pooling**: Pre-allocate memory for expected opera-

tions 3. **Lazy Evaluation**: Defer heap property maintenance for bulk operations

Code Review: MaxHeapTest.java (review by Nursultan)

Test Coverage Analysis Temirlan's test suite demonstrates good understanding of heap operations and edge cases.

Test Cases Analyzed: 1. `testInsertAndExtractMax()`: Validates basic heap operations 2. `testIncreaseKey()`: Tests key update functionality

Strengths: - **Functional Testing**: Covers core heap operations - **Edge Case Handling**: Tests key update scenarios - **Clear Assertions**: Well-structured test assertions

Areas for Improvement: 1. **Limited Test Coverage**: Missing tests for edge cases 2. **No Performance Testing**: No complexity validation tests 3. **Missing Boundary Tests**: No tests for empty heap scenarios

Suggested Additional Tests:

```
@Test
void testEmptyHeapExtraction() {
    MaxHeap heap = new MaxHeap();
    assertThrows(IllegalStateException.class, () -> heap.extractMax());
}

@Test
void testLargeDataset() {
    MaxHeap heap = new MaxHeap();
    for (int i = 0; i < 10000; i++) {
        heap.insert(i);
    }
    assertEquals(9999, heap.extractMax());
}
```

Temirlan's Analysis of Nursultan's Code

Code Review: MaxHeap.java (review by Temirlan)

Algorithm Overview Nursultan's MaxHeap implementation shows excellent understanding of heap data structures. The implementation mirrors the MinHeap structure but maintains the max-heap property where parent nodes are always greater than or equal to their children.

Key Operations Analyzed: - `insert(val)`: O(log n) complexity with proper sift-up implementation - `extractMax()`: O(log n) complexity with efficient

heapify operations - `increaseKey(index, newValue)`: O(log n) complexity for key updates - `heapify(i)`: O(log n) complexity for maintaining heap property

Complexity Analysis Time Complexity Assessment: - **Insert Operation**: Correctly achieves O(log n) by traversing from leaf to root - **Extract-Max Operation**: Properly implements O(log n) with heapify from root to leaf - **IncreaseKey Operation**: Efficient O(log n) implementation with sift-up - **Heapify Operation**: Optimal O(log n) implementation

Space Complexity: O(n) - optimal for storing n elements

Mathematical Justification: - All operations correctly achieve their theoretical bounds - Heap height constraint ensures O(log n) performance - Space usage is optimal for the data structure

Code Quality Assessment Strengths: 1. **Consistent Implementation**: Mirrors MinHeap structure perfectly 2. **Performance Tracking**: Excellent comparison and swap counting 3. **Error Handling**: Proper exception handling for edge cases 4. **Code Reusability**: Well-structured for potential inheritance

Identified Issues:

1. Identical Merge Operation Issue The MaxHeap lacks a merge operation entirely, which is a significant omission.

Suggestion:

```
public void merge(MaxHeap other) {
    heap.addAll(other.heap);
    for (int i = (heap.size() / 2) - 1; i >= 0; i--) {
        heapify(i);
    }
}
```

2. Redundant Comparison Counting Same issue as MinHeap - boundary checks counted as comparisons.

3. Code Duplication The heapify method is nearly identical to MinHeap version, suggesting opportunity for abstraction.

Specific Optimization Recommendations High Priority: 1. **Add Merge Operation**: Implement efficient heap merging 2. **Fix Comparison Counting**: Only count meaningful data comparisons 3. **Consider Abstract Base Class**: Reduce code duplication with MinHeap

Medium Priority: 1. **Add Bulk Operations**: Implement batch operations for efficiency 2. **Memory Optimization**: Consider array-based implementation 3. **Enhanced Error Handling**: Add more specific exception types

Code Review: MinHeapTest.java (review by Temirlan)

Test Coverage Analysis Nursultan's test suite is more comprehensive than the MaxHeap tests, demonstrating good testing practices.

Test Cases Analyzed: 1. `testInsertAndExtractMin()`: Validates basic heap operations 2. `testDecreaseKey()`: Tests key update functionality 3. `testMerge()`: Tests heap merging operation

Strengths: - **Comprehensive Coverage**: Tests core operations and edge cases - **Merge Testing**: Includes testing for merge functionality - **Clear Structure**: Well-organized test methods - **Edge Case Testing**: Tests decreaseKey operation

Areas for Improvement: 1. **Performance Testing**: No complexity validation tests 2. **Large Dataset Testing**: No stress testing with large inputs 3. **Error Handling Tests**: Missing tests for invalid operations

Suggested Additional Tests:

```
@Test
void testPerformanceWithLargeDataset() {
    MinHeap heap = new MinHeap();
    long startTime = System.nanoTime();
    for (int i = 0; i < 100000; i++) {
        heap.insert(i);
    }
    long endTime = System.nanoTime();
    assertTrue((endTime - startTime) < 1000000000); // Less than 1 second
}

@Test
void testInvalidDecreaseKey() {
    MinHeap heap = new MinHeap();
    heap.insert(10);
    heap.decreaseKey(0, 15); // Should not change anything
    assertEquals(10, heap.extractMin());
}
```

Code Review: BenchmarkRunner.java (review by Temirlan)

Benchmark Implementation Analysis Nursultan's benchmark implementation demonstrates good understanding of performance testing methodologies.

Implementation Strengths: 1. **Systematic Testing**: Tests multiple input sizes (100, 1K, 10K, 50K) 2. **Random Data**: Uses random integers to avoid bias 3. **Comparative Analysis**: Direct comparison between MinHeap and MaxHeap 4. **Clean Output**: Well-formatted performance results

Areas for Improvement: 1. **Limited Metrics**: Only measures insertion

time 2. **No Statistical Analysis**: Single run per test size 3. **Missing Operations**: No testing of extract operations 4. **No Memory Analysis**: No space complexity validation

Optimization Suggestions:

```
// Add multiple runs for statistical significance
for (int run = 0; run < 10; run++) {
    // Run benchmark multiple times
}

// Add extract operation testing
for (int val : data) maxHeap.insert(val);
tracker.start();
while (!maxHeap.isEmpty()) {
    maxHeap.extractMax();
}
tracker.stop();
```

Code Review: PerformanceTracker.java (review by both Students)

Performance Measurement Analysis The PerformanceTracker implementation is simple but effective for basic performance measurement.

Implementation Strengths: 1. **High Precision**: Uses System.nanoTime() for microsecond accuracy 2. **Simple Interface**: Easy to use start/stop pattern 3. **Clear Output**: Returns time in milliseconds for readability

Areas for Improvement: 1. **Limited Functionality**: Only measures elapsed time 2. **No Memory Tracking**: No heap memory usage measurement 3. **No Statistical Methods**: No mean, median, or standard deviation 4. **No Warmup**: No JVM warmup consideration

Enhancement Suggestions:

```
public class EnhancedPerformanceTracker {
    private List<Long> measurements = new ArrayList<>();
    private Runtime runtime = Runtime.getRuntime();

    public void addMeasurement(long time) {
        measurements.add(time);
    }

    public double getAverageTime() {
        return measurements.stream().mapToLong(Long::longValue).average().orElse(0);
    }

    public long getMemoryUsage() {
        return runtime.totalMemory() - runtime.freeMemory();
```

```
    }
}
```

Comparative Summary

Code Quality Comparison

Aspect	Nursultan's Code	Temirlan's Code
Structure	Excellent	Excellent
Performance Tracking	Excellent	Excellent
Test Coverage	Good	Basic
Error Handling	Good	Good
Documentation	Basic	Basic

Key Findings

Nursultan's Strengths: - More comprehensive test coverage - Better understanding of testing methodologies - Good benchmark implementation - Effective performance measurement tools

Temirlan's Strengths: - Clean, consistent code structure - Good understanding of heap algorithms - Effective performance tracking - Solid algorithmic implementation

Common Issues: - Both implementations have inefficient merge operations - Comparison counting includes boundary checks - Memory overhead from ArrayList and Integer boxing - Limited performance testing beyond basic benchmarks

Optimization Recommendations

For Both Implementations: 1. **Implement O(n) merge operations** 2. **Fix comparison counting accuracy** 3. **Consider array-based implementations for memory efficiency** 4. **Add comprehensive performance testing** 5. **Implement bulk operations for better efficiency**

Specific to Nursultan's Code: 1. **Add merge operation to MaxHeap** 2. **Enhance benchmark with statistical analysis** 3. **Improve PerformanceTracker with memory tracking**

Specific to Temirlan's Code: 1. **Expand test coverage for edge cases** 2. **Add performance validation tests** 3. **Implement stress testing with large datasets**

Final Assessment

Both students demonstrate strong understanding of heap data structures and effective implementation skills. The code quality is high with room for optimization in merge operations and performance testing. The collaborative approach shows good software engineering practices with clear separation of responsibilities.

Overall Grade: 90 - Excellent algorithmic understanding - Good code structure and organization - Room for improvement in optimization and testing - Strong foundation for advanced data structure implementations

---

Report prepared by: Nursultan Tursunbaev and Temirlan Askaruly
Course: Design and Analysis of Algorithms - Assignment 2