Peer Analysis Report: MinHeap Implementation

Student: Saniya Yerzhan
Partner's Algorithm: MinHeap(Ismail Zhanatay)
Date: 05.10.2025

Algorithm Overview

The **MinHeap** is a binary heap data structure in which the parent node is always smaller than its child nodes. This property allows for efficient retrieval of the minimum element in **O(1)** time and supports insertion and deletion operations in **O(log n)** time. The MinHeap implementation in this project is designed to track **algorithmic metrics** such as comparisons, swaps, and array accesses using the PerformanceTracker class.

Implementation Details

The MinHeap implementation supports two constructors:

1. **Empty heap with fixed capacity**:

MinHeap(int capacity, PerformanceTracker tracker)

- Initializes an empty heap array of the specified capacity.
- Tracks metrics using PerformanceTracker.

2. **Bulk build from array**:

MinHeap(int[] array, PerformanceTracker tracker)

- Clones the input array.
- Builds the heap in **O(n)** time using buildHeap().
- Calls heapifyDown starting from the last parent node up to the root.

3. Core Operations

- **Insert**: Adds an element to the heap, ensures capacity, then performs heapifyUp to maintain the heap property.
- **ExtractMin**: Removes the smallest element (root), replaces it with the last element, and performs heapifyDown.
- **Peek**: Returns the minimum element without removing it.
- **HeapifyUp / HeapifyDown**: Restore the heap property after insertion or extraction.
- **Swap & Array Access Tracking**: Each swap and array access is recorded via PerformanceTracker for empirical performance analysis.

4. Performance Metrics

The PerformanceTracker class collects:

- **Comparisons** – number of element comparisons.
- **Swaps** – number of element swaps.
- **Array Accesses** – number of times the array is read/written.
- **Execution Time** – in nanoseconds and milliseconds.

These metrics allow quantitative analysis of heap operations and evaluation of optimization improvements.

5. Benchmarking

Two benchmarking methods are used:

1. **CLI Benchmark (BenchmarkRunner)**:
    - Generates random arrays of specified size.
    - Performs a **bulk build** followed by repeated extractMin() calls until the heap is empty.
    - Writes results to CSV (results.csv / newresults.csv).
2. **Microbenchmark with JMH (MinHeapJmh)**:
    - Benchmarks heap operations on arrays in different states:

    - Random, sorted, reverse-sorted, and nearly sorted.
    - Measures **average execution time per operation** using multiple iterations for statistical accuracy.

Complexity analysis

Heaps are specialized tree-based data structures that efficiently support priority queue operations. In this analysis, I examine the **MaxHeap** implementation from my code and compare it with my partner's **MinHeap**. I derive the **time and space complexities** for core operations, justify them mathematically using Big-O, Θ, and Ω notations, and validate them against **benchmark results** provided in benchmark_all.csv.

MaxHeap Operations and Complexity

2.1 Insertion

In MaxHeap, insertion involves:

1. Placing the new element at the end of the array.
2. "Heapifying up" by repeatedly swapping with the parent until the heap property is restored.

- **Best case ($\Omega(1)$)**: The new element is smaller than its parent; no swaps required.
- **Worst case ($O(\log n)$)**: The element is larger than all ancestors and moves from leaf to root.
- **Average case ($\Theta(\log n)$)**: On random input, it moves up roughly $\log_2(n)$ levels.

Example: Inserting 50 into a heap [100, 40, 30, 20] requires comparing with the parent node. If 50 < 40, no swap is needed; if 50 > 40, one swap occurs.

Time Complexity: Best $\Omega(1)$, Average $\Theta(\log n)$, Worst $O(\log n)$
Space Complexity: $O(1)$ (in-place swaps only)

2.2 Extract Max

Extraction involves:

1. Removing the root (heap[0]).
2. Moving the last element to the root.
3. "Heapifying down" to maintain the heap property.

- **Best case ($\Omega(1)$)**: Root is larger than children; no swaps.
- **Worst case ($O(\log n)$)**: Element moves from root to leaf.
- **Average case ($\Theta(\log n)$)**: Element moves partially down the tree depending on subtree structure.

Example: Extracting the max from [100, 50, 30, 20] swaps 20 with 50 to maintain the heap property.

Time Complexity: Best $\Omega(1)$, Average $\Theta(\log n)$, Worst $O(\log n)$
Space Complexity: $O(1)$

2.3 Build Heap

Building a heap from an array of n elements uses **bottom-up heapify**:

For $i=(n/2)-1$ down to 0, call heapify(i)

- **Best/Average/Worst cases:** $O(n)$ time, because lower-level nodes require fewer swaps and higher-level nodes are fewer in number.
- **Space Complexity:** $O(n)$ for the array; heapify is in-place.

Mathematical Justification:

T(n)=∑(n/2 to i =0)−O(height of subtree)=O(n)

Example: For [3, 9, 2, 1, 4, 5], heapify starts at index 2 and moves up to index 0, creating a valid MaxHeap in linear time.

2.4 Increase Key

Increasing a key involves setting a new value at index i and heapifying up. Complexity is **identical to insertion**: $O(\log n)$ worst case, $\Theta(\log n)$ average, $\Omega(1)$ best.

Empirical Observations (From `benchmark_all.csv` (MaxHeap):

Size	Type	Swaps	Comparisons
100	nearly	360	765
1,000	nearly	5,357	11,236
10,000	nearly	70,898	147,329
100,000	nearly	875,454	1,806,394

Observations:

- MaxHeap performs fewer swaps when elements are **partially ordered** (e.g., nearly sorted).
- Comparisons roughly follow $O(n \log n)$, consistent with theory.

Comparison with MinHeap

Partner’s MinHeap CSV (new results):

Size	Swaps	Comparisons	Array Accesses
100,000	1,574,479	3,019,044	1,574,479

Comparison with MaxHeap (100,000 elements):

- MaxHeap swaps: 875,454
- MinHeap swaps: 1,574,479

Efficiency Insight: MaxHeap performs **fewer swaps**, reducing both array accesses and execution time.

Mathematical Justification:

MaxHeap swaps / MinHeap swaps$\approx 0.556 \Longrightarrow$MaxHeap is more efficient

- Time complexity remains O(log n) for both heaps, but **practical performance improves** when the input is partially ordered because fewer heapify operations are needed.

Conclusion

- **MaxHeap** is theoretically optimal: O(log n) per operation, O(n) for building.
- Empirically, MaxHeap **reduces swap counts**, particularly on partially ordered inputs, improving runtime.
- **MinHeap** matches asymptotic complexity but has **higher practical swap counts**, as seen in 100,000-element benchmarks.
- Input order significantly affects **real-world heap efficiency**, despite identical theoretical Big-O behavior.

Code Review: MinHeap Implementation

1. Inefficient Section #1: ensureCapacity() During Insertion

Issue:

- In insert(int value), when the heap is full, the array is doubled and all elements are copied:

```
private void ensureCapacity() {
    if (size == heap.length) {
        int[] newHeap = new int[heap.length * 2];
        System.arraycopy(heap, 0, newHeap, 0, heap.length);
        heap = newHeap;
    }
}
```

- For large arrays, e.g., 100,000 elements, this is **expensive in both time and memory**.

Proof from CLI Benchmark:

Input Size	Array Accesses	Time (ms)
100,000	1,574,479	24

- Many array accesses are caused by repeated doubling and copying.

- Nearly-sorted arrays should be efficient, but large inserts trigger multiple `System.arraycopy` operations.

Optimization Suggestion:

- If the total size is known, allocate slightly more upfront:

```
this.heap = new int[(int)(expectedSize * 1.1)]; // 10% extra
```

- This reduces **memory reallocations** and improves performance for large heaps.

Effect:

- Fewer memory copies → fewer array accesses → faster insertions.
- Keeps the space complexity **O(n)**.

Inefficient Section #2: heapifyDown Swaps

Issue:

- heapifyDown swaps at every level instead of storing the node temporarily:

```
private void heapifyDown(int i) {
    while (true) {
        int left = leftChild(i);
        int right = rightChild(i);
        int smallest = i;

        if (left < size) { if (heap[left] < heap[smallest]) smallest = left; }
        if (right < size) { if (heap[right] < heap[smallest]) smallest = right; }

        if (smallest == i) break;
        swap(i, smallest);
        i = smallest;
    }
}
```

- Each swap involves **3 array accesses** (read/write temp + two assignments).
- For 100,000 nearly-sorted elements, CLI shows **1,574,479 swaps** – a large number caused by unnecessary repeated swaps.

Example:

- Node value 50 moves down multiple levels, swapping each time with a child.
- Instead, we could **store the node in a temp variable** and shift children up until the correct position is found:

```
int temp = heap[i];
while (left < size) {
    int smallest = left;
    if (right < size && heap[right] < heap[left]) smallest = right;
    if (heap[smallest] >= temp) break;
    heap[i] = heap[smallest]; // shift child up
    i = smallest;
    left = leftChild(i);
    right = rightChild(i);
}
heap[i] = temp; // place original value
```

Effect:

- Reduces swaps dramatically.
- For the same 100,000-element nearly-sorted input, swaps could drop from **1,574,479 to ~1,200,000**, improving runtime from **24 ms to ~18–20 ms** in your CLI benchmark.

Empirical Results – MinHeap Analysis

1. Performance Plots (Time vs Input Size)

Using your CLI benchmark (newresults.csv) and JMH results (jmhresults.csv), we can visualize how **execution time scales** with input size.
We focus on four types of input arrays: **random, sorted, reverse, and nearly sorted**.

Observations from the CLI benchmark:

Input Size	Random	Sorted	Reverse	Nearly Sorted
100	0 ms	0 ms	0 ms	0 ms
1,000	1 ms	1 ms	1 ms	1 ms
10,000	4 ms	4 ms	4 ms	4 ms
100,000	24 ms	24 ms	24 ms	24 ms

JMH results (nearly sorted vs random for larger arrays):

Input Size	Nearly Sorted	Random
100	0.002 ms	0.002 ms
1,000	0.034 ms	0.037 ms
10,000	0.930 ms	1.120 ms
100,000	10.483 ms	15.381 ms

From these, **time grows roughly linearly with n log n** as expected for heap operations (bulk build is O(n), extract operations are O(log n) per element).

2. Validation of Theoretical Complexity

Theoretical expectations:

- **BuildHeap**: O(n)
- **Insert/ExtractMin**: O(log n) per operation
- **Overall sort using heap**: O(n log n)

Evidence from CLI and JMH results:

- Nearly sorted arrays consistently require **slightly fewer swaps** and comparisons, confirming that input order affects the constant factors but not asymptotic complexity.
- For 100,000 elements:
  - CLI: swaps ≈ 1,574,479, comparisons ≈ 3,019,044
  - JMH: 10–15 ms per run for build + extract
- This aligns with O(n log n), since $\log_2(100{,}000) \approx 17$, so 100,000 × 17 ≈ 1,700,000 theoretical operations, matching empirical swap counts.

Example validation:

- Nearly sorted input: fewer reorders → fewer swaps (1,574,479 vs ~1.7M expected)
- Random input: closer to worst-case, slightly higher swap/comparison counts (~1,574,479–1,590,000)

3. Analysis of Constant Factors and Practical Performance

Key insights:

1. **HeapifyDown and HeapifyUp dominate execution time.**
   - Every extract/insert triggers a logarithmic number of swaps and comparisons.
2. **Input type matters for constants:**

    - Nearly sorted arrays reduce the number of swaps by ~5–10% compared to random arrays.
    - Sorted input arrays are slightly faster in practice because fewer heapify operations are triggered.
3. **Memory usage is stable:**
    - All runs report ~1.2–10 MB used memory, matching theoretical O(n) space for the array.
4. **CLI vs JMH differences:**
    - CLI times are slightly higher because JMH runs multiple iterations with warmup and avoids JVM startup overhead.
    - Constant factors in JMH are more precise (ms/op), showing nearly linear growth for large n.

Practical takeaway:

- MinHeap is **efficient for both nearly sorted and random data**, confirming that theoretical O(n log n) matches observed empirical performance.
- Optimizations like **pre-allocating extra capacity** or **reducing array accesses in heapify** could further improve constants, especially for large arrays (>100,000 elements).

Conclusion

Key Findings

From analyzing and testing the MinHeap code, I can see several clear patterns:

1. **Performance scales as expected**
    - As the number of elements increases, the time taken grows roughly in line with the theoretical O(n log n) complexity.
    - Small arrays run almost instantly, while the largest arrays (100,000 elements) still finish in a fraction of a second in JMH benchmarks.
2. **Input type affects performance**
    - Nearly sorted or sorted arrays require fewer swaps and comparisons than random or reverse arrays.
    - This shows that the implementation handles partially ordered data more efficiently, even though the asymptotic complexity remains the same.
3. **Heap operations dominate runtime**
    - insert and extractMin operations are where most of the comparisons and swaps happen.
    - Bulk building of the heap from an array is fast and close to linear time, which aligns with theory.
4. **Memory usage is stable**
    - Memory grows linearly with input size and there are no unexpected spikes, confirming that the heap uses space efficiently.

Optimization Recommendations

Based on the findings, a few practical improvements could make the heap even faster:

1. **Pre-allocate extra space**
    - By increasing the initial heap array size by about 10%, we can avoid frequent resizing during inserts.

- This is especially helpful when the number of elements grows unpredictably.

2. **Improve heapify efficiency**
    - Minor tweaks in heapifyUp and heapifyDown can reduce unnecessary swaps and array accesses.
    - For example, combining some comparisons with swaps or stopping early when the heap is already correct can save time.
3. **Handle partially sorted inputs smartly**
    - Adding early checks in heapify could prevent unnecessary work for nearly sorted arrays.
    - These changes don’t affect O(n log n) complexity but can reduce execution time in practical cases.

Overall Conclusion

The MinHeap code is solid, works as intended, and performs well across all tested input sizes and types. The suggested optimizations are about **fine-tuning constant factors** rather than changing the fundamental algorithm. Implementing them would make the heap slightly faster and more efficient, particularly for large datasets or real-world scenarios where data may already be partially ordered.