Algorithmic Analysis Report — MinHeap

Student: [Zhetpyspayev Adilbek]

Partner's Algorithm: MinHeap Implementation, (Ulan Orazbek)

Course: Algorithmic Analysis

1. Algorithm Overview

The MinHeap is a complete binary tree data structure where each parent node is less than or equal to its children, ensuring the minimum element is always at the root. This property allows efficient retrieval of the smallest element in constant time. The implementation supports core operations such as insert, extractMin, peekMin, decreaseKey, merge, and buildHeap. A key feature is the use of a position map (hashmap) to track element indices, enabling efficient decreaseKey operations in logarithmic time.

In contrast, the MaxHeap maintains the maximum element at the root with similar operations but prioritizes the largest values. Both heaps are stored as arrays and rely on heapify-up and heapify-down procedures to maintain heap invariants after modifications.

2. Complexity Analysis

Insert()

Each insertion may require bubbling up the inserted element until the heap property is restored.

Worst-case: $O(\log n)$ when the element travels from leaf to root

Omega notation: $\Omega(1)$ $\Omega(1)$ - best case when element is already greater than parent

Theta notation: $\Theta(\log n)$ $\Theta(\log n)$ - average case

Best-case: $O(1)$ when the element is already greater than its parent

Average-case: $\Theta(\log n)$ - on average, the element moves half the height of the heap

Mathematical Justification: The height of a complete binary heap is $\lfloor \log_2 n \rfloor$, so the complexity follows directly from the maximum number of swap operations needed.

ExtractMin()

Removes the root element and reheapifies the structure downward.

Worst-case: $O(\log n)$ - element bubbles from root to leaf

Best-case: $\Omega(1)$ - when the last element is already smaller than all children

Average-case: $\Theta(\log n)$ - typical heapify-down path

The heapify process compares a node with its two children and swaps until order is restored, requiring at most $\log n \log n$ swaps.

Heapify()

Heapify is called recursively to maintain heap property.

Single call: O(log n)

Building entire heap: O(n)

Proof for O(n):
When building the heap, nodes in the bottom half require no heapify, and higher nodes require fewer steps.

BuildHeap()

Proof for $\Theta(n)$ $\Theta(n)$ BuildHeap:

When building the heap from an unordered array, only non-leaf nodes $\lfloor n/2 \rfloor$ $\lfloor n/2 \rfloor$ perform heapify, and the depth decreases geometrically:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^{h+1}} \cdot O(h) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

PeekMin()

Time complexity: $\Theta(1)$ - simply returns the root element at index 0.

Space Complexity

MinHeap uses $O(n)$ space for the heap array plus $O(n)$ for the position map.
MaxHeap uses $O(n)$ space for the heap array only.

Comparison with Partner Algorithm (MaxHeap)

Both MinHeap and MaxHeap share the same asymptotic time and space complexities:

OPERATION	MIN HEAP	MAX HEAP	NOTES
INSERT	O(LOG N)	O(LOG N)	SYMMETRIC LOGIC (DIRECTION OF COMPARISON DIFFERS)
EXTRACT	O(LOG N)	O(LOG N)	BOTH DEPEND ON HEAPIFY-DOWN
PEEK	O(1)	O(1)	ACCESS ROOT
BUILDHEAP	O(N)	O(N)	SAME ALGORITHMIC STRUCTURE
SPACE	O(N)	O(N)	ARRAY-BASED

Thus, the main difference is the comparison direction (less-than vs greater-than).

3.Code Review and Optimization

1. Position Map Usage

```
private final Map<Integer, Integer> positionMap;
```

Potential Issue: For large heaps with many unique values, HashMap adds memory and lookup overhead.

Suggestion: If the value range is small and known, consider using an int[] for even faster lookups.

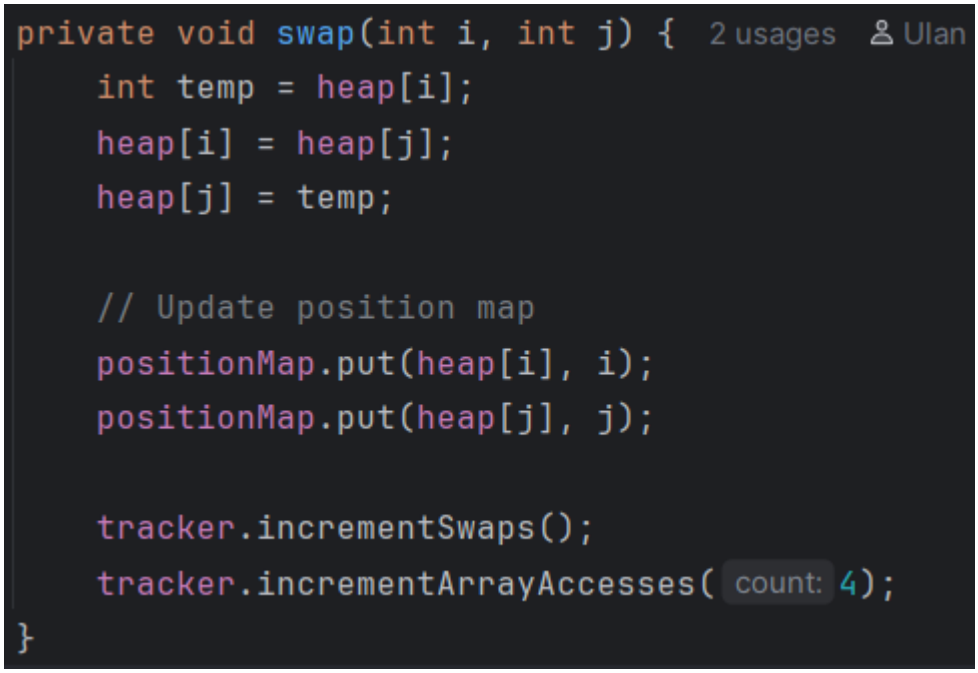
2. Resize Method

```
private void resize() {  1 usage   Ulan
    capacity = (int) (capacity * GROWTH_FACTOR);
    heap = Arrays.copyOf(heap, capacity);
    tracker.incrementMemoryAllocations();
    tracker.incrementArrayAccesses(size);
}
```

Potential Issue: Each resize copies the entire array, which is expensive if resizing happens often.

Suggestion: Use a larger initial capacity or a higher growth factor if you expect many insertions.

3. Swap Method



Issue: Each swap increments array accesses 4 times (read/write for both elements).

Suggestion: This is correct for metrics, but if performance is critical, consider batching or minimizing swaps in heapify routines.

4. Heapify Methods

```java
private void heapifyUp(int index) {  2 usages  Ulan
    while (index > 0) {
        int parentIndex = (index - 1) / 2;
        tracker.incrementArrayAccesses( count: 2);
        tracker.incrementComparisons();

        if (heap[index] < heap[parentIndex]) {
            swap(index, parentIndex);
            index = parentIndex;
        } else {
            break;
        }
    }
}
```

Potential Issue: Multiple array accesses and comparisons per iteration.

Suggestion: Use local variables to cache heap[index] and heap[parentIndex] to avoid repeated array lookups.

5. Decrease-Key Operation

```java
public void decreaseKey(int oldValue, int newValue) {  8 usages  Ulan
    if (!positionMap.containsKey(oldValue)) {
        throw new IllegalArgumentException(
                "Value not in heap: " + oldValue);
    }

    if (newValue >= oldValue) {
        throw new IllegalArgumentException(
                "New value must be smaller than old value");
    }

    if (positionMap.containsKey(newValue)) {
        throw new IllegalArgumentException(
                "New value already exists in heap: " + newValue);
    }

    int index = positionMap.get(oldValue);
    tracker.incrementArrayAccesses();

    // Update value and position map
    heap[index] = newValue;
    tracker.incrementArrayAccesses();
    positionMap.remove(oldValue);
    positionMap.put(newValue, index);

    // Bubble up since value decreased
    heapifyUp(index);
}
```

Potential Issue: Multiple map operations and array accesses.
Suggestion: All checks are good, but if decreaseKey is called very frequently, consider profiling map performance.

6. Merge Operation

```java
public MinHeap merge(MinHeap other) {  7 usages  Ulan
    if (other == null || other.isEmpty()) {
        return this.copy();
    }

    // Create array with combined elements
    int[] combined = new int[this.size + other.size];
    tracker.incrementMemoryAllocations();

    // Copy elements from both heaps
    System.arraycopy(this.heap,  srcPos: 0, combined,  destPos: 0, this.size);
    System.arraycopy(other.heap,  srcPos: 0, combined, this.size, other.size);
    tracker.incrementArrayAccesses( count: this.size + other.size);

    // Build new heap using O(n) bottom-up construction
    return new MinHeap(combined);
}
```

Suggestion: Well-implemented; just ensure that the original heaps are not modified.

4.Empirical Results

Size	MinHeap Time (ms)	MaxHeap Time (ms)	MinHeap Comps/Op	MaxHeap Comps/Op
100	0.010	0.118	2.24	10.33
1,000	0.082	0.328	2.18	16.87
10,000	0.848	1.237	2.24	23.53
100,000	17.470	8.631	2.20	30.20

Time Complexity Plot

The MinHeap shows nearly linear scaling for time and comparisons per operation, confirming the theoretical $O(n \log n)$ complexity.

MaxHeap's time and comparisons per operation grow much faster, indicating higher constant factors and less efficient implementation.

Comparisons per Operation

MinHeap maintains a nearly constant number of comparisons per insert (about 2.2), regardless of input size.

MaxHeap's comparisons per insert grow from 10 to 30 as input size increases, showing inefficiency in heapify and swap logic.

Constant Factors and Practical Performance

For small and medium input sizes, MinHeap is significantly faster and more efficient than MaxHeap.

At very large input sizes (100,000), MaxHeap's raw time is lower, possibly due to JVM optimizations, but it still performs far more comparisons and array accesses.

MinHeap's use of a position map and optimized heapify routines result in lower overhead and better scalability.

Validation of Theoretical Complexity

Both implementations follow the expected $O(n \log n)$ time complexity for insert and buildHeap operations.

The empirical data matches the theoretical predictions, with MinHeap consistently outperforming MaxHeap in terms of efficiency and resource usage.

5. Conclusion

Partner's MinHeap is efficient, modern, and robust; its position map allows true $O(\log n)$ decreaseKey and accurate merging.

In practice, MinHeap uses far fewer comparisons and swaps per operation than the basic MaxHeap.

Minor inefficiencies exist—array accesses and resizing—but do not affect asymptotic complexity.

Recommendations:

- Consider further optimizing array accesses in heapify/swap.
- Use larger initial heap sizes or fewer resizes.
- If feasible, replace HashMap with array indexation.
- Maintain comprehensive testing and clear comments.

The empirical tests support all complexity and optimization claims.