Peer Code Analysis — MaxHeap (Partner's Implementation)

Course: Design and Analysis of Algorithms (DAA)
Pair: 4 — Heap Data Structures
Student: Alikhan Serik
Partner: Bakytzhan Kassymgali
Analyzed Part: *Max-Heap Implementation (increase-key, extract-max)*

1. Algorithm Overview & Theoretical Background

This report conducts a comprehensive analysis of a **Max-Heap** data structure implemented by Bakytzhan Kassymgali. This implementation is one half of the "Heap Data Structures" pair for Assignment 2, which requires the implementation and analysis of both Min-Heap and Max-Heap structures.

A Max-Heap is a specialized tree-based data structure that satisfies the **max-heap property**: for any given node i, the value of i is greater than or equal to the values of its children. This property ensures that the largest element in the heap is always at the root. The heap is also a **complete binary tree**, meaning all levels of the tree are fully filled, except possibly for the last level, which is filled from left to right. This structure makes it highly efficient for implementation using an array.

The partner's MaxHeap.java is a classic **array-based binary heap** implementation. It dynamically resizes its underlying array to accommodate a growing number of elements, a crucial feature for a general-purpose data structure. The implementation correctly provides the required core functionalities for this assignment:

- **insert(int value)**: Adds a new element to the heap. To maintain the max-heap property, the new element is added to the end of the array and then moved up the tree (a process often called siftUp or percolateUp) until its correct position is found.
- **extractMax()**: Removes and returns the maximum element from the heap (the root). To fill the vacancy at the root, the last element in the heap is moved to the root's position. Then, it is moved down the tree (siftDown or heapifyDown) to restore the max-heap property.
- **increaseKey(int idx, int newValue)**: Increases the value of an element at a given index. Following the update, the siftUp operation is applied to the element to ensure the max-heap property is preserved, as the element's new value may be larger than its parent's.
- **buildHeapBottomUp()**: An efficient O(n) method to construct a heap from an unordered array. This approach is more performant than inserting n elements one by one (which would be O(n log n)) because it starts from the last non-leaf node and applies the siftDown operation iteratively up to the root. The partner's code correctly leverages this optimal construction method in one of its constructors.

Furthermore, the implementation is instrumented with a PerformanceTracker class. This allows for the empirical measurement of key operations such as comparisons, swaps, and array accesses.

2. Asymptotic Complexity Analysis

This section provides a rigorous analysis of the time and space complexity for the primary operations in the partner's MaxHeap.java implementation. The analysis adheres to the assignment's requirement to use Big-O (O), Big-Omega (Ω), and Big-Theta (Θ) notations to define the upper, lower, and tight bounds for the best, average, and worst-case scenarios.

The heap's performance is fundamentally tied to its height, which is **$h = \lfloor \log_2(n) \rfloor$** for a complete binary tree with *n* elements. All logarithmic complexities derive from operations traversing the tree from root to leaf or vice versa.

Time Complexity

`insert(int value)`

The insert operation adds an element to the next available leaf position and calls the siftUp helper method to restore the heap property. The complexity is therefore determined by siftUp.

- **Worst Case: Θ(log n)**
    - **Justification:** This occurs when the inserted element is larger than all elements along the path from the new leaf to the root. The siftUp method must perform swaps at each level, traversing the full height of the tree. The number of operations is directly proportional to the tree's height, *h*.
- **Best Case: Θ(1)**
    - **Justification:** The best case happens when the new element is smaller than or equal to its parent. The while loop condition in siftUp (heap[i] > heap[parent]) fails immediately, and no swaps are needed.
- **Average Case: Θ(1)**
    - **Justification:** While a formal proof is extensive, analysis shows that a newly inserted element is unlikely to travel far up the tree. On average, an element undergoes a constant number of swaps (approximately 1.6). Therefore, the amortized cost of an insertion is constant time.

`extractMax()`

This operation removes the root element, replaces it with the last leaf, and calls siftDown to restore the heap property. The complexity is dominated by the siftDown method.

- **Worst Case: Θ(log n)**
    - **Justification:** This occurs when the element moved to the root is smaller than all elements on a path to a leaf. It must be "sifted down" the entire height of the tree, requiring a comparison and possible swap at each level.
- **Best Case: Ω(log n)** and **O(log n) => Θ(log n)**
    - **Justification:** Unlike siftUp, siftDown must always traverse to a leaf to ensure the heap property is maintained, even if no swaps occur. At each level i, it must perform comparisons with both children (heap[left] and heap[right]) to find the larger one

before deciding whether to swap. Therefore, the number of comparisons is always proportional to the height of the tree. There is no "early exit" as in siftUp.

- **Average Case: Θ(log n)**
    - **Justification:** The element replacing the root is chosen from the last level of the tree and is likely to be small. Therefore, on average, it will travel most of the way down the tree, making the average number of operations proportional to log n.

`increaseKey(int idx, int newValue)`

This operation is functionally similar to insert, as it updates a value and calls siftUp to restore the heap property.

- **Worst Case: Θ(log n)**
    - **Justification:** Occurs if a key at a leaf is increased to become the largest value in the heap, requiring it to travel all the way to the root.
- **Best Case: Θ(1)**
    - **Justification:** The new value does not violate the heap property (i.e., it is still smaller than or equal to its parent). No swaps are needed.

`merge(MaxHeap other)`

The partner's implementation combines two heaps of size *n* and *m* by creating a new array of size n + m, copying all elements, and then calling buildHeapBottomUp.

- **All Cases: Θ(n + m)**
    - **Justification:** The complexity is the sum of copying the elements (which is Θ(n + m)) and building the new heap. The buildHeapBottomUp operation has a tight bound of O(n + m). Therefore, the total complexity is Θ(n + m).

Space Complexity

- **Overall Data Structure: Θ(n)**
    - **Justification:** The heap stores *n* elements in an underlying array. The space used is directly proportional to the number of elements. The partner's implementation correctly uses a dynamic array that grows as needed.
- **Auxiliary Space per Operation: Θ(1)**
    - **Justification:** All primary operations (insert, extractMax, increaseKey, siftUp, siftDown) are performed in-place. They use a constant amount of extra memory for temporary variables (like loop counters and the t variable in swap). Thus, the auxiliary space complexity is constant. The merge operation is an exception, as it temporarily allocates a new array of size n+m, but this is part of rebuilding the primary data structure itself.

3. Code Review & Optimization Suggestions

The partner's implementation of MaxHeap.java is robust, functionally correct, and follows good software engineering practices. The code is clean, well-structured, and effectively uses helper methods to encapsulate logic. However, a deeper analysis reveals several opportunities for significant algorithmic and design improvements that would enhance both performance and reusability.

Strengths

- **Code Quality & Readability:** The code adheres to standard Java conventions. Methods like siftUp and siftDown are well-named and logically separated, making the implementation easy to follow.
- **Correctness and Edge Cases:** The implementation correctly handles critical edge cases, such as operations on an empty heap (NoSuchElementException), invalid index access (IndexOutOfBoundsException), and invalid arguments (IllegalArgumentException). The inclusion of unit tests in MaxHeapTest.java further ensures correctness.
- **Optimal Heap Construction:** The constructor MaxHeap(int[] input) correctly utilizes the buildHeapBottomUp method, which constructs the heap in optimal **O(n)** time. This is a crucial performance feature for initializing a heap from a pre-existing collection.
- **Integrated Metrics:** The PerformanceTracker is seamlessly integrated, allowing for precise empirical analysis, which is a key requirement of this assignment.

Identification of Inefficiencies & Optimization Opportunities

1. Algorithmic Inefficiency in merge() Operation

- **Inefficiency:** The current merge method works by creating a new, larger array, copying all elements from both heaps, and then running buildHeapBottomUp on the entire new array. While functionally correct with a complexity of **Θ(n + m)**, this approach is suboptimal. It discards the fact that the two original structures are already valid heaps.
- **Optimization Suggestion (Time & Space Complexity Improvement):** A more efficient approach would be to iterate through the smaller of the two heaps and insert each of its elements into the larger heap. If heap *A* has *n* elements and heap *B* has *m* elements (where $m \leq n$), this strategy involves *m* insertions into a heap that grows from size *n* to *n+m*. The complexity would be **O(m log(n + m))**. For cases where one heap is significantly smaller than the other (e.g., m << n), this is a major performance gain over O(n + m).

```
// Suggested alternative merge logic

public void merge(MaxHeap other) {

    if (other == null || other.size == 0) return;

    if (other == this) throw new IllegalArgumentException("Cannot merge heap with itself");


    // More efficient: insert elements from the smaller heap into the larger one
```

```
    // This avoids a full O(n+m) rebuild
    for (int i = 0; i < other.size; i++) {
        this.insert(other.heap[i]);
    }
}
```

2. Lack of Generics and Type Safety

- **Inefficiency:** The current implementation is hardcoded for the int primitive type. This severely limits its reusability. To use it with other data types (e.g., Double, String, or custom objects), one would need to duplicate the entire class and modify the comparison logic.
- **Optimization Suggestion (Code Quality & Reusability):** Refactor the class to use Java Generics. This would make the heap type-safe and applicable to any object that implements the Comparable interface. The comparison logic in siftUp and siftDown would be replaced with calls to compareTo().

```
// Example of a generic MaxHeap
public class MaxHeap<T extends Comparable<T>> {
    private T[] heap;
    // ... constructor and other methods would use T instead of int

    private void siftUp(int i) {
        while (i > 0) {
            int parent = (i - 1) / 2;
            metrics.comparisons++;
            // Change comparison to use compareTo
            if (heap[i].compareTo(heap[parent]) > 0) {
                swap(i, parent);
                i = parent;
            } else break;
        }
    }
}
```

3. Suboptimal Capacity Growth Policy

- **Inefficiency (Memory Usage):** The ensureCapacity method doubles the array size (heap.length * 2) whenever it runs out of space. While simple, this can lead to significant

wasted memory. For example, if the heap has 1000 elements and one more is added, the array capacity will jump to 2000, leaving nearly half the array empty.

- **Optimization Suggestion (Space Complexity Improvement):** A more memory-conscious growth factor, such as **1.5x**, is often preferred in production systems (e.g., ArrayList in Java). This provides a better balance between the amortized cost of insertions and memory overhead.

```
// Suggested change in ensureCapacity
private void ensureCapacity() {
    if (size >= heap.length) {
        // Use a 1.5x growth factor to conserve memory
        int newCap = heap.length + (heap.length >> 1); // equivalent to * 1.5
        heap = Arrays.copyOf(heap, newCap);
        metrics.allocations++;
    }
}
```

4. Empirical Validation & Performance Analysis

To validate the theoretical complexity analysis from Section 2, a series of benchmarks were conducted on the partner's MaxHeap implementation. The tests were executed using the provided BenchmarkRunner on randomly generated integer arrays of sizes n = 100, 1 000, and 10 000. Each test configuration was run three times to ensure the stability of the results, and the measurements were then averaged.

Benchmark Results

The integrated PerformanceTracker was used to collect data on key metrics. The table below summarizes the averaged results from maxheap_results.csv. The full benchmark consists of inserting n elements and then extracting all of them, resulting in a total of $2n$ heap operations.

Input Size (n)	Avg. Comparisons	Avg. Swaps	Avg. Array Accesses	Avg. Time (ms)
100	1,059	525	4,522	0.07
1 000	17,234	8,607	71,896	0.38
10 000	239,511	119,685	987,728	3.11

Analysis of Asymptotic Growth

The empirical data strongly corroborates the theoretical **O(n log n)** time complexity for a sequence of *n* insertions and *n* extractions. We can observe this relationship directly in the data:

- When the input size *n* increases **10-fold** (from 1,000 to 10,000), the number of comparisons increases **13.9-fold** (from 17,234 to 239,511).
- Similarly, the average runtime increases **8.2-fold** (from 0.38 ms to 3.11 ms).

This super-linear growth is characteristic of the *n log n* complexity class. A purely linear O(n) algorithm would exhibit a ~10x increase in operations for a 10x increase in input size. The additional factor of ~1.4x in comparisons aligns with the log n component of the complexity: `(10000 * log(10000)) / (1000 * log(1000)) = 10 * (4/3) ≈ 13.3`. The measured growth is remarkably close to this theoretical prediction.

Performance Visualization

The performance plots, generated by plot_benchmarks.py, provide a clear visual confirmation of the asymptotic behavior.

- **Runtime vs. Input Size Plot:** The graph uses a logarithmic scale for the x-axis ("Input size (n)"). The distinct upward curve of the plot line for both MinHeap and MaxHeap visually demonstrates a growth rate faster than linear. This graphical representation is the expected signature of an O(n log n) algorithm on a log-linear plot.

Comparison with Min-Heap Implementation

As shown on the plots, the performance metrics for the partner's MaxHeap are nearly identical to those of the MinHeap implementation. This is expected, as the underlying algorithmic logic is symmetrical; the only difference is the direction of the comparison (> vs. <). This symmetric performance provides strong evidence that both implementations are not only correct but also equally efficient, confirming a solid understanding of the heap data structure by both partners.

In summary, the empirical results align precisely with the theoretical analysis. The partner's MaxHeap implementation demonstrates the expected O(n log n) performance profile and operates with high practical efficiency across all tested input sizes.

5. Conclusion

This analysis confirms that the MaxHeap implementation by Bakytzhan Kassymgali is **correct, efficient, and well-structured**. It successfully meets all core requirements for Assignment 2, demonstrating a strong command of heap data structures and algorithmic analysis.

The key findings of this report are as follows:

1. **Theoretical Consistency**: The implementation is asymptotically optimal. The theoretical analysis confirmed that all primary operations adhere to their expected time complexities—**Θ(log n)** for insertions and extractions, and **Θ(n)** for heap construction and merging.
2. **Empirical Validation**: The benchmark results provide definitive proof of the theoretical analysis. The measured performance data shows a clear **O(n log n)** growth pattern for a series of *n* operations, aligning perfectly with academic expectations for heap-based algorithms.
3. **Code Quality**: The source code is of high quality, characterized by its readability, modular design, and robust handling of edge cases.

While the current implementation is excellent, this report has identified several actionable recommendations for improvement. The most impactful suggestions include **implementing a more efficient merge strategy**, **refactoring the class to use generics** for broader applicability, and **tuning the memory allocation policy** to be more space-efficient.

Overall, the partner's work is exemplary and serves as a solid foundation. The successful completion of both the Min-Heap and Max-Heap implementations provides a comprehensive and effective solution to the "Heap Data Structures" problem set.

Prepared by:
Serik Alikhan
Design and Analysis of Algorithms — Assignment 2