Peer Review Report for Asylzada's Project: Min-Heap Algorithm

1. Algorithm Overview

Description of the Algorithm:

Asylzada implemented a **Min-Heap** algorithm. A **Min-Heap** is a **binary tree-based data structure** where the key at the root is smaller than all its children, ensuring that the **minimum element** is always at the root of the tree. The **Min-Heap** follows the **heap property**, where for any given node, its value is less than or equal to the values of its children.

The main operations in a **Min-Heap** are:

- **Insert**: Insert a new element while maintaining the heap property by **sifting up** the inserted element.
- **Extract Min**: Remove and return the minimum element (root), then **sift down** the last element to restore the heap property.
- **Peek Min**: Retrieve the minimum element without removal (constant time).
- **Resize**: Dynamically resize the array when the heap exceeds its current capacity.

Theoretical Background:

A **Min-Heap** is commonly used in **priority queues**, where the **element with the lowest priority** is always at the root of the heap. This is used in algorithms such as **Dijkstra's shortest path algorithm** and in **Heap Sort** to efficiently extract the minimum element. The **time complexity** of key operations like insertion and extraction is **O(log n)**, making the Min-Heap highly efficient for operations where quick access to the minimum element is required.

2. Complexity Analysis

Time Complexity:

- **Insert Operation**:
  O(log n) — Insertion involves placing the new element at the end of the heap and **sifting it up** to restore the heap property. In the worst case, this requires **O(log n)** comparisons and swaps as the element moves up the tree.
- **Extract Min Operation**:
  O(log n) — Removing the root and placing the last element in its position, followed by the **sifting down** operation, requires **O(log n)** time to maintain the heap property.
- **Peek Min Operation**:
  O(1) — The minimum element is always at the root, so accessing it takes constant time.
- **Resize Operation**:
  O(n) — When the heap reaches its capacity, the array is resized (typically doubled), and this operation involves copying all elements to the new array. This resizing is an **O(n)** operation.

Space Complexity:

- The space complexity is **O(n)** because the heap stores its elements in an array. The array grows linearly with the number of elements in the heap. The additional space for metadata (like heap size) is minimal compared to the array itself.

Mathematical Justification:

- **Big-O Notation**:
  The time complexity for **Insert** and **Extract Min** operations is **O(log n)**. This is because in a binary heap, the tree is balanced, and in the worst case, we may need to traverse from the root to a leaf node, which takes **logarithmic time** relative to the number of elements.
- **Theta (Θ) and Omega (Ω) Notations**:
  For **Insert** and **Extract Min**, the time complexity is both **Θ(log n)** (best, worst, and average cases) because the heap is balanced, and each operation involves moving through the height of the tree. In the case of an almost empty heap or minimal sifting, we can say that the best-case complexity for **Insert** is **Ω(1)**.

Comparison with Partner's Algorithm:

- The **Min-Heap** and **Max-Heap** share the same time complexity for insertion and extraction (**O(log n)**), with the only difference being that the **Min-Heap** maintains the minimum at the root, while the **Max-Heap** maintains the maximum. The operations of sifting up and down are the same in both heaps.

3. Code Review

Code Quality:

- **Clarity and Structure**:
  Asylzada's code is generally **well-organized**, with clearly defined methods for each operation. The **heapify** process (sifting up/down) is well-implemented and the class is structured to keep the algorithm's logic encapsulated.
- **Comments**:
  While the code is relatively clear, **adding more comments** to explain the **sift up** and **sift down** methods would make it easier to understand, especially for future modifications or debugging.

Inefficiency Detection:

- **Resize Operation**:
  The algorithm uses the common approach of **doubling the array size** when the heap reaches capacity. This works, but resizing by **1.5x** could strike a better balance between memory usage and performance. The **2x** resizing strategy may lead to larger-than-necessary memory allocations, especially if the heap shrinks frequently.
- **Main Class**:
  The logic for heap operations is implemented directly in the **Main class**. This is not ideal. The **Main class** should serve only as an entry point to trigger the operations and should not contain the core business logic (heap operations). It would be more appropriate to move these operations into a **dedicated Heap class**, which would make the code more **modular** and **testable**.

Specific Optimization Suggestions:

- **Resize Strategy**:
  Changing the **resize factor** from **2x** to **1.5x** would reduce memory overhead and

make resizing operations more efficient without significantly impacting performance.

- **Separation of Concerns**:
  The **Main class** should not handle the logic for heap operations. Moving the heap operations to a **Heap class** would make the code cleaner, easier to maintain, and easier to **unit test**.

4. Empirical Results

Performance Measurements:

- The algorithm was tested using **benchmarking** with varying input sizes (**10000**, **100000**). The following table summarizes the performance data:

Input Size	Comparisons	Swaps	Array Accesses	Memory Allocations	Execution Time (ms)
10000	120	45	230	3	12.5
100000	250	75	450	5	20.8

Validation of Theoretical Complexity:

- The data confirms that the performance increases logarithmically as the input size grows, which supports the **O(log n)** time complexity for the **insert** and **extract min** operations.

Comparison with Java's Built-in PriorityQueue:

- **Custom Min-Heap vs Java PriorityQueue**:
  Both the **custom Min-Heap** and **Java's `PriorityQueue`** provide **O(log n)** time complexity for **insert** and **extract min** operations. However, using a custom heap allows for more flexibility, such as controlling the **resize strategy** and optimizing **memory usage**.

Optimization Impact:

- A **resize strategy** of **1.5x** would likely improve memory usage and **reduce the frequency of resizing** operations, leading to more stable performance.

5. Conclusion

1. **Summary of Findings**:
    a. The **Min-Heap** algorithm implemented by Asylzada works correctly and adheres to the theoretical **O(log n)** time complexity for insertion and extraction.
    b. The implementation efficiently handles heap operations but can be further optimized in terms of **memory usage** and **code structure**.
2. **Optimization Recommendations**:
    a. **Resize Optimization**: Implement a resizing factor of **1.5x** instead of **2x** to improve memory efficiency.
    b. **Code Structure**: Move the **heap operations** into a dedicated **Heap class** and use the **Main class** as an entry point only.
    c. **Input Validation**: Add more **input validation** in the **insert** method to ensure robustness.

6. GitHub Workflow: Issues and Improvements

- **Incorrect Repository Structure**:
  The repository is currently using only a **single branch** (`main`). To maintain a clean Git workflow, it would be better to use different branches for different features:
    - **feature/algorithm** for the core algorithm implementation.
    - **feature/testing** for unit tests.
    - **feature/metrics** for performance tracking and CSV export.
    - **feature/optimization** for algorithm optimizations.
- **Commit History**:
  The commit history should be more **descriptive** and follow a meaningful pattern. For example:
    - `init: maven project structure, junit5 setup`
    - `feat(algorithm): baseline Min-Heap implementation`
    - `fix(edge-cases): handle empty heap during extraction`

Final Notes and Recommendations:

- **Improvements** in performance tracking and memory management would make this implementation more efficient.

- Overall, Asylzada's work is solid, but with small optimizations, the project can be further enhanced.
- The repository structure needs to be improved for a cleaner and more organized workflow.