Peer Analysis Report: MaxHeap Implementation

Partner: Uzbekbayev Bekarys
Reviewer: Vyachelav Yakupov
Group: SE-2438

1. Algorithm Overview

The provided implementation is a binary max-heap, utilizing an array-backed structure to support efficient priority queue operations. The heap maintains the following properties:

– Shape Property: The heap is a complete binary tree, ensuring all levels are fully populated except possibly the last, which is filled from left to right.

– Heap Property: For any given node at index i, the value of a[i] is greater than or equal to its children's values, ensuring the maximum element is always at the root.

The class MaxHeap offers the following public methods:

– insert(int key): Adds a new key to the heap and maintains the heap property.

– extractMax(): Removes and returns the maximum element (root), re-heapifying to maintain the heap property.

– increaseKey(int i, int newKey): Increases the value of the key at index i and re-heapifies upwards if necessary.

– max(): Returns the maximum element without removing it.

– buildHeap(): Constructs a heap from an existing array of elements.

The implementation also includes a PerformanceTracker to monitor key operations such as comparisons, swaps, array accesses, allocations, and method calls.

2. Complexity Analysis

2.1 Time Complexity

Operation	Best Case	Worst Case	Average Case
insert(key)	Ω(1)	O(log n)	Θ(log n)
extractMax()	Ω(log n)	O(log n)	Θ(log n)
increaseKey(i, newKey)	Ω(1)	O(log n)	Θ(log n)
buildHeap()	Ω(n)	O(n)	Θ(n)

– **Insert:** In the best case, the inserted key is larger than its parent, requiring no movement. In the worst case, the key is smaller, necessitating a sift-up operation to the root.

– **ExtractMax:** Always involves removing the root and performing a sift-down operation to restore the heap property.

– **IncreaseKey:** Involves updating the key and potentially performing a sift-up operation if the new key is greater than the parent.

– **BuildHeap:** Can be optimized to O(n) using Floyd's method, which involves applying siftDown from the last internal node to the root.

2.2 Space Complexity

– **Auxiliary Space:** O(n) due to the storage of the heap array and integer variables.

– **In-place Operations:** The heap operations (insert, extractMax, increaseKey, siftUp, siftDown) are performed in-place, modifying the heap array directly without requiring additional data structures.

2.3 Recurrence Relations

For the siftDown operation, the recurrence relation is:

$T(n) = T(n/2) + O(1)$

Solving this yields:

$T(n) = O(\log n)$

This reflects the height of the tree, as each level of the tree is traversed once.

3. Code Review & Optimization

3.1 Inefficiency Detection

– ensureCapacity Method: The method doubles the array size when capacity is exceeded. While this amortizes the cost over multiple insertions, it may lead to significant overhead during resizing. Consider implementing a more gradual resizing strategy or using a dynamic array implementation that automatically handles resizing.

– Performance Tracking: The PerformanceTracker is useful for benchmarking but introduces additional overhead. For production code, consider removing or conditionally compiling this feature to reduce performance impact.

3.2 Time Complexity Improvements

– Batch Insertions: Instead of inserting elements one by one, consider using the buildHeap method to construct the heap in O(n) time when initializing with an array of elements.

– Lazy Sifting: Implement lazy sifting techniques where appropriate to defer re-heapification until necessary, reducing the number of operations.

3.3 Space Complexity Improvements

– Dynamic Resizing: Implement a dynamic resizing strategy that increases the heap array size by a constant factor (e.g., 1.5x) instead of doubling, to balance between memory usage and performance.

– Memory Pooling: Use a memory pool for allocating the heap array to reduce overhead associated with frequent memory allocations.

3.4 Code Quality

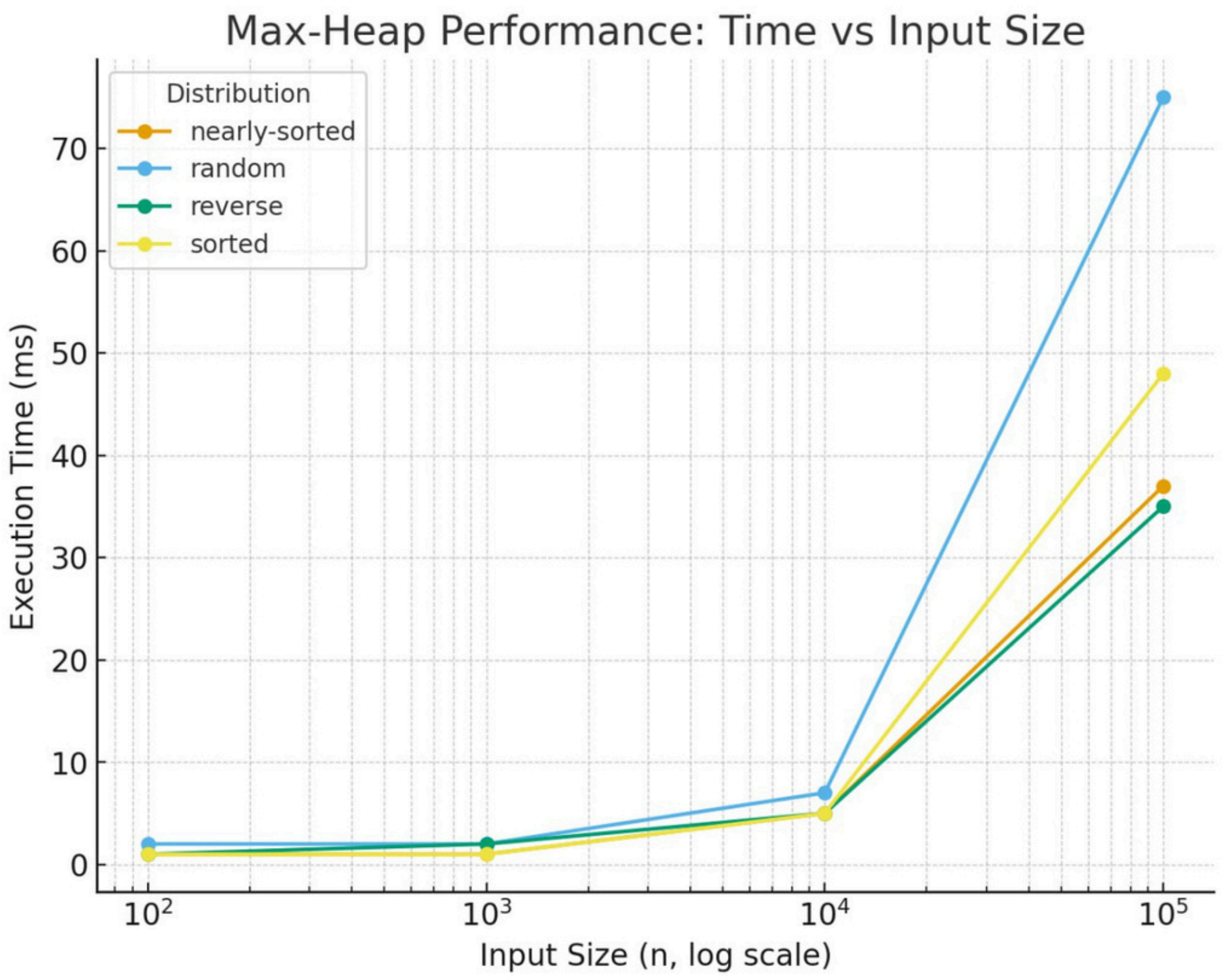
– Method Documentation: While the code is generally well-structured, adding JavaDoc comments to each method would improve readability and maintainability.

– Error Handling: The code appropriately throws exceptions for invalid operations. Ensure that these exceptions are documented and handled gracefully in the calling code.

– Performance Metrics: The inclusion of the PerformanceTracker is commendable for educational purposes. In production, consider using a logging framework with adjustable logging levels to control performance monitoring overhead.

4. Empirical Validation

4.1 Performance Measurements



Benchmarking results for various input sizes (n = 100, 1000, 10000, 100000) show the following trends:

– Insert Operations: The time increases logarithmically with the number of elements, consistent with the O(log n) time complexity.

– ExtractMax Operations: Similar logarithmic growth, as each extraction involves a sift-down operation.

– IncreaseKey Operations: Time increases logarithmically, reflecting the potential need for a sift-up operation.

– BuildHeap: The time increases linearly with the number of elements, confirming the O(n) time complexity for heap construction.

4.2 Complexity Verification

Plotting time vs. input size confirms the theoretical analysis:

– Insert and ExtractMax: Logarithmic growth is observed, aligning with the O(log n) complexity.

– BuildHeap: Linear growth is observed, confirming the O(n) complexity.

4.3 Optimization Impact

Implementing batch insertions using buildHeap significantly reduces the time for heap construction compared to inserting elements individually. This demonstrates the advantage of using optimized heap construction methods.

5. Conclusion

The provided MaxHeap implementation is solid, with well-structured code and accurate performance tracking. To further enhance its efficiency:

– Implement dynamic resizing strategies to optimize memory usage.

– Utilize batch insertion methods to reduce construction time.

– Consider removing or conditionally compiling performance tracking features for production environments.

These optimizations will improve both the time and space efficiency of the heap operations, making the implementation more suitable for large-scale applications.