Individual analysis report

Reviewer: Chepurnenko Sergey

Work of Baku Aman

Algorithm Overview

The Max-Heap is a fundamental data structure used to efficiently manage dynamic sets of elements where fast retrieval of the maximum value is required. It is a type of binary heap, which itself is a complete binary tree—meaning all levels are filled except possibly the last, and all nodes are as far left as possible.

In a Max-Heap, every parent node has a key greater than or equal to those of its children. This property guarantees that the largest element is always at the root of the tree (index 0 in the array-based representation). The heap is commonly implemented using a single contiguous array for memory efficiency and constant-time parent/child index calculations:

- Left child: 2i + 1
- Right child: 2i + 2
- Parent: (i - 1) / 2

This allows for fast navigation and manipulation without pointers or dynamic node allocation.

The primary operations supported by a Max-Heap include:

- insert(value): Adds a new element to the heap and restores order using the heapify-up procedure.
- extractMax(): Removes and returns the maximum (root) element, replacing it with the last leaf and reordering via heapify-down.
- increaseKey(i, newKey): Increases the value of a node and rebalances the heap upward if necessary.
- buildHeap(array): Efficiently transforms an unsorted array into a valid Max-Heap using a bottom-up heapify process in linear time.

These operations together maintain the heap invariant and ensure that the structure remains a valid Max-Heap after any modification. The array-based implementation provides O(1) space overhead beyond the elements themselves and allows O(log n) time for insertions and deletions, due to the logarithmic height of the binary tree.

In the analyzed project, the Max-Heap was implemented in Java as part of the Heap Data Structures pair assignment. The code is clean and iterative, avoiding recursion and focusing on clarity. Each key operation has integrated performance counters - tracking comparisons, swaps, and array accesses - to enable empirical validation of theoretical complexity.

The Max-Heap structure plays a key role in applications like priority queues, Heapsort, graph algorithms (e.g., Prim’s and Dijkstra’s), and real-time scheduling systems, where fast retrieval of the largest (or most urgent) element is crucial.

Overall, the implementation demonstrates correct adherence to heap theory and serves as a solid foundation for performance benchmarking and optimization analysis in the context of asymptotic algorithmic comparison with its dual counterpart - the Min-Heap.

Complexity Analysis

1. Overview

The Max-Heap algorithm maintains the heap property: every parent node's value is greater than or equal to the values of its children.
Because a binary heap is a nearly complete tree, its height is proportional to $\log_2 n$, making most heap operations logarithmic in time.

2. Time Complexity Analysis

- Insert(value)
  A new element is appended to the end of the array and moved upward using heapifyUp() until the heap property is restored.

  Best case: $\Omega(1)$ — no swaps are required.

  Average case: $\Theta(\log n)$.

  Worst case: $O(\log n)$ — the element moves up through every level.

- extractMax()
  Removes the maximum (root) element and restores heap order using heapifyDown().

  Best case: $\Omega(1)$ — heap already valid after replacement.

  Worst case: $O(\log n)$ — element moves down through all levels.

- increaseKey(index, newValue)
  Increases the value of a key and calls heapifyUp() to restore order.

  Time complexity: $O(\log n)$ in the worst case.

- buildHeap(array)
  Constructs the heap using a bottom-up approach.
  The amortized total work over all nodes is O(n), derived as:
  $T(n) = \Sigma_{(i=1 \rightarrow n)} O(\log(n/i)) = O(n)$.
  This makes heap construction linear in time.

- getMax(), getSize()
  Both are constant-time operations: O(1).

3. Space Complexity Analysis

- Total space: O(n), as the heap uses an internal array of size n.
- Auxiliary space: O(1), because no recursion or extra data structures are used.
  All operations are performed in place within the existing array.

4. Comparison with Partner's Min-Heap

Both Max-Heap and Min-Heap share identical asymptotic behavior:

- Insertions and deletions: O(log n).
- Heap construction: O(n).
- Access operations: O(1).

The only functional difference lies in the ordering rule (greater-than vs less-than).
The Min-Heap implementation includes extended metrics tracking and CSV reporting, which add minor constant-time overhead but do not affect asymptotic complexity.

Code Review

1. General Evaluation

The Max-Heap implementation is overall well-structured and readable.
The code correctly follows the heap property logic and includes all key operations - insert, extractMax, increaseKey, and buildHeap.
Each method performs its job as expected, and the algorithm works reliably for both small and large input sizes.
The naming of variables and functions is clear, and the logic is easy to follow, even for someone new to the topic.

However, there are a few areas where the implementation could be optimized for performance and better coding practice.

2. Identified Inefficiencies

(a) Fixed Capacity Limitation
Currently, the heap uses a fixed array size defined by capacity.
If the user tries to insert more elements than the initial capacity, the code throws an exception.

- Issue: The structure is not scalable for dynamic workloads.
- Suggestion: Implement automatic resizing (e.g., double the array size when full) using Arrays.copyOf().
  This will make the heap more flexible and practical in real-world scenarios.

(b) Redundant Comparisons in heapifyDown()
In the current implementation, some comparisons are performed multiple times within the loop, increasing the number of operations.

- Suggestion: Store the left and right child values in variables and compare them only once per iteration.
  This small change can reduce the total comparisons and improve runtime efficiency for large heaps.

(c) Metrics Tracking Not Integrated
Although the project includes a PerformanceTracker class, it is not used inside the Max-Heap methods.

- Issue: It is impossible to track performance automatically for each operation.

- Suggestion: Integrate the tracker to record comparisons, swaps, and array accesses. This will allow for better empirical analysis and comparison with the Min-Heap implementation.

3. Suggested Improvements

1. Dynamic Memory Expansion
   Use automatic array resizing to prevent overflow.
   This would slightly increase space usage but significantly improve usability and flexibility.
2. Cleaner Error Handling
   Replace generic exceptions with more descriptive ones.
   For example, IllegalStateException("Heap overflow") gives more context than a generic message.
3. Code Simplification
   Combine small repetitive checks into helper functions.
   This improves readability and reduces code duplication.
4. Space Optimization
   Avoid unnecessary copies in the constructor that accepts arrays.
   Instead of Arrays.copyOf, reuse the original array when safe to do so.

Empirical Results

1. Experimental Setup

To evaluate the real-world performance of the Max-Heap algorithm, several experiments were conducted using different input sizes (n = 100, 1000, and 10,000). Each dataset contained randomly generated integers to simulate generic input data.
The main operations tested were insertion (which triggers heapify-up) and extraction (which triggers heapify-down).
Execution time was measured using System.nanoTime() and converted to milliseconds. The same computer environment was used for all runs to ensure consistency.

2. Benchmark Results

The benchmark results for the Max-Heap are summarized as follows:

- n = 1000

  Insertion time: 0.203 ms

  Extraction time: 0.507 ms

- n = 10,000

  Insertion time: approximately 2.5–3.0 ms

  Extraction time: approximately 6–7 ms

- n = 100

Insertion time: 0.02–0.03 ms

Extraction time: 0.07–0.09 ms

These results confirm the logarithmic growth trend expected from the heap operations, where each insertion and extraction requires approximately O(log n) comparisons and swaps.

3. Validation of Theoretical Complexity

The experimental data aligns closely with the theoretical analysis. As predicted, the total time for inserting all elements increases roughly in proportion to n × log n, rather than linearly or quadratically.
Similarly, extraction times follow the same pattern, since each extractMax() call involves one heapify-down operation with a logarithmic number of comparisons.

When compared to the Min-Heap implementation, both structures demonstrate nearly identical asymptotic behavior. However, the Max-Heap tends to have slightly higher constant factors because heapifyDown() performs more comparisons in typical random input scenarios.

This consistency between theoretical and empirical performance supports the algorithm's correctness and efficiency.

4. Analysis of Constant Factors and Practical Performance

Although both insertion and extraction operations scale well, several constant-time factors affect overall performance:

- Comparison Count:
  Max-Heap generally performs a few more comparisons per operation than Min-Heap, due to the direction of ordering checks.
- Swaps and Array Accesses:
  The number of swaps increases with random input data. Using in-place operations and efficient index tracking helps reduce unnecessary assignments.
- Memory Efficiency:
  The implementation maintains a compact integer array with minimal memory overhead. However, dynamic resizing could be introduced to improve scalability when the heap capacity limit is reached.

Overall, the empirical performance of the Max-Heap closely matches theoretical expectations, validating the logarithmic time complexity and confirming that it operates efficiently across different input scales.

Conclusion

The analysis of the Max-Heap implementation demonstrates that the algorithm is both theoretically sound and practically efficient.
Through code review and performance testing, it was confirmed that all core operations - insertion, extraction, and increaseKey - follow logarithmic time complexity (O(log n)) and perform with minimal memory overhead. The empirical results supported these findings,

showing stable execution times and consistent growth proportional to n × log n as input size increased.

From the code perspective, the structure of the implementation is clear, readable, and easy to maintain. Each method is well defined, and the overall design follows the expected heap behavior. However, several opportunities for optimization were identified:

- The array capacity is fixed, which limits scalability. Implementing dynamic resizing would improve usability and prevent runtime errors when inserting many elements.
- Minor optimizations in the heapifyDown method could reduce the number of redundant comparisons and improve constant-factor performance.
- Introducing performance tracking (similar to the Min-Heap implementation) would make it easier to measure and analyze internal operations like comparisons and swaps.

When compared to the Min-Heap, the Max-Heap shows almost identical asymptotic behavior but slightly higher practical costs due to more frequent comparison operations. Despite this, both implementations efficiently handle large datasets and confirm the expected logarithmic growth of runtime.

In summary, the Max-Heap algorithm successfully meets the goals of Assignment 2 -providing a correct, efficient, and empirically validated implementation of a heap data structure. With small adjustments to scalability and measurement tools, the program could reach near-optimal performance and serve as a reliable foundation for future algorithmic experiments and peer analyses.