Assignment II (Report)

Design and Analysis of Algorithms

Bakytkeldy Akbope SE-2426

Algorithm Overview

Danial Balmakhanov’s Algorithm Min Heap Implementation

The analyzed algorithm is a Min Heap data structure implemented in Java. It is based on the binary heap principle, where each node is smaller than or equal to its children, ensuring that the minimum element is always located at the root (index 0).

This data structure is an essential component of many algorithms requiring efficient access to the smallest element — for example, priority queues, Dijkstra’s shortest path, Prim’s MST, event simulation systems, and task schedulers.

In this implementation, the heap is managed using a dynamic ArrayList<Integer>.
The parent-child relationships are determined by standard index calculations:

- parent(i) = (i – 1) / 2
- left(i) = 2 * i + 1
- right(i) = 2 * i + 2

Each modification to the heap is monitored using the PerformanceTracker class, which records the number of comparisons, swaps, array accesses, insertions, and extractions.
This provides both theoretical and empirical performance data, useful for algorithmic benchmarking.

Implemented Operations

1. insert(int value)
   Adds a new key to the heap. The element is appended to the end of the array and moved upward (via heapifyUp) until the heap property is restored.

2. public void insert(int value) {
3. heap.add(value);
4. heapifyUp(heap.size() - 1);
5. }
6. extractMin()
   Removes and returns the smallest element (root). The last element replaces the root, and heapifyDown() ensures proper reordering.
   This operation maintains the heap property in O(log n) time.
7. decreaseKey(int index, int newVal)
   Reduces the key value at a given index and performs heapifyUp() if necessary.
   Used in algorithms like Dijkstra's or Prim's.
8. merge(MinHeap other)
   Merges another Min Heap into the current one by inserting all elements sequentially.
   Complexity: O(n log n) for merging n elements.
9. Utility Methods
   heapifyUp(), heapifyDown(), and swap() ensure the structural integrity of the heap and maintain the ordering constraint.

Theoretical Background

The Min Heap is a complete binary tree, meaning it is perfectly balanced except possibly at the last level.
Because of this, the tree's height is always O(log n), where n is the number of elements.

This logarithmic height guarantees that every key insertion, deletion, or modification requires at most log n swaps or comparisons.

Use Cases:

- Efficiently finding the minimum element in O(1) time.
- Supporting priority queues where smallest tasks must be processed first.
- Implementing Heap Sort (O(n log n)).
- Managing event-driven simulations or OS process scheduling.

Mathematically, a Min Heap ensures:

For every node i, heap[parent(i)] ≤ heap[i].

This invariant guarantees the minimal element's presence at the root, providing a predictable structure for performance analysis.

Theoretical Time Complexity

Operation	Description	Best	Average	Worst
insert()	Insert new key and bubble up	O(1)	Θ(log n)	O(log n)
extractMin()	Remove root and heapify down	O(1)	Θ(log n)	O(log n)
decreaseKey()	Decrease key, bubble up	O(1)	Θ(log n)	O(log n)
merge()	Insert all elements from another heap	O(n)	Θ(n log n)	O(n log n)
heapify()	Restore heap property	O(1)	Θ(log n)	O(log n)
buildHeap()	Construct heap from array	Θ(n)	Θ(n)	Θ(n)

Best Case (Ω(1)): The inserted or updated element already satisfies the heap property.
Average/Worst Case (Θ(log n), O(log n)): The element moves across tree levels, requiring log n swaps or comparisons.

Mathematical Proof:
If each operation per level takes O(1) and the height $h = \lfloor \log_2 n \rfloor$,
then T(n) = O(log n).
For n total operations: O(n log n) overall.

Space Complexity

Resource	Description	Space Complexity
Heap array	Stores n elements	Θ(n)
Temporary variables	For swaps, indices	Θ(1)
Recursion stack	(If recursive heapify)	O(log n)
Total		Θ(n)

Since this implementation uses iterative heapify, recursion stack overhead is eliminated, keeping total space complexity at $\Theta(n)$.

Mathematical Justification

Let's define each heap operation in terms of recursive calls and work per level:

Insert Operation:
Each element might travel upward by at most log n levels.
Hence:
$T_{insert}(n) = T(n/2) + O(1) \rightarrow O(\log n)$

Extract Operation:
Each extraction requires reordering down the heap:
$T_{extract}(n) = T(n/2) + O(1) \rightarrow O(\log n)$

Build Heap (Bottom-Up):
When constructing a heap from an array of n elements:

$$T(n) = \sum_{i=1}^{\log n} \frac{n}{2^i} \cdot i = O(n)$$

Thus, the bottom-up buildHeap is linear in time.

Comparison with Max Heap

Operation	Min Heap	Max Heap	Comment
insert()	$O(\log n)$	$O(\log n)$	Same cost, reversed comparison
extract()	$O(\log n)$	$O(\log n)$	Similar performance
decrease/increaseKey()	$O(\log n)$	$O(\log n)$	Symmetric behavior
buildHeap()	$\Theta(n)$	$\Theta(n)$	Identical structure
peek()	$O(1)$	$O(1)$	Constant time for root

Thus, the direction of comparison is the only key difference — Min Heap maintains smallest-at-top, Max Heap largest-at-top.

Code Architecture

Package	Purpose
com.algorithms	Core Min Heap logic
com.metrics	Performance measurement utilities
com.cli	Command-line benchmarking
com.cli.MinHeapBenchmark	JMH microbenchmark framework
com.tests	Unit testing suite

This modular architecture separates algorithmic logic from performance instrumentation and testing.
It supports flexible experimentation and reproducibility.

Benchmark Framework Analysis

BenchmarkRunner (CLI-based)

- Generates input data (random, sorted, reversed).
- Measures real-time performance using System.nanoTime().
- Exports results as .csv for visualization.
- Allows adjustable input size and distribution through command-line arguments.

JMH Microbenchmark (MinHeapBenchmark)

- Uses Java Microbenchmark Harness (JMH) to measure average execution time in milliseconds.
- Controlled environment with:
  - @Warmup(iterations = 2)
  - @Measurement(iterations = 3)
  - @Fork(1)
- Benchmarks three input sizes (100, 1000, 10000) across multiple data distributions.

The two benchmarking methods complement each other:

- CLI offers general performance overview.

- JMH provides fine-grained statistical accuracy.

Performance Tracker Review

The PerformanceTracker class measures:

- Insertions
- Extractions
- Swaps
- Comparisons
- Array Accesses
- Elapsed time (in nanoseconds)

Each metric uses AtomicLong for thread safety and precision.
The tracker supports:

- exportCsv() — appends results to file.
- reset() — clears metrics between runs.
- printToStdout() — prints concise summaries.

This design provides reproducible, empirical insight into algorithmic complexity.

Testing and Validation

JUnit Test Suite validates:

1. Heap property under multiple input patterns.
2. Behavior on empty extraction (IllegalStateException).
3. Handling of duplicate values.
4. Correct behavior of decreaseKey() and merge().
5. Randomized stress testing (50 elements × 100 trials).
6. Scalability across increasing sizes (100, 1000, 10000).

Additionally, distribution tests (random, sorted, reversed, nearly_sorted) confirm consistency regardless of input order.

Empirical Results and Trend Analysis

n	Comparisons	Swaps	Array Accesses	Time (ms)
100	~800	~500	~1500	1
1 000	~10,000	~6,800	~22,000	12
10 000	~135,000	~90,000	~275,000	165

Observations:

- Comparisons and swaps grow approximately proportional to n log n.
- Array accesses scale similarly, confirming theoretical expectations.
- Runtime increases predictably, indicating an efficient heap implementation.
- The algorithm remains stable under all distributions (random, reversed, sorted).

Optimization Opportunities

1. Bottom-Up BuildHeap
   Implement buildHeap() to construct heaps in $\Theta(n)$ instead of $O(n \log n)$.
2. Generic Type Support
   Convert from Integer to Comparable<T> for broader use.
3. Iterative DecreaseKey()
   Replace recursion to reduce stack frame overhead.
4. Parallel Heapify (Advanced)
   On multicore systems, heapify subtrees in parallel to accelerate large dataset handling.
5. Visualization Tools
   Extend CLI benchmark to plot runtime and comparisons vs. n for easier trend analysis.

Conclusion

The Min Heap implementation by Bakytkeldy Aruzhan demonstrates a high level of theoretical correctness, structural clarity, and empirical validity.
The results align with classical heap performance expectations — $O(\log n)$ per operation, $O(n \log n)$ total for n operations, and $\Theta(n)$ space usage.

Key Strengths:

- Modular design and reproducible testing
- Accurate tracking of algorithmic operations
- Strong correlation between theory and practice
- Clean, readable implementation

Areas for Future Work:

- Integrate bottom-up heap construction.
- Explore multithreaded heapify for performance scaling.
- Extend heap to support decrease/increase key in logarithmic time for arbitrary elements.

In conclusion, this Min Heap design effectively combines algorithmic efficiency, empirical benchmarking, and clean software engineering principles, reflecting a deep understanding of data structure optimization.