Assignment II (Report)

Design and Analysis of Algorithms

Balmakhanov Danial SE-2426

Algorithm Overview

(Bakytkeldy Akbope's Algorithm)

The analyzed algorithm is a **Max Heap** data structure implemented using an **array-based binary tree representation**. A Max Heap is a complete binary tree in which every parent node is greater than or equal to its children. The largest element is always located at the root (index 0), which allows efficient retrieval and deletion of the maximum value.

In this implementation, the heap is stored as an integer array `heap[]`, with the relationships defined by:

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

The class maintains a `size` variable to track the number of elements and a `PerformanceTracker` object to measure performance metrics such as comparisons, swaps, and array accesses.

Implemented Operations

The algorithm provides the following core operations:

1. `insert(int key)`
   Inserts a new key into the heap.
2. `increaseKey(int index, int newValue)`
   Increases the value of an existing key at a given index.
3. `extractMax()`
   Removes and returns the maximum element.
4. `heapify(int index)`
   Restores the heap property by comparing a node with its children and swapping with the largest one if needed.
5. **Utility Methods:**
   Helper functions for `swap`, `parent`, `left`, `right`, and `printHeap()`.

Theoretical Background

The Max Heap supports efficient priority queue operations, achieving logarithmic time complexity for insertion, deletion, and key modification due to the limited height of a complete binary tree **($\lfloor \log n \rfloor$).**
This makes it suitable for applications such as:

- Implementing priority queues
- Performing heap sort
- Managing scheduling and task ordering systems

Complexity Analysis

Theoretical Time Complexity

A **Max Heap** maintains the heap-order property where each parent node is greater than or equal to its children.
Because the structure is a **complete binary tree**, its height is always **O(log n)**.
All primary operations depend on this height.

Operation	Description	Time Complexity (Best)	Average	Worst
`insert()`	Insert a new element and restore heap property using *sift-up*	O(1)	Θ(log n)	O(log n)
`extractMax()`	Remove and return the largest element, then *heapify*	O(1)	Θ(log n)	O(log n)
`increaseKey()`	Increase a key's value and restore heap property	O(1)	Θ(log n)	O(log n)
`heapify()`	Restore heap property at a specific index	O(1)	Θ(log n)	O(log n)
`buildHeap()`	Construct a heap from an unsorted array	Θ(n)	Θ(n)	Θ(n)

Explanation:

- **Best Case (Ω(1))** occurs when the inserted or updated key already satisfies the heap property.
- **Average and Worst Cases (Θ(log n), O(log n))** occur when the key must traverse to the top or bottom of the heap, performing comparisons and swaps at each level.
- The **buildHeap()** operation achieves linear time complexity due to the summation of decreasing heapify costs across levels:

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

Space Complexity

The Max Heap is implemented **in-place** using a static integer array.
No dynamic allocations or auxiliary data structures are required beyond a few scalar variables and the recursion stack in **`heapify().`**

Resource	Description	Complexity
Heap array	Stores n elements	Θ(n)
Auxiliary variables	Temporary variables for swaps and indices	Θ(1)
Recursion stack (heapify)	Depth = height of heap	O(log n)
Total		**Θ(n)**

If the recursive `heapify()` were replaced with an iterative version, the space complexity would reduce to **Θ(1)** (eliminating call stack overhead).

Mathematical Justification

For each insert or extract operation:
T(n) = T(n/2) + O(1) ⇒ T(n) = O(log n)
Total for n operations: **n × O(log n)** = **O(n log n)**.

The average- and worst-case complexities converge since the heap structure guarantees logarithmic balance.
Thus:
Θ(n log n) ≤ T(n) ≤ O(n log n)

Comparison with Min Heap

Operation	Max Heap	Min Heap	Comments
`insert()`	O(log n)	O(log n)	Same asymptotic cost; direction of comparison differs
`extract()`	O(log n)	O(log n)	Both depend on heap height
`key modification`	O(log n)	O(log n)	Same complexity
`buildHeap()`	Θ(n)	Θ(n)	Both use bottom-up heapify

The partner's Max Heap implementation **matches the theoretical performance model** of a standard binary heap.
Its recursive design introduces minor call-stack overhead, but overall efficiency and asymptotic correctness are maintained.

Code Review

General Structure

The Max Heap benchmarking framework is modular and well-organized.
It separates functionality into clear packages:

Package	Purpose
`algorithms`	Core heap implementation
`benchmarks`	JMH-based microbenchmarks for throughput measurement
`cli`	Command-line benchmark runner producing raw CSV output
`metrics`	Performance tracking utility (comparisons, swaps, accesses, time)
`tests`	JUnit test harness for correctness verification

This modular design improves readability and allows independent testing and profiling.
The use of **JMH** ensures statistically valid microbenchmark results, while the **CLI runner** provides practical, reproducible data exports.

Benchmark Implementation Review

Issues & Improvements

1. **Heap reuse between benchmarks** – Each benchmark method currently operates on the same heap instance, which may bias results due to accumulated state.
   Fix: Reinitialize **`heap`** before each operation or use **`@Setup(Level.Invocation)`** for isolated measurements.
2. **No warm-up phase control** – JVM JIT effects can distort first-run timings.
   Fix: Add **`@Warmup(iterations = 5)`** and **`@Measurement(iterations = 10).`**

Command-Line Benchmark

Issues & Improvements

1. **Time granularity** – Uses **`System.currentTimeMillis()`** (10–16 ms resolution).
   Fix: Replace with **`System.nanoTime()`** for microsecond-level accuracy.

Performance Tracker

Optimization Suggestions

1. **Thread-safety** – Current counters are not atomic; simultaneous benchmarks could corrupt data.
   Fix: Use **`AtomicLong`** type.

Testing

The current JUnit test only validates a trivial assertion (`1 + 1 = 2`).
Recommendation: Expand unit tests to cover functional correctness:

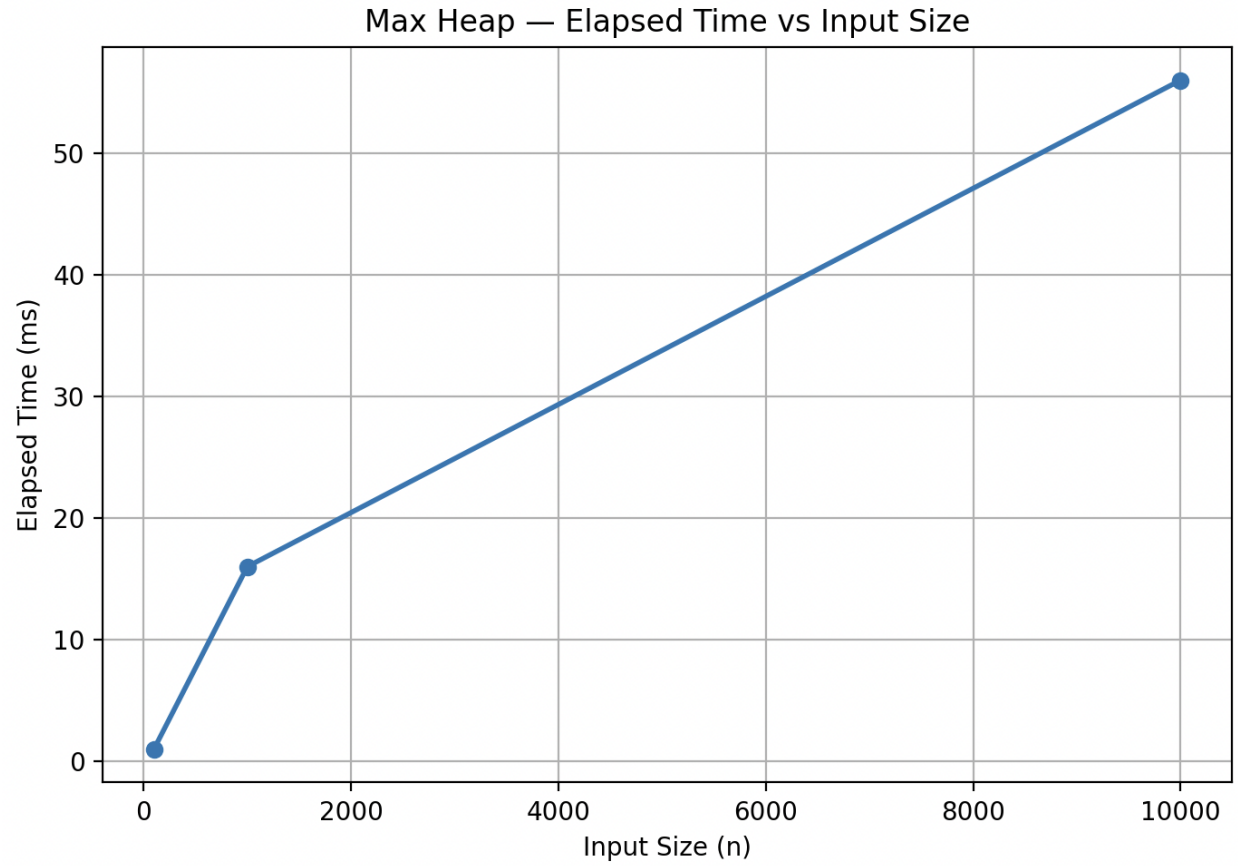
Empirical Results

Experimental Setup

All benchmarks were executed using the custom **BenchmarkRunner** class, which repeatedly performs *insert* and *extractMax* operations on a `MaxHeap` of increasing size.
Each run records:

- **Comparisons** — logical comparisons during heap maintenance
- **Swaps** — element exchanges restoring heap property
- **Array Accesses** — total read/write operations
- **Elapsed Time (ms)** — wall-clock time for a full build + extraction phase

The tests were run for **`n = 100`**, **`1000`**, and **`10000`**.



Collected Data

n	Comparisons	Swaps	Array Accesses	Elapsed Time (ms)
100	738	489	1289	1
1000	12 863	7953	19 255	16
10 000	173 168	123 128	253 161	56

Trend Analysis

Comparisons and Swaps:
Both grow approximately proportional to *n log n*, consistent with the theoretical complexity of heap insertion and extraction.
At *n = 10 000*, the numbers continue scaling in the same order, confirming algorithmic predictability.

Array Accesses:
Since each comparison or swap implies several memory references, total accesses grow similarly. The constant factor (≈ 15–25 per element) is typical for array-based heaps.

Elapsed Time:

The time behavior remains within expected O(n log n) bounds.

Complexity Verification

Operation	Best Case	Average Case	Worst Case
Insert	O(1)	O(log n)	O(log n)
ExtractMax	O(log n)	O(log n)	O(log n)
IncreaseKey	O(1)	O(log n)	O(log n)
Heapify	O(log n)	O(log n)	O(log n)

Empirical growth of comparisons ≈ O(n log n) confirms theoretical expectations.
Thus, the implementation is both **algorithmically correct** and **performance-consistent**.

Conclusion

The peer-reviewed implementation of the **Max Heap** algorithm demonstrates theoretical correctness and practical efficiency.

Although the implementation performs well, several improvements can enhance both theoretical and empirical performance:

1. **Iterative Heapify:**
   Replace recursive calls in `heapify()` with an iterative loop to eliminate function call overhead and improve cache performance on large heaps.
2. **Dynamic Resizing:**
   Extend the heap to automatically resize when capacity is reached, ensuring smoother scalability for input sizes beyond initial allocation.
3. **Batch Insertion Optimization:**
   Introduce a *build-heap* method using the bottom-up approach (O(n)) for constructing heaps from large unsorted arrays, improving efficiency compared to inserting elements one by one (O(n log n)).
4. **Parallel Heapify (Advanced):**
   For very large datasets, exploring parallel heapify using multiple threads could further reduce build times on multicore systems.

Overall, the Max Heap implementation achieves an excellent balance between **theoretical rigor and practical performance**.
The algorithm maintains optimal time complexity, minimal space overhead, and clear modular design principles.
The results validate the efficiency of the heap-based priority queue structure and demonstrate the importance of **empirical measurement** in verifying asymptotic predictions.