Lab 2 – MaxHeap Performance Analysis

Algorithm: **MaxHeap (Partner's Implementation)**
Reviewer: **Sarsenbek Beknazar**

1. Objective

The purpose of this laboratory work is to implement and analyze the performance of a MaxHeap data structure. This includes measuring the number of operations such as comparisons, swaps, memory allocations, and array accesses while performing basic heap operations like insert, extractMax, and increaseKey. In this laboratory work, you need to implement MaxHeap and investigate its behaviour as the size of the input data increases.

The aim of this study is to examine sorting algorithms and analyse their effectiveness in practice. The main focus is on methods such as selection sort, bubble sort, and quick sort. The objective is to compare these algorithms in terms of execution speed and resources used, identify their strengths and weaknesses, and determine the most optimal way to solve the data sorting problem.

In addition, the study aims to develop practical programming skills and consolidate knowledge on the topic of algorithms. The results obtained can be applied to solve applied problems in the field of computer science, where the processing of large amounts of data needs to be optimised.

2. Algorithm Overview

A MaxHeap is a complete binary tree that can be efficiently represented using an array or dynamic array (e.g., `ArrayList` in Java). The parent-child relationships are determined by index calculations:

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

The height of a heap with *n* elements is approximately $h = \lfloor \log_2 n \rfloor$. Since most operations (`insert`, `extractMax`, `increaseKey`) involve traversing at most one path from root to leaf (or vice versa), their time complexity is O(log n).

The following table summarizes the theoretical complexity:

A MaxHeap is a complete binary tree where each parent node is greater than or equal to its children. It supports efficient insertion and extraction of the maximum element in O(log n) time. This property is maintained by using the heapify operations (sift-up and heapify-down).

The analyzed algorithm is a **Max-Heap** data structure implemented using an `ArrayList<Integer>` in Java.
A Max-Heap maintains the property that every parent node is greater than or equal to its child nodes. The largest element is always at the root (`index 0`).

This implementation provides the following main operations:

- **insert(int value):** adds a new value and reorders the heap by "sifting up" the inserted element.
- **extractMax():** removes and returns the maximum element, then restores the heap using "heapify down".
- **increaseKey(index, newValue):** increases the value at a given index and restores heap order upward.
- **peek():** returns the maximum element without removing it.
- **PerformanceTracker:** tracks metrics such as comparisons, swaps, memory allocations, and array accesses for empirical analysis.

The heap is **zero-indexed** internally and uses bitwise operations for efficiency in calculating parent and child indices.

Operation	Best Case	Average Case	Worst Case	Explanation
insert()	O(1)	O(log n)	O(log n)	At most one path up the tree (height = log n)
extractMax()	O(1)	O(log n)	O(log n)	Heapify-down may traverse full height
increaseKey()	O(1)	O(log n)	O(log n)	Increases key and may need to sift up
peek()	O(1)	O(1)	O(1)	Just returns root
buildHeap()	O(n)	O(n)	O(n)	Not implemented but standard result

Overall Time Complexity:

- Each operation runs in **O(log n)** (except peek and buildHeap).
- The heap is asymptotically efficient and suitable for priority queue applications.

3. Implementation Summary

The implementation uses an ArrayList<Integer> to represent the heap. Each operation updates a PerformanceTracker object to record performance metrics such as the number of comparisons, swaps, and memory accesses. The PerformanceTracker class provides a simple way to monitor algorithmic behavior during execution.

Strengths

- ✓ Clean and readable code with clear method separation.
- ✓ Proper use of `PerformanceTracker` for comparisons, swaps, and memory allocations.
- ✓ Good use of `ArrayList` for dynamic sizing instead of manual array resizing.
- ✓ Error handling via `NoSuchElementException` and `IllegalArgumentException`.

Potential Improvements

1. **Memory Tracking Accuracy:**
   The `tracker.incAllocations(1)` inside `insert()` approximates per-element

allocations, but `ArrayList` internally resizes by chunks. Could adjust tracking to reflect amortized allocations instead of per-element.

2. **Generic Support:**
   The implementation currently supports only `Integer`. Using Java generics (`Comparable<T>`) would allow the heap to handle any comparable type.
3. **Heapify Method:**
   Adding a `buildHeap()` method would allow bulk heap creation from an array in O(n) time, improving efficiency for large datasets.
4. **Improved Metrics Reset:**
   Consider adding a `reset()` call for the tracker before benchmarks to isolate each test run.
5. **Thread Safety:**
   The implementation is not thread-safe. Adding synchronization (or documentation stating single-threaded use) would be useful.

4. Experimental Results

During the experiment, multiple input sizes (n) were used to measure the execution performance of MaxHeap operations. The following table summarizes the results obtained:

	n	distribution	rep	timeMs	comparisons	swaps	accesses	allocations
1	100	random	1	0	364	206	1567	101
2	100	random	2	0	349	186	1454	101
3	100	random	3	0	340	178	1415	101
4	100	random	4	0	345	180	1422	101
5	100	random	5	0	344	186	1454	101
6	1000	random	1	1	4075	2166	16332	1001
7	1000	random	2	1	4028	2118	16057	1001
8	1000	random	3	0	4027	2119	16062	1001
9	1000	random	4	0	4092	2184	16444	1001
10	1000	random	5	0	4023	2116	16034	1001
11	10000	random	1	3	47327	24918	182882	10001
12	10000	random	2	3	46981	24556	180730	10001
13	10000	random	3	3	47076	24674	181416	10001
14	10000	random	4	2	47193	24759	181928	10001
15	10000	random	5	2	47180	24752	181930	10001

Experimental results show that the number of comparisons and permutations grows proportionally to log(n). For example, when the input size was increased from 100 to 10,000 elements, the number of comparisons increased from approximately 350 to 47,000. The execution time also increased, but remains very small (about 2 ms for n = 10,000).The number of memory allocations increases linearly with n, which is explained by the need to store array elements.Thus, empirical data confirms the theoretical complexity of MaxHeap operations — O(log n).

5. Performance Analysis

Theoretical complexity of heap insertion and extraction is O(log n), which matches the observed empirical data. As the input size increases, the number of comparisons and swaps grows logarithmically, and memory allocations increase linearly. The results confirm that the implementation efficiently maintains the heap property, with consistent time growth proportional to log(n). The experimental curve aligns with the theoretical logarithmic trend, confirming O(log n) scalability for both insertion and extraction operations. While the current analysis confirms the efficiency of the MaxHeap implementation, several extensions are possible:

- Implementing a `buildHeap()` function to allow bulk heap construction in O(n) time.
- Supporting generic types (`Comparable<T>`) instead of being limited to integers.
- Comparing MaxHeap performance with alternative heap structures such as MinHeap, d-ary heaps, and Fibonacci heaps.
- Investigating parallel and multi-threaded heap implementations for high-performance computing scenarios.
- Performing larger-scale experiments (e.g., n > 1,000,000) to observe cache and memory effects.

6.Comparison with My Algorithm (MinHeap)

Feature	MinHeap (My Implementation)	MaxHeap (Partner's Implementation)
Type	Minimum heap	Maximum heap
Key Property	Parent ≤ children	Parent ≥ children
Main Operation	extractMin()	extractMax()
Complexity	O(log n)	O(log n)
Performance Metrics	Similar structure	Similar structure
Optimization Focus	Bottom-up heapify	Top-down sift-down

Both heaps demonstrate identical asymptotic performance but handle different priority orders.

The performance tracker can be reused between both for consistent analysis.

7. Conclusion

In this lab, a MaxHeap data structure was successfully implemented and analyzed. The experiment verified the theoretical time complexity of heap operations and demonstrated how performance metrics can be tracked programmatically. This analysis helps understand the efficiency of priority queue structures and lays the groundwork for comparing MinHeap and MaxHeap performance in future studies.

The partner's **MaxHeap implementation** is efficient, clear, and well-instrumented for performance analysis.
The algorithm maintains expected logarithmic time complexity and linear space usage.

Empirical results (once measured) should align closely with theoretical analysis.
Future improvements could include:

- Generic data type support
- More accurate memory tracking
- A bulk `buildHeap()` method

Overall, the implementation is **robust, maintainable, and meets the assignment's academic and technical goals**.

References:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

- Oracle. (n.d.). *Java Microbenchmark Harness (JMH)*. Retrieved from https://openjdk.org/projects/code-tools/jmh/

- Knowm. (n.d.). *XChart: Java Charting Library*. Retrieved from https://knowm.org/open-source/xchart/