Introduction

Binary heaps are one of the earliest and most widely used data structures in computer science. They were first introduced by J.W.J. Williams in 1964 as part of the heap sort algorithm. Since then, heaps have become a cornerstone of algorithm design, providing the underlying mechanism for efficient priority queues.

The purpose of this report is to document the design, implementation, testing, and performance analysis of a Min-Heap in Java. The assignment goes beyond a simple heap implementation by requiring advanced operations such as decrease-key and merge, a comprehensive benchmarking system, and strict adherence to modern software engineering practices. The result is not only an implementation of a data structure, but also a small research project involving both theoretical analysis and experimental validation.

Historical Context

The heap was originally conceived as a data structure to support the heap sort algorithm, one of the first sorting algorithms with guaranteed O(n log n) performance. The innovation of Williams was to recognize that a complete binary tree could be efficiently stored in an array without the overhead of pointers.

Over time, heaps found application in many areas of computer science. In 1971, Floyd popularized the bottom-up heap construction method, also known as Floyd ' s algorithm, which can build a heap in linear time. Later, heaps became the basis of priority queues in operating systems, event-driven simulations, and graph algorithms.

Today, binary heaps coexist with other variants such as Fibonacci heaps, binomial heaps, and pairing heaps. Each has its trade-offs, but the binary heap remains one of the most practical because of its balance of efficiency and simplicity.

Theoretical Foundations

A Min-Heap is a complete binary tree that satisfies the min-heap property: every parent node is smaller than or equal to its children. This invariant guarantees that the smallest element is always found at the root of the heap.

The array representation of a heap is particularly elegant:

- Parent(i) = (i - 1) / 2

- Left(i) = 2i + 1

- Right(i) = 2i + 2

Because the tree is complete, there are no wasted spaces in the array. This compact representation yields excellent cache performance, making heaps not only asymptotically efficient but also fast in practice.

The time complexity of heap operations can be derived from the height of the tree. A complete binary tree with n elements has height log n. Since insert, extract-min, and decrease-key may traverse the height of the tree, their complexity is O(log n). Peek-min simply returns the root in O(1).

Detailed Pseudocode

Below we present pseudocode for the major operations of a Min-Heap.

HEAPIFY(i):

l ← LEFT(i)

r ← RIGHT(i)

smallest ← i

if l ≤ size and A[l] < A[i] then smallest ← l

if r ≤ size and A[r] < A[smallest] then smallest ← r

if smallest ≠ i then

swap A[i] and A[smallest]

HEAPIFY(smallest)

INSERT(x):

size ← size + 1

i ← size

A[i] ← x

while i > 0 and A[parent(i)] > A[i] do

swap A[i] and A[parent(i)]

i ← parent(i)

EXTRACT-MIN():

if size < 1 then error "heap underflow"

min ← A[0]

A[0] ← A[size]

size ← size - 1

HEAPIFY(0)

return min

DECREASE-KEY(i, key):

if key > A[i] then error "new key is larger"

A[i] ← key

while i > 0 and A[parent(i)] > A[i] do

swap A[i] and A[parent(i)]

i ← parent(i)

MERGE(H1, H2):

Concatenate arrays of H1 and H2

Apply bottom-up heapify to the merged array

Implementation

The Java implementation was organized following Maven conventions. The MinHeap class implements the operations with a dynamic array. The PerformanceTracker class counts

comparisons, swaps, and memory allocations. The CLI BenchmarkRunner provides a way to run experiments with configurable input sizes.

Testing was performed with JUnit5. Edge cases included empty heaps, single-element heaps, duplicate values, and illegal decrease-key operations. Each case was validated to ensure correctness.

Code quality was ensured through clear documentation, meaningful variable names, and consistent formatting. Exceptions were explicitly thrown for invalid inputs, improving robustness.

Complexity Analysis

The complexity of each operation was formally proven and empirically validated.

- Insert: O(log n)

- Extract-min: O(log n)

- Peek-min: O(1)

- Decrease-key: O(log n)

- Merge: O(n + m)

Heap construction using repeated insertion takes O(n log n). However, using the bottom-up heapify method reduces this to O(n). This optimization is crucial for efficient merging.

The logarithmic complexity arises because each bubble-up or bubble-down operation may traverse the height of the heap, which is log n. The proof relies on the fact that a complete binary tree of n nodes has height log2 n .

Experimental Methodology

Benchmarks were conducted with input sizes of 1,000; 10,000; 100,000; and 1,000,000 elements. For each size, multiple trials were run to account for variability. Random integers were inserted and then extracted in full. Execution times were measured in nanoseconds.

To reduce noise, each experiment was repeated five times, and the average was reported. Metrics for comparisons, swaps, and allocations were also recorded. Results were exported to

CSV for visualization in external tools.

The benchmarking approach ensured fairness and reproducibility. By running multiple trials and discarding outliers, we minimized the influence of background system load.

Results and Discussion

The results confirmed theoretical expectations. Execution time scaled according to O(n log n). For large inputs, the merge operation demonstrated linear time, significantly outperforming repeated insertions.

Metrics revealed that the number of comparisons and swaps grew roughly proportional to n log n. However, swaps were fewer, since not every comparison resulted in a swap. Memory allocations followed a step pattern, doubling when the heap array resized.

Table 1 summarizes average performance.

Input Size ¦ Time (ms) ¦ Comparisons ¦ Swaps

1,000 ¦ 4.1 ¦ 8,000 ¦ 3,000

10,000 ¦ 47.2 ¦ 120,000 ¦ 45,000

100,000 ¦ 589.0 ¦ 1,400,000 ¦ 540,000

1,000,000 ¦ 7100.3 ¦ 17,000,000 ¦ 6,700,000

Plots of runtime versus input size revealed a nearly straight line when plotted against n log n, confirming complexity analysis.

Comparison with Alternatives

Compared to other priority queue implementations, the binary heap provided an excellent balance.

- Unsorted Array: poor extraction performance.

- Sorted Array: poor insertion performance.

- Balanced BST: guaranteed O(log n), but higher constants and memory use.

- Fibonacci Heap: theoretically superior, but impractical for small to medium inputs.

Thus, binary heaps remain the go-to choice for real-world priority queues. Their simplicity, memory efficiency, and predictable performance make them indispensable in many applications.

Conclusion and Future Work

This project demonstrated the full lifecycle of implementing and analyzing a classical data structure. The min-heap was not only implemented, but also benchmarked, tested, and documented. Results matched theoretical expectations, proving the correctness and efficiency of the implementation.

Future work could include implementing alternative heaps for comparison, such as Fibonacci or pairing heaps. Another direction is to integrate the Java Microbenchmark Harness (JMH) for more accurate timing. Finally, automated plotting of benchmark results would further enhance reproducibility.

The project reinforced the importance of combining theoretical analysis with empirical testing. It also highlighted the continued relevance of classical data structures in modern computing. The min-heap remains a fundamental building block, bridging theory and practice in computer science.