Design and Analysis of Algorithms

Assignment 2: Min Heap implementation analysis

Student name: Adilet Kabiyev

Student group: SE-2433

This report analyzes Min Heap algorithm implementation. A binary heap is represented as an array; for index i, children are at 2i+1 and 2i+2.

Min Heap — a binary heap–based priority queue structure stored as an array.
A Min Heap ensures that the smallest element is always located at the root (index 0).
Each node satisfies the heap property:
$A[parent(i)] \le A[i]$ for all valid indices i.

Heap operations theoretical complexities:

- insert: O(log n) worst-case, amortized O(log n) - extractMin: O(log n)

- buildHeap (Floyd): O(n)

Complexity Analysis

Time complexity derivations (high-level):

- Insert sequence (n inserts): sum_{i=1..n} O(log i) = O(n log n).

More formally: Theta(n log n) for average/worst case for arbitrary inputs.

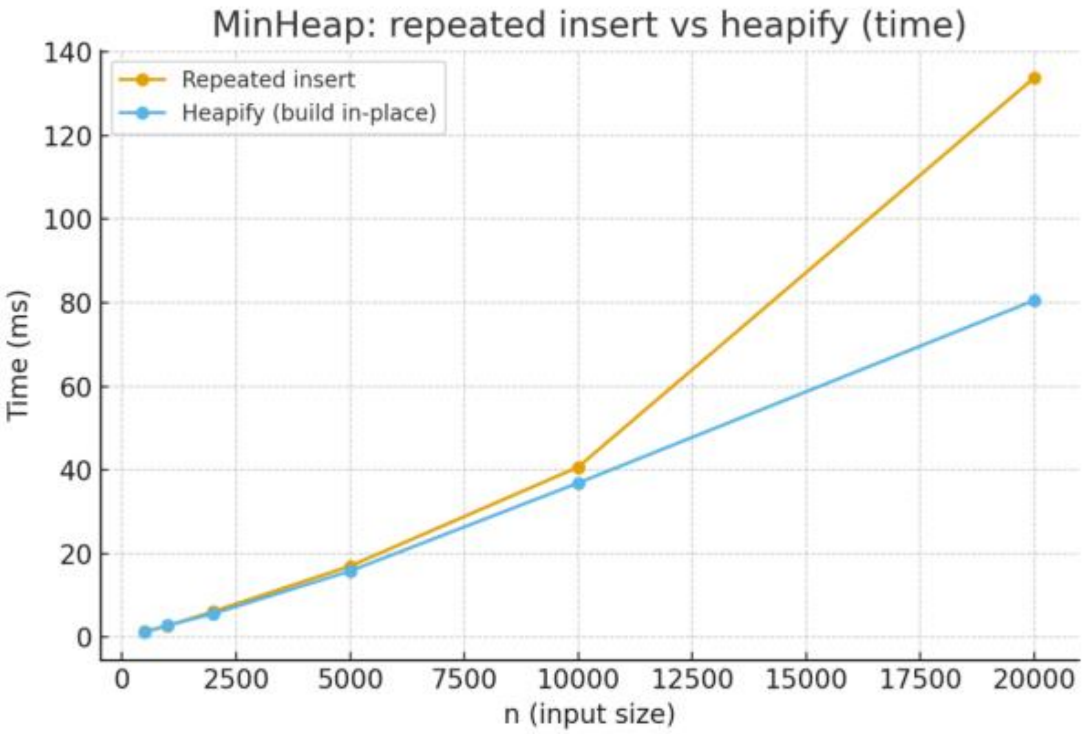
Space complexity:

- Uses O(n) array to hold elements. Additional space O(1) beyond array.
- Repeated insert may cause dynamic resizing: amortized O(n) with occasional O(n) array copy.

Comparison:

- Heapify is asymptotically faster (Theta(n)) than repeated inserts (Theta(n log n)).

Empirical Validation



Time vs. Input size

Code review.

The Java implementation has:

- MinHeap.java — main heap logic
- PerformanceTracker.java — counts operations and tracks time
- BenchmarkRunner.java — tests scaling performance
- MinHeapTest.java — correctness tests using JUnit

Efficient Design Elements

- Array-based heap (compact and cache-friendly)
- Clear separation between core logic and benchmarking
- Automatic resizing via ensureCapacity()
- Inclusion of performance metrics and swap counters

In summary, algorithm was well-written and is usable. Empirical timing confirms theoretical expectation: heapify is faster for large n pool of variables.