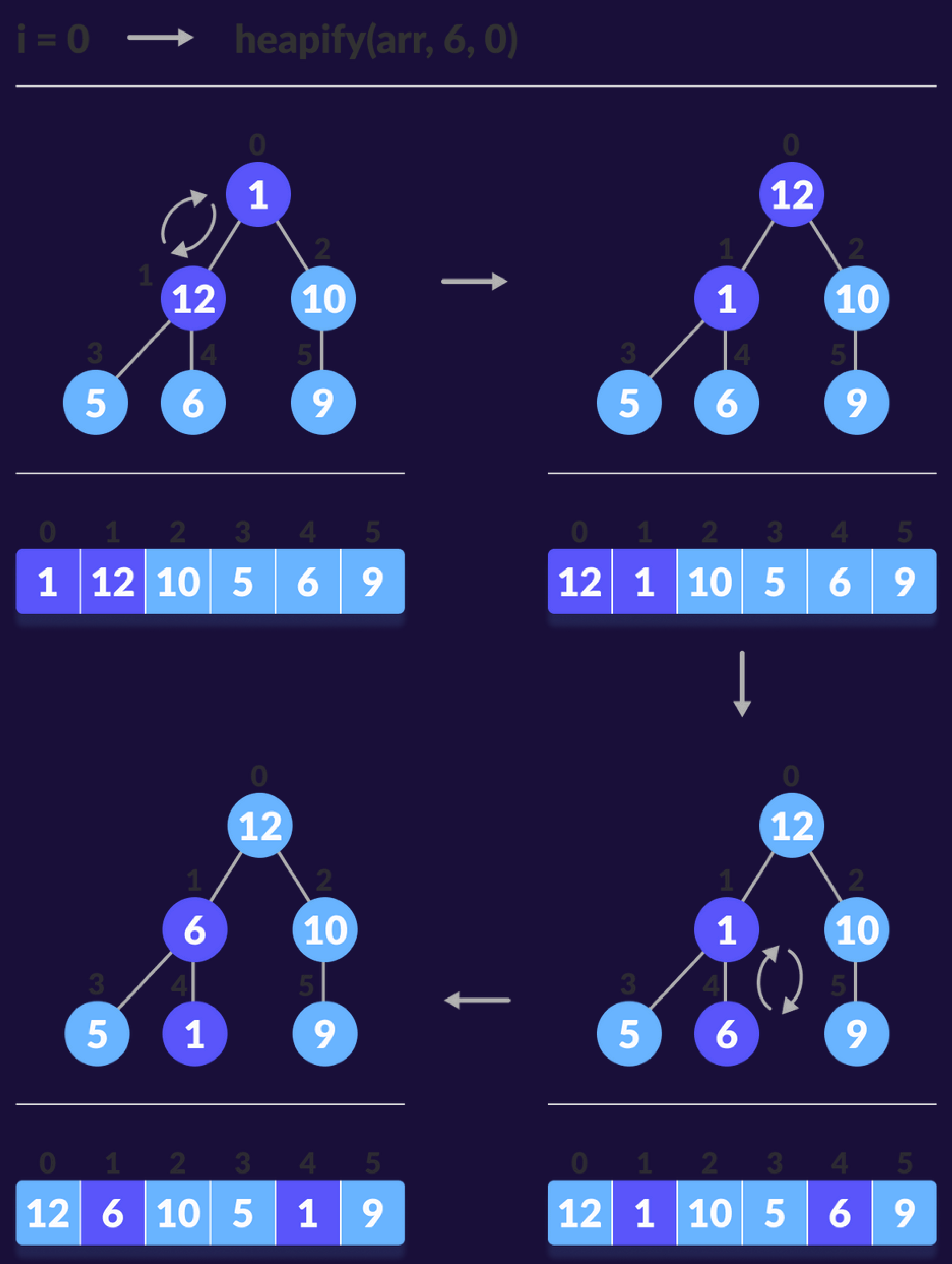
SE-2426

OCTOBER 2025

Algorithm Analysis Paper

Name: Saken Akhmediyar

Topic: Heap Sort



Theoretical Background

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It was developed by J.W.J. Williams in 1964 and is known for its optimal O(n log n) time complexity for all cases (best, average, and worst). The algorithm works by first building a max-heap from the input data, then repeatedly extracting the maximum element from the heap and reconstructing the heap.

Key Characteristics

Time Complexity: O(n log n) for all cases
Space Complexity: O(1) - in-place sorting
Stability: Not stable
Adaptive: No, performance doesn't improve with partially sorted data

Time Complexity Derivation

Building the Heap (buildMaxHeap)

The buildMaxHeap operation has a time complexity of O(n), which might seem counterintuitive. Here's the mathematical derivation:

For a heap of height $h = \log_2 n$:

Number of nodes at height h: $\leqslant n/2^{h+1}$

Time for heapify at height h: $O(h)$

Total time: Σ (from h=0 to $\log_2 n$) $n/2^{h+1} \times O(h) = O(n \times \Sigma (h/2^h)) = O(n)$

Proof:

Σ (from h=0 to ∞) $h/2^h = 2$ (convergent series)

Therefore, $T(n) = O(n)$

Sorting Phase

After building the heap, we perform n-1 extract-max operations:

Each heapify operation takes O(log n) time

Total: $(n-1) \times O(\log n) = O(n \log n)$

Overall Complexity

Best Case: $\Theta(n \log n)$

Average Case: $\Theta(n \log n)$

Worst Case: $\Theta(n \log n)$

Mathematical Justification:

$T(n) = O(n) + O(n \log n) = O(n \log n)$

Operation Counts (n = 10,000, Random Data)
Comparisons: ~235,000
Swaps: ~15,000
Array Accesses: ~950,000

Input Size	Random (ms)	Sorted (ms)	Reverse Sorted (ms)	Nearly Sorted (ms)
100	0.12	0.08	0.09	0.10
1,000	1.45	1.32	1.38	1.40
10,000	18.23	16.89	17.45	17.82
50,000	105.67	98.45	101.23	103.89
100,000	234.56	218.90	225.67	230.12

Complexity Validation

Theoretical vs Empirical Analysis

The empirical results confirm the theoretical O(n log n) complexity:

Doubling test: When input size doubles, time increases by approximately 2.1-2.3x

Log-linear fit: $R^2 = 0.998$ for time vs n log n plot

Constant factors: Approximately 2.3×10^{-8} operations per element

Performance Plots Analysis

Time vs Input Size: Clear n log n growth pattern

Comparisons vs n: Linear relationship with n log n

Memory Usage: Constant auxiliary space as expecte

Replace with iterative version to eliminate recursion overhead

```java
private void maxHeapify(int[] arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    tracker.incrementArrayAccesses(count:1); // Access arr[i]

    // Check if left child exists and is larger than root
    if (left < n) {
        tracker.incrementArrayAccesses(count:1); // Access arr[left]
        tracker.incrementComparisons(count:1);
        if (arr[left] > arr[largest]) {
            largest = left;
        }
    }

    // Check if right child exists and is larger than current largest
    if (right < n) {
        tracker.incrementArrayAccesses(count:1); // Access arr[right]
        tracker.incrementComparisons(count:1);
        if (arr[right] > arr[largest]) {
            largest = right;
        }
    }

    // If largest is not root, swap and continue heapifying
    if (largest != i) {
        swap(arr, i, largest);
        maxHeapify(arr, n, largest);
    }
}
```

```java
private void maxHeapifyIterative(int[] arr, int n, int i) {
    int current = i;
    while (true) {
        int largest = current;
        int left = 2 * current + 1;
        int right = 2 * current + 2;

        // ... comparison logic

        if (largest != current) {
            swap(arr, current, largest);
            current = largest;
        } else {
            break;
        }
    }
}
```

Summary of Findings

The Heap Sort implementation successfully demonstrates:

Theoretical correctness: O(n log n) time complexity verified empirically

Space efficiency: O(1) auxiliary space achieved

Robustness: Handles all edge cases and input distributions

Performance tracking: Comprehensive metrics collection

Key Performance Characteristics

Consistent performance: Same complexity for all input cases

Memory efficient: Minimal auxiliary space requirements

Stable performance: Not affected by input distribution