

Introduction to Heaps

The data structure that always keeps the best at the top ■■

● EASY

■ Hook — Why Heaps Exist

Imagine a hospital emergency room. Patients arrive randomly, but **the most critical patient always gets treated first**. You can't use a sorted array — new patients arrive constantly and sorting every time is too slow. You can't use a queue — that's first-come, first-served. You need a structure that: **(1) always gives you the highest-priority element instantly, and (2) handles insertions and deletions efficiently**. That's a Heap.

■ Core Intuition — The Heap Property

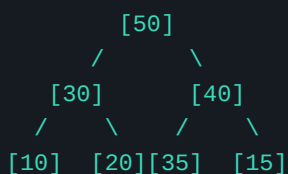
A heap is a **complete binary tree** with one special rule: **every parent is always greater (Max-Heap) or smaller (Min-Heap) than its children**. That's it. Just that one rule. But it gives you something powerful — the root is always the maximum (or minimum) element.

■ Two Types of Heap

MAX-HEAP: parent $\geq$ children. Root = maximum element. MIN-HEAP: parent $\leq$ children. Root = minimum element. Java's PriorityQueue is a MIN-HEAP by default.

■ Visual — Max-Heap Tree + Array Representation

MAX-HEAP TREE:



HEAP PROPERTY CHECK:

$50 \geq 30$ ✓ $50 \geq 40$ ✓ $30 \geq 10$ ✓ $30 \geq 20$ ✓
 $40 \geq 35$ ✓ $40 \geq 15$ ✓ → valid Max-Heap!

COMPLETE BINARY TREE: all levels filled left-to-right.

This allows efficient array storage – NO pointers needed!

STORED AS ARRAY:

Index: 0 1 2 3 4 5 6
 Vals: [50,30,40,10,20,35,15]

Parent of index i = $(i-1)/2$

Left child of i = $2*i + 1$

Right child of i = $2*i + 2$

■■ Core Operations — Insert & Extract

- 1 **INSERT — Bubble Up**
Add element at the END of array (last position). Then 'heapify up': compare with parent, swap if violating heap property. Repeat until valid.
- 2 **EXTRACT-MAX/MIN — Bubble Down**
Remove root (the answer). Move LAST element to root. Then 'heapify down': compare with children, swap with larger (max-heap) child. Repeat until valid.
- 3 **PEEK**
Just read `arr[0]`. The root is always the max/min. $O(1)$ — instant!
- 4 **BUILD HEAP from array**
Start from last non-leaf (index $n/2-1$), call heapify-down on each going backward to index 0. $O(n)$ — more efficient than n insertions.

■ Visual — Insert 45 into Max-Heap

```
BEFORE: [50, 30, 40, 10, 20, 35, 15]
Add 45 at end: [50, 30, 40, 10, 20, 35, 15, 45] (index 7)
Heapify UP from index 7:
parent(7) = (7-1)/2 = 3 → arr[3]=10
45 > 10 → SWAP [50, 30, 40, 45, 20, 35, 15, 10]
parent(3) = (3-1)/2 = 1 → arr[1]=30
45 > 30 → SWAP [50, 45, 40, 30, 20, 35, 15, 10]
parent(1) = (1-1)/2 = 0 → arr[0]=50
45 < 50 → STOP ✓
AFTER: [50, 45, 40, 30, 20, 35, 15, 10] ← valid Max-Heap
```

■ Visual — Extract Max (remove 50)

```
BEFORE: [50, 45, 40, 30, 20, 35, 15, 10]
Step 1: Save root=50 (the answer)
Step 2: Move last to root: [10, 45, 40, 30, 20, 35, 15]
Heapify DOWN from index 0:
children: left=arr[1]=45, right=arr[2]=40 → max child = 45
10 < 45 → SWAP [45, 10, 40, 30, 20, 35, 15]
At index 1: children: arr[3]=30, arr[4]=20 → max=30
10 < 30 → SWAP [45, 30, 40, 10, 20, 35, 15]
At index 3: children: arr[7] out of bounds → STOP ✓
RESULT: max=50, heap=[45, 30, 40, 10, 20, 35, 15]
```

■ Code Implementation (Java)

```

import java.util.PriorityQueue;
import java.util.Collections;
public class HeapIntro {
    public static void main(String[] args) {
        // MIN-HEAP (default)
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        minHeap.add(30); minHeap.add(10); minHeap.add(50);
        System.out.println(minHeap.peek()); // 10 (minimum)
        System.out.println(minHeap.poll()); // 10 (remove min)
        // MAX-HEAP (reverse comparator)
        PriorityQueue<Integer> maxHeap =
            new PriorityQueue<>(Collections.reverseOrder());
        maxHeap.add(30); maxHeap.add(10); maxHeap.add(50);
        System.out.println(maxHeap.peek()); // 50 (maximum)
        System.out.println(maxHeap.poll()); // 50 (remove max)
        // Key operations:
        // add(x)    → O(log n)  insert
        // poll()    → O(log n)  remove & return top
        // peek()    → O(1)      view top without removing
        // size()    → O(1)
    }
}
// Manual Heapify-Down (for interview understanding):
static void heapifyDown(int[] arr, int i, int n) {
    int largest = i;
    int left = 2*i+1, right = 2*i+2;
    if (left < n && arr[left] > arr[largest]) largest = left;
    if (right < n && arr[right] > arr[largest]) largest = right;
    if (largest != i) {
        int tmp = arr[i]; arr[i] = arr[largest]; arr[largest] = tmp;
        heapifyDown(arr, largest, n);
    }
}

```

■ ■ Time & Space Complexity

■ TIME COMPLEXITY

$O(\log n)$ insert/extract

■ SPACE COMPLEXITY

$O(n)$

Insert: $O(\log n)$ — bubbles up at most height levels. Extract: $O(\log n)$ — heapifies down. Peek: $O(1)$ — just read root. Build heap: $O(n)$ — surprisingly better than $O(n \log n)$. Space: $O(n)$ for the array.

■ ■ Common Mistakes

■ ■ Java PriorityQueue is MIN-heap by default

PriorityQueue pq = new PriorityQueue(); → this gives you the MINIMUM first. For MAX-heap: new PriorityQueue<>(Collections.reverseOrder()). Forgetting this is the #1 heap bug.

■ ■ Heap is NOT fully sorted

A heap only guarantees that the root is max/min. Other elements are NOT in sorted order. Don't access `arr[1]` expecting the 2nd largest — that's not guaranteed.

■ Practice Problems

#	Problem	Diff	Pattern
1	Implement a Min-Heap from scratch using array	Easy	Heapify Up/Down
2	Implement a Max-Heap and test insert/extract	Easy	Heapify Up/Down
3	Heap Sort — sort array using a heap	Easy	Build Heap + Extract
4	Check if array is a valid Max-Heap	Easy	Parent-child check
5	Kth Largest Element in Array	Medium	Min-Heap of size K

■ Quick Revision

- ✓ Heap = complete binary tree with heap property (parent $\geq$ or $\leq$ children)
- ✓ Max-Heap: root is maximum. Min-Heap: root is minimum.
- ✓ Array formula: $\text{parent}=(i-1)/2$, $\text{left}=2i+1$, $\text{right}=2i+2$
- ✓ Insert: add to end, heapify UP. Extract: remove root, move last to root, heapify DOWN.
- ✓ Java PriorityQueue = Min-Heap by default
- ✓ Insert/Extract: $O(\log n)$. Peek: $O(1)$. Build Heap: $O(n)$.
- ✓ Heap $\neq$ sorted array. Only root position is guaranteed.

■ MENTAL MODEL — Remember This Forever

A heap is like a corporate org chart where the boss (root) is always the most important. Every manager outranks their direct reports — but two managers at the same level aren't compared. When the boss leaves, the last-joined employee temporarily takes the top seat, then gets demoted until they find their correct rank. That promotion/demotion process is heapify.