

# Find Median from Data Stream

Two heaps that perfectly balance themselves — elegant and powerful ■■

● HARD

## ■ Hook

Real-time analytics, live leaderboards, financial tick data — you need the median of a growing dataset without re-sorting every time. Sorting after each insert is  $O(n \log n)$ . The **two-heap approach** gives  **$O(\log n)$  insert and  $O(1)$  median**. This is one of the most elegant data structure designs in all of DSA.

## ■ The Two-Heap Insight — Split the Data in Half

Maintain two heaps that together hold all numbers, split at the median:

- **maxHeap (left half)**: stores the smaller half. Root = largest of the small half.
- **minHeap (right half)**: stores the larger half. Root = smallest of the large half.

### ■ The Balance Invariant

Keep both heaps balanced: sizes differ by at most 1. If total count is EVEN:  $\text{median} = (\text{maxHeap.peek()} + \text{minHeap.peek()}) / 2.0$  If total count is ODD:  $\text{median} = \text{the larger heap's peek() (whichever has the extra element)}$ .

## ■ Visual — Insert stream: 5, 10, 1, 7, 3

[illegible]

```
Add 5:  maxHeap=[5]          minHeap=[]
        sizes: 1 vs 0 → balanced ✓    median = 5.0

Add 10: Add to maxHeap → [10,5]. 10 > minHeap? minHeap empty,
        rebalance: move max(10) to minHeap.
        maxHeap=[5]  minHeap=[10]    sizes: 1 vs 1 ✓
        median = (5+10)/2 = 7.5

Add 1:  Add to maxHeap → [5,1]. sizes 2 vs 1 → unbalanced!
        move max(5) to minHeap → maxHeap=[1] minHeap=[5,10]
        wait, sizes 1 vs 2 → move min(5) back to maxHeap
        maxHeap=[5,1]  minHeap=[10]  sizes 2 vs 1 ✓
        median = maxHeap.peek() = 5.0

Add 7:  both halves, rebalance → maxHeap=[5,1]  minHeap=[7,10]
        sizes 2 vs 2 ✓    median=(5+7)/2 = 6.0

Add 3:  maxHeap=[5,3,1]  minHeap=[7,10]  sizes 3 vs 2 ✓
        median = maxHeap.peek() = 5.0
```

## Code Implementation (Java)

```
import java.util.*;

public class MedianFinder {
    // maxHeap: left half (smaller numbers, max at top)
    private PriorityQueue<Integer> maxHeap =
        new PriorityQueue<>(Collections.reverseOrder());
    // minHeap: right half (larger numbers, min at top)
    private PriorityQueue<Integer> minHeap =
        new PriorityQueue<>();

    public void addNum(int num) {
        // Step 1: Always add to maxHeap first
        maxHeap.offer(num);
        // Step 2: Ensure all maxHeap values ≤ all minHeap values
        if (!minHeap.isEmpty() && maxHeap.peek() > minHeap.peek()) {
            minHeap.offer(maxHeap.poll());
        }
        // Step 3: Balance sizes (differ by at most 1)
        if (maxHeap.size() > minHeap.size() + 1) {
            minHeap.offer(maxHeap.poll());
        } else if (minHeap.size() > maxHeap.size()) {
            maxHeap.offer(minHeap.poll());
        }
    }

    public double findMedian() {
        if (maxHeap.size() == minHeap.size()) {
            return (maxHeap.peek() + minHeap.peek()) / 2.0;
        }
        return maxHeap.peek(); // maxHeap has the extra element
    }
}
```

## Time & Space Complexity

#### ■ TIME COMPLEXITY

**$O(\log n)$  add /  $O(1)$  median**

#### ■ SPACE COMPLEXITY

**$O(n)$**

addNum: at most 3 heap operations (each  $O(\log n)$ ). findMedian:  $O(1)$  — just peek() both roots. Space:  $O(n)$  — all elements stored across both heaps.

## ■ Common Mistakes

### ■ Not cross-checking before balancing

After adding to maxHeap, check if `maxHeap.peek() > minHeap.peek()`. If so, the 'partition line' shifted — move the maxHeap root to minHeap to restore the invariant. Skipping this step corrupts the median.

### ■ Integer overflow in median calculation

$(a + b) / 2.0$  — both a and b are ints so a+b might overflow for large values. Cast first:  $(\text{double})a/2 + (\text{double})b/2$ , or use long arithmetic.

## ■ Practice Problems

| # | Problem                               | Diff   | Pattern                      |
|---|---------------------------------------|--------|------------------------------|
| 1 | Find Median from Data Stream (LC 295) | Hard   | Two Heaps                    |
| 2 | Sliding Window Median (LC 480)        | Hard   | Two Heaps + removal          |
| 3 | IPO (LC 502)                          | Hard   | Two Heaps (profit + capital) |
| 4 | Maximum Sum Circular Subarray         | Medium | Kadane variant               |
| 5 | Median of Two Sorted Arrays (LC 4)    | Hard   | Binary Search                |

## ■ Quick Revision

- ✓ maxHeap = left/smaller half minHeap = right/larger half
- ✓ Invariant: `maxHeap.peek() ≤ minHeap.peek()` always
- ✓ Size balance:  $|\text{maxHeap.size()} - \text{minHeap.size()}| \leq 1$
- ✓ Even total:  $\text{median} = (\text{max.peek()} + \text{min.peek()}) / 2.0$
- ✓ Odd total:  $\text{median} = \text{peek() of larger heap}$
- ✓ addNum:  $O(\log n)$  | findMedian:  $O(1)$

### ■ MENTAL MODEL — Remember This Forever

Imagine a number line split perfectly in half. Left half lives in a max-heap (so you can see the right edge). Right half lives in a min-heap (so you can see the left edge). The median is the meeting point of these two edges. Every insertion adjusts the split line — at most 2 moves to rebalance.