

Topic Introduction

Today, we're diving into one of the most classic interview themes: **finding the Kth element**. Whether it's the Kth smallest, largest, or closest, these problems pop up everywhere—trees, arrays, and more.

Let's clarify the central idea:

Finding the Kth Element means efficiently determining the value that would appear in the Kth position if you sorted the structure according to some rule (e.g., smallest to largest).

Why is this useful?

- It tests your knowledge of common data structures and algorithms.
- It's a building block for more complex problems, like median-finding, order statistics, or range queries.
- Interviewers love it because it reveals your understanding of algorithmic trade-offs.

Where does this pattern show up?

- Arrays: Find the Kth largest or smallest number.
- Trees: Kth smallest in a Binary Search Tree (BST).
- Proximity problems: Find the K elements closest to a target value.

Let's see a simple example (not from our main problems):

> **Example:** Given [7, 2, 5, 3, 9], find the 2nd smallest element.

Sort the array: [2, 3, 5, 7, 9]. The 2nd smallest is 3.

But in interviews, you're usually *not* allowed to just sort the whole thing, or the data is in a different structure (like a BST). So the real challenge is: **How can you find the answer more efficiently, using the properties of the data structure?**

Why Group These Three Problems?

Today's trio is united by the task of finding the Kth element, but each does it in a different context:

- **Kth Smallest Element in a BST:** Uses the BST's in-order traversal property.
- **Kth Largest Element in an Array:** Uses heap or quickselect techniques to avoid full sorting.
- **Find K Closest Elements:** Combines binary search and two pointers to efficiently select a window of K elements closest to a given value.

These showcase how the core challenge adapts based on the data structure and problem constraints.

Problem 1: Kth Smallest Element in a BST

[Leetcode 230](#)

Problem Restatement

PrepLetter: Kth Smallest Element in a BST and similar

Given a Binary Search Tree (BST), return the Kth smallest element in it.

Example:

If the BST is:



and $k = 3$, the answer is **3** (the third smallest element).

Another Test Case:

Given the tree:



and $k = 2$, the answer is **2**.

Approach

Pen-and-paper tip:

If this tree structure feels tricky, try drawing a small BST and tracing in-order traversal.

Brute-force

- Collect all values by traversing the tree.
- Sort them, then return the Kth smallest.

Time Complexity: $O(N \log N)$

Why? Traversal is $O(N)$, sorting is $O(N \log N)$.

Optimal Approach: In-order Traversal

Key Pattern:

In a BST, in-order traversal (left, root, right) visits nodes *in sorted order*.

How it works:

- Traverse the tree in-order.
- Keep a count as you visit nodes.
- When you reach the Kth node, return its value.

Step-by-step:

- Start at the root.
- Recursively visit left subtree.
- Visit root node (increment count).
- If count == k, record answer.
- Recurse right.

Why is this efficient?

No need to collect or sort all values. Just count as you go.

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def kthSmallest(root, k):
    # In-order traversal: left, root, right
    stack = []
    current = root
    count = 0

    while True:
        # Go as left as possible
        while current:
            stack.append(current)
            current = current.left
        if not stack:
            break
        current = stack.pop()
        count += 1
        if count == k:
            return current.val
        # Move to right subtree
        current = current.right
```

Time Complexity: $O(H + k)$ in balanced tree (H is tree height), worst $O(N)$

Space Complexity: $O(H)$ for the stack

Walkthrough

Let's trace this input:

5

PrepLetter: Kth Smallest Element in a BST and similar

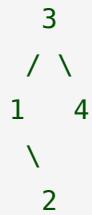


k = 3

- Stack: []
- current: root (5)
- Go left: 5 -> 3 -> 2 -> 1 (stack: [5,3,2,1])
- Pop 1: count=1
- Pop 2: count=2
- Pop 3: count=3 -> Return 3

Try this one yourself:

Tree:



k = 2

What should the answer be? (*Try doing the in-order traversal on paper!*)

Take a moment to try solving this yourself before reviewing the code.

Reflection:

Did you know you could also solve this recursively? Try implementing a recursive in-order traversal that stops after visiting k nodes!

Problem 2: Kth Largest Element in an Array

[Leetcode 215](#)

Problem Restatement

Given an unsorted array, find the Kth largest element.

Example:

nums = [3,2,1,5,6,4], k = 2

Return 5 (the second largest element).

Another Test Case:

```
nums = [7,10,4,3,20,15], k = 3
```

Return 10

How is this Similar or Different?

- Similar: We're still finding a Kth "ordered" element.
- Different: The data is an *array*, not a BST. No inherent ordering; need a different approach.

Brute-force

- Sort the array, return the element at index -k.

Time Complexity: $O(N \log N)$

Optimal Approach: Min-Heap

Pattern:

Keep a *min-heap* of size k.

- For each number:
 - Add to the heap.
 - If heap size exceeds k, remove the smallest.
- After processing all numbers, the heap's root is the Kth largest.

Why?

- Min-heap keeps the k largest elements seen so far.
- The smallest of these k is the Kth largest overall.

Step-by-step:

- Build a min-heap of the first k elements.
- For each remaining element:
 - If it's larger than the heap's root, pop the root and push the new element.
- Heap's root is the answer.

Pseudocode:

```
function findKthLargest(nums, k):  
    heap = min_heap(nums[0..k-1])  
    for num in nums[k:]:  
        if num > heap[0]:  
            pop heap[0]  
            push num into heap  
    return heap[0]
```

Time Complexity: $O(N \log k)$

Space Complexity: $O(k)$

Alternative:

Quickselect (average O(N)), but heap is simpler and more reliable for interviews.

Walkthrough

Example:

```
nums = [3,2,1,5,6,4], k = 2
```

- First k: [3,2] => heap: [2,3]
- Next: 1 (not greater than 2), skip.
- Next: 5 (5 > 2), pop 2, push 5 => heap: [3,5]
- Next: 6 (6 > 3), pop 3, push 6 => heap: [5,6]
- Next: 4 (4 < 5), skip.

Final answer: 5

Try this one:

```
nums = [1, 23, 12, 9, 30, 2, 50], k = 3
```

What should the answer be?

Trace of Example Case:

- Heap after first k: [3,2] -> [2,3]
- Next: 1 (skip); 5 (push, heap=[3,5]); 6 (push, heap=[5,6]); 4 (skip)
- Result: 5

Time Complexity: Each heap operation is $\log k$, and we do it N times.

Problem 3: Find K Closest Elements

[Leetcode 658](#)

Problem Restatement

Given a sorted array, a value **x**, and an integer **k**, find the **k** closest elements to **x**. If there is a tie, select the smaller number.

Example:

```
arr = [1,2,3,4,5], k = 4, x = 3
```

Return [1,2,3,4]

Another Test Case:

```
arr = [1,2,3,4,5], k = 4, x = -1
```

Return [1,2,3,4]

What's Different?

- Now we care about *closeness* to x, not about being largest/smallest.
- The array is sorted, which lets us use binary search and two pointers.

Brute-force

- Compute abs difference to x for every element.
- Sort by difference, then by value.
- Take first k elements, sort them.

Time Complexity: $O(N \log N)$

Optimal Approach: Binary Search for Window

Pattern:

We want a window of k consecutive elements whose endpoints are as close as possible to x.

Step-by-step:

- Set left = 0, right = len(arr) - k
- While left < right:
 - mid = (left + right) // 2
 - Compare abs(x - arr[mid]) vs abs(x - arr[mid + k])
 - If arr[mid] is farther, move left to mid + 1; else, move right to mid.
- Return arr[left:left + k]

Pseudocode:

```
function findClosestElements(arr, k, x):  
    left = 0  
    right = len(arr) - k  
    while left < right:  
        mid = (left + right) // 2  
        if abs(x - arr[mid]) > abs(x - arr[mid + k]):  
            left = mid + 1  
        else:  
            right = mid  
    return arr[left : left + k]
```

Time Complexity: $O(\log(N-k) + k)$

Space Complexity: $O(k)$

Walkthrough

Example:

arr = [1,2,3,4,5], k = 4, x = 3

- left = 0, right = 1
- mid = 0, compare abs(3-1)=2 vs abs(3-5)=2
- Since tie, move right = mid = 0
- Return arr[0:4] = [1,2,3,4]

Try this one:

`arr = [1,3,3,4,5], k = 3, x = 3`

What's the answer?

Nudge:

Think about why we use binary search on the window's starting index, not on the elements themselves. Could you solve this problem with a heap instead? Try it!

Summary and Next Steps

Today you saw how the **Kth element** theme reappears in arrays and trees, but each setting calls for a different tool:

- **BST**: In-order traversal gives sorted order.
- **Array**: Heap (min or max) efficiently tracks the top k.
- **Sorted Array, Closest Elements**: Binary search pinpoints the best window.

Key insights:

- Always look for properties of the data structure (BST, sorted, etc.).
- Brute-force is often easy but slow; optimal solutions use traversal, heaps, or binary search.
- Be careful with indexing and off-by-one errors!

Common traps:

- Sorting when you can do better.
- Not using the BST's ordering in tree problems.
- Forgetting to handle ties or edge cases in "closest" problems.

Action List:

- Solve all 3 problems yourself, including one with code provided.
- Try solving Problem 2 and 3 using a different technique (e.g., quickselect for largest element, heap for closest elements).
- Look for other "Kth element" problems (like median, order statistics).
- Compare your solution to others, especially for tricky edge cases.
- Don't be discouraged by complexity—the more you practice, the more these patterns will stick!

Happy coding—your next interview is closer than you think!