

Search in BST

● EASY

■ Introduction & Real-World Analogy

Imagine a library where books are arranged strictly: every shelf on the left holds books with smaller catalog numbers, and every shelf on the right holds larger ones. To find a book, you never scan every shelf — you just compare at each junction and cut half the search space. That is exactly how a Binary Search Tree works.

Why it matters in interviews: BST Search is the gateway to understanding all BST operations. It also teaches the divide-and-conquer mindset used throughout DSA.

■ Core Intuition

The BST Property

- Left subtree of a node contains ONLY nodes with keys LESS THAN the node's key.
- Right subtree contains ONLY nodes with keys GREATER THAN the node's key.
- Both left and right subtrees are themselves BSTs.
- No duplicates (in standard definition).

Search exploits this property. At every node you make a 3-way decision: found it? go left? go right? This eliminates half the remaining nodes at each step — giving $O(\log n)$ average-case performance.

■ How It Works Internally

Step 1 — Start at root: The root is your entry point. The entire tree is reachable from here.

Step 2 — Compare target with current node: If $\text{target} == \text{node.val} \rightarrow \text{FOUND}$. If $\text{target} < \text{node.val} \rightarrow \text{search LEFT subtree}$. If $\text{target} > \text{node.val} \rightarrow \text{search RIGHT subtree}$.

Step 3 — Recurse or move pointer: Each call/move descends one level. Call stack depth = height of tree.

Step 4 — Base case: If current node is null $\rightarrow$ target does NOT exist in tree. Return null/false.

■ Visual — BST Search Trace

ASCII Tree Diagram

10 ← Root

/ \

5 15

/ \ / \

3 7 12 20

Search for 7:


```

return searchBST(root.right, val);
}
}

// ■■ Iterative Approach (No recursion stack overhead) ■■■■■■■■
class SolutionIterative {
public TreeNode searchBST(TreeNode root, int val) {
    TreeNode curr = root;
    while (curr != null) {
        if (val == curr.val) return curr; // Found
        else if (val < curr.val) curr = curr.left;
        else curr = curr.right;
    }
    return null; // Not found
}
}

```

■■ Time & Space Complexity

Case	Time	Space	Why
Best	$O(1)$	$O(1)$	Target is the root node
Average	$O(\log n)$	$O(\log n)$	Balanced BST — each step halves search space
Worst	$O(n)$	$O(n)$	Skewed tree (like linked list) — degenerate case

Iterative vs Recursive Space

- Recursive: $O(h)$ stack space where h = tree height.
- Iterative: $O(1)$ space — no call stack used. Always prefer iterative for space efficiency.

■■ Common Mistakes

Avoid These Errors

- Forgetting the null check first — always check `root==null` before `root.val`.
- Using `==` for object comparison instead of `.val` — compare integers, not node references.
- Assuming BST is always balanced — worst case is $O(n)$ on skewed trees.

- Returning wrong thing — the problem asks to return the SUBTREE rooted at found node, not just true/false.

■ Patterns & Connections

Search is the foundation of every BST operation. Insert, Delete, Validate — all rely on the same navigation logic. Binary Search on arrays uses the same 3-way comparison. This pattern reappears in segment trees, trie traversal, and decision trees in ML.

■ Practice Problems

Case	Time	Space	Why
Easy	LC 700	Search in a BST	Direct application
Easy	LC 270	Closest BST Value	Modify search to track min diff
Medium	LC 285	Inorder Successor in BST	Search + successor logic
Hard	LC 220	Contains Duplicate III	BST-based sliding window

■ Quick Revision

- BST Property: $\text{left} < \text{root} < \text{right}$ (for each subtree)
- At each node: compare $\rightarrow$ go left/right/return
- Recursive: elegant, $O(h)$ space | Iterative: optimal, $O(1)$ space
- Average $O(\log n)$, Worst $O(n)$ on skewed tree
- Always check null base case first

■ The Mental Model

Remember This Forever

- "Every node is a bouncer with one rule: numbers smaller than me go left, bigger go right."
- You never guess — you always KNOW which half to search. This is the power of sorted structure.