

# ■ Binary Search Trees

Complete DSA Notes — Beginner to Advanced

7 Chapters • Java Code • Visual Diagrams • Interview Ready

|    |                               |        |
|----|-------------------------------|--------|
| 01 | Search in BST                 | EASY   |
| 02 | Insert into BST               | MEDIUM |
| 03 | Delete Node in BST            | MEDIUM |
| 04 | Validate Binary Search Tree   | MEDIUM |
| 05 | Kth Smallest in BST           | MEDIUM |
| 06 | Lowest Common Ancestor of BST | MEDIUM |
| 07 | Two Sum IV — Input is BST     | EASY   |

# 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.

# Insert into BST

● MEDIUM

## ■ Introduction

Think of inserting a new employee into a company org-chart that is always sorted by employee ID. You walk down the hierarchy, at each manager asking "is your ID smaller or larger?" until you find an empty desk — that's where the new person sits. BST Insert works exactly the same way.

## ■ Core Intuition

Key insight: **A new node ALWAYS becomes a LEAF.** Why? Because we navigate using the BST property until we fall off the tree — and that null spot is where we place the new node. We never insert in the middle.

### The Golden Rule

- Navigate exactly like Search.
- When you would return null (not found), instead CREATE and return a new node.
- The BST property is automatically maintained because you followed it to get there.

## ■■ How It Works Internally

**Step 1:** If root is null → create new node here (this IS the insertion point).

**Step 2:** If  $val < root.val$  → recursively insert into LEFT subtree. Attach result back to root.left.

**Step 3:** If  $val > root.val$  → recursively insert into RIGHT subtree. Attach result back to root.right.

**Step 4:** Return root at every level — this rebuilds the tree structure on the way back up.

## ■ Visual — Insert 6 into BST

### Insert Visualization

Before Insert(6): After Insert(6):

10 10

/ \ / \

5 15 5 15

\ \

7 7

/

6 ← NEW LEAF

Path taken: 10→left→5→right→7→left→null → INSERT HERE



```

while (true) {
    if (val < curr.val) {
        if (curr.left == null) { curr.left = newNode; break; }
        curr = curr.left;
    } else {
        if (curr.right == null) { curr.right = newNode; break; }
        curr = curr.right;
    }
}
return root;
}
}

```

## ■ Time & Space Complexity

| Case    | Time        | Space       | Why                                           |
|---------|-------------|-------------|-----------------------------------------------|
| Best    | $O(\log n)$ | $O(\log n)$ | Balanced BST, insert near top                 |
| Average | $O(\log n)$ | $O(\log n)$ | Random insertions, tree stays balanced        |
| Worst   | $O(n)$      | $O(n)$      | Sorted input → skewed tree (like linked list) |

## ■ Common Mistakes

### Common Errors

- NOT returning root at each level — without "return root", you lose the tree structure.
- Not re-assigning root.left/right — the line `root.left = insertIntoBST(...)` is crucial.
- Inserting duplicates — standard BST does not allow them; clarify with interviewer.
- Forgetting iterative needs explicit break — infinite loop if you don't break on insertion.

## ■ Patterns & Connections

This "navigate and place" pattern appears in Tries (insert character by character), Segment Trees (update a position), and AVL/Red-Black trees (insert + rebalance). The recursive return-and-reattach idiom is a fundamental tree pattern.

## ■ Practice Problems

| Case | Time | Space | Why |
|------|------|-------|-----|
|------|------|-------|-----|

|        |         |                                         |                       |
|--------|---------|-----------------------------------------|-----------------------|
| Medium | LC 701  | Insert into a Binary Search Tree        | Direct application    |
| Medium | LC 1305 | All Elements in Two Binary Search Trees | Insert + merge        |
| Hard   | LC 449  | Serialize and Deserialize BST           | Insert to reconstruct |

## ■ Mental Model

### Remember This Forever

- "Follow the search path. Where you would return NOT FOUND — plant the new node."
- New nodes are ALWAYS leaves. The tree's past structure is never disturbed.

# Delete Node in BST

● MEDIUM

## ■ Introduction

Delete is the trickiest BST operation. Unlike insert (always a leaf), deleting a node can disrupt the tree structure. The challenge: remove a node while keeping the BST property intact. There are three distinct cases you must handle.

## ■ The Three Cases

### Case Analysis

- Case 1 — Node is a LEAF (no children): Simply remove it. Return null.
- Case 2 — Node has ONE child: Replace node with that child. Return the child.
- Case 3 — Node has TWO children: Find inorder successor (smallest in right subtree). Copy its value here. Delete the successor from right subtree.

## ■■ Inorder Successor

The **inorder successor** of a node is the smallest node in its right subtree (go right once, then go left as far as possible). It's the next larger value — perfect to replace the deleted node while maintaining BST order.

## ■ Visual — All Three Cases

### Delete Cases Visual

Case 1: Delete 3 (leaf) Case 2: Delete 5 (one child)

10 10

/ \ / \

5 15 → 5 stays, 5 15

/ \ 3 removed \

3 7 7

Case 3: Delete 5 (two children)

10 10

/ \ → / \

5 15 6 15

/ \ / \

3 7 3 7

/

```
6 ← inorder successor of 5 is 6 (leftmost of right subtree)

Copy 6 to node 5, delete 6 from right subtree
```

## ■ Java Implementation

```
Java

class Solution {

    public TreeNode deleteNode(TreeNode root, int key) {

        if (root == null) return null; // Not found

        if (key < root.val) {
            // Key is in left subtree
            root.left = deleteNode(root.left, key);
        } else if (key > root.val) {
            // Key is in right subtree
            root.right = deleteNode(root.right, key);
        } else {
            // Found the node to delete!

            // Case 1 & 2: no left child → replace with right
            if (root.left == null) return root.right;
            // Case 2: no right child → replace with left
            if (root.right == null) return root.left;

            // Case 3: two children → find inorder successor
            TreeNode successor = findMin(root.right);
            root.val = successor.val; // Copy successor value
            // Delete successor from right subtree
            root.right = deleteNode(root.right, successor.val);
        }

        return root;
    }

    // Helper: find minimum node in a subtree
    private TreeNode findMin(TreeNode node) {
        while (node.left != null) node = node.left;
        return node;
    }
}
```

```
}
```

## ■ Time & Space Complexity

| Case    | Time        | Space       | Why                            |
|---------|-------------|-------------|--------------------------------|
| Best    | $O(\log n)$ | $O(\log n)$ | Balanced tree, delete near top |
| Average | $O(\log n)$ | $O(\log n)$ | Search + findMin both $O(h)$   |
| Worst   | $O(n)$      | $O(n)$      | Skewed tree                    |

## ■ Common Mistakes

### Watch Out For

- Forgetting to return root — always return root to maintain tree linkage.
- Using predecessor instead of successor — both work, but be consistent.
- Not updating root.right after deleting successor — you must re-delete.
- Mixing up Cases 1 and 2 — Case 1 is a subset of "left == null" check.

## ■ Practice Problems

| Case   | Time   | Space                               | Why                   |
|--------|--------|-------------------------------------|-----------------------|
| Medium | LC 450 | Delete Node in a BST                | Direct application    |
| Medium | LC 669 | Trim a Binary Search Tree           | Delete range of nodes |
| Hard   | LC 315 | Count of Smaller Numbers After Self | BST with deletion     |

## ■ Mental Model

### Remember Forever

- "Leaf → vanish. One child → step aside, let child take your place. Two children → find your replacement (inorder successor), hand over your identity."

# Validate Binary Search Tree

● MEDIUM

## ■ Introduction

You are given a binary tree. Is it a valid BST? This sounds simple but has a famous trap. The naive check "left < root < right" at each node is WRONG. A valid BST has a global constraint: every node in the left subtree must be less than ALL ancestors above it. You need to pass bounds down the tree.

## ■ The Key Insight — Min/Max Bounds

### Why Local Check Fails

- Consider: root=10, left=5, left.right=12. Locally,  $5 < 10$  and  $12 > 5$  look fine.
- But  $12 > 10$  (the ancestor) — this violates BST property!
- Fix: Pass a valid RANGE [min, max] to each node. Node must lie strictly within range.
- Root starts with range (-infinity, +infinity).
- Going LEFT: new range = (min, parent.val)
- Going RIGHT: new range = (parent.val, max)

## ■ Visual — Bounds Propagation

### Bounds Visualization

```

10 [range: -inf, +inf]
 / \
5 15
[range: -inf, 10] [range: 10, +inf]
 / \
3 7
[-inf, 5] [5, 10] ← 7 is in [5, 10] ✓

```

Invalid Example:

```

10 [-inf, +inf]
 / \
5 15
[-inf, 10] [10, +inf]
 \
12 → 12 must be in [-inf, 10] but 12 > 10 → INVALID ✗

```

## ■ Java Implementation

### Java

```
class Solution {  
  
    public boolean isValidBST(TreeNode root) {  
  
        return validate(root, Long.MIN_VALUE, Long.MAX_VALUE);  
    }  
  
    // Each node must satisfy: min < node.val < max  
  
    private boolean validate(TreeNode node, long min, long max) {  
  
        if (node == null) return true; // Empty is valid  
  
        // Check current node is within bounds  
        if (node.val <= min || node.val >= max) return false;  
  
        // Left child must be in (min, node.val)  
        // Right child must be in (node.val, max)  
        return validate(node.left, min, node.val)  
        && validate(node.right, node.val, max);  
    }  
}  
  
// ■■ Alternative: Inorder Traversal should be strictly increasing ■■  
  
class SolutionInorder {  
  
    long prev = Long.MIN_VALUE;  
  
    public boolean isValidBST(TreeNode root) {  
  
        if (root == null) return true;  
        if (!isValidBST(root.left)) return false;  
        if (root.val <= prev) return false; // Not strictly increasing  
        prev = root.val;  
        return isValidBST(root.right);  
    }  
}
```

## ■■ Time & Space Complexity

| Case      | Time   | Space  | Why                                                 |
|-----------|--------|--------|-----------------------------------------------------|
| All cases | $O(n)$ | $O(h)$ | Visit every node exactly once; stack depth = height |

## ■ Common Mistakes

### Classic Traps

- Using int bounds — use Long.MIN/MAX to handle Integer.MIN/MAX values in tree.
- Using local check only ( $\text{left} < \text{root} < \text{right}$ ) — this misses ancestor violations.
- Using  $\geq$  instead of  $>$  — BST requires STRICTLY greater/less (no duplicates).
- Inorder approach: using a global variable can cause issues in multi-threaded context.

## ■ Practice Problems

| Case   | Time   | Space                       | Why                       |
|--------|--------|-----------------------------|---------------------------|
| Medium | LC 98  | Validate Binary Search Tree | Direct application        |
| Medium | LC 530 | Min Absolute Diff in BST    | Inorder traversal variant |
| Hard   | LC 99  | Recover Binary Search Tree  | Find two swapped nodes    |

## ■ Mental Model

### Remember Forever

- "Each node carries a passport. The passport stamps a valid range. If a node's value is outside its passport range — the BST is invalid."
- Inorder traversal of valid BST gives STRICTLY increasing sequence — use this as a quick validator.



## ■ Time & Space Complexity

| Case | Time       | Space  | Why                                                                |
|------|------------|--------|--------------------------------------------------------------------|
| Time | $O(h + k)$ | $O(h)$ | $h$ to reach leftmost, then $k$ steps. $h = O(\log n)$<br>balanced |

|       |        |        |                      |
|-------|--------|--------|----------------------|
| Worst | $O(n)$ | $O(n)$ | Skewed tree, $k = n$ |
|-------|--------|--------|----------------------|

## Common Mistakes

### Avoid These

- Doing full inorder and sorting — wasteful, you can stop at  $k$ .
- Forgetting inorder is Left→Root→Right (not Root→Left→Right).
- Off-by-one:  $k$  is 1-indexed, so count starts at 0 and you check  $\text{count} == k$ .

## Practice Problems

| Case   | Time   | Space                          | Why                   |
|--------|--------|--------------------------------|-----------------------|
| Medium | LC 230 | Kth Smallest Element in BST    | Direct application    |
| Medium | LC 671 | Second Min Node in Binary Tree | Similar counting idea |
| Hard   | LC 378 | Kth Smallest in Sorted Matrix  | Binary search variant |

## Mental Model

### Remember Forever

- "BST inorder is a free sort. Every tree problem involving order/rank → think inorder traversal."

# Lowest Common Ancestor of BST

● MEDIUM

## ■ Introduction

The Lowest Common Ancestor (LCA) of two nodes  $p$  and  $q$  is the deepest node that has both  $p$  and  $q$  as descendants. For a generic binary tree, this requires  $O(n)$ . But for a BST, the BST property gives us a shortcut: we can find LCA in  $O(h)$ !

## ■ Core Intuition

### The Fork Intuition

- Walk from root. At some node,  $p$  and  $q$  go in different directions  $\rightarrow$  THAT node is LCA.
- If both  $p$  and  $q$  are less than current node  $\rightarrow$  LCA is in LEFT subtree.
- If both  $p$  and  $q$  are greater  $\rightarrow$  LCA is in RIGHT subtree.
- If they diverge (one left, one right) OR  $\text{current} == p$  or  $q \rightarrow$  current IS the LCA.

## ■ Visual

### LCA Visualization

```

10
 / \
5 15 LCA(3,7) = 5 (3<5, 7>5  $\rightarrow$  diverge at 5)
 / \ \ LCA(3,5) = 5 (5 is p itself  $\rightarrow$  LCA)
3 7 20 LCA(5,15) = 10 (5<10, 15>10  $\rightarrow$  diverge at 10)

At node 10: LCA(3,7)  $\rightarrow$  both < 10  $\rightarrow$  go LEFT
At node 5: LCA(3,7)  $\rightarrow$  3<5 and 7>5  $\rightarrow$  FORK! Return node 5
  
```

## ■ Java Implementation

### Java

```

class Solution {
    // Iterative (O(1) space) – preferred
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        TreeNode curr = root;

        while (curr != null) {
            if (p.val < curr.val && q.val < curr.val) {
  
```

## ■ Time & Space Complexity

| Case      | Time   | Space  | Why                               |
|-----------|--------|--------|-----------------------------------|
| Iterative | $O(h)$ | $O(1)$ | $h$ = tree height; no extra space |
| Recursive | $O(h)$ | $O(h)$ | Call stack depth = $h$            |

## Common Mistakes

## Watch Out For

- Using generic binary tree LCA algorithm — BST LCA is much faster, use BST property.
- Forgetting the case where p or q IS the ancestor of the other.
- Not handling equal values — if `curr.val == p.val` or `q.val`, it can be the LCA.

## ■ Practice Problems

| Case   | Time    | Space                       | Why                     |
|--------|---------|-----------------------------|-------------------------|
| Medium | LC 235  | LCA of BST                  | Direct application      |
| Medium | LC 236  | LCA of Binary Tree (no BST) | Generic version, $O(n)$ |
| Hard   | LC 1123 | LCA of Deepest Leaves       | LCA generalization      |

## ■ Mental Model

### Remember Forever

- "Walk down. The moment p and q stop agreeing on which direction to go — you've found their common ancestor."

## ■ Introduction

## ■ Approach 1 — HashSet + DFS

## ■ Approach 2 — Inorder + Two Pointers

## ■ Java Implementation

```

while (left < right) {
    int sum = sorted.get(left) + sorted.get(right);
    if (sum == k) return true;
    else if (sum < k) left++;
    else right--;
}
return false;
}

private void inorder(TreeNode node, List<Integer> list) {
    if (node == null) return;
    inorder(node.left, list);
    list.add(node.val);
    inorder(node.right, list);
}
}

```

## ■ Time & Space Complexity

| Case           | Time   | Space  | Why                                        |
|----------------|--------|--------|--------------------------------------------|
| HashSet DFS    | $O(n)$ | $O(n)$ | Visit all nodes, HashSet stores all values |
| Inorder + 2ptr | $O(n)$ | $O(n)$ | Inorder $O(n)$ , two pointer $O(n)$        |

## ■ Common Mistakes

### Avoid These

- Using the same node twice — "two distinct nodes" means different nodes (even if same value).
- Forgetting  $k - \text{root.val} == \text{root.val}$  case — node cannot pair with itself.
- Treating it like a sorted array Two Sum — BST is not a flat array, traverse fully.

## ■ Practice Problems

| Case   | Time    | Space                      | Why                     |
|--------|---------|----------------------------|-------------------------|
| Easy   | LC 653  | Two Sum IV<br>Input is BST | Direct application      |
| Medium | LC 1214 | Two Sum BSTs               | Two separate BSTs       |
| Hard   | LC 15   | 3Sum                       | Extend two pointer idea |

## ■ Mental Model

### Remember Forever

- "BST gives you sorted order for free via inorder. Any sorted-array algorithm (binary search, two pointers) can be applied after inorder traversal."
- HashSet approach is more general — works even on non-BST binary trees.

## ■ BST — Overall Quick Revision

- BST Property:  $\text{left} < \text{root} < \text{right}$  at EVERY subtree level
- Search/Insert/Delete: all  $O(\log n)$  average,  $O(n)$  worst (skewed)
- Validate: pass min/max bounds down; or check inorder is strictly increasing
- Inorder BST = sorted array — use for Kth smallest, Two Sum, range queries
- LCA in BST: walk down, divergence point is the answer —  $O(h)$  no extra space
- Delete 3 cases: leaf, one child, two children (inorder successor trick)
- Always use Long bounds in validation to handle Integer edge values