

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."