Documentation of Delete Node in a Binary Search Tree (BST)

Table of Contents

1. Problem Statement

Given a Binary Search Tree (BST) and a key, delete the node with the given key while maintaining the BST properties. Return the updated BST after deletion.

Example 1:

Input:root = [5,3,6,2,4,null,7], key = 3
Output:[5,4,6,2,null,null,7]

Explanation:

- The node with value 3 is deleted.
- It has two children (2 and 4).
- We replace 3 with its inorder successor (4) and delete 4 from the right subtree.

2. Intuition

Deleting a node in a BST involves three cases:

- Node has no children (leaf node) → Simply remove the node.
- Node has one child → Replace the node with its child.
- Node has two children → Replace the node with its inorder successor (smallest node in right subtree), then delete the successor.

The BST properties must be preserved after deletion.

3. Key Observations

- The inorder successor of a node is the smallest node in its right subtree.
- If a node has only one child, we can directly replace it with its child.
- If the node has no child, removing it does not affect the BST structure.

4. Approach

Step 1: Search for the Node

- Compare the key with the root value:
  - If key < root.val, move to the left subtree.
  - If key > root.val, move to the right subtree.
  - If key == root.val, we found the node to delete.

Step 2: Delete the Node

- Case 1: If the node has no children, return None.
- Case 2: If the node has one child, return that child.
- Case 3: If the node has two children, find the inorder successor, replace the node's value with it, and delete the successor.

Step 3: Return the Modified Tree

- The function should return the modified root node after deletion.

5. **Edge Cases**

- Tree is empty (root = None) → Return None.
- Key is not found in the tree → Return the original tree.
- Node to delete is a leaf node → Remove it directly.
- Node to delete has only one child → Replace it with its child.
- Node to delete has two children → Replace with inorder successor.

6. **Complexity Analysis**

Time Complexity

- Search for the node: O(height)
- Delete the node:
    - Finding the inorder successor takes O(height) in the worst case.
    - Overall, the worst case is O(height).
- Best case (Balanced BST): O(logN)
- Worst case (Skewed BST): O(N)

Space Complexity

- Recursive Approach: O(height) due to recursion stack.
- Iterative Approach: O(1) extra space.

7. **Alternative Approaches**

Iterative Approach

- Instead of recursion, use an iterative method to find and delete the node.
- Avoids recursion overhead but increases code complexity.

Using Parent Pointers

- Store parent pointers to traverse the tree efficiently.
- Useful when modifying the tree structure.

8. Test Cases

```
def test():
    solution = Solution()

    # Example 1
    root = TreeNode(5, TreeNode(3, TreeNode(2), TreeNode(4)), TreeNode(6, None, TreeNode(7)))
    assert solution.deleteNode(root, 3)  # Should return a BST without node 3

    # Example 2: Key not in tree
    root = TreeNode(5, TreeNode(3, TreeNode(2), TreeNode(4)), TreeNode(6, None, TreeNode(7)))
    assert solution.deleteNode(root, 0)  # Should return the same tree

    # Example 3: Empty tree
    assert solution.deleteNode(None, 0) == None

    # Example 4: Delete root node with two children
    root = TreeNode(10, TreeNode(5), TreeNode(15))
    assert solution.deleteNode(root, 10)  # Should replace 10 with successor

test()
```

9. Final Thoughts

- This approach efficiently deletes a node while preserving BST properties.
- Time complexity is optimal (O(height).
- Handles all edge cases (empty tree, key not found, different node cases).
- Alternative methods (iterative approach, parent pointers) could be used based on constraints.