450. Delete Node in a BST

Medium | Tree | Binary Search Tree | Binary Tree | Leetcode Link

Problem Description

This problem involves modifying a Binary Search Tree (BST) by deleting a specific node with a given key. The BST has a property where for every node, the left children are less than the current node, and the right children are greater. The problem can be broken into two main parts:

1. Locating the node that should be removed.
2. Once the node is found, executing the removal process.

The complication in deletion comes from ensuring that the BST property is maintained after the node is removed. This means correctly rearranging the remaining nodes so that the tree remains a valid BST.

Intuition

The solution follows the property of BST. If the key to be deleted is less than the root node's key, then it lies in the left subtree. If the key to be deleted is greater than the root's key, then it lies in the right subtree. If the key is equal to the root's key, then the root is the node to be deleted.

For the deletion, there are three scenarios:

1. Node with only one child or no child: If the node to be deleted has one or no children, we can simply replace the node with its child (if any) or set it to `None`.
2. Node with two children: Find the inorder successor (smallest in the right subtree or largest in the left subtree) of the node. Copy the inorder successor's content to the node and delete the inorder successor. This is because inorder successors are always a leaf or have a single child, making them easier to remove.
3. Leaf node: If the node is a leaf and needs to be deleted, we can simply remove the node from the tree.

The given solution first finds the node to delete, then based on its children, replaces the node accordingly. When a node with two children is deleted, instead of searching for the inorder predecessor, the algorithm finds the inorder successor, which is the smallest node in the right subtree, and swaps the values. Then the algorithm recursively deletes the successor node.

Solution Approach

The solution to the delete operation in a BST uses recursion to simplify the deletion process. Let's walk through the implementation approach:

1. **Base Case**: If the root is `None`, there is nothing to delete, so we return `None`.

2. **Searching Phase**: If the `key` is less than the root's value, we need to go to the left subtree: `root.left = self.deleteNode(root.left, key)` If the `key` is greater than the root's value, we need to go to the right subtree: `root.right = self.deleteNode(root.right, key)`

   Here, the algorithm is using recursion to traverse the tree in a BST property-aware manner (lesser values to the left, greater values to the right).

3. **Node Found**: Once we find the node with the value equal to the key (i.e., `root.val == key`), we need to delete this node while keeping the tree structure.

   a. **Node with One or No Child**: If the node to be deleted has either no children (`root.left` is `None` and `root.right` is `None`) or one child, it can be deleted by replacing it with its non-null child or with `None`.

   If the left child is `None`, it means the node either has a right child or no child at all. Hence, we return `root.right`.

   If the right child is `None`, it means the node has a left child only, so we return `root.left`.

   b. **Node with Two Children**: If the node to be deleted has two children, we need to find the inorder successor of the node, which is the smallest node in the right subtree. We do this by traversing to the leftmost child in the right subtree.

   Once the inorder successor is found, we replace the node's value with the successor's value. In the code, this is accomplished by swapping the root node with its right child first and then attaching the original left subtree to the new root.

   Finally, we delete the inorder successor node which has been moved to the root position by calling delete on the right subtree: `root.right = self.deleteNode(root.right, successor.val)` (not explicitly shown in the given reference code but implied by the structure of the recursion).

This intuitive approach of handling different cases separately and utilizing the BST property ensures that the updated tree maintains its BST properties after the deletion is performed.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Consider the following BST where we want to delete the node with key `7`:

```
      5
     / \
    3   8
       / \
      7   9
```

Following the solution approach:

1. **Base Case**: The root is not `None`, so we proceed with the operation.

2. **Searching Phase**: The key `7` is greater than the root's value `5`, so we examine the right subtree:

```
    8
   / \
  7   9
```

Since `7` is lesser than `8`, we now go to the left subtree of node `8`, where we find our node with the value `7`.

3. **Node Found**: As the node's key matches, we proceed with deletion.

a. **Node with One or No Child**: Node `7` is a leaf node in this scenario (no children). According to the third case, it can be simply removed from the tree. This means the left child of the node `8` will be set to `None`.

b. **Node with Two Children**: This does not apply here since our node `7` is a leaf.

Finally, the updated BST looks like this after deleting node `7`:

```
    5
   / \
  3   8
       \
        9
```

The BST properties are maintained, and `7` has been successfully removed.

Python Solution

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution:
    def deleteNode(self, root: Optional[TreeNode], key: int) -> Optional[TreeNode]:
        # If the root is None, then there is nothing to delete, return None
        if root is None:
            return None

        # If the key to be deleted is smaller than the root's key,
        # then it lies in the left subtree
        if key < root.val:
            root.left = self.deleteNode(root.left, key)
            return root

        # If the key to be deleted is greater than the root's key,
        # then it lies in the right subtree
        if key > root.val:
            root.right = self.deleteNode(root.right, key)
            return root

        # If the key is the same as root's key, then this is the node to be deleted

        # If the node has only one child or no child
        if root.left is None:
            return root.right

        if root.right is None:
            return root.left

        # If the node has two children, get the inorder successor
        # (smallest in the right subtree)
        min_right_subtree = root.right
        while min_right_subtree.left:
            min_right_subtree = min_right_subtree.left

        # Copy the inorder successor's content to this node
        root.val = min_right_subtree.val

        # Delete the inorder successor
        root.right = self.deleteNode(root.right, min_right_subtree.val)

        return root

```

Java Solution

```java
// Definition for a binary tree node.
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    // Constructor with no arguments
    TreeNode() {}

    // Constructor with value only
    TreeNode(int val) { this.val = val; }

    // Constructor with value, left, and right child nodes
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {

    // Function to delete a node with a given key from a binary search tree
    public TreeNode deleteNode(TreeNode root, int key) {
        // Base case: if the root is null, return null
        if (root == null) {
            return null;
        }

        // If the key is smaller than root's value, delete in the left subtree
        if (root.val > key) {
            root.left = deleteNode(root.left, key);
            return root;
        }

        // If the key is greater than root's value, delete in the right subtree
        if (root.val < key) {
            root.right = deleteNode(root.right, key);
            return root;
        }

        // If the root itself is the node to be deleted

        // If the root has no left child, return the right child directly
        if (root.left == null) {
            return root.right;
        }

        // If the root has no right child, return the left child directly
        if (root.right == null) {
            return root.left;
        }

        // If the root has both left and right children
        TreeNode successor = root.right;

        // Find the successor (smallest in the right subtree)
        while (successor.left != null) {
            successor = successor.left;
        }

        // Move the left subtree of the root to the left of the successor
        successor.left = root.left;

        // The new root should be the right child of the deleted node
        root = root.right;

        // Return the modified tree
        return root;
    }
}

```

C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    TreeNode* deleteNode(TreeNode* root, int key) {
        // If root is null, return immediately
        if (!root) {
            return root;
        }

        // If the key to be deleted is smaller than the root's value,
        // then it lies in left subtree
        if (root->val > key) {
            root->left = deleteNode(root->left, key);
            return root;
        }

        // If the key to be deleted is greater than the root's value,
        // then it lies in right subtree
        if (root->val < key) {
            root->right = deleteNode(root->right, key);
            return root;
        }

        // If key is the same as root's value, then this is the node
        // to be deleted

        // Node with only one child or no child
        if (!root->left) {
            return root->right;
        }
        if (!root->right) {
            return root->left;
        }

        // Node with two children: Get the inorder successor (smallest
        // in the right subtree)
        TreeNode* successorNode = root->right;
        while (successorNode->left) {
            successorNode = successorNode->left;
        }

        // Copy the inorder successor's content to this node
        root->val = successorNode->val;

        // Delete the inorder successor since its value is now copied
        root->right = deleteNode(root->right, successorNode->val);

        return root;
    }
};

```

Typescript Solution

```typescript
/**
 * Definition for a binary tree node.
 */
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

/**
 * Deletes a node with the specified key from the binary search tree.
 * @param {TreeNode | null} root - The root of the binary search tree.
 * @param {number} key - The value of the node to be deleted.
 * @returns {TreeNode | null} - The root of the binary search tree after deletion.
 */
function deleteNode(root: TreeNode | null, key: number): TreeNode | null {
  // If the root is null, return null (base case).
  if (root === null) {
      return root;
  }

  // Check if the key to delete is smaller or greater than the root's value to traverse the tree.
  if (root.val > key) {
      // The key to be deleted is in the left subtree.
      root.left = deleteNode(root.left, key);
  } else if (root.val < key) {
      // The key to be deleted is in the right subtree.
      root.right = deleteNode(root.right, key);
  } else {
      // When the node to be deleted is found
      if (root.left === null && root.right === null) {
          // Case 1: Node has no children (it is a leaf), simply remove it.
          root = null;
      } else if (root.left === null || root.right === null) {
          // Case 2: Node has one child, replace the node with its child.
          root = root.left || root.right;
      } else {
          // Case 3: Node has two children.
          if (root.right.left === null) {
              // If the immediate right child has no left child, promote the right child.
              root.right.left = root.left;
              root = root.right;
          } else {
              // If the right child has a left child, find the in-order predecessor.
              let minPredecessorNode = root.right;
              while (minPredecessorNode.left && minPredecessorNode.left.left !== null) {
                  minPredecessorNode = minPredecessorNode.left;
              }
              // Replace the root's value with the in-order predecessor's value.
              const minValue = minPredecessorNode.left.val;
              root.val = minValue;
              // Delete the in-order predecessor.
              minPredecessorNode.left = deleteNode(minPredecessorNode.left, minValue);
          }
      }
  }

  // Return the updated root.
  return root;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code for deleting a node from a BST (Binary Search Tree) depends on the height of the tree `h`.

1. Finding the node to delete takes `O(h)` time since in the worst case, we might have to traverse from the root to the leaf.
2. The deletion process itself is `O(1)` for nodes with either no child or a single child.
3. For nodes with two children, we find the minimum element in the right subtree, which also takes `O(h)` time in the worst case (when the tree is skewed).

Therefore, the overall time complexity is `O(h)`, which would be `O(log n)` for a balanced BST and `O(n)` for a skewed BST (where `n` is the number of nodes).

Space Complexity

The space complexity of the code is determined by the maximum amount of space used at any one time during the recursive calls (the call stack size).

1. The maximum depth of recursive calls is equal to the height `h` of the tree.
2. There are no additional data structures used that grow with the input size.

The space complexity is therefore `O(h)`, which corresponds to `O(log n)` for a balanced BST and `O(n)` for a skewed BST due to the recursive function calls.