510. Inorder Successor in BST II

Medium | Tree | Binary Search Tree | Binary Tree

Problem Description

The problem involves finding the in-order successor of a given node in a binary search tree (BST). In BSTs, each node can have up to two children, referred to as the left and right child. If a given node has a right child, the in-order successor is the node with the smallest key greater than that of the original node, which lies in the right subtree. If the node lacks a right child, the in-order successor is one of its ancestors. However, not every ancestor can be the successor; it is specifically the ancestor that the given node is in the left subtree of. This would be the first ancestor for which the given node lies in the left subtree when traveling up the tree. If no such ancestor exists, it means the node is the last node in of in-order traversal, and thus it has no successor, so we return `null`.

The structure of a BST and the property that values to the left are smaller and values to the right are larger is essential to solving this problem efficiently. The node class definition given provides not only left and right children but also a reference to the parent node, which allows traversal upwards in the tree.

Intuition

Facing this problem, we can think in two steps depending on whether the given node has a right subtree:

1. **Right Child Exists:** If the node has a right child, the in-order successor must be in that right subtree. To find it, we go to the right child and then keep moving to the leftmost child, because in a BST the leftmost child is the smallest node in a subtree.
2. **Right Child Does Not Exist:** If the node does not have a right child, we must look upward to the ancestors. We iterate up the parent chain until we find a node that is the left child of its parent. This parent will be the in-order successor since it's the first node greater than the given node when looking up the tree.

In both approaches, if we cannot find such a node by following the above logic, it means that the given node is the largest in the BST and thus has no in-order successor, so we return `null`.

The provided code implements these two intuitive steps cleanly, choosing the right strategy based on the existence of a right child and taking advantage of the BST properties and the parent reference to efficiently find the in-order successor.

Solution Approach

The implementation in the reference solution follows the logical steps outlined in the intuition:

1. **Right Child Exists:** If `node.right` is present, we start traversing the BST. The successor is the leftmost node in the right subtree. To find this, the code moves one step to the right `node = node.right` and then continues moving to the left `while node.left` until it finds the leftmost node.
   - The implementation applies a simple while loop to traverse to the leftmost node:
     ```
     while node.left:
         node = node.left
     ```
   - Once it reaches the leftmost node, that node is returned as the in-order successor.
2. **Right Child Does Not Exist:** When there is no right child, we must traverse up using the parent pointers. The code executes a while loop that continues as long as `node` is the right child of its parent:
   - The loop checks whether `node` is the right child via `node == node.parent.right`. If true, it moves up the tree: `node = node.parent`.
   - This loop stops when either `node` does not have a parent (meaning we have reached the root and no successor exists), or when `node` is a left child, which means its parent is the in-order successor. At this point, the node's parent (`node.parent`) is returned.
   - In code, it's expressed as:
     ```
     while node.parent and node == node.parent.right:
         node = node.parent
     return node.parent
     ```
     This part of the code returns either the parent that is the in-order successor or `null` if the given node has no in-order successor.

The solution uses a linked node structure inherent to the BST and does not employ any additional data structures. The parent pointer allows us to traverse upwards, which is essential for the second case where the node has no right child. By understanding the properties and traversal patterns of a BST, we can efficiently find the in-order successor with these two main moves: going to the leftmost node in the right subtree or ascending to the first parent for which the current node is in the left subtree.

Example Walkthrough

Let's consider a BST and find the in-order successor of a given node using the solution approach:

```
      20
     /  \
   10    30
   / \     \
  5  15    35
     /
   12
```

Case 1: Node has a right child.

Suppose we want to find the in-order successor of node with value `10`. Since this node has a right child (`15`), we look for the leftmost node in its right subtree:

- We go to the right child (`15`).
- Then we keep moving to its leftmost child, but `15` has a left child (`12`), which has no left child itself.
- The leftmost node in the right subtree of `10` is `12`, which is the in-order successor of `10`.

Case 2: Node has no right child.

Now, let's find the in-order successor of the node with value `15`. This node doesn't have a right child, so:

- We check if `15` is the left child of its parent (`10`), but it's not; it is the right child.
- We move up to the parent node (`10`) and check if `10` is a left child. Since `10` is not a child anymore (it's the root of the subtree we are considering in this example), we stop here.
- The in-order successor of `15` is the parent of `10`, which in this broader tree is `20`.

Case 3: Node with no successor.

If we wish to find the in-order successor of `35`, we notice it doesn't have a right child. Upon moving up, we find:

- `35` is the right child of its parent (`30`).
- We move up to the parent node (`30`), which is the right child of its parent (`20`).
- Since there's no ancestor where `30` (or `35`) is a left child, we realize that `35` has no in-order successor in this BST, and we return `null`.

Solution Implementation

Python

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.parent = None


class Solution:
    def inorder_successor(self, node: 'Node') -> 'Node':
        # If the given node has a right child, we need to find the left-most child of the right subtree.
        # This is the next node in an in-order traversal.
        if node.right:
            curr_node = node.right
            while curr_node.left:
                curr_node = curr_node.left
            return curr_node

        # If the given node has no right child, the successor is one of its ancestors.
        # Specifically, the successor is the ancestor of which the given node is in the left subtree.
        while node.parent and node == node.parent.right:
            node = node.parent

        # Once we exit the loop, the node's parent is the in-order successor.
        # If we exited because node.parent is None, then there is no in-order successor,
        # hence 'node.parent' will return None.
        return node.parent
```

Java

```java
class Solution {

    // Finds the inorder successor of a given node in a binary search tree
    public Node inorderSuccessor(Node node) {
        // If the right subtree of node is not null, find the leftmost node in the right subtree
        if (node.right != null) {
            node = node.right;
            // Keep moving to the left child until the leftmost child is reached
            while (node.left != null) {
                node = node.left;
            }
            // The leftmost node in the right subtree is the inorder successor
            return node;
        }

        // If the right subtree is null, the inorder successor is one of the ancestors.
        // Go up the tree until we find a node that is the left child of its parent
        while (node.parent != null && node == node.parent.right) {
            node = node.parent;
        }

        // The parent of the last node visited before the while loop terminates
        // is the inorder successor if it exists
        return node.parent;
    }
}
```

C++

```cpp
/*
// Definition for a binary tree node with a pointer to the parent.
class Node {
public:
    int val;          // The value contained in the node.
    Node* left;       // Pointer to the left child node.
    Node* right;      // Pointer to the right child node.
    Node* parent;     // Pointer to the parent node.
};
*/

class Solution {
public:
    // Function to find the inorder successor of a given node in a BST.
    Node* inorderSuccessor(Node* node) {
        if (node->right) {
            // If there is a right child, the successor is the leftmost node of
            // the right subtree.
            node = node->right;
            while (node->left) {
                node = node->left; // Find the leftmost node.
            }
            return node;
        }
        // If there is no right child, the successor is one of the ancestors.
        // Travel up using the parent pointers until we find a node which is
        // the left child of its parent. The parent of such a node is the successor.
        while (node->parent && node == node->parent->right) {
            node = node->parent; // Go upwards until the node is a left child.
        }
        // The parent of the node found is the successor, or nullptr if not found.
        return node->parent;
    }
};
```

TypeScript

```typescript
// Node definition with TypeScript types
class Node {
    val: number;
    left: Node | null;
    right: Node | null;
    parent: Node | null;

    constructor(val: number) {
        this.val = val;
        this.left = null;
        this.right = null;
        this.parent = null;
    }
}

/**
 * Find the in-order successor of a given node in a Binary Search Tree (BST).
 *
 * @param {Node} node - The starting node to find the in-order successor for.
 * @return {Node | null} - The in-order successor node if it exists, otherwise null.
 */
function inorderSuccessor(node: Node): Node | null {
    if (node.right) {
        // Successor is the leftmost child of node's right subtree
        let currentNode: Node = node.right;
        while (currentNode.left) {
            currentNode = currentNode.left;
        }
        return currentNode;
    }
    // Traverse up the tree and find the node which is the left child of its parent,
    // the parent will be the successor
    let currentNode: Node | null = node;
    while (currentNode.parent && currentNode === currentNode.parent.right) {
        currentNode = currentNode.parent;
    }
    // The successor is the parent of the detached subtree
    return currentNode.parent;
}
```

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.parent = None


class Solution:
    def inorder_successor(self, node: 'Node') -> 'Node':
        # If the given node has a right child, we need to find the left-most child of the right subtree.
        # This is the next node in an in-order traversal.
        if node.right:
            curr_node = node.right
            while curr_node.left:
                curr_node = curr_node.left
            return curr_node

        # If the given node has no right child, the successor is one of its ancestors.
        # Specifically, the successor is the ancestor of which the given node is in the left subtree.
        while node.parent and node == node.parent.right:
            node = node.parent

        # Once we exit the loop, the node's parent is the in-order successor.
        # If we exited because node.parent is None, then there is no in-order successor,
        # hence 'node.parent' will return None.
        return node.parent
```

Time and Space Complexity

The given code finds the in-order successor of a given node in a binary tree with parent pointers. The time complexity and space complexity are analyzed below:

Time Complexity

The time taken by the code depends on two scenarios:

1. **Right child exists**: If the node has a right child, the algorithm finds the leftmost child in the right subtree which is the in-order successor. The complexity in this case is O(h), where `h` is the height of the subtree, potentially O(n) in the worst case if the tree is skewed.
2. **Right child does not exist**: If the node does not have a right child, the algorithm travels up the tree to find the first ancestor node of which the given node is in the left subtree. The complexity can be O(h) as well, which again, could be O(n) in a skewed tree structure where `n` is the number of nodes in the tree.

Hence, the worst-case time complexity is `O(n)`.

Space Complexity

The space complexity of the algorithm is determined by the space used to store the information needed for iteration. Since the code uses only a few pointers and no recursive calls or additional data structures, the space complexity is `O(1)`, meaning it requires constant extra space.