285. Inorder Successor in BST

Medium Tree Depth-First Search Binary Search Tree Binary Tree

Problem Description

The problem is set within the context of a binary search tree (BST), a type of binary tree where each node has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Given such a tree and a particular node `p` in it, you need to find the "in-order successor" of that node. The in-order successor for a given node `p` in a BST is defined as the node with the smallest key that is greater than `p.val`. If such a node doesn't exist, in other words, if `p` is the node with the highest value in the tree, the function should return `null`.

The in-order traversal of a BST produces the node values in an ascending order. So the task could also be seen as finding the next node in the in-order traversal sequence.

Intuition

When trying to understand the solution, it's critical to grasp what it means by in-order traversal and how the properties of a binary search tree can streamline our search for the successor node.

The in-order traversal for a BST involves visiting the left subtree, then the node itself, and then the right subtree. When looking for the successor of node `p`, if `p` has a right subtree, then the successor would be the leftmost node in that right subtree. If `p` does not have a right subtree, the successor would be one of its ancestors, specifically the one for whom `p` is situated in the left subtree.

The proposed solution uses this understanding and traverses the tree starting from the root. At each step, it compares the current node's value to `p.val` to decide the direction of the search:

- If the current node's value is greater than `p.val`, the current node is a potential successor, and we go left to find a smaller one that is still larger than `p.val`.
- If the current node's value is less than or equal to `p.val`, the current node can't be the successor, and we go right to find a larger one.

The variable `ans` is used to keep track of the latest potential successor. This works because of the BST property: going left finds smaller keys and going right finds larger keys. When the algorithm finishes traversing the tree (`root` becomes `None`), the `ans` variable either contains the in-order successor node or remains `None`, meaning there is no successor (if `p` is the maximum element).

This algorithm ensures that we only travel down the tree once, providing an efficient solution with O(h) time complexity, where h is the height of the tree.

Solution Approach

The implementation of the solution follows a straightforward approach utilizing the binary search tree properties to efficiently find the in-order successor.

Firstly, let's outline the core components of the algorithm:

- **While Loop**: The algorithm uses a loop to iterate through the nodes of the tree, starting at the `root`. The loop continues until `root` becomes `None`, which means we have reached the end of our search path in the BST.
- **Conditional Checking**: Inside the loop, a conditional check compares the current node's value (`root.val`) with the value of the node `p` (`p.val`). This comparison dictates the traversal direction because of the binary search tree arrangement.
- **Potential Successor Update (`ans`)**: If the current node's value is greater than `p.val`, it means this node could potentially be the successor. We store this node to `ans` and continue searching to the left for a smaller value that would still be larger than `p.val`.
- **Traversal Direction**:
  - If `root.val > p.val`, we move left (`root = root.left`) in search of a smaller yet greater value than `p.val`.
  - If `root.val <= p.val`, we move right (`root = root.right`) since we know that all values in the left subtree are smaller and cannot be the successor.

The algorithm makes use of the BST property that left descendants are smaller and right descendants are larger than the node itself. Hence, we can discard half of the tree in each step of the while loop, similar to a binary search.

There is no need to keep track of already visited nodes because the potential successor (`ans`) is updated only when moving left, ensuring that we always have the closest higher value to `p.val`.

Additionally, no extra data structures are required, and the solution makes in-place updates without modifying the tree, hence the space complexity is O(1).

The algorithm terminates when we can no longer traverse the tree (when `root` is `None`), which means we can return the `ans` value as the in-order successor. If no such successor exists (i.e., `p` is the maximum value in the BST), the `ans` remains `None`, which the function returns as specified.

Here's a closer look at the code snippet which encapsulates the algorithm:

```
class Solution:
    def inorderSuccessor(self, root: TreeNode, p: TreeNode) -> Optional[TreeNode]:
        ans = None
        while root:
            if root.val > p.val:
                ans = root
                root = root.left
            else:
                root = root.right
        return ans
```

The single pass through the tree and constant time operations on each node's value guarantee the solution is time-efficient, running in O(h) time where h is the height of the tree. This effectively means that in the worst case scenario (a skewed tree), where the tree shape resembles a linked list, the algorithm could take O(n) time where n is the number of nodes, since the tree's height would be equivalent to the number of nodes.

Example Walkthrough

Let's illustrate the solution approach using a simple binary search tree and following the steps to find the in-order successor of a given node `p`. Consider the following BST:

```
    4
   / \
  2   6
 / \ / \
1  3 5  7
```

Let's find the in-order successor of the node `p` with a value of `3`.

1. We start at the root of the tree, which is the node with the value `4`.
2. We compare `p.val` (which is `3`) with `root.val` (which is `4`).
   - Since `4` is greater than `3`, the node with `4` could be the in-order successor. We save this node and move to the left to look for an even closer value greater than `3`.
3. Now, `root` is at the node with the value `2`.
4. We compare `p.val` with the new `root.val` (`2`).
   - Since `2` is less than `3`, we discard the left subtree including the current node and move right to find a larger value.
5. However, the right child of `2` is the node `p` itself. So, we move to the right subtree of `p`.
6. Because `p` doesn't have a right subtree, our loop terminates, and we return the stored `ans`, which is `4`.

In this example, the in-order successor of node `p` with value `3` is the node with value `4`.

Here's a breakdown of how the code snippet implements this logic:

```
class Solution:
    def inorderSuccessor(self, root: TreeNode, p: TreeNode) -> Optional[TreeNode]:
        ans = None
        while root:
            if root.val > p.val:
                ans = root  # Potentially a successor, so we update 'ans'.
                root = root.left  # Search for an even closer value on the left.
            else:
                root = root.right  # Current or left values are not greater, we go right.
        return ans
```

This walk-through demonstrates the algorithm's ability to leverage the BST properties to efficiently find the in-order successor with minimal time complexity.

Python Solution

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, value):
        self.val = value
        self.left = None
        self.right = None

class Solution:
    def inorderSuccessor(self, root: TreeNode, p: TreeNode) -> TreeNode:
        # Initialize variable to store the inorder successor
        successor = None

        # Traverse the tree starting with the root
        while root:
            # If current node's value is greater than 'p's value,
            # tentative successor is found (potentially there could be a closer one).
            if root.val > p.val:
                successor = root
                # Move to the left subtree to find the closest ancestor
                root = root.left
            else:
                # If current node's value is less than or equal to 'p's value,
                # the successor must be in the right subtree.
                root = root.right

        # Return the successor node
        return successor
```

Java Solution

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

    TreeNode(int x) {
        val = x;
    }
}

class Solution {

    /**
     * Finds the inorder successor of a given node in a BST.
     * The inorder successor of a node is the node with the smallest key greater than the current node's key.
     *
     * @param root the root of the BST
     * @param p the target node for which we need to find the inorder successor
     * @return the inorder successor node if exists, otherwise null
     */
    public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
        TreeNode successor = null; // This will hold the successor as we traverse the tree

        while (root != null) {
            if (root.val > p.val) {
                // If current node's value is greater than p's value,
                // go left to find a smaller value, but closer to p's value than the current value.
                successor = root; // The potential successor is updated
                root = root.left;
            } else {
                // If current node's value is less than or equal to p's value,
                // successor must be in the right subtree.
                root = root.right;
            }
        }

        return successor; // Return the successor found during traversal
    }
}
```

C++ Solution

```
/**
 * The function `inorderSuccessor` finds the in-order successor of a given node `p`
 * in a binary search tree. The in-order successor of a node is defined as the
 * node with the smallest key greater than `p->val`. The BST property is utilized here,
 * which implies that the in-order successor of `p` is either in `p`'s right subtree
 * or it is an ancestor node. If neither of these is true, then `p` has no in-order successor.
 *
 * @param root: A pointer to the root node of the binary search tree.
 * @param p: A pointer to the node in the binary search tree whose successor is to be found.
 * @return A pointer to the in-order successor node of `p` if it exists, or `nullptr` if `p` has no in-order successor in the tree.
 */
class Solution {
public:
    TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
        TreeNode* successor = nullptr;

        while (root != nullptr) {
            // If the current node's value is greater than that of p, then the successor must be in the left subtree
            // (or the current node itself could be a potential successor).
            if (root->val > p->val) {
                successor = root; // Update potential successor
                root = root->left; // Move left to find smaller value
            } else {
                // If the current node's value is not greater than that of p, move to the right subtree to find a greater value.
                // Note: This means 'root' cannot be the successor, as its value is not greater than the 'p' value
                root = root->right;
            }
        }
        // Return the last recorded potential successor which is the actual in-order successor if it exists.
        return successor;
    }
};
```

Typescript Solution

```
/**
 * Finds the in-order successor of a given node in a binary search tree (BST).
 * In a BST, the in-order successor of a node is the node with the smallest key greater than the input node's key.
 *
 * @param {TreeNode | null} root - The root node of the BST.
 * @param {TreeNode | null} targetNode - The node whose in-order successor is to be found.
 * @return {TreeNode | null} The in-order successor node if it exists, otherwise null.
 */
function inorderSuccessor(root: TreeNode | null, targetNode: TreeNode | null): TreeNode | null {
    // Variable to hold the in-order successor as we traverse the tree.
    let successor: TreeNode | null = null;

    // Loop through the tree starting from the root.
    while (root !== null) {
        // If the current node's value is greater than the targetNode's value,
        // it could be the successor, so we also move to the left child,
        // because if there is a smaller value than the current node's value that
        // is still larger than the targetNode's value, it would be in the left subtree.
        if (root.val > targetNode.val) {
            successor = root;
            root = root.left;
        } else {
            // If current node's value is less than or equal to the targetNode's value,
            // we move to the right child to look for a larger value.
            root = root.right;
        }
    }

    // Return the found successor, which might be null if there's no successor in the tree.
    return successor;
}
```

Time and Space Complexity

The time complexity of the given code is `O(h)` where `h` is the height of the binary search tree. This is because in the worst case, the while loop will traverse from root to the leaf, following one branch and visiting each level of the tree once.

The space complexity of the given code is `O(1)` because it does not use any additional space that is dependent on the size of the input tree. It only uses a fixed number of pointers regardless of the number of nodes in the tree.