98. Validate Binary Search Tree Tree **Binary Tree** Medium **Depth-First Search Binary Search Tree**

Problem Description

The problem asks to verify whether a given binary tree is a valid binary search tree (BST). By definition, a valid BST is characterized by the following properties:

- Every node's left subtree contains only nodes with keys that are less than the node's own key. Every node's right subtree contains only nodes with keys that are greater than the node's own key.
- Both the left and right subtrees must also themselves be valid BSTs.
- The objective is to use these criteria to check every node in the tree and ensure that the structure adheres to the rules of a BST.
- Intuition

The solution uses the concept of in-order traversal. In a BST, an in-order traversal yields the nodes' values in ascending order. The

BST.

approach involves a Depth-First Search (DFS) where we traverse the tree in an in-order manner (left, node, right), keeping track of the value of the previously visited node (prev). During the traversal, we ensure that each subsequent node has a greater value than the previous node.

2. If we encounter a None (indicative of reaching the end of a path), we return True, because an empty tree is technically a valid

3. As we traverse the left subtree, we check for violations of the BST properties.

1. A recursive function dfs is defined, which will do an in-order traversal of the tree.

- 4. Before visiting the current node, we ensure that we've finished validating the left subtree. If the left subtree is invalid, the function returns False. 5. We then check the current node's value against prev (the previous node's value, initialized to negative infinity). The value must
- be strictly greater to satisfy BST conditions. 6. Update prev to the current node's value.
- 8. If every recursive call returns True, the entire tree is a valid BST, and thus the function returns True. This approach ensures that we confirm the BST property where each node's value must be greater than all values in its left subtree
- and less than all values in its right subtree.
- Solution Approach
- The provided Python code implements the DFS in-order traversal to check the validity of the BST, and it uses the TreeNode data structure, which is a standard representation of a node in a binary tree, consisting of a value val and pointers to left and right child nodes.

Here is the step-by-step breakdown of the solution approach:

1. In-order traversal (DFS): We recursively traverse the binary tree using in-order traversal, where we first visit the left subtree,

2. Global variable: A global variable prev (initialized to negative infinity) is used to keep track of the value of the last visited node in the traversal.

the right subtree.

subtree is valid by definition.

Example Walkthrough

7. Proceed to validate the right subtree.

then the node, and finally the right subtree.

3. Checking BST properties:

 After checking the left subtree, we examine the current node by comparing its value with prev. If the current node's value is less than or equal to prev, this means the in-order traversal sequence is broken, and hence, the tree is not a valid BST. We return False in this case.

contains a violation of the BST rules, and thus, we return False immediately to stop checking further.

When visiting a node, we first call the dfs function on its left child. If this function call returns False, it means the left subtree

The prev variable is then updated to the current node's value, indicating that this node has been processed, and we move to

5. Final validation: After the entire tree has been traversed, if no violations were found, the initial call to dfs(root) will ultimately return True, confirming the tree is a valid BST.

The key algorithmic pattern used here is recursion, combined with the properties of in-order traversal for a BST. The recursive

function dfs combines the logic for traversal and validation, effectively checking the BST properties as it moves through the tree.

4. Recursive base case: For a None node, which signifies the end of a branch, the dfs function returns True, as an empty tree or

Let's consider a small binary tree with the following structure to illustrate the solution approach:

The in-order traversal dictates that we visit the left subtree first, then the root node, and finally the right subtree. 2. Step 2 - Global variable: Initially, the prev variable is set to negative infinity. It will help us to keep track of the last visited node's value.

1. Step 1 - In-order traversal (DFS): We begin the depth-first search (DFS) with an in-order traversal from the root node which is 2.

We start with the left child (node 1). Since node 1 has no children, we compare it to prev, which is negative infinity. Node 1 is

o In the right subtree, we have node 3. We call the dfs function on node 3. As it has no children, we compare it to prev (which

3. Step 3 - Checking BST properties:

o Now, we are at node 2 and compare its value with prev which is now 1. The value of node 2 is greater; therefore, it satisfies the BST condition. We update prev to 2 and proceed to the right subtree.

False, the whole tree is thus confirmed to be a valid BST. The final call to dfs(root) returns True.

In summary, the example binary tree with nodes 1, 2, and 3 is indeed a valid binary search tree.

If the current node is None, it's a leaf, so return True.

* Validates if the given binary tree is a valid binary search tree (BST).

previousValue = null; // Initialize previousValue as null before starting traversal

* Performs an in-order depth-first traversal on the binary tree to validate BST property.

* @return true if the subtree rooted at 'node' satisfies BST properties, false otherwise.

// Traverse the left subtree. If it's not a valid BST, return false immediately.

previousValue = node.val; // Update previousValue with the current node's value.

// Traverse the right subtree. If it's not a valid BST, return false immediately.

* It ensures that every node's value is greater than the values of all nodes in its left subtree

* @param root The root of the binary tree to check.

private boolean inOrderTraversal(TreeNode node) {

if (!inOrderTraversal(node.left)) {

if (!inOrderTraversal(node.right)) {

return true; // All checks passed, it's a valid BST.

// Constructor to initialize a node with a given value,

// with left and right pointers set to nullptr by default

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

// Constructor to create a node with given value, left, and right children

TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}

public boolean isValidBST(TreeNode root) {

return inOrderTraversal(root);

if (node == null) {

return false;

* Definition for a binary tree node.

* @return true if the given tree is a BST, false otherwise.

* and less than the values of all nodes in its right subtree.

* @param node The current node being visited in the traversal.

return true; // Base case: An empty tree is a valid BST.

greater, so we update prev to 1, and then we return back to node 2.

is now 2). Node 3 is greater than 2, so we update prev to 3.

given tree meets all the properties of a valid BST:

def __init__(self, val=0, left=None, right=None):

Our target is to walk through the solution approach provided above and confirm whether this tree is a valid BST.

- 4. Step 4 Recursive base case: Since we reached the leaf nodes without encountering any None nodes, we confirm that all subtrees are valid BSTs as well. 5. Step 5 - Final validation: After visiting all the nodes following the in-order traversal, and none of the recursive dfs calls returned
- Each node's left subtree contains only nodes with values less than the node's own value. Each node's right subtree contains only nodes with values greater than the node's own value. · All the subtrees are valid BSTs on their own.

By following the in-order traversal method and checking each node against the previous node's value, we have verified that the

1 class TreeNode: # A binary tree node has a value, and references to left and right child nodes.

class Solution: def isValidBST(self, root: TreeNode) -> bool: # Helper function to perform an in-order traversal of the tree. def in_order_traversal(node):

Check if the previous value in the in-order traversal is greater than or equal to the current node value,

nonlocal last_value_visited # To keep track of the last value visited in in-order traversal.

```
26
27
               # Update the last value visited with the current node's value.
28
                last_value_visited = node.val
29
```

12

13

18

19

20

21

24

25

30

31

32

33

Python Solution

self.val = val

self.left = left

self.right = right

if node is None:

return True

return False

return False

return False

Traverse the left subtree.

if not in_order_traversal(node.left):

which would violate the BST property.

if not in_order_traversal(node.right):

if last_value_visited >= node.val:

Traverse the right subtree.

```
34
               # No violations found, return True.
35
               return True
36
37
           # We initialize the last visited value with negative infinity (smallest possible value)
38
           # to make sure the first comparison is always True.
39
           last_value_visited = float('-inf')
40
           # Start the in-order traversal of the tree with the root node.
41
42
           return in_order_traversal(root)
43
Java Solution
    * Definition for a binary tree node.
     * This class represents a node of a binary tree which includes value, pointer to left child
      * and pointer to right child.
  6 class TreeNode {
         int val; // node's value
         TreeNode left; // pointer to left child
  8
         TreeNode right; // pointer to right child
  9
 10
 11
         // Default constructor.
         TreeNode() {}
 12
 13
 14
         // Constructor with node value.
 15
         TreeNode(int val) {
 16
             this.val = val;
 17
 18
 19
         // Constructor with node value, left child, and right child.
 20
         TreeNode(int val, TreeNode left, TreeNode right) {
             this.val = val;
 21
 22
             this.left = left;
 23
             this.right = right;
 24
 25 }
 26
 27 /**
     * This class contains method to validate if a binary tree is a binary search tree (BST).
 29
     */
    class Solution {
         private Integer previousValue; // variable to store the previously visited node's value
 31
 32
```

58 return false; 59 60 // Check the current node value with the previous node's value. // As it's an in-order traversal, previousValue should be less than the current node's value. 61 if (previousValue != null && previousValue >= node.val) { 62 return false; // The BST property is violated. 63

C++ Solution

/**

*/

9

10

11

12

13

14

16

15 };

1 #include <climits>

struct TreeNode {

int val;

17 class Solution {

TreeNode *left;

TreeNode *right;

33

34

35

36

37

38

39

40

41

42

43

44

45

47

48

49

50

51

52

53

54

55

56

57

64

65

66

67

68

69

70

71

73

72 }

/**

42 43 44 45

15

16

17

18

21

22

23

24

25 26

27

30

32

33

34

35

36

37

39

42

50 //

55

53 // };

};

/**

*/

```
private:
 18
        // Pointer to store the last visited node in the binary search tree
 19
 20
        // to compare its value with the current node's value.
 21
         TreeNode* lastVisited;
 22
 23
         // Helper function to perform an in-order traversal of the tree
 24
         // and check if it is a valid binary search tree.
 25
         bool inorderTraversal(TreeNode* node) {
 26
             // Base case: If the current node is null, then it's valid
 27
             if (!node) return true;
 28
 29
             // Recursively traverse the left subtree.
 30
             // If the left subtree is not a valid BST, the entire tree is not.
             if (!inorderTraversal(node->left)) return false;
 31
 32
 33
             // If the last visited node is not null and the value of the last visited node
 34
             // is greater than or equal to the current node's value, then it's not a valid BST.
 35
             if (lastVisited && lastVisited->val >= node->val) return false;
 36
 37
             // Update the last visited node to the current node
 38
             lastVisited = node;
 39
 40
             // Recursively traverse the right subtree.
 41
             // If the right subtree is not a valid BST, the entire tree is not.
             if (!inorderTraversal(node->right)) return false;
             // If both subtrees are valid, return true
             return true;
 46
 47
    public:
         // Function to check whether a binary tree is a valid binary search tree.
 50
         bool isValidBST(TreeNode* root) {
 51
             // Initialize the last visited node as null before starting the traversal
 52
             lastVisited = nullptr;
 53
 54
             // Call the helper function to check if the tree is a valid BST
 55
             return inorderTraversal(root);
 56
 57 };
 58
Typescript Solution
1 // Define the structure of a TreeNode with TypeScript interface
   interface TreeNode {
     val: number;
     left: TreeNode | null;
     right: TreeNode | null;
6
   /**
    * Validates if the provided tree is a binary search tree.
    * @param {TreeNode | null} root - The root node of the binary tree to validate.
    * @return {boolean} - True if the tree is a valid BST; otherwise, false.
12
    */
```

43 // Start the recursive in-order traversal from the root return inorderTraversal(root); 46 }; 47 // Sample usage (in TypeScript you would normally type the input, here it is implicit for brevity)

// let tree: TreeNode = {

val: 2,

const isValidBST = (root: TreeNode | null): boolean => {

let previous: TreeNode | null = null;

if (node === null) {

// Traverse the left subtree

// Traverse the right subtree

return inorderTraversal(node.right);

52 // right: { val: 3, left: null, right: null }

<u>Time and Space Complexity</u>

if (!inorderTraversal(node.left)) {

if (previous && node.val <= previous.val) {</pre>

// Update the previous node to the current node

left: { val: 1, left: null, right: null },

// console.log(isValidBST(tree)); // Outputs: true

return true;

return false;

return false;

previous = node;

// Define the previous node to keep track of during in-order traversal

* @param {TreeNode | null} node - The current node in the traversal.

const inorderTraversal = (node: TreeNode | null): boolean => {

* @return {boolean} - True if the subtree is valid; otherwise, false.

* Performs in-order traversal to check each node's value strictly increasing.

// Check if the current node's value is greater than the previous node's value

The given Python code defines a method isValidBST to determine if a binary tree is a valid binary search tree. It uses depth-first search (DFS) to traverse the tree. **Time Complexity:**

The time complexity of this algorithm is O(n), where n is the number of nodes in the binary tree. This is because the DFS visits each node exactly once to compare its value with the previously visited node's value.

Space Complexity:

The space complexity of this code is O(h), where h is the height of the tree. This is determined by the maximum number of function calls on the call stack at any one time, which occurs when the algorithm is traversing down one side of the tree. In the worst case of

a completely unbalanced tree (like a linked list), the space complexity would be O(n). For a balanced tree, it would be O(log n).