Function to Check if a Binary Tree is a Valid BST

Mr. Akash Yadav
Assistant Professor
Artificial Intelligence & Data Science

Lesson Plan

Subject/Course	**Competitive Coding**
Lesson Title	**Function to Check if a Binary Tree is a Valid BST**

Lesson Objectives
Understand the concept and properties of a Binary Search Tree (BST).
Learn how to validate whether a given binary tree is a valid BST.
Implement BST validation using recursion and range checking.
Analyze the time and space complexity of the BST validation algorithm.

Problem Statement

Write a program to check whether a given Binary Tree is a Valid Binary Search Tree (BST) or not.

Conditions:

1. Each node’s left subtree must contain values less than the node’s key.
2. Each node’s right subtree must contain values greater than the node’s key.
3. Both left and right subtrees must themselves be valid BSTs.

Concept

A Binary Search Tree (BST) is a special kind of binary tree that maintains sorted order.

1. Property: left < root < right
2. The inorder traversal of a BST produces a sorted sequence.
3. To verify a binary tree, this property must hold recursively for every node.

Algorithm/Logic

Approach 1 – Recursive Range Check:

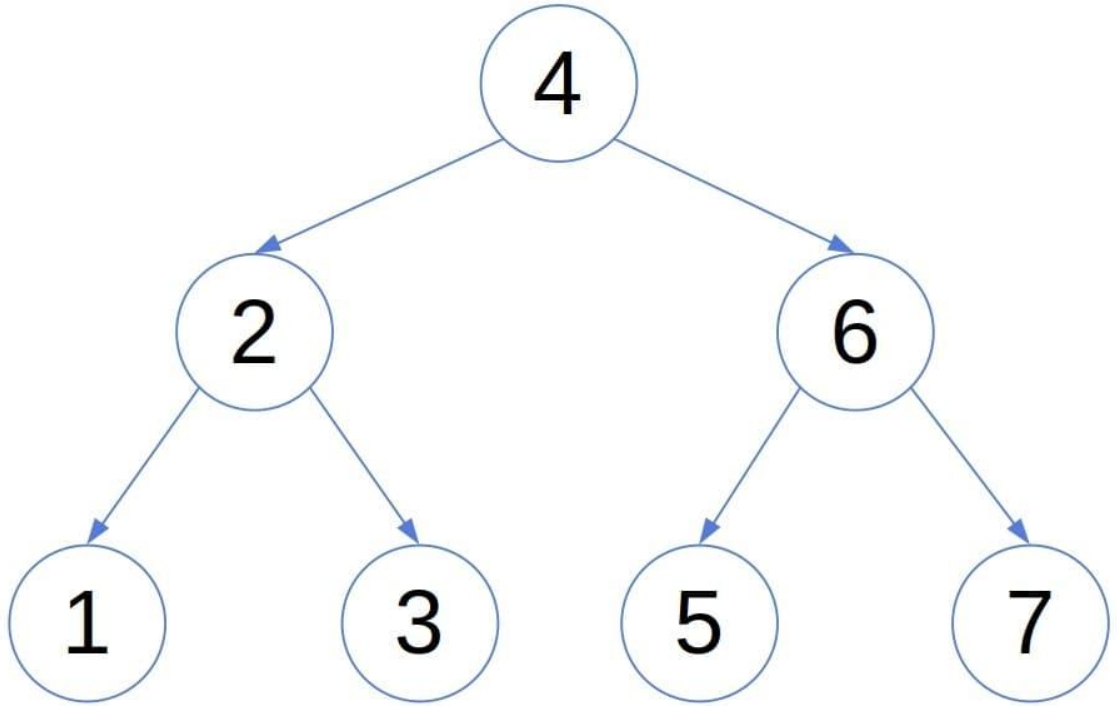
1. Start from the root node.
2. Define valid range (min, max) for each node.
3. Check if node value lies within range.
4. Recursively verify both subtrees.
5. Left subtree uses range (min, node.data)
6. Right subtree uses range (node.data, max)
7. If all nodes satisfy the condition, the tree is a valid BST.

Algorithm/Logic

Approach 2 – Inorder Traversal Check:

1. Perform inorder traversal and track previous node.
2. If current node ≤ previous node → Not a BST.
3. If traversal is sorted → Valid BST.
4. This method also runs in O(n) time.

Visualization



Inorder Traversal: 1,2,3,4,5,6,7
Result: Valid BST

Code Implementation

```
class Node {
    int data;
    Node left, right;

    Node(int val) {
        data = val;
        left = right = null;
    }
}
class BSTValidator {
    boolean isBST(Node root) {
        return isBSTUtil(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }
    boolean isBSTUtil(Node node, long min, long max) {
        if (node == null)
            return true;

        if (node.data <= min || node.data >= max)
            return false;

        return isBSTUtil(node.left, min, node.data)
            && isBSTUtil(node.right, node.data, max);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Node root = new Node(10);
        root.left = new Node(5);
        root.right = new Node(15);
        root.right.left = new Node(12);
        root.right.right = new Node(18);

        BSTValidator bst = new BSTValidator();

        if (bst.isBST(root))
            System.out.println("The given tree is a valid BST.");
        else
            System.out.println("The given tree is NOT a valid BST.");
    }
}
```

Output

```
The given tree is a valid BST.
```

Example Walkthrough

1. Root (10): range $(-\infty, +\infty)$ → OK
2. Left (5): range $(-\infty, 10)$ → OK
3. Right (15): range $(10, +\infty)$ → OK
4. All nodes satisfy range → Valid BST.

Time & Space Complexity

Time Complexity: O(n)

Each node is visited exactly once.

Space Complexity: O(h)

Recursion stack depends on tree height (h).

Summary

1. Verifies BST property using range limits.
2. Inorder traversal is always sorted for BST.
3. Efficient: O(n) time, O(h) space.
4. Helps in tree validation and search problems.

Practice Questions:

1. Validate Binary Search Tree – LeetCode #98

🔗 Link: https://leetcode.com/problems/validate-binary-search-tree/

Concept:
Use **inorder traversal** or **recursive range checking** to ensure that every node follows the rule:
left < root < right.

Why Practice:
This is the **exact same concept** as your Practical 10 — validating BST structure and property correctness.
Reinforces recursion, boundary conditions, and tree traversal understanding.

Practice Questions:

2. Recover Binary Search Tree – LeetCode #99

🔗 https://leetcode.com/problems/recover-binary-search-tree/

Concept:
Two nodes in a Binary Search Tree are swapped by mistake.
Use **inorder traversal** to detect and restore the correct BST order by identifying the misplaced nodes.

Why Practice:
This question builds on the **BST validation logic** — understanding how and why inorder traversal works for BSTs.
It helps strengthen your grasp of **BST properties**, **recursive traversal**, and **error correction** in trees.

Practice Questions:

3. Convert Sorted Array to Binary Search Tree — LeetCode #108

🔗
https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/

Concept:
Build a **balanced BST** from a sorted array by recursively selecting the middle element as the root.

Why Practice:
Helps understand **how valid BSTs are structured** — complements your validation logic by learning how BSTs are built.

Thanks