Previous Next

98. Validate Binary Search Tree

Dec. 12, 2018 | 316.7K views

Average Rating: 4.17 (148 votes)

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

- 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.

Example 1:

```
    2
   / \
  1   3

Input: [2,1,3]
Output: true
```

Example 2:

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

Input: [5,1,4,null,null,3,6]
Output: false
Explanation: The root node's value is 5 but its right child's value is 4.
```

Solution

Tree definition

First of all, here is the definition of the `TreeNode` which we would use.

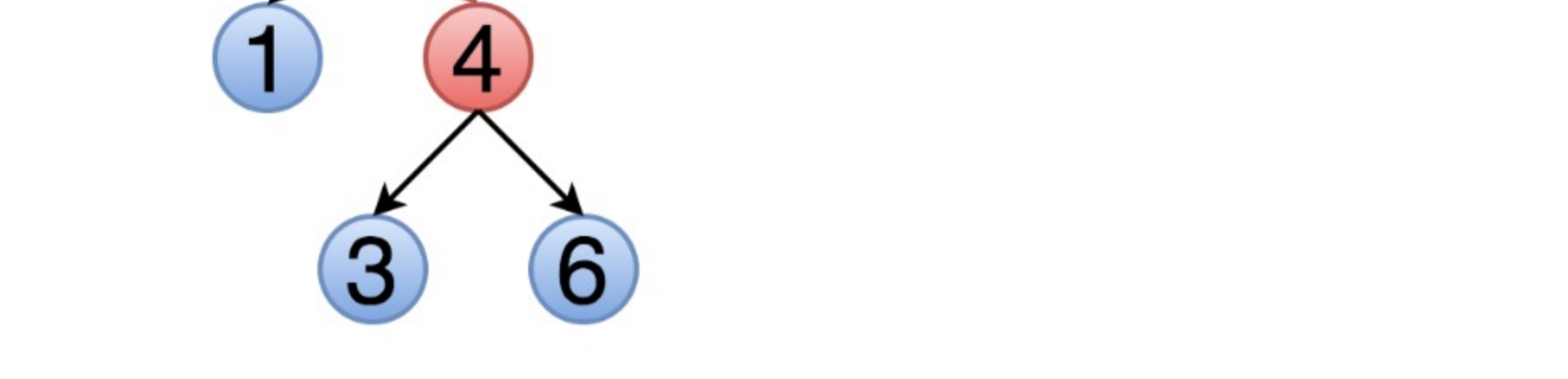
Java Python Copy

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

Intuition

On the first sight, the problem is trivial. Let's traverse the tree and check at each step if `node.right.val > node.val` and `node.left.val < node.val`. This approach would even work for some trees

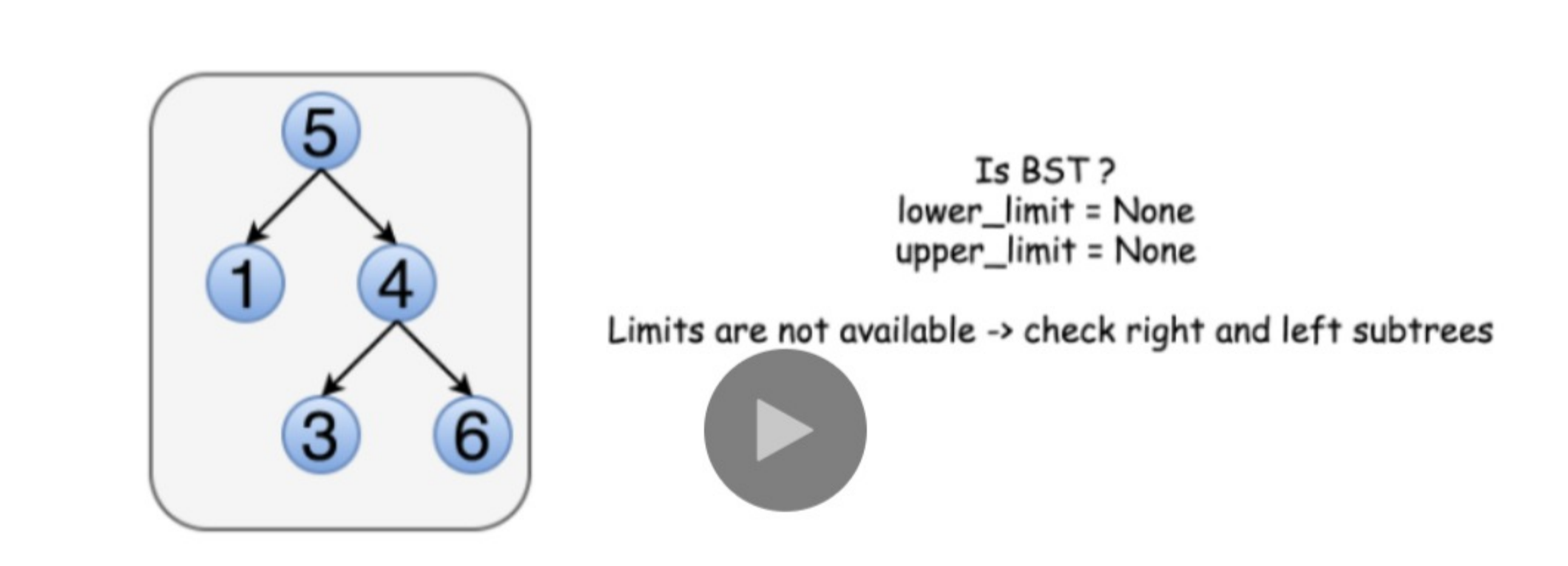
The problem is this approach will not work for all cases. Not only the right child should be larger than the node but all the elements in the right subtree. Here is an example :



That means one should keep both upper and lower limits for each node while traversing the tree, and compare the node value not with children values but with these limits.

Approach 1: Recursion

The idea above could be implemented as a recursion. One compares the node value with its upper and lower limits if they are available. Then one repeats the same step recursively for left and right subtrees.



Java Python Copy

```python
class Solution:
    def isValidBST(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        def helper(node, lower = float('-inf'), upper = float('inf')):
            if not node:
                return True

            val = node.val
            if val <= lower or val >= upper:
                return False

            if not helper(node.right, val, upper):
                return False
            if not helper(node.left, lower, val):
                return False
            return True

        return helper(root)
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N)$ since we visit each node exactly once.
- Space complexity : $\mathcal{O}(N)$ since we keep up to the entire tree.

Approach 2: Iteration

The above recursion could be converted into iteration, with the help of stack. DFS would be better than BFS since it works faster here.

Java Python Copy

```python
class Solution:
    def isValidBST(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        if not root:
            return True

        stack = [(root, float('-inf'), float('inf')), ]
        while stack:
            root, lower, upper = stack.pop()
            if not root:
                continue
            val = root.val
            if val <= lower or val >= upper:
                return False
            stack.append((root.right, val, upper))
            stack.append((root.left, lower, val))
        return True
```

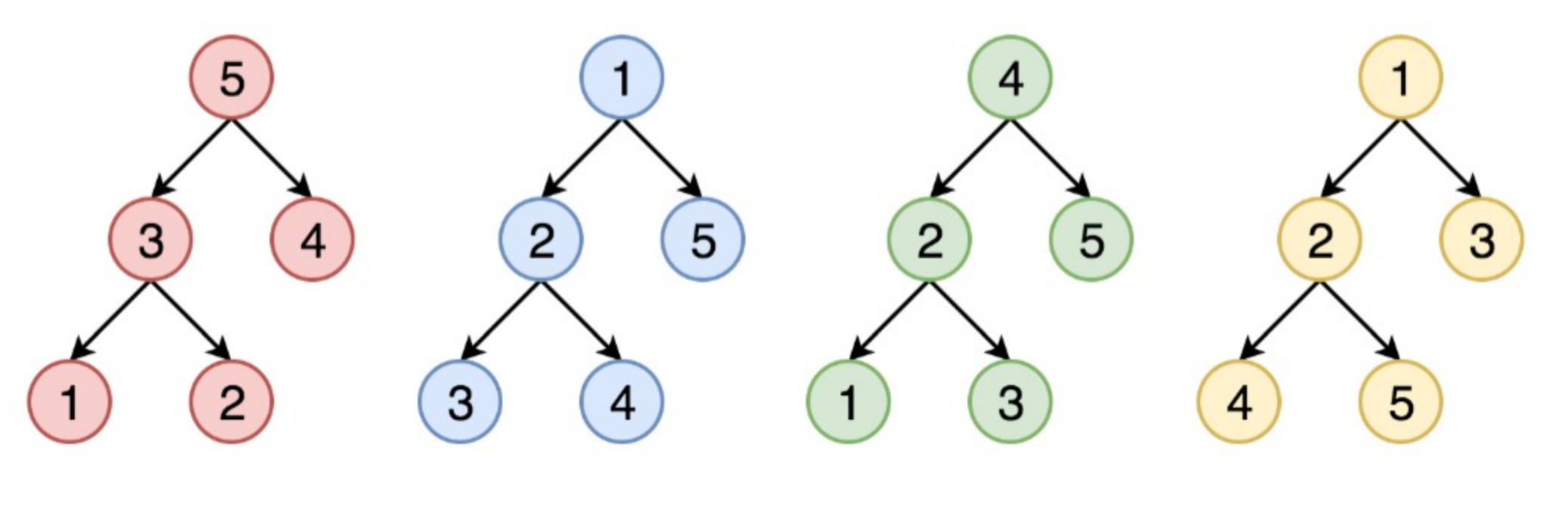
Complexity Analysis

- Time complexity : $\mathcal{O}(N)$ since we visit each node exactly once.
- Space complexity : $\mathcal{O}(N)$ since we keep up to the entire tree.

Approach 3: Inorder traversal

Algorithm

Let's use the order of nodes in the inorder traversal `Left -> Node -> Right`.

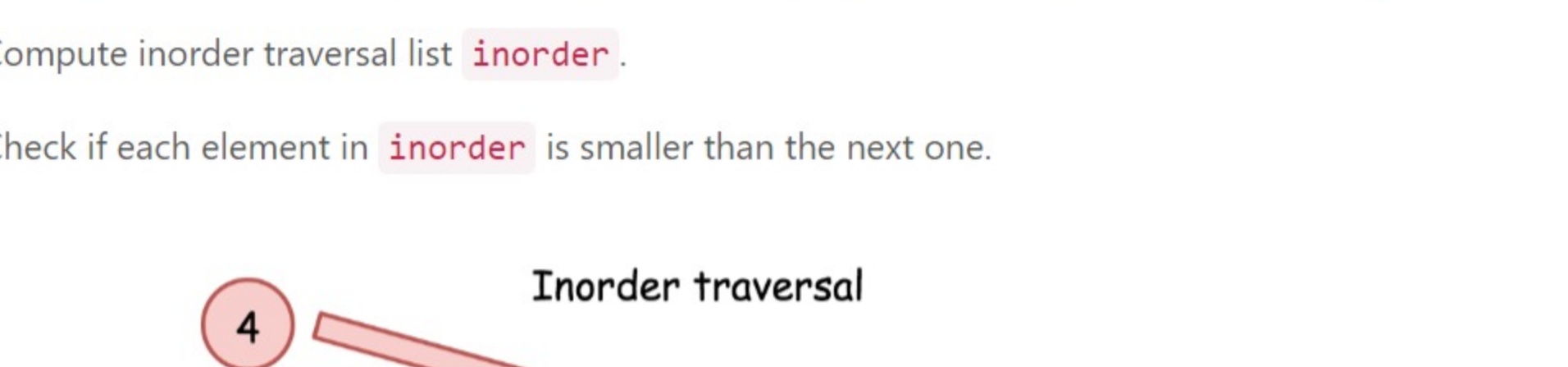


Here the nodes are enumerated in the order you visit them, and you could follow `1-2-3-4-5` to compare different strategies.

`Left -> Node -> Right` order of inorder traversal means for BST that each element should be smaller than the next one.

Hence the algorithm with $\mathcal{O}(N)$ time complexity and $\mathcal{O}(N)$ space complexity could be simple:

- Compute inorder traversal list `inorder`.
- Check if each element in `inorder` is smaller than the next one.



> Do we need to keep the whole `inorder` traversal list?

Actually, no. The last added inorder element is enough to ensure at each step that the tree is BST (or not). Hence one could merge both steps into one and reduce the used space.

Implementation

Java Python Copy

```python
class Solution:
    def isValidBST(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        stack, inorder = [], float('-inf')

        while stack or root:
            while root:
                stack.append(root)
                root = root.left
            root = stack.pop()
            # If next element in inorder traversal
            # is smaller than the previous one
            # that's not BST.
            if root.val <= inorder:
                return False
            inorder = root.val
            root = root.right

        return True
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N)$ in the worst case when the tree is BST or the "bad" element is a rightmost leaf.
- Space complexity : $\mathcal{O}(N)$ to keep `stack`.

Rate this article:

Previous Next

Comments: 71 Sort By

Type comment here... (Markdown is supported)

Preview Post

neeraj_seth ★116 December 13, 2018 10:19 AM

Other approach to solve this problem would be to use inorder traversal properties where previous element in output would always be lesser than the current output.

106 Share Reply

SHOW 1 REPLY

fudonglai ★964 March 14, 2019 9:36 PM Report

Share a concise solution, 5 lines

```
bool isValidBST(TreeNode* root, TreeNode* min=NULL, TreeNode* max=NULL) {
        if (!root) return true;
```

Read More

48 Share Reply

SHOW 3 REPLIES

durumu ★43 December 13, 2018 2:51 AM

Heaps definitely aren't "usually implemented as binary search trees." They're usually implemented as flat arrays.

42 Share Reply

SHOW 5 REPLIES

lucasmonsteer ★32 January 13, 2019 5:39 AM

Just curisou, why BFS is slower than DFS?

18 Share Reply

SHOW 4 REPLIES

jianchao-li ★14335 February 10, 2019 8:13 AM Report

The Java code of the first recursive solution can be simplified.

```
class Solution {
    private boolean isBSTHelper(TreeNode node, long lower_limit, long upper_limi
```

Read More

31 Share Reply

SHOW 4 REPLIES

v1s1on ★499 January 23, 2019 10:15 AM Report

The iterative solution is unnecessarily complicated. You can leverage the fact that an **in-order traversal** for a BST yields a monotonically increasing sequence. Obviously, it's the same asymptotic complexity but the code is much shorter & easier to follow (IMHO):

Read More

15 Share Reply

SHOW 3 REPLIES

Phatlobster ★11 May 26, 2019 11:39 AM

Anyone know why double inorder = - Double.MAX_VALUE instead of int inorder = -Integer.MIN_VALUE

12 Share Reply

SHOW 2 REPLIES

granola ★303 February 3, 2019 1:21 AM

I think, In the first approach, space will be O(h)

13 Share Reply

SHOW 5 REPLIES

iguluasg ★8 February 23, 2019 4:42 PM Report

Something recursive here:

```
class Solution {
    public boolean isValidBST(TreeNode root) {
```

Read More

8 Share Reply

SHOW 4 REPLIES

silvia0818 ★5 May 24, 2019 10:48 AM

Why in Approach 1: Recursion, the Space complexity is O(N)? I think no other auxiliary space is needed?

5 Share Reply

SHOW 4 REPLIES

‹ 1 2 3 4 5 6 7 8 ›