# 700. Search in a Binary Search Tree

**Binary Search Tree Binary Tree** Easy

### **Problem Description**

child is less than the node's value, and the right child's value is greater. We are required to implement a function that takes the root of a BST and an integer val as its input. The function should search for a node in the BST whose value is equal to val. If such a node is found, our function should return the subtree that is rooted at this node. In case there is no node with the value equal to val, the function should return null.

The given problem is about searching for a node with a specific value in a Binary Search Tree (BST). In a BST, every node's left

## The intuition behind the provided solution leverages the properties of a BST. In a BST, for any given node, all values in its left

Intuition

subtree are smaller and all values in its right subtree are larger than the node's value. This characteristic greatly optimizes the search operation. Here's how we can think about the solution step-by-step:

2. If the root is null, it means we've reached the end of a branch without finding the value, so we return null.

1. We start at the root of the BST.

- 3. If the root's value is equal to val, we've found the node we're looking for, and we return the root (thus returning the subtree rooted at the found node).
- 4. If the root's value is less than val, due to the BST property, we know that if a node with value val exists, it must be in the right subtree. So, we recursively search in the right subtree for val.
- By recursively subdividing the problem, we either find the node we're looking for or conclude it doesn't exist in the BST. This
- approach is efficient because it discards half of the BST from consideration in each step.

5. Conversely, if the root's value is greater than val, we search in the left subtree.

trees. This strategy involves breaking down a problem into smaller subproblems of the same type and solving them independently.

The solution is a classic example of the divide-and-conquer strategy, which is a recurring pattern in algorithms that deal with

The provided solution uses a simple recursive function that follows the binary search property according to which, for any node:

the tree.

**Solution Approach** 

 All values in the left subtree are less than the node's value. All values in the right subtree are greater than the node's value.

1. The function searchBST takes in two parameters: root, which is a TreeNode object, and val, which is the integer value you are searching for in

Here's a step-by-step breakdown of the code and its logic:

- If root is None, this indicates that we've reached a leaf's child (which is None), and because we haven't returned earlier, we know val is not present in the tree. Therefore, we return None.
- If root.val is equal to val, then we have found the node we are looking for, and we return the current root node, which by definition will

3. The decision cases dictate the flow of the recursive search:

call, leading to a time complexity of O(h), where h is the height of the BST.

have the desired subtree rooted at it.

2. The base cases handle two situations:

make a recursive call to searchBST(root.right, val). o Otherwise, if the current node's value is greater than val, the required node must be in the left subtree. Hence, we make a recursive call to searchBST(root.left, val).

If the current node's value is less than val, then, according to BST properties, we should search in the right subtree. Consequently, we

value val is found, or None if it is absent in the BST. The solution approach is efficient and takes advantage of the BST properties to minimize the search space with each recursive

The recursive calls continue until the base case is satisfied, which will either return the desired node (TreeNode object) if the

Let's illustrate the solution approach with a small example. Suppose we have a Binary Search Tree and an integer value val that we want to search for in the BST.

BST:

359

property.

**Example Walkthrough** 

3. The new root node to consider is now the left child of the previous root, which has the value 4.

def \_\_\_init\_\_\_(self, value=0, left=None, right=None):

if root is None or root.value == value:

# otherwise, search in the left subtree.

public TreeNode searchBST(TreeNode root, int val) {

if (root == null || root.val == val) {

// Otherwise, search in the left subtree.

TreeNode \*left; // Pointer to the left child

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

return root;

\* Definition for a binary tree node.

TreeNode \*right;

def search\_bst(self, root: TreeNode, value: int) -> TreeNode:

:return: TreeNode, the node containing the value, or None if not found

# If root is None, or root's value matches the search value, return root.

# If the value to be searched is greater than the root's value, search in the right subtree,

// Base case: if the root is null or root value equals the search value, return the root.

// If the value of the root is less than the search value, search in the right subtree.

return root.val < val ? searchBST(root.right, val) : searchBST(root.left, val);</pre>

// The value stored in the tree node

// Pointer to the right child

// Constructor to initialize the node with given value, left and right children

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

1. Begin at the root of the BST, which has the value 6.

Let's say val = 5. According to the solution approach:

- 7. Return the subtree rooted at the node with value 5.
- Since we have found the value, our function will return the subtree rooted at node 5, which, since node 5 is a leaf, is just the node itself:

2. Compare the root's value (6) with val (5). Since 5 is less than 6, look into the left subtree based on BST property.

5. The new root node to consider is now the right child of the previous root, which has the value 5.

4. Compare the new root's value (4) with val (5). Since 5 is greater than 4, we need to search in the right subtree of node 4 based on BST

6. Compare the new root's value (5) with val (5). They are equal, hence we have found the node with the value we are looking for.

If val were not in the BST, for instance val = 7, the search process would eventually reach a point where the next recursive call would try to access a child of a leaf node (which is null), leading to step 2 of our solution approach and returning None.

**Python** 

#### Searches for a node with a given value in a binary search tree. :param root: TreeNode, the root node of the binary search tree :param value: int, the value to search for

Solution Implementation

self.value = value

self.right = right

return root

if root.value < value:</pre>

self.left = left

class TreeNode:

class Solution:

```
return self.search_bst(root.right, value)
       else:
            return self.search_bst(root.left, value)
Java
/**
* Definition for a binary tree node.
*/
public class TreeNode {
   int val;
   TreeNode left;
   TreeNode right;
   TreeNode() {}
   TreeNode(int val) {
       this.val = val;
   TreeNode(int val, TreeNode left, TreeNode right) {
       this.val = val;
       this.left = left;
       this.right = right;
class Solution {
    * Searches for a node with a given value in a Binary Search Tree.
    * @param root The root of the binary search tree.
    * @param val The value to search for.
    * @return The subtree rooted with the target value; or null if value doesn't exist in the tree.
```

```
// Constructor to initialize the node with default values
TreeNode() : val(0), left(nullptr), right(nullptr) {}
// Constructor to initialize the node with a given value
```

**}**;

C++

/\*\*

class TreeNode {

int val;

class Solution {

public:

/\*\*

public:

\*/

```
* Searches the Binary Search Tree for a node with the given value.
    * @param root A pointer to the root node of the BST.
    * @param value The value to search for in the BST.
    * @return The TreeNode pointer to the found node or nullptr if not found.
    */
    TreeNode* searchBST(TreeNode* root, int value) {
       // Base case: If the root is nullptr or the root's value is the one we're searching for
       if (!root || root->val == value) {
            return root;
       // If the given value is greater than the root's value, search in the right subtree.
       if (root->val < value) {</pre>
            return searchBST(root->right, value);
       // If the given value is smaller than the root's value, search in the left subtree.
       return searchBST(root->left, value);
};
TypeScript
// Definition for a binary tree node.
class TreeNode {
                   // The value stored in the tree node
    val: number;
    left: TreeNode | null; // Pointer to the left child
    right: TreeNode | null; // Pointer to the right child
    // Constructor to initialize the node with default values or with given values for val, left, and right.
    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
       this.val = val;
       this.left = left;
       this.right = right;
/**
* Searches for a node with the given value in a Binary Search Tree.
 * @param root A TreeNode representing the root of the BST.
 * @param value The value to search for in the BST.
 * @return A TreeNode pointer to the found node or null if not found.
function searchBST(root: TreeNode | null, value: number): TreeNode | null {
    // Base case: if the root is null or the root's value matches the search value
```

```
class TreeNode:
   def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

class Solution:

if (!root || root.val === value) {

return searchBST(root.left, value);

return searchBST(root.right, value);

def search\_bst(self, root: TreeNode, value: int) -> TreeNode:

:param value: int, the value to search for

Time and Space Complexity

Searches for a node with a given value in a binary search tree.

:param root: TreeNode, the root node of the binary search tree

return root;

if (value > root.val) {

```
:return: TreeNode, the node containing the value, or None if not found
# If root is None, or root's value matches the search value, return root.
if root is None or root.value == value:
    return root
# If the value to be searched is greater than the root's value, search in the right subtree,
# otherwise, search in the left subtree.
if root.value < value:</pre>
    return self.search_bst(root.right, value)
else:
    return self.search_bst(root.left, value)
```

// If the search value is greater than the root's value, search in the right subtree.

// If the search value is less than the root's value, search in the left subtree.

tree, this would be O(log n), where n is the number of nodes in the tree. However, if the tree is skewed, the height h can become n, resulting in a time complexity of O(n). The space complexity of the code is also 0(h) due to the recursive nature of the algorithm. Each recursive call adds a new layer to the call stack, which means that in the worst case, we could have h recursive calls on the stack at the same time. In the best

case of a balanced tree, this would be  $0(\log n)$ , while in the worst case of a skewed tree, this would be 0(n).

The time complexity of the given code is O(h) where h is the height of the tree. This is because in the worst case, we may have to

traverse from the root to the leaf which involves going through the height of the tree. In the case of a balanced binary search