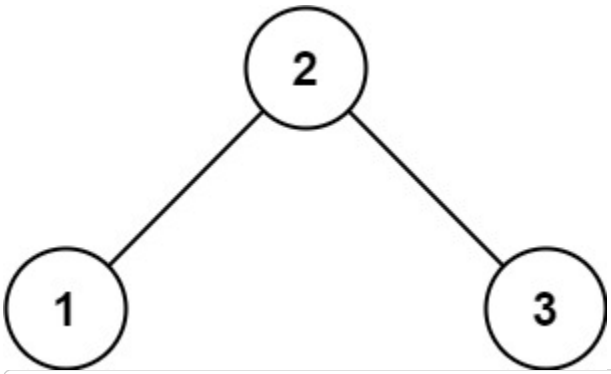
98. Validate Binary Search Tree

Given the `root` of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

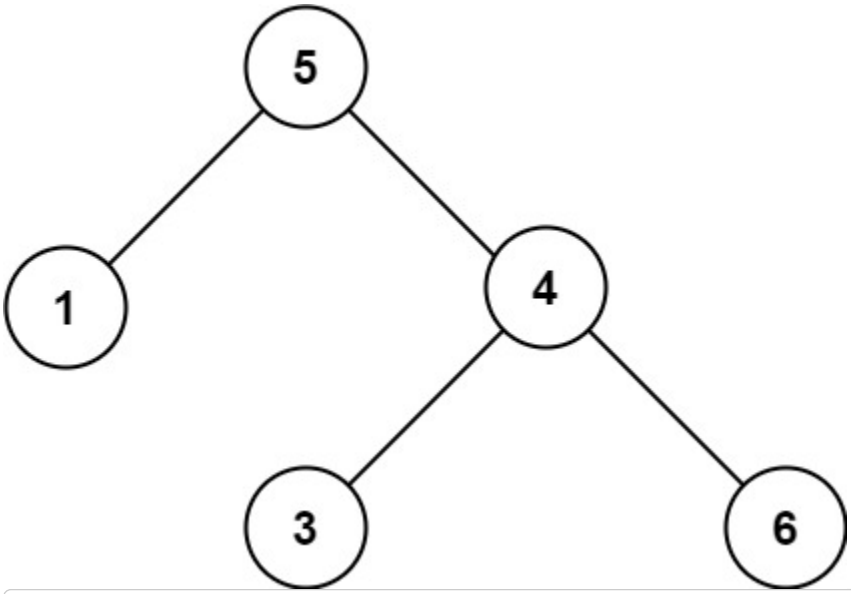
- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:




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

Example 2:




```
Input: root = [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.
```

Constraints:

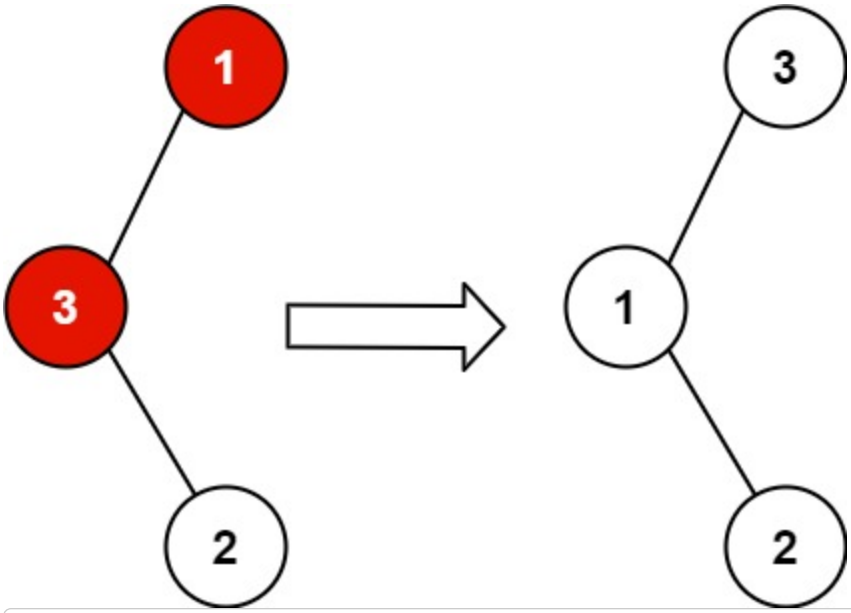
- The number of nodes in the tree is in the range `[1, 10^4]`.
- `-2^31 <= Node.val <= 2^31 - 1`

```
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        res = True
        prev = None
        def inorder(root):
            nonlocal prev,res
            if not root:return True
            inorder(root.left)
            if prev != None and prev >= root.val:
                res = False
                return
            prev = root.val
            inorder(root.right)
        inorder(root)
        return res
```

99. Recover Binary Search Tree

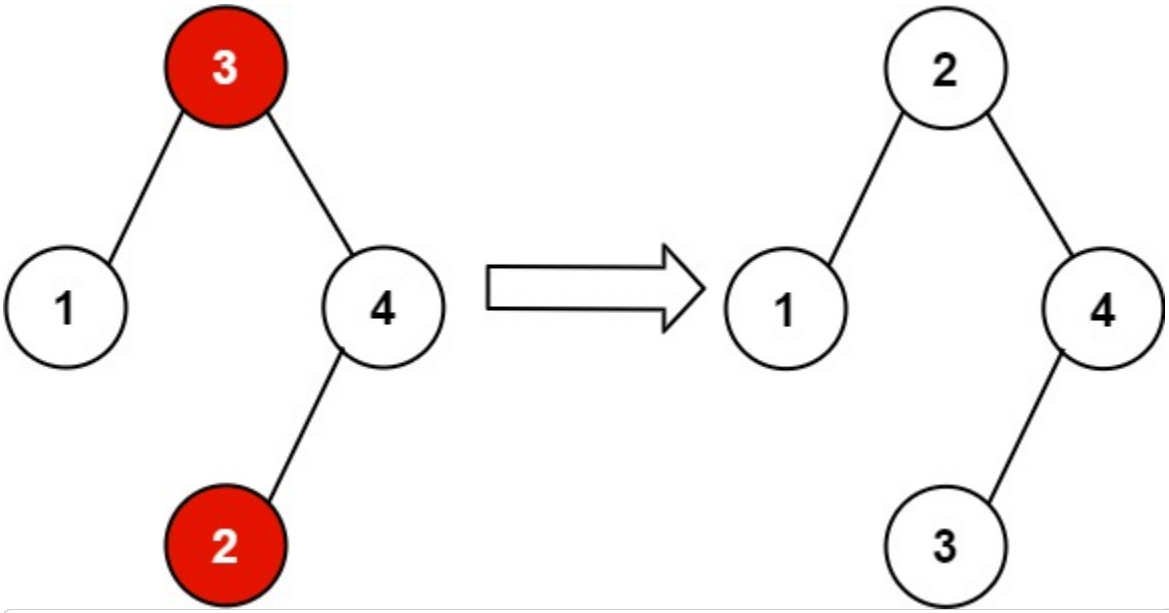
You are given the `root` of a binary search tree (BST), where the values of **exactly** two nodes of the tree were swapped by mistake. *Recover the tree without changing its structure*.

Example 1:




```
Input: root = [1,3,null,null,2]
Output: [3,1,null,null,2]
Explanation: 3 cannot be a left child of 1 because 3 > 1. Swapping 1 and 3 makes the
```

Example 2:




```
Input: root = [3,1,4,null,null,2]
Output: [2,1,4,null,null,3]
Explanation: 2 cannot be in the right subtree of 3 because 2 < 3. Swapping 2 and 3 ma
```

Constraints:

- The number of nodes in the tree is in the range `[2, 1000]`.
- -2^{31} <= Node.val <= $2^{31} - 1$

Follow up: A solution using `O(n)` space is pretty straight-forward. Could you devise a constant `O(1)` space solution?

```
class Solution:
    def recoverTree(self, node: Optional[TreeNode]) -> None:
        first,middle,last,prev = None,None,None,None
        def inorder(root):
            nonlocal first,middle,last,prev
            if not root:
                return None
            inorder(root.left)
            if prev and root.val < prev.val:
                if not first:
                    first = prev
                    middle = root
                else:
                    last = root
            prev = root
            inorder(root.right)
        inorder(node)
        if first and last:
            first.val,last.val = last.val,first.val
        else:
            middle.val,first.val = first.val,middle.val
        return node
```

173. Binary Search Tree Iterator

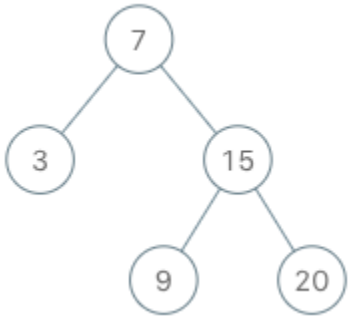
Implement the `BSTIterator` class that represents an iterator over the **in-order traversal (https://en.wikipedia.org/wiki/Tree_traversal#In-order_(LNR))** of a binary search tree (BST):

- `BSTIterator(TreeNode root)` Initializes an object of the `BSTIterator` class. The `root` of the BST is given as part of the constructor. The pointer should be initialized to a non-existent number smaller than any element in the BST.
- `boolean hasNext()` Returns `true` if there exists a number in the traversal to the right of the pointer, otherwise returns `false`.
- `int next()` Moves the pointer to the right, then returns the number at the pointer.

Notice that by initializing the pointer to a non-existent smallest number, the first call to `next()` will return the smallest element in the BST.

You may assume that `next()` calls will always be valid. That is, there will be at least a next number in the in-order traversal when `next()` is called.

Example 1:




```
Input
["BSTIterator", "next", "next", "hasNext", "next", "hasNext", "next", "hasNext", "nex
[[[7, 3, 15, null, null, 9, 20]], [], [], [], [], [], [], [], [], []]
Output
[null, 3, 7, true, 9, true, 15, true, 20, false]

Explanation
BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]);
bSTIterator.next();    // return 3
bSTIterator.next();    // return 7
bSTIterator.hasNext(); // return True
bSTIterator.next();    // return 9
bSTIterator.hasNext(); // return True
bSTIterator.next();    // return 15
bSTIterator.hasNext(); // return True
bSTIterator.next();    // return 20
bSTIterator.hasNext(); // return False
```

Constraints:

- The number of nodes in the tree is in the range `[1, 10^5]`.
- `0 <= Node.val <= 10^6`
- At most `10^5` calls will be made to `hasNext`, and `next`.

Follow up:

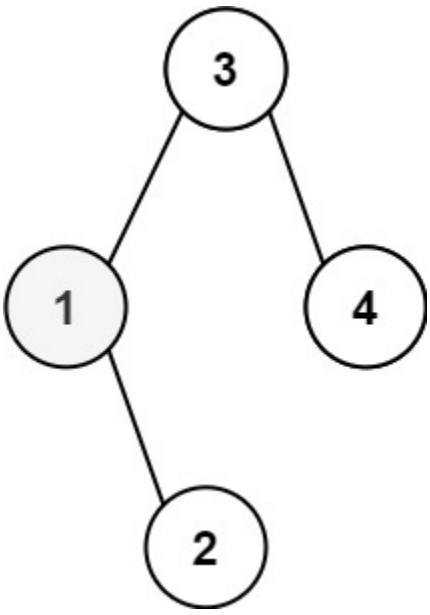
- Could you implement `next()` and `hasNext()` to run in average `O(1)` time and use `O(h)` memory, where `h` is the height of the tree?

```
class BSTIterator:
    def fill_stack(self,stack,root):
        while root:
            stack.append(root)
            root = root.left
    def __init__(self, root: Optional[TreeNode]):
        self.root = root
        self.stack = []
        if root:
            self.fill_stack(self.stack,self.root)
    def next(self) -> int:
        node = self.stack.pop()
        if node.right:
            self.fill_stack(self.stack,node.right)
        return node.val

    def hasNext(self) -> bool:
        return True if self.stack else False
```

230. Kth Smallest Element in a BST

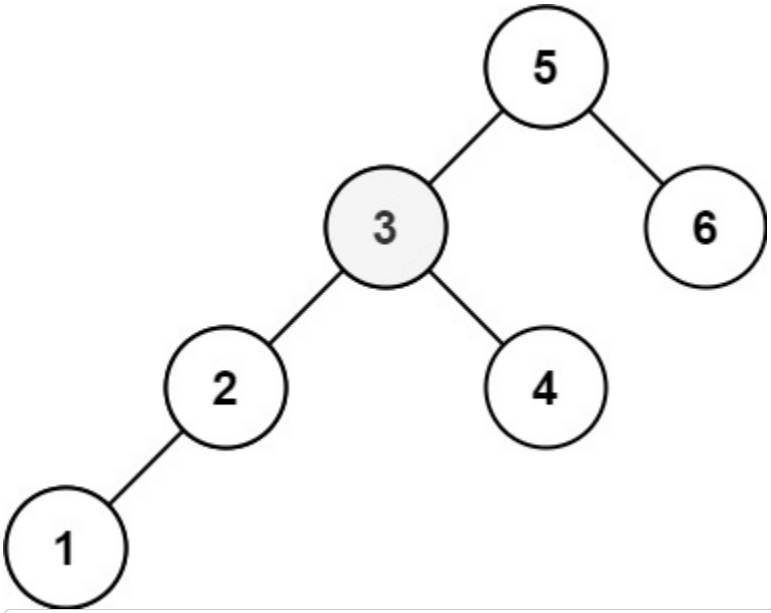
Given the `root` of a binary search tree, and an integer `k`, return *the* k^{th} *smallest value (**1-indexed**) of all the values of the nodes in the tree.*

Example 1:



```
Input: root = [3,1,4,null,2], k = 1
Output: 1
```

Example 2:




```
Input: root = [5,3,6,2,4,null,null,1], k = 3
Output: 3
```

Constraints:

- The number of nodes in the tree is `n`.
- `1 <= k <= n <= 10^4`
- `0 <= Node.val <= 10^4`

Follow up: If the BST is modified often (i.e., we can do insert and delete operations) and you need to find the kth smallest frequently, how would you optimize?

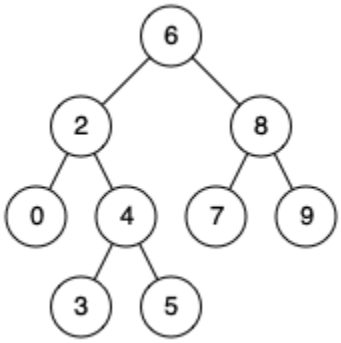
```
class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        res = -1
        def inorder(root):
            nonlocal k,res
            if not root or k < 1 :
                return
            inorder(root.left)
            if  1 == k :
                res = root.val
            k-=1
            inorder(root.right)
        inorder(root)
        return res
```

235. Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

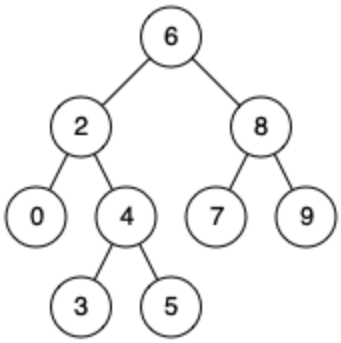
According to the definition of LCA on Wikipedia (https://en.wikipedia.org/wiki/Lowest_common_ancestor): "The lowest common ancestor is defined between two nodes `p` and `q` as the lowest node in `T` that has both `p` and `q` as descendants (where we allow **a node to be a descendant of itself**)."

Example 1:




```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: The LCA of nodes 2 and 8 is 6.
```

Example 2:




```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
Output: 2
Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itsel
```

Example 3:

```
Input: root = [2,1], p = 2, q = 1
Output: 2
```

Constraints:

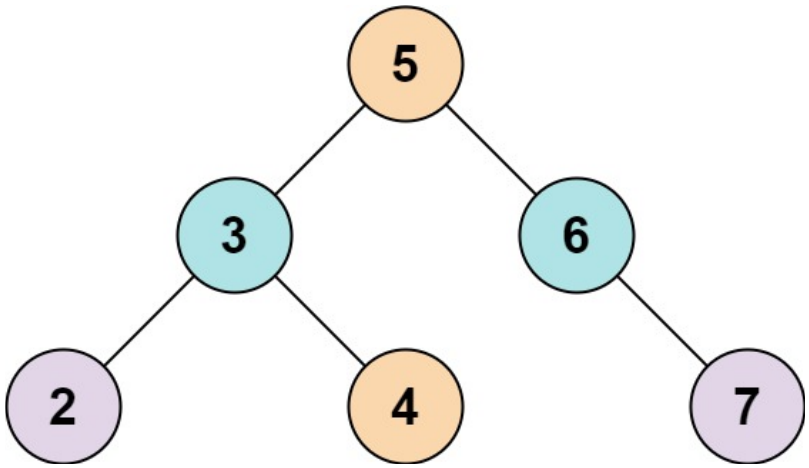
- The number of nodes in the tree is in the range `[2, 10^5]`.
- `-10^9 <= Node.val <= 10^9`
- All `Node.val` are **unique**.
- `p != q`
- `p` and `q` will exist in the BST.

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode')
-> 'TreeNode':
        if not root:
            return None

        cur  = root.val
        if  p.val < cur and q.val <cur:
            return self.lowestCommonAncestor(root.left,p,q)
        elif p.val > cur and q.val > cur:
            return  self.lowestCommonAncestor(root.right,p,q)
        return root
```

653. Two Sum IV - Input is a BST

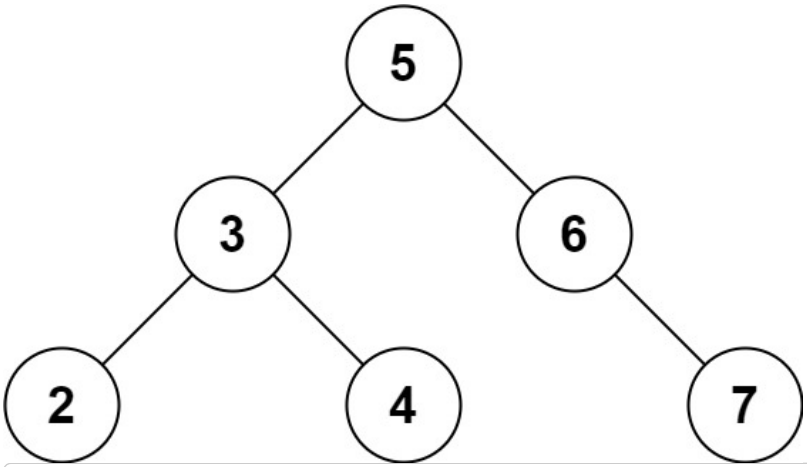
Given the `root` of a binary search tree and an integer `k`, return `true` *if there exist two elements in the BST such that their sum is equal to* `k`, *or* `false` *otherwise.*

Example 1:



```
Input: root = [5,3,6,2,4,null,7], k = 9
Output: true
```

Example 2:




```
Input: root = [5,3,6,2,4,null,7], k = 28
Output: false
```

Constraints:

- The number of nodes in the tree is in the range `[1, 10^4]`.
- `-10^4 <= Node.val <= 10^4`
- `root` is guaranteed to be a **valid** binary search tree.
- `-10^5 <= k <= 10^5`

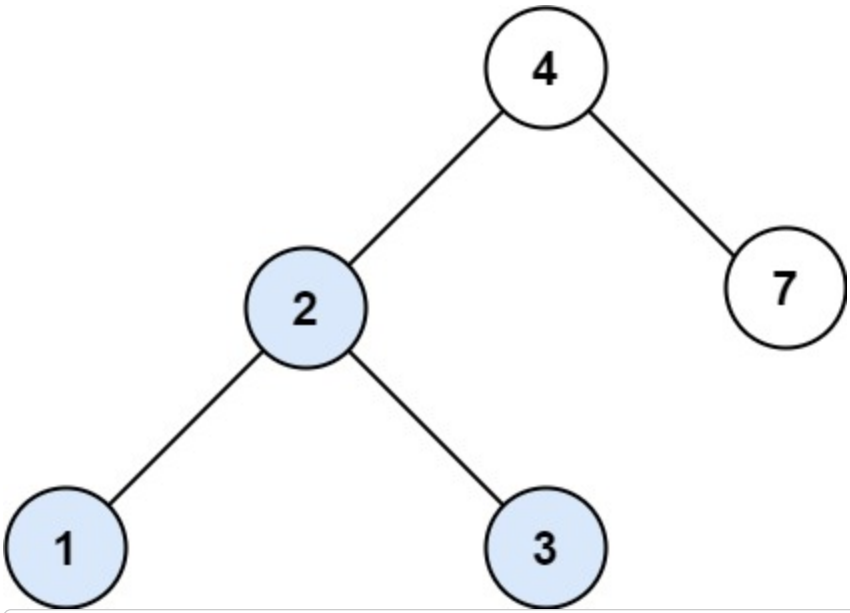
```
class Solution:
    def findTarget(self, root: Optional[TreeNode], k: int) -> bool:
        hashmap = set()
        def inorder(root):
            if not root:
                return False
            if k - root.val in hashmap:
                return True
            hashmap.add(root.val)
            return inorder(root.left) or inorder(root.right)
        return inorder(root)
```

700. Search in a Binary Search Tree

You are given the `root` of a binary search tree (BST) and an integer `val`.

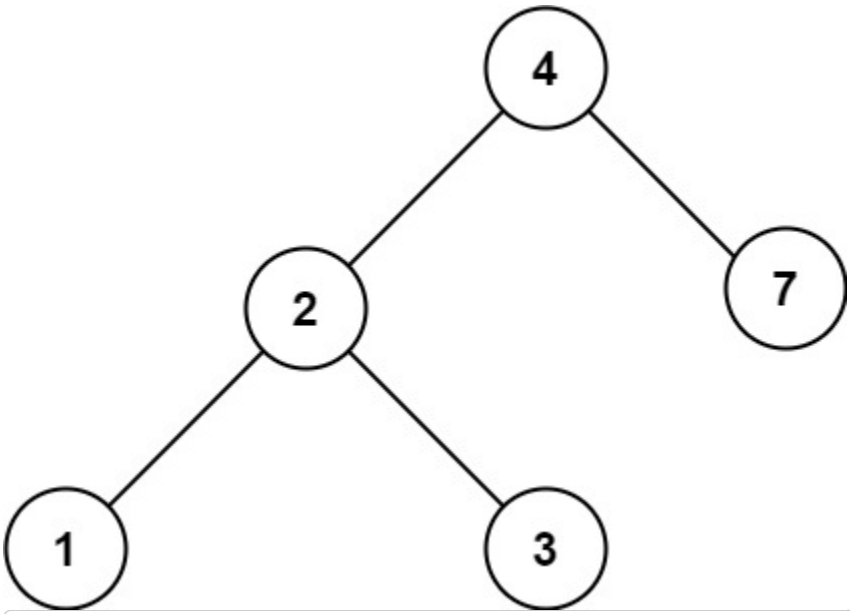
Find the node in the BST that the node's value equals `val` and return the subtree rooted with that node. If such a node does not exist, return `null`.

Example 1:




```
Input: root = [4,2,7,1,3], val = 2
Output: [2,1,3]
```

Example 2:




```
Input: root = [4,2,7,1,3], val = 5
Output: []
```

Constraints:

- The number of nodes in the tree is in the range `[1, 5000]`.
- `1 <= Node.val <= 10^7`
- `root` is a binary search tree.
- `1 <= val <= 10^7`

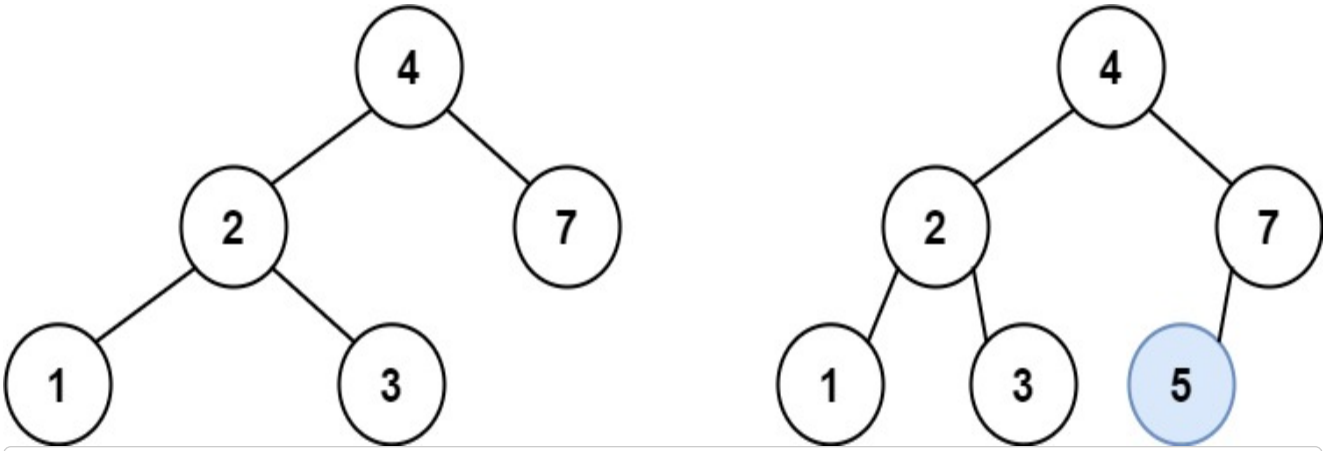
```
class Solution:
    def searchBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:

        def dfs(root,val):
            if not root:return None
            if root.val == val:
                return root
            if val < root.val:
                return  dfs(root.left,val)
            else:
                return dfs(root.right,val)
            return None

        return dfs(root,val)
```

701. Insert into a Binary Search Tree

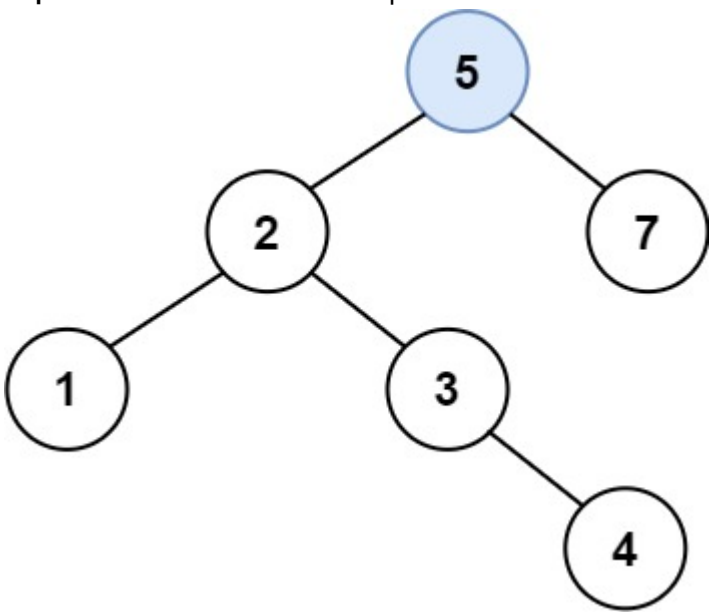
You are given the `root` node of a binary search tree (BST) and a `value` to insert into the tree. Return *the root node of the BST after the insertion*. It is **guaranteed** that the new value does not exist in the original BST.

Notice that there may exist multiple valid ways for the insertion, as long as the tree remains a BST after insertion. You can return **any of them**.

Example 1:




```
Input: root = [4,2,7,1,3], val = 5
Output: [4,2,7,1,3,5]
Explanation: Another accepted tree is:
```



Example 2:

```
Input: root = [40,20,60,10,30,50,70], val = 25
Output: [40,20,60,10,30,50,70,null,null,25]
```

Example 3:

```
Input: root = [4,2,7,1,3,null,null,null,null,null,null], val = 5
Output: [4,2,7,1,3,5]
```

Constraints:

- The number of nodes in the tree will be in the range `[0, 10^4]`.
- `-10^8 <= Node.val <= 10^8`

- All the values `Node.val` are **unique**.
- `-10^8 <= val <= 10^8`
- It's **guaranteed** that `val` does not exist in the original BST.

```
class Solution:
    def insertIntoBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNod
e]:
        cur = root
        def dfs(node):
            nonlocal val
            if not node:
                return TreeNode(val)
            if val < node.val:
                node.left = dfs(node.left)
            else:
                node.right = dfs(node.right)
            return node
        return dfs(cur)
```

1008. Construct Binary Search Tree from Preorder Traversal

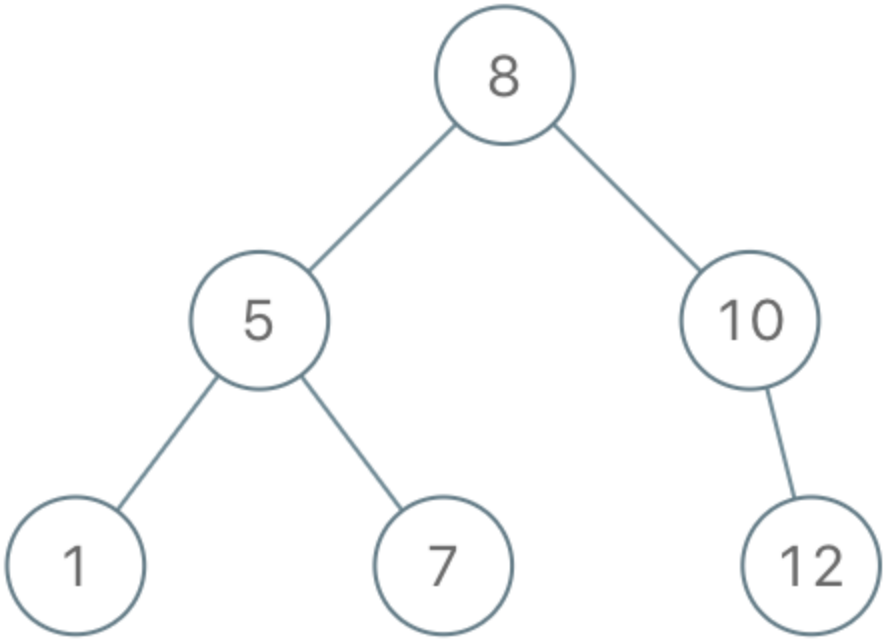
Given an array of integers preorder, which represents the **preorder traversal** of a BST (i.e., **binary search tree**), construct the tree and return *its root*.

It is **guaranteed** that there is always possible to find a binary search tree with the given requirements for the given test cases.

A **binary search tree** is a binary tree where for every node, any descendant of `Node.left` has a value **strictly less than** `Node.val`, and any descendant of `Node.right` has a value **strictly greater than** `Node.val`.

A **preorder traversal** of a binary tree displays the value of the node first, then traverses `Node.left`, then traverses `Node.right`.

Example 1:




```
Input: preorder = [8,5,1,7,10,12]
Output: [8,5,10,1,7,null,12]
```

Example 2:

```
Input: preorder = [1,3]
Output: [1,null,3]
```

Constraints:

- `1 <= preorder.length <= 100`
- `1 <= preorder[i] <= 1000`
- All the values of `preorder` are **unique**.

```
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        ind  = 0
        def build(bound):
            nonlocal ind
            if ind >= len(preorder):
                return None
            if preorder[ind] <= bound:
                root = TreeNode(preorder[ind])
            else:
                return None
            ind+=1
            root.left = build(root.val)
            root.right = build(bound)
            return root

        return build(float("inf"))
```
