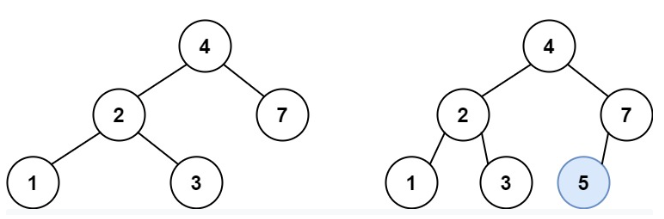
<https://leetcode.com/problems/insert-into-a-binary-search-tree/>

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, 104].
* -108 <= Node.val <= 108
* All the values Node.val are **unique**.
* -108 <= val <= 108
* It's **guaranteed** that val does not exist in the original BST.

**Attempt 1: 2022-10-24**

**Solution 1:  Recursive traversal (10min)**

/\*\*

\* 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 {

public TreeNode insertIntoBST(TreeNode root, int val) {

if(root == null) {

return new TreeNode(val);

}

if(root.val > val) {

root.left = insertIntoBST(root.left, val);

} else {

root.right = insertIntoBST(root.right, val);

}

return root;

}

}

Time complexity: O(N) (not O(H)) Typical dfs

Space complexity: O(H) as considering recursion stack, takes place in internal memory, if not consider then O(1)

----------------------------------------------------------------------------------------------------------------

Discuss on Time complexity

Refer to

https://leetcode.com/problems/insert-into-a-binary-search-tree/discuss/1683942/Well-Detailed-Explaination-Java-C++-Python-oror-Easy-for-mind-to-Accept-it/1215525

A: yetiiiiii (best answer)

@hi-malik Big-O is the upper bound and TC of recursive solution will be O(n) and not O(h). We are generally only considerate about the worst case of any complexity and in the worst case, Height of the BST = # of nodes (in a skewed tree).

and we will need to traverse each node once and thus, TC:O(n)

----------------------------------------------------------------------------------------------------------------

Q: PanPip

@Nam\_22 In this case the height h will be equal to n, so it's still O(h), right?

----------------------------------------------------------------------------------------------------------------

A: yetiiiiii

@PanPip Not necessarily. It must have been O(h) if it was mentioned it is a balanced tree.

Height depends on the structure of the tree. So, it could be O(n) or O(log n).

O(h) will be a weaker bound though still valid

10 40

\ / \

20 20 60

\ / \

30 10 30

\

40

'h = n' 'h=logn'

----------------------------------------------------------------------------------------------------------------

A: IRun\_Man

@PanPip O(h) is a better and more accurate representation. Whether the tree is skewed or not, the no. of comparisons will always be bound by the tree's height. If the tree is highly skewed then h=n and still the time complexity is O(h). NOTE : h != log n, its only for balanced tree.

**Solution 2:  Iterative traversal  with while(true) loop (20min)**

/\*\*

\* 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 {

public TreeNode insertIntoBST(TreeNode root, int val) {

if(root == null) {

return new TreeNode(val);

}

// Since we are going to change the tree structure, create a pointer for traversal

TreeNode cur = root;

// Running an infinity loop, look for the place for new node to add

while(true) {

// Go on left subtree

if(cur.val > val) {

if(cur.left != null) {

cur = cur.left;

} else {

// Otherwise add new value to current node's left

cur.left = new TreeNode(val);

// Breaking this infinity loop

break;

}

// Go on right subtree

} else {

if(cur.right != null) {

cur = cur.right;

} else {

// Otherwise add new value to current node's right

cur.right = new TreeNode(val);

// Breaking this infinity loop

break;

}

}

}

return root;

}

}

Time complexity: O(N)

Space complexity: O(N)

**Refer to**

<https://leetcode.com/problems/insert-into-a-binary-search-tree/discuss/1683942/Well-Detailed-Explaination-Java-C%2B%2B-Python-oror-Easy-for-mind-to-Accept-it>

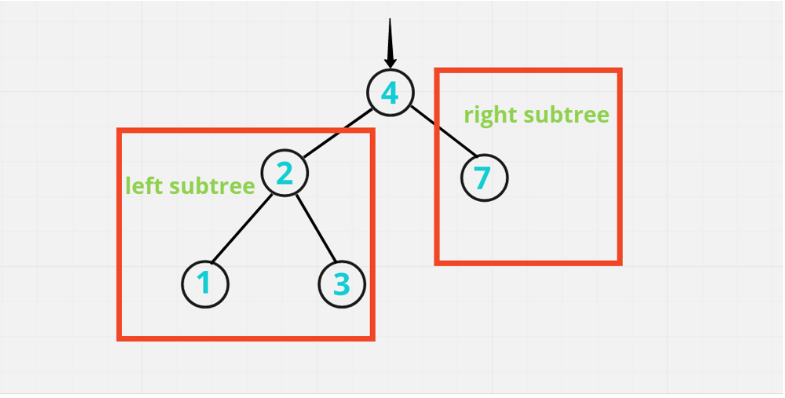
**Approach Explain :**

**Summary :**If the **root** is empty, the new tree node can be returned as the root node.

Otherwise compare **root. val** is related to the size of the target value:

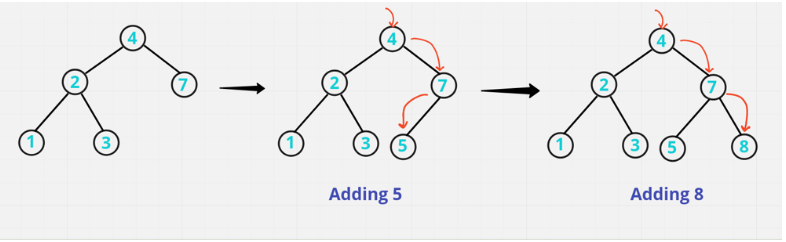
* If **root.val** is greater than the **target value**, indicating that the target value should be inserted into the **left subtree of the root**, and the problem becomes root. Insert the target value in the left and recursively call the current function;
* If **root.val** is less than the **target value**, indicating that the target value should be inserted into the **right subtree of the root**, and the problem becomes root. Insert the target value in right and recursively call the current function.

**Explanation:** In Binary search tree follow the property, all the nodes on **right subtree**, value is **greater** then the **root value** & all the nodes on **left subtree**, value is **less** then the **root value**.



So, now let's say we need to **add 5**. We will first compare with the **root node**. If it is less then we go on to the left subtree, if it is greater then we go to the right subtree. So, in this **example** **right subtree exist** and we will compare with **7**. Then it will go to the left & left subtree doesn't exist over here, then we will add the new node over here.

Now let's say we need to add **8**, then we will go again the right step and here we will **add 8**



*Hope you got the point*

**Let's Code it up:**

**Method - 1: Recursive**

*Recursive Code line explain's :* Similar for C++, Java, Python **{Only Syntax Difference}** approach same

{

if(root == null) return new TreeNode(val); // if root doesn't exist, then return new TreeNode value

if(root.val > val) root.left = insertIntoBST(root.left, val); // if root value is greater then value, it means our root value exist on left side

else root.right = insertIntoBST(root.right, val); // otherwise root value is lesser then value, it means our root value exist on right side

return root; // returning original root node

**Java**

class Solution {

public TreeNode insertIntoBST(TreeNode root, int val) {

if(root == null) return new TreeNode(val);

if(root.val > val) root.left = insertIntoBST(root.left, val);

else root.right = insertIntoBST(root.right, val);

return root;

}

}

ANALYSIS :-

* **Time Complexity :-** BigO(N)
* **Space Complexity :-** BigO(H) as considering recursion stack, takes place in internal memory, if not consider then O(1)

**Method - 2: Iterative**

*Iterative Code line explain's :* Similar for C++, Java **{Only Syntax Difference}** approach same

if(root == null) return new TreeNode(val);

TreeNode curr = root;

while(true){ // running an infinity loop, look for the place for new node to add

if(curr.val < val){

if(curr.right != null) curr = curr.right; // update current on right

else {

curr.right = new TreeNode(val); // otherwise add current of right to new value TreeNode

break; // breaking this infinity loop

}

}

else{

if(curr.left != null) curr = curr.left; // update current on left

else{

curr.left = new TreeNode(val); // otherwise add current of left to new value TreeNode

break; // breaking this infinity loop

}

}

}

return root;

**Java**

class Solution {

public TreeNode insertIntoBST(TreeNode root, int val) {

if(root == null) return new TreeNode(val);

TreeNode curr = root;

while(true){

if(curr.val < val){

if(curr.right != null) curr = curr.right;

else {

curr.right = new TreeNode(val);

break;

}

}

else{

if(curr.left != null) curr = curr.left;

else{

curr.left = new TreeNode(val);

break;

}

}

}

return root;

}

}

ANALYSIS :-

* **Time Complexity :-** BigO(N), where N is height of binary search tree
* **Space Complexity :-** BigO(1)

**Refer to**

<https://leetcode.com/problems/insert-into-a-binary-search-tree/discuss/1683942/Well-Detailed-Explaination-Java-C++-Python-oror-Easy-for-mind-to-Accept-it/1215460>

There are 3 facts to BST :

1. Inorder Traversal of BST is a sorted sequence.
2. In a BST,
   * all the values smaller then root, exists in the left subtree.
   * all the values greater then root, exists in the right subtree.

and this is recursively true for every node.

1. A new key is always inserted at the leaf.

In order to find the position to insert a value, we need to check the condition-2 at every node while traversing : ( and update the links while inserting )

* TC : O(n) -> Typical dfs
* SC : O(n) -> Stack Space ( in the worst case, the tree could be pathological tree )

class Solution {

public TreeNode insertIntoBST(TreeNode node, int val) {

if(node == null) return new TreeNode(val);

if(val < node.val) {

node.left = insertIntoBST(node.left, val);

}

else if(val > node.val) {

node.right = insertIntoBST(node.right, val);

}

return node;

}

}

**Solution 3:  Iterative traversal  with Stack  (20min)**

/\*\*

\* 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 {

public TreeNode insertIntoBST(TreeNode root, int val) {

if(root == null) {

return new TreeNode(val);

}

TreeNode cur = null;

Stack<TreeNode> stack = new Stack<TreeNode>();

stack.push(root);

while(!stack.isEmpty()) {

cur = stack.pop();

if(cur.val > val) {

if(cur.left != null) {

// Different than while(true) solution as no need cur = cur.left,

// since we will update 'cur' by 'cur = stack.pop()' at beginning

// of while loop at each iteration, for example, first iteration

// in while loop we pop out 'root' as 'cur', second iteration since

// we stored 'cur.left' on the stack in the first iteration then

// pop it out in second itearation, we will directly get 'cur.left'

stack.push(cur.left);

} else {

cur.left = new TreeNode(val);

break;

}

} else {

if(cur.right != null) {

stack.push(cur.right);

} else {

cur.right = new TreeNode(val);

break;

}

}

}

return root;

}

}

Time complexity: O(N)

Space complexity: O(N)

**Refer to**

<https://leetcode.com/problems/insert-into-a-binary-search-tree/discuss/150757/java-iterative-100/296563>

Similar idea but with a stack instead of the while true

public TreeNode insertIntoBST(TreeNode root, int val) {

if(root == null) return new TreeNode(val);

TreeNode current = root;

Deque<TreeNode> stack = new ArrayDeque<>();

stack.offerLast(root);

while(!stack.isEmpty()) {

current = stack.pollLast();

if(val > current.val) {

if(current.right == null) {

current.right = new TreeNode(val);

return root;

} else {

stack.offerLast(current.right);

}

} else {

if(current.left == null) {

current.left = new TreeNode(val);

return root;

} else {

stack.offerLast(current.left);

}

}

}

return root;

}