**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.
* 
* **Input**
* ["BSTIterator", "next", "next", "hasNext", "next", "hasNext", "next", "hasNext", "next", "hasNext"]
* [[[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

**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:**

**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.



**Input:** preorder = [8,5,1,7,10,12]

**Output:** [8,5,10,1,7,null,12]

**GFG : Binary Tree to Binary Search Tree Conversion**

Given a Binary Tree, convert it to a Binary Search Tree. The conversion must be done in such a way that keeps the original structure of Binary Tree.

Example 1

Input:

10

/ \

2 7

/ \

8 4

Output:

8

/ \

4 10

/ \

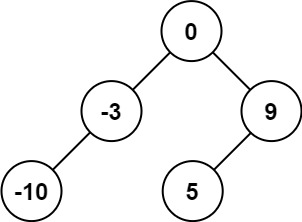
2 7

**108. Convert Sorted Array to Binary Search Tree**

Given an integer array nums where the elements are sorted in ascending order, convert *it to a height-balanced binary search tree*.

A height-balanced binary tree is a binary tree in which the depth of the two subtrees of every node never differs by more than one.

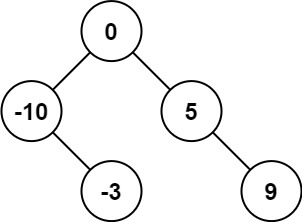
Example 1:



Input: nums = [-10,-3,0,5,9]

Output: [0,-3,9,-10,null,5]

Explanation: [0,-10,5,null,-3,null,9] is also accepted:



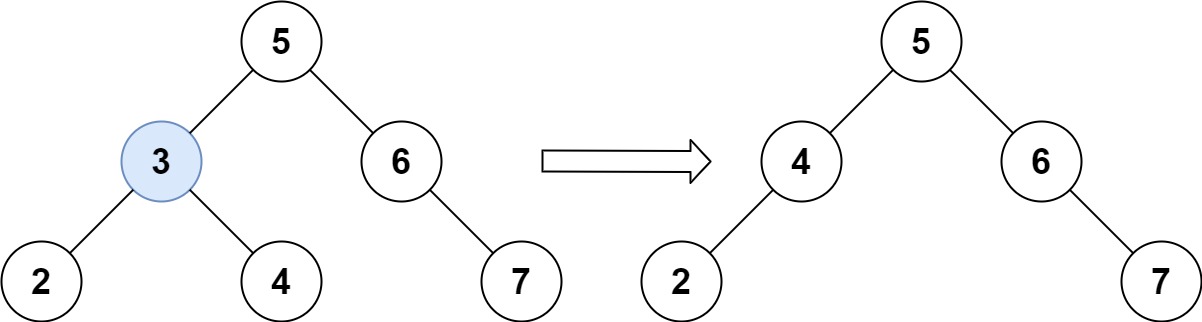
**450. Delete Node in a BST**

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return *the****root node reference****(possibly updated) of the BST*.

Basically, the deletion can be divided into two stages:

1. Search for a node to remove.
2. If the node is found, delete the node.

**Example 1:**



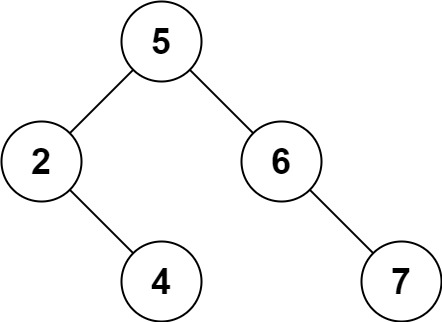
**Input:** root = [5,3,6,2,4,null,7], key = 3

**Output:** [5,4,6,2,null,null,7]

**Explanation:** Given key to delete is 3. So we find the node with value 3 and delete it.

One valid answer is [5,4,6,2,null,null,7], shown in the above BST.

Please notice that another valid answer is [5,2,6,null,4,null,7] and it's also accepted.

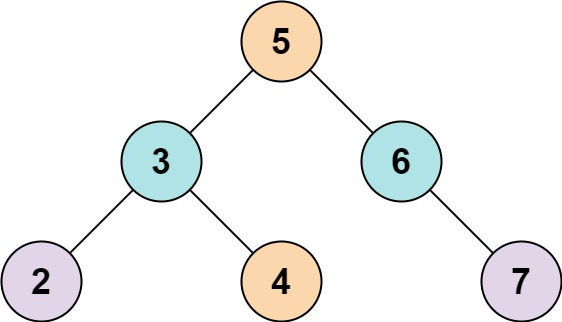


**Q: Find distance between two nodes in BST**

**653. Two Sum IV - Input is a BST**

Given the root of a Binary Search Tree and a target number k, return *true if there exist two elements in the BST such that their sum is equal to the given target*.

**Example 1:**



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

**Output:** true

**GFG: Inorder predecessor and successor for a given key in BST**

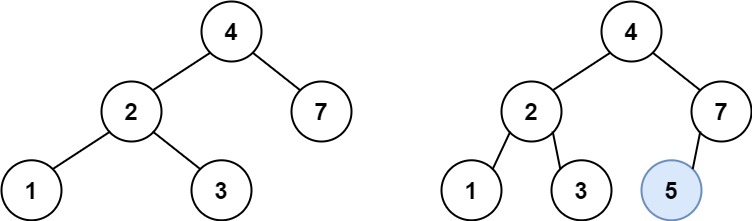
There is BST given with root node with key part as integer only. You need to find the inorder successor and predecessor of a given key. In case the given key is not found in BST, then return the two values within which this key will lie

**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:**



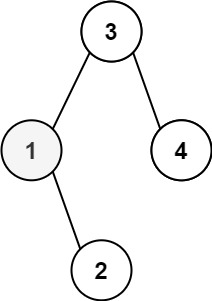
**Input:** root = [4,2,7,1,3], val = 5

**Output:** [4,2,7,1,3,5]

**230. Kth Smallest Element in a BST**

Given the root of a binary search tree, and an integer k, return *the* kth *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

**Largest BST in binary tree**

1st approach bruteforce.

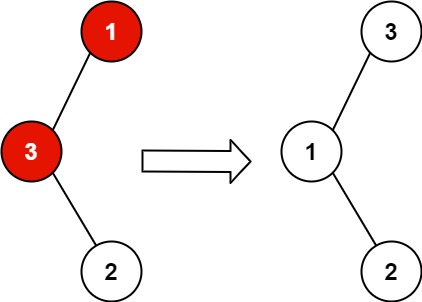
1) start with the root node and try to check if it is the valid BST. If it is then send this root node to other method which will give the count of the node in that BST. Do this for all the node and return the max of it.

2) To be a BST, its root should be left.maxNode < root.data && root.data < right.minNode. here we need to compute first left subtree and then right subtree then check with root data. So we need to use post order traversal.

single node is always a BST. So it gives a clue that we will start our computation from leaf node and if condition left.maxNode < root.data && root.data < right.minNode satisfies then we will expand size.

**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.



1st approach brute force

1) traverse BST in any order and sort it.

2) again do the inorder traversal of the BST, and compare the node data with the array data. If it is not same then just then make array data as root data.

1) check if there is one violation or 2 violation…..if there is 2 violation then swap first and last violation

2) if there is only one violation then swap adj node of the violated node.