

binary search tree



December 5, 2020

Table of Contents

[Binary Search Tree (Background): 2](#_Toc58159282)

[Introduction to Binary Search Trees 2](#_Toc58159283)

[Properties 2](#_Toc58159284)

[Searching a Key 2](#_Toc58159285)

[Insertion of Key 3](#_Toc58159286)

[Deletion of Key 4](#_Toc58159287)

[Problems 8](#_Toc58159288)

[1. Print Common Nodes in two BSTs 8](#_Toc58159289)

[2. Lowest Common Ancestor in a BST 10](#_Toc58159290)

[3. Print BST elements in given range 12](#_Toc58159291)

[4. Pair Sum in BST 12](#_Toc58159292)

[5. Smaller on Right 13](#_Toc58159293)

[6. Floor in BST 14](#_Toc58159294)

[7. Ceil in BST 15](#_Toc58159295)

[8. Find kth smallest in BST 16](#_Toc58159296)

[9. Preorder to postorder 19](#_Toc58159297)

[10. Merge two BST 's 20](#_Toc58159298)

[11. Fixing Two nodes of a BST 22](#_Toc58159299)

[Self-Balancing BST 23](#_Toc58159300)

[How do Self-Balancing BSTs Balance? 23](#_Toc58159301)

[Types of Self-Balancing BSTs 24](#_Toc58159302)

[AVL Trees 24](#_Toc58159303)

[Rotations in AVL Trees 25](#_Toc58159304)

[Insertion examples 26](#_Toc58159305)

[Red-Black Tree 29](#_Toc58159306)

[Introduction 29](#_Toc58159307)

[Rules for Red-Black Tree 29](#_Toc58159308)

[Why Red-Black Trees? 29](#_Toc58159309)

[Comparison with AVL 29](#_Toc58159310)

[Black Height of a Red-Black Tree 29](#_Toc58159311)

[Applications of Self-Balancing Binary Search Trees 30](#_Toc58159312)

[Problems 30](#_Toc58159313)

[1. Check for BST 30](#_Toc58159314)

[2. Find closest element in BST 31](#_Toc58159315)

[3. Convert level-order traversal to BST 31](#_Toc58159316)

[4. Count BST nodes that lie in the given range 33](#_Toc58159317)

[TreeSet and TreeMap in Java 34](#_Toc58159318)

[Problems on TreeSet/TreeMap 35](#_Toc58159319)

[1. Vertical Traversal of binary tree 35](#_Toc58159320)

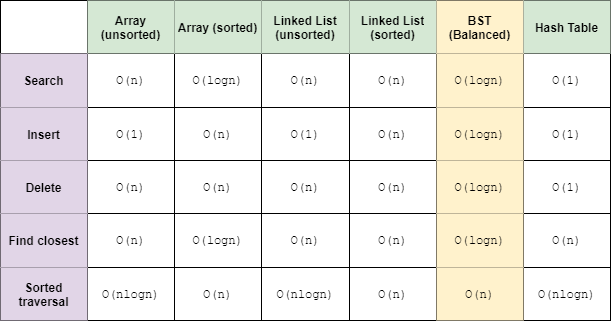
[2. Top view of Binary Tree 37](#_Toc58159321)

[3. Bottom view of Binary Tree 39](#_Toc58159322)

[4. Ceiling on left side in an Array 40](#_Toc58159323)

[5. Vertical sum 41](#_Toc58159324)

# Binary Search Tree (Background):



Balanced BSTs does all the operations in time. If BST is not balanced it requires O(height of BST) time on average (consider left skewed BST).

# Introduction to Binary Search Trees

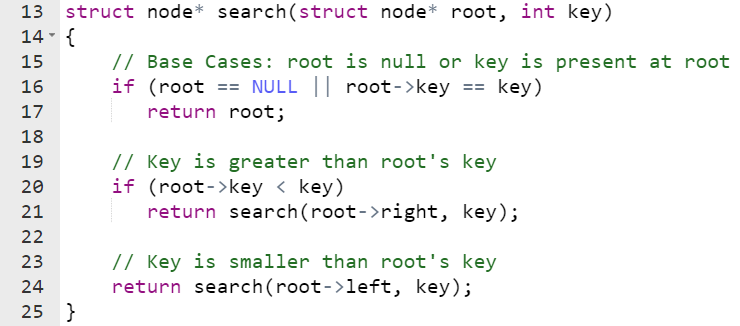
## Properties

* The left subtree of a node contains only nodes with keys lesser than or equal to the node's key.
* The right subtree of a node contains only nodes with keys greater than the node's key.
* The left and right subtree each must also be a binary search tree.
* There must be no duplicate nodes.
* In-order traversal of BST gives sorted list.

The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast in comparison to normal Binary Trees. If there is no ordering, then we may have to compare every key to search a given key.

## Searching a Key

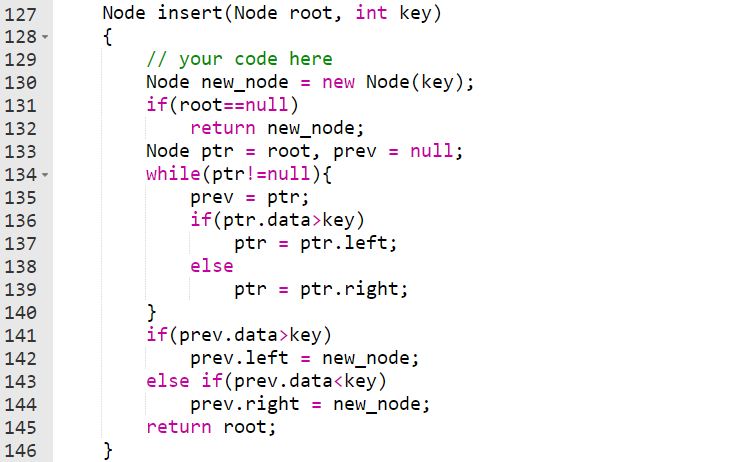
Using the property of Binary Search Tree, we can search for an element in O(h) time complexity where h is the height of the given BST.



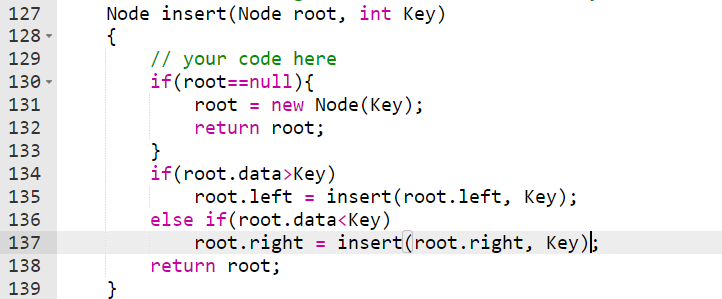
## Insertion of Key

Inserting a new node in the Binary Search Tree is always done at the leaf nodes to maintain the order of nodes in the Tree. The idea is to start searching the given node to be inserted from the root node till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

**Solution 1 (Iterative)**

****

**Solution 2 (Recursive)**

****

**Time Complexity:**The worst-case time complexity of search and insert operations is O(h) where **h**, is height of Binary Search Tree. In the worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

## Deletion of Key

The task is to search that node in the given BST and delete it from the BST if it is present.

When we delete a node, three cases may arise:

1. **Node to be deleted is leaf:** Simply remove from the tree. *(Super-simple)*
2. **Node to be deleted has only one child:** Copy the child to the node and delete the child. (Simple)



1. **Node to be deleted has two children:** We have two choices here either we use in-order successor or in-order predecessor of node to be deleted.

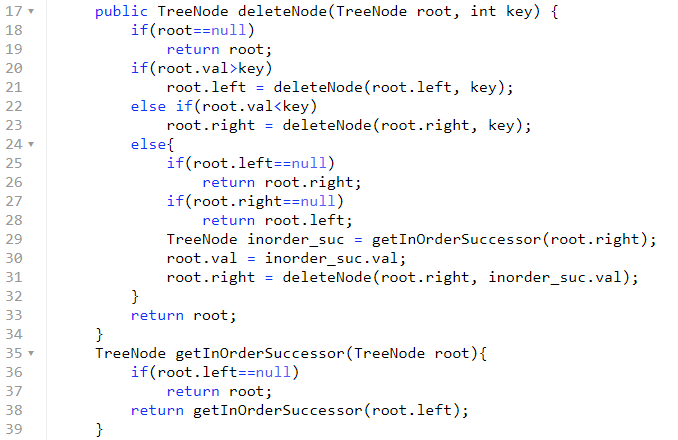
Find in-order successor of the node. Copy contents of the in-order successor to the node and delete the in-order successor. Note that in-order predecessor can also be used.



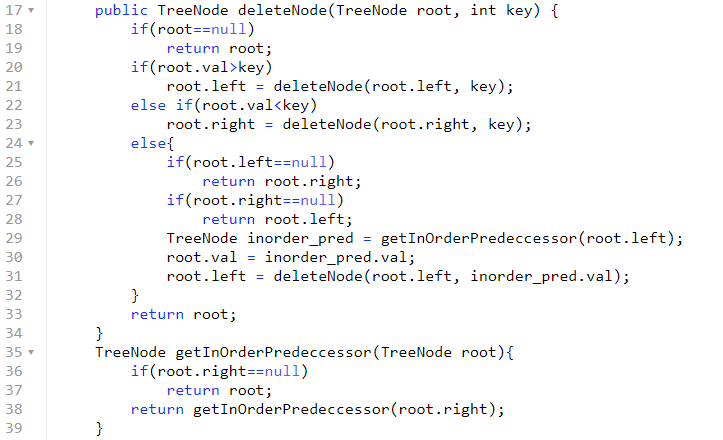
**Solution 1 (Recursive)**

**Using in-order successor**

1. Recursively find the node that has the same value as the key, while setting the left/right nodes equal to the returned subtree
2. Once the node is found, have to handle the below 4 cases
   1. node doesn't have left or right - return null
   2. node only has left subtree- return the left subtree
   3. node only has right subtree- return the right subtree
   4. node has both left and right - find the minimum value in the right subtree, set that value to the currently found node, then recursively delete the minimum value in the right subtree

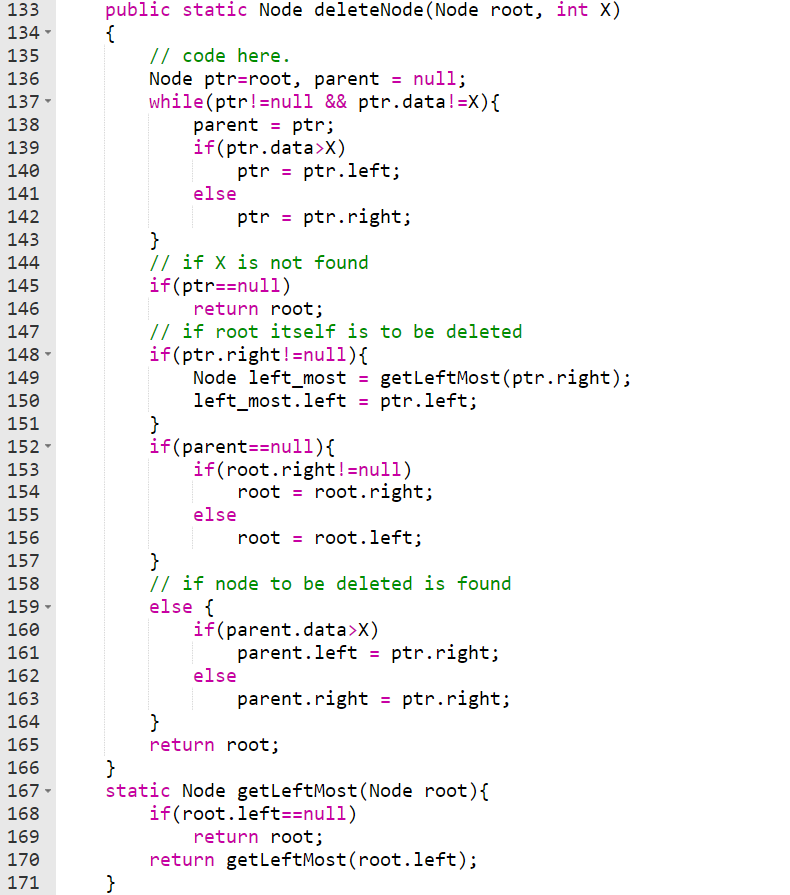


**Using in-order predecessor**

****

**Solution 2 (Iterative)**

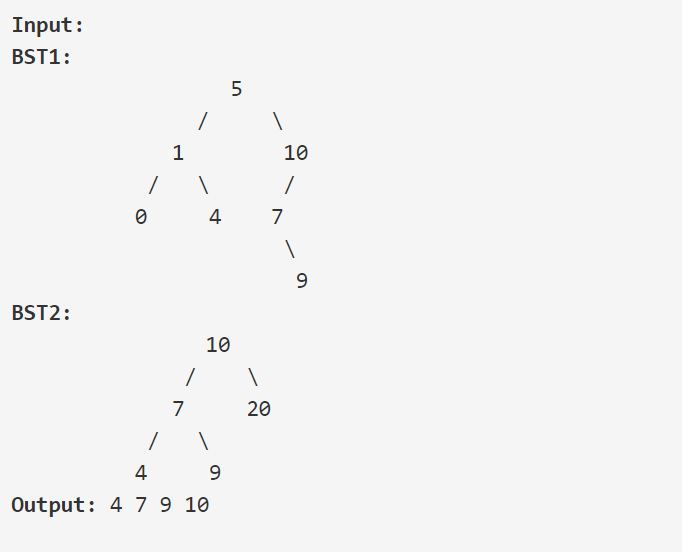
1. Find node with value X [logn time] keep track of parent
2. If node is not present in given root return root
3. Else, if node.right exists then append the node.left (left subtree) to the leftmost node of node.right (right subtree)
4. If node to be deleted is not root itself
   1. If node.right exist return root.right
   2. Else return root.left
5. Else
   1. If node is left child of parent then parent.left = ptr.right
   2. Else parent.right = ptr.right



# Problems

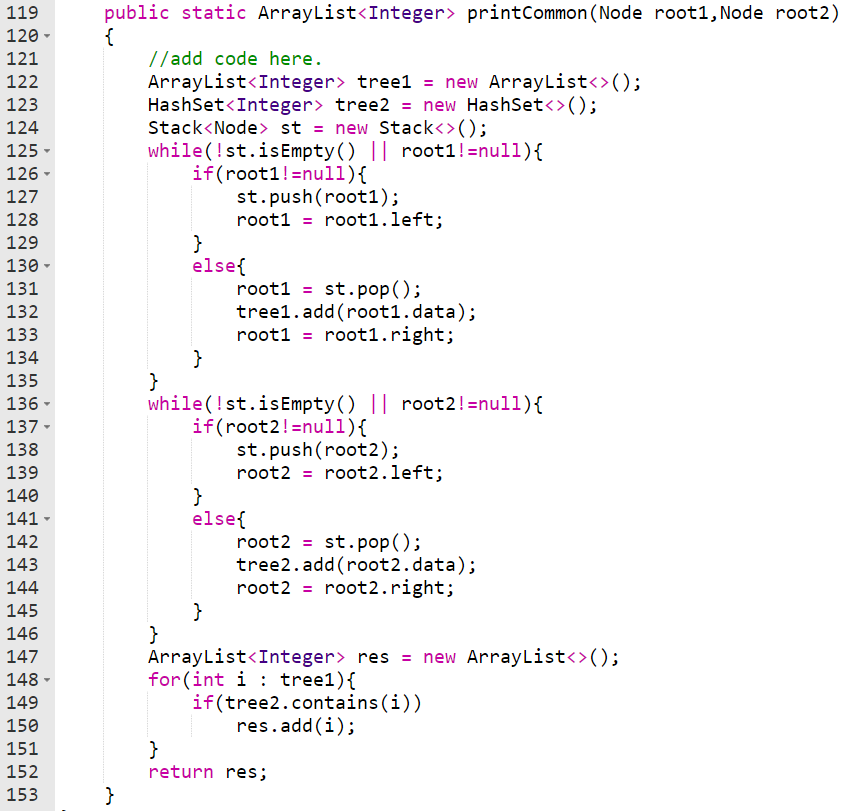
## Print Common Nodes in two BSTs

Given two Binary Search Trees (without duplicates). Find need to print the common nodes in them. In other words, find intersection of two BSTs.



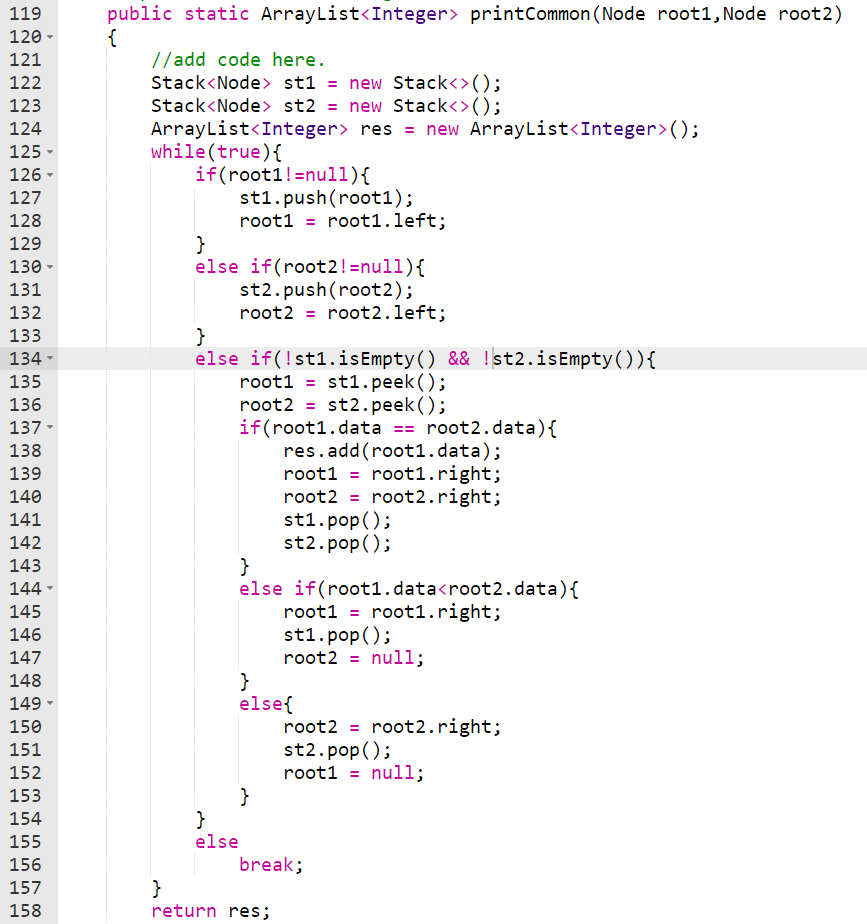
**Solution 1 (Uses extra space)**

1. Do in-order traversal of both the trees. Store one traversal in ArrayList and other in HashSet (this will do time complexity optimization).
2. Traverse ArrayList and check this element present in HashSet or not. If yes add it to res otherwise not.



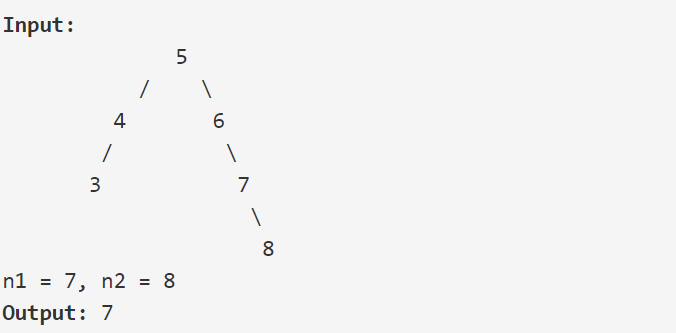
**[Better] Solution 2 (Uses only O(height pf BST) space)**

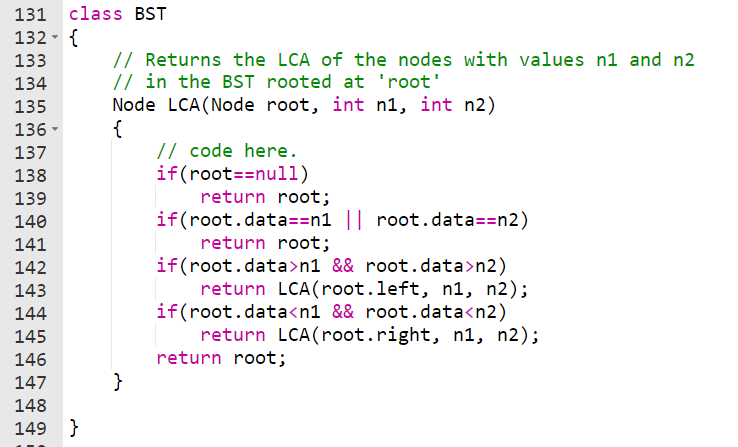
The idea is to use iterative inorder traversal. We use two auxiliary stacks for two BSTs. Since we need to find common elements, whenever we get same element during the inorder traversal, we print it. Else, if the elements are not same, we should accordingly go to right of first or second tree. Also, when you go for the right subtree if elements are not equal, then you should keep track of node of another subtree.

****

## Lowest Common Ancestor in a BST

Given a Binary Search Tree (with all values unique) and two node values. Find the Lowest Common Ancestors of the two nodes in the BST.

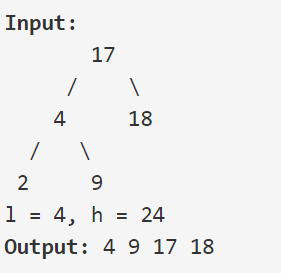


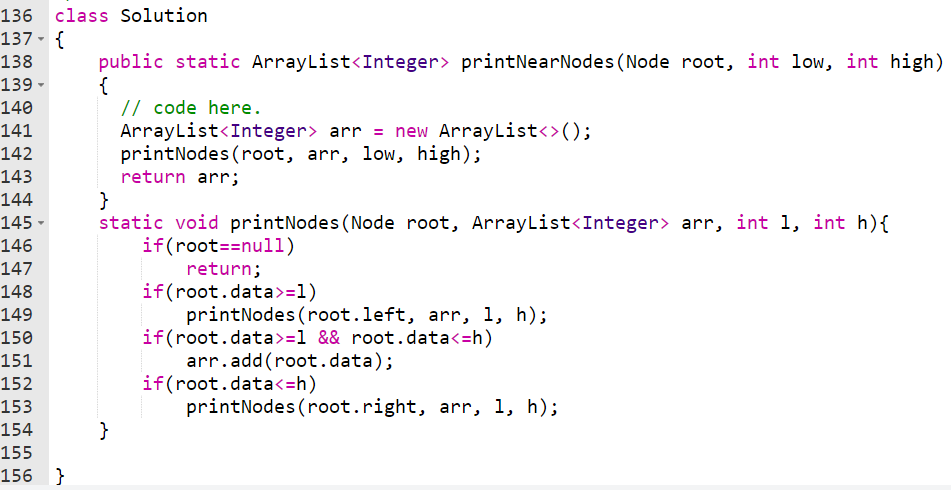


## Print BST elements in given range

Given a Binary Search Tree and a range. Find all the numbers in the BST that lie in the given range.

**Note**: Element greater than or equal to root go to the right side.





## Pair Sum in BST

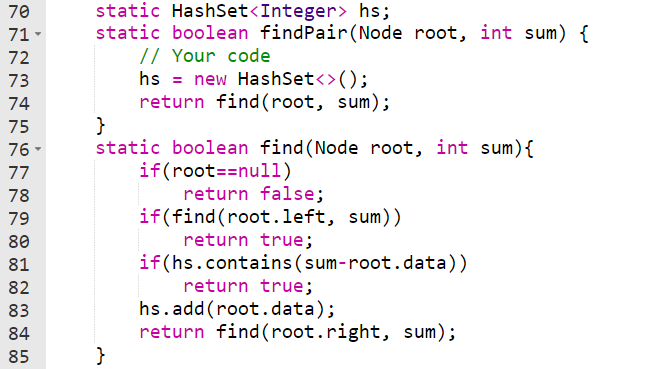
Given a BST and a number X. The task is to check if any pair exists in BST or not whose sum is equal to X.

One method is to use auxialiary array to store in-order traversal of BST. Then we can apply two pointer approach to find given sum.

Second one is,

1. Traverse tree inorder way to find if any pair exists which gives sum x.

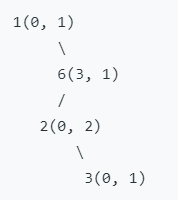
2. Use hashset to keep check is pair exist or not. If not then add the root data to hashset.



## Smaller on Right

You are given an integer array *nums* and you have to return a new *counts* array. The *counts* array has the property where counts[i] is the number of smaller elements to the right of nums[i].

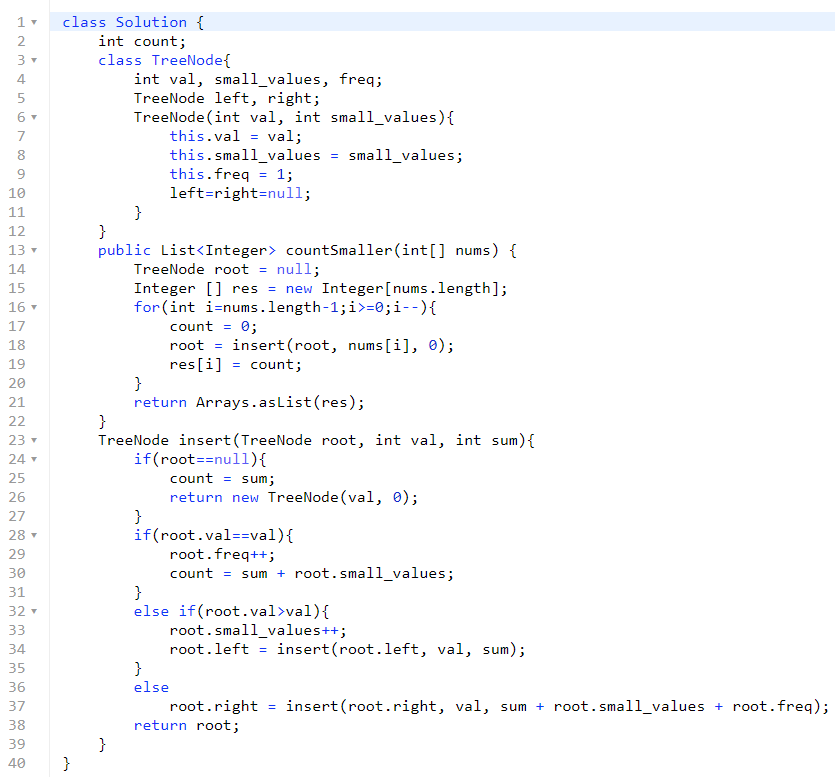
Every node will maintain a val, small\_value recording the total of number on it's left bottom side, freq counts the frequency. For example, [3, 2, 2, 6, 1], from back to beginning, we would have:



When we try to insert a number, the total number of smaller numbers would be adding freq and small\_values of the nodes where we turn right.

for example, if we insert 5, it should be inserted on the way down to the right of 3, the nodes where we turn right is 1(0,1), 2(0,2), 3(0,1), so the answer should be (0 + 1)+(0 + 2)+ (0 + 1) = 4

if we insert 7, the right-turning nodes are 1(0,1), 6(3,1), so answer should be (0 + 1) + (3 + 1) = 5

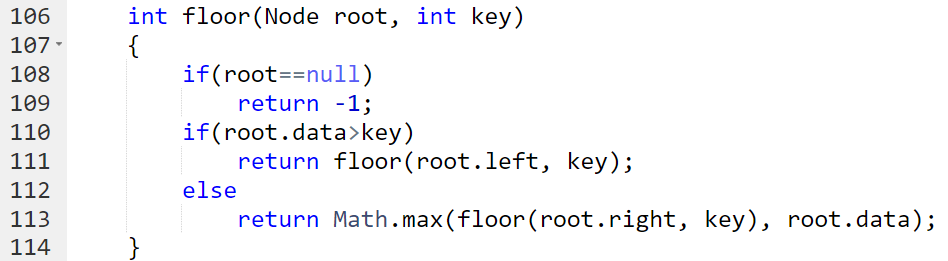


## Floor in BST

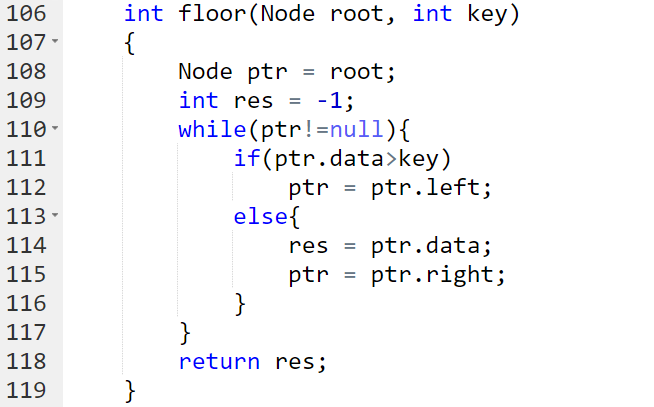
Given a Binary search tree and a value X, the task is to complete the function which will return the floor of x.

**Note:** Floor(X) is an element that is either equal to X or immediately smaller to X. If no such element exits return -1.

**Solution 1[Recursive]: Uses O(h) extra space and function call overhead**



**Solution 2[Iterative]: Uses O(1) extra space and O(h) time**

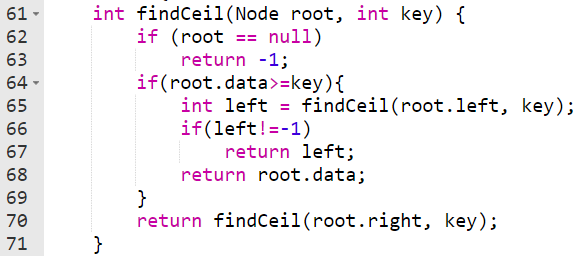


## Ceil in BST

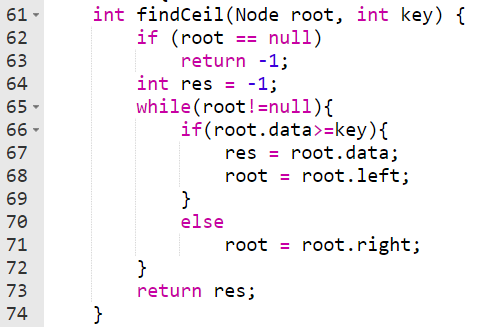
Given a BST and a number X. The task it to find Ceil of X.

**Note**: Ceil(X) is a number that is either equal to X or is immediately greater than X.

**Solution 1[Recursive]: Uses O(h) extra space and function call overhead**

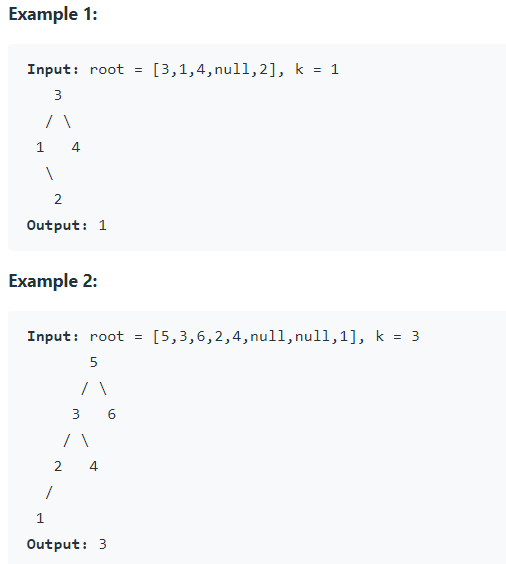


**Solution 2[Iterative]: Uses O (1) extra space and O(h) time**

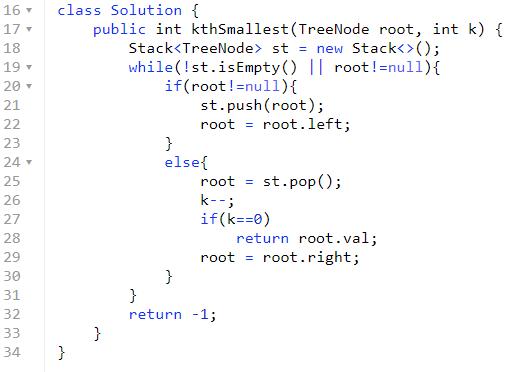
****

## Find kth smallest in BST

Given a binary search tree, write a function kthSmallest to find the kth smallest element in it.



**Solution 1 [DFS]: faster than solution 2**



Solution 2 [modify BST structure]: this solution is slower as compared to solution 1. But whenever there is requirement to frequent search for kthSmallest it gives in O(log n) time.

**Follow up:**

What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

Insert and delete in a BST were discussed last week, the time complexity of these operations is O(H), where H is a height of binary tree, and H = logN for the balanced tree.

Hence without any optimisation insert/delete + search of kth element has O(2*H*+*k*) complexity. How to optimise that?

That's a design question, basically we're asked to implement a structure which contains a BST inside and optimises the following operations:

Insert

Delete

Find kth smallest

Seems like a database description, isn't it? Let's use here the same logic as for LRU cache design, and combine an indexing structure (we could keep BST here) with a double linked list.

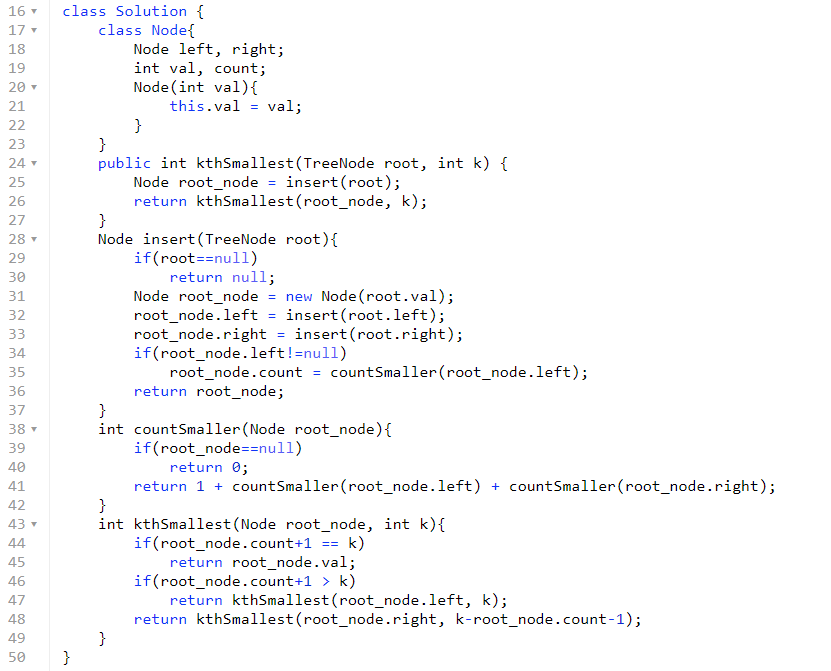
Such a structure would provide:

O(*H*) time for the insert and delete.

O(*k*) for the search of kth smallest.

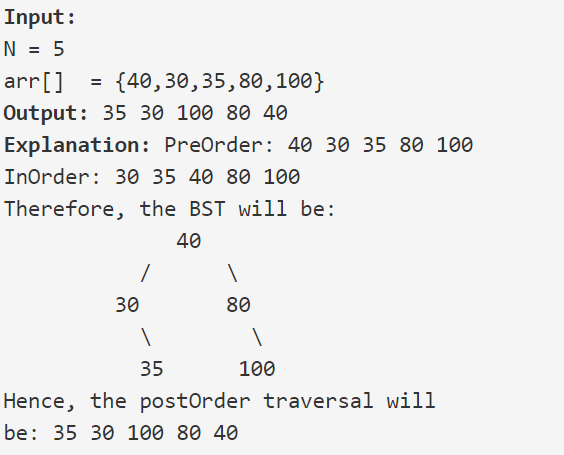
The idea is to maintain class member lcount which keeps track of number node on left subtree.

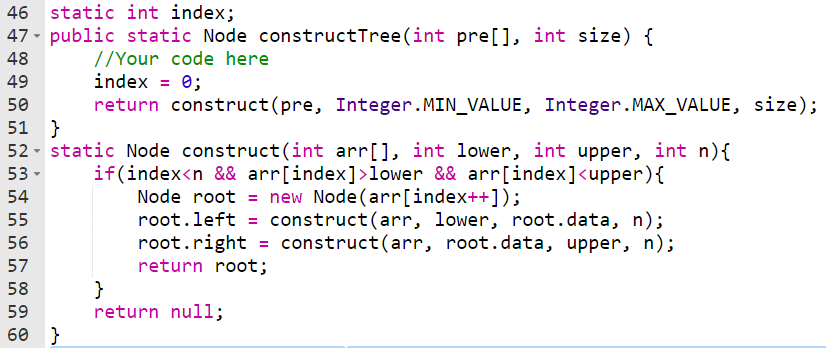
**Solution 2:**



## Preorder to postorder

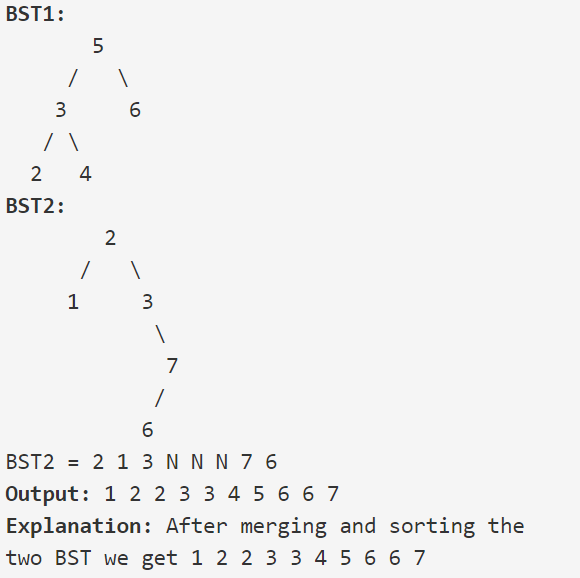
Given an array arr[] of N nodes representing preorder traversal of BST. The task is to print its postorder traversal.

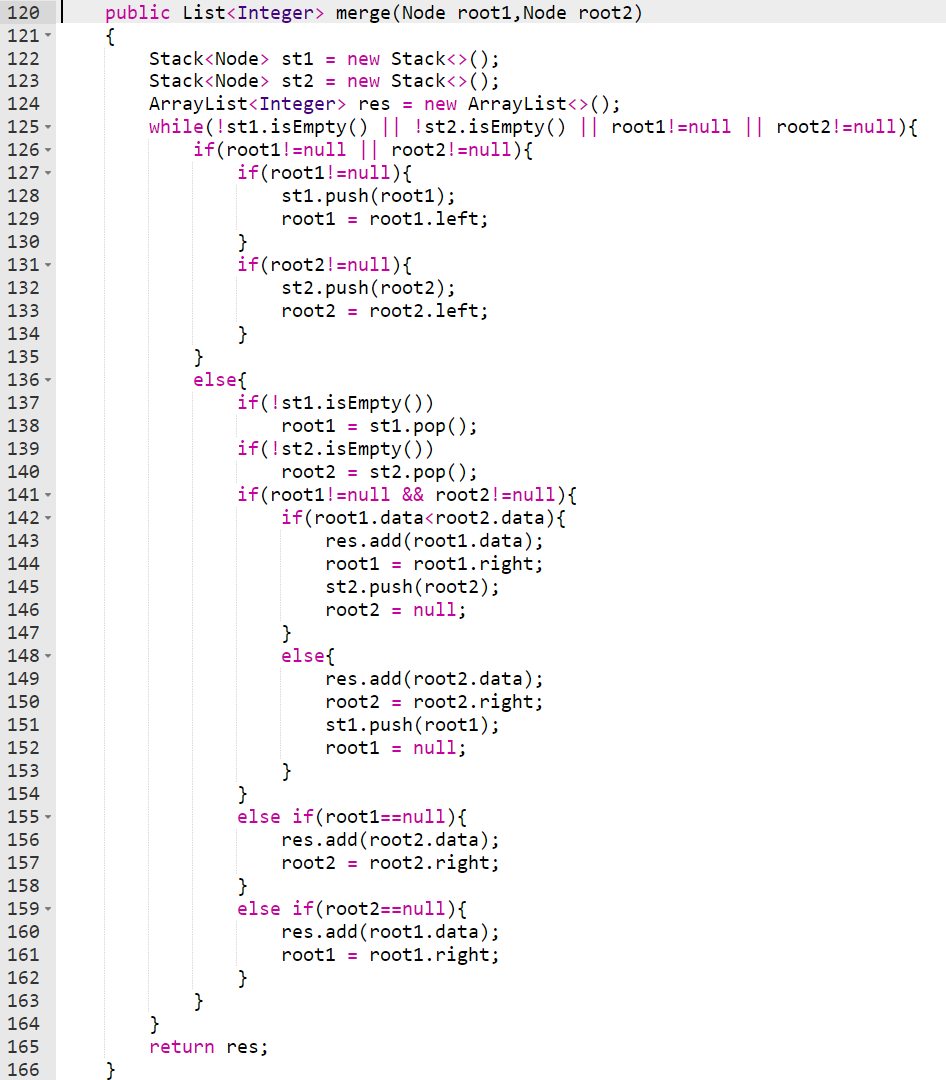




## Merge two BST 's

Given two BST, Return elements of both BSTs in sorted form.





## Fixing Two nodes of a BST

Two of the nodes of a Binary Search Tree (BST) are swapped. Fix (or correct) the BST by swapping them back. Do not change the structure of the tree.

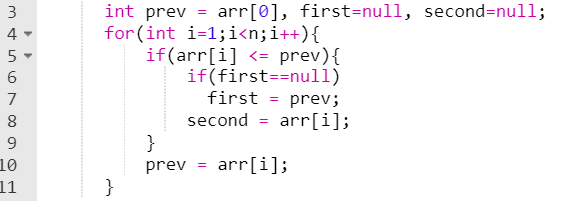
Note: It is guaranteed than the given input will form BST, except for 2 nodes that will be wrong.

**Solution:**

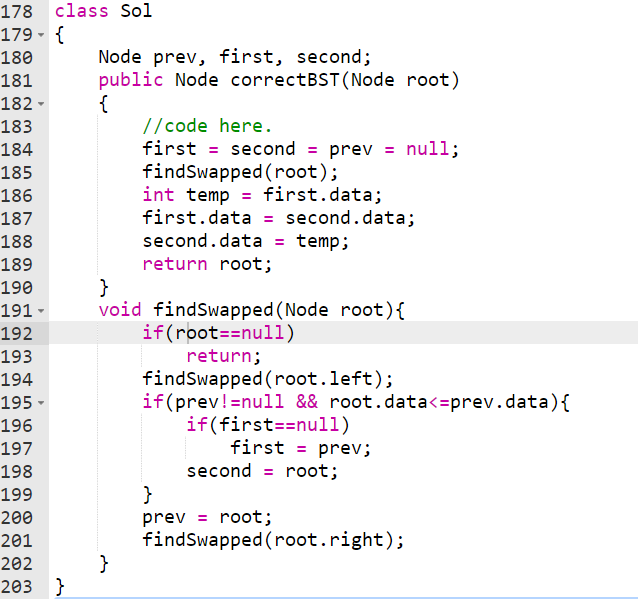
The idea is similar to finding two elements from sorted array which are miss-ordered. Given an sorted array which contains two element as misplaced.

[4, 60, 10, 20, 8, 80, 100] 🡪 [60, 8] are swapped

[4, 8, 10, 60, 20, 80, 100] 🡪 [60, 20] are swapped



The given problem reduces to this if we do in-order traversal of given BST. Inorder traversal will give us sorted list. On this sorted list we can apply above approach. But in below approach we don’t need to maintain arraylist.

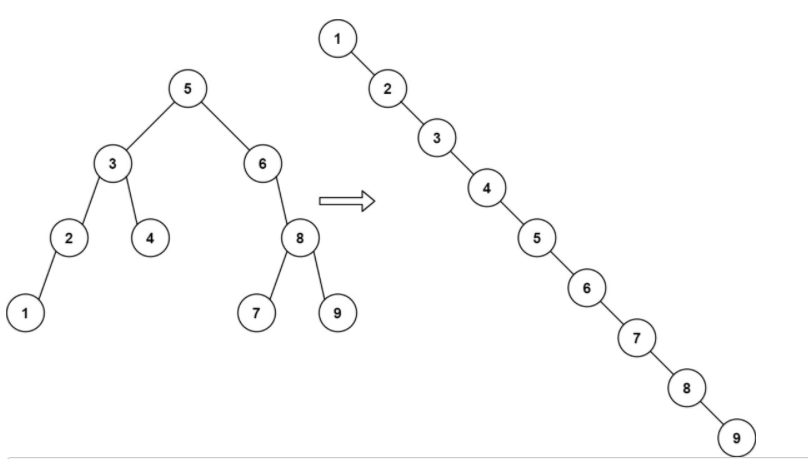


# Self-Balancing BST

In normal BST we can perform all the operations (search, insert, delete, floor, ceil, smaller and greater) in Big O(h) time. But if tree is not balanced sometimes it will take O(n) time.

Where, h: height of the BST, and n: number of nodes in BST

Suppose, we are inserting (stream of) data in BST in ascending/descending order we will get left/right skewed BST. In such BST it will take O(n) time search data.



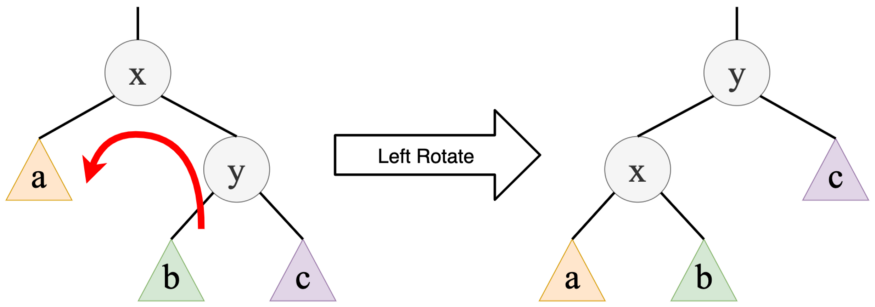
**Self-Balancing Binary Search Trees** are **height-balanced** binary search trees that automatically keeps height as small as possible when insertion and deletion operations are performed on tree. The height is typically maintained in order of so that all operations take θ () time on average.

## How do Self-Balancing BSTs Balance?

When it comes to self-balancing, BSTs perform rotations after performing insert and delete operations. Given below are the two types of rotation operations that can be performed to balance BSTs without violating the binary-search-tree property.

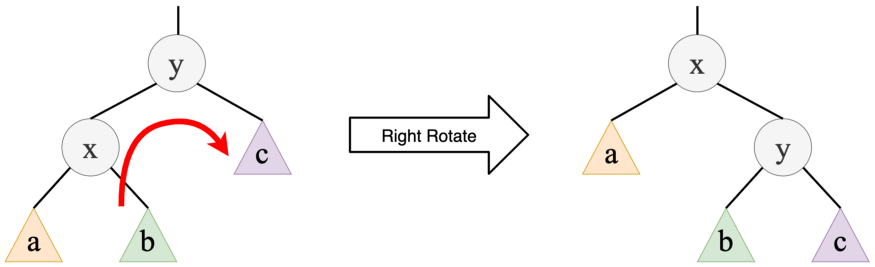
1. **Left rotation**

Left rotation on node x



1. **Right rotation**

Right rotation on node x

****

## Types of Self-Balancing BSTs

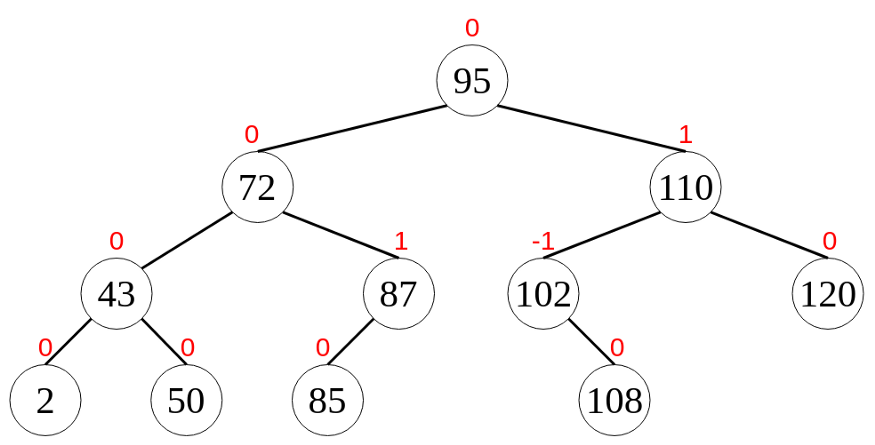
* AVL trees
* Red-black trees
* Splay trees
* Treaps

## AVL Trees

All the node in an AVL tree stores their own balance factor.

In an AVL tree, the balance factor of every node is either -1, 0 or +1. In other words, the difference between the height of the left subtree and the height of the right subtree cannot be more than 1 for all of the nodes in an AVL tree.

In Figure, the values in red colour above the nodes are their corresponding balance factors. You can see that the balance factor condition is satisfied in all the nodes of the AVL tree shown in Figure.



### Rotations in AVL Trees

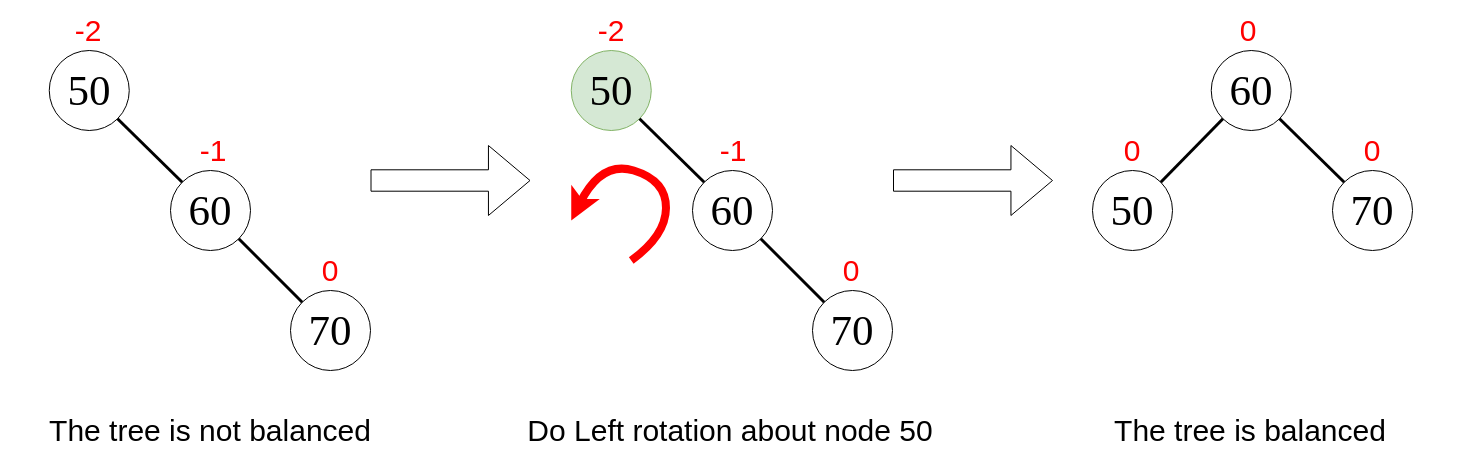
After performing insertions or deletions in an AVL tree, we have to check whether the balance factor condition is satisfied by all the nodes. If the tree is not balanced, then we have to do rotations to make it balanced.

Rotations performed on AVL trees can be of four main types that are grouped under two categories. They are,

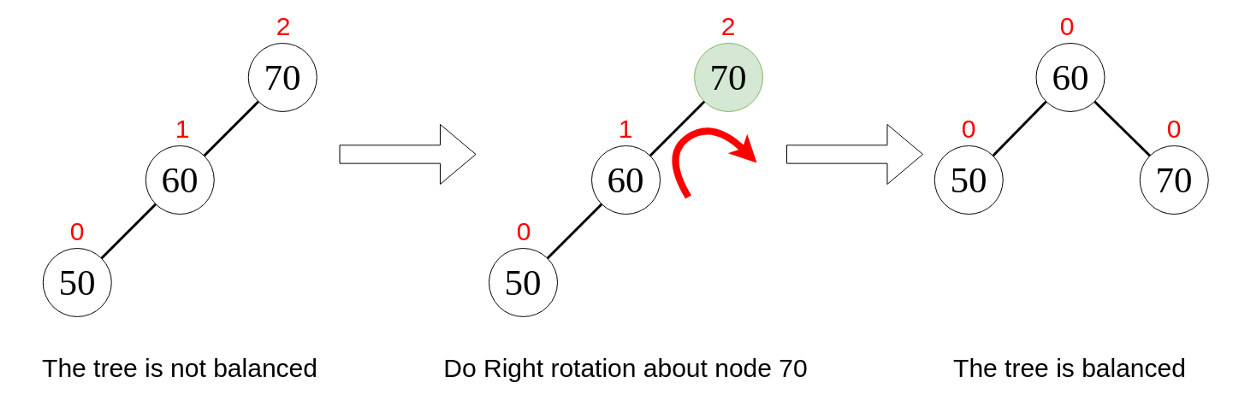
**Single rotations — Left (LL) Rotation and Right (RR) Rotation**

**Double rotations — Left Right (LR) Rotation and Right Left (RL) Rotation**

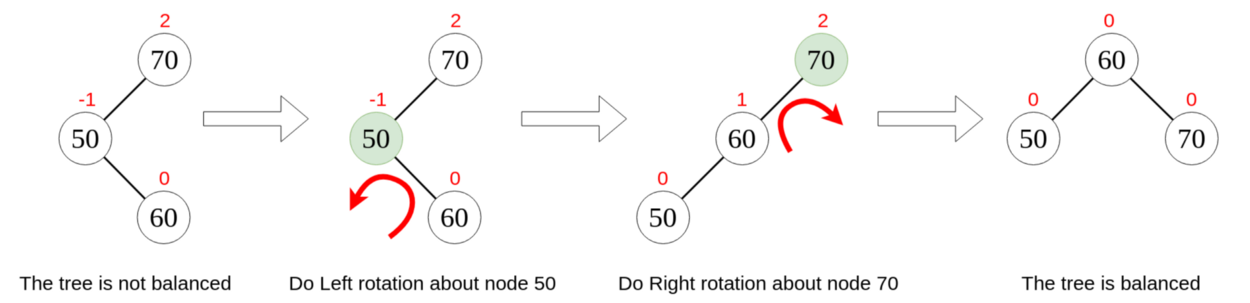
1. **Single Left Rotation (LL Rotation)**

****

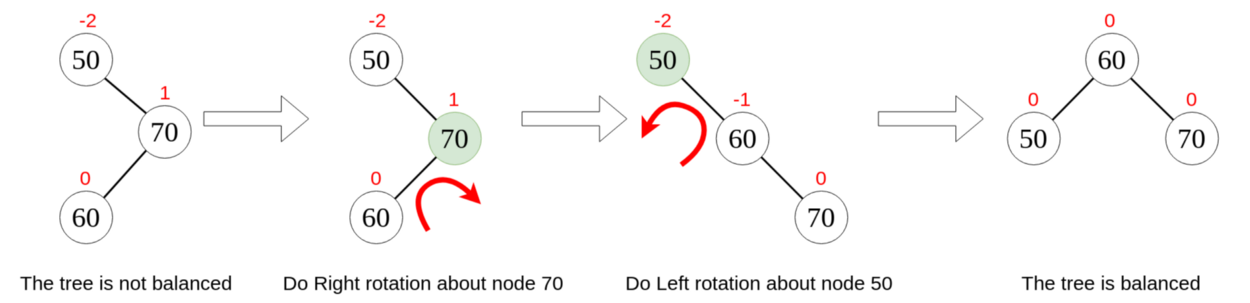
1. **Single Right Rotation (RR Rotation)**

****

1. **Left Right Rotation (LR Rotation)**

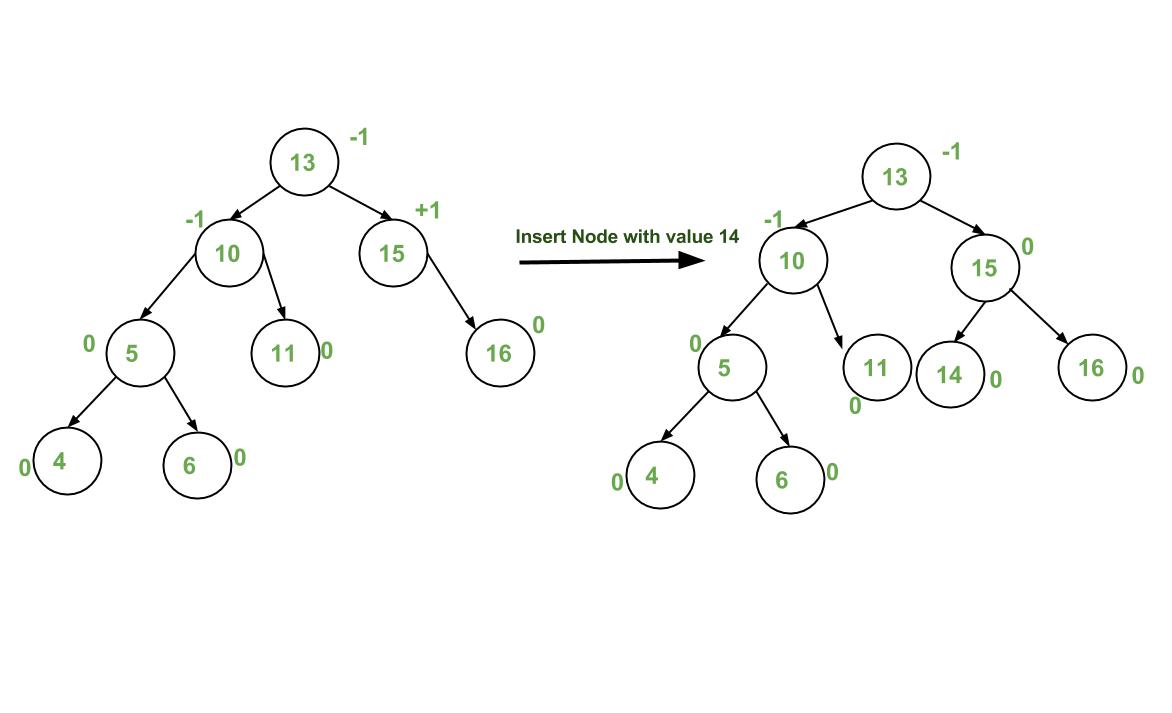
****

1. **Right Left Rotation (RL Rotation)**

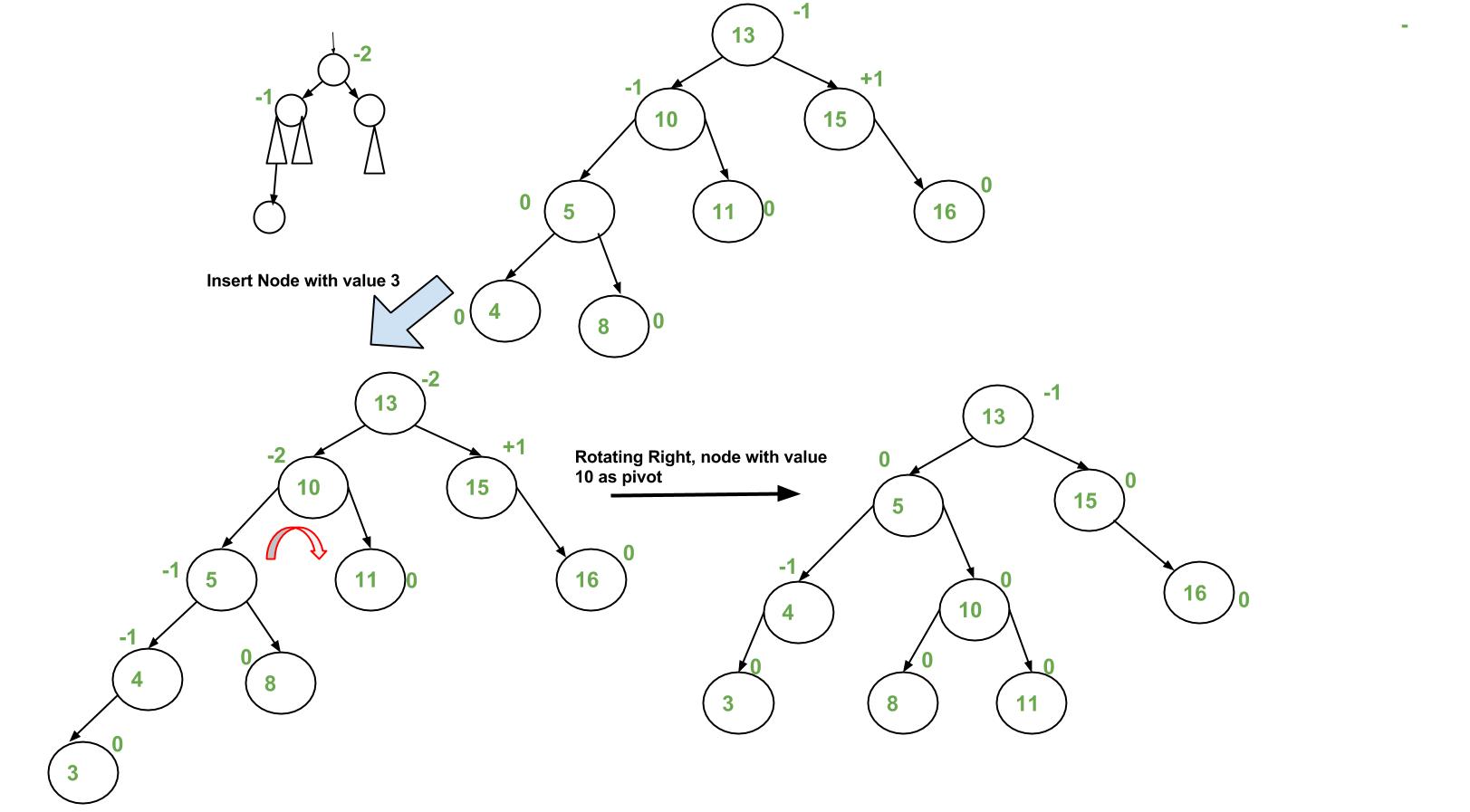
****

### Insertion examples

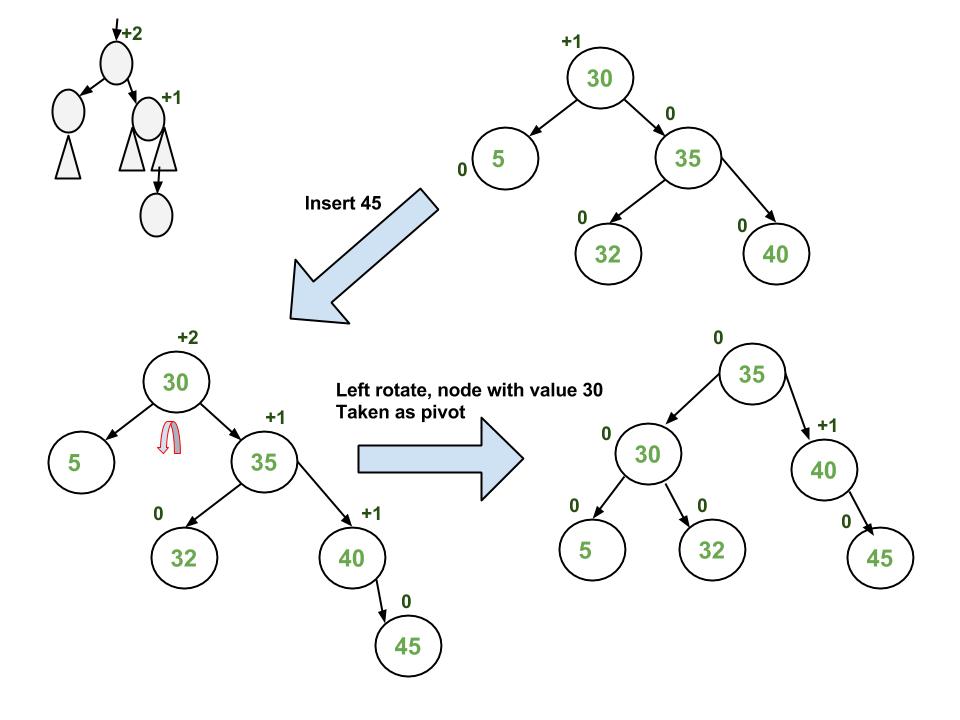
1. **No need to rotate BST**



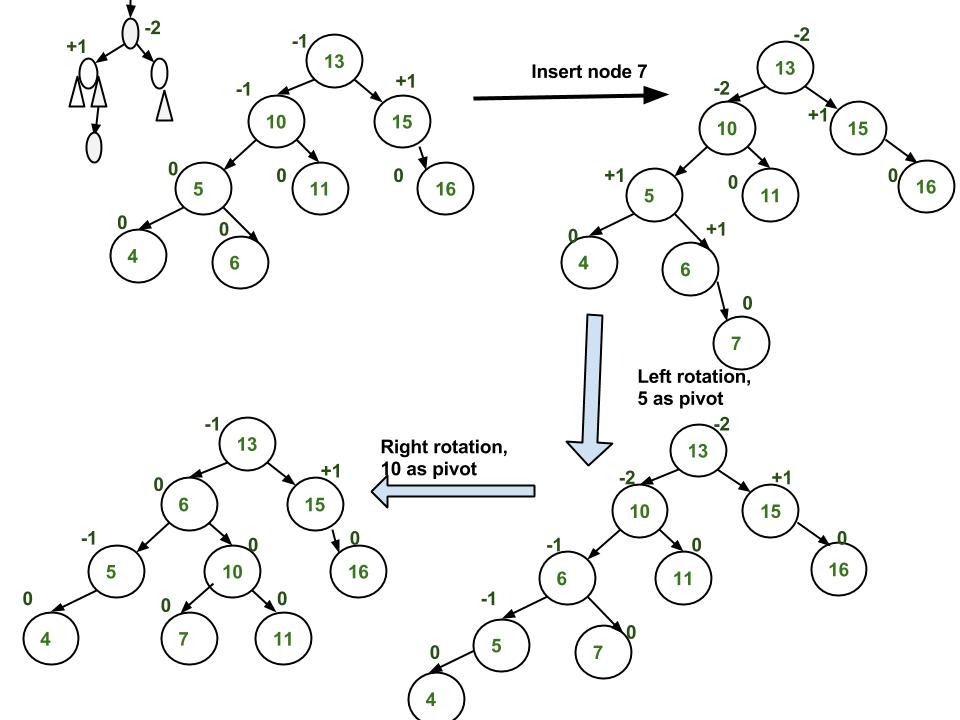
1. **Right rotate**



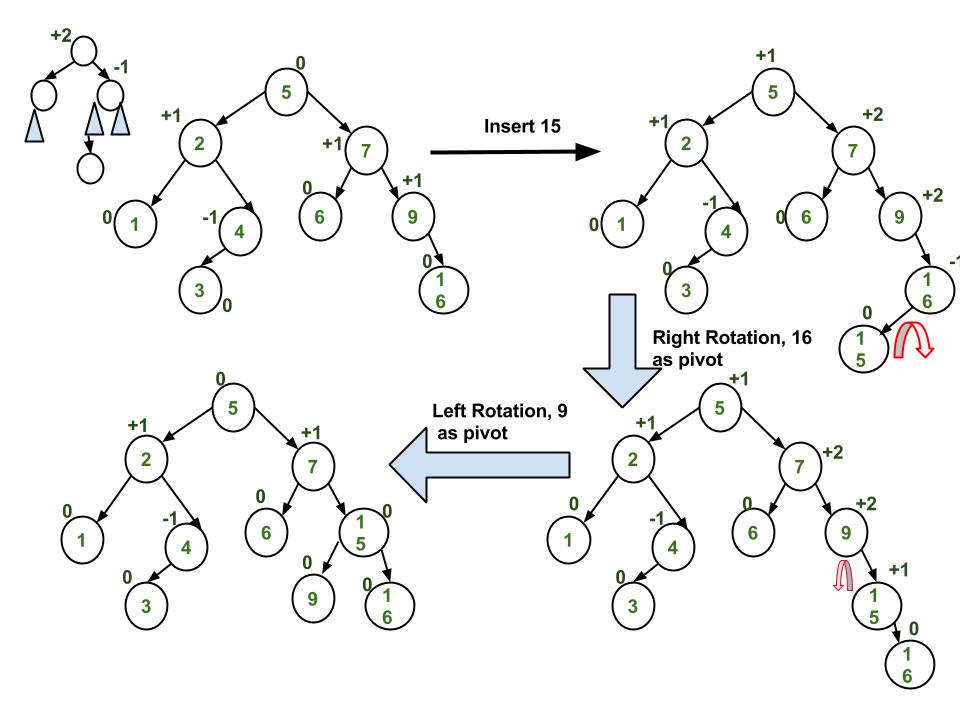
1. **Left rotate**

****

1. **Left Right rotate**

****

1. **Right rotate**

****

It seems that we are doing extra work in every insertion. But if we look closer, we’ll found out that in every insertion we only need to check valance factor all the ancestors of newly inserted node. And we can do this in O (), if find any of the ancestor with invalid balance factor we do rotation accordingly. And this rotation is constant work. Furthermore, to find balance factor of node we’ll require height of left-subtree and right-subtree, and finding height of any binary tree is also O () work.

## Red-Black Tree

### Introduction

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (red or black). These colours are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, but is good enough to reduce the searching time and maintain it around O (log n) time, where n is the total number of elements in the tree.

### Rules for Red-Black Tree

1. Every node has a colour either red or black.
2. The root of tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendant NULL node (leaf node) has the same number of black nodes.

### Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete, ... etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O (log n) after every insertion and deletion, then we can guarantee an upper bound of O (log n) for all these operations. The height of a Red-Black tree is always O (log n) where n is the number of nodes in the tree.

### Comparison with AVL

The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So, if your application involves frequent insertions and deletions, then Red-Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

### Black Height of a Red-Black Tree

Black height is the number of black nodes on a path from the root to a leaf. Leaf nodes are also counted black nodes. From the above properties 3 and 4, we can derive, a Red-Black Tree of **height h has black-height >= h/2.**

*Number of nodes from a node to its farthest descendant leaf is no more than twice as the number of nodes to the nearest descendant leaf.*

## Applications of Self-Balancing Binary Search Trees

**To maintain stream of data** in sorted order (stream of data coming in sorted order but not necessary to be sorted).

**To implement doubly ended priority queue.** Singly ended priority queue can be implemented by Heap data structure. Singly ended priority queues gives either maximum or minimum in O(1) time. Self-balancing BSTs provide both maximum and minimum in O(1) time.

To solve problems like:

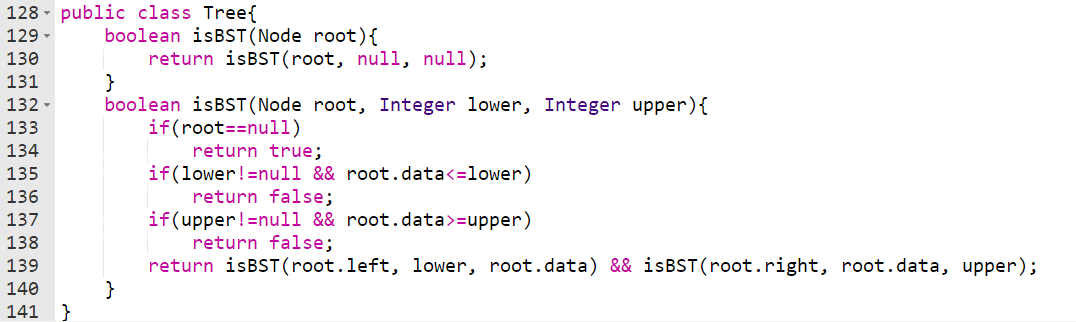
Count smaller/greater in stream

Find floor, ceil, greater, smaller, etc, … in a stream

# Problems

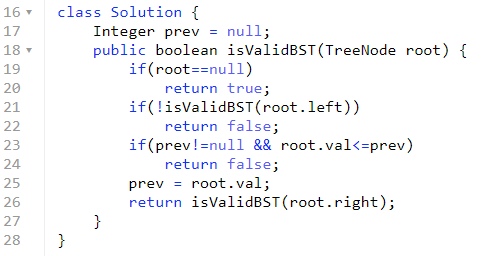
## Check for BST

Given a binary tree. Check whether it is a BST or not.



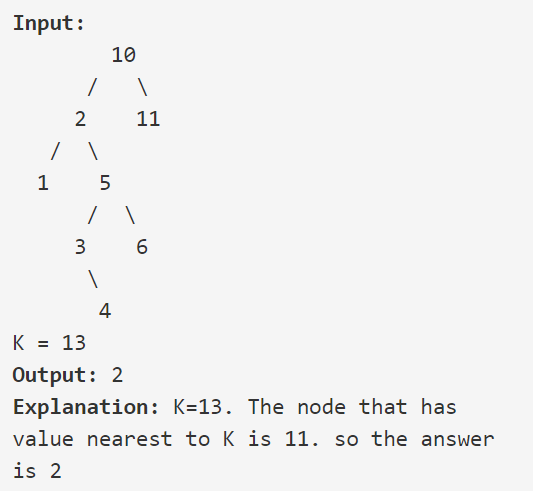
Another approach is to use in-order traversal of BST.

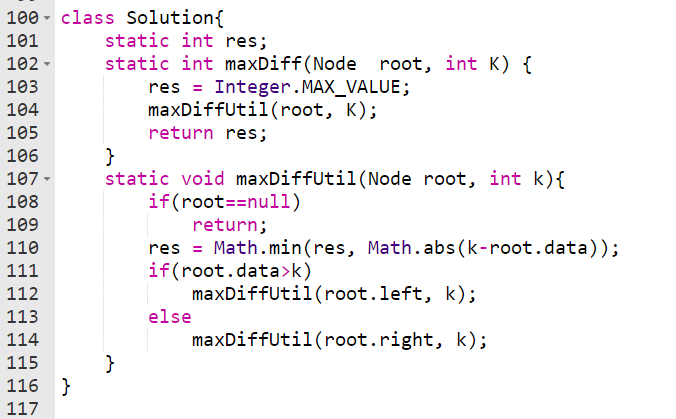
In-order traversal of BST gives sorted list of elements. There are two solution to use in-order traversal: One thing we can do is do in-order traversal of BST and check whether the returned list of elements is sorted or not. Second way is to maintain global variable prev.



## Find closest element in BST

Given a BST and an integer. Find the least absolute difference between any node value of the BST and the given integer.

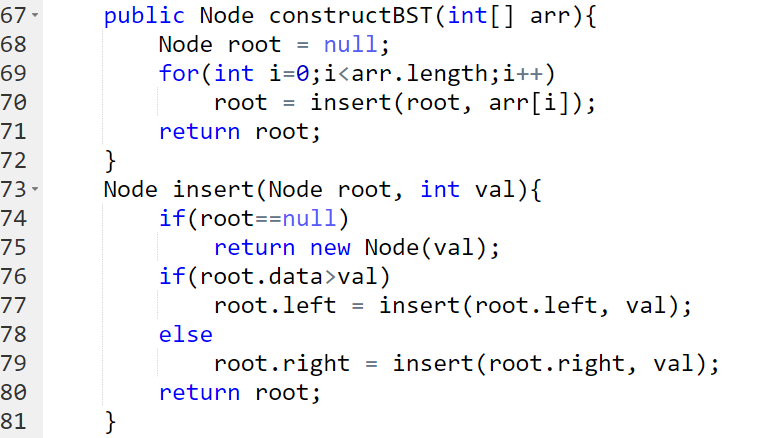




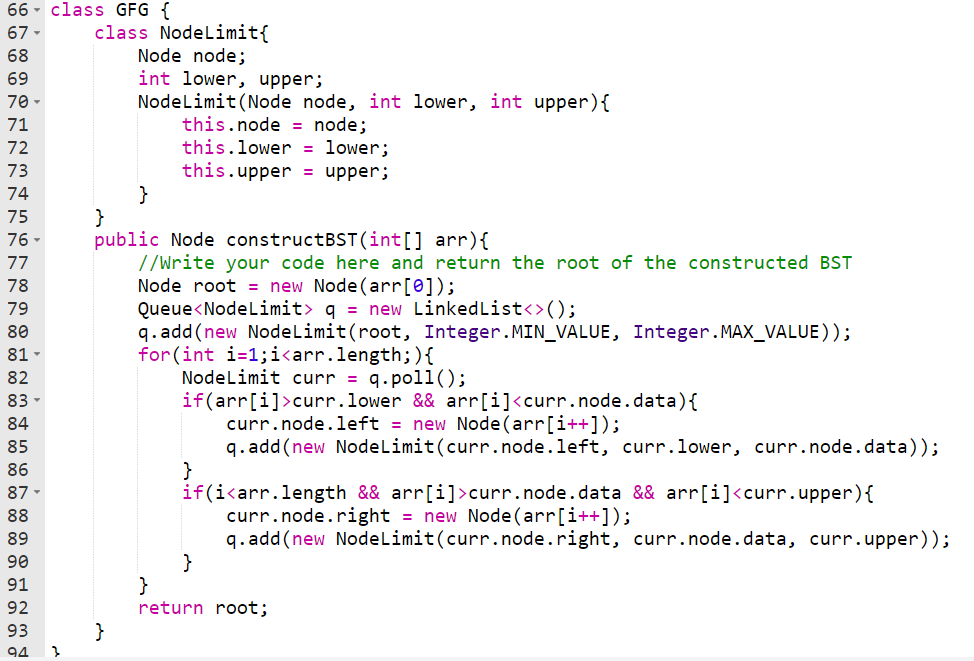
## Convert level-order traversal to BST

Given an array of size N containing level order traversal of a BST. The task is to complete the function constructBst(), that construct the BST (Binary Search Tree) from its given level order traversal.

**Solution 1[Recursive]: Uses O(h) extra space (function call overhead) and O () time**

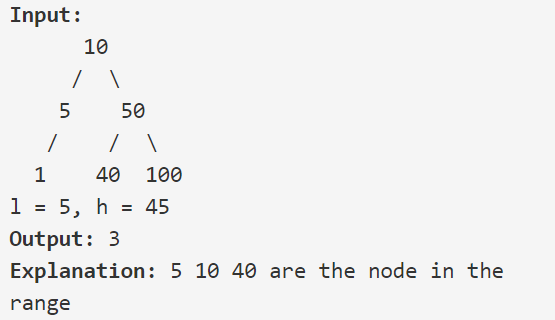
****

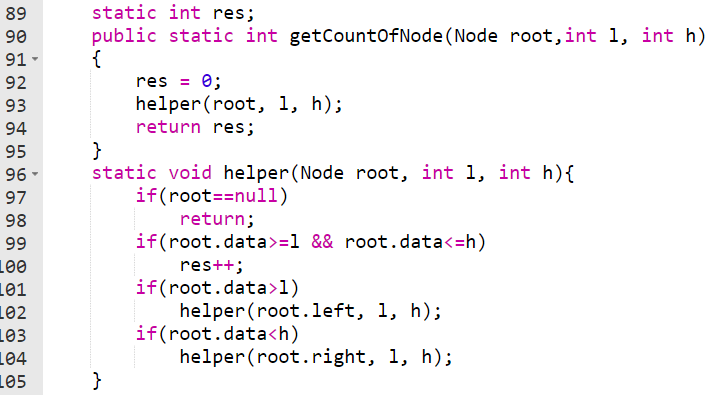
**Solution 2[BFS]: Uses O(N) extra space (maintain queue) and O (n) time**

****

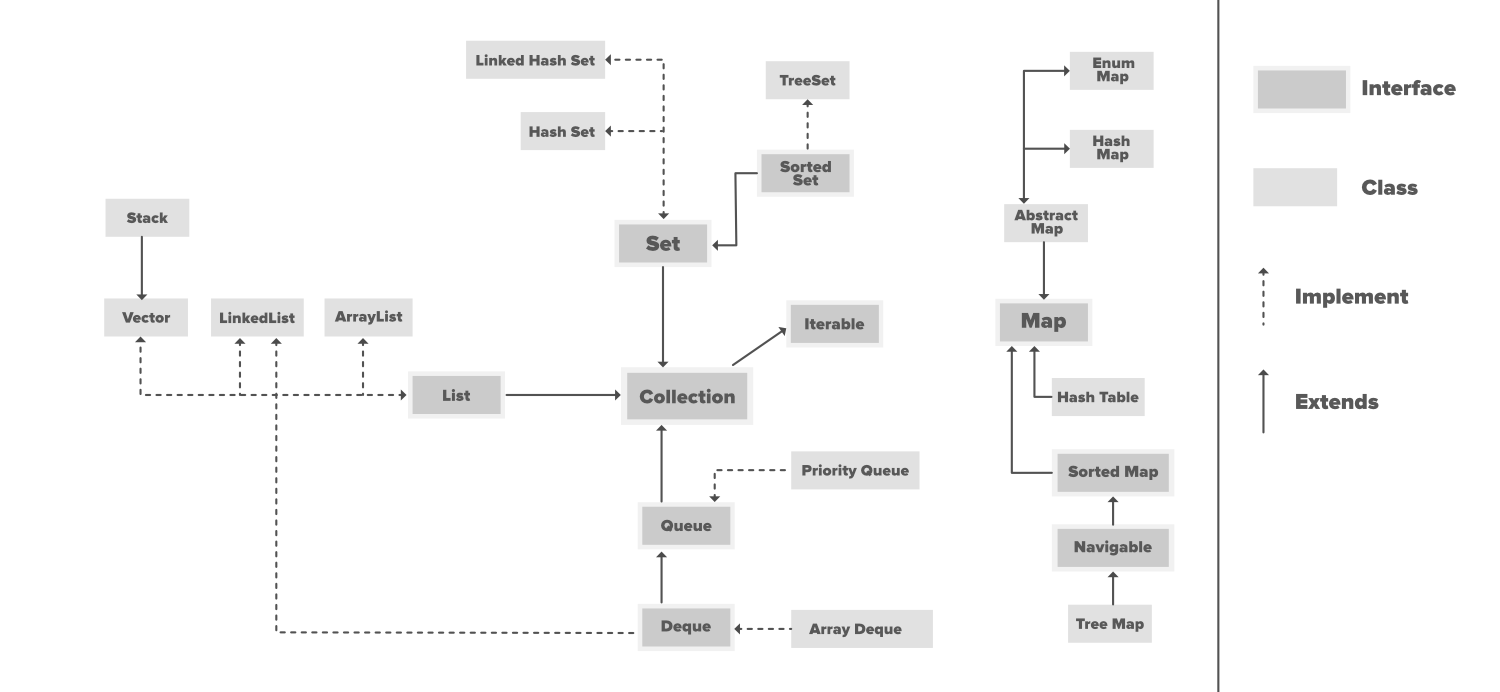
## Count BST nodes that lie in the given range

Given a Binary Search Tree (BST) and a range l-h(inclusive), count the number of nodes in the BST that lie in the given range.





# TreeSet and TreeMap in Java



TreeSet and TreeMap both are self-balancing BSTs, implemented using red-black binary tree. It is similar to HashSet and HashMap.

It provides functionalities that are similar to HashSet/HashMap but in additional functionality it provides finding floor, ceiling, higher, and lower.

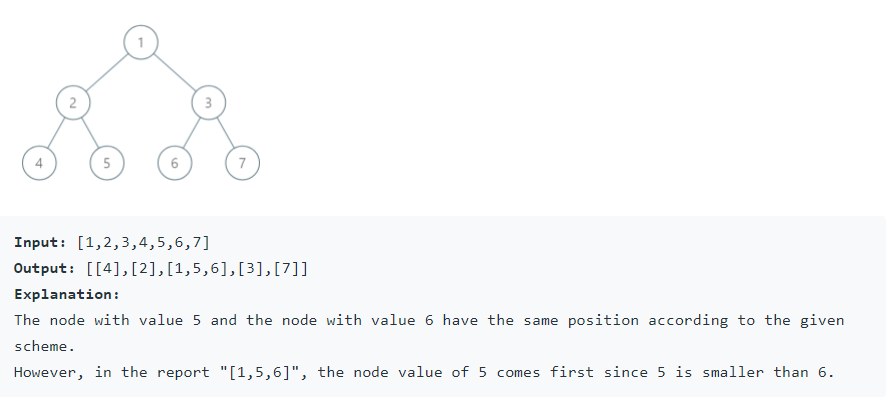
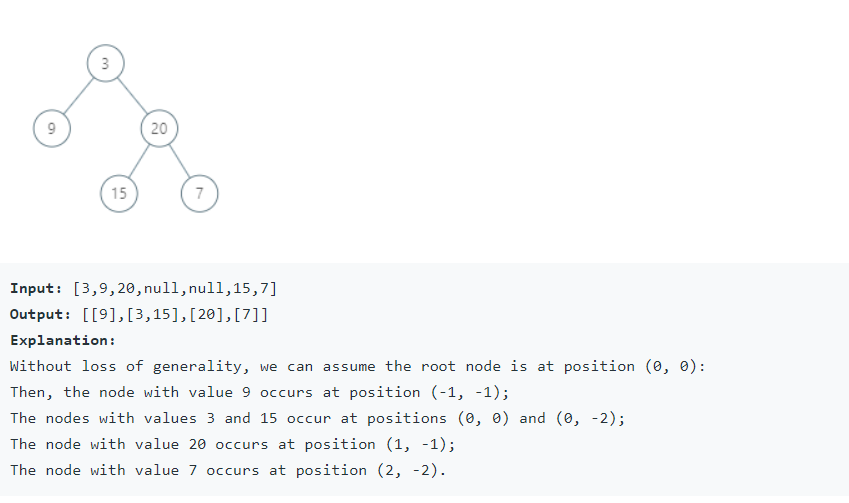
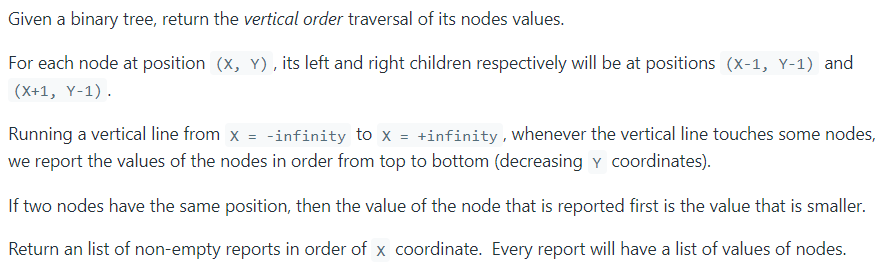
Similar to HashSet and HashMap TreeSet and TreeMap does not maintain inertion order but instead it **inserts element in sorted order (ascending)**.

Since it is implementation of self-balancing binary trees it performs operations like **insert, deleted, search, floor, ceiling, higher, lower in O ()** time.

TreeSet maintains only keys. While TreeMap gives functionalities to store value corresponding to key (key-value pair).

## Problems on TreeSet/TreeMap

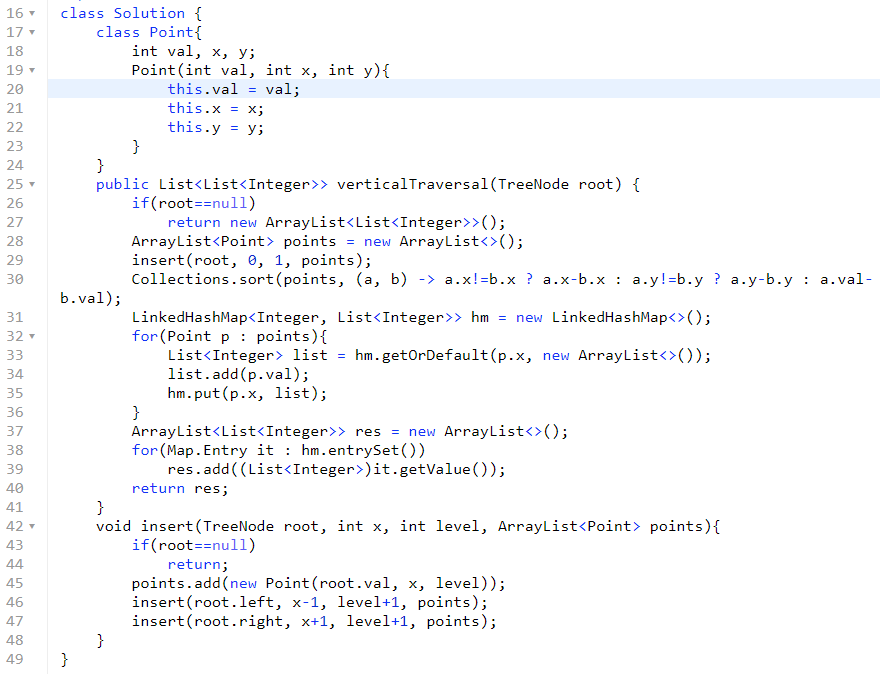
### Vertical Traversal of binary tree



**Solution 1 [DFS]:**

Time complexity: --> O(insert in arraylist + sort arraylist + insert in hashmap from sorted list + insert in resultant list from hashmap)

Space complexity: --> O(arraylist + recursion call stack + hashmap + resultant arraylist)

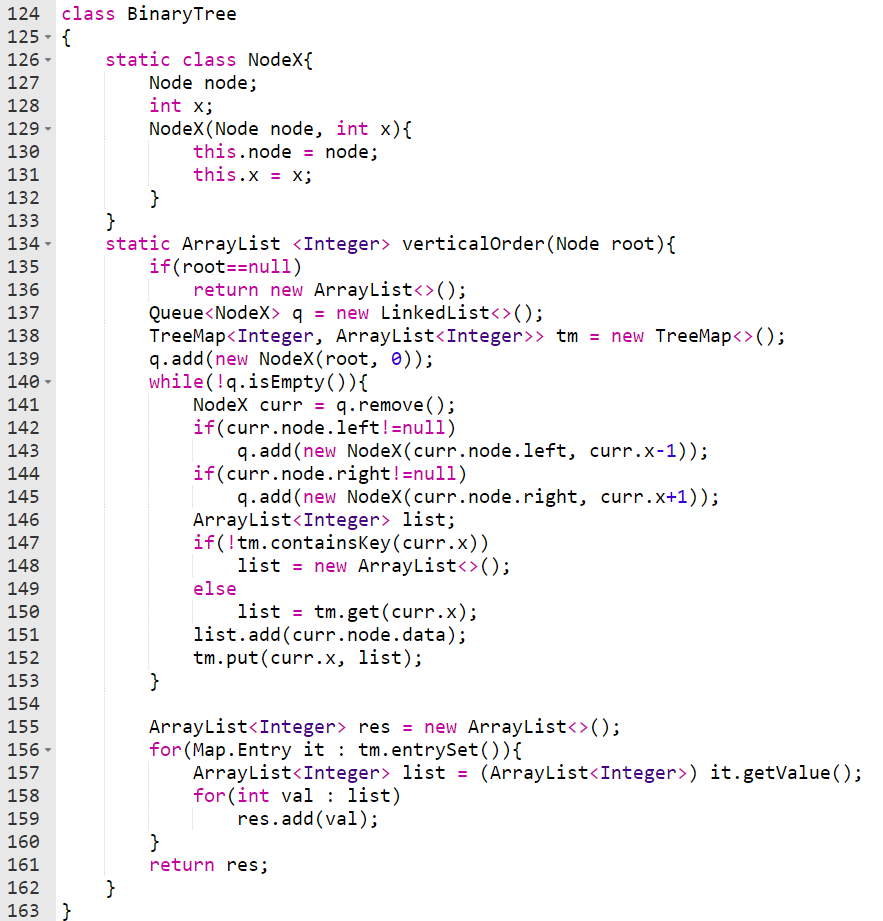


**Solution 2 [BFS]: Uses TreeMap**

Time complexity: --> O(tree traversal with accessing treemap + insert into resultant list from treemap)

Inserting/searching from treemap takes time.

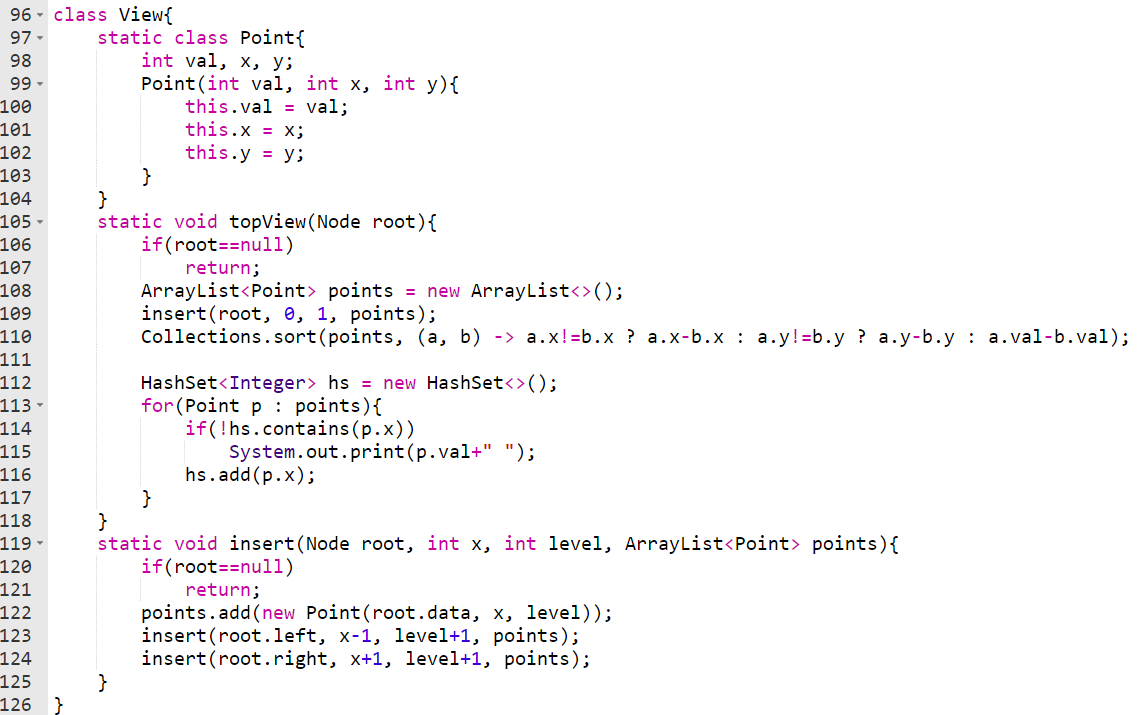
Space complexity:



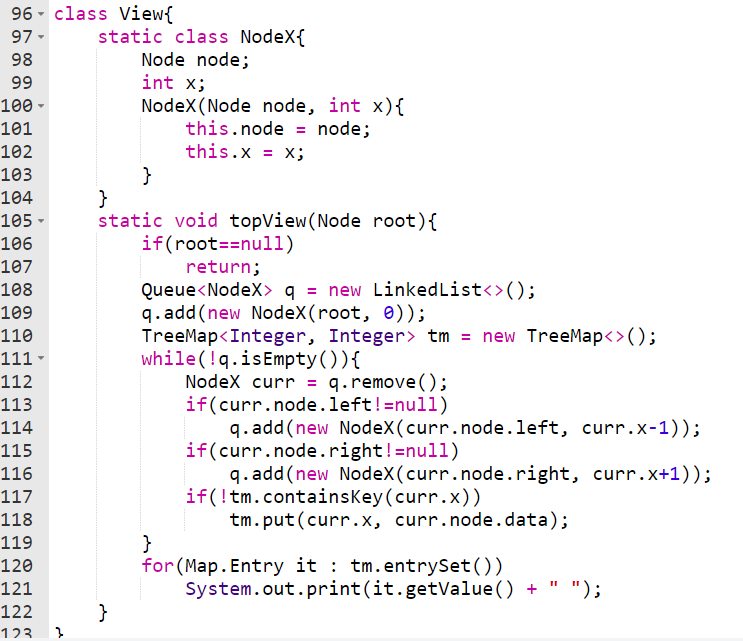
### Top view of Binary Tree

Given below is a binary tree. The task is to print the top view of binary tree. Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. For the given below tree.

**Solution 1[using HashMap]: extra space: function call overhead and O () time (insert in arraylist + to sort according to position of node + traverse hashmap).**

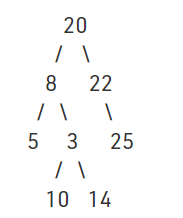


**Solution 2[using TreeMap]: extra space: Queue to do BFS and O () time (traversing tree and add it to queue + traversing treemap).**



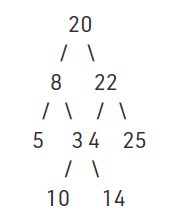
### Bottom view of Binary Tree

Given a binary tree, print the bottom view from left to right.

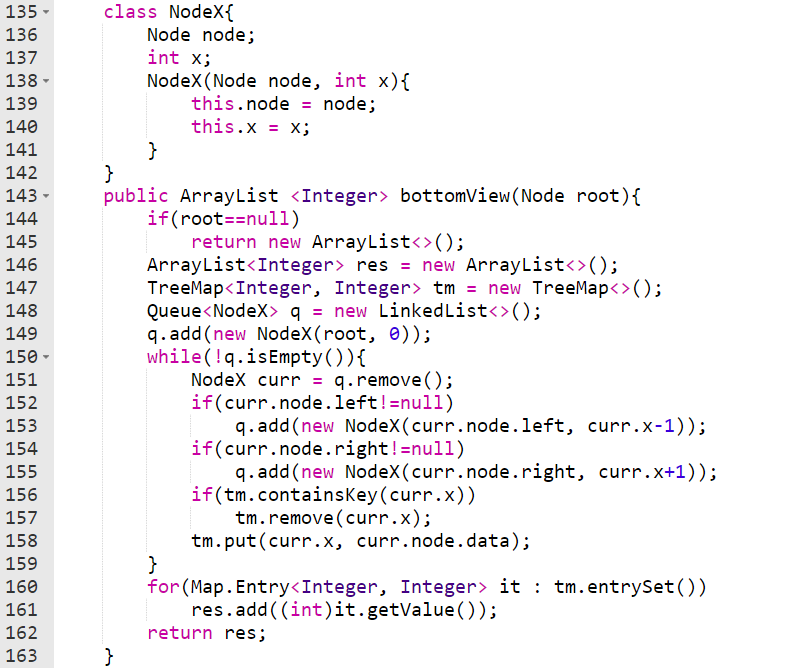


For the above tree, the bottom view is 5 10 3 14 25.

If there are multiple bottom-most nodes for a horizontal distance from root, then print the later one in level traversal. For example, in the below diagram, 3 and 4 are both the bottommost nodes at horizontal distance 0, we need to print 4.



For the above tree the output should be 5 10 4 14 25.

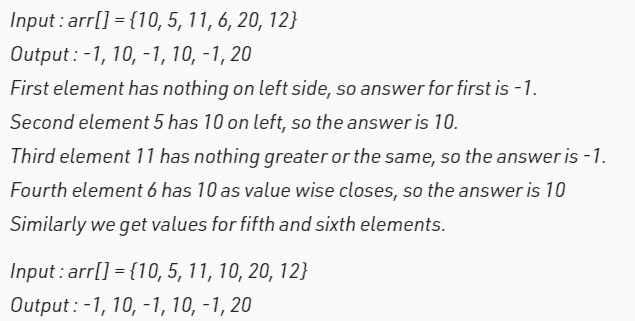


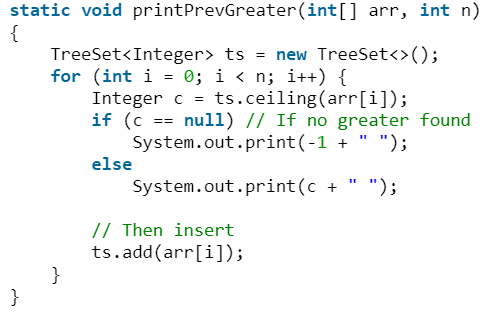
### Ceiling on left side in an Array

Given array of integers we need to find ceiling of every element from left side.

(Ceiling means element that are smallest greater than equal to itself)

We did similar problem named previous greater which was implement using stack. But this problem different here we are asked to find ceiling of element (smallest greater element).





### Vertical sum

Given a Binary Tree, find vertical sum of the nodes that are in same vertical line. Print all sums through different vertical lines starting from left-most vertical line to right-most vertical line.

