Binary Tree:

Each node can have at most 2 children.

Strict/Proper Binary tree:

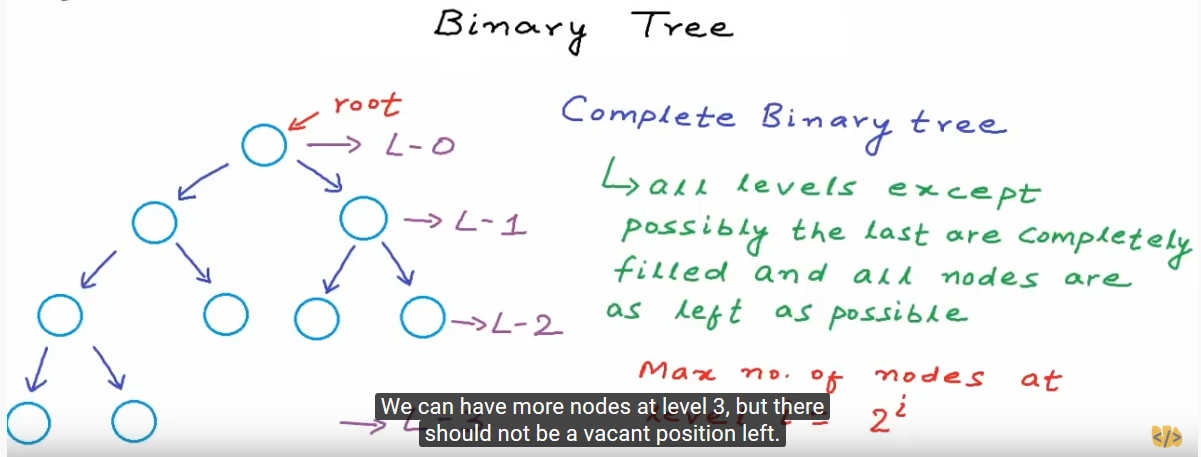
Each node can have either 2 or 0 children.

Complete Binary Tree:

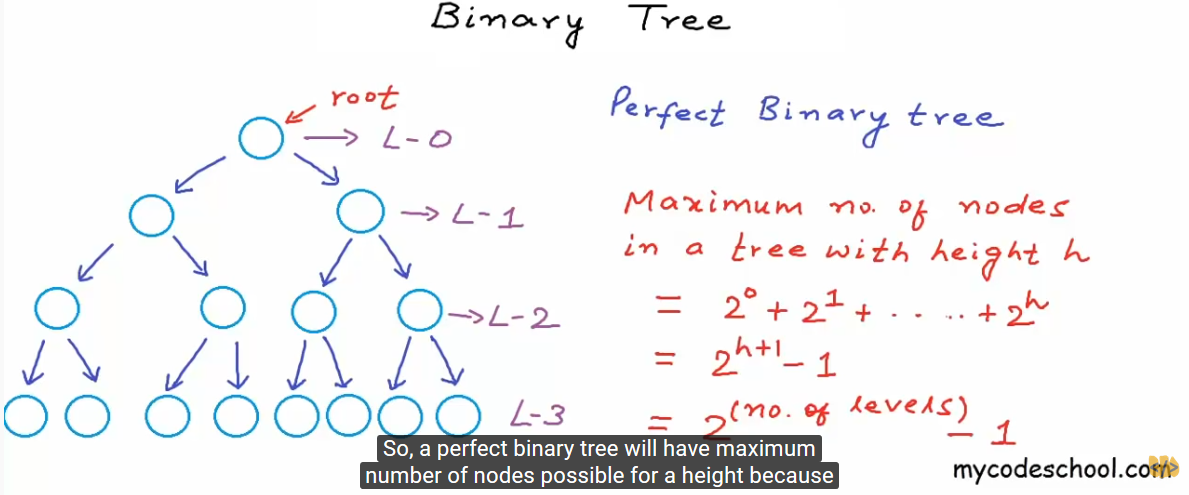
All levels except possible the last are completely filled and all the nodes are as left as possible.

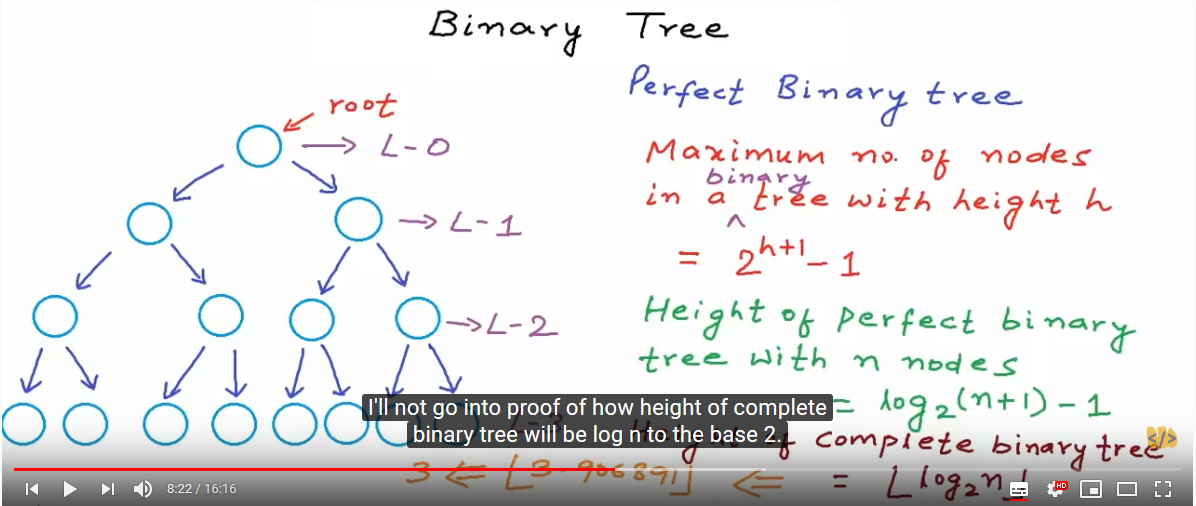
Max no. of nodes at level I = 2^i

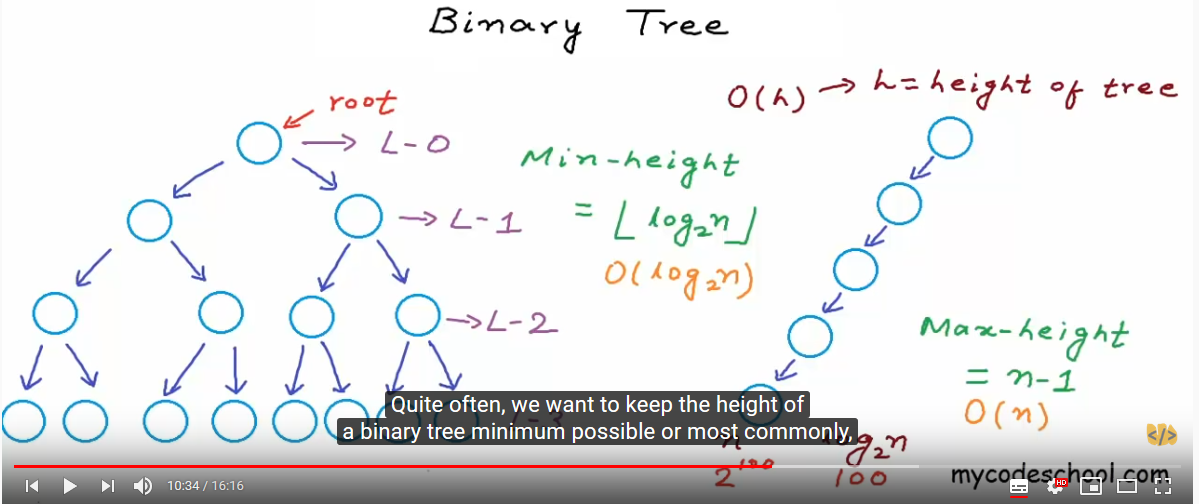
Below is completely binary tree. (there should not be a vacant position left)



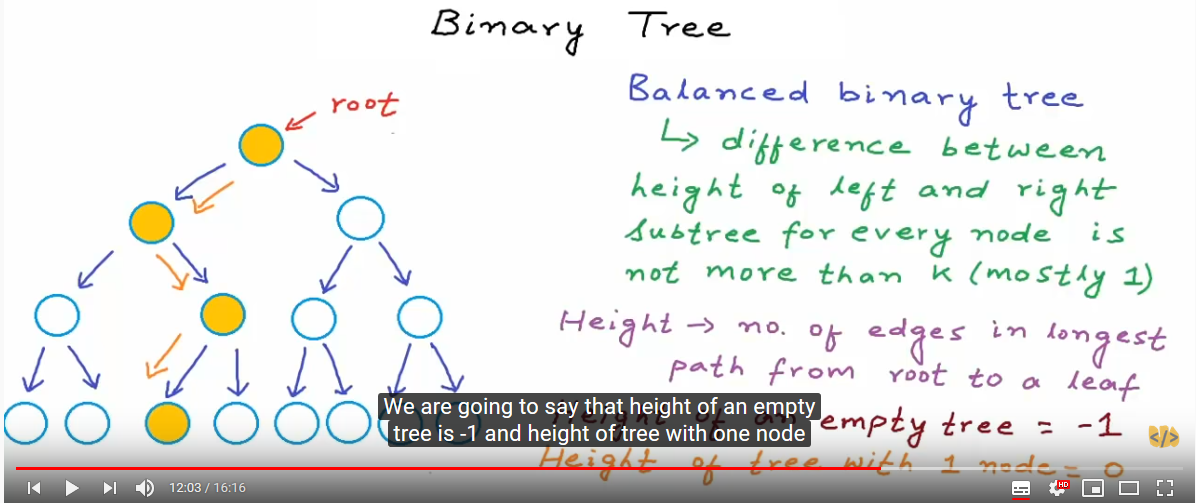
Perfect Binary tree:







Balanced Binary Tree:



Difference between = |Hleft-Hrihth|

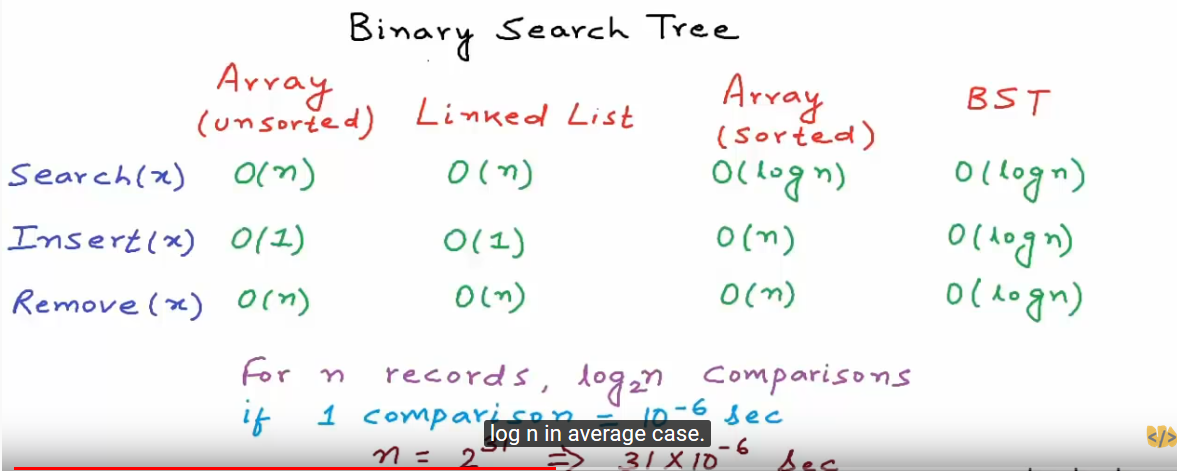
Binary Tree Implementation:

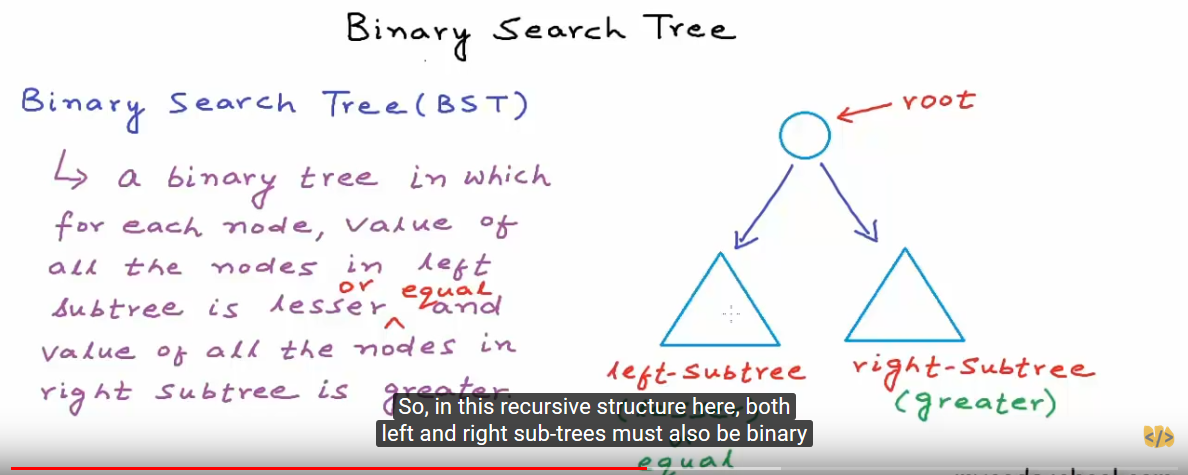


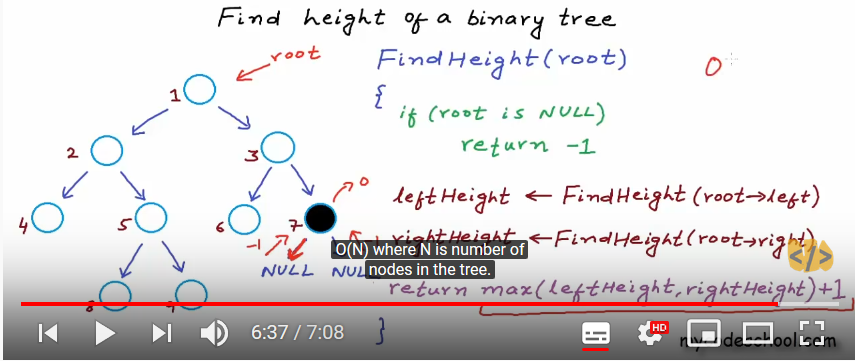
**Binary Search Tree:**

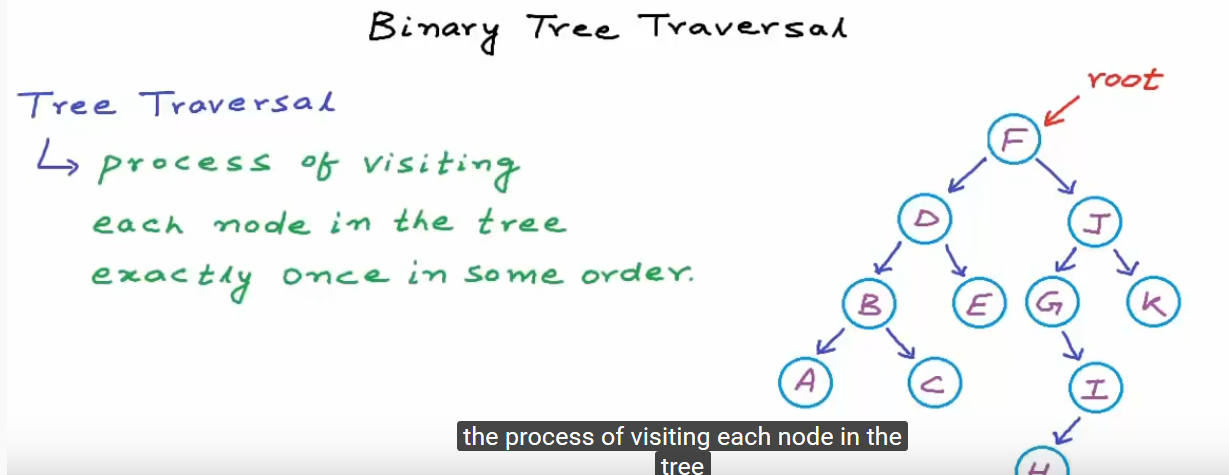
Why Binary Search tree?

Binary search of array is O(logn) work similar







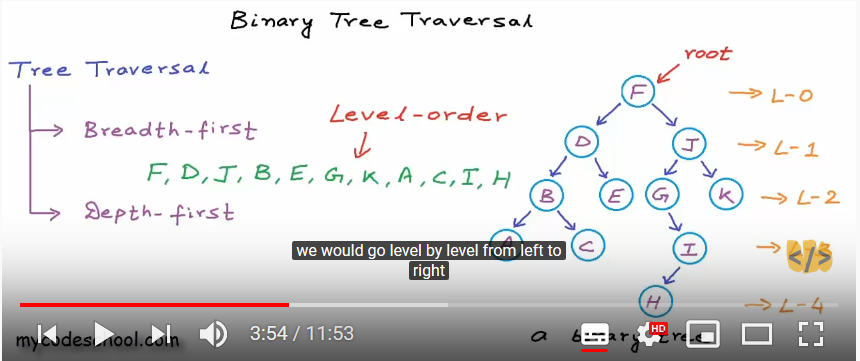


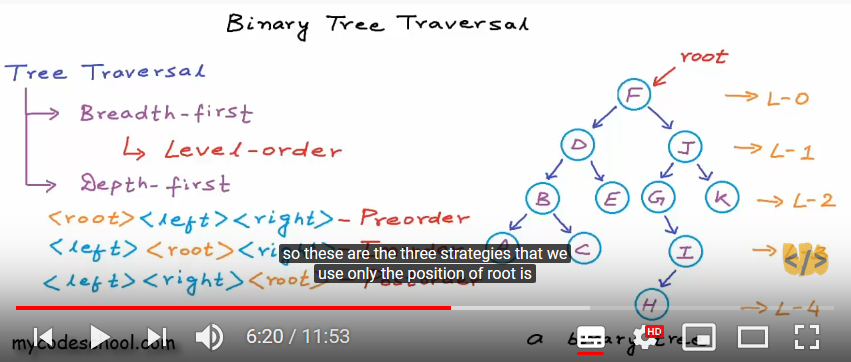
Tree Traversal of two type:

Breadth First (Level order Traversal)

F-> D -> J -> B -> E -> G -> K -> A -> C -> I -> H

Depth First





Preorder: (Visit, Left, Right)

F-> D-> B -> A-> C-> E -> J -> G -> I-> H -> k

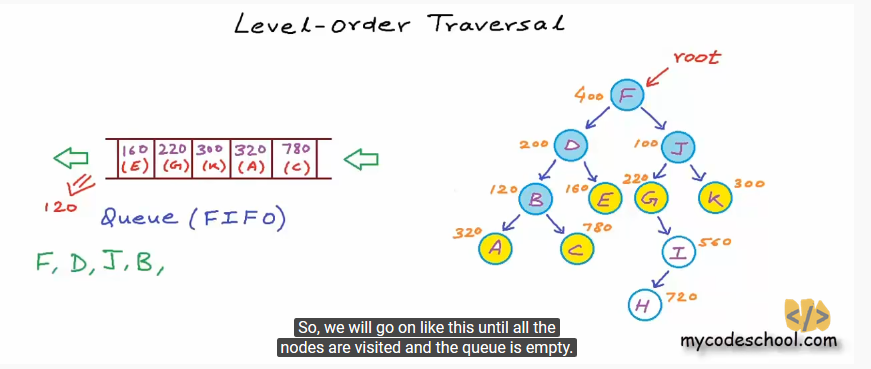
InOrder: (Left, Visit, Right) (Sorted list)

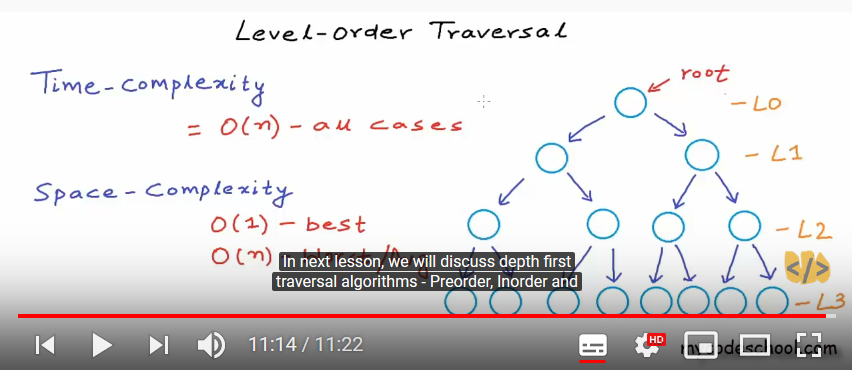
A->B-> C -> D -> E -> F -> G -> H -> I -> J -> K

PostOrder : (Left, Right, Visit)

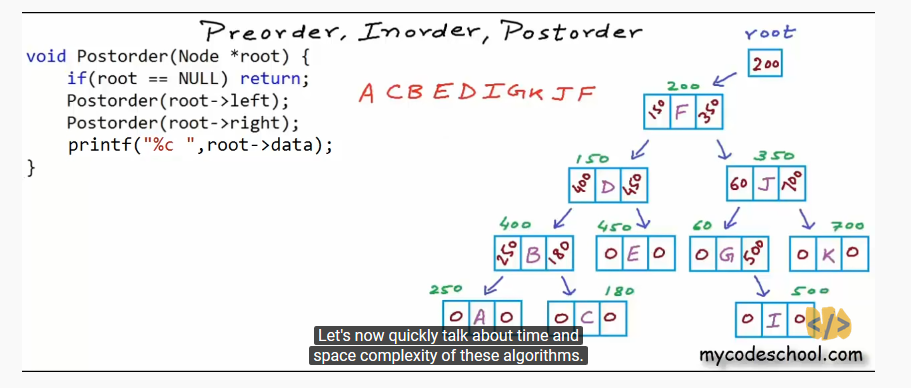
A-> C -> B -> E -> D -> H -> I -> G -> K -> J -> F

Level order Traversal: (F-> D -> J -> B -> E -> G -> K -> A -> C -> I -> H)





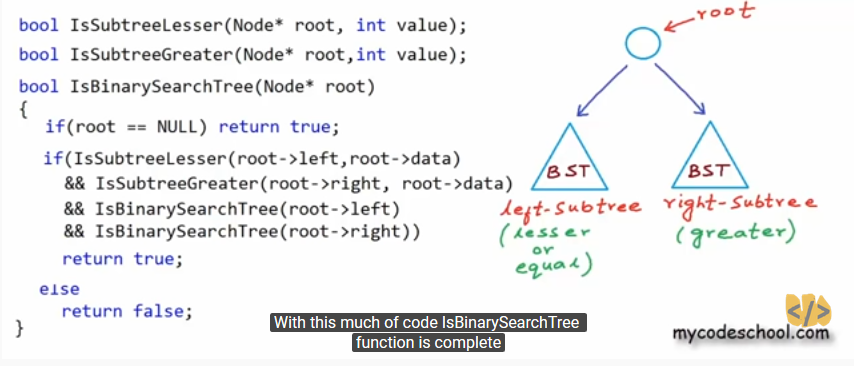
Depth – first Traversal:



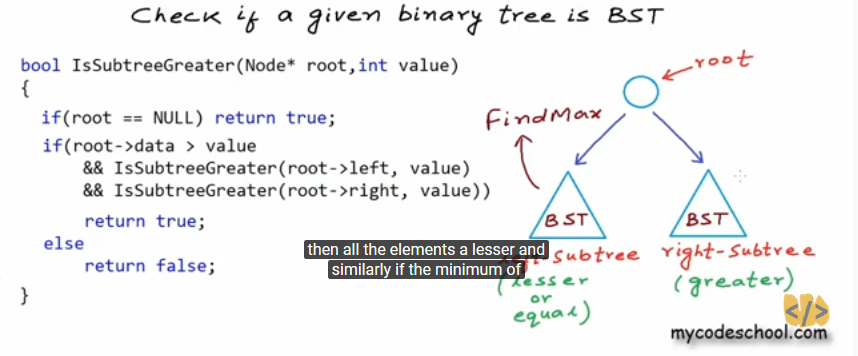
**Check if a binary tree is binary search tree or not?**

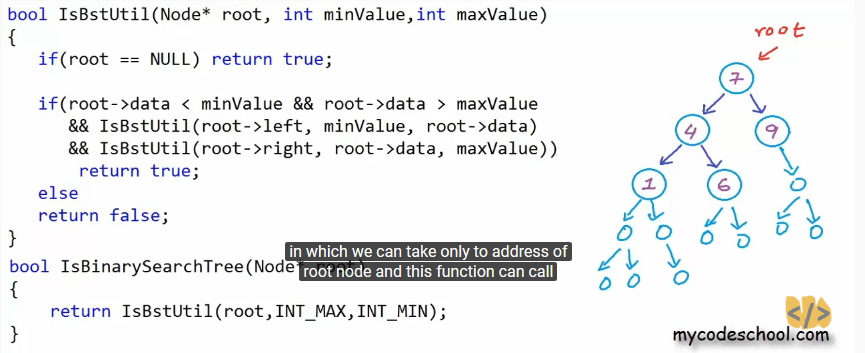
There are three approaches:

* With isSubtreeLesser(), isSubtreeGreater()
* With Finding Max from left subtree and finding min from right subtree.
* Check if each of node in given range.
* Perform inorder traversal, keep track of previous node value and compare with current node value for min value.

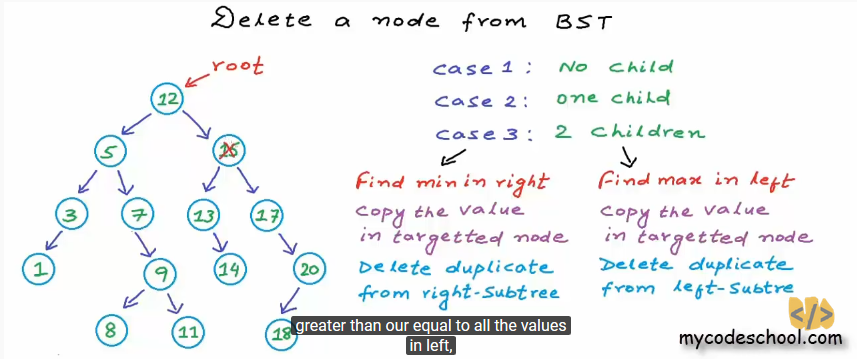


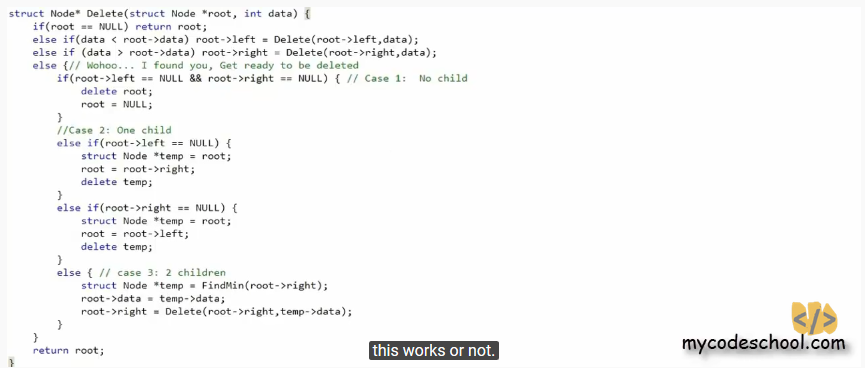






**Deleting the value in BST:**





**Zig-Zig Traversal:**

**package** com.sachin.binary;

**import** java.util.Stack;

**class** Node{

**int** value;

Node leftChild;

Node rightChild;

Node(**int** value){

**this**.value = value;

**this**.leftChild=**null**;

**this**.rightChild=**null**;

}

}

**public** **class** BinaryTree {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Node root = **null**;

root = **new** Node(20);

root.leftChild= **new** Node(21);

root.rightChild= **new** Node(23);

root.leftChild.leftChild = **new** Node(1);

//root.leftChild.rightChild = new Node

root.rightChild.leftChild = **new** Node(3);

root.rightChild.rightChild = **new** Node(8);

//addValue(root, 20);

*printZigZag*(root);

}

**private** **static** **void** printZigZag(Node root) {

// **TODO** Auto-generated method stub

**if**(root == **null**)

**return**;

Stack<Node> cl = **new** Stack<Node>();

Stack<Node> nl = **new** Stack<Node>();

cl.push(root);

**boolean** flag = **true**;

**while**(!cl.isEmpty()) {

Node node =cl.pop();

System.***out***.print(node.value + "->");

**if**(flag) {

**if**(node.leftChild != **null**)

nl.push(node.leftChild);

**if**(node.rightChild != **null**)

nl.push(node.rightChild);

}**else** {

**if**(node.rightChild != **null**)

nl.push(node.rightChild);

**if**(node.leftChild != **null**)

nl.push(node.leftChild);

}

**if**(cl.empty()) {

Stack<Node> temp = cl;

cl=nl;

nl = temp;

flag =!flag;

}

}

}

}

package com.sachin.tree;

import java.util.LinkedList;

import java.util.Queue;

public class BinaryTree {

static class BstNode{

public int value;

public BstNode left;

public BstNode right;

BstNode(int value){

this.value = value;

this.left = null;

this.right = null;

}

}

public static void main(String[] arg) {

BstNode root = null;

root = insertNode(root, 15);

root= insertNode(root, 10);

root= insertNode(root, 20);

root= insertNode(root, 25);

root= insertNode(root, 8);

root= insertNode(root, 12);

System.out.println("search for 8 " + searchNode(root, 8));

System.out.println("Find the Min Value in BST : " + minBstItrarative(root));

System.out.println("Find the Max value in BST : " + maxBstItravative(root));

System.out.println("Find the Min Value in BST : " + minBstRecursive(root));

System.out.println("Find the Max Value in BST : " + maxBstRecrsive(root));

System.out.println("Find the Hight of BST : " + findHightBst(root));

levelOrderTraversal(root);

System.out.println("PerOrder Traversal :");

perOrderTraversal(root);

System.out.println("InOrder Traversal : ");

inOrderTraversal(root);

System.out.println("PostOrder Traversal : ");

postOrderTraversal(root);

System.out.println("Checking the tree is Binary Search tree using range: " + isBST(root));

System.out.println("Checking the tree is Binary Search tree using inorder traversal: " + isBSTInorder(root));

System.out.println(deleteNode(root,25));

System.out.println(deleteNode(root,20));

System.out.println(deleteNode(root,10));

}

private static BstNode deleteNode(BstNode root, int value) {

// TODO Auto-generated method stub

if(root == null) return root;

if(value < root.value) root.left = deleteNode(root.left, value);

else if(value > root.value) root.right = deleteNode(root.right, value);

else {

if(root.left == null && root.right == null) {

System.out.println("Dleted case 1: " + root.value);

root=null;

return root;

}

else if(root.left == null) {

System.out.println("Dleted case 2: " + root.value);

root = root.right;

return root;

}

else if(root.right == null) {

System.out.println("Dleted case 3: " + root.value);

root= root.left;

return root;

}

else {

BstNode temp= findMin(root);

root.value= temp.value;

System.out.println("Dleted case 4: " + root.value);

root.right = deleteNode(root.right, temp.value);

}

}

return root;

}

private static BstNode findMin(BstNode root) {

// TODO Auto-generated method stub

if(root == null) return root;

while(root.left != null)

root = root.left;

return root;

}

private static boolean isBSTInorder(BstNode root) {

// TODO Auto-generated method stub

BstNode prev = null;

if(root != null) {

if(!isBSTInorder(root.left)) {

return false;

}

if(prev != null && root.value <= prev.value)

return false;

prev = root;

return isBSTInorder(root.right);

}

return true;

}

private static boolean isBST(BstNode root) {

// TODO Auto-generated method stub

return checkingBST(root,-1000,1000);

}

private static boolean checkingBST(BstNode root, int min, int max) {

// TODO Auto-generated method stub

if(root== null) return true;

if(root.value < max && root.value > min && checkingBST(root.left, min , root.value) && checkingBST(root.right, root.value, max))

return true;

else

return false;

}

private static void postOrderTraversal(BstNode root) {

// TODO Auto-generated method stub

if(root == null) return;

postOrderTraversal(root.left);

postOrderTraversal(root.right);

System.out.println(root.value);

}

private static void inOrderTraversal(BstNode root) {

// TODO Auto-generated method stub

if(root== null) return;

inOrderTraversal(root.left);

System.out.println(root.value);

inOrderTraversal(root.right);

}

private static void perOrderTraversal(BstNode root) {

// TODO Auto-generated method stub

if(root == null) return;

System.out.println(root.value);

perOrderTraversal(root.left);

perOrderTraversal(root.right);

}

private static void levelOrderTraversal(BstNode root) {

// TODO Auto-generated method stub

if(root == null) return;

Queue<BstNode> queue = new LinkedList<>();

queue.add(root);

while(!queue.isEmpty()) {

BstNode current = queue.peek();

if(current.left != null) queue.add(current.left);

if(current.right != null) queue.add(current.right);

System.out.println(queue.remove().value);

}

}

private static int findHightBst(BstNode root) {

// TODO Auto-generated method stub

if(root == null) return -1;

int leftHight = findHightBst(root.left);

int rightHight = findHightBst(root.right);

return Math.max(leftHight, rightHight) + 1;

}

private static int maxBstRecrsive(BstNode root) {

// TODO Auto-generated method stub

if(root == null) return -1;

if(root.right == null) return root.value;

return maxBstRecrsive(root.right);

}

private static int minBstRecursive(BstNode root) {

// TODO Auto-generated method stub

if(root == null) return -1;

if(root.left == null) return root.value;

return minBstRecursive(root.left);

}

private static int maxBstItravative(BstNode root) {

// TODO Auto-generated method stub

if(root == null) return -1;

while(root.right != null)

root = root.right;

return root.value;

}

private static int minBstItrarative(BstNode root) {

// TODO Auto-generated method stub

if(root == null)return -1;

while(root.left != null)

root= root.left;

return root.value;

}

private static Boolean searchNode(BstNode root, int data) {

// TODO Auto-generated method stub

if(root == null) return false;

else if(root.value == data) return true;

else if(data<=root.value)return searchNode(root.left, data);

else return searchNode(root.right, data);

}

private static BstNode insertNode(BstNode root, int value) {

// TODO Auto-generated method stub

if(root == null) {

BstNode newNode = new BstNode(value);

return newNode;

}

if(value <= root.value) {

root.left=insertNode(root.left,value);

}else {

root.right = insertNode(root.right, value);

}

return root;

}

}