In the pre-order traversal for a given node 'n',   
1. We first visit the node 'n' itself.   
2. Then we traverse left-subtree of 'n' by calling printPreorder(n.left)   
3. And finally we traverse right-subtree of 'n' by calling printPreorder(n.right)

# Order of the Algorithm

### Time Complexity is O((n)) Space Complexity is O((1))

### In the post-order traversal for a given node 'n',  1. We first traverse left-subtree of 'n' by calling printPostorder(n.left)  2. Then we traverse right-subtree of 'n' by calling printPostorder(n.right) 3. And finally we visit node 'n' itself.

# Order of the Algorithm

### Time Complexity is O((n)) Space Complexity is O((1))

### In the in-order traversal for a given node 'n',  1. We first traverse left-subtree of 'n' by calling in\_order\_print(n.left)  2. Then we visit node 'n' itself. 3. And finally we traverse right-subtree of 'n' by calling in\_order\_print(n.right)

### Level Order Traversal of a Binary Tree     1. Create a queue and add root to the queue.     2. Till the queue is not empty, repeat the following steps (3-5):     3. Get a node from the queue and print it     4. If the node has a left child, add the left child to the queue.     5. If the node has a right child, add the right child to the queue.

# Order of the Algorithm

### Time Complexity is O(n) Space Complexity is O(n)

**private** **void** printLevelOrderTraversalHelper(TreeNode root) {

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

**return**;

}

LinkedList<TreeNode> queue = **new** LinkedList<TreeNode>();

queue.add(root);

**int** i=0;

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

TreeNode node = queue.remove();

// if(i>2) //to print nodes at level 2

System.*out*.print(node.getData() + " ");

// i++;

**if**(node.getLeft() != **null**) {

queue.add(node.getLeft());

}

**if**(node.getRight() != **null**) {

queue.add(node.getRight());

}

### }

### Print all nodes of a binary tree that do not have sibling

### This is a simple problem solved using traversal of the tree. We can use any traversal method and while traversing we need to check if a node has only one child. If it does then we need to print that out. In this algorithm, we have used pre-order traversal method. Time taken is O(n) and space taken is O(1) for this algorithm.

**public** **void** printNonSiblingNodes(TreeNode currentNode)

{

**if** (currentNode == **null**)

{

**return**;

}

**if** (currentNode.left == **null** && currentNode.right != **null**)

{

System.*out*.println(currentNode.right.value);

}

**if** (currentNode.right == **null** && currentNode.left != **null**)

{

System.*out*.println(currentNode.left.value);

}

printNonSiblingNodes(currentNode.left);

printNonSiblingNodes(currentNode.right);

### }

### To add

### To add an element to a binary search tree, we begin as if we were searching for the  element, following the appropriate links from parent to child.  When the desired  child is null, we add the element as a new child node.  For example, if we were to add  14 to the above tree, we would go down the tree.  Once we reached 15, we would see  that the node has no left child, so we would add 14 as a left child.

### BST --- The  important thing to remember is that insertion, removal, and lookup take O(log n)  time (where n is the number of elements in the tree), since the height of a well‐ balanced binary search tree is O(log n).  Although in the worst case, a binary search  tree might have a height of O(n), there are "self‐balancing" binary search trees that  periodically reorganize a BST to ensure a height of O(log n).  Many self‐balancing  BST's guarantee that operations take O(log n) time

### 

### Path Between Nodes in a Binary Tree :

### Good Answer: There will always be exactly one path: from the starting node to the  lowest common ancestor of the nodes to the second node.  The goal is to identify the  lowest common ancestor. For each node, keep track of a set of nodes in the binary tree (using a hash table or a  BST) as well as a current node.  At each iteration, for each of the two current nodes,  change the current node to be its parent and add it to the appropriate set.  The first  element that is added to one set when it is already present in the other set is the  lowest common ancestor.  This algorithm takes O(n) time, where n is the length of  the path.  For example, if we were finding the lowest common ancestor of 3 and 15  in the above tree, our algorithm would do the following: Current node 1 | Current node 2 | Set 1 | Set 2 -------------------------------------------------------- 3 | 15 | 3 | 15 6 | 12 | 3, 6 | 15, 12 17 | 6 | 3, 6, 17 | 15, 12, 6 To improve the solution, we actually only need to use one set instead of two.

### Solution to find the path between nodes can also get by 🡪

### Find the root of node from root node to source node

### Find the root of node from root node to target node

### Then there will be common node or intersection node for both.

### Then change link at common node to path to target node.

### Question: Compute 2^x

### How can you quickly compute 2^x?

### Good answer: 1 << x (1 left‐shifted by x)

### Question: Is Power of 2

### How can you quickly determine whether a number is a power of 2?

### Good answer: Check whether x & (x ‐ 1) is 0.  If x is not an even power of 2, the

### highest position of x with a 1 will also have a 1 in x ‐ 1; otherwise, x will be 100...0

### and x ‐ 1 will be 011...1; and'ing them together will return 0.

### Check if a given binary tree is symmetric tree or not

## Write a program to check if the given binary tree is symmetric tree or not. A symmetric tree is defined as a tree which is mirror image of itself about the root node. For example, following tree is a symmetric tree.

=======================================================

### check-if-n-ary-tree-is-symmetric-tree 🡪

### Print right view of a binary tree Given a binary tree, print the values of nodes which would be present in right view of binary tree. A node will be in the right-view if it is the right-most node at its level (imagine level order traversal). For example,                      1           2             3       4      5       6           8              9           10 In the above tree,  node 1 is right-most node for level 0. node 3 is right-most node for level 1. node 6 is right-most node for level 2. node 9 is right-most node for level 3. node 10 is right-most node for level 4.                 And therefore right view for this tree is 1,3,6,9,10

# **Print all Root to Leaf paths of a Binary Tree**

## Algorithm/Insights

The algorithm traverses the tree in pre-order manner and uses an array list to store the paths.  
When a leaf node is reached, the path is printed.  
path - an array list that stores the current path.  
  
Step 1: Add root data to the array list.   
Step 2: If root is a leaf, print the path and return.  
Step 3: Recursively traverse the left subtree.  
Step 4: Recursively traverse the right subtree.  
  
Note: A new array list is created in the recursive calls (Steps 3 and 4) because we do not want to share the same array list in left and right subtree calls as the paths will be different.  
We add nodes up to the current path to the array list because the paths up to the current node are common for left and right subtrees.

|  |
| --- |
| import java.util.ArrayList; |
| 2 |  |
| 3 |  |
| 4 | public class RootToLeafPathsTest { |
| 5 | public static void main(String[] args) { |
| 6 | Tree tree = new Tree(); |
| 7 | tree.createSampleTree(); |
| 8 | tree.printRootToLeafPaths(); |
| 9 | } |
| 10 | } |
| 11 |  |
| 12 | class Tree { |
| 13 |  |
| 14 | private TreeNode root; |
| 15 |  |
| 16 | public TreeNode getRoot() { |
| 17 | return root; |
| 18 | } |
| 19 |  |
| 20 | public void printRootToLeafPaths() { |
| 21 | if(root == null) { |
| 22 | return; |
| 23 | } |
| 24 | ArrayList<Integer> path = new ArrayList<Integer>(); |
| 25 | printRootToLeafPaths(root, path); |
| 26 | } |
| 27 |  |
| 28 | private void printRootToLeafPaths(TreeNode root, ArrayList<Integer> path) { |
| 29 | path.add(root.getData()); |
| 30 |  |
| 31 | if(root.getLeft() == null && root.getRight() == null) { |
| 32 | printList(path); |
| 33 | return; |
| 34 | } |
| 35 | printRootToLeafPaths(root.getLeft(),new ArrayList<Integer>(path)); |
| 36 | printRootToLeafPaths(root.getRight(),new ArrayList<Integer>(path)); |
| 37 | } |
| 38 |  |
| 39 | private void printList(ArrayList<Integer> path) { |
| 40 | for(Integer i: path) { |
| 41 | System.out.print(i + " " ); |
| 42 | } |
| 43 | System.out.println(); |
| 44 | } |
| 45 |  |
| 46 | public void setRoot(TreeNode root) { |
| 47 | this.root = root; |
| 48 | } |
| 49 |  |
| 50 |  |
| 51 | public void createSampleTree() { |
| 52 | root = new TreeNode(1, new TreeNode(2, new TreeNode(4), new TreeNode(5)), new TreeNode(3, new TreeNode(6), new TreeNode(7))); |
| 53 | } |
| 54 | } |
| 55 |  |
| 56 | class TreeNode { |
| 57 |  |
| 58 | private int data; |
| 59 | private TreeNode left; |
| 60 | private TreeNode right; |
| 61 |  |
| 62 | public TreeNode(int data, TreeNode left, TreeNode right) { |
| 63 | this.data = data; |
| 64 | this.left = left; |
| 65 | this.right = right; |
| 66 | } |
| 67 |  |
| 68 | public TreeNode(int i) { |
| 69 | data = i; |
| 70 | } |
| 71 |  |
| 72 | public int getData() { |
| 73 | return data; |
| 74 | } |
| 75 |  |
| 76 | public void setData(int data) { |
| 77 | this.data = data; |
| 78 | } |
| 79 |  |
| 80 | public TreeNode getLeft() { |
| 81 | return left; |
| 82 | } |
| 83 |  |
| 84 | public void setLeft(TreeNode left) { |
| 85 | this.left = left; |
| 86 | } |
| 87 |  |
| 88 | public TreeNode getRight() { |
| 89 | return right; |
| 90 | } |
| 91 |  |
| 92 | public void setRight(TreeNode right) { |
| 93 | this.right = right; |
| 94 | } |
| 95 | } |

# **Print left view of a binary tree**

### Given a binary tree, print the values of nodes which would be present in left view of binary tree. A node will be in the left-view if it is the left-most node at its level (imagine level order traversal). For example,                      1           2             3       4      5       6           8              9           10 In the above tree,  node 1 is left-most node for level 0. node 2 is left-most node for level 1. node 4 is left-most node for level 2. node 8 is left-most node for level 3. node 10 is left-most node for level 4.                 And therefore left view for this tree is 1,2,4,8,10

## Algorithm/Insights

If we do a level order traversal of tree such that the left most node of each level is visited before all other nodes in that level then all we need to do is to print out the first visited node at each level to print the left view of tree. This can be done by doing a level order traversal from left end to right end and keeping track of max-level seen so far to find out when the new level starts. As soon as we find out that the new level has started, we print out the current node that is being visited.  
  
The steps for this algorithm are -   
1. Initialize maxLevelSeenSoFar to -1 and call printLeftViewLevelOrder(currentNode = root)  
2. In function printLeftViewLevelOrder if currentNode is null, do nothing and return.  
3. Else, add tuple (node = currentNode, level = 0) to the 'queue'.  
4. while 'queue' is not empty do   
       - remove the first tuple(currentNode, level) from the queue.   
       - if (level > maxLevelSeenSoFar) then we know that we are starting to traverse a new level and this is the first(and left most) node for this new  level and therefore we print currentNode's value and update maxLevelSeenSoFar to level.  
       - if left child of currentNode is not null then we add tuple (currentNode.left, level + 1) to 'queue'.  
       - if right child of currentNode is not null then we add tuple (currentNode.right, level + 1) to 'queue'.      
       \* Please note that we are adding left child of node to the 'queue' before right child to make sure that the left-most node at any level is visited before other nodes at that level.    
  
After execution of step #4, left view of tree would be printed.  
  
Notice that this algorithm takes O(n) extra space. With the following algorithm, we can save on extra space.  
  
This algorithm basically uses modified pre-order traversal. In this modified pre-order traversal we make sure that -   
a. The left-most node of any given level is visited before other nodes in that level. This can be easily achieved by visiting left sub-tree of node before right sub-tree. Basically, in this traversal, we visit the node first, then visit the left sub-tree and finally the right sub-tree(N-L-R).  
b. We print out the node value as soon as we encounter a node in the level that is greater than the maximum level seen so far and update maximum level seen so far to current level.   
  
The steps of this algorithm are as following -   
1. Initialize maxLevelSeenSoFar to -1 and call printLeftView(currentNode = root, level = 0)  
2. In function printLeftView(currentNode, level),  
    a. If currentNode is null, then we do nothing and return.  
    b. Else, if (level > maxLevelSeenSoFar) we print out the currentNode's value and update maxLevelSeenSoFar to level.  
    c. Make a recursive call printLeftView(currentNode.left, level + 1) to make sure nodes in the left sub-tree are visited.  
    d. Make a recursive call printLeftView(currentNode.right, level + 1) to make sure nodes in the right sub-tree are visited.      
    \*\* Notice that while visiting nodes with recursive calls, printLeftView(currentNode.left, level + 1) is called before printLeftView(currentNode.right, level + 1) in order to make sure that for any node, left sub-tree of that node is visited before right sub-tree. This guarantees that at any level, the left-most node is visited before other nodes at that level.  
  
After execution of step #2, left view of tree would be printed.  
         
Please checkout code snippet and algorithm visualization section for more details of the algorithm.

|  |
| --- |
| import java.util.LinkedList; |
| 2 |  |
| 3 |  |
| 4 | public class LeftViewOfBinaryTree { |
| 5 |  |
| 6 | class QueueNode |
| 7 | { |
| 8 | TreeNode node; |
| 9 | int level; |
| 10 |  |
| 11 | QueueNode(TreeNode node, int level) |
| 12 | { |
| 13 | this.node = node; |
| 14 | this.level = level; |
| 15 | } |
| 16 | } |
| 17 |  |
| 18 | class TreeNode |
| 19 | { |
| 20 | TreeNode left; |
| 21 | TreeNode right; |
| 22 | int val; |
| 23 |  |
| 24 | public TreeNode(int x) |
| 25 | { |
| 26 | this.val = x; |
| 27 | } |
| 28 | } |
| 29 |  |
| 30 | TreeNode root; |
| 31 |  |
| 32 |  |
| 33 |  |
| 34 |  |
| 35 | private TreeNode createTree() |
| 36 | { |
| 37 | this.root = new TreeNode(1); |
| 38 | TreeNode n1   = new TreeNode(2); |
| 39 | TreeNode n2   = new TreeNode(3); |
| 40 | TreeNode n3   = new TreeNode(4); |
| 41 | TreeNode n4   = new TreeNode(5); |
| 42 | TreeNode n5   = new TreeNode(6); |
| 43 | TreeNode n6   = new TreeNode(8); |
| 44 | TreeNode n7   = new TreeNode(9); |
| 45 | TreeNode n8   = new TreeNode(10); |
| 46 |  |
| 47 | root.left  = n1; |
| 48 | root.right = n2; |
| 49 |  |
| 50 | n1.left  = n3; |
| 51 | n1.right = n4; |
| 52 |  |
| 53 | n2.left  = n5; |
| 54 |  |
| 55 | n3.right = n6; |
| 56 |  |
| 57 | n5.right = n7; |
| 58 |  |
| 59 | n6.right = n8; |
| 60 |  |
| 61 | return root; |
| 62 | } |
| 63 |  |
| 64 |  |
| 65 | int maxLevelSoFar = -1; |
| 66 |  |
| 67 | public void printLeftViewLevelOrder(TreeNode currentNode) |
| 68 | { |
| 69 | if (currentNode == null) return; |
| 70 |  |
| 71 | LinkedList<QueueNode> queue = new LinkedList(); |
| 72 |  |
| 73 | queue.add(new QueueNode(currentNode, 0)); |
| 74 |  |
| 75 | while (!queue.isEmpty()) |
| 76 | { |
| 77 | QueueNode queueEntry = queue.remove(); |
| 78 | if (queueEntry.level > maxLevelSoFar) |
| 79 | { |
| 80 | maxLevelSoFar = queueEntry.level; |
| 81 | System.out.println(queueEntry.node.val); |
| 82 | } |
| 83 |  |
| 84 | if (queueEntry.node.left != null) |
| 85 | queue.add(new QueueNode(queueEntry.node.left, queueEntry.level + 1)); |
| 86 |  |
| 87 | if (queueEntry.node.right != null) |
| 88 | queue.add(new QueueNode(queueEntry.node.right, queueEntry.level + 1)); |
| 89 | } |
| 90 |  |
| 91 | } |
| 92 |  |
| 93 |  |
| 94 | public void printLeftView(TreeNode currentNode, int currentLevel) |
| 95 | { |
| 96 | if (currentNode == null) return; |
| 97 |  |
| 98 | if (currentLevel > maxLevelSoFar) |
| 99 | { |
| 100 | System.out.println(currentNode.val); |
| 101 | maxLevelSoFar = currentLevel; |
| 102 | } |
| 103 |  |
| 104 | printLeftView(currentNode.left, currentLevel + 1); |
| 105 | printLeftView(currentNode.right, currentLevel + 1); |
| 106 |  |
| 107 | } |
| 108 |  |
| 109 | public static void main(String[] args) |
| 110 | { |
| 111 | LeftViewOfBinaryTree tree = new LeftViewOfBinaryTree(); |
| 112 |  |
| 113 | tree.createTree(); |
| 114 |  |
| 115 | tree.printLeftView(tree.root, 0); |
| 116 |  |
| 117 |  |
| 118 | } |
| 119 | } |

# **Find sum of all left leaves of a binary tree**

### Write a program to find the sum of all left leaves of a given binary tree. For example, for the following shown tree output of the program should be 15 as there are two left leaves - node 9 and node 6. http://www.ideserve.co.in/learn/img/leftLeafSums.gif

## Algorithm/Insights

The solution to this problem is as simple as traversing the complete tree using any traversal method in order to visit all nodes and checking if a node has a left-child which is also a leaf node. If that is the case then add that left-child's value to the sum. As you can see in code snippet, we have used pre-order traversal to implement this algorithm.  
  
Please checkout findLeftLeavesSum(TreeNode currentNode, int[] leftLeavesSum) function in code snippet for implementation details.

|  |
| --- |
| public class SumOfAllLeftLeavesBinaryTree { |
| 2 |  |
| 3 | class TreeNode |
| 4 | { |
| 5 | TreeNode left; |
| 6 | TreeNode right; |
| 7 | int val; |
| 8 |  |
| 9 | public TreeNode(int x) |
| 10 | { |
| 11 | this.val = x; |
| 12 | } |
| 13 | } |
| 14 |  |
| 15 | TreeNode root; |
| 16 |  |
| 17 |  |
| 18 |  |
| 19 | private TreeNode createTree() |
| 20 | { |
| 21 | TreeNode n1   = new TreeNode(1); |
| 22 | TreeNode n2   = new TreeNode(2); |
| 23 | TreeNode n3   = new TreeNode(3); |
| 24 | TreeNode n4   = new TreeNode(4); |
| 25 | TreeNode n5   = new TreeNode(5); |
| 26 | TreeNode n6   = new TreeNode(6); |
| 27 | TreeNode n7   = new TreeNode(7); |
| 28 | TreeNode n8   = new TreeNode(8); |
| 29 | TreeNode n9   = new TreeNode(9); |
| 30 |  |
| 31 | this.root = n1; |
| 32 |  |
| 33 | root.left  = n2; |
| 34 | root.right = n3; |
| 35 |  |
| 36 | n2.left  = n4; |
| 37 | n2.right = n5; |
| 38 |  |
| 39 | n3.left  = n6; |
| 40 | n3.right = n7; |
| 41 |  |
| 42 | n4.right = n8; |
| 43 |  |
| 44 | n8.left = n9; |
| 45 |  |
| 46 | return root; |
| 47 | } |
| 48 |  |
| 49 | private boolean isLeafNode(TreeNode currentNode) |
| 50 | { |
| 51 | if (currentNode == null) |
| 52 | { |
| 53 | return false; |
| 54 | } |
| 55 |  |
| 56 | if (currentNode.left == null && currentNode.right == null) |
| 57 | { |
| 58 | return true; |
| 59 | } |
| 60 |  |
| 61 | return false; |
| 62 | } |
| 63 |  |
| 64 |  |
| 65 | public void findLeftLeavesSum(TreeNode currentNode, int[] leftLeavesSum) |
| 66 | { |
| 67 | if (currentNode == null) |
| 68 | { |
| 69 | return; |
| 70 | } |
| 71 | if (isLeafNode(currentNode.left)) |
| 72 | { |
| 73 | leftLeavesSum[0] += currentNode.left.val; |
| 74 | } |
| 75 |  |
| 76 | findLeftLeavesSum(currentNode.left, leftLeavesSum); |
| 77 | findLeftLeavesSum(currentNode.right, leftLeavesSum); |
| 78 | } |
| 79 |  |
| 80 | public static void main(String[] args) |
| 81 | { |
| 82 | SumOfAllLeftLeavesBinaryTree tree = new SumOfAllLeftLeavesBinaryTree(); |
| 83 |  |
| 84 |  |
| 85 |  |
| 86 | tree.createTree(); |
| 87 |  |
| 88 | int[] leftLeavesSum = new int[1]; |
| 89 | tree.findLeftLeavesSum(tree.root, leftLeavesSum); |
| 90 |  |
| 91 | System.out.println(leftLeavesSum[0]); |
| 92 | } |
| 93 | } |

# **Find depth of deepest odd level leaf node**

### Given a binary tree, find depth of deepest odd level leaf node. For example, in the following binary tree               1             /   \            2     3                 /  \                  4    5               /      \              6        7               \      /                  8    9                        /                   10 deepest odd level leaf node is 8 and depth is 5.

## Algorithm/Insights

Traverse the tree and keep track of current level of the node.  
Current level of a node can be easily tracked by maintaining a variable 'currLevel'. Level of Root node is 1. Increment 'currLevel' while traversing for left and right subtrees.   
While traversing:  
1: If root is null, return 0.  
2: Else if we find a leaf node and the current level is odd, return the current level.  
3: Else return maximum of (maximum depth of odd level leaf node found in left subtree, maximum depth of odd level leaf node found in right subtree).

|  |
| --- |
| public class Tree { |
| 2 |  |
| 3 | private Node root; |
| 4 |  |
| 5 | public int depthDeepestOddLevelLeafNode() { |
| 6 | return depthDeepestOddLevelLeafNode(root, 1); |
| 7 | } |
| 8 |  |
| 9 | private int depthDeepestOddLevelLeafNode(Node root, int currLevel) { |
| 10 | if (root == null) |
| 11 | return 0; |
| 12 | if (root.left == null && root.right == null && ((currLevel & 1) != 0)) |
| 13 | return currLevel; |
| 14 | return Integer.max(depthDeepestOddLevelLeafNode(root.left, currLevel + 1), depthDeepestOddLevelLeafNode(root.right, currLevel + 1)); |
| 15 | } |

public static void main(String[] args) {

        Tree tree = new Tree();

        tree.createSampleTree();

        int depthDeepestOddLevelLeafNode = tree.depthDeepestOddLevelLeafNode();

        if (depthDeepestOddLevelLeafNode != 0)

            System.out.println("Depth of deepest odd level leaf node is: " + depthDeepestOddLevelLeafNode);

        else

            System.out.println("No odd level leaf node exists");

    }

}

class Node {

    int data;

    Node left;

    Node right;

    public Node(int data, Node left, Node right) {

        this.data = data;

        this.left = left;

        this.right = right;

    }

    public Node(int data) {

        this.data = data;

    }

}

# **Check whether a binary tree is a full binary tree or not**

### Write a program to check if a given binary tree is a full binary tree or not. A binary tree is a full binary tree if all its nodes have either both children or no children. In other words, if any of its node has only one child then it is not a full binary tree. Both of the following trees are full binary trees. http://www.ideserve.co.in/learn/img/tree_3.gif Following two trees are not full binary trees. http://www.ideserve.co.in/learn/img/tree_4.gif In above tree, node 3 violates the constraint. http://www.ideserve.co.in/learn/img/tree_5.gif In above tree, node 4 violates the constraint.

## Algorithm/Insights

A simple level-order traversal is sufficient to solve this problem. While traversing the tree using level order traversal, if we visit any node with only one child, we return false.  
  
Please checkout code snippet and algorithm visualization section for details of the algorithm.

public class FullBinaryTreeCheck {

    class QueueNode

    {

        TreeNode node;

        QueueNode(TreeNode node)

        {

            this.node = node;

        }

    }

    class TreeNode

    {

        TreeNode left;

        TreeNode right;

        int val;

        public TreeNode(int x)

        {

            this.val = x;

        }

    }

private TreeNode createNonFullBinaryTree()

    {

        this.root = new TreeNode(1);

        TreeNode n1   = new TreeNode(2);

        TreeNode n2   = new TreeNode(3);

        TreeNode n3   = new TreeNode(4);

        TreeNode n4   = new TreeNode(5);

        TreeNode n5   = new TreeNode(6);

        TreeNode n6   = new TreeNode(7);

        root.left  = n1;

        root.right = n2;

        n1.left  = n3;

        n1.right = n4;

        n2.left  = n5;

        n3.left  = n6;

        return root;

    }

    public boolean checkIfFull()

    {

        if (root == null) return true;

        LinkedList<TreeNode> queue = new LinkedList();

        boolean hasLeftChild, hasRightChild;

        queue.add(root);

        while (!queue.isEmpty())

        {

            TreeNode currentNode = queue.remove();

             hasLeftChild = (currentNode.left != null);

             hasRightChild = (currentNode.right != null);

            if ((hasLeftChild && !hasRightChild) || (!hasLeftChild && hasRightChild))

            {

                return false;

            }

            if (hasLeftChild && hasRightChild)

            {

                queue.add(currentNode.left);

                queue.add(currentNode.right);

            }

        }

        return true;

    }

    public static void main(String[] args)

    {

        FullBinaryTreeCheck tree = new FullBinaryTreeCheck();

        tree.createNonFullBinaryTree();

        System.out.println(tree.checkIfFull());

        tree.createFullBinaryTree();

        System.out.println(tree.checkIfFull());

    }

}

# **Check if two nodes are cousins in a Binary tree**

### Given two nodes in a binary tree, check if they are cousins. Two nodes are cousins if:  1: they are not siblings (children of same parent). 2: they are on the same level. For example, in the following binary tree http://www.ideserve.co.in/learn/img/BinaryTree.gif nodes 5 and 6 are cousins but nodes 4 and 5 are not.

### 1: If node A == B return false as a node cannot be cousin of itself. 2: Check nodes A and B are not siblings and 3: Check nodes A and B are on same level. How to find if 2 nodes are siblings: 1. If A and B are left and right children of the root, then they are siblings. 2. Else check Step 1 in left and right subtrees. 3. If the condition in Step 1 is not true for any node, then the nodes are not siblings

|  |
| --- |
| public class Tree { |
| 2 |  |
| 3 | private Node root; |
| 4 |  |
| 5 | public void setRoot(Node root) { |
| 6 | this.root = root; |
| 7 | } |
| 8 |  |
| 9 |  |
| 10 | public Boolean isCousin(Node a, Node b) { |
| 11 |  |
| 12 | if (a == b) { |
| 13 | return false; |
| 14 | } |
| 15 | return (!isSibling(a, b) && getLevel(a) == getLevel(b)); |
| 16 | } |
| 17 |  |
| 18 | public int getLevel(Node a) { |
| 19 | return getLevel(root, a, 1); |
| 20 | } |
| 21 |  |
| 22 | private int getLevel(Node root, Node a, int currLevel) { |
| 23 | if (root == null) |
| 24 | return 0; |
| 25 | if (root == a) |
| 26 | return currLevel; |
| 27 | int level = getLevel(root.left, a, currLevel + 1); |
| 28 | if (level != 0) { |
| 29 | return level; |
| 30 | } else |
| 31 | return getLevel(root.right, a, currLevel + 1); |
| 32 | } |
| 33 |  |
| 34 | public boolean isSibling(Node a, Node b) { |
| 35 | return isSibling(root, a, b); |
| 36 | } |
| 37 |  |
| 38 | private boolean isSibling(Node root, Node a, Node b) { |
| 39 | if (root == null) |
| 40 | return false; |
| 41 | return ((root.left == a && root.right == b) || (root.right == a && root.left == b) || |
| 42 | isSibling(root.left, a, b) || isSibling(root.right, a, b)); |
| 43 | } |
| 44 |  |
| 45 | public static void main(String[] args) { |
| 46 | Tree tree = new Tree(); |
| 47 |  |
| 48 | Node root = new Node(1); |
| 49 | Node n2 = new Node(2); |
| 50 | Node n3 = new Node(3); |
| 51 | Node n4 = new Node(4); |
| 52 | Node n5 = new Node(5); |
| 53 | Node n6 = new Node(6); |
| 54 | Node n7 = new Node(7); |
| 55 |  |
| 56 | root.left = n2; |
| 57 | root.right = n3; |
| 58 |  |
| 59 | n2.left = n4; |
| 60 | n2.right = n5; |
| 61 |  |
| 62 | n3.left = n6; |
| 63 | n3.right = n7; |
| 64 | tree.setRoot(root); |
| 65 | System.out.println("Nodes "  + n5.data + " and "  + n6.data + (tree.isCousin(n5, n6) ? " are cousins."  : " are not cousins." )); |
| 66 | } |
| 67 | } |
| 68 |  |
| 69 | class Node { |
| 70 |  |
| 71 | int data; |
| 72 | Node left; |
| 73 | Node right; |
| 74 |  |
| 75 | public Node(int data, Node left, Node right) { |
| 76 | this.data = data; |
| 77 | this.left = left; |
| 78 | this.right = right; |
| 79 | } |
| 80 |  |
| 81 | public Node(int data) { |
| 82 | this.data = data; |
| 83 | } |
| 84 | } |

# **Check if two binary trees are identical**

### Given two trees, find if they are identical.  Two trees are said to be identical if they are structurally same and have same data in all nodes. Example 1: Tree 1:                 1         2                3     4        5        6        7 Tree 2:                 1         2                3     4        5        6        7 are identical. Example 2: Tree 1:                 1         2                3     4        6        5        7 Tree 2:                 1         2                3     4        5        6        7 are not identical. Their structure is same but values at 5 and 6 nodes differ. Example 3: Tree 1:                 1         2                3     4        5        6 Tree 2:                 1         2                3     4        5                6 are not identical. The nodes have same values but structure is not same as 6 is left sub tree of node 3 in tree 1 but right sub tree of node 3 in tree 2.

## Algorithm/Insights

Solution uses recursion to find out if the trees are identical.  
1. If root of both trees are null, then they are same. Return true.  
2. If roots of both the trees are not null, check if the data in the two nodes is same and recursively check if left and right subtrees are identical.  
3. If the roots of only one of the trees is null, then the trees are not identical, so return false.  
Please checkout code snippet and algorithm visualization section for understanding the algoithm.

 public static boolean areIdenticalTrees(TreeNode root1, TreeNode root2) {

        // If both the tree nodes are null, then both are identical

        if(root1 == null && root2 == null) {

            return true;

        }

        if(root1 != null && root2 != null) {

            // Check if the 2 nodes have same data and recursively check if left and right subtrees are identical

            return ((root1.getData() == root2.getData()) &&

                    (areIdenticalTrees(root1.getLeft(), root2.getLeft()) &&

                    (areIdenticalTrees(root1.getRight(), root2.getRight()))));

        }

        // If either of root1 or root2 is null but not both, then the trees are not identical

        return false;

    }

# **Convert a binary tree to its mirror tree**

### Given a tree, convert the tree to its corresponding mirror tree. A mirror tree of a tree is where left node of the root of the tree is right node of the mirror tree and right node of the tree is left node of mirror tree and left and right subtrees of the root are also mirror trees.  Example 1: Tree:                 1         2                3     4                5         Mirror Tree:                 1         3                2             5                4

### Boundary condition: If root is null, then return. Recursive step: Recursively convert left and right sub trees to their mirror. Actual conversion to mirror: Swap left and right sub trees of the current node. Please checkout code snippet and algorithm visualization section for understanding the algorithm.

public class MirrorTree {

    private TreeNode root;

    public void convertToMirror() {

        convertToMirror(root);

    }

    private void convertToMirror(TreeNode root) {

        if(root == null) {

            return;

        }

        // Mirror left subtree

        convertToMirror(root.getLeft());

        // Mirror right subtree

        convertToMirror(root.getRight());

        // Interchange left and right subtree root nodes

        TreeNode t = root.getLeft();

        root.setLeft(root.getRight());

        root.setRight(t);

    }

# **Print bottom view of a binary tree**

### Given a binary tree, Given a binary tree, print the values of nodes which would be present in bottom of view of binary tree. You are not allowed to use level order traversal. Print the node values starting from left-most end. A node will be in the bottom-view if it is the bottommost node at its horizontal distance from the root. Horizontal distance of root from itself is 0. Horizontal distance of right child of root node is 1 and horizontal distance of left child of root node is -1. Horizontal distance of node 'n' from root = horizontal distance of its parent from root + 1, if node 'n' is right child of its parent. Horizontal distance of node 'n' from root = horizontal distance of its parent from root - 1, if node 'n' is left child of its parent. If more than one nodes are at the same horizontal distance and are the bottom-most nodes for  that horizontal distance, then you can choose to include either of the nodes in the bottom view. example -      For the following tree:                 1         2                3     4        5        6        7 Horizontal distance of 1 = +0 Horizontal distance of 2 = -1 Horizontal distance of 3 = +1 Horizontal distance of 4 = -2 Horizontal distance of 5 = +0 Horizontal distance of 6 = +0 Horizontal distance of 7 = +2     and the bottom view would be 4, 2, 6, 3, 7. Please note that though there are three nodes(nodes 1,5,6) with horizontal distance of 0, only node 6 is in bottom view because it is at the bottom-most level of all three nodes. Note that you could have chosen to keep node 5 instead of node 6 in bottom view since it is also at the bottom-most layer..  For the following tree:                 1            2          3         4    5     6     7           8 9        10    11            Horizontal distance of 1 = +0 Horizontal distance of 2 = -1 Horizontal distance of 3 = +1 Horizontal distance of 4 = -2 Horizontal distance of 5 = +0 Horizontal distance of 6 = +0 Horizontal distance of 7 = +2     Horizontal distance of 8 = -1 (right child of 4) Horizontal distance of 9 = -1 (left child of 5) Horizontal distance of 10 = +1 (right child of 6) Horizontal distance of 11 = +3 (right child of 7)      and the bottom view would be 4, 9, 6, 10, 7, 11

# **Print binary tree in vertical order**

### Given a binary tree, print the nodes of binary tree grouped together in vertical order. Vertical order of a node is defined using its horizontal distance from the root node. Horizontal distance of root from itself is 0. Horizontal distance of right child of root node is +1 and horizontal distance of left child of root node is -1. Horizontal distance of node 'n' from root = horizontal distance of its parent from root + 1, if node 'n' is right child of its parent. Horizontal distance of node 'n' from root = horizontal distance of its parent from root - 1, if node 'n' is left child of its parent.                 3          4              5       6     7        8     9               1            In the above tree, horizontal distance of node 3 is  0 horizontal distance of node 4 is -1 horizontal distance of node 5 is +1 horizontal distance of node 6 is -2 horizontal distance of node 7 is  0 horizontal distance of node 8 is  0 horizontal distance of node 9 is +2 horizontal distance of node 1 is +1 and therefore expected vertical order print for this tree is:   6   4   3  7  8   5  1   9 Note that, the requirement is that vertical order should be printed starting from left-most end(node 4 cannot be printed before node 6) and in top down manner(node 1 cannot be printed before node 5 though both have same horizontal distance).

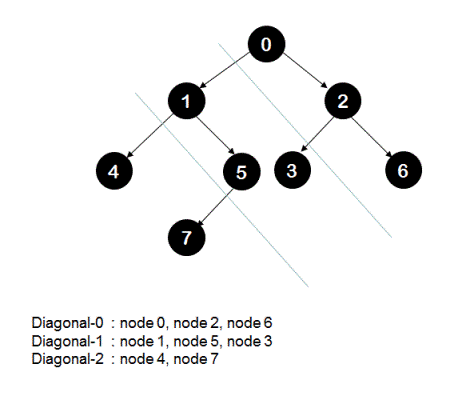
## Algorithm/Insights

The basic idea that will be used here is a modified level-order traversal where we keep track of the horizontal distance of each node while traversing the tree. And once we know horizontal distance of a node, we add it to the list of nodes which have same horizontal distance as that of current node. This list of nodes is maintained using a treeMap where key is horizontal distance 'd' and value is list of nodes which are at horizontal distance 'd' from the root. A treeMap is modified hashMap where keys are inserted in sorted order. (A treeMap is implemented using Red-Black tree and hence insertion/retrieval takes O(log n) time instead of O(1) time as is the case for a regular hashMap)  
  
The formal algorithm is as following -   
1. Initialize verticalOrderMap to an empty treeMap.  
2. Call fillUpVerticalOrderMap(currentNode = root, horizontalDistFromRoot = 0, verticalOrderMap = verticalOrderMap);  
3. If currentNode == null, return.   
4. Initialize queue to an empty queue.  
5. Add tuple(currentNode, horizontalDistFromRoot) to queue.  
6. While (queue is not empty)  
6a. Remove an entry(node, horizontalDistanceOfNode) from queue.  
6b. Add node to a list of nodes in verticalOrderMap at key horizontalDistanceOfNode. If list does not exist, create a new list and add this list to verticalOrderMap at key horizontalDistanceOfNode.  
6c. Because we don't want to add null nodes to queue, we check of node.left is null. If it is not, we add an entry (node.left, horizontalDistanceOfNode - 1) to queue.  
6d. Similar to above step, we add an entry (node.right, horizontalDistanceOfNode + 1) to queue if node.right is not null.  
7. After this step, verticalOrderMap is filled and we print out this map.

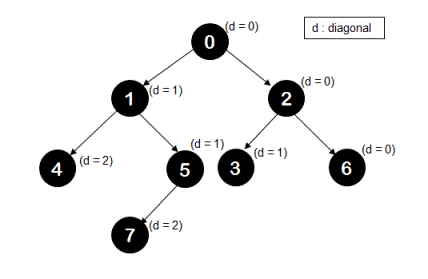
# **Populate right neighbors for all nodes in a binary tree**

### Class definition of tree node having reference to neighbor node looks like following in java. class TreeNode     {         TreeNode left;         TreeNode right;         TreeNode neighbor;         int value;              public TreeNode(int value)         {             this.value = value;         }     } As illustrated below, original tree on left hand side would be modified to the tree on the right hand side after populating  neighbors for all nodes.     http://www.ideserve.co.in/learn/img/populateNeighbors_0.gif

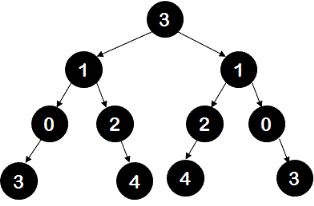
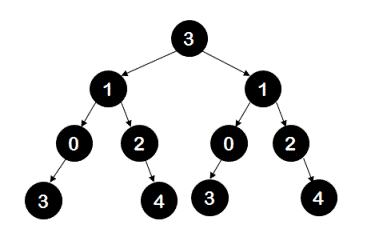
# **Diagonal Sum of a Binary Tree.**

Consider lines drawn at an angle of 135 degrees(that is slope = -1) which cut through the left branches of a given binary tree. A diagonal is formed by nodes which lie between two such consecutive lines. If we are able to draw 'n' lines then the complete tree is divided into 'n+1' diagonals. Diagonal sum in a binary tree is sum of all node's values lying between these lines. Given a binary tree, print all diagonal sums. Please note that all right branches are drawn parallel to each other and all left branches are also drawn parallel to each other.  
  
For example,  
  
in the above tree there are total of three diagonals.   
Diagonal-0 has 3 nodes in it: node-0, node-2 and node-6. Therefore, sum of all nodes in diagonal-0 is 8.  
Similarly, sum of nodes in diagonal-1 is 9(node-1, node-5 and node-3). And sum of nodes in diagonal-2 is 11(node-4, node-7).

## Algorithm/Insights

Please observe below tree carefully.   
  
As is marked in the tree, root node-0 is placed in diagonal 0, then its left child node-1 is placed in diagonal 1 and its right child node-2 is placed in the same diagonal as node-0 that is diagonal 0. This observation can be easily generalized - if current node is placed in diagonal 'd' then its left child would be placed in diagonal 'd+1' and its right child would be placed in the same diagonal as that of current node that is diagonal 'd'.  
  
Now using above insight, we can easily find the diagonal in which a node is placed in and then add that node's value to that diagonal.   
  
We traverse the complete tree using inorder traversal starting from the root node. During this traversal, we pass the 'currDiag' to each node indicating which diagonal that current node is placed in. We start by passing 'currDiag = 0' for the root node. Each node passes down 'currDiag + 1' to its left child and 'currDiag' to its right child. Now using this 'currDiag' value, we add current node to appropriate diagonal's sum. To store diagonal sum for each diagonal separately, HashMap is used with its key as diagonal number and value as sum of nodes in that diagonal.  
  
Time complexity of this algorithm is O(n) since each node would be visited once. Space complexity in the worst case would be O(n). Worst case space complexity occurs for the skewed tree where right child for each node is null. Average case space complexity of this algorithm would be O(log(n)).  
  
Please check out the function 'computeDiagSum(TreeNode currentNode, int currDiag, HashMapdiagSum)' in code snippet for implementation details. You can also watch the video above for step by step explanation.

# **Check if a given binary tree is symmetric tree or not**

Write a program to check if the given binary tree is symmetric tree or not. A symmetric tree is defined as a tree which is mirror image of itself about the root node. For example, following tree is a symmetric tree.  
  
  
whereas, following tree is not a symmetric tree.   


## Algorithm/Insights

The algorithm is an implementation of a simple idea that -   
1. For given two trees, if both trees are empty then they are mirror images of one another.  
Else they have to satisfy following conditions:  
2. Root values of both trees have to be same.  
3. Left sub-tree of tree1 should be mirror image of right sub-tree of tree2.  
4. Right sub-tree of tree1 should be mirror image of left sub-tree of tree2.   
  
Initial call to isSymmetric(TreeNode root1, TreeNode root2) is made with root1 = root and root2 = root.  
  
Time complexity of this algorithm is O(n) since in the worst case each node need to be visited once.