1.Write a Java program to find the nodes which are at the maximum distance in a Binary Tree

Description:  
Algorithm  
Define Node class which has three attributes namely: data left and right. Here, left represents the left child of the node and right represents the right child of the node.  
When a node is created, data will pass to data attribute of the node and both left and right will be set to null.  
Define another class which has two attribute root and treeArray.  
Root represents the root node of the tree and initializes it to null.  
treeArray will store the array representation of the binary tree.  
nodesAtMaxDistance() will find out the nodes which are present at the maximum distance:  
It will calculate the distance between all the nodes present in the binary tree and store it in the variable distance.  
MaxDistance keeps tracks of maximum possible distance between nodes. If maxDistance is less than distance, then the value of distance will be stored in maxDistance. Clears the array to get rid of previously stored values. Nodes which are at the maximum distance will be stored in an array arr.  
If more than one pair of nodes is at maxDistance then, store them in the array arr.  
a. calculateSize() will count the number of nodes present in the tree.  
b. convertBTtoArray() will convert the binary tree to its array representation by traversing the tree and adding elements to treeArray.

c. getDistance() will calculate the distance of a given node from the root.

d. LowestCommonAncestor() will find out the lowest common ancestor for the nodes n1 and n2.

If any of the nodes is equal to the root node, then return root as the lowest common ancestor.  
Else, search nodes n1 and n2 in left subtree and right subtree.  
If a node is found such that n1 is the left child of that node and n2 is right child of that node or vice-versa. Returns that node as the lowest common ancestor.  
a. FindDistance() will calculate the distance between two nodes.

First, it calculates the distance of each node from the root node.  
Subtract the 2 \* distance of lowest common ancestor from this root node

import java.util.ArrayList;

public class Main{

public static class Node{

int data;

Node left;

Node right;

public Node(int data){

this.data = data;

this.left = null;

this.right = null;

}

}

public Node root;

int[] treeArray;

int index = 0;

public Main(){

root = null;

}

public int calculateSize(Node node)

{

int size = 0;

if (node == null)

return 0;

else {

size = calculateSize (node.left) + calculateSize (node.right) + 1;

return size;

}

}

public void convertBTtoArray(Node node) {

if(root == null){

System.out.println("Tree is empty");

return;

}

else {

if(node.left != null)

convertBTtoArray(node.left);

treeArray[index] = node.data;

index++;

if(node.right != null)

convertBTtoArray(node.right);

}

}

public int getDistance(Node temp, int n1) {

if (temp != null) {

int x = 0;

if ((temp.data == n1) || (x = getDistance(temp.left, n1)) > 0

|| (x = getDistance(temp.right, n1)) > 0) {

return x + 1;

}

return 0;

}

return 0;

}

public Node lowestCommonAncestor(Node temp, int node1, int node2) {

if (temp != null) {

if (temp.data == node1 || temp.data == node2) {

return temp;

}

Node left = lowestCommonAncestor(temp.left, node1, node2);

Node right = lowestCommonAncestor(temp.right, node1, node2);

if (left != null && right != null) {

return temp;

}

if (left != null) {

return left;

}

if (right != null) {

return right;

}

}

return null;

}

public int findDistance(int node1, int node2) {

int d1 = getDistance(root, node1) - 1;

int d2 = getDistance(root, node2) - 1;

Node ancestor = lowestCommonAncestor(root, node1, node2);

int d3 = getDistance(root, ancestor.data) - 1;

return (d1 + d2) - 2 \* d3;

}

public void nodesAtMaxDistance(Node node) {

int maxDistance = 0, distance = 0;

ArrayList<Integer> arr = new ArrayList<>();

int treeSize = calculateSize(node);

treeArray = new int[treeSize];

convertBTtoArray(node);

for(int i = 0; i < treeArray.length; i++) {

for(int j = i; j < treeArray.length; j++) {

distance = findDistance(treeArray[i], treeArray[j]);

if(distance > maxDistance) {

maxDistance = distance;

arr.clear();

arr.add(treeArray[i]);

arr.add(treeArray[j]);

}

else if(distance == maxDistance) {

arr.add(treeArray[i]);

arr.add(treeArray[j]);

}

}

}

System.out.println("Nodes which are at maximum distance: ");

for(int i = 0; i < arr.size(); i = i + 2) {

System.out.println("( " + arr.get(i) + "," + arr.get(i+1) + " )");

}

}

public static void main(String[] args) {

Main bt = new Main();

bt.root = new Node(1);

bt.root.left = new Node(2);

bt.root.right = new Node(3);

bt.root.left.left = new Node(4);

bt.root.left.right = new Node(5);

bt.root.right.left = new Node(6);

bt.root.right.right = new Node(7);

bt.root.right.right.right = new Node(8);

bt.root.right.right.right.left = new Node(9);

bt.nodesAtMaxDistance(bt.root);

}

}

**Output:**

