1 Write a java program that inserts a node into its proper sorted positioninasorted linked list.

```
public void insertAfter(Node prev_node,
                        int new_data)
{
    // 1. Check if the given Node is null
    if (prev_node == null)
    {
        System.out.println(
               "The given previous node cannot be null");
        return;
    }

    /* 2. Allocate the Node &
       3. Put in the data*/
    Node new_node = new Node(new_data);

    // 4. Make next of new Node as next
    // of prev_node
    new_node.next = prev_node.next;

    // 5. make next of prev_node as new_node
    prev_node.next = new_node;
}
```

2 Write a java program to compute the height of the binary tree.

```
public class BinaryTree {

    //Represent the node of binary tree
    public static class Node{
        int data;
        Node left;
        Node right;

        public Node(int data){
            //Assign data to the new node, set left and right children to null
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }

    //Represent the root of binary tree
    public Node root;
    public BinaryTree(){
        root = null;
    }

    //findHeight() will determine the maximum height of the binary tree
    public int findHeight(Node temp){
        //Check whether tree is empty
        if(root == null) {
             System.out.println("Tree is empty");
            return 0;
        }
```

```
        else {
            int leftHeight = 0, rightHeight = 0;

            //Calculate the height of left subtree
            if(temp.left != null)
                leftHeight = findHeight(temp.left);

            //Calculate the height of right subtree
            if(temp.right != null)
                rightHeight = findHeight(temp.right);

            //Compare height of left subtree and right subtree
            //and store maximum of two in variable max
            int max = (leftHeight > rightHeight) ? leftHeight : rightHeight;

            //Calculate the total height of tree by adding height of root
            return (max + 1);
        }
     }

    public static void main(String[] args) {

        BinaryTree bt = new BinaryTree();
        //Add nodes to the binary tree
        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.right.left = new Node(5);
        bt.root.right.right = new Node(6);
        bt.root.right.right.right= new Node(7);
        bt.root.right.right.right.right = new Node(8);

        //Display the maximum height of the given binary tree
        System.out.println("Maximum height of given binary tree: " +
bt.findHeight(bt.root));
  }
}
```

3 Write a java program to determine whether a given binary tree is a BSTor not

```
class GFG {

/* A binary tree node has data, pointer to left child
            and a pointer to right child */
static class node {
      int data;
      node left, right;
}

/* Helper function that allocates a new node with the
            given data and NULL left and right pointers. */
static node newNode(int data)
{
      node Node = new node();
```

```
        Node.data = data;
        Node.left = Node.right = null;

        return Node;
}

static int maxValue(node Node)
{
        if (Node == null) {
        return Integer.MIN_VALUE;
        }
        int value = Node.data;
        int leftMax = maxValue(Node.left);
        int rightMax = maxValue(Node.right);

        return Math.max(value, Math.max(leftMax, rightMax));
}

static int minValue(node Node)
{
        if (Node == null) {
        return Integer.MAX_VALUE;
        }
        int value = Node.data;
        int leftMax = minValue(Node.left);
        int rightMax = minValue(Node.right);

        return Math.min(value, Math.min(leftMax, rightMax));
}

/* Returns true if a binary tree is a binary search tree
        */
static int isBST(node Node)
{
        if (Node == null) {
        return 1;
        }

        /* false if the max of the left is > than us */
        if (Node.left != null
                && maxValue(Node.left) > Node.data) {
        return 0;
        }

        /* false if the min of the right is <= than us */
        if (Node.right != null
                && minValue(Node.right) < Node.data) {
        return 0;
        }

        /* false if, recursively, the left or right is not a
                * BST*/
        if (isBST(Node.left) != 1
                || isBST(Node.right) != 1) {
        return 0;
        }

        /* passing all that, it's a BST */
        return 1;
```

```
}

public static void main(String[] args)
{
      node root = newNode(4);
      root.left = newNode(2);
      root.right = newNode(5);

      // root->right->left = newNode(7);
      root.left.left = newNode(1);
      root.left.right = newNode(3);

      // Function call
      if (isBST(root) == 1) {
      System.out.print("Is BST");
      }
      else {
      System.out.print("Not a BST");
      }
}
}
```

4 Write a java code to Check the given below expression is balancedor not . (using stack)
{ { [[(( ))]) } }

```
class Main
{
    // Function to check if the given expression is balanced or not
    public static boolean isBalanced(String exp)
    {
        // base case: length of the expression must be even
        if (exp == null || exp.length() % 2 == 1) {
            return false;
        }

        // take an empty stack of characters
        Stack<Character> stack = new Stack<>();

        // traverse the input expression
        for (char c: exp.toCharArray())
        {
            // if the current character in the expression is an opening brace,
            // push it into the stack
            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
            }

            // if the current character in the expression is a closing brace
            if (c == ')' || c == '}' || c == ']')
            {
                // return false if a mismatch is found (i.e., if the stack is
empty,
                // the expression cannot be balanced since the total number of
opening
                // braces is less than the total number of closing braces)
                if (stack.empty()) {
```

```
                return false;
            }

            // pop character from the stack
            Character top = stack.pop();

            // if the popped character is not an opening brace or does not pair
            // with the current character of the expression
            if ((top == '(' && c != ')') || (top == '{' && c != '}')
                    || (top == '[' && c != ']')) {
                return false;
            }
        }
    }

    // the expression is balanced only when the stack is empty at this point
    return stack.empty();
}

public static void main(String[] args)
{
    String exp = "{()}[{}]";

    if (isBalanced(exp)) {
        System.out.println("The expression is balanced");
    }
    else {
        System.out.println("The expression is not balanced");
    }
}
```

5 Write a java program to Print left view of a binary tree using queue

```
import java.util.ArrayDeque;
import java.util.Queue;

// A class to store a binary tree node
class Node
{
    int key;
    Node left = null, right = null;

    Node(int key) {
        this.key = key;
    }
}

class Main
{
    // Iterative function to print the left view of a given binary tree
    public static void leftView(Node root)
    {
        // return if the tree is empty
        if (root == null) {
            return;
        }

        // create an empty queue and enqueue the root node
        Queue<Node> queue = new ArrayDeque<>();
```

```
        queue.add(root);

        // to store the current node
        Node curr;

        // loop till queue is empty
        while (!queue.isEmpty())
        {
            // calculate the total number of nodes at the current level
            int size = queue.size();
            int i = 0;

            // process every node of the current level and enqueue their
            // non-empty left and right child
            while (i++ < size)
            {
                curr = queue.poll();

                // if this is the first node of the current level, print it
                if (i == 1) {
                    System.out.print(curr.key + " ");
                }

                if (curr.left != null) {
                    queue.add(curr.left);
                }

                if (curr.right != null) {
                    queue.add(curr.right);
                }
            }
        }
    }

    public static void main(String[] args)
    {
        Node root = new Node(1);
        root.left = new Node(2);
        root.right = new Node(3);
        root.left.right = new Node(4);
        root.right.left = new Node(5);
        root.right.right = new Node(6);
        root.right.left.left = new Node(7);
        root.right.left.right = new Node(8);

        leftView(root);
    }
}
```