Q.1 :

Ans:

```
class Node {
   int data;
   Node next;

   public Node(int data) {
      this.data = data;
      this.next = null;
   }
}

class SortedLinkedList {
   Node head;

   public SortedLinkedList() {
      this.head = null;
   }

   // Method to insert a node into its proper sorted position
   public void insert(int data) {
      Node newNode = new Node(data);

      if (head == null || data < head.data) {
         newNode.next = head;
         head = newNode;
      } else {
         Node current = head;
```

```
        while (current.next != null && current.next.data < data) {
            current = current.next;
        }

        newNode.next = current.next;
        current.next = newNode;
    }
}

// Method to print the linked list
public void display() {
    Node current = head;

    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }

    System.out.println();
}

public static void main(String[] args) {
    SortedLinkedList sortedList = new SortedLinkedList();

    sortedList.insert(5);
    sortedList.insert(10);
    sortedList.insert(2);
    sortedList.insert(8);
```

```
        System.out.println("Sorted Linked List:");
        sortedList.display();
    }
}
```

Q.2 :

```
class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}

class BinaryTree {
    Node root;

    public BinaryTree() {
        root = null;
    }

    // Method to compute the height of the binary tree
    public int height(Node root) {
        if (root == null) {
            return 0;
        } else {
            int leftHeight = height(root.left);
```

```
        int rightHeight = height(root.right);

        // Return the height of the taller subtree plus 1
        return Math.max(leftHeight, rightHeight) + 1;
      }
    }

    public static void main(String[] args) {
      BinaryTree tree = new BinaryTree();

      // Construct a sample binary tree
      tree.root = new Node(1);
      tree.root.left = new Node(2);
      tree.root.right = new Node(3);
      tree.root.left.left = new Node(4);
      tree.root.left.right = new Node(5);
      tree.root.right.left = new Node(6);
      tree.root.right.right = new Node(7);

      int treeHeight = tree.height(tree.root);
      System.out.println("Height of the binary tree is: " + treeHeight);
    }
}
```

Q.3 :

Ans:

```
class Node {
    int data;
    Node left, right;
```

```
    public Node(int item) {
        data = item;
        left = right = null;
    }
}

class BinaryTree {
    Node root;

    public BinaryTree() {
        root = null;
    }

    // Helper method to check if the tree is a BST
    private boolean isBSTUtil(Node node, int min, int max) {
        if (node == null) {
            return true;
        }

        // Check if the current node's data is within the valid range
        if (node.data < min || node.data > max) {
            return false;
        }

        // Recursively check the left and right subtrees with updated ranges
        return isBSTUtil(node.left, min, node.data - 1) && isBSTUtil(node.right, node.data + 1, max);
    }
```

```
    // Method to check if the tree is a BST
    public boolean isBST() {
        return isBSTUtil(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }

    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();

        // Construct a sample binary tree
        tree.root = new Node(2);
        tree.root.left = new Node(1);
        tree.root.right = new Node(3);

        // Check if the tree is a BST
        boolean isBST = tree.isBST();

        if (isBST) {
            System.out.println("The binary tree is a BST.");
        } else {
            System.out.println("The binary tree is not a BST.");
        }
    }
}
```

Q.4:

Ans:

```
import java.util.Stack;

public class BalancedExpression {
```

```
// Method to check if the given expression is balanced
public static boolean isBalanced(String expression) {
    Stack<Character> stack = new Stack<>();

    // Iterate through each character in the expression
    for (char ch : expression.toCharArray()) {
        if (ch == '{' || ch == '[' || ch == '(') {
            // If opening bracket, push onto the stack
            stack.push(ch);
        } else if (ch == '}' || ch == ']' || ch == ')') {
            // If closing bracket, check if the stack is empty
            if (stack.isEmpty()) {
                return false;
            }

            // Pop the top element from the stack
            char top = stack.pop();

            // Check if the popped bracket matches the current closing bracket
            if (!((top == '{' && ch == '}') || (top == '[' && ch == ']') || (top == '(' && ch == ')'))) {
                return false;
            }
        }
    }

    // Check if the stack is empty after processing all characters
    return stack.isEmpty();
}
```

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

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

Q.5:

Ans:

```
import java.util.LinkedList;
import java.util.Queue;

class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}

class BinaryTree {
    Node root;
```

```
public BinaryTree() {
    root = null;
}

// Method to print the left view of the binary tree
public void leftView() {
    if (root == null) {
        return;
    }

    Queue<Node> queue = new LinkedList<>();
    queue.add(root);

    while (!queue.isEmpty()) {
        // Get the number of nodes at the current level
        int levelSize = queue.size();

        for (int i = 0; i < levelSize; i++) {
            Node current = queue.poll();

            // Print the first node at each level (leftmost node)
            if (i == 0) {
                System.out.print(current.data + " ");
            }

            // Add left and right children to the queue
            if (current.left != null) {
                queue.add(current.left);
            }
```

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

public static void main(String[] args) {
    BinaryTree tree = new BinaryTree();

    // Construct a sample binary tree
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.right = new Node(4);
    tree.root.right.left = new Node(5);
    tree.root.right.right = new Node(6);
    tree.root.right.left.left = new Node(7);

    System.out.println("Left view of the binary tree:");
    tree.leftView();
}
}
```