Worksheet 6

Q1.

```
class LinkedList {
    Node head;

    class Node {
        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    void sortedInsert(int newData) {
        Node newNode = new Node(newData);

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

```
    }

    void printList() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        LinkedList list = new LinkedList();

        list.sortedInsert(5);
        list.sortedInsert(10);
        list.sortedInsert(7);
        list.sortedInsert(3);
        list.sortedInsert(1);
        list.sortedInsert(9);

        System.out.println("Linked List after inserting nodes in sorted order:");
        list.printList();
    }
}
```

Explanation:

1. **Node Class:** Represents each node in the linked list with a data field and a reference to the next node.
2. **sortedInsert Method:**

- Inserts a new node into its proper position to maintain the sorted order.
- If the list is empty or the new data is smaller than the head, the new node becomes the new head.
- Otherwise, it finds the correct position by traversing the list and inserts the new node.

3. **printList Method:** Prints the linked list elements.
4. **Main Method:**
   - Creates a linked list and inserts elements in a way that they maintain sorted order.
   - Prints the final sorted linked list.

Q2.

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

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

   Node root;

   int computeHeight(Node node) {
      if (node == null) {
         return 0;
      } else {
         int leftHeight = computeHeight(node.left);
```

```
        int rightHeight = computeHeight(node.right);

        return Math.max(leftHeight, rightHeight) + 1;
    }
  }

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

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

Explanation:

1. **Node Class:**
    - Represents each node in the binary tree with data, left, and right children.
2. **computeHeight Method:**
    - Recursively calculates the height of the binary tree.
    - The base case is when the node is null, returning a height of 0.
    - For non-null nodes, it computes the height of the left and right subtrees and returns the maximum of the two heights plus one (to account for the current node).
3. **Main Method:**
    - Constructs a sample binary tree.
    - Calls the computeHeight method and prints the height of the binary tree.

Q3.

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

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

    Node root;

    boolean isBST() {
        return isBSTUtil(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }

    boolean isBSTUtil(Node node, int min, int max) {
        if (node == null) {
            return true;
        }

        if (node.data <= min || node.data >= max) {
            return false;
        }

        return isBSTUtil(node.left, min, node.data) &&
```

```
            isBSTUtil(node.right, node.data, max);
    }

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

        tree.root = new Node(4);
        tree.root.left = new Node(2);
        tree.root.right = new Node(5);
        tree.root.left.left = new Node(1);
        tree.root.left.right = new Node(3);

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

Explanation:

1. **Node Class:**
    - Represents each node in the binary tree with fields data, left, and right.
2. **isBST Method:**
    - This is a utility method that initiates the check by calling isBSTUtil with the initial range of possible values (Integer.MIN_VALUE to Integer.MAX_VALUE).
3. **isBSTUtil Method:**
    - This helper method checks whether the binary tree rooted at the given node is a BST.
    - It checks whether each node's value is within a valid range:
        - For the left child, the range is min to node.data.

        - For the right child, the range is node.data to max.
    - The method returns true if both subtrees are valid BSTs; otherwise, it returns false.

4. **Main Method:**
    - Constructs a sample binary tree and checks if it is a BST by calling the isBST method.

Q4.

```
import java.util.Stack;


public class BalancedExpression {
   static boolean isBalanced(String expression) {
      Stack<Character> stack = new Stack<>();


      for (char ch : expression.toCharArray()) {
         if (ch == '{' || ch == '[' || ch == '(') {
            stack.push(ch);
         } else if (ch == '}' || ch == ']' || ch == ')') {
            if (stack.isEmpty()) {
               return false;
            }


            char top = stack.pop();
            if ((ch == '}' && top != '{') || (ch == ']' && top != '[') || (ch == ')' && top != '(')) {
               return false;
            }
         }
      }


      return stack.isEmpty();
```

```
    }

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

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

Explanation:

1. **Stack Data Structure:**
    - The program uses a stack to keep track of opening brackets ({, [, ().
    - When a closing bracket is encountered, the program checks if it matches the top of the stack (i.e., the most recent unmatched opening bracket).
2. **isBalanced Method:**
    - The method iterates over each character in the expression.
    - For each opening bracket, it pushes it onto the stack.
    - For each closing bracket, it checks if the stack is empty (which would indicate an unbalanced expression) and whether the top of the stack matches the closing bracket.
    - After processing the entire expression, the stack should be empty for a balanced expression.
3. **Main Method:**
    - Defines the given expression {{[[(())]]}}.
    - Calls the isBalanced method and prints whether the expression is balanced.

Q5.

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

```
import java.util.Queue;

class BinaryTreeLeft {
    static class Node {
        int data;
        Node left, right;

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

    Node root;

    void printLeftView() {
        if (root == null) {
            return;
        }

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

        while (!queue.isEmpty()) {
            int numberOfNodes = queue.size();

            for (int i = 0; i < numberOfNodes; i++) {
                Node currentNode = queue.poll();
```

```
            if (i == 0) {
                System.out.print(currentNode.data + " ");
            }

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

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

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

    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.right = new Node(6);
    tree.root.left.right.left = new Node(7);

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

}

Explanation:

1. **Node Class:**
    - Represents each node in the binary tree with fields data, left, and right.
2. **printLeftView Method:**
    - This method prints the left view of the binary tree using a queue.
    - A queue is used to perform a level-order traversal (breadth-first search) of the tree.
    - At each level, the first node encountered is printed because it represents the leftmost node at that level.
3. **Main Method:**
    - Constructs a sample binary tree.
    - Calls the printLeftView method to print the left view of the binary tree.