Worksheet - 6

Ques 1. Write a java program that inserts a node into its proper sorted position in a sorted linked list.

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

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

// LinkedList class
class LinkedList {
    Node head;

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

    // Method to insert a node in sorted order
    public void insertInSortedOrder(int data) {
        Node newNode = new Node(data);

        // If the list is empty or the new node should be inserted at the head
        if (head == null || head.data >= newNode.data) {
            newNode.next = head;
            head = newNode;
        } else {
            // Locate the node before the point of insertion
            Node current = head;
            while (current.next != null && current.next.data < newNode.data) {
                current = current.next;
            }
            newNode.next = current.next;
            current.next = newNode;
        }
```

```java
    }

    // Method to print the linked list
    public void printList() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }
}

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

        // Inserting nodes into the linked list
        list.insertInSortedOrder(10);
        list.insertInSortedOrder(5);
        list.insertInSortedOrder(20);
        list.insertInSortedOrder(15);

        // Print the sorted linked list
        list.printList();  // Output should be: 5 10 15 20
    }
}
```

Output : 5 10 15 20

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

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

    public Node(int item) {
```

```
        data = item;
        left = right = null;
    }
}

// BinaryTree class
class BinaryTree {
    Node root;

    public BinaryTree() {
        root = null;
    }

    // Method to compute the height of the binary tree
    public int height(Node node) {
        if (node == null) {
            return 0;
        }

        // Compute the height of each subtree
        int leftHeight = height(node.left);
        int rightHeight = height(node.right);

        // Return the larger one and add 1 (for the current node)
        return Math.max(leftHeight, rightHeight) + 1;
    }

    // Method to start the height computation from the root
    public int height() {
        return height(root);
    }
}

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

        // Creating a 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);

        // Computing the height of the binary tree
        System.out.println("The height of the binary tree is: " + tree.height());
    }
}
```

Output : The height of the binary tree is: 3

```
      1
     / \
    2   3
   / \
  4   5
```

Ques 3. Write a java program to determine whether a given binary tree is a BST or not.

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

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

// BinaryTree class
class BinaryTree {
    Node root;

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

```
    // Method to check if the tree is a BST
    boolean isBST(Node node, int min, int max) {
        // Base case: empty tree is a BST
        if (node == null) {
            return true;
        }

        // If this node violates the min/max constraints, return false
        if (node.data <= min || node.data >= max) {
            return false;
        }

        // Recursively check the left and right subtrees with updated constraints
        return isBST(node.left, min, node.data) && isBST(node.right, node.data, max);
    }

    // Overloaded method to start the check from the root
    boolean isBST() {
        return isBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }
}

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

        // Creating a binary tree that is also a BST
        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);

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

```
    }
}
```

Output : The binary tree is a BST.

```
    4
   / \
  2   5
 / \
1   3
```

Ques 4. Write a java code to Check the given expression below is balanced or not . (using stack)

{ { [[(( ))]] } }

```
import java.*;

public class BalancedExpressionChecker {

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

        // Traverse through the expression
        for (int i = 0; i < expression.length(); i++) {
            char currentChar = expression.charAt(i);

            // If currentChar is an opening bracket, push it onto the stack
            if (currentChar == '{' || currentChar == '[' || currentChar == '(') {
                stack.push(currentChar);
            }
            // If currentChar is a closing bracket
            else if (currentChar == '}' || currentChar == ']' || currentChar == ')') {
                // If the stack is empty or the top of the stack doesn't match the current closing
bracket, return false
                if (stack.isEmpty()) {
                    return false;
                }

                char topChar = stack.pop();
```

```java
            if (!isMatchingPair(topChar, currentChar)) {
                return false;
            }
        }
    }

    // If stack is empty, all opening brackets were matched; hence, the expression is balanced
    return stack.isEmpty();
}

// Method to check if the opening and closing brackets are a matching pair
private static boolean isMatchingPair(char opening, char closing) {
    return (opening == '{' && closing == '}') ||
           (opening == '[' && closing == ']') ||
           (opening == '(' && closing == ')');
}

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

    // Check if the expression is balanced and print the result
    if (isBalanced(expression)) {
        System.out.println("The expression is balanced.");
    } else {
        System.out.println("The expression is NOT balanced.");
    }
}
}
```

Output : The expression is NOT balanced.

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

```java
import java.util.*;

// Node class representing a tree node
class Node {
    int data;
```

```
    Node left, right;

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

// BinaryTree class containing methods to work with the binary tree
class BinaryTree {
    Node root;

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

        // Queue to hold nodes at each level
        Queue<Node> queue = new LinkedList<>();
        queue.add(root);

        while (!queue.isEmpty()) {
            // Number of nodes at the current level
            int nodeCount = queue.size();

            // Traverse all nodes of the current level
            for (int i = 0; i < nodeCount; i++) {
                Node currentNode = queue.poll();

                // Print the first node of each level (left view)
                if (i == 0) {
                    System.out.print(currentNode.data + " ");
                }

                // Enqueue left child
                if (currentNode.left != null) {
                    queue.add(currentNode.left);
                }
```

```
                // Enqueue right child
                if (currentNode.right != null) {
                    queue.add(currentNode.right);
                }
            }
        }
    }
}

// Main class to run the program
public class Main {
    public static void main(String[] args) {
        // Create a sample binary tree
        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);
        tree.root.right.left = new Node(6);
        tree.root.right.right = new Node(7);
        tree.root.left.left.left = new Node(8);

        // Print the left view of the binary tree
        System.out.print("Left view of the binary tree: ");
        tree.printLeftView();
    }
}
```

The tree is:

```
    1
   / \
  2   3
 / \ / \
4  5 6  7
```

Output : Left view of the binary tree: 1 2 4 8