FULL STACK DEVELOPMENT – WORKSHEET – 6

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

Algorithm:
Let input linked list is sorted in increasing order.
1) If Linked list is empty then make the node as head and return it.

2) If the value of the node to be inserted is smaller than the value of the head node, then insert the node at the start and make it head.

3) In a loop, find the appropriate node after which the input node (let 9) is to be inserted.

To find the appropriate node start from the head, keep moving until you reach a node GN (10 in the below diagram) who's value is greater than the input node. The node just before GN is the appropriate node (7).

4) Insert the node (9) after the appropriate node (7) found in step 3.

```
// Java Program to insert in a sorted list

class LinkedList {

	Node head; // head of list


	/* Linked list Node*/

	class Node {

		int data;

		Node next;

		Node(int d)

		{

			data = d;
```

```
        next = null;
    }
}

/* function to insert a new_node in a list. */
void sortedInsert(Node new_node)
{
    Node current;

    /* Special case for head node */
    if (head == null || head.data >= new_node.data) {
        new_node.next = head;
        head = new_node;
    }
    else {

        /* Locate the node before point of insertion. */
        current = head;

        while (current.next != null && current.next.data < new_node.data) {

        current = current.next;
        }

        new_node.next = current.next;
```

```
            current.next = new_node;
        }
    }

    /*Utility functions*/

    /* Function to create a node */
    Node newNode(int data)
    {
        Node x = new Node(data);
        return x;
    }

    /* Function to print linked list */
    void printList()
    {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
    }

    /* Driver function to test above methods */
    public static void main(String args[])
```

```
    {
        LinkedList llist = new LinkedList();
        Node new_node;
        new_node = llist.newNode(5);
        llist.sortedInsert(new_node);
        new_node = llist.newNode(10);
        llist.sortedInsert(new_node);
        new_node = llist.newNode(7);
        llist.sortedInsert(new_node);
        new_node = llist.newNode(3);
        llist.sortedInsert(new_node);
        new_node = llist.newNode(1);
        llist.sortedInsert(new_node);
        new_node = llist.newNode(9);
        llist.sortedInsert(new_node);
        System.out.println("Created Linked List");
        llist.printList();
    }
}
```

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

Follow the below steps to Implement the idea:

- Recursively do a Depth-first search.
- If the tree is empty then return 0
- Otherwise, do the following
    - Get the max depth of the left subtree recursively i.e. call maxDepth(tree->left-subtree)
    - Get the max depth of the right subtree recursively i.e. call maxDepth(tree->right-subtree)

    - Get the max of max depths of **left** and **right** subtrees and **add 1** to it for the current node.
    - Max_depth=max(max depth of the left subtree , max depth of the right subtree)+1

- Return max_depth.

```
// Java program to find height of tree


// A binary tree node

class Node {

	int data;

	Node left, right;


	Node(int item)

	{

		data = item;

		left = right = null;

	}

}


class BinaryTree {

	Node root;


	/* Compute the "maxDepth" of a tree -- the number of

	nodes along the longest path from the root node

	down to the farthest leaf node.*/
```

```java
int maxDepth(Node node)
{
    if (node == null)
        return 0;
    else {
        /* compute the depth of each subtree */
        int lDepth = maxDepth(node.left);
        int rDepth = maxDepth(node.right);

        /* use the larger one */
        if (lDepth > rDepth)
            return (lDepth + 1);
        else
            return (rDepth + 1);
    }
}

/* Driver program to test above functions */
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);


		System.out.println("Height of tree is "

							+ tree.maxDepth(tree.root));

	}

}
```

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

- If the current node is null then return true
- If the value of the left child of the node is greater than or equal to the current node then return false
- If the value of the right child of the node is less than or equal to the current node then return false
- If the left subtree or the right subtree is not a BST then return false
- Else return true

Below is the implementation of the above approach:

```
// Java implementation for the above approach

import java.io.*;


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");
        }
}
}
```

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

Follow the steps mentioned below to implement the idea:

- Declare a character stack (say **temp**).
- Now traverse the string exp.
    - If the current character is a starting bracket (**'(' or '{' or '['**) then push it to stack.
    - If the current character is a closing bracket (**')' or '}' or ']'**) then pop from the stack and if the popped character is the matching starting bracket then fine.
    - Else brackets are **Not Balanced**.
- After complete traversal, if some starting brackets are left in the stack then the expression is **Not balanced**, else **Balanced**.

Below is the implementation of the above approach:

```
// Java program for checking
// balanced brackets
import java.util.*;
```

```
public class BalancedBrackets {

    // function to check if brackets are balanced
    static boolean areBracketsBalanced(String expr)
    {
        // Using ArrayDeque is faster than using Stack class
        Deque<Character> stack
            = new ArrayDeque<Character>();

        // Traversing the Expression
        for (int i = 0; i < expr.length(); i++) {
            char x = expr.charAt(i);

            if (x == '(' || x == '[' || x == '{') {
                // Push the element in the stack
                stack.push(x);
                continue;
            }

            // If current character is not opening
            // bracket, then it must be closing. So stack
            // cannot be empty at this point.
            if (stack.isEmpty())
                return false;
            char check;
```

```
        switch (x) {
        case ')':
            check = stack.pop();
            if (check == '{' || check == '[')
                return false;
            break;

        case '}':
            check = stack.pop();
            if (check == '(' || check == '[')
                return false;
            break;

        case ']':
            check = stack.pop();
            if (check == '(' || check == '{')
                return false;
            break;
        }
    }

    // Check Empty Stack
    return (stack.isEmpty());
}
```

```
    // Driver code
    public static void main(String[] args)
    {
        String expr = "([{}])";

        // Function call
        if (areBracketsBalanced(expr))
            System.out.println("Balanced ");
        else
            System.out.println("Not Balanced ");
    }
}
```

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

```
// Java program to print left view of binary tree

/* Class containing left and right child of current
node and key value*/
class Node {
    int data;
    Node left, right;

    public Node(int item)
    {
```

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

/* Class to print the left view */
class BinaryTree {
    Node root;
    static int max_level = 0;

    // recursive function to print left view
    void leftViewUtil(Node node, int level)
    {
        // Base Case
        if (node == null)
            return;

        // If this is the first node of its level
        if (max_level < level) {
            System.out.print(node.data + " ");
            max_level = level;
        }

        // Recur for left and right subtrees
        leftViewUtil(node.left, level + 1);
```

```
        leftViewUtil(node.right, level + 1);
    }

    // A wrapper over leftViewUtil()
    void leftView()
    {
        max_level = 0;
        leftViewUtil(root, 1);
    }

    /* testing for example nodes */
    public static void main(String args[])
    {
        /* creating a binary tree and entering the nodes */
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(10);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(7);
        tree.root.left.right = new Node(8);
        tree.root.right.right = new Node(15);
        tree.root.right.left = new Node(12);
        tree.root.right.right.left = new Node(14);

        tree.leftView();
```

```
        }
    }
```