FULL STACK DEVELOPMENT – WORKSHEET-A- ANS

1.Program Output:

The modified linked list is:
1 —> 2 —> 3 —> 4 —> 5 —> 6 —> 7 —> 8 —> 9 —> null

```
package com.java.sortedlinkedlist;
//A Linked List Node
class Node
{
 int data;
 Node next;

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

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

class SortedElementLinkedList
{
 // Helper function to print a given linked list
 public static void printList(Node head)
 {
     Node ptr = head;
     while (ptr != null)
     {
         System.out.print(ptr.data + " —> ");
         ptr = ptr.next;
     }

     System.out.println("null");
 }

 // Function to insert a given node at its correct sorted position into
 // a given list sorted in increasing order
 public static Node sortedInsert(Node head, Node newNode)
 {
     // special case for the head end
     if (head == null || head.data >= newNode.data)
     {
         newNode.next = head;
         head = newNode;
         return head;
     }

     // 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;

     return head;
 }

 public static void main(String[] args)
 {
     // input keys
     int[] keys = {2,3, 4, 6,7,8};

     // points to the head node of the linked list
     Node head = null;

     // construct a linked list
     for (int i = keys.length - 1; i >= 0; i--) {
         head = new Node(keys[i], head);
     }

     head = sortedInsert(head, new Node(5));
     head = sortedInsert(head, new Node(9));
     head = sortedInsert(head, new Node(1));

     // print linked list
     printList(head);
 }
}
```

2. Program Output:

The height of binary tree is: 4

```
package com.java.heightofBinarytree;

/*Defining a class to store a node of the binary tree.*/
class Node {
    int data;
    Node left, right;
    /* The constructor will add nodes to the binary tree. Its left and the right
      pointer will initially point to NULL as there is no child currently */
    Node(int item) {
        data = item;
        left = right = null;
    }
}

public class BinaryTree {
    // creating reference of the Node class.
    Node root;

    /*
    A recursive function that finds the height of the binary tree
    by maximum height between the left and right subtree. */
    int findHeight(Node node) {

        // Base case: If the tree is empty, return -1 as we cannot find its
height.

        if (node == null)
```

```
            return 0;

        else {
            /* Call the findHeight() function for the left and right sub tree and
store the results.
            */
            int leftHeight = findHeight(node.left);
            int rightHeight = findHeight(node.right);

            // returning the (maximum + 1) as the height of the binary tree.
            if (leftHeight > rightHeight)
                return (leftHeight + 1);
            else
                return (rightHeight + 1);
        }
    }

    public static void main(String[] args) {

        // creating object of the BinaryTree class.
        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(9);
        tree.root.right.right=new Node(10);
        tree.root.right.right.left=new Node(20);

        System.out.println("The height of binary tree is: " +
                tree.findHeight(tree.root));
    }
}
```

3. Program Output:

IT Is a BST

```
package com.java.binarysearchtree;

class BST {

  /* 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.left.left = newNode(1);
    root.left.right = newNode(3);

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

4. Program Output:

The expression is not balanced

```
package com.java.BalancedBrackets;
import java.util.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 == '[' || c=='[' || c=='(' || c==')') {
                stack.push(c);
            }

            // if the current character in the expression is a closing brace
            if (c == ')' ||c == ')' ||c == ']' || 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.Program Output:

```
The left view is:
1 2 4 8

package com.java.leftviewbyQueue;

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.left=new Node(4);
        root.left.right = new Node(5);
        root.right.left = new Node(6);
        root.right.right = new Node(7);
        root.right.left.left = new Node(8);
        root.right.left.right = new Node(9);

   System.out.println("The left view is:");
        leftView(root);
    }
}
```