**Question - 1**

Solution:

import java.util.Stack;

public class BalancedBrackets {

public static boolean isBalanced(String str) {

Stack<Character> stack = new Stack<>();

// Iterate through the characters in the string

for (char ch : str.toCharArray()) {

if (ch == '(' || ch == '[' || ch == '{') {

// If an opening bracket is encountered, push it onto the stack

stack.push(ch);

} else if (ch == ')' || ch == ']' || ch == '}') {

// If a closing bracket is encountered

// Check if the stack is empty (no matching opening bracket)

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 ((ch == ')' && top != '(') || (ch == ']' && top != '[') || (ch == '}' && top != '{')) {

return false;

}

}

}

// If the stack is empty at the end, all brackets were balanced

return stack.isEmpty();

}

public static void main(String[] args) {

String input1 = "([{}])";

String input2 = "([{}]) )";

System.out.println("Sample Input 1:");

System.out.println(input1);

System.out.println("Sample Output 1:");

if (isBalanced(input1)) {

System.out.println("The entered String has Balanced Brackets");

} else {

System.out.println("The entered String does not contain Balanced Brackets");

}

System.out.println("\nSample Input 2:");

System.out.println(input2);

System.out.println("Sample Output 2:");

if (isBalanced(input2)) {

System.out.println("The entered String has Balanced Brackets");

} else {

System.out.println("The entered String does not contain Balanced Brackets");

}

}

}

**Question – 2**

Solution:

class TreeNode {

int data;

TreeNode left;

TreeNode right;

public TreeNode(int data) {

this.data = data;

this.left = null;

this.right = null;

}

}

public class BSTPairWithSum {

private TreeNode root;

public void insert(int data) {

root = insertRec(root, data);

}

private TreeNode insertRec(TreeNode root, int data) {

if (root == null) {

root = new TreeNode(data);

return root;

}

if (data < root.data) {

root.left = insertRec(root.left, data);

} else if (data > root.data) {

root.right = insertRec(root.right, data);

}

return root;

}

public boolean findPair(TreeNode root, int sum) {

if (root == null) {

return false;

}

// Create two pointers for in-order traversal

TreeNode left = getSmallestNode(root);

TreeNode right = getLargestNode(root);

while (left.data < right.data) {

int currentSum = left.data + right.data;

if (currentSum == sum) {

System.out.println("Pair is (" + left.data + ", " + right.data + ")");

return true;

}

if (currentSum < sum) {

left = getInorderSuccessor(root, left);

} else {

right = getInorderPredecessor(root, right);

}

}

System.out.println("Nodes are not found.");

return false;

}

private TreeNode getSmallestNode(TreeNode node) {

while (node.left != null) {

node = node.left;

}

return node;

}

private TreeNode getLargestNode(TreeNode node) {

while (node.right != null) {

node = node.right;

}

return node;

}

private TreeNode getInorderSuccessor(TreeNode root, TreeNode node) {

if (node.right != null) {

return getSmallestNode(node.right);

}

TreeNode successor = null;

while (root != null) {

if (node.data < root.data) {

successor = root;

root = root.left;

} else if (node.data > root.data) {

root = root.right;

} else {

break;

}

}

return successor;

}

private TreeNode getInorderPredecessor(TreeNode root, TreeNode node) {

if (node.left != null) {

return getLargestNode(node.left);

}

TreeNode predecessor = null;

while (root != null) {

if (node.data > root.data) {

predecessor = root;

root = root.right;

} else if (node.data < root.data) {

root = root.left;

} else {

break;

}

}

return predecessor;

}

public static void main(String[] args) {

BSTPairWithSum tree = new BSTPairWithSum();

tree.insert(4);

tree.insert(2);

tree.insert(6);

tree.insert(1);

tree.insert(3);

tree.insert(5);

tree.insert(7);

int sum = 130;

if (!tree.findPair(tree.root, sum)) {

System.out.println("Pair not found.");

}

}

}