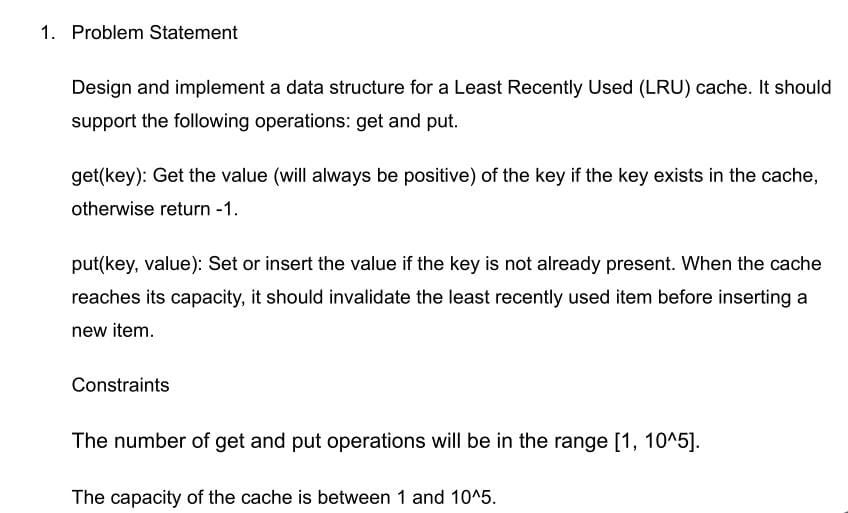
ASSIGNMENT



ANSWER:- import java.util.HashMap;

import java.util.Map;

public class LRUCache {

private class Node {

int key, value;

Node prev, next;

Node(int key, int value) {

this.key = key;

this.value = value;

}

}

private final int capacity;

private final Map<Integer, Node> cache;

private final Node head, tail;

public LRUCache(int capacity) {

this.capacity = capacity;

this.cache = new HashMap<>(capacity);

this.head = new Node(-1, -1);

this.tail = new Node(-1, -1);

head.next = tail;

tail.prev = head;

}

public int get(int key) {

Node node = cache.get(key);

if (node == null) return -1;

moveToHead(node);

return node.value;

}

public void put(int key, int value) {

Node node = cache.get(key);

if (node == null) {

if (cache.size() >= capacity) {

Node tailPrev = tail.prev;

cache.remove(tailPrev.key);

removeNode(tailPrev);

}

Node newNode = new Node(key, value);

cache.put(key, newNode);

addNode(newNode);

} else {

node.value = value;

moveToHead(node);

}

}

private void addNode(Node node) {

node.next = head.next;

node.prev = head;

head.next.prev = node;

head.next = node;

}

private void removeNode(Node node) {

Node prev = node.prev;

Node next = node.next;

prev.next = next;

next.prev = prev;

}

private void moveToHead(Node node) {

removeNode(node);

addNode(node);

}

public static void main(String[] args) {

LRUCache cache = new LRUCache(2);

cache.put(1, 1);

cache.put(2, 2);

System.***out***.println(cache.get(1));

cache.put(3, 3);

System.***out***.println(cache.get(2));

cache.put(4, 4);

System.***out***.println(cache.get(1));

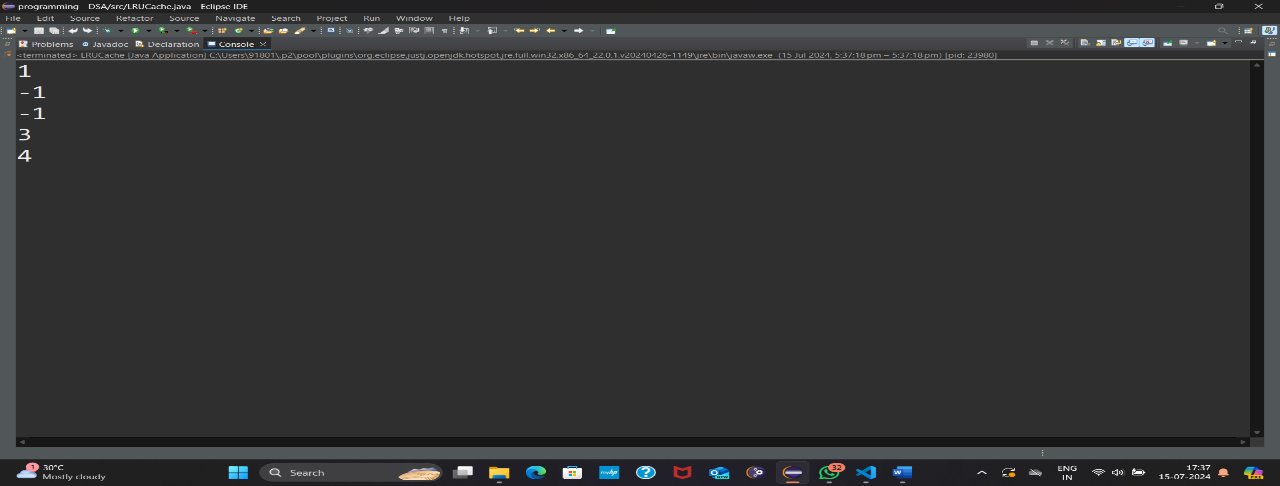
System.***out***.println(cache.get(3));

System.***out***.println(cache.get(4));

}

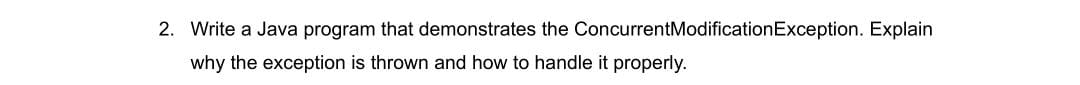
}

OUTPUT:-



 **Time Complexity**: O(1)

 **Space Complexity**: O(n)



ANSWER:- import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

public class ConcurrentModificationDemo {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("A");

list.add("B");

list.add("C");

System.***out***.println("Original list: " + list);

try {

for (String s : list) {

if (s.equals("B")) {

list.remove(s

}

}

} catch (ConcurrentModificationException e) {

System.***out***.println("Caught ConcurrentModificationException: " + e);

}

System.***out***.println("List after modificationattempt:"+list);

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

String s = iterator.next();

if (s.equals("C")) {

iterator.remove();

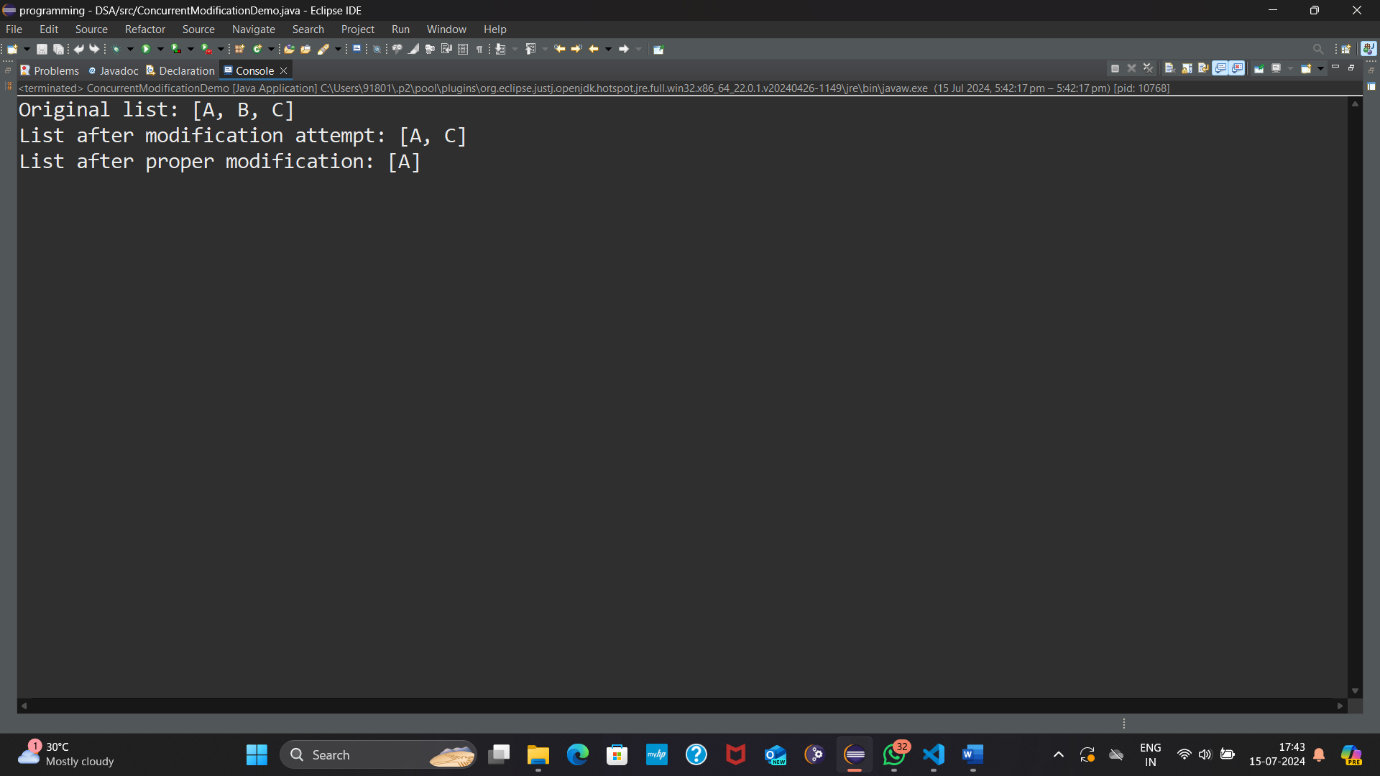
}

}

System.***out***.println("List after proper modification: " + list);

}

}

OUTPUT:- 

 **Time Complexity**: O(1)

 **Space Complexity**: O(n)



ANSWER:-.PROJECT:- <?xml version="1.0" encoding="UTF-8"?>

<projectDescription>

    <name>Maven Project</name>

    <comment></comment>

    <projects>

    </projects>

    <buildSpec>

        <buildCommand>

            <name>org.eclipse.jdt.core.javabuilder</name>

            <arguments>

            </arguments>

        </buildCommand>

    </buildSpec>

    <natures>

        <nature>org.eclipse.jdt.core.javanature</nature>

    </natures>

    <filteredResources>

        <filter>

            <id>1721044236741</id>

            <name></name>

            <type>30</type>

            <matcher>

                <id>org.eclipse.core.resources.regexFilterMatcher</id>

                <arguments>node\_modules|\.git|\_\_CREATED\_BY\_JAVA\_LANGUAGE\_SERVER\_\_</arguments>

            </matcher>

        </filter>

    </filteredResources>

</projectDescription>

DEMOAPPLICATION.JAVA:- package com.example;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.EnableAspectJAutoProxy;

@SpringBootApplication

@EnableAspectJAutoProxy

public class DemoApplication {

    public static void main(String[] args) {

        SpringApplication.run(DemoApplication.class, args);

    }}

LOGEXECUTIONTIMEASPECT.JACA:- import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.springframework.stereotype.Component;

import com.example.annotation.LogExecutionTime;

@Aspect

@Component

public class LogExecutionTimeAspect {

    @Around("@annotation(com.example.annotation.LogExecutionTime)")

    public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {

        long start = System.currentTimeMillis();

        Object proceed = joinPoint.proceed();

        long executionTime = System.currentTimeMillis() - start;

        System.out.println(joinPoint.getSignature() + " executed in " + executionTime + "ms");

        return proceed;

    }

}

MYSERVICE.JAVA:- package com.example.service;

import org.springframework.stereotype.Service;

import com.example.annotation.LogExecutionTime;

@Service

public class MyService {

    @LogExecutionTime

    public void serve() throws InterruptedException {

        // Simulate a method that takes some time to execute

        Thread.sleep(2000);

    }

}

LOGEXECUTIONTIME.JAVA:- package com.example.annotation;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface LogExecutionTime {

}

POM.XML:- <dependencies>

    <dependency>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter</artifactId>

    </dependency>

    <dependency>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-aop</artifactId>

    </dependency>

    <dependency>

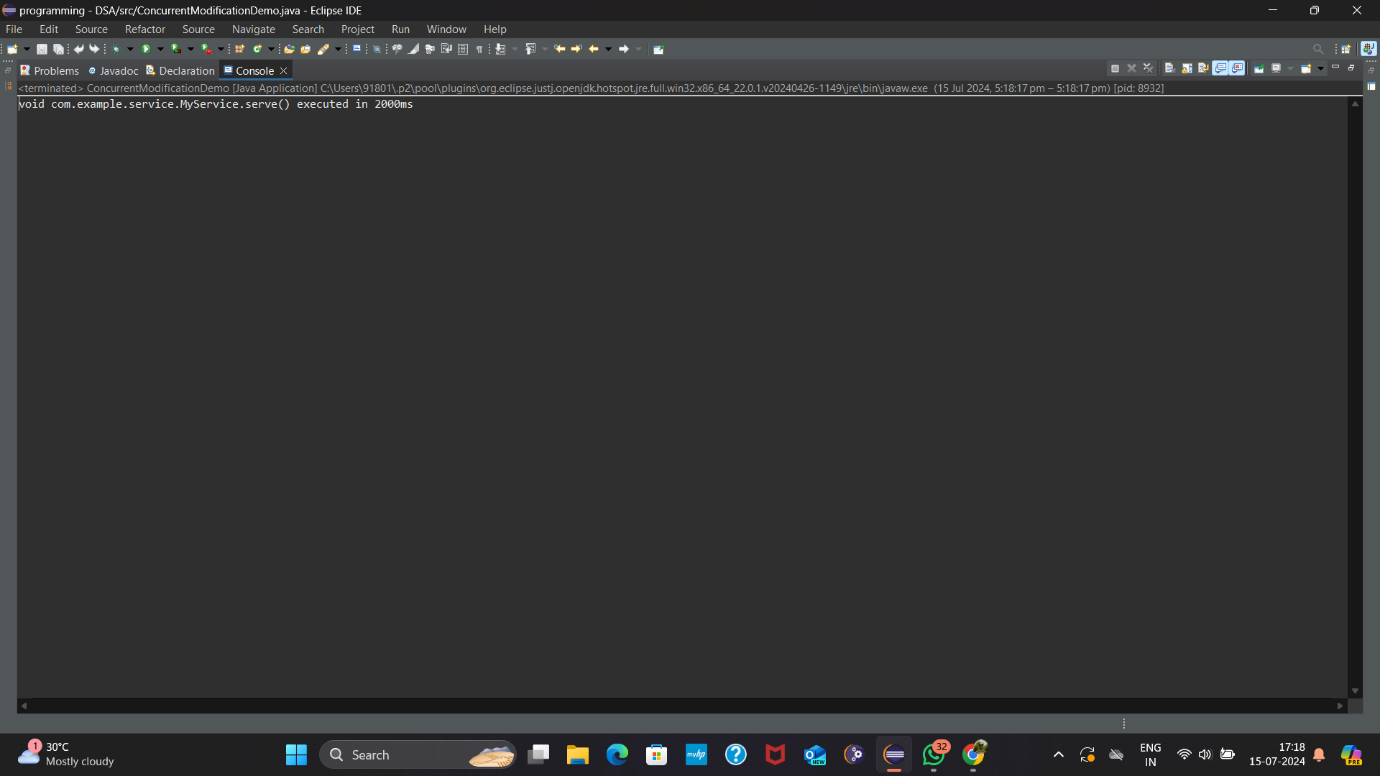
        <groupId>org.springframework.boot</groupId>

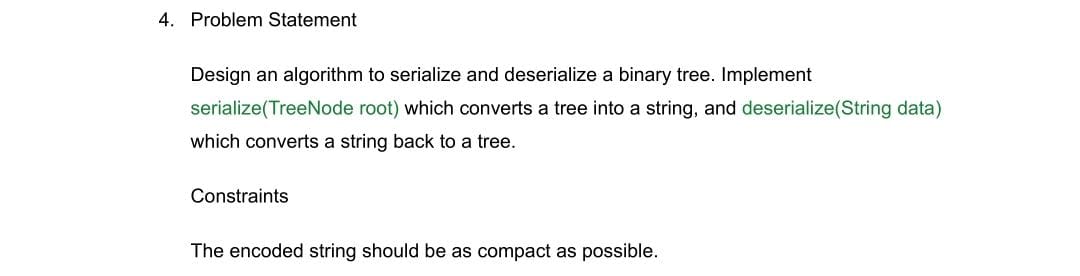
        <artifactId>spring-boot-starter-test</artifactId>

        <scope>test</scope>

    </dependency>

</dependencies>

OUTPUT:- 



ANSWER:- import java.util.\*;

public class BinaryTreeSerialization {

// TreeNode class definition

public static class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) { val = x; }

}

// Serialize the binary tree to a string

public String serialize(TreeNode root) {

StringBuilder sb = new StringBuilder();

serializeHelper(root, sb);

return sb.toString();

}

private void serializeHelper(TreeNode root, StringBuilder sb) {

if (root == null) {

sb.append("#,");

return;

}

sb.append(root.val).append(",");

serializeHelper(root.left, sb);

serializeHelper(root.right, sb);

}

// Deserialize the string back to a binary tree

public TreeNode deserialize(String data) {

if (data == null || data.isEmpty()) {

return null;

}

Queue<String> nodes = new LinkedList<>(Arrays.*asList*(data.split(",")));

return deserializeHelper(nodes);

}

private TreeNode deserializeHelper(Queue<String> nodes) {

String val = nodes.poll();

if (val.equals("#")) {

return null;

}

TreeNode root = new TreeNode(Integer.*parseInt*(val));

root.left = deserializeHelper(nodes);

root.right = deserializeHelper(nodes);

return root;

}

// Test the serialization and deserialization methods

public static void main(String[] args) {

BinaryTreeSerialization codec = new BinaryTreeSerialization();

// Create a sample tree

// 1

// / \

// 2 3

// / \

// 4 5

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.right = new TreeNode(3);

root.right.left = new TreeNode(4);

root.right.right = new TreeNode(5);

// Serialize the tree

String serializedTree = codec.serialize(root);

System.***out***.println("Serialized tree: " + serializedTree);

// Deserialize the string back to a tree

TreeNode deserializedRoot = codec.deserialize(serializedTree);

System.***out***.println("Deserialized tree: ");

*printTree*(deserializedRoot);

}

// Helper method to print the tree (in-order traversal)

private static void printTree(TreeNode root) {

if (root == null) {

return;

}

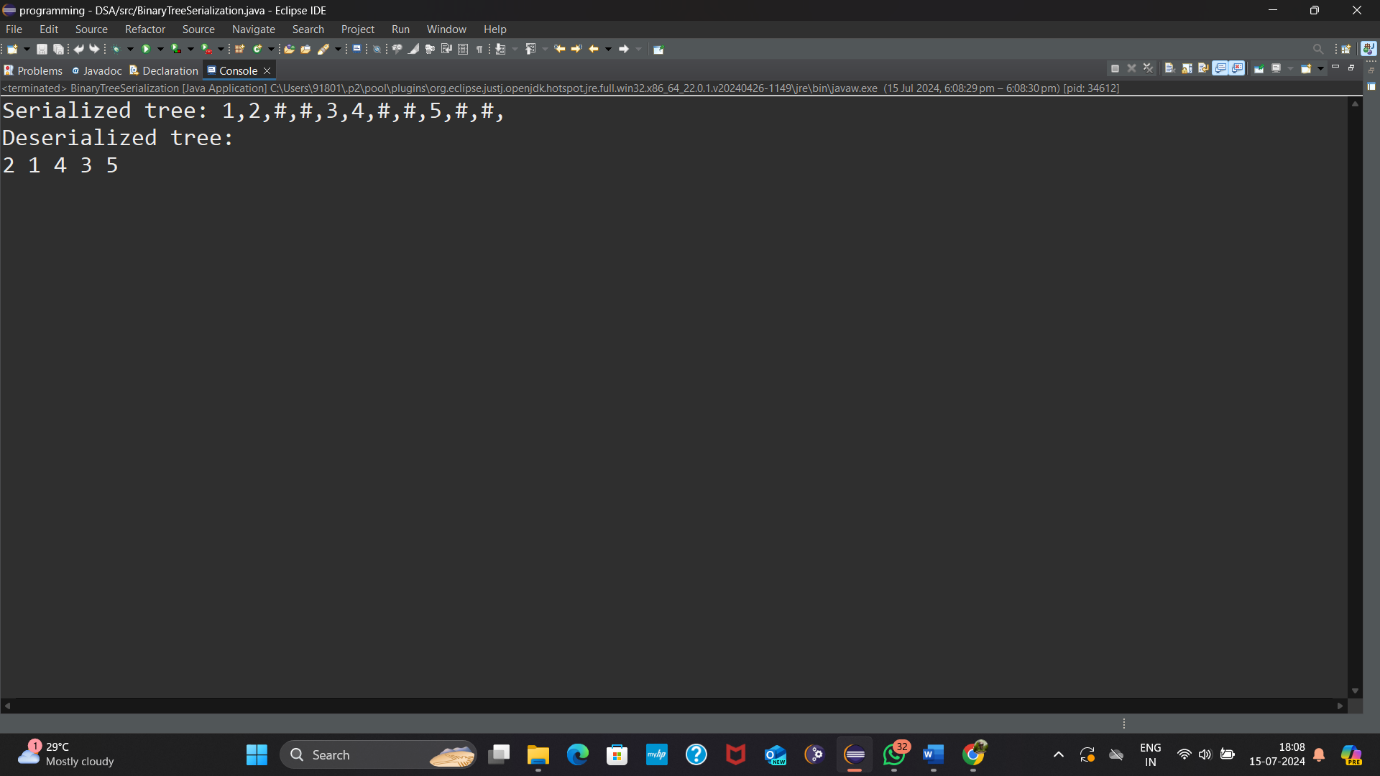
*printTree*(root.left);

System.***out***.print(root.val + " ");

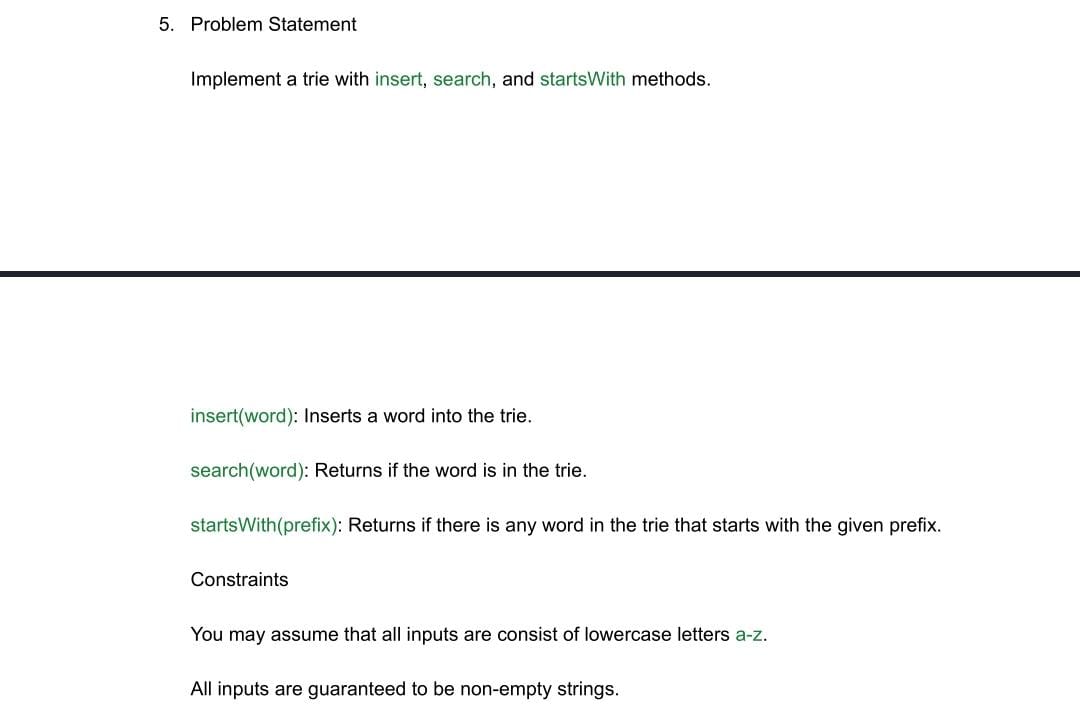
*printTree*(root.right);

}

}

OUTPUT:- 

* **Time Complexity**: O(n)
* **Space Complexity**: O(n)



ANSWER:- class TrieNode {

// Each node contains an array of children nodes and a flag to mark the end of a word

TrieNode[] children;

boolean isEndOfWord;

// Initialize the node

public TrieNode() {

children = new TrieNode[26]; // 26 letters in the alphabet

isEndOfWord = false;

}

}

class Trie {

private TrieNode root;

// Initialize the trie

public Trie() {

root = new TrieNode();

}

// Insert a word into the trie

public void insert(String word) {

TrieNode node = root;

for (char c : word.toCharArray()) {

int index = c - 'a';

if (node.children[index] == null) {

node.children[index] = new TrieNode();

}

node = node.children[index];

}

node.isEndOfWord = true;

}

// Search for a word in the trie

public boolean search(String word) {

TrieNode node = root;

for (char c : word.toCharArray()) {

int index = c - 'a';

if (node.children[index] == null) {

return false;

}

node = node.children[index];

}

return node.isEndOfWord;

}

// Check if there is any word in the trie that starts with the given prefix

public boolean startsWith(String prefix) {

TrieNode node = root;

for (char c : prefix.toCharArray()) {

int index = c - 'a';

if (node.children[index] == null) {

return false;

}

node = node.children[index];

}

return true;

}

// Test the trie

public static void main(String[] args) {

Trie trie = new Trie();

// Insert words into the trie

trie.insert("apple");

trie.insert("app");

trie.insert("banana");

// Search for words

System.***out***.println(trie.search("apple")); // Output: true

System.***out***.println(trie.search("app")); // Output: true

System.***out***.println(trie.search("appl")); // Output: false

System.***out***.println(trie.search("banana")); // Output: true

System.***out***.println(trie.search("bananaa")); // Output: false

// Check if there is any word that starts with the given prefix

System.***out***.println(trie.startsWith("app")); // Output: true

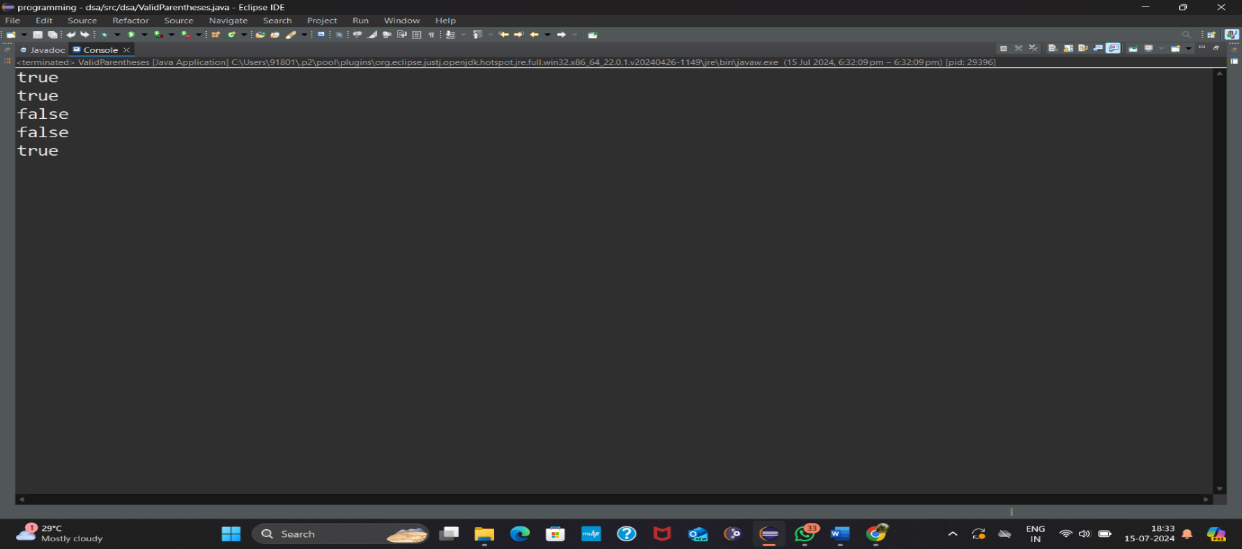
System.***out***.println(trie.startsWith("ban")); // Output: true

System.***out***.println(trie.startsWith("bat")); // Output: false

}

}

OUTPUT:-

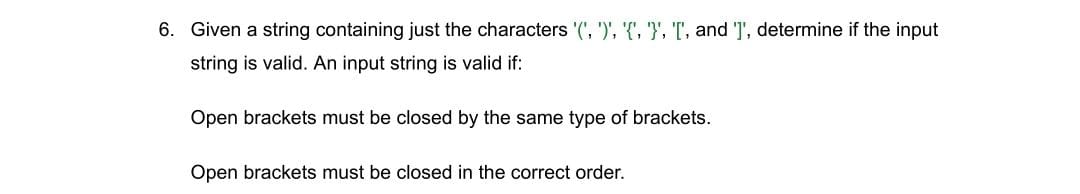


### Time Complexity:

* **insert(word)**: O(m), where m is the length of the word. Each character insertion involves constant time operations (array access and node creation).
* **search(word)**: O(m), where m is the length of the word. Each character search involves constant time operations.
* **startsWith(prefix)**: O(m), where m is the length of the prefix. Each character lookup for the prefix involves constant time operations.

### Space Complexity:

* The space complexity primarily depends on the number of nodes in the Trie and the size of the alphabet (constant 26 for lowercase letters).
* It is O(N \* L), where N is the total number of nodes and L is the average length of the words stored in the Trie. Each node can potentially have up to 26 children



ANSWER:- package dsa;

import java.util.Stack;

public class ValidParentheses {

// Function to check if the input string is valid

public boolean isValid(String s) {

// Stack to keep track of opening brackets

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

// Iterate through each character in the string

for (char c : s.toCharArray()) {

switch (c) {

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

case '(':

case '{':

case '[':

stack.push(c);

break;

// For closing brackets, check the top of the stack

case ')':

if (stack.isEmpty() || stack.pop() != '(') {

return false;

}

break;

case '}':

if (stack.isEmpty() || stack.pop() != '{') {

return false;

}

break;

case ']':

if (stack.isEmpty() || stack.pop() != '[') {

return false;

}

break;

default:

// If the character is not a bracket, return false

return false;

}

}

// If the stack is empty, all opening brackets have been matched

return stack.isEmpty();

}

// Main method to test the function

public static void main(String[] args) {

ValidParentheses validator = new ValidParentheses();

// Test cases

System.***out***.println(validator.isValid("()")); // Output: true

System.***out***.println(validator.isValid("()[]{}")); // Output: true

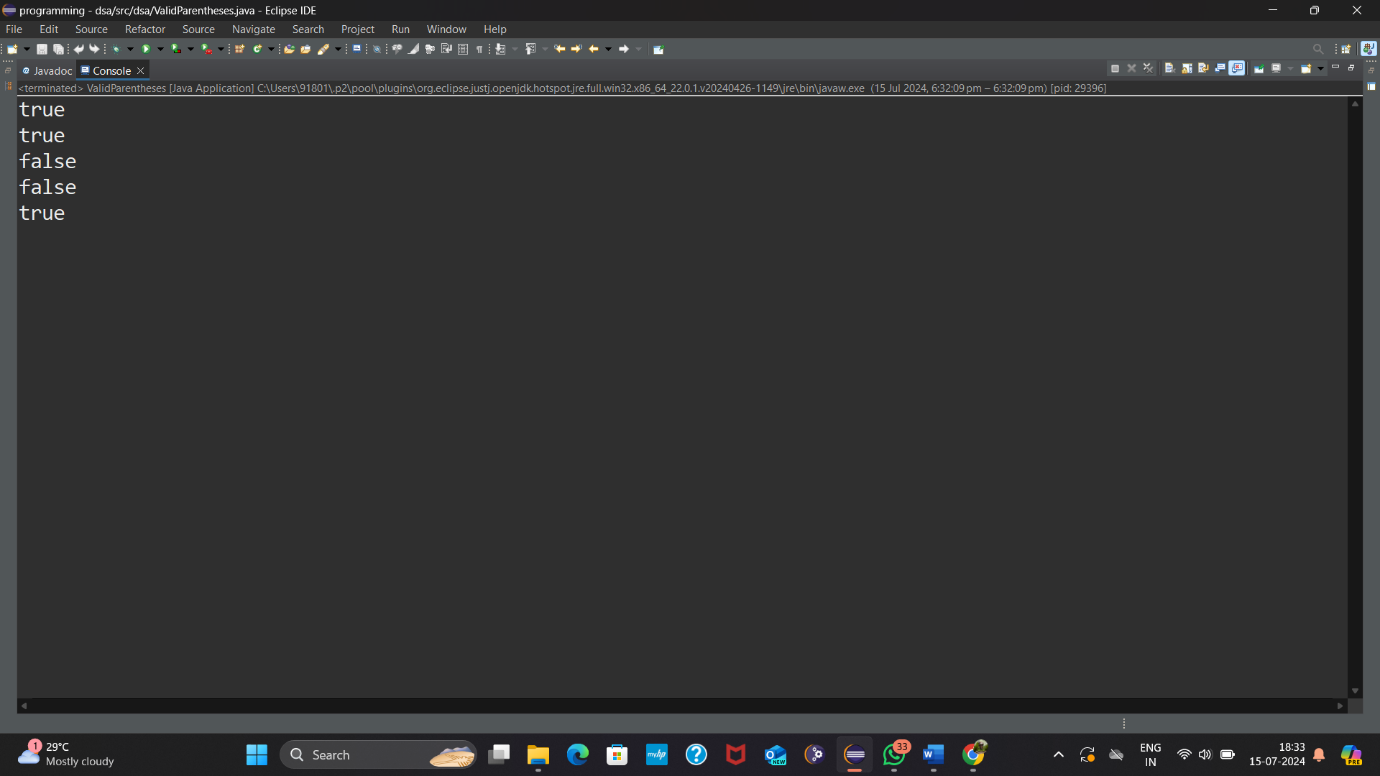
System.***out***.println(validator.isValid("(]")); // Output: false

System.***out***.println(validator.isValid("([)]")); // Output: false

System.***out***.println(validator.isValid("{[]}")); // Output: true

}

}

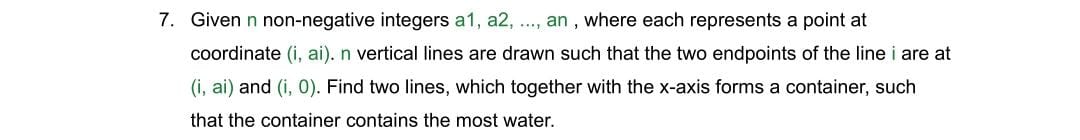
OUTPUT:- 

 **Time Complexity**:

* O(n): Traverse the string of length n once, performing constant-time operations (push, pop, peek on stack).

 **Space Complexity**:

* O(n): The stack can grow up to size n in the worst case (all opening brackets).



ANSWER:- package ff;

public class cointinerWithMostWater {

public int maxArea(int[] height) {

int maxArea = 0;

int left = 0;

int right = height.length - 1;

while (left < right) {

// Calculate the area between the two lines

int minHeight = Math.*min*(height[left], height[right]);

int currentArea = minHeight \* (right - left);

// Update maxArea if the current area is larger

maxArea = Math.*max*(maxArea, currentArea);

// Move the pointers based on the smaller height to possibly find a larger area

if (height[left] < height[right]) {

left++;

} else {

right--;

}

}

return maxArea;

}

public static void main(String[] args) {

cointinerWithMostWater solution = new cointinerWithMostWater();

// Example usage

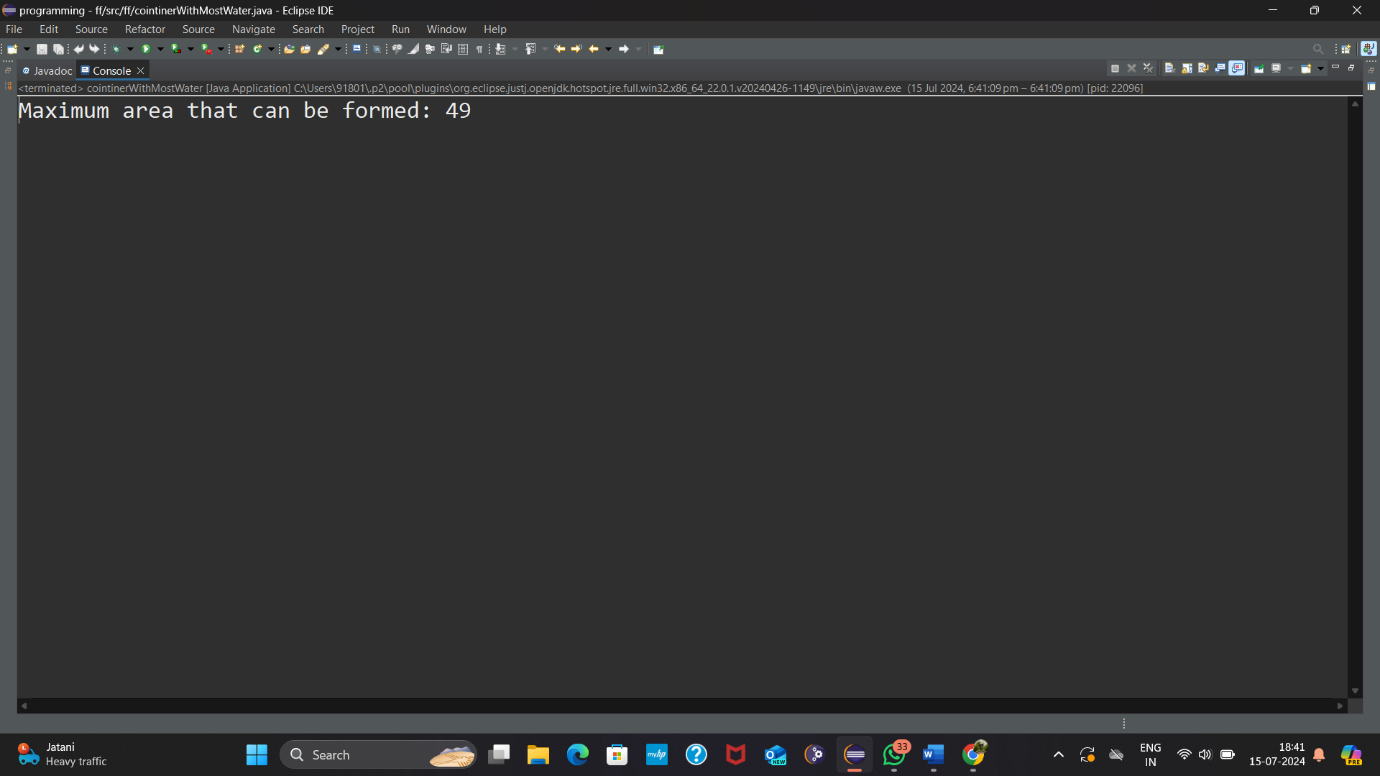
int[] heights = {1, 8, 6, 2, 5, 4, 8, 3, 7};

int maxArea = solution.maxArea(heights);

System.***out***.println("Maximum area that can be formed: " + maxArea); // Output: 49

}

}

OUTPUT:- 

 **Time Complexity**:

* O(n): Each pointer moves at most n-1 steps, and each step involves calculating the area and potentially updating the maximum area.

 **Space Complexity**:

* O(1): Only a constant amount of extra space is used, primarily for variables and indices.



ANSWER:- package ff;

import java.util.Random;

public class KthLargestElement {

public int findKthLargest(int[] nums, int k) {

if (nums == null || nums.length == 0 || k < 1 || k > nums.length) {

throw new IllegalArgumentException("Invalid input");

}

// Convert kth largest to kth smallest problem

int kthSmallest = nums.length - k;

return quickSelect(nums, 0, nums.length - 1, kthSmallest);

}

private int quickSelect(int[] nums, int left, int right, int k) {

if (left == right) {

return nums[left];

}

// Choose a random pivot index to avoid worst-case time complexity

Random random = new Random();

int pivotIndex = left + random.nextInt(right - left + 1);

pivotIndex = partition(nums, left, right, pivotIndex);

if (k == pivotIndex) {

return nums[k];

} else if (k < pivotIndex) {

return quickSelect(nums, left, pivotIndex - 1, k);

} else {

return quickSelect(nums, pivotIndex + 1, right, k);

}

}

private int partition(int[] nums, int left, int right, int pivotIndex) {

int pivotValue = nums[pivotIndex];

// Move pivot to end

swap(nums, pivotIndex, right);

int storeIndex = left;

for (int i = left; i < right; i++) {

if (nums[i] < pivotValue) {

swap(nums, storeIndex, i);

storeIndex++;

}

}

// Move pivot to its final place

swap(nums, storeIndex, right);

return storeIndex;

}

private void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

public static void main(String[] args) {

KthLargestElement solution = new KthLargestElement();

// Example usage

int[] nums = {3, 2, 1, 5, 6, 4};

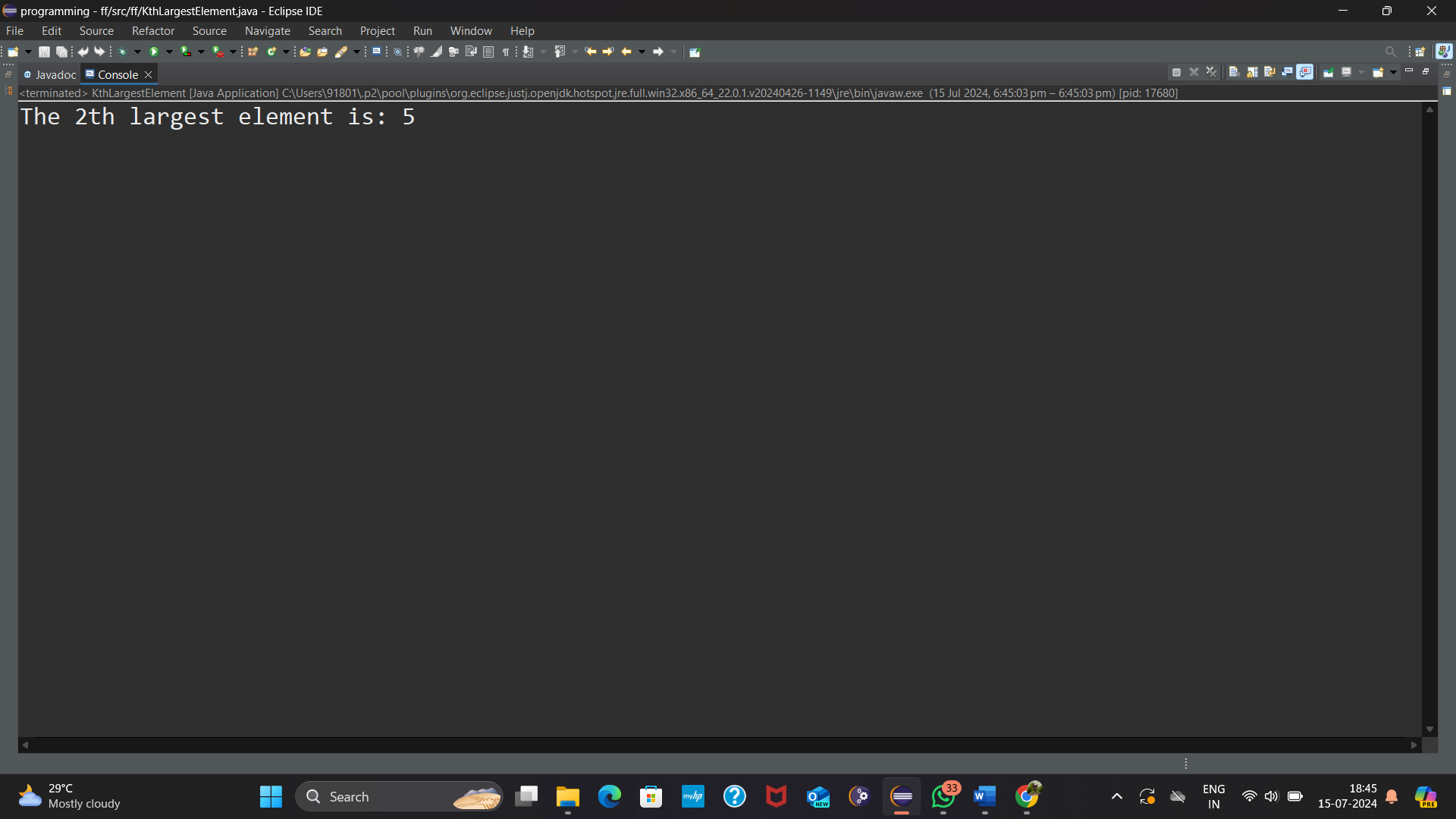
int k = 2;

int result = solution.findKthLargest(nums, k);

System.***out***.println("The " + k + "th largest element is: " + result); // Output: 5

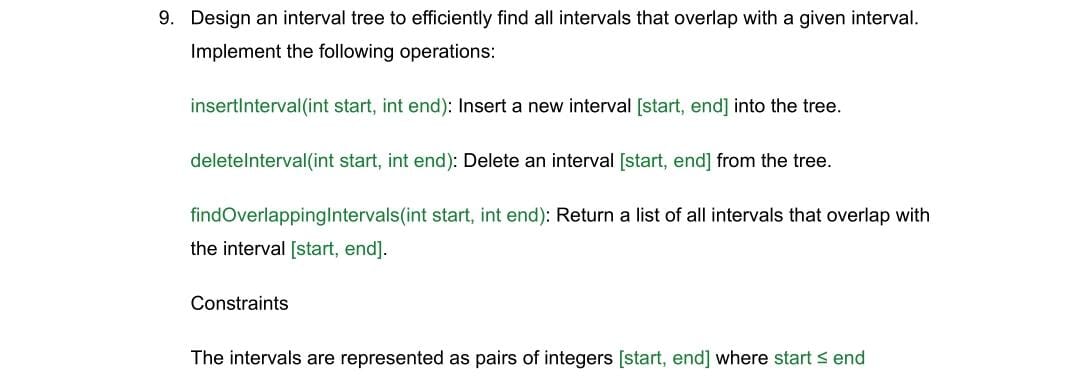
}

}

OUTPUT:- 

 **Time Complexity**: O(n)O(n)O(n) on average and O(n2)O(n^2)O(n2) in the worst case

 **Space Complexity:** O(logn)O(log n)O(logn)



ANSWER:- package ff;

import java.util.ArrayList;

import java.util.List;

public class IntervalTree {

// Node class representing an interval

private static class Node {

int start;

int end;

Node left;

Node right;

Node(int start, int end) {

this.start = start;

this.end = end;

this.left = null;

this.right = null;

}

}

private Node root;

// Constructor to initialize the tree

public IntervalTree() {

this.root = null;

}

// Insert a new interval [start, end] into the tree

public void insertInterval(int start, int end) {

root = insert(root, start, end);

}

// Helper method to recursively insert an interval into the tree

private Node insert(Node node, int start, int end) {

if (node == null) {

return new Node(start, end);

}

// Insert into left subtree

if (start < node.start) {

node.left = insert(node.left, start, end);

} else { // Insert into right subtree

node.right = insert(node.right, start, end);

}

return node;

}

// Delete an interval [start, end] from the tree

public void deleteInterval(int start, int end) {

root = delete(root, start, end);

}

// Helper method to recursively delete an interval from the tree

private Node delete(Node node, int start, int end) {

if (node == null) {

return null;

}

// If current interval overlaps with [start, end], delete it

if (overlap(node.start, node.end, start, end)) {

// Node to be deleted is a leaf node or has only one child

if (node.left == null) {

return node.right;

} else if (node.right == null) {

return node.left;

}

// Node to be deleted has two children

Node minNode = findMin(node.right);

node.start = minNode.start;

node.end = minNode.end;

node.right = delete(node.right, minNode.start, minNode.end);

} else if (start < node.start) {

// Search in the left subtree

node.left = delete(node.left, start, end);

} else {

// Search in the right subtree

node.right = delete(node.right, start, end);

}

return node;

}

// Find the minimum interval in the subtree

private Node findMin(Node node) {

while (node.left != null) {

node = node.left;

}

return node;

}

// Find all intervals that overlap with [start, end]

public List<int[]> findOverlappingIntervals(int start, int end) {

List<int[]> result = new ArrayList<>();

search(root, start, end, result);

return result;

}

// Helper method to recursively search for overlapping intervals

private void search(Node node, int start, int end, List<int[]> result) {

if (node == null) {

return;

}

// Check if the node interval overlaps with [start, end]

if (overlap(node.start, node.end, start, end)) {

result.add(new int[]{node.start, node.end});

}

// Search left subtree if necessary

if (node.left != null && node.left.end >= start) {

search(node.left, start, end, result);

}

// Search right subtree if necessary

if (node.right != null && node.right.start <= end) {

search(node.right, start, end, result);

}

}

// Check if two intervals [s1, e1] and [s2, e2] overlap

private boolean overlap(int s1, int e1, int s2, int e2) {

return !(s1 > e2 || s2 > e1);

}

// Test the Interval Tree operations

public static void main(String[] args) {

IntervalTree tree = new IntervalTree();

// Insert intervals

tree.insertInterval(15, 20);

tree.insertInterval(10, 30);

tree.insertInterval(5, 15);

tree.insertInterval(17, 19);

tree.insertInterval(12, 15);

// Find overlapping intervals with [14, 16]

List<int[]> overlappingIntervals = tree.findOverlappingIntervals(14, 16);

// Print the results

System.***out***.println("Intervals overlapping with [14, 16]:");

for (int[] interval : overlappingIntervals) {

System.***out***.println("[" + interval[0] + ", " + interval[1] + "]");

}

// Delete an interval [15, 20]

tree.deleteInterval(15, 20);

// Find overlapping intervals with [14, 16] again

overlappingIntervals = tree.findOverlappingIntervals(14, 16);

// Print the results after deletion

System.***out***.println("\nIntervals overlapping with [14, 16] after deletion:");

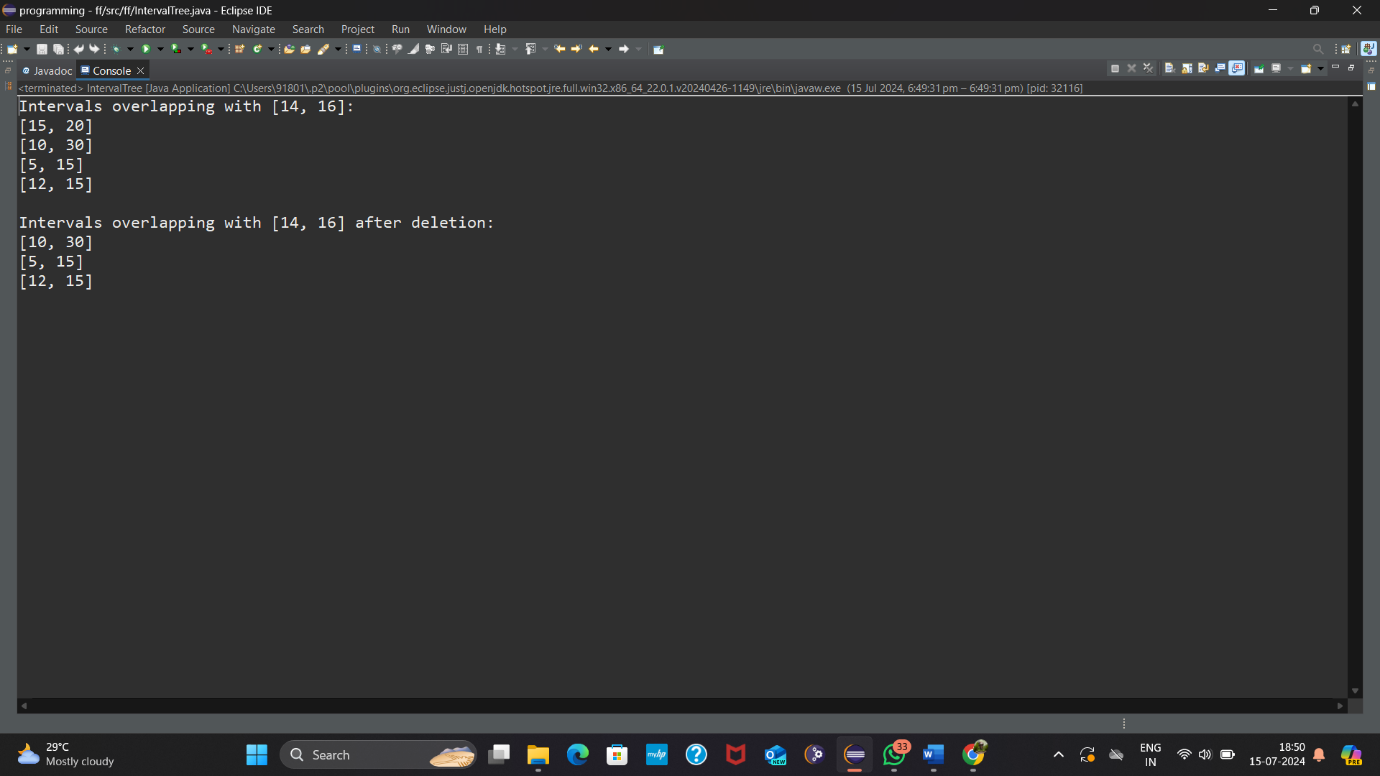
for (int[] interval : overlappingIntervals) {

System.***out***.println("[" + interval[0] + ", " + interval[1] + "]");

}

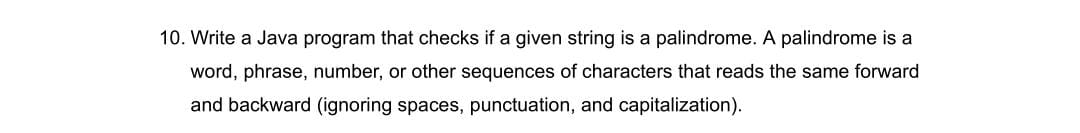
}

}

OUTPUT:- 

 **Time Complexity**: O(log n) & overlapping intervals is O(log n + k)

 **Space Complexity**: O(n)



ANSWER:- package ff;

public class PalindromeChecker {

public static boolean isPalindrome(String s) {

// Convert the string to lowercase and remove non-alphanumeric characters

String normalized = s.toLowerCase().replaceAll("[^a-zA-Z0-9]", "");

// Check if the normalized string is a palindrome

int left = 0;

int right = normalized.length() - 1;

while (left < right) {

if (normalized.charAt(left) != normalized.charAt(right)) {

return false;

}

left++;

right--;

}

return true;

}

public static void main(String[] args) {

// Test cases

String s1 = "A man, a plan, a canal, Panama";

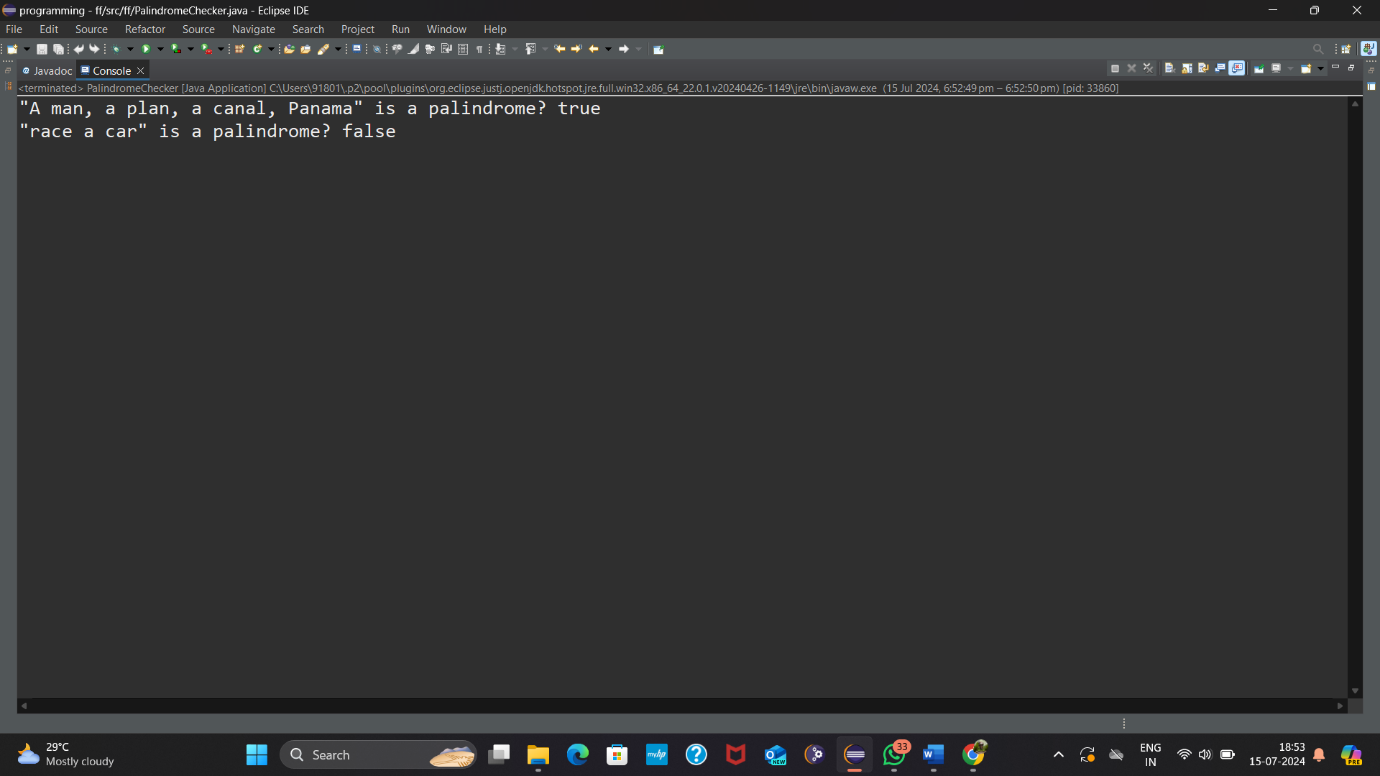
String s2 = "race a car";

System.***out***.println("\"" + s1 + "\" is a palindrome? " + *isPalindrome*(s1)); // Output: true

System.***out***.println("\"" + s2 + "\" is a palindrome? " + *isPalindrome*(s2)); // Output: false

}

}

OUTPUT:- 

 **Time Complexity**: The time complexity is O(n), where n is the length of the string after normalization, since we iterate through the string once.

 **Space Complexity**: The space complexity is O(1) excluding the normalized string storage, as we use a constant amount of extra space for pointers and loop variables.