1. Problem Statement

Design and implement a data structure for a Least Recently Used (LRU) cache. It should support the following operations: get and put.

get(key): Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

put(key, value): Set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least recently used item before inserting a new item.

Constraints

The number of get and put operations will be in the range [1, 10^5].

The capacity of the cache is between 1 and 10^5.

Code:

**package** Java1;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** LRUCache {

// Define the doubly-linked list node

**class** Node {

**int** key;

**int** value;

Node prev;

Node 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;

**private** **final** Node tail;

**public** LRUCache(**int** capacity) {

**this**.capacity = capacity;

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

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

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

head.next = tail;

tail.prev = head;

}

**public** **int** get(**int** key) {

**if** (cache.containsKey(key)) {

Node node = cache.get(key);

removeNode(node);

addToFront(node);

**return** node.value;

}

**return** -1;

}

**public** **void** put(**int** key, **int** value) {

**if** (cache.containsKey(key)) {

Node node = cache.get(key);

node.value = value;

removeNode(node);

addToFront(node);

} **else** {

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

Node lru = tail.prev;

removeNode(lru);

cache.remove(lru.key);

}

Node newNode = **new** Node(key, value);

cache.put(key, newNode);

addToFront(newNode);

}

}

**private** **void** removeNode(Node node) {

node.prev.next = node.next;

node.next.prev = node.prev;

}

**private** **void** addToFront(Node node) {

node.next = head.next;

node.next.prev = node;

head.next = node;

node.prev = head;

}

**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)); // returns 1

cache.put(3, 3); // evicts key 2

System.***out***.println(cache.get(2)); // returns -1 (not found)

cache.put(4, 4); // evicts key 1

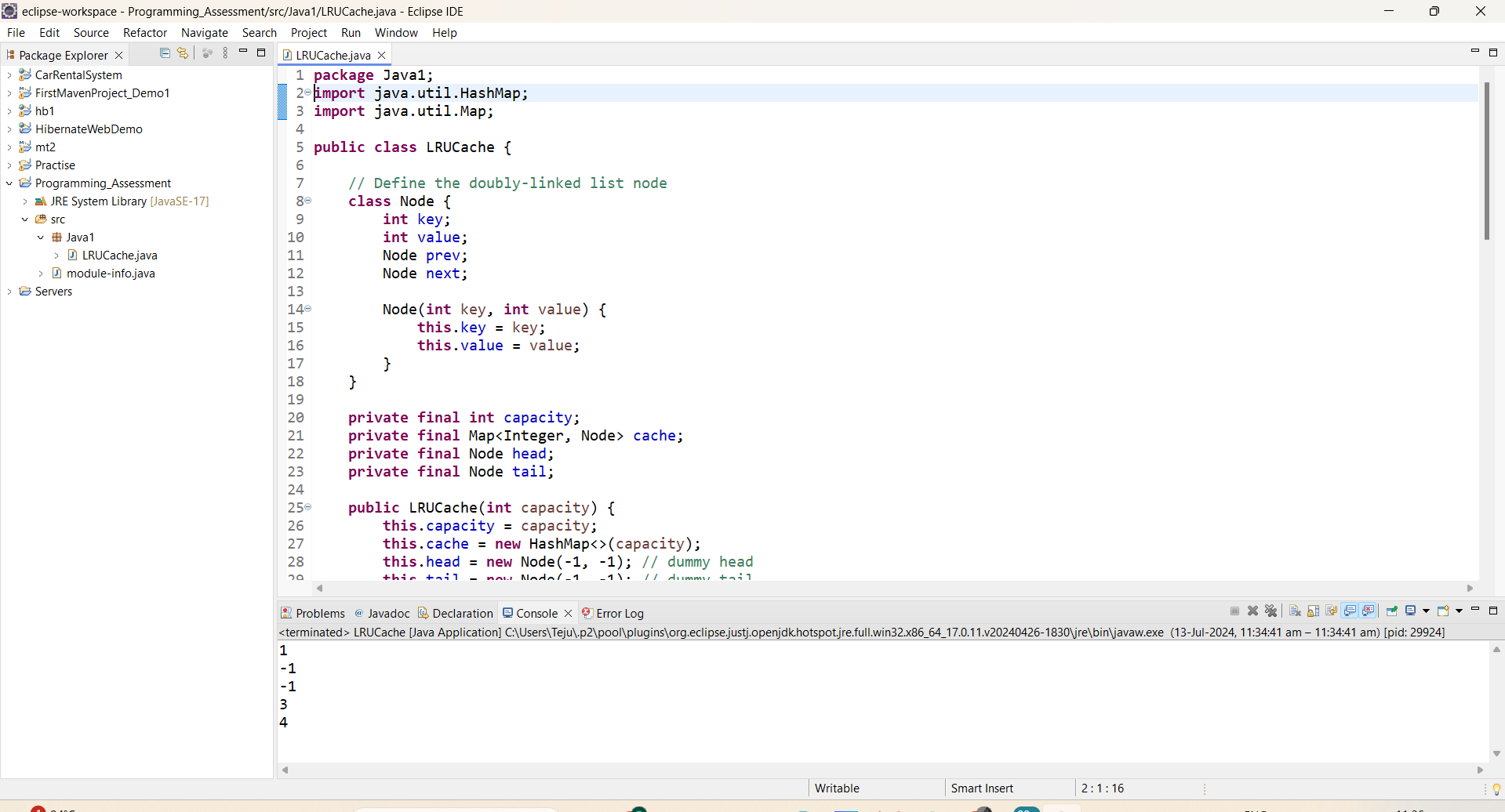
System.***out***.println(cache.get(1)); // returns -1 (not found)

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

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

}

}



1. Write a Java program that demonstrates the ConcurrentModificationException. Explain why the exception is thrown and how to handle it properly.

Code:

package Java1;

import java.util.ArrayList;

import java.util.ConcurrentModificationException;

import java.util.Iterator;

import java.util.List;

public class ConcurrentModificationDemo {

public static void main(String[] args) {

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

list.add("one");

list.add("two");

list.add("three");

System.out.println("Demonstrating ConcurrentModificationException:");

try {

for (String item : list) {

if ("two".equals(item)) {

list.remove(item); // This line will cause ConcurrentModificationException

}

}

} catch (ConcurrentModificationException e) {

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

}

// Proper way to modify a collection during iteration

System.out.println("Handling modification properly:");

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

while (iterator.hasNext()) {

String item = iterator.next();

if ("two".equals(item)) {

iterator.remove(); // Proper way to remove elements during iteration

}

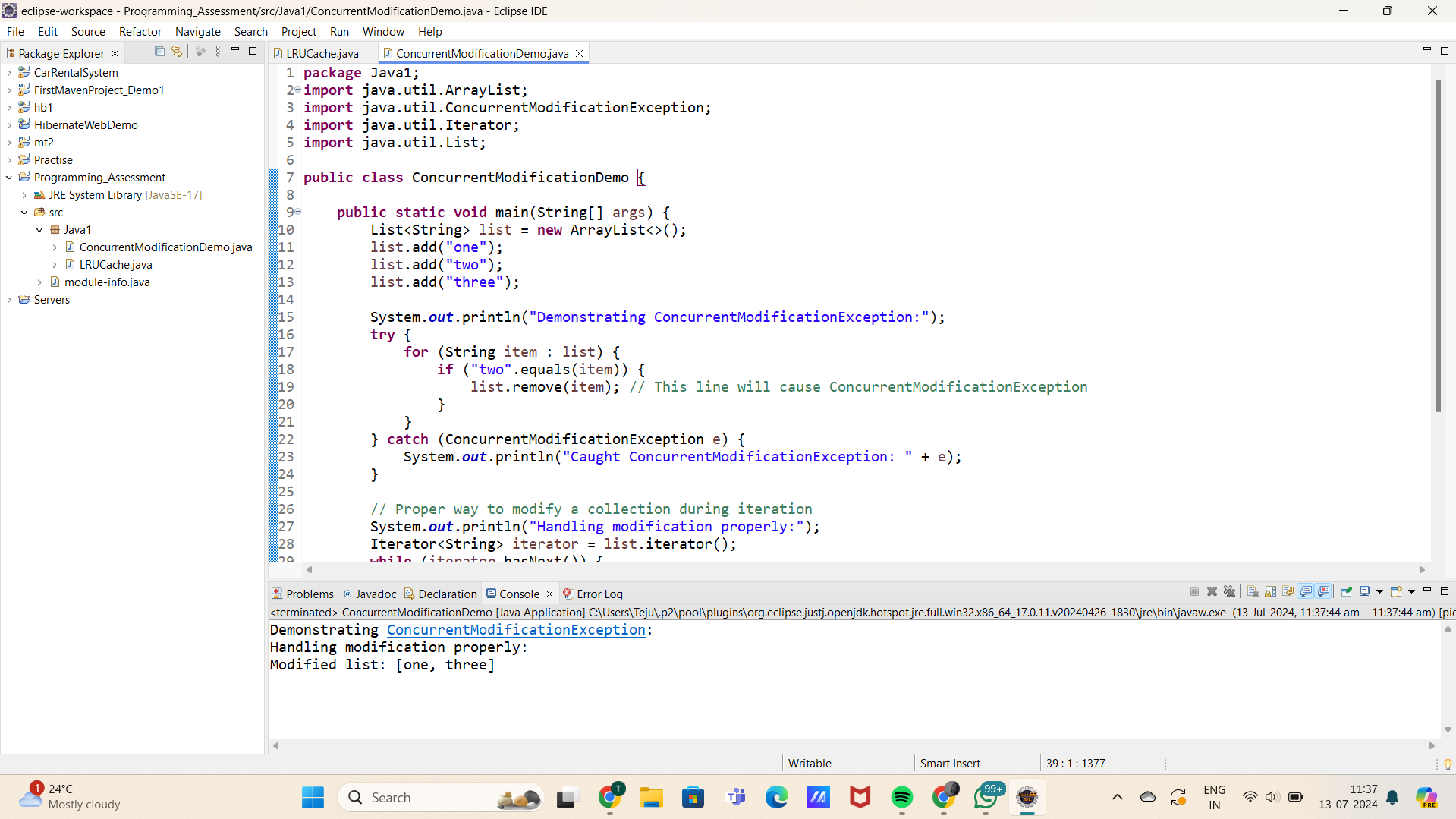
}

// Printing the list to show the result of proper modification

System.out.println("Modified list: " + list);

}

}



5. Problem Statement

Implement a trie with insert, search, and startsWith methods.

insert(word): Inserts a word into the trie.

search(word): Returns if the word is in the trie.

startsWith(prefix): Returns if there is any word in the trie that starts with the given prefix.

Constraints

You may assume that all inputs are consist of lowercase letters a-z.

All inputs are guaranteed to be non-empty strings.

Code:

**package** Java1;

**class** TrieNode {

TrieNode[] children;

**boolean** isEndOfWord;

**public** TrieNode() {

children = **new** TrieNode[26];

isEndOfWord = **false**;

}

}

**public** **class** Trie {

**private** TrieNode root;

**public** Trie() {

root = **new** TrieNode();

}

// Inserts a word into the trie.

**public** **void** insert(String word) {

TrieNode current = root;

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

**int** index = c - 'a';

**if** (current.children[index] == **null**) {

current.children[index] = **new** TrieNode();

}

current = current.children[index];

}

current.isEndOfWord = **true**;

}

// Returns if the word is in the trie.

**public** **boolean** search(String word) {

TrieNode node = searchPrefix(word);

**return** node != **null** && node.isEndOfWord;

}

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

**public** **boolean** startsWith(String prefix) {

TrieNode node = searchPrefix(prefix);

**return** node != **null**;

}

// Helper method to search a prefix or a whole word in the trie.

**private** TrieNode searchPrefix(String word) {

TrieNode current = root;

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

**int** index = c - 'a';

**if** (current.children[index] == **null**) {

**return** **null**;

}

current = current.children[index];

}

**return** current;

}

**public** **static** **void** main(String[] args) {

Trie trie = **new** Trie();

trie.insert("apple");

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

System.***out***.println("Search 'app': " + trie.search("app")); // Output: false

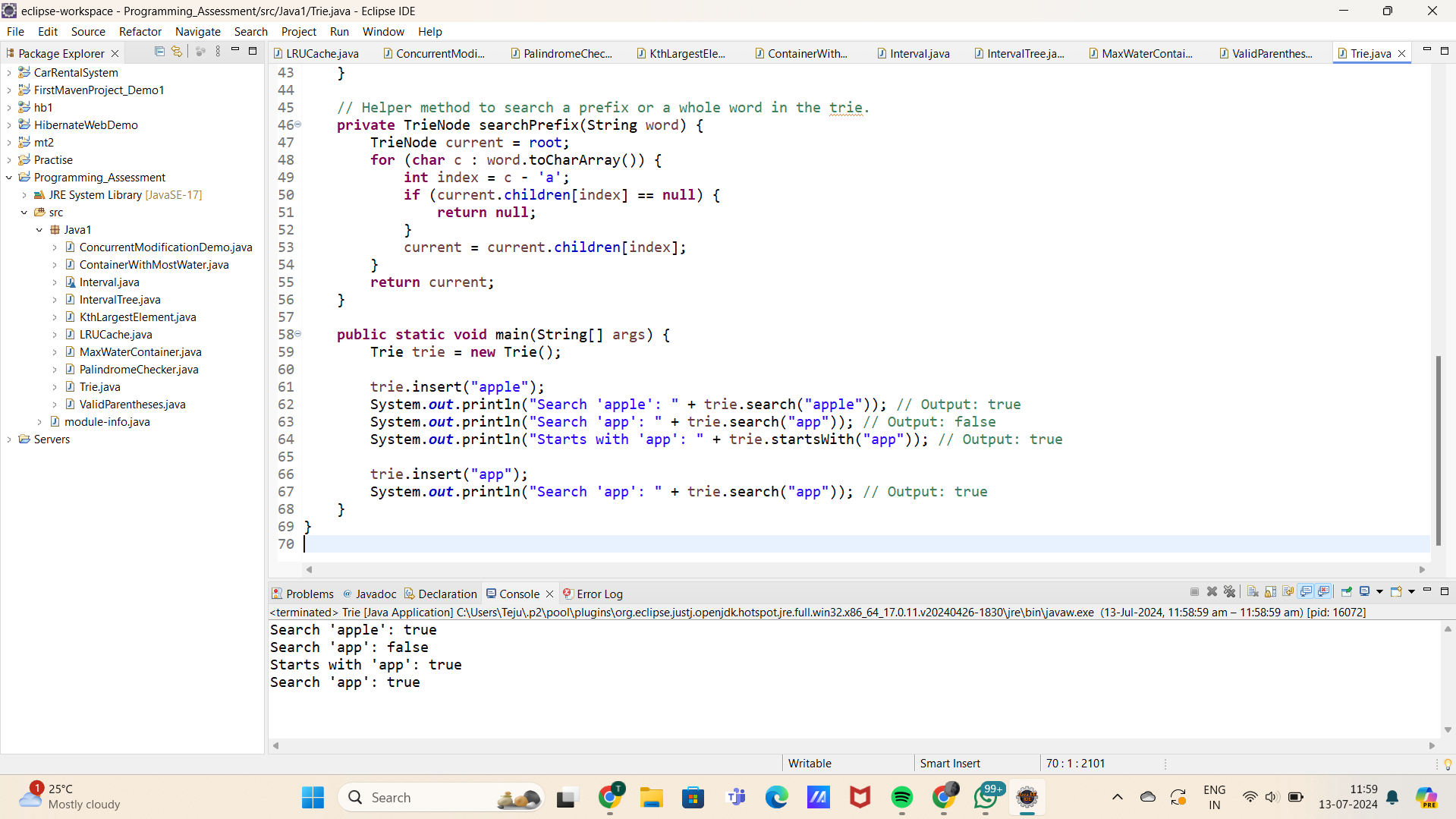
System.***out***.println("Starts with 'app': " + trie.startsWith("app")); // Output: true

trie.insert("app");

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

}

}



6. Given a string containing just the characters '(', ')', '{', '}', '[', and ']', determine if the input string is valid. An input string is valid if:

Open brackets must be closed by the same type of brackets.

Open brackets must be closed in the correct order.

Code:

**package** Java1;

**import** java.util.Stack;

**public** **class** ValidParentheses {

**public** **boolean** isValid(String s) {

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

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

**if** (c == '(' || c == '{' || c == '[') {

stack.push(c);

} **else** **if** (c == ')' && !stack.isEmpty() && stack.peek() == '(') {

stack.pop();

} **else** **if** (c == '}' && !stack.isEmpty() && stack.peek() == '{') {

stack.pop();

} **else** **if** (c == ']' && !stack.isEmpty() && stack.peek() == '[') {

stack.pop();

} **else** {

**return** **false**; // Unmatched closing bracket or invalid character

}

}

**return** stack.isEmpty(); // Stack should be empty if all opening brackets are matched

}

**public** **static** **void** main(String[] args) {

ValidParentheses solution = **new** ValidParentheses();

String input1 = "()"; // Valid

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

String input3 = "(]"; // Invalid

String input4 = "([)]"; // Invalid

String input5 = "{[]}"; // Valid

System.***out***.println(input1 + " is valid: " + solution.isValid(input1));

System.***out***.println(input2 + " is valid: " + solution.isValid(input2));

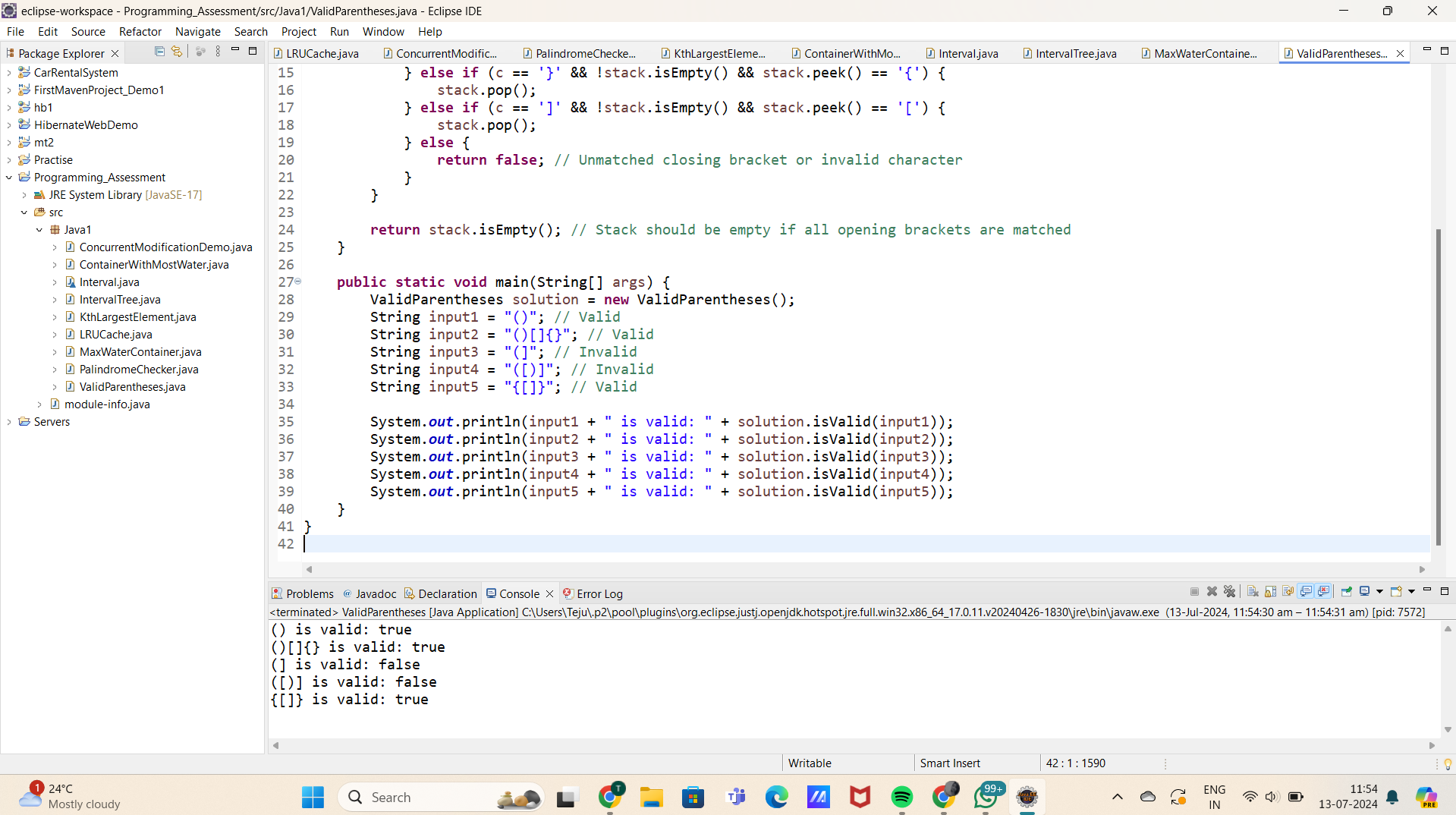
System.***out***.println(input3 + " is valid: " + solution.isValid(input3));

System.***out***.println(input4 + " is valid: " + solution.isValid(input4));

System.***out***.println(input5 + " is valid: " + solution.isValid(input5));

}

}



7. Given n non-negative integers a1, a2, ..., an , where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of the line i are at (i, ai) and (i, 0). Find two lines, which together with the x-axis forms a container, such that the container contains the most water.

CODE:

**package** Java1;

**public** **class** MaxWaterContainer {

**public** **int** maxArea(**int**[] height) {

**int** maxArea = 0;

**int** left = 0;

**int** right = height.length - 1;

**while** (left < right) {

// Calculate the area formed between the lines at the left and right pointers

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

**int** currentWidth = right - left;

**int** currentArea = currentWidth \* currentHeight;

// Update maxArea if the current area is larger

**if** (currentArea > maxArea) {

maxArea = currentArea;

}

// Move the pointer that points to the shorter line inward

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

left++;

} **else** {

right--;

}

}

**return** maxArea;

}

**public** **static** **void** main(String[] args) {

MaxWaterContainer solution = **new** MaxWaterContainer();

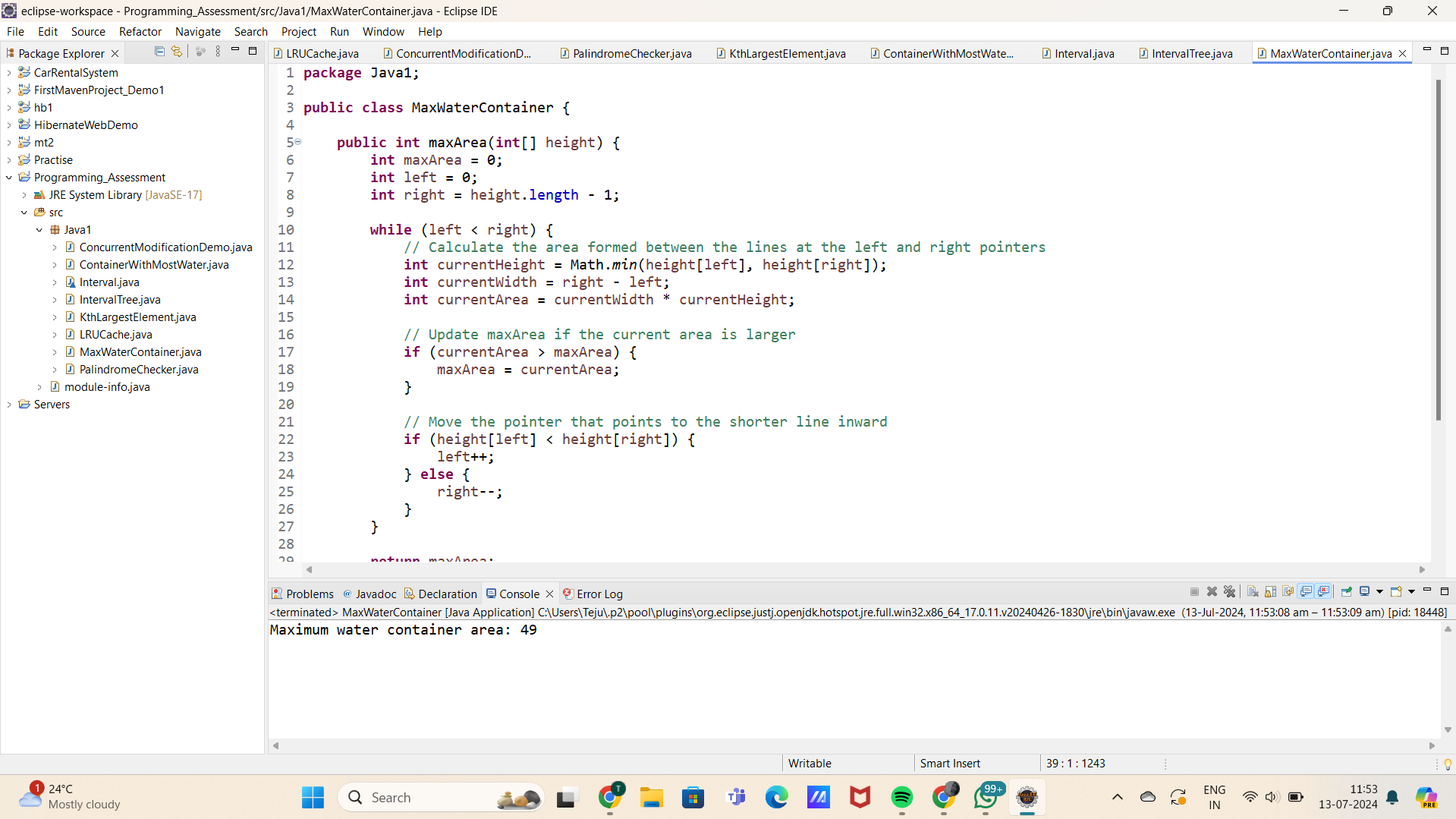
**int**[] height = {1, 8, 6, 2, 5, 4, 8, 3, 7};

**int** maxArea = solution.maxArea(height);

System.***out***.println("Maximum water container area: " + maxArea); // Output: Maximum water container area: 49

}

}



8. Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

CODE:

**package** Java1;

**import** java.util.Random;

**public** **class** KthLargestElement {

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

**return** quickselect(nums, 0, nums.length - 1, nums.length - k);

}

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

**if** (left == right) {

**return** nums[left];

}

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];

swap(nums, pivotIndex, right); // Move pivot to end

**int** storeIndex = left;

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

**if** (nums[i] < pivotValue) {

swap(nums, storeIndex, i);

storeIndex++;

}

}

swap(nums, right, storeIndex); // Move pivot to its final place

**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();

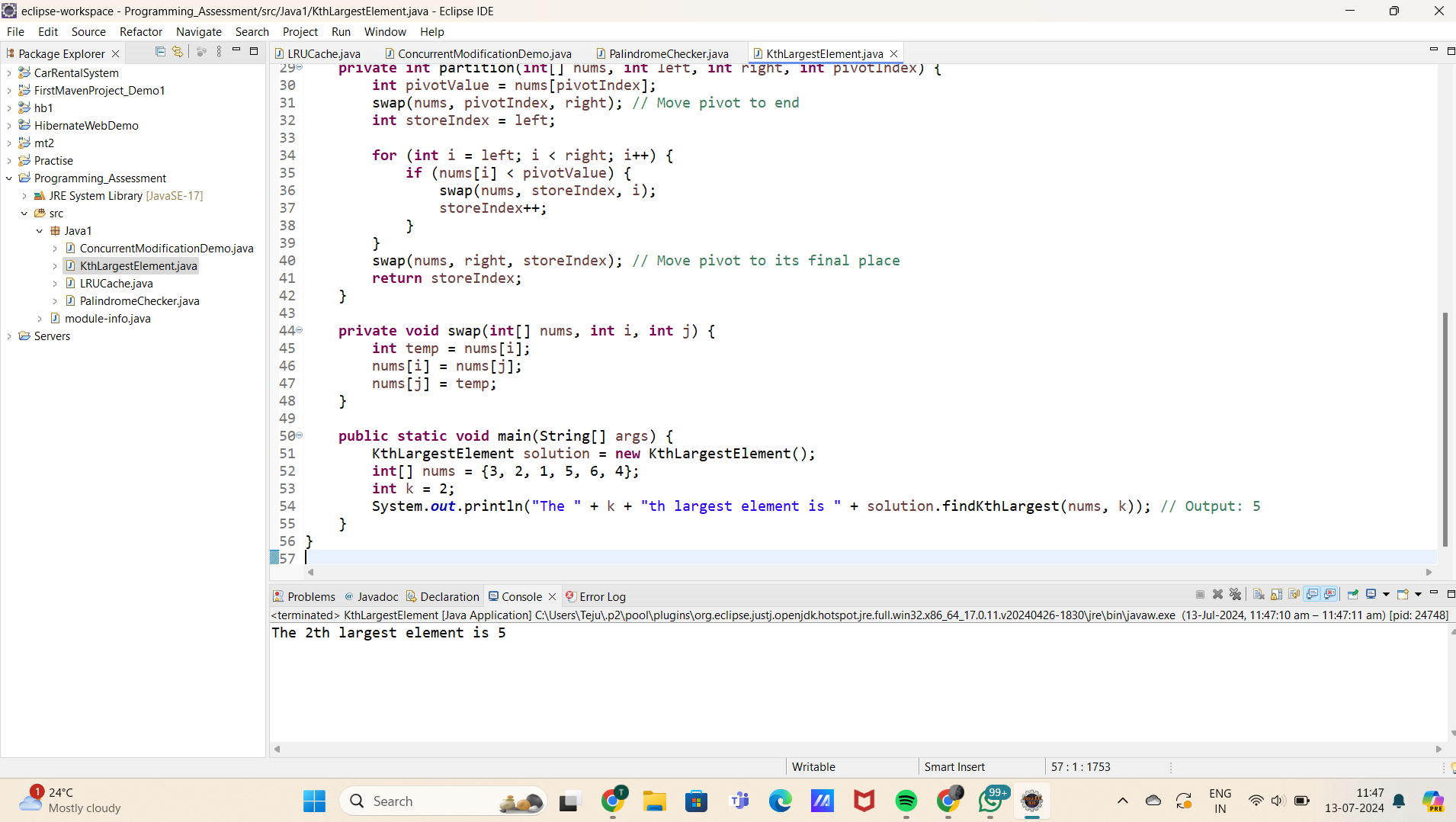
**int**[] nums = {3, 2, 1, 5, 6, 4};

**int** k = 2;

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

}

}



9.Design an interval tree to efficiently find all intervals that overlap with a given interval. Implement the following operations:

insertInterval(int start, int end): Insert a new interval [start, end] into the tree.

deleteInterval(int start, int end): Delete an interval [start, end] from the tree.

findOverlappingIntervals(int start, int end): Return a list of all intervals that overlap with the interval [start, end].

Constraints

The intervals are represented as pairs of integers [start, end] where start ≤ end

CODE:

Interval.java

**package** Java1;

**class** Interval {

**int** start;

**int** end;

**public** Interval(**int** start, **int** end) {

**this**.start = start;

**this**.end = end;

}

}

**class** IntervalTreeNode {

Interval interval;

**int** maxEnd;

IntervalTreeNode left;

IntervalTreeNode right;

**public** IntervalTreeNode(Interval interval) {

**this**.interval = interval;

**this**.maxEnd = interval.end;

**this**.left = **null**;

**this**.right = **null**;

}

}

Intervaltree.java

package Java1;

import java.util.ArrayList;

import java.util.List;

public class IntervalTree {

private IntervalTreeNode root;

public IntervalTree() {

this.root = null;

}

public void insertInterval(int start, int end) {

Interval newInterval = new Interval(start, end);

root = insertInterval(root, newInterval);

}

private IntervalTreeNode insertInterval(IntervalTreeNode node, Interval newInterval) {

if (node == null) {

return new IntervalTreeNode(newInterval);

}

// Check if newInterval overlaps with node's interval

if (newInterval.start <= node.interval.end && newInterval.end >= node.interval.start) {

// Update maxEnd if necessary

if (newInterval.end > node.maxEnd) {

node.maxEnd = newInterval.end;

}

// Recursively insert into left or right subtree

if (newInterval.start < node.interval.start) {

node.left = insertInterval(node.left, newInterval);

} else {

node.right = insertInterval(node.right, newInterval);

}

}

return node;

}

public void deleteInterval(int start, int end) {

Interval intervalToDelete = new Interval(start, end);

root = deleteInterval(root, intervalToDelete);

}

private IntervalTreeNode deleteInterval(IntervalTreeNode node, Interval intervalToDelete) {

if (node == null) {

return null;

}

// Recursively search for the interval to delete

if (intervalToDelete.end < node.interval.start) {

node.left = deleteInterval(node.left, intervalToDelete);

} else if (intervalToDelete.start > node.interval.end) {

node.right = deleteInterval(node.right, intervalToDelete);

} else {

// Found the interval to delete

if (node.left == null) {

return node.right;

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

return node.left;

} else {

// Node has two children, find successor and replace

IntervalTreeNode successor = findMin(node.right);

node.interval = successor.interval;

node.maxEnd = successor.maxEnd;

node.right = deleteInterval(node.right, successor.interval);

}

}

// Update maxEnd after deletion

if (node != null) {

node.maxEnd = Math.max(node.interval.end, getMaxEnd(node.right));

}

return node;

}

private IntervalTreeNode findMin(IntervalTreeNode node) {

while (node.left != null) {

node = node.left;

}

return node;

}

private int getMaxEnd(IntervalTreeNode node) {

return node == null ? Integer.MIN\_VALUE : node.maxEnd;

}

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

List<Interval> result = new ArrayList<>();

Interval queryInterval = new Interval(start, end);

findOverlappingIntervals(root, queryInterval, result);

return result;

}

private void findOverlappingIntervals(IntervalTreeNode node, Interval queryInterval, List<Interval> result) {

if (node == null) {

return;

}

// Check if node's interval overlaps with queryInterval

if (queryInterval.start <= node.interval.end && queryInterval.end >= node.interval.start) {

result.add(node.interval);

}

// If left child is not null and maxEnd >= queryInterval.start, check left subtree

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

findOverlappingIntervals(node.left, queryInterval, result);

}

// Check right subtree always

findOverlappingIntervals(node.right, queryInterval, result);

}

public static void main(String[] args) {

IntervalTree intervalTree = new IntervalTree();

intervalTree.insertInterval(15, 20);

intervalTree.insertInterval(10, 30);

intervalTree.insertInterval(5, 15);

intervalTree.insertInterval(17, 19);

intervalTree.insertInterval(12, 16);

List<Interval> overlappingIntervals = intervalTree.findOverlappingIntervals(14, 17);

System.out.println("Overlapping intervals with [14, 17]:");

for (Interval interval : overlappingIntervals) {

System.out.println("[" + interval.start + ", " + interval.end + "]");

}

intervalTree.deleteInterval(15, 20);

overlappingIntervals = intervalTree.findOverlappingIntervals(14, 17);

System.out.println("After deleting [15, 20], overlapping intervals with [14, 17]:");

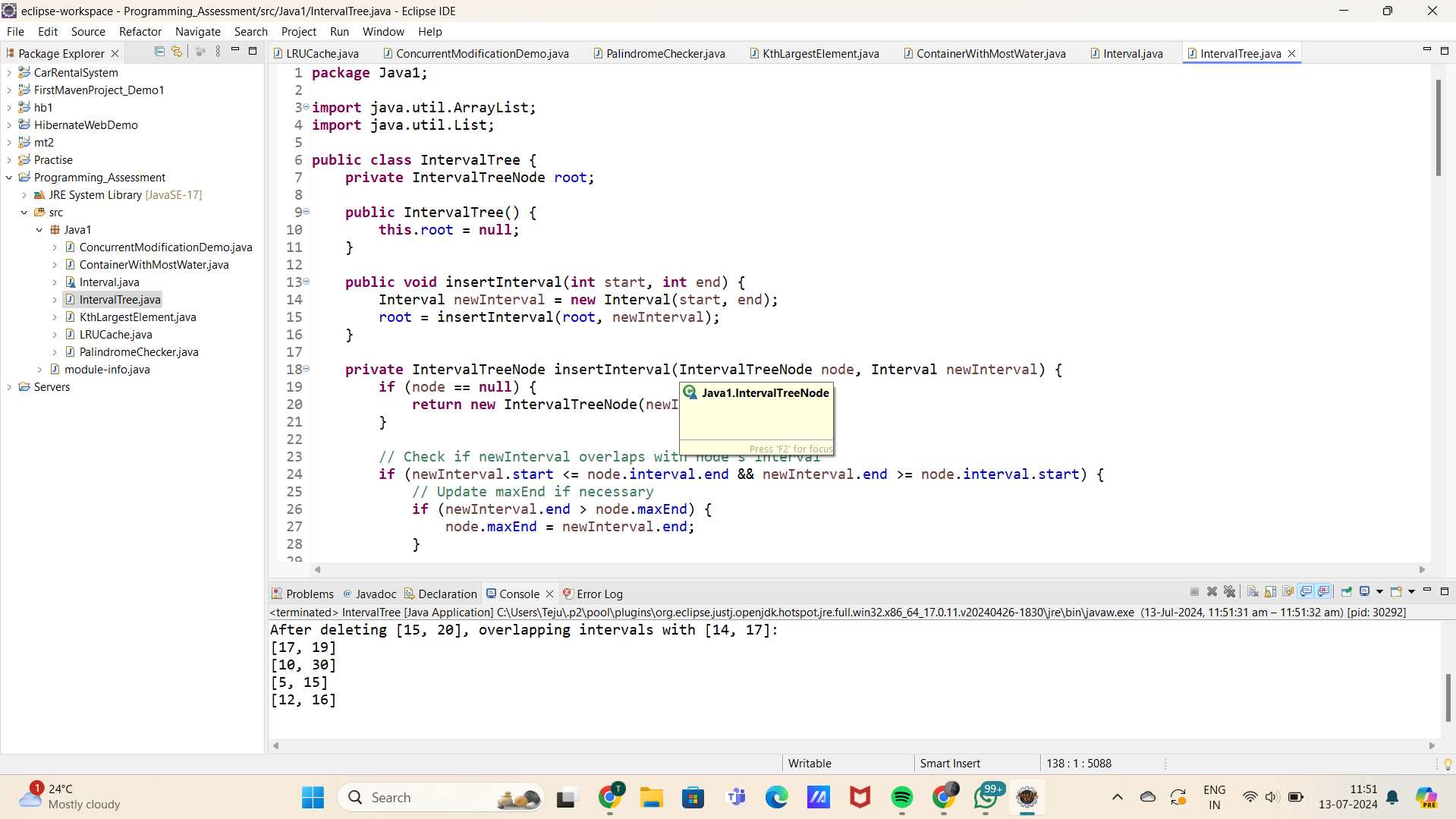
for (Interval interval : overlappingIntervals) {

System.out.println("[" + interval.start + ", " + interval.end + "]");

}

}

}



10. Write a Java program that checks if a given string is a palindrome. A palindrome is a word, phrase, number, or other sequences of characters that reads the same forward and backward (ignoring spaces, punctuation, and capitalization).

CODE:

**package** Java1;

**public** **class** PalindromeChecker {

**public** **static** **boolean** isPalindrome(String s) {

**if** (s == **null**) {

**return** **false**;

}

// Preprocess the string: remove non-alphanumeric characters and convert to lowercase

StringBuilder filtered = **new** StringBuilder();

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

**if** (Character.*isLetterOrDigit*(c)) {

filtered.append(Character.*toLowerCase*(c));

}

}

// Check if the preprocessed string is a palindrome

String filteredStr = filtered.toString();

**int** left = 0;

**int** right = filteredStr.length() - 1;

**while** (left < right) {

**if** (filteredStr.charAt(left) != filteredStr.charAt(right)) {

**return** **false**;

}

left++;

right--;

}

**return** **true**;

}

**public** **static** **void** main(String[] args) {

String[] testCases = {

"A man, a plan, a canal, Panama",

"racecar",

"No 'x' in Nixon",

"Hello, World!",

"Was it a car or a cat I saw?",

" ",

"12321",

"Not a palindrome"

};

**for** (String testCase : testCases) {

System.***out***.println("\"" + testCase + "\" is " + (*isPalindrome*(testCase) ? "" : "not ") + "a palindrome.");

}

}

}

