**GLOBAL TREND PROGRAMMING ASSESSMENT**

**Problem Statement:**

1. 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.

**Solution:**

**Code:**

import java.util.HashMap;

class LRU {

    // Node class to represent each entry in the cache

    private class Node {

        int key;

        int value;

        Node prev;

        Node next;

        Node(int key, int value) {

            this.key = key;

            this.value = value;

        }

    }

    private int capacity; // Maximum capacity of the cache

    private HashMap<Integer, Node> map; // HashMap for O(1) access

    private Node head; // Dummy head of the doubly linked list

    private Node tail; // Dummy tail of the doubly linked list

    public LRU(int capacity) {

        this.capacity = capacity;

        this.map = new HashMap<>();

        // Initialize the dummy head and tail nodes

        this.head = new Node(0, 0);

        this.tail = new Node(0, 0);

        head.next = tail;

        tail.prev = head;

    }

    // Get the value of the key if it exists in the cache

    public int get(int key) {

        if (map.containsKey(key)) {

            Node node = map.get(key);

            // Move the accessed node to the head of the linked list

            remove(node);

            insertToHead(node);

            return node.value;

        } else {

            return -1; // Return -1 if the key is not found

        }

    }

    // Set or insert the value of the key in the cache

    public void put(int key, int value) {

        if (map.containsKey(key)) {

            Node node = map.get(key);

            node.value = value;

            // Move the updated node to the head of the linked list

            remove(node);

            insertToHead(node);

        } else {

            if (map.size() == capacity) {

                // Remove the least recently used item

                map.remove(tail.prev.key);

                remove(tail.prev);

            }

            Node newNode = new Node(key, value);

            map.put(key, newNode);

            // Insert the new node at the head of the linked list

            insertToHead(newNode);

        }

    }

    // Remove a node from the linked list

    private void remove(Node node) {

        node.prev.next = node.next;

        node.next.prev = node.prev;

    }

    // Insert a node at the head of the linked list

    private void insertToHead(Node node) {

        node.next = head.next;

        node.prev = head;

        head.next.prev = node;

        head.next = node;

    }

    public static void main(String[] args) {

        LRU cache = new LRU(2);

        cache.put(1, 1);

        cache.put(2, 2);

        System.out.println("Key: 1 Value: "+cache.get(1));   // returns 1

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

        System.out.println("Key: 2 Value: "+cache.get(2));    // returns -1 (Not Found)

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

        System.out.println("Key: 1 Value: "+cache.get(1));    //returns -1 (Not Found)

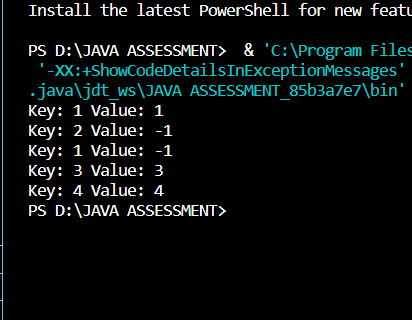
        System.out.println("Key: 3 Value: "+cache.get(3));   // returns 3

        System.out.println("Key: 4 Value: "+cache.get(4));    // returns 4

    }

}

**Output:**

****

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

**Solution:  
Code:**

import java.util.ArrayList;

import java.util.ConcurrentModificationException;

import java.util.Iterator;

public class ConcurrentmodificationexceptionDemo {

    public static void main(String[] args) {

        ArrayList<Integer> list = new ArrayList<>();

        list.add(1);

        list.add(2);

        list.add(3);

        list.add(4);

        list.add(5);

        // This will throw ConcurrentModificationException

        try {

            Iterator<Integer> it = list.iterator();

            while (it.hasNext()) {

                Integer value = it.next();

                System.out.println("List Value:" + value);

                if (value.equals(3)) {

                    list.remove(value);

                }

            }

        } catch (ConcurrentModificationException e) {

            System.out.println("Caught ConcurrentModificationException");

        }

        // Proper way to remove elements using an iterator

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

        while (iterator.hasNext()) {

            Integer value = iterator.next();

            if (value.equals(3)) {

                iterator.remove();

            }

        }

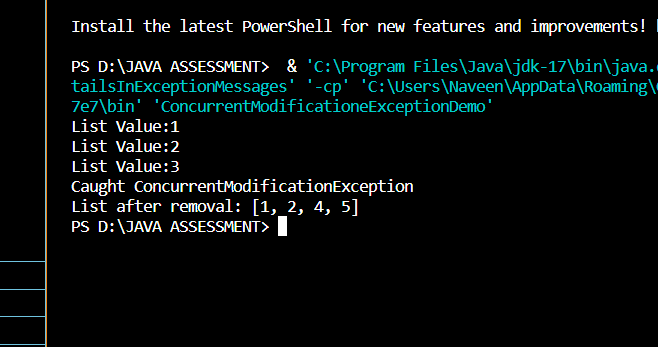
        // Printing the list after modification

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

    }

}

**Output:**

****

* **Why the Exception is Thrown:**

When using a for-each loop to iterate over the list and trying to remove an element directly using the list.remove(item) method, the ConcurrentModificationException is thrown. This is because the internal state of the list is changed directly during iteration, and the iterator detects this change and throws the exception to signal that the collection was modified outside of its expected mechanisms.

* **Proper Handling:**

The correct way to modify the collection during iteration is to use the Iterator and its remove method. The Iterator's remove method ensures that the collection is modified in a way that does not trigger the ConcurrentModificationException.

3. Create a custom annotation @LogExecutionTime to log the execution time of annotated methods. Implement an annotation processor to handle this annotation.

**Solution:**

**Code:**

**LogExecutionTime.java:**

// src/com/example/annotations/LogExecutionTime.java

package com.example.annotations;

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 {

}

**LogExecutionTimeProcessor.java:**

// src/com/example/processors/LogExecutionTimeProcessor.java

package com.example.processors;

import com.example.annotations.LogExecutionTime;

import java.lang.reflect.Method;

public class LogExecutionTimeProcessor {

    public static void process(Object obj) {

        Method[] methods = obj.getClass().getDeclaredMethods();

        for (Method method : methods) {

            if (method.isAnnotationPresent(LogExecutionTime.class)) {

                try {

                    long start = System.currentTimeMillis();

                    method.setAccessible(true);

                    method.invoke(obj);

                    long end = System.currentTimeMillis();

                    System.out.println("Execution time of " + method.getName() + ": " + (end - start) + "ms");

                } catch (Exception e) {

                    e.printStackTrace();

                }

            }

        }

    }

}

**TestClass.java:**

// src/com/example/TestClass.java

package com.example;

import com.example.annotations.LogExecutionTime;

import com.example.processors.LogExecutionTimeProcessor;

public class TestClass {

    @LogExecutionTime

    public void testMethod() {

        try {

            Thread.sleep(2000); // Simulate some work with a 2-second sleep

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

    public static void main(String[] args) {

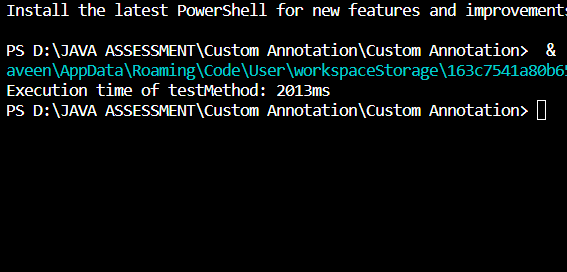
        TestClass test = new TestClass();

        LogExecutionTimeProcessor.process(test);

    }

}

**Output:**



4. Design an algorithm to serialize and deserialize a binary tree. Implement serialize(TreeNode root) which converts a tree into a string, and deserialize(String data) which converts a string back to a tree.

Constraints

The encoded string should be as compact as possible.

**Solution:**

**Code:**

import java.util.\*;

// Definition for a binary tree node.

class TreeNode {

    int val;

    TreeNode left;

    TreeNode right;

    TreeNode(int x) {

        val = x;

    }

}

public class Codec {

    // Encodes a tree to a single 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("null,");

            return;

        }

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

        serializeHelper(root.left, sb);

        serializeHelper(root.right, sb);

    }

    // Decodes your encoded data to tree.

    public TreeNode deserialize(String data) {

        String[] nodes = data.split(",");

        Queue<String> queue = new LinkedList<>(Arrays.asList(nodes));

        return deserializeHelper(queue);

    }

    private TreeNode deserializeHelper(Queue<String> queue) {

        String val = queue.poll();

        if (val.equals("null")) {

            return null;

        }

        TreeNode node = new TreeNode(Integer.parseInt(val));

        node.left = deserializeHelper(queue);

        node.right = deserializeHelper(queue);

        return node;

    }

    // Print the tree in level order to verify deserialization

    private void printTree(TreeNode root) {

        if (root == null) {

            System.out.println("Tree is empty");

            return;

        }

        Queue<TreeNode> queue = new LinkedList<>();

        queue.add(root);

        while (!queue.isEmpty()) {

            TreeNode current = queue.poll();

            if (current == null) {

                System.out.print("null ");

                continue;

            }

            System.out.print(current.val + " ");

            queue.add(current.left);

            queue.add(current.right);

        }

        System.out.println();

    }

    // Test the Codec

    public static void main(String[] args) {

        Codec codec = new Codec();

        // Example 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

        String serializedData = codec.serialize(root);

        System.out.println("Serialized data: " + serializedData);

        // Deserialize

        TreeNode deserializedRoot = codec.deserialize(serializedData);

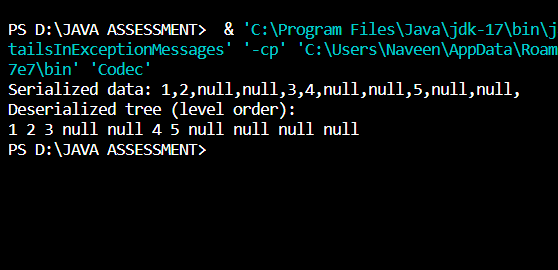
        System.out.println("Deserialized tree (level order):");

        codec.printTree(deserializedRoot);

    }

}

**Output:**



5.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.

**Solution:  
Code:**

class TrieNode {

    // Each node has an array of children and a boolean to mark the end of a word

    TrieNode[] children;

    boolean isEndOfWord;

    // Constructor to initialize the node

    public TrieNode() {

        children = new TrieNode[26]; // Each index corresponds to a letter 'a' to 'z'

        isEndOfWord = false;

    }

}

public class Trie {

    private TrieNode root;

    // Constructor to initialize the Trie with a root node

    public Trie() {

        root = new TrieNode();

    }

    // Method to 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; // Mark the end of the word

    }

    // Method to 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; // Return true only if it's the end of the word

    }

    // Method to check if any word in the trie starts with a 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; // If we can traverse the trie up to the end of the prefix, return true

    }

    // Main method for testing

    public static void main(String[] args) {

        Trie trie = new Trie();

        // Inserting words into the trie

        trie.insert("apple");

        trie.insert("app");

        // Testing search method

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

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

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

        // Testing startsWith method

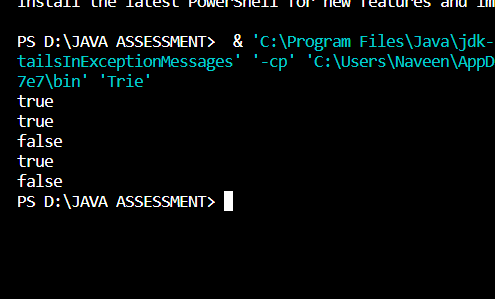
        System.out.println(trie.startsWith("app")); // true

        System.out.println(trie.startsWith("apl")); // false

    }

}

**Output:**



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.

**Solution:  
Code**:

import java.util.Stack;

public class ValidParentheses {

    public static boolean isValid(String s) {

        // Use a stack to track opening brackets

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

        // Iterate through each character in the string

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

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

                // If it's an opening bracket, push onto the stack

                stack.push(c);

            } else {

                // If it's a closing bracket

                if (stack.isEmpty()) {

                    // If stack is empty, no matching opening bracket

                    return false;

                }

                // Pop the top of the stack

                char top = stack.pop();

                // Check if it matches the corresponding opening bracket

                if ((c == ')' && top != '(') ||

                    (c == '}' && top != '{') ||

                    (c == ']' && top != '[')) {

                    return false; // Mismatched brackets

                }

            }

        }

        // Stack should be empty if all opening brackets have matching closing brackets

        return stack.isEmpty();

    }

    public static void main(String[] args) {

        String input1 = "([{}])";

        String input2 = "([)]";

        // Test cases

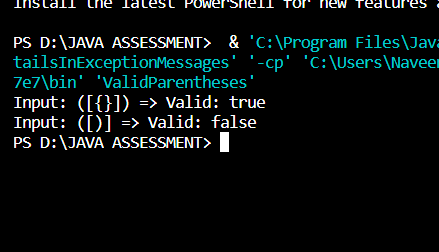
        System.out.println("Input: " + input1 + " => Valid: " + isValid(input1));

        System.out.println("Input: " + input2 + " => Valid: " + isValid(input2));

    }

}

**Output:**



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.

**Solution:**

**Code:**

public class ContainerWithMostWater {

    public static int maxArea(int[] height) {

        int maxArea = 0;

        int left = 0;

        int right = height.length - 1;

        while (left < right) {

            // Calculate current area

            int currentArea = (right - left) \* Math.min(height[left], height[right]);

            // Update max area if current area is greater

            maxArea = Math.max(maxArea, currentArea);

            // Move the pointer pointing to the smaller height towards the center

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

                left++;

            } else {

                right--;

            }

        }

        return maxArea;

    }

    public static void main(String[] args) {

        // Example usage

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

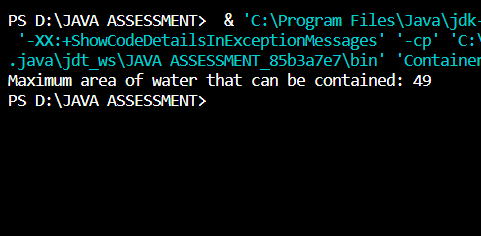
        int maxWater = maxArea(height);

        System.out.println("Maximum area of water that can be contained: " + maxWater);

    }

}

**Output:**



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.

**Solution:**

**Code**:

import java.util.Arrays;

public class KthLargestElement {

    // Function to find the kth largest element in the array

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

        // Sort the array in ascending order

        Arrays.sort(nums);

        // Return the kth largest element (since arrays are 0-indexed, nums.length - k gives the correct index)

        return nums[nums.length - k];

    }

    // Main method for testing

    public static void main(String[] args) {

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

        int k = 2;

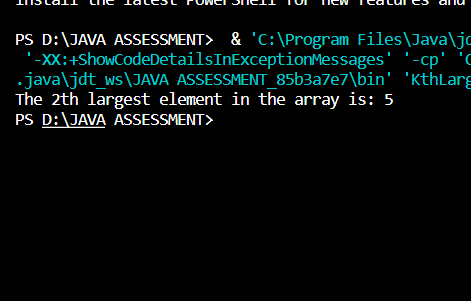
        int result = findKthLargest(nums, k);

        System.out.println("The " + k + "th largest element in the array is: " + result);

    }

}

**Output:**



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

**Solution:**

**Code:**

import java.util.\*;

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;

    }

}

public class IntervalTree {

    private IntervalTreeNode root;

    public IntervalTree() {

        this.root = null;

    }

    // Insert an interval into the interval tree

    public void insertInterval(int start, int end) {

        Interval newInterval = new Interval(start, end);

        root = insertInterval(root, newInterval);

    }

    private IntervalTreeNode insertInterval(IntervalTreeNode node, Interval interval) {

        if (node == null) {

            return new IntervalTreeNode(interval);

        }

        // Insert into left subtree

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

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

        } else { // Insert into right subtree

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

        }

        // Update maxEnd in the current node

        if (node.maxEnd < interval.end) {

            node.maxEnd = interval.end;

        }

        return node;

    }

    // Delete an interval from the interval tree

    public void deleteInterval(int start, int end) {

        Interval deleteInterval = new Interval(start, end);

        root = deleteInterval(root, deleteInterval);

    }

    private IntervalTreeNode deleteInterval(IntervalTreeNode node, Interval interval) {

        if (node == null) {

            return null;

        }

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

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

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

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

        } else { // Found the node to delete

            if (node.left == null) {

                return node.right;

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

                return node.left;

            }

            // Node with two children: Get the inorder successor (smallest in the right subtree)

            IntervalTreeNode minNode = findMin(node.right);

            // Copy the inorder successor's content to this node

            node.interval = minNode.interval;

            // Delete the inorder successor

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

        }

        // Update maxEnd in the current node

        if (node != null) {

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

        }

        return node;

    }

    private IntervalTreeNode findMin(IntervalTreeNode node) {

        while (node.left != null) {

            node = node.left;

        }

        return node;

    }

    private int maxEnd(IntervalTreeNode node) {

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

    }

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

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

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

        findOverlappingIntervals(root, start, end, result);

        return result;

    }

    private void findOverlappingIntervals(IntervalTreeNode node, int start, int end, List<Interval> result) {

        if (node == null) {

            return;

        }

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

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

            result.add(node.interval);

        }

        // Recursively search left and right subtrees if necessary

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

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

        }

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

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

        }

    }

    public static void main(String[] args) {

        IntervalTree intervalTree = new IntervalTree();

        // Insert intervals

        intervalTree.insertInterval(15, 20);

        intervalTree.insertInterval(10, 30);

        intervalTree.insertInterval(17, 19);

        intervalTree.insertInterval(5, 20);

        intervalTree.insertInterval(12, 15);

        intervalTree.insertInterval(30, 40);

        // Find intervals overlapping with [14, 16]

        List<Interval> overlappingIntervals = intervalTree.findOverlappingIntervals(14, 16);

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

        for (Interval interval : overlappingIntervals) {

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

        }

        // Delete an interval [15, 20]

        intervalTree.deleteInterval(15, 20);

        // Find intervals overlapping with [14, 16] after deletion

        overlappingIntervals = intervalTree.findOverlappingIntervals(14, 16);

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

        for (Interval interval : overlappingIntervals) {

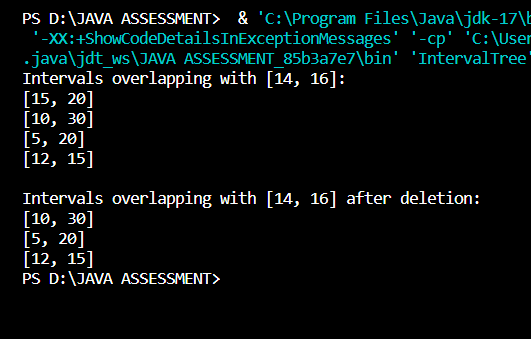
            System.out.println("[" + interval.start + ", " + interval.end + "]");

        }

    }

}

**Output:**



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).

**Solution:**

**Code:**

public class PalindromeChecker {

    // Method to check if a string is a palindrome

    public static boolean isPalindrome(String str) {

        // Remove spaces and convert to lowercase

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

        // Check palindrome

        int left = 0;

        int right = cleanStr.length() - 1;

        while (left < right) {

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

                return false; // Not a palindrome

            }

            left++;

            right--;

        }

        return true; // It is a palindrome

    }

    public static void main(String[] args) {

        // Test cases

        String str1 = "racecar";

        String str2 = "race a car";

        String str3 = "Was it a car or a cat I saw?";

        // Check if each string is a palindrome

        System.out.println("\"" + str1 + "\" is a palindrome: " + isPalindrome(str1));

        System.out.println("\"" + str2 + "\" is a palindrome: " + isPalindrome(str2));

        System.out.println("\"" + str3 + "\" is a palindrome: " + isPalindrome(str3));

    }

}

**Output:**

