**1. Write a Java program to find the longest substring from a given string that doesn’t**

**contain any duplicate characters.**

import java.util.HashMap;

import java.util.Scanner;

public class LongestSubstringWithoutDuplicates {

public static String findLongestSubstring(String s) {

if (s == null || s.length() == 0) {

return "";

}

s = s.toLowerCase();

int n = s.length();

int start = 0, end = 0;

int maxLength = 0;

int maxStart = 0;

HashMap<Character, Integer> charIndexMap = new HashMap<>();

while (end < n) {

char currentChar = s.charAt(end);

if (charIndexMap.containsKey(currentChar)) {

start = Math.max(charIndexMap.get(currentChar) + 1, start);

}

charIndexMap.put(currentChar, end);

if (end - start + 1 > maxLength) {

maxLength = end - start + 1;

maxStart = start;

}

end++;

}

if (maxStart + maxLength <= n) {

return s.substring(maxStart, maxStart + maxLength);

} else {

return "";

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter a string: ");

String input = scanner.nextLine();

String longestSubstring = findLongestSubstring(input);

System.out.println("Input String: " + input);

System.out.println("Longest Substring Without Duplicates: " + longestSubstring);

}

}

This Java program efficiently finds and displays the longest substring without duplicate characters in a user-inputted string. Employing a sliding window approach, it iterates through the string while maintaining a HashMap to track the last index of each character. The program dynamically adjusts the start index upon encountering duplicates, ensuring the window moves forward. Throughout this process, it keeps tabs on the length and starting index of the longest substring without duplicates. The user-friendly `main` method prompts for a string input and outputs both the original string and the calculated longest substring without duplicates.

**2. Write a Program for the first 10 numbers of Fibonacci series.**

public class FibonacciSeries {

public static void main(String[] args) {

int n = 10;

int[] fibonacciSeries = new int[n];

fibonacciSeries[0] = 0;

fibonacciSeries[1] = 1;

for (int i = 2; i < n; i++) {

fibonacciSeries[i] = fibonacciSeries[i - 1] + fibonacciSeries[i - 2];

}

System.out.println("First " + n + " numbers of Fibonacci series:");

for (int i = 0; i < n; i++) {

System.out.print(fibonacciSeries[i] + " ");

}

}

}

This Java program generates and prints the first 10 numbers of the Fibonacci series. It initializes an array to store the series, setting the first two values manually to 0 and 1. The subsequent Fibonacci numbers are calculated by summing the previous two elements in the series using a for loop. Finally, the program prints the first 10 Fibonacci numbers, displaying them in the console. The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones, typically starting with 0 and 1.

**3. Write a program to print all permutations of a string.**

import java.util.Scanner;

public class StringPermutations {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter a string: ");

String input = scanner.nextLine();

System.out.println("Input: " + input);

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

printPermutations(input);

}

public static void printPermutations(String input) {

printPermutationsHelper(input.toCharArray(), 0, input.length() - 1);

}

private static void printPermutationsHelper(char[] charArray, int start, int end) {

if (start == end) {

System.out.println(new String(charArray));

} else {

for (int i = start; i <= end; i++) {

swap(charArray, start, i);

printPermutationsHelper(charArray, start + 1, end);

swap(charArray, start, i);

}

}

}

private static void swap(char[] charArray, int i, int j) {

char temp = charArray[i];

charArray[i] = charArray[j];

charArray[j] = temp;

}

}

This Java program prompts the user to input a string and then recursively generates and prints all possible permutations of the entered string. The recursive algorithm is implemented in the `printPermutationsHelper` method, which swaps characters to create permutations. The program employs a simple interactive approach, making it user-friendly as it dynamically showcases the permutations of the provided string.

**4. Write a program to check if two strings are Anagrams?**

import java.util.Arrays;

import java.util.Scanner;

public class AnagramChecker {

public static boolean areAnagrams(String str1, String str2) {

if (str1 == null || str2 == null || str1.length() != str2.length()) {

return false;

}

char[] charArray1 = str1.toLowerCase().toCharArray();

char[] charArray2 = str2.toLowerCase().toCharArray();

Arrays.sort(charArray1);

Arrays.sort(charArray2);

return Arrays.equals(charArray1, charArray2);

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the first string: ");

String str1 = scanner.nextLine();

System.out.print("Enter the second string: ");

String str2 = scanner.nextLine();

if (areAnagrams(str1, str2)) {

System.out.println(str1 + " and " + str2 + " are anagrams.");

} else {

System.out.println(str1 + " and " + str2 + " are not anagrams.");

}

}

}

This Java program checks if two inputted strings are anagrams. It utilizes character arrays and sorting to compare the characters in a case-insensitive manner. The user is prompted to enter two strings, and the program outputs whether the provided strings are anagrams or not. The straightforward design allows users to quickly assess the anagram relationship between the inputted strings.

**5. Implement stack using queue without using an array or linked list.**

import java.util.LinkedList;

import java.util.Queue;

import java.util.Scanner;

public class InteractiveStackUsingQueues {

private Queue<Integer> primaryQueue;

private Queue<Integer> auxiliaryQueue;

public InteractiveStackUsingQueues() {

primaryQueue = new LinkedList<>();

auxiliaryQueue = new LinkedList<>();

}

public void push(int value) {

while (!primaryQueue.isEmpty()) {

auxiliaryQueue.add(primaryQueue.poll());

}

primaryQueue.add(value);

while (!auxiliaryQueue.isEmpty()) {

primaryQueue.add(auxiliaryQueue.poll());

}

}

public int pop() {

if (primaryQueue.isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

return primaryQueue.poll();

}

public int top() {

if (primaryQueue.isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

return primaryQueue.peek();

}

public boolean isEmpty() {

return primaryQueue.isEmpty();

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

InteractiveStackUsingQueues stack = new InteractiveStackUsingQueues();

while (true) {

System.out.println("\nOptions:");

System.out.println("1. Push");

System.out.println("2. Pop");

System.out.println("3. Top");

System.out.println("4. Exit");

System.out.print("Enter your choice: ");

int choice = scanner.nextInt();

switch (choice) {

case 1:

System.out.print("Enter the element to push: ");

int pushValue = scanner.nextInt();

stack.push(pushValue);

System.out.println("Element " + pushValue + " pushed onto the stack.");

break;

case 2:

try {

int poppedValue = stack.pop();

System.out.println("Popped element: " + poppedValue);

} catch (IllegalStateException e) {

System.out.println("Error: " + e.getMessage());

}

break;

case 3:

try {

int topValue = stack.top();

System.out.println("Top element: " + topValue);

} catch (IllegalStateException e) {

System.out.println("Error: " + e.getMessage());

}

break;

case 4:

System.out.println("Exiting the program.");

System.exit(0);

default:

System.out.println("Invalid choice. Please enter a valid option.");

}

}

}

}

This Java program implements an interactive stack using two queues. The `InteractiveStackUsingQueues` class defines two queues (`primaryQueue` and `auxiliaryQueue`) and provides methods for pushing, popping, and retrieving the top element from the stack. The `main` method creates an instance of this stack, and the user is presented with a menu to push a new element, pop the top element, check the top element, or exit the program. The program continuously accepts user input, executes the chosen operation, and displays appropriate messages or errors. The implementation ensures the stack operations are performed efficiently using the two queues, providing a user-friendly interface for stack manipulation.

**6. Write a program to reverse a string.**

import java.util.Scanner;

public class StringReverser {

public static String reverseString(String input) {

char[] charArray = input.toCharArray();

int start = 0;

int end = charArray.length - 1;

while (start < end) {

char temp = charArray[start];

charArray[start] = charArray[end];

charArray[end] = temp;

start++;

end--;

}

return new String(charArray);

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter a string: ");

String input = scanner.nextLine();

String reversedString = reverseString(input);

System.out.println("Original string: " + input);

System.out.println("Reversed string: " + reversedString);

}

}

This Java program reverses a user-inputted string using an in-place character swapping technique. The `StringReverser` class defines a `reverseString` method that converts the input string into a character array and then swaps characters from both ends until the entire string is reversed. The `main` method prompts the user to enter a string, calls the `reverseString` method, and prints both the original and reversed strings. The program provides an interactive way for users to quickly see the reversed version of their inputted string.