LAB 5

.

NAME:ĢHULAM MUSTAFA CMS ID:023-22-0099

TASK 0

DESCRIPTION: THE FOLLOWING CODE WILL IMPLEMENTS ALL GIVEN REQUIREMENTS

import java.util.Scanner;

public class Task0 {

public static int RecursionAscending(int no) {

if (no > 0) {

RecursionAscending(no - 1);

System.out.println(no);

}

return 0;

}

public static int RecursionDescending(int no) {

if (no > 0) {

System.out.println(no);

RecursionDescending(no - 1);

}

return 0;

}

public static void IdRec(String str, int count) {

if (count < str.length()) {

System.out.println(str.charAt(count));

count++;

IdRec(str, count);

}

}

public static void main(String[] args) {

int no;

Scanner sc = new Scanner(System.in);

System.out.println("Enter the range:");

no = sc.nextInt();

sc.nextLine(); // Consume the newline character

System.out.println("Series in Ascending order:");

RecursionAscending(no);

System.out.println("Series in Descending order:");

RecursionDescending(no);

System.out.println("Enter the String:");

String str = sc.nextLine();

IdRec(str, 0);

}

}

TASK 1

DESCRIPTION: THE FOLLOWING CODE WILL IMPLEMENTS ALL GIVEN REQUIREMENTS

Part A

import java.util.Scanner;

public class FibonacciIterative {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the value of N: ");

int n = sc.nextInt();

sc.close();

int prev = 0, curr = 1;

System.out.print("Fibonacci Series (Iterative): ");

while (prev <= n) {

System.out.print(prev + " ");

int next = prev + curr;

prev = curr;

curr = next;

}

}

}

PART B

import java.util.Scanner;

public class FibonacciRecursive {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the value of N: ");

int n = sc.nextInt();

sc.close();

System.out.print("Fibonacci Series (Recursive): ");

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

System.out.print(fibonacci(i) + " ");

}

}

public static int fibonacci(int n) {

if (n <= 1) {

return n;

} else {

return fibonacci(n - 1) + fibonacci(n - 2);

}

}

}

PART C

import java.util.Scanner;

public class FibonacciComparison {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the value of N: ");

int n = sc.nextInt();

sc.close();

long startTimeIterative = System.nanoTime();

generateFibonacciIterative(n);

long endTimeIterative = System.nanoTime();

long startTimeRecursive = System.nanoTime();

generateFibonacciRecursive(n, 0, 1);

long endTimeRecursive = System.nanoTime();

long durationIterative = (endTimeIterative - startTimeIterative) / 1000; // Convert to microseconds

long durationRecursive = (endTimeRecursive - startTimeRecursive) / 1000; // Convert to microseconds

System.out.println("Time taken by Iterative approach (microseconds): " + durationIterative);

System.out.println("Time taken by Recursive approach (microseconds): " + durationRecursive);

if (durationIterative < durationRecursive) {

System.out.println("Iterative approach is faster.");

} else {

System.out.println("Recursive approach is faster.");

}

}

public static void generateFibonacciIterative(int n) {

int prev = 0, curr = 1;

System.out.print("Fibonacci Series (Iterative): ");

while (prev <= n) {

System.out.print(prev + " ");

int next = prev + curr;

prev = curr;

curr = next;

}

System.out.println();

}

public static void generateFibonacciRecursive(int n, int prev, int curr) {

if (prev <= n) {

System.out.print(prev + " ");

generateFibonacciRecursive(n, curr, prev + curr);

} else {

System.out.println();

}

}

}

TASK2

DESCRIPTION: THE FOLLOWING CODE WILL IMPLEMENTS ALL GIVEN REQUIREMENTS

//Iterative approach;

import java.util.Scanner;

public class FactorialIterative {

public static long calculateFactorial(int n) {

if (n < 0) {

return -1; // Handle invalid input (negative number)

}

long factorial = 1;

for (int i = 1; i <= n; i++) {

factorial \*= i;

}

return factorial;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter a non-negative integer N: ");

int n = sc.nextInt();

sc.close();

if (n < 0) {

System.out.println("Factorial is not defined for negative numbers.");

} else {

long startTime = System.nanoTime();

long result = calculateFactorial(n);

long endTime = System.nanoTime();

System.out.println("Factorial (Iterative) of " + n + ": " + result);

long duration = endTime - startTime; // Calculate time difference in nanoseconds

System.out.println("Time taken (nanoseconds): " + duration);

}

}

}

//   Recursive Approach

import java.util.Scanner;

public class FactorialRecursive {

public static long calculateFactorial(int n) {

if (n < 0) {

return -1; // Handle invalid input (negative number)

}

if (n == 0 || n == 1) {

return 1; // Base case: factorial of 0 and 1 is 1

}

return n \* calculateFactorial(n - 1);

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter a non-negative integer N: ");

int n = sc.nextInt();

sc.close();

if (n < 0) {

System.out.println("Factorial is not defined for negative numbers.");

} else {

long startTime = System.nanoTime();

long result = calculateFactorial(n);

long endTime = System.nanoTime();

System.out.println("Factorial (Recursive) of " + n + ": " + result);

long duration = endTime - startTime; // Calculate time difference in nanoseconds

System.out.println("Time taken (nanoseconds): " + duration);

}

}

}

TASK 3

DESCRIPTION: THE FOLLOWING CODE WILL IMPLEMENTS ALL GIVEN REQUIREMENTS

//Iterative approach;

import java.util.LinkedList;

public class LinkedListPrintIterativeTimeComplexity {

public static void main(String[] args) {

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

// Populate the linked list with some data

for (int i = 1; i <= 1000; i++) {

list.add(i);

}

long startTime = System.nanoTime();

System.out.print("Linked List (Iterative): ");

for (Integer num : list) {

System.out.print(num + " ");

}

System.out.println();

long endTime = System.nanoTime();

long duration = endTime - startTime; // Calculate time difference

System.out.println("Time taken (nanoseconds): " + duration);

}

}

// Recursive Approach

import java.util.LinkedList;

public class LinkedListPrintRecursiveTimeComplexity {

public static void printLinkedListRecursive(LinkedList<Integer> list, int index) {

if (index < list.size()) {

System.out.print(list.get(index) + " ");

printLinkedListRecursive(list, index + 1);

}

}

public static void main(String[] args) {

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

// Populate the linked list with some data

for (int i = 1; i <= 1000; i++) {

list.add(i);

}

long startTime = System.nanoTime();

System.out.print("Linked List (Recursive): ");

printLinkedListRecursive(list, 0);

System.out.println();

long endTime = System.nanoTime();

long duration = endTime - startTime; // Calculate time difference

System.out.println("Time taken (nanoseconds): " + duration);

}

}

TASK 4

DESCRIPTION: THE FOLLOWING CODE WILL IMPLEMENTS ALL GIVEN REQUIREMENTS

import java.util.Random;

import java.util.Scanner;

public class LinearSearchComparison {

public static int linearSearchIterative(int[] arr, int searchValue) {

for (int i = 0; i < arr.length; i++) {

if (arr[i] == searchValue) {

return 1; // Search value found

}

}

return 0; // Search value not found

}

public static int linearSearchRecursive(int[] arr, int searchValue, int index) {

if (index < arr.length) {

if (arr[index] == searchValue) {

return 1; // Search value found

} else {

return linearSearchRecursive(arr, searchValue, index + 1);

}

}

return 0; // Search value not found

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

Random rand = new Random();

System.out.print("Enter the size of the array: ");

int size = sc.nextInt();

int[] arr = new int[size];

// Populate the array with random values

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

arr[i] = rand.nextInt(100); // Random values between 0 and 99

}

System.out.print("Enter the search value: ");

int searchValue = sc.nextInt();

sc.close();

// Linear Search - Iterative

long startTimeIterative = System.nanoTime();

int resultIterative = linearSearchIterative(arr, searchValue);

long endTimeIterative = System.nanoTime();

long durationIterative = endTimeIterative - startTimeIterative;

// Linear Search - Recursive

long startTimeRecursive = System.nanoTime();

int resultRecursive = linearSearchRecursive(arr, searchValue, 0);

long endTimeRecursive = System.nanoTime();

long durationRecursive = endTimeRecursive - startTimeRecursive;

System.out.println("Using Iterative Linear Search:");

if (resultIterative == 1) {

System.out.println("Search value found.");

} else {

System.out.println("Search value not found.");

}

System.out.println("Time taken (nanoseconds): " + durationIterative);

System.out.println("\nUsing Recursive Linear Search:");

if (resultRecursive == 1) {

System.out.println("Search value found.");

} else {

System.out.println("Search value not found.");

}

System.out.println("Time taken (nanoseconds): " + durationRecursive);

}

}

TASK 5

DESCRIPTION: THE FOLLOWING CODE WILL IMPLEMENTS ALL GIVEN REQUIREMENTS

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

public class ReverseLinkedList {

public Node makeReverse(Node head) {

Node prev = null;

Node current = head;

Node nextNode = null;

while (current != null) {

nextNode = current.next; // Store the next node

current.next = prev; // Reverse the pointer

// Move prev and current pointers one step forward

prev = current;

current = nextNode;

}

// Update the head to the last node (prev)

head = prev;

return head;

}

// Utility method to print the linked list

public static void printLinkedList(Node head) {

Node current = head;

while (current != null) {

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

current = current.next;

}

System.out.println();

}

public static void main(String[] args) {

// Create a sample linked list

Node head = new Node(1);

head.next = new Node(2);

head.next.next = new Node(3);

head.next.next.next = new Node(4);

head.next.next.next.next = new Node(5);

System.out.print("Original Linked List: ");

printLinkedList(head);

ReverseLinkedList reverseList = new ReverseLinkedList();

head = reverseList.makeReverse(head);

System.out.print("Reversed Linked List: ");

printLinkedList(head);

}

}

TASK 6

DESCRIPTION: THE FOLLOWING CODE WILL IMPLEMENTS ALL GIVEN REQUIREMENTS

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

public class PrintReverseLinkedList {

private Node head;

// Constructor to initialize the head of the linked list

public PrintReverseLinkedList() {

head = null;

}

// Method to insert a new node at the end of the linked list

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

} else {

Node current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

// Recursive method to print the linked list in reverse order

public void printReverse() {

printReverseRecursive(head);

}

private void printReverseRecursive(Node current) {

if (current == null) {

return; // Base case: reached the end of the list

}

// Recursively call the function for the next node

printReverseRecursive(current.next);

// Print the data of the current node in reverse order

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

}

public static void main(String[] args) {

PrintReverseLinkedList linkedList = new PrintReverseLinkedList();

linkedList.insert(1);

linkedList.insert(2);

linkedList.insert(3);

linkedList.insert(4);

linkedList.insert(5);

System.out.print("Linked List in Reverse Order: ");

linkedList.printReverse();

}

}

TASK 7

DESCRIPTION: THE FOLLOWING CODE WILL IMPLEMENTS ALL GIVEN REQUIREMENTS

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

public class LinkedList {

private Node head;

public boolean hasCycle() {

if (head == null || head.next == null) {

return false; // No cycle if the list is empty or has only one node

}

Node slow = head;

Node fast = head;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

if (slow == fast) {

return true; // Cycle detected

}

}

return false; // No cycle detected

}

}

TASK 8

DESCRIPTION: THE FOLLOWING CODE WILL IMPLEMENTS ALL GIVEN REQUIREMENTS

import java.util.Stack;

import java.util.Scanner;

public class BalancedBracketsSimple {

public static boolean isBalanced(String input) {

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

for (char bracket : input.toCharArray()) {

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

stack.push(bracket);

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

stack.pop();

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

stack.pop();

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

stack.pop();

} else {

return false; // Unmatched bracket

}

}

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

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

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

String input = sc.nextLine();

sc.close();

boolean balanced = isBalanced(input);

if (balanced) {

System.out.println("The string has balanced brackets.");

} else {

System.out.println("The string does not have balanced brackets.");

}

}

}