Task 1:

Create  a custom node , add elements to it and traverse it..

import java.util.ArrayList;

import java.util.List;

class Node {

// Each node holds a value and a list of children nodes

double value;

List<Node> children;

// Constructor to create a node with a given value

public Node(double value) {

this.value = value;

this.children = new ArrayList<>();

}

// Add a child to the current node

public void addChild(Node child) {

this.children.add(child);

}

}

public class Main {

// DFS Traversal method

public static void dfsTraverse(Node node) {

// Visit the current node

System.out.print(node.value + " ");

// Recursively visit all children of the current node

for (Node child : node.children) {

dfsTraverse(child);

}

}

public static void main(String[] args) {

// Create nodes

Node root = new Node(9.37);

Node child1 = new Node(9.40);

Node child2 = new Node(9.45);

// Add children to the root node

root.addChild(child1);

root.addChild(child2);

// Optionally, add subchildren to the child nodes

child1.addChild(new Node(9.41));

child2.addChild(new Node(9.46));

// Perform DFS Traversal

System.out.println("DFS Traversal:");

dfsTraverse(root);

}

}

Task 2:

What do you understand by traversing a linked list?

class Node {

int value;

Node next;

// Constructor to initialize a node

public Node(int value) {

this.value = value;

this.next = null; // The next pointer is initially null

}

}

class LinkedList {

Node head;

// Constructor to initialize an empty linked list

public LinkedList() {

head = null;

}

// Method to add a node at the end of the list

public void add(int value) {

Node newNode = new Node(value);

if (head == null) {

head = newNode; // If the list is empty, the new node becomes the head

} else {

Node current = head;

while (current.next != null) {

current = current.next; // Traverse to the last node

}

current.next = newNode; // Append the new node to the end of the list

}

}

// Method to traverse and print the linked list

public void traverse() {

Node current = head;

while (current != null) {

System.out.print(current.value + " "); // Visit the current node

current = current.next; // Move to the next node

}

System.out.println();

}

}

public class Main {

public static void main(String[] args) {

LinkedList list = new LinkedList();

// Add some nodes

list.add(10);

list.add(20);

list.add(30);

list.add(40);

// Traverse and print the linked list

System.out.println("Linked List Traversal:");

list.traverse(); // Output: 10 20 30 40

}

}

Task 3:

Create a Circular Linked list using Task 1 Singly linked list/ doubly linked list.

class Node {

int value;

Node next;

// Constructor to initialize a node

public Node(int value) {

this.value = value;

this.next = null; // The next pointer is initially null

}

}

class CircularLinkedList {

Node head;

// Constructor to initialize an empty circular linked list

public CircularLinkedList() {

head = null;

}

// Method to add a node at the end of the circular linked list

public void add(int value) {

Node newNode = new Node(value);

if (head == null) {

head = newNode; // If the list is empty, the new node becomes the head

newNode.next = head; // Point the new node's next to itself (circular)

} else {

Node current = head;

// Traverse the list to find the last node

while (current.next != head) {

current = current.next;

}

current.next = newNode; // Link the last node to the new node

newNode.next = head; // Make the new node point back to the head (circular)

}

}

// Method to traverse and print the circular linked list

public void traverse() {

if (head == null) {

System.out.println("The list is empty.");

return;

}

Node current = head;

do {

System.out.print(current.value + " "); // Visit the current node

current = current.next; // Move to the next node

} while (current != head); // Stop when we return to the head

System.out.println();

}

}

public class Main {

public static void main(String[] args) {

CircularLinkedList list = new CircularLinkedList();

// Add some nodes to the circular linked list

list.add(10);

list.add(20);

list.add(30);

list.add(40);

// Traverse and print the circular linked list

System.out.println("Circular Linked List Traversal:");

list.traverse(); // Output: 10 20 30 40

}

}

Task 4:

Delete a node in the circular linked list

class Node {

int value;

Node next;

// Constructor to initialize a node

public Node(int value) {

this.value = value;

this.next = null; // The next pointer is initially null

}

}

class CircularLinkedList {

Node head;

// Constructor to initialize an empty circular linked list

public CircularLinkedList() {

head = null;

}

// Method to add a node at the end of the circular linked list

public void add(int value) {

Node newNode = new Node(value);

if (head == null) {

head = newNode;

newNode.next = head; // Point to itself (circular)

} else {

Node current = head;

// Traverse the list to find the last node

while (current.next != head) {

current = current.next;

}

current.next = newNode; // Link last node to new node

newNode.next = head; // Make new node point to head (circular)

}

}

// Method to delete a node with a specific value

public void delete(int value) {

if (head == null) {

System.out.println("The list is empty.");

return;

}

// If the node to delete is the head node

if (head.value == value) {

if (head.next == head) { // Only one node

head = null; // List becomes empty

} else {

Node current = head;

// Traverse to the last node

while (current.next != head) {

current = current.next;

}

current.next = head.next; // Make the last node point to the second node

head = head.next; // Move head to the next node

}

return;

}

// Delete a node other than the head

Node current = head;

while (current.next != head && current.next.value != value) {

current = current.next;

}

// If the node is found

if (current.next.value == value) {

current.next = current.next.next; // Skip the node to delete

} else {

System.out.println("Node with value " + value + " not found.");

}

}

// Method to traverse and print the circular linked list

public void traverse() {

if (head == null) {

System.out.println("The list is empty.");

return;

}

Node current = head;

do {

System.out.print(current.value + " "); // Visit the current node

current = current.next; // Move to the next node

} while (current != head); // Stop when we return to the head

System.out.println();

}

}

public class Main {

public static void main(String[] args) {

CircularLinkedList list = new CircularLinkedList();

// Add some nodes to the circular linked list

list.add(10);

list.add(20);

list.add(30);

list.add(40);

// Traverse and print the circular linked list

System.out.println("Circular Linked List Traversal:");

list.traverse(); // Output: 10 20 30 40

// Delete a node (e.g., node with value 30)

System.out.println("\nDeleting node with value 30:");

list.delete(30);

list.traverse(); // Output: 10 20 40

// Try to delete a node that doesn't exist

System.out.println("\nTrying to delete a node with value 50:");

list.delete(50); // Output: Node with value 50 not found.

// Delete the head node (e.g., node with value 10)

System.out.println("\nDeleting head node with value 10:");

list.delete(10);

list.traverse(); // Output: 20 40

}

}

import java.util.Stack;public class Stack1 {    public static void main(String[] args) {        Stack<Integer> stack = new Stack<>();        stack.push(10);        stack.push(20);        stack.push(30);        System.out.println("Stack: " + stack);        int poppedElement = stack.pop();        System.out.println("Popped element: " + poppedElement);        System.out.println("Stack after pop: " + stack);    }}

Task 6:

Find an element in the stack and display the position

Hint 👍

Int position = names.search(“value”);

import java.util.Stack;

public class StackSearchExample {

public static void main(String[] args) {

// Creating a stack of String

Stack<String> names = new Stack<>();

// Pushing some elements onto the stack

names.push("Alice");

names.push("Bob");

names.push("Charlie");

names.push("David");

names.push("Eve");

// Searching for an element (e.g., "Charlie")

String searchElement = "Charlie";

int position = names.search(searchElement);

// Displaying the result

if (position != -1) {

System.out.println("Element '" + searchElement + "' is found at position: " + position);

} else {

System.out.println("Element '" + searchElement + "' not found in the stack.");

}

// Example of searching for an element not in the stack

String notFoundElement = "John";

int notFoundPosition = names.search(notFoundElement);

if (notFoundPosition == -1) {

System.out.println("Element '" + notFoundElement + "' not found in the stack.");

}

}

}

Task 7:

Peek the element and print it ..

import java.util.Stack;

public class StackPeekExample {

public static void main(String[] args) {

// Creating a stack of String

Stack<String> names = new Stack<>();

// Pushing some elements onto the stack

names.push("Alice");

names.push("Bob");

names.push("Charlie");

names.push("David");

names.push("Eve");

// Using peek() to view the top element without removing it

String topElement = names.peek();

// Displaying the top element

System.out.println("Top element in the stack: " + topElement);

}

}

Task 8:

Check if the stack is empty or not?

import java.util.Stack;

public class StackEmptyCheckExample {

public static void main(String[] args) {

// Creating a stack of String

Stack<String> names = new Stack<>();

// Checking if the stack is empty (Initially it is)

if (names.isEmpty()) {

System.out.println("The stack is empty.");

} else {

System.out.println("The stack is not empty.");

}

// Pushing some elements onto the stack

names.push("Alice");

names.push("Bob");

names.push("Charlie");

// Checking if the stack is empty after adding elements

if (names.isEmpty()) {

System.out.println("The stack is empty.");

} else {

System.out.println("The stack is not empty.");

}

// Popping all elements to make the stack empty again

names.pop();

names.pop();

names.pop();

// Checking if the stack is empty after popping all elements

if (names.isEmpty()) {

System.out.println("The stack is empty.");

} else {

System.out.println("The stack is not empty.");

}

}

}

Task 9:

What are the methods of the stack class.. List them down.. With a one liner..

import java.util.Stack;

public class StackMethodsExample {

public static void main(String[] args) {

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

// Push elements onto the stack

stack.push("Alice");

stack.push("Bob");

stack.push("Charlie");

// Peek at the top element

System.out.println("Top element: " + stack.peek()); // Output: Charlie

// Pop the top element

System.out.println("Popped element: " + stack.pop()); // Output: Charlie

// Check if the stack is empty

System.out.println("Is stack empty? " + stack.isEmpty()); // Output: false

// Search for an element in the stack

System.out.println("Position of 'Bob': " + stack.search("Bob")); // Output: 1 (Topmost is at position 1)

// Get the size of the stack

System.out.println("Size of stack: " + stack.size()); // Output: 2

// Clear the stack (optional way, since clear() is inherited from Vector)

stack.removeAllElements();

System.out.println("Stack after clearing: " + stack); // Output: []

}

}

Task 9:

What are the methods of the stack class.. List them down.. With a one liner..

**Methods of the Stack Class:**

1. **push(E item)**
   * Adds the specified element to the top of the stack.
2. **pop()**
   * Removes and returns the top element of the stack.
3. **peek()**
   * Returns the top element without removing it from the stack.
4. **isEmpty()**
   * Returns true if the stack is empty, otherwise false.
5. **search(Object o)**
   * Returns the 1-based position of the element in the stack, or -1 if not found.
6. **size()**
   * Returns the number of elements in the stack.
7. **removeAllElements()**
   * Clears all elements from the stack (this method is inherited from the Vector class, which Stack extends).

Task 10:

Wap to create  a queue with custom methods

Is empty ()

Is full()

Enque

Deque

Peek

display()

class CustomQueue {

private int[] queue;

private int front;

private int rear;

private int size;

// Constructor to initialize the queue with a given size

public CustomQueue(int size) {

this.size = size;

queue = new int[size];

front = -1;

rear = -1;

}

// Check if the queue is empty

public boolean isEmpty() {

return front == -1;

}

// Check if the queue is full

public boolean isFull() {

return rear == size - 1;

}

// Enqueue: Add an element to the queue

public void enqueue(int item) {

if (isFull()) {

System.out.println("Queue is full, cannot enqueue " + item);

} else {

if (front == -1) {

front = 0; // First element to be enqueued

}

rear++;

queue[rear] = item;

System.out.println(item + " added to the queue.");

}

}

// Dequeue: Remove and return the front element

public int dequeue() {

if (isEmpty()) {

System.out.println("Queue is empty, cannot dequeue.");

return -1; // Return -1 if the queue is empty

} else {

int item = queue[front];

front++;

if (front > rear) {

front = rear = -1; // Reset the queue when it becomes empty

}

return item;

}

}

// Peek: Return the front element without removing it

public int peek() {

if (isEmpty()) {

System.out.println("Queue is empty, cannot peek.");

return -1; // Return -1 if the queue is empty

} else {

return queue[front];

}

}

// Display: Print all elements in the queue

public void display() {

if (isEmpty()) {

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

} else {

System.out.print("Queue contents: ");

for (int i = front; i <= rear; i++) {

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

}

System.out.println();

}

}

}

public class Main {

public static void main(String[] args) {

// Creating a queue of size 5

CustomQueue queue = new CustomQueue(5);

// Displaying the initial state of the queue

queue.display();

// Enqueuing elements

queue.enqueue(10);

queue.enqueue(20);

queue.enqueue(30);

queue.enqueue(40);

queue.enqueue(50);

// Display the queue after enqueue operations

queue.display();

// Attempting to enqueue when the queue is full

queue.enqueue(60); // Queue is full

// Peeking the front element

System.out.println("Peek front element: " + queue.peek());

// Dequeuing elements

System.out.println("Dequeued: " + queue.dequeue());

System.out.println("Dequeued: " + queue.dequeue());

// Display the queue after dequeue operations

queue.display();

// Checking if the queue is empty

System.out.println("Is queue empty? " + queue.isEmpty());

// Dequeuing remaining elements

queue.dequeue();

queue.dequeue();

queue.dequeue();

// Display after all elements are dequeued

queue.display();

}

}