Task 1:

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

package July4;  
  
// Node class  
class Node {  
 int data;  
 Node next;  
  
 Node(int data) {  
 this.data = data;  
 this.next = null;  
 }  
}  
  
// Singly Linked List class  
public class SinglyLinkedList {  
 Node head;  
  
 // Add node at the end  
 public void append(int data) {  
 Node newNode = new Node(data);  
  
 if (head == null) {  
 head = newNode;  
 return;  
 }  
  
 Node current = head;  
 while (current.next != null) {  
 current = current.next;  
 }  
  
 current.next = newNode;  
 }  
  
 // Traverse and print the list  
 public void traverse() {  
 Node current = head;  
 while (current != null) {  
 System.*out*.print(current.data + " -> ");  
 current = current.next;  
 }  
 System.*out*.println("null");  
 }  
  
 // Main method to run the code  
 public static void main(String[] args) {  
 SinglyLinkedList list = new SinglyLinkedList();  
  
 list.append(10);  
 list.append(20);  
 list.append(30);  
 list.append(40);  
  
 list.traverse();  
 }  
}

output:

10 -> 20 -> 30 -> 40 -> null

Process finished with exit code 0

Task 2

What do you understand by traversing elements in a linked list.

* To display all the elements
* To search for a specific value
* To count the number of elements
* To perform operations on each node (e.g., sum, modification)

Task 3:

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

// Node class for Circular Linked List  
class Node1 {  
 int data;  
 Node1 next;  
  
 Node1(int data) {  
 this.data = data;  
 this.next = null;  
 }  
}  
  
// Custom Circular Linked List class  
class CircularLinkedList {  
 Node1 head = null;  
 Node1 tail = null;  
  
 // Method to add a node to the list  
 public void add(int data) {  
 Node1 newNode = new Node1(data);  
  
 if (head == null) {  
 head = newNode;  
 tail = newNode;  
 newNode.next = head; // Points to itself (circular)  
 } else {  
 tail.next = newNode; // Link new node after tail  
 tail = newNode; // Update tail  
 tail.next = head; // Make it circular  
 }  
 }  
  
 // Method to display the list  
 public void display() {  
 if (head == null) {  
 System.*out*.println("List is empty.");  
 return;  
 }  
  
 Node1 current = head;  
 System.*out*.print("Circular Linked List: ");  
 do {  
 System.*out*.print(current.data + " -> ");  
 current = current.next;  
 } while (current != head);  
 System.*out*.println("(back to head)");  
 }  
}  
  
// Main class to run the example  
public class circular {  
 public static void main(String[] args) {  
 CircularLinkedList cll = new CircularLinkedList();  
  
 cll.add(10);  
 cll.add(20);  
 cll.add(30);  
 cll.add(40);  
  
 cll.display();  
 }  
}

// Node class for Circular Linked List  
class Node1 {  
 int data;  
 Node1 next;  
  
 Node1(int data) {  
 this.data = data;  
 this.next = null;  
 }  
}  
  
// Custom Circular Linked List class  
class CircularLinkedList {  
 Node1 head = null;  
 Node1 tail = null;  
  
 // Method to add a node to the list  
 public void add(int data) {  
 Node1 newNode = new Node1(data);  
  
 if (head == null) {  
 head = newNode;  
 tail = newNode;  
 newNode.next = head; // Points to itself (circular)  
 } else {  
 tail.next = newNode; // Link new node after tail  
 tail = newNode; // Update tail  
 tail.next = head; // Make it circular  
 }  
 }  
  
 // Method to display the list  
 public void display() {  
 if (head == null) {  
 System.*out*.println("List is empty.");  
 return;  
 }  
  
 Node1 current = head;  
 System.*out*.print("Circular Linked List: ");  
 do {  
 System.*out*.print(current.data + " -> ");  
 current = current.next;  
 } while (current != head);  
 System.*out*.println("(back to head)");  
 }  
}  
  
// Main class to run the example  
public class circular {  
 public static void main(String[] args) {  
 CircularLinkedList cll = new CircularLinkedList();  
  
 cll.add(10);  
 cll.add(20);  
 cll.add(30);  
 cll.add(40);  
  
 cll.display();  
 }  
}

// Node class for Circular Linked List  
class Node1 {  
 int data;  
 Node1 next;  
  
 Node1(int data) {  
 this.data = data;  
 this.next = null;  
 }  
}  
  
// Custom Circular Linked List class  
class CircularLinkedList {  
 Node1 head = null;  
 Node1 tail = null;  
  
 // Method to add a node to the list  
 public void add(int data) {  
 Node1 newNode = new Node1(data);  
  
 if (head == null) {  
 head = newNode;  
 tail = newNode;  
 newNode.next = head; // Points to itself (circular)  
 } else {  
 tail.next = newNode; // Link new node after tail  
 tail = newNode; // Update tail  
 tail.next = head; // Make it circular  
 }  
 }  
  
 // Method to display the list  
 public void display() {  
 if (head == null) {  
 System.*out*.println("List is empty.");  
 return;  
 }  
  
 Node1 current = head;  
 System.*out*.print("Circular Linked List: ");  
 do {  
 System.*out*.print(current.data + " -> ");  
 current = current.next;  
 } while (current != head);  
 System.*out*.println("(back to head)");  
 }  
}  
  
// Main class to run the example  
public class circular {  
 public static void main(String[] args) {  
 CircularLinkedList cll = new CircularLinkedList();  
  
 cll.add(10);  
 cll.add(20);  
 cll.add(30);  
 cll.add(40);  
  
 cll.display();  
 }  
}

Output:

Circular Linked List: 10 -> 20 -> 30 -> 40 -> (back to head)

Process finished with exit code 0

Task 4:

Delete a node in the circular linked list

Code:

package July4;

// Node class

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

}

}

// Circular Linked List class

public class CircularLinkedList {

Node head = null;

Node tail = null;

// Append a node to the end

public void append(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

tail = newNode;

tail.next = head; // Circular link

} else {

tail.next = newNode;

tail = newNode;

tail.next = head;

}

}

// Delete a node by value

public void delete(int key) {

if (head == null) {

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

return;

}

Node current = head, prev = tail;

// If the node to be deleted is the only node

if (head == tail && head.data == key) {

head = null;

tail = null;

System.out.println("Deleted node: " + key);

return;

}

// Deleting the head node

if (head.data == key) {

head = head.next;

tail.next = head;

System.out.println("Deleted node: " + key);

return;

}

// Deleting other nodes

do {

if (current.data == key) {

prev.next = current.next;

// If tail is to be deleted

if (current == tail) {

tail = prev;

}

System.out.println("Deleted node: " + key);

return;

}

prev = current;

current = current.next;

} while (current != head);

System.out.println("Node not found: " + key);

}

// Traverse the circular list

public void traverse() {

if (head == null) {

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

return;

}

Node current = head;

do {

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

current = current.next;

} while (current != head);

System.out.println("(back to head)");

}

// Main method to test the functionality

public static void main(String[] args) {

CircularLinkedList list = new CircularLinkedList();

list.append(10);

list.append(20);

list.append(30);

list.append(40);

System.out.println("Original list:");

list.traverse();

list.delete(10); // Delete head

list.traverse();

list.delete(30); // Delete middle

list.traverse();

list.delete(40); // Delete tail

list.traverse();

list.delete(99); // Try to delete non-existent

list.traverse();

list.delete(20); // Delete last remaining

list.traverse();

}

}

Output:

Original list:

10 -> 20 -> 30 -> 40 -> (back to head)

Deleted node: 10

20 -> 30 -> 40 -> (back to head)

Deleted node: 30

20 -> 40 -> (back to head)

Deleted node: 40

20 -> (back to head)

Node not found: 99

20 -> (back to head)

Deleted node: 20

List is empty.

Task 5:

Create  a  stack and pop the element also print the popped element.

Code:

package July4;  
  
import java.util.Stack;  
  
public class StackExample {  
 public static void main(String[] args) {  
  
 Stack<Integer> stack = new Stack<>();  
  
  
 stack.push(10);  
 stack.push(20);  
 stack.push(30);  
  
 System.*out*.println("Stack after pushes: " + stack);  
  
  
 int popped = stack.pop();  
 System.*out*.println("Popped element: " + popped);  
  
  
 System.*out*.println("Final stack: " + stack);  
 }  
}

output:

Stack after pushes: [10, 20, 30]

Popped element: 30

Final stack: [10, 20]

Process finished with exit code 0

Task 6:

Find an element in the stack and display the position

package July4;  
  
import java.util.Stack;  
  
public class StackSearchExample {  
 public static void main(String[] args) {  
 Stack<Integer> stack = new Stack<>();  
  
 // Push elements  
 stack.push(100);  
 stack.push(200);  
 stack.push(300);  
 stack.push(400);  
  
 System.*out*.println("Stack: " + stack);  
  
 // Search for an element  
 int elementToFind = 300;  
 int position = stack.search(elementToFind); // Position from top (1-based)  
  
 if (position != -1) {  
 System.*out*.println("Element " + elementToFind + " found at position (from top): " + position);  
 } else {  
 System.*out*.println("Element " + elementToFind + " not found in the stack.");  
 }  
 }  
}

output:

Stack: [100, 200, 300, 400]

Element 300 found at position (from top): 2

Process finished with exit code 0

Task 7:

Peek the element and print it ..

package July4;  
  
import java.util.Stack;  
  
public class Task7\_PeekStack {  
 public static void main(String[] args) {  
 Stack<Integer> stack = new Stack<>();  
  
 // Push elements  
 stack.push(10);  
 stack.push(20);  
 stack.push(30);  
  
 // Check if stack is not empty before peeking  
 if (!stack.isEmpty()) {  
 int topElement = stack.peek(); // Peek returns the top without removing  
 System.*out*.println("Top element (peek): " + topElement);  
 } else {  
 System.*out*.println("The stack is empty.");  
 }  
 }  
}

output:

Top element (peek): 30

Process finished with exit code 0

Task 8:

Check if the stack is empty or not?

import java.util.Stack;  
  
public class Task8\_CheckIfEmpty {  
 public static void main(String[] args) {  
 Stack<Integer> stack = new Stack<>();  
  
 // Check if stack is empty  
 if (stack.isEmpty()) {  
 System.*out*.println("The stack is empty.");  
 } else {  
 System.*out*.println("The stack is NOT empty.");  
 }  
  
 // Push an element  
 stack.push(100);  
  
 // Check again  
 if (stack.isEmpty()) {  
 System.*out*.println("The stack is empty.");  
 } else {  
 System.*out*.println("The stack is NOT empty.");  
 }  
 }  
}

output:

The stack is empty.

The stack is NOT empty.

Process finished with exit code 0

Task 9:

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

| **Method** | **Description** |
| --- | --- |
| push(E item) | Adds an element to the top of the stack. |
| pop() | Removes and returns the top element of the stack. |
| peek() | Returns the top element without removing it. |
| isEmpty() | Returns true if the stack contains no elements. |
| search(Object o) | Returns 1-based position of the element from the top, or -1 if not found. |
| size() | Returns the number of elements in the stack. |
| clear() | Removes all elements from the stack. |
| contains(Object o) | Returns true if the stack contains the specified element. |
| get(int index) | Returns the element at the specified position (inherited from Vector). |
| set(int index, E element) | Replaces the element at the specified index (also from Vector). |
| iterator() | Returns an iterator over the stack elements (LIFO order). |
| elementAt(int index) | Returns the element at the specified index (from Vector). |
| firstElement() | Returns the first element in the stack (bottom-most). |
| lastElement() | Returns the last element in the stack (top-most). |
| remove(int index) | Removes the element at the specified index (from Vector). |

Task 10:

Wap to create  a queue with custom methods

* Is empty ()
* Is full()
* Enque
* Deque
* Peek
* display()