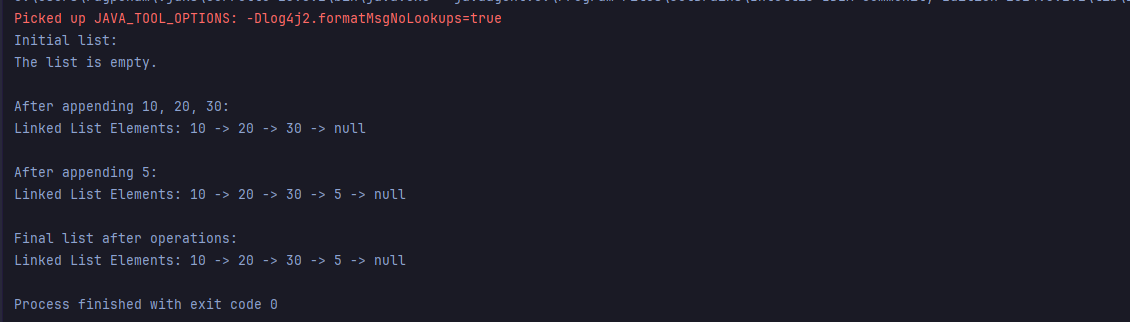
Task1:

class Node {  
 int data;  
 Node next;  
  
 public Node(int data) {  
 this.data = data;  
 this.next = null;  
 }  
}  
  
class LinkedList {  
 Node head;  
  
 public LinkedList() {  
 this.head = null;  
 }  
  
 public void append(int data) {  
 Node newNode = new Node(data);  
 if (this.head == null) {  
 this.head = newNode;  
 return;  
 }  
  
 Node lastNode = this.head;  
 while (lastNode.next != null) {  
 lastNode = lastNode.next;  
 }  
 lastNode.next = newNode;  
 }  
  
 public void traverse() {  
 Node current = this.head;  
 if (current == null) {  
 System.*out*.println("The list is empty.");  
 return;  
 }  
  
 System.*out*.print("Linked List Elements: ");  
 while (current != null) {  
 System.*out*.print(current.data + " -> ");  
 current = current.next;  
 }  
 System.*out*.println("null");  
 }  
  
 public static void main(String[] args) {  
 LinkedList myList = new LinkedList();  
 System.*out*.println("Initial list:");  
 myList.traverse();  
  
 myList.append(10);  
 myList.append(20);  
 myList.append(30);  
 System.*out*.println("**\n**After appending 10, 20, 30:");  
 myList.traverse();  
  
 myList.append(5);  
 System.*out*.println("**\n**After appending 5:");  
 myList.traverse();  
  
 System.*out*.println("**\n**Final list after operations:");  
 myList.traverse();  
 }  
}



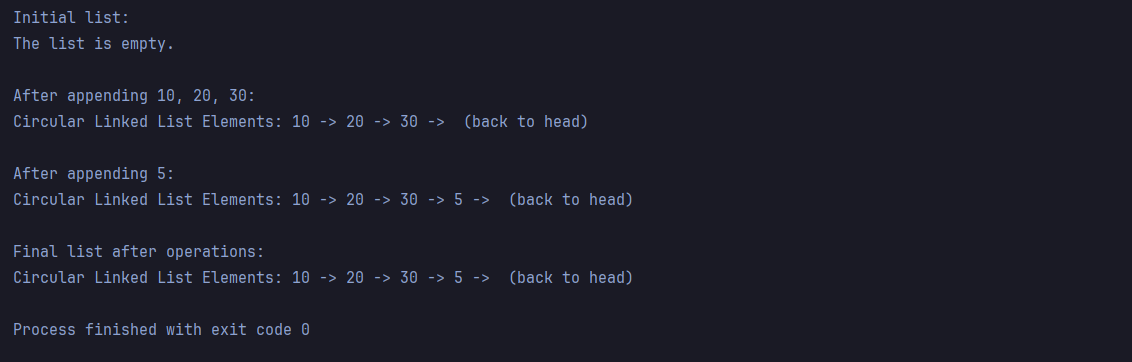
Task2:

Traversing means visiting each node in the list from the beginning till end or till we reach the required element.

Printing an element is optional and can be integrated in the traverse method if needed.

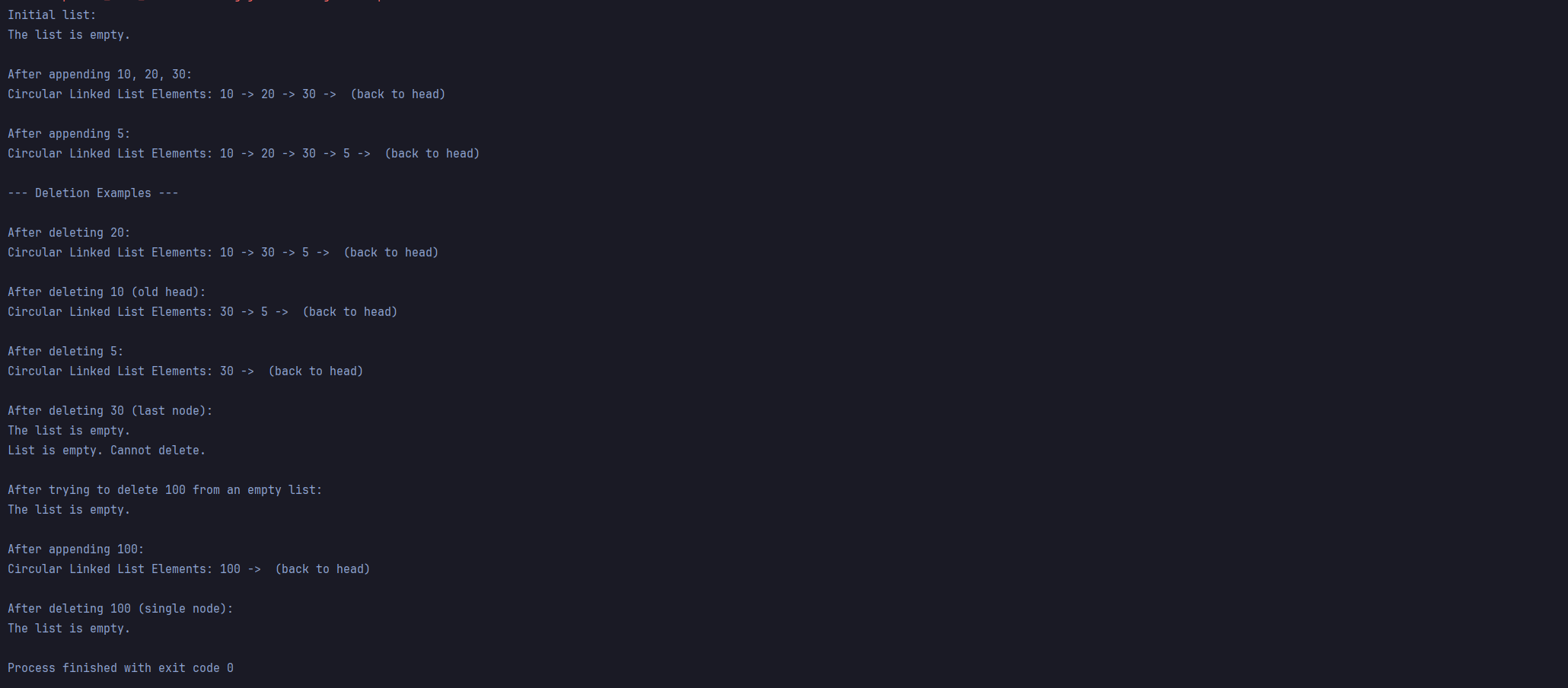
Task3:

class Node {  
 int data;  
 Node next;  
  
 public Node(int data) {  
 this.data = data;  
 this.next = null; // In a circular list, this will be updated  
 }  
}  
  
class LinkedList {  
 Node head;  
  
 public LinkedList() {  
 this.head = null;  
 }  
  
 public void append(int data) {  
 Node newNode = new Node(data);  
 if (this.head == null) {  
 this.head = newNode;  
 newNode.next = this.head; // For a single node, it points to itself  
 return;  
 }  
  
 Node current = this.head;  
 // Traverse to the last node (the one whose next points to head)  
 while (current.next != this.head) {  
 current = current.next;  
 }  
  
 current.next = newNode; // The old last node now points to the new node  
 newNode.next = this.head; // The new node points back to the head  
 }  
  
 public void traverse() {  
 if (this.head == null) {  
 System.*out*.println("The list is empty.");  
 return;  
 }  
  
 Node current = this.head;  
 System.*out*.print("Circular Linked List Elements: ");  
 do {  
 System.*out*.print(current.data + " -> ");  
 current = current.next;  
 } while (current != this.head); // Continue until we loop back to the head  
 System.*out*.println(" (back to head)"); // Indicate the circular nature  
 }  
  
 public static void main(String[] args) {  
 LinkedList myList = new LinkedList();  
 System.*out*.println("Initial list:");  
 myList.traverse();  
  
 myList.append(10);  
 myList.append(20);  
 myList.append(30);  
 System.*out*.println("**\n**After appending 10, 20, 30:");  
 myList.traverse();  
  
 myList.append(5);  
 System.*out*.println("**\n**After appending 5:");  
 myList.traverse();  
  
 System.*out*.println("**\n**Final list after operations:");  
 myList.traverse();  
 }  
}



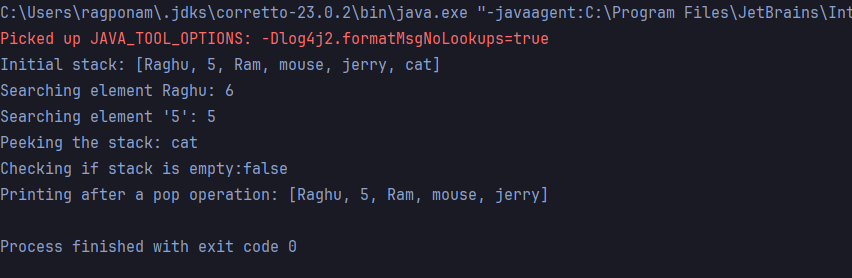
Task4:

import java.util.Queue;  
import java.util.Stack;  
  
class Node {  
 int data;  
 Node next;  
  
 public Node(int data) {  
 this.data = data;  
 this.next = null;  
 }  
}  
  
class LinkedList {  
 Node head;  
  
 public LinkedList() {  
 this.head = null;  
 }  
  
 public void append(int data) {  
 Node newNode = new Node(data);  
 if (this.head == null) {  
 this.head = newNode;  
 newNode.next = this.head;  
 return;  
 }  
  
 Node current = this.head;  
 while (current.next != this.head) {  
 current = current.next;  
 }  
  
 current.next = newNode;  
 newNode.next = this.head;  
 }  
  
 public void deleteNode(int key) {  
 if (this.head == null) {  
 System.*out*.println("List is empty. Cannot delete.");  
 return;  
 }  
  
 Node current = this.head;  
 Node prev = null;  
  
 // Case 1: Deleting the only node in the list  
 if (current.data == key && current.next == this.head) {  
 this.head = null;  
 return;  
 }  
  
 // Case 2: Deleting the head node (when there are multiple nodes)  
 if (current.data == key) {  
 // Find the last node  
 while (current.next != this.head) {  
 current = current.next;  
 }  
 current.next = this.head.next;  
 this.head = this.head.next;  
 return;  
 }  
  
 // Case 3: Deleting a node other than the head  
 prev = current;  
 current = current.next;  
 while (current != this.head && current.data != key) {  
 prev = current;  
 current = current.next;  
 }  
  
 // If the node was found (and it's not the head, which was handled above)  
 if (current != this.head) { // Ensure we didn't loop back to head without finding it  
 prev.next = current.next;  
 } else {  
 System.*out*.println("Node with data '" + key + "' not found.");  
 }  
 }  
  
 public void traverse() {  
 if (this.head == null) {  
 System.*out*.println("The list is empty.");  
 return;  
 }  
  
 Node current = this.head;  
 System.*out*.print("Circular Linked List Elements: ");  
 do {  
 System.*out*.print(current.data + " -> ");  
 current = current.next;  
 } while (current != this.head);  
 System.*out*.println(" (back to head)");  
 }  
  
 public static void main(String[] args) {  
 LinkedList myList = new LinkedList();  
 System.*out*.println("Initial list:");  
 myList.traverse();  
  
 myList.append(10);  
 myList.append(20);  
 myList.append(30);  
 System.*out*.println("**\n**After appending 10, 20, 30:");  
 myList.traverse();  
  
 myList.append(5);  
 System.*out*.println("**\n**After appending 5:");  
 myList.traverse();  
  
 System.*out*.println("**\n**--- Deletion Examples ---");  
  
 myList.deleteNode(20);  
 System.*out*.println("**\n**After deleting 20:");  
 myList.traverse();  
  
 myList.deleteNode(10);  
 System.*out*.println("**\n**After deleting 10 (old head):");  
 myList.traverse();  
  
 myList.deleteNode(5);  
 System.*out*.println("**\n**After deleting 5:");  
 myList.traverse();  
  
 myList.deleteNode(30);  
 System.*out*.println("**\n**After deleting 30 (last node):");  
 myList.traverse();  
  
 myList.deleteNode(100);  
 System.*out*.println("**\n**After trying to delete 100 from an empty list:");  
 myList.traverse();  
  
 myList.append(100);  
 System.*out*.println("**\n**After appending 100:");  
 myList.traverse();  
 myList.deleteNode(100);  
 System.*out*.println("**\n**After deleting 100 (single node):");  
 myList.traverse();  
 }  
}



Task5to8:

import java.util.Stack;  
  
public class Task5to8 {  
 public static void main(String[] args) {  
 Stack stack = new Stack<>();  
 stack.push("Raghu");  
 stack.push(5);  
 stack.push("Ram");  
 stack.push("mouse");  
 stack.push("jerry");  
 stack.push("cat");  
  
  
 System.*out*.println("Initial stack: "+stack);  
 System.*out*.println("Searching element "+"Raghu: "+stack.search("Raghu"));  
 System.*out*.println("Searching element '5': "+stack.search(5));  
 System.*out*.println("Peeking the stack: "+stack.peek());  
 System.*out*.println("Checking if stack is empty:"+stack.isEmpty());  
 stack.pop();  
  
 System.*out*.println("Printing after a pop operation: "+stack);  
  
 }  
}



Task9:

Push(): Add elements at the last position.

Pop(): Remove the last element from the stack.

Peek(): Access the last element from the stack.

isEmpty(): check if the stack is empty, return true if empty, else false.

search(): Search for the position of an element.

Task10:

import **java.util.NoSuchElementException**;  
  
class **Node**<T>{  
 T data;  
 **Node**<T> next;  
 public Node(T data){  
 this.data = data;  
 this.next = null;  
 }  
}  
public class **Task10**<T>{  
 private **Node**<T> front;  
 private **Node**<T> rear;  
 private int size;  
 public Task10(){  
 this.front = null;  
 this.rear=null;  
 this.size=0;  
 }  
 public boolean isEmpty(){  
 return this.front ==null;  
 }  
 public void enqueue(T item) {  
 **Node**<T> newNode = new Node<>(item);  
 if (isEmpty()) {  
 this.front = newNode;  
 }else {  
 this.rear.next = newNode;  
 }  
 this.rear=newNode;  
 this.size++;  
 **System**.*out*.println("Enqueued: "+ item);  
  
 }  
 public T dequeue(){  
 if (isEmpty()){  
 throw new NoSuchElementException("Queue is empty. Cannot dequeue.");  
 }  
 T data = this.front.data;  
 this.front = this.front.next;  
 if (this.front == null){  
 this.rear = null;  
 }  
 this.size--;  
 **System**.*out*.println("Dequeued: "+ data);  
 return data;  
 }  
 public T peek(){  
 if (isEmpty()){  
 throw new NoSuchElementException("Queue is empty. No elements to peek");  
 }  
 return this.front.data;  
 }  
 public void display(){  
 if (isEmpty()){  
 **System**.*out*.println("Queue: [Empty]");  
 return;  
 }  
 **System**.*out*.print("Queue: [");  
 **Node**<T> current = this.front;  
 while (current!=null){  
 **System**.*out*.print(current.data);  
 if (current.next!=null){  
 **System**.*out*.print(", ");  
 }  
 current=current.next;  
 }  
 **System**.*out*.println("]");  
 }  
  
 public static void main(**String**[] args) {  
 **Task10**<**String** > strQueue = new Task10<>();  
 **System**.*out*.println("Initial queue: ");  
 strQueue.display();  
 strQueue.enqueue("Apple");  
 strQueue.enqueue("Ball");  
 strQueue.enqueue("Cat");  
 strQueue.enqueue("Dog");  
 strQueue.display();  
 **System**.*out*.println("Peeking front element: "+ strQueue.peek());  
 strQueue.dequeue();  
 strQueue.dequeue(); // Apple is removed  
 strQueue.display();  
 **System**.*out*.println("Is queue empty? "+ strQueue.isEmpty());  
  
  
 }  
  
 }

