## Topics

1. Implement Node Class
2. Implement CircularlyLinkedList Class
3. Implement Basic Methods of CircularlyLinkedList

* isEmpty()
* size()
* first()
* last()
* addFirst()
* addLast()
* removeFirst()
* rotate()

## Homewor

1. Consider the implementation of CircularlyLinkedList.addFirst, in Code Fragment 3.16. The else body at lines 39 and 40 of that method relies on a locally declared variable, newest. Redesign that clause to avoid use of any local variable.

public void addFirst(T data) {  
 if (isEmpty()) {  
 tail = new Node<>(data);  
 tail.next = tail;  
 tail.prev = tail;  
 head = tail;  
 } else {Node<T> newNode = new Node<>(data);  
 newNode.next = head;  
 head.prev = newNode;  
 newNode.prev = tail;  
 tail.next = newNode;  
 head = newNode;  
 }  
 }

1. Give an implementation of the size( ) method for the CircularlyLinkedList class, assuming that we did not maintain size as an instance variable.

public class CircularlyLinkedList<T> {  
  
 private Node<T> head;  
  
 public int size() {  
 if (isEmpty()) {  
 return 0;  
 }  
  
 Node<T> current = head;  
 int count = 1; do {  
 current = current.next;  
 count++;  
 } while (current != head);  
  
 return count;  
 }  
private class Node<T> {  
 T data;  
 Node<T> next;  
  
 public Node(T data) {  
 this.data = data;  
 }  
 }  
}

3-I

3-

3-mplement the equals( ) method for the CircularlyLinkedList class, assuming that two lists are equal if they have the same sequence of elements, with corresponding elements currently at the front of the list.

public class CircularlyLinkedList<T> {  
  
 private Node<T> head;  
  
 @Override  
 public boolean equals(Object obj) {  
 if (obj == this) {  
 return true; }  
 if (!(obj instanceof CircularlyLinkedList)) {  
 return false; }CircularlyLinkedList<T> other = (CircularlyLinkedList<T>) obj;  
if (isEmpty() && other.isEmpty()) {  
 return true;}  
if (size() != other.size()) {  
 return false; }  
Node<T> current1 = head;  
 Node<T> current2 = other.head;  
 do {  
 if (!current1.data.equals(current2.data)) {  
 return false;}  
 current1 = current1.next;  
 current2 = current2.next;  
 } while (current1 != head); return true;}  
private class Node<T> {  
 T data;  
 Node<T> next;  
  
 public Node(T data) {  
 this.data = data;  
 }  
 }  
}

4-Suppose you are given two circularly linked lists, L and M. Describe an algorithm for telling if L and M store the same sequence of elements (but perhaps with different starting points).

public class CircularlyLinkedList<T> {  
  
 private Node<T> head;  
  
 @Override  
 public boolean equals(Object obj) {  
 if (obj == this) {  
 return true; */* }  
 if (!(obj instanceof CircularlyLinkedList)) {  
 return false; }CircularlyLinkedList<T> other = (CircularlyLinkedList<T>) obj;if (isEmpty() && other.isEmpty()) {  
 return true; *// Both empty* }  
if (size() != other.size()) {  
 return false; *// Different sizes* }Node<T> current1 = head;  
 Node<T> current2 = other.head;  
 do {  
 if (!current1.data.equals(current2.data)) {  
 return false; }  
 current1 = current1.next;  
 current2 = current2.next;  
 } while (current1 != head); return true; }  
private class Node<T> {  
 T data;  
 Node<T> next;  
  
 public Node(T data) {  
 this.data = data;  
 }  
 }  
}

1. Given a circularly linked list L containing an even number of nodes, describe how to split L into two circularly linked lists of half the size.

public static Pair<CircularlyLinkedList<T>, CircularlyLinkedList<T>> splitHalf(CircularlyLinkedList<T> L) {  
 if (L.size() % 2 != 0) {  
 throw new IllegalArgumentException("List must have even number of nodes");  
 }Node<T> slow = L.head;  
 Node<T> fast = L.head;  
 while (fast != L.head || fast.next != L.head) {  
 slow = slow.next;  
 fast = fast.next.next;  
 }Node<T> head1 = slow.next;  
 Node<T> tail1 = slow;  
 tail1.next = head1;Node<T> last = tail1.prev;  
 Node<T> head2 = last.next;  
 last.next = head2;  
 head2.prev = last;  
L.head = head2;  
  
 return new Pair<>(new CircularlyLinkedList<>(head1), new CircularlyLinkedList<>(head2));  
 }

6-Implement the clone( ) method for the CircularlyLinkedList class.

public class CircularlyLinkedList<T> implements Cloneable {  
  
 private Node<T> head;  
  
 @Override  
 public CircularlyLinkedList<T> clone() throws CloneNotSupportedException {  
 if (!super.cloneSupported()) {  
 throw new CloneNotSupportedException("CircularlyLinkedList cannot be cloned");  
 }  
  
 *// Create a new CircularlyLinkedList object* CircularlyLinkedList<T> clone = new CircularlyLinkedList<>();  
  
 if (head != null) {  
 Node<T> current = head;  
 Node<T> prevCloneNode = null;  
 do {Node<T> clonedNode = new Node<>(deepCopy(current.data));  
 if (prevCloneNode != null) {  
 prevCloneNode.next = clonedNode;  
 clonedNode.prev = prevCloneNode;  
 } else {  
 clone.head = clonedNode;  
 }  
 prevCloneNode = clonedNode;  
 current = current.next;  
 } while (current != head);  
 clone.head.prev = prevCloneNode; *// Close the circle in the cloned list* }  
  
 return clone;  
 }  
  
 private T deepCopy(T data) {  
 return data;  
 }  
  
 private class Node<T> {  
 T data;  
 Node<T> next;  
 Node<T> prev;  
  
 public Node(T data) {  
 this.data = data;  
 }  
 }  
}