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

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

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.
2. Give an implementation of the size( ) method for the CircularlyLinkedList class, assuming that we did not maintain size as an instance variable.
3. Implement 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.
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).
5. 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.

Implement the clone( ) method for the CircularlyLinkedList class

1. تنفيذ فئة العقدة (Node):

public class Node<E> {

private E element;

private Node<E> next;

public Node(E element, Node<E> next) {

this.element = element;

this.next = next;

}

public E getElement() {

return element;

}

public void setElement(E element) {

this.element = element;

}

public Node<E> getNext() {

return next;

}

public void setNext(Node<E> next) {

this.next = next;

}

}

تم تنفيذ فئة العقدة (Node) التي تحتوي على عنصر ومرجع إلى العقدة التالية.

1. تنفيذ فئة القائمة المتداورة (CircularlyLinkedList):

public class CircularlyLinkedList<E> {

private Node<E> tail;

private int size;

public CircularlyLinkedList() {

tail = null;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public E first() {

if (isEmpty()) {

return null;

}

return tail.getNext().getElement();

}

public E last() {

if (isEmpty()) {

return null;

}

return tail.getElement();

}

public void addFirst(E element) {

Node<E> newNode = new Node<>(element, null);

if (isEmpty()) {

newNode.setNext(newNode);

tail = newNode;

} else {

newNode.setNext(tail.getNext());

tail.setNext(newNode);

}

size++;

}

public void addLast(E element) {

addFirst(element);

tail = tail.getNext();

}

public E removeFirst() {

if (isEmpty()) {

return null;

}

Node<E> head = tail.getNext();

if (head == tail) {

tail = null;

} else {

tail.setNext(head.getNext());

}

size--;

return head.getElement();

}

public void rotate() {

if (tail != null) {

tail = tail.getNext();

}

}

}

تم تنفيذ فئة القائمة المتداورة (CircularlyLinkedList) التي تحتوي على العديد من العمليات الأساسية للتعامل مع العناصر في القائمة.

1. تنفيذ الطرق الأساسية لفئة CircularlyLinkedList:

* isEmpty(): تحقق مما إذا كانت القائمة المتداورة فارغة أم لا.
* size(): يُرجع حجم القائمة المتداورة.
* first(): يُرجع العنصر الأول في القائمة المتداورة.
* last(): يُرجع العنصر الأخير في القائمة المتداورة.
* addFirst(): يُضيف عنصرًا في بداية القائمة المتداورة.
* addLast(): يُضيف عنصرًا في نهاية القائمة المتداورة.
* removeFirst(): يُزيل العنصر الأول من القائمة المتداورة ويُرجعه.
* rotate(): يقوم بتحويل العناصر في القائمة المتداورة.

1. إعادة تصميم جملة else في طريقة addFirst() لتجنب استخدام أي متغير محلي:

public void addFirst(E element) {

Node<E> newNode = new Node<>(element, null);

if (isEmpty()) {

newNode.setNext(newNode);

} else {

newNode.setNext(tail.getNext());

}

tail.setNext(newNode);

tail = newNode;

size++;

}

تم تعديل جملة else لتعيين العقدة الجديدة كـ tail مباشرة بدلاً من استخدام متغير محلي.

1. تنفيذ طريقة size() لفئة CircularlyLinkedList، مع افتراض عدم الاحتفاظ بحجم القائمة كمتغير الخاص:

public int size() {

if (isEmpty()) {

return 0;

}

int count = 1;

Node<E> current = tail.getNext();

while (current != tail) {

count++;

current = current.getNext();

}

return count;

}

تتم هذه الطريقة عن طريق العد من العقدة التالية لـ tail وحتى نصل إلى tail مرة أخرى.

1. تنفيذ طريقة equals() لفئة CircularlyLinkedList، مع افتراض أن قوائمين متداورتين متساويتين إذا كان لديهما نفس تسلسل العناصر، مع العناصر المقابلة حاليًا في بداية القائمة:

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null || getClass() != obj.getClass()) {

return false;

}

CircularlyLinkedList<?> other = (CircularlyLinkedList<?>) obj;

if (size() != other.size()) {

return false;

}

Node<E> current = tail.getNext();

Node<?> otherCurrent = other.tail.getNext();

while (current != tail) {

if (!current.getElement().equals(otherCurrent.getElement())) {

return false;

}

current = current.getNext();

otherCurrent = otherCurrent.getNext();

}

return tail.getElement().equals(other.tail.getElement());

}

تُقارن العناصر في القوائم المتداورتين بدءًا من العقدة التالية لـ tail وحتى نصل إلى tail مرة أخرى.

1. لمعرفة ما إذا كانت قوائم L و M تحتوي على نفس تسلسل العناصر (لكن ربما بنقاط البداية المختلفة)، يمكن استخدام الخوارزمية التالية:

* يتم التحقق من أن حجم قوائم L و M متساوٍ.
* يتم تحويل L بنقاط البداية المختلفة ومقارنة عناصرها بنقاط البداية في M. إذا كانت كل العناصر متطابقة في أحد النقاط، يعتبرون نفس التسلسل.
* إذا لم تتطابق العناصر في أي نقطة، فإنهم لا يحتوون على نفس التسلسل.