## Topics

1. Implement Node Class

public class Node<T> {

private T data;

private Node<T> next;

public Node(T data) {

this.data = data;

this.next = null;

}

public T getData() {

return data;

}

public void setData(T data) {

this.data = data;

}

public Node<T> getNext() {

return next;

}

public void setNext(Node<T> next) {

this.next = next;

}

}

1. Implement CircularlyLinkedList Class

public class CircularlyLinkedList<T> {

private Node<T> head;

private int size;

public CircularlyLinkedList() {

head = null;

size = 0;

}

public int size() {

return size;

}

public boolean isEmpty() {

return size == 0;

}

public void addFirst(T data) {

Node<T> newNode = new Node<>(data);

if (isEmpty()) {

newNode.setNext(newNode);

head = newNode;

} else {

newNode.setNext(head);

Node<T> tail = getNodeAt(size - 1);

tail.setNext(newNode);

head = newNode;

}

size++;

}

public void addLast(T data) {

if (isEmpty()) {

addFirst(data);

} else {

Node<T> newNode = new Node<>(data);

Node<T> tail = getNodeAt(size - 1);

newNode.setNext(head);

tail.setNext(newNode);

size++;

}

}

public T removeFirst() {

if (isEmpty()) {

throw new NoSuchElementException("The CircularlyLinkedList is empty");

}

T removedData = head.getData();

if (size == 1) {

head = null;

} else {

Node<T> tail = getNodeAt(size - 1);

head = head.getNext();

tail.setNext(head);

}

size--;

return removedData;

}

public T getFirst() {

if (isEmpty()) {

throw new NoSuchElementException("The CircularlyLinkedList is empty");

}

return head.getData();

}

public T getLast() {

if (isEmpty()) {

throw new NoSuchElementException("The CircularlyLinkedList is empty");

}

Node<T> tail = getNodeAt(size - 1);

return tail.getData();

}

private Node<T> getNodeAt(int index) {

if (index < 0 || index >= size) {

throw new IndexOutOfBoundsException("Index is out of bounds");

}

Node<T> current = head;

for (int i = 0; i < index; i++) {

current = current.getNext();

}

return current;

}

}

1. Implement Basic Methods of CircularlyLinkedList

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

import java.util.NoSuchElementException;

public class CircularlyLinkedList<T> {

private Node<T> head;

private int size;

public CircularlyLinkedList() {

head = null;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public T first() {

if (isEmpty()) {

throw new NoSuchElementException("The CircularlyLinkedList is empty");

}

return head.getData();

}

public T last() {

if (isEmpty()) {

throw new NoSuchElementException("The CircularlyLinkedList is empty");

}

return head.getNext().getData();

}

public void addFirst(T data) {

Node<T> newNode = new Node<>(data);

if (isEmpty()) {

newNode.setNext(newNode);

head = newNode;

} else {

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

newNode.setNext(tail);

head.setNext(newNode);

head = newNode;

}

size++;

}

public void addLast(T data) {

addFirst(data);

head = head.getNext();

}

public T removeFirst() {

if (isEmpty()) {

throw new NoSuchElementException("The CircularlyLinkedList is empty");

}

T removedData = head.getData();

if (size == 1) {

head = null;

} else {

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

head.setNext(tail.getNext());

tail.setNext(null);

head = head.getNext();

}

size--;

return removedData;

}

}

## Homework

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) {

Node<T> newNode = new Node<>(data);

if (isEmpty()) {

newNode.setNext(newNode);

head = newNode;

} else {

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

head.setNext(newNode);

newNode.setNext(tail);

head = newNode;

}

size++;

}

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

public int size() {

if (isEmpty()) {

return 0;

}

int count = 1;

Node<T> current = head.getNext();

while (current != head) {

count++;

current = current.getNext();

}

return count;

}

1. 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.

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

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

return false;

}

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

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

return false;

}

Node<T> currentThis = head;

Node<?> currentOther = otherList.head;

for (int i = 0; i < size(); i++) {

if (!currentThis.getData().equals(currentOther.getData())) {

return false;

}

currentThis = currentThis.getNext();

currentOther = currentOther.getNext();

}

return true;

}

1. 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).

تحقق مما إذا كانت القائمتان L وM فارغتين. إذا كان كلاهما فارغًا، فارجع صحيحًا لأنهما يخزنان نفس التسلسل الفارغ.

قم باجتياز إحدى القوائم، L، ولكل عقدة، قارن بياناتها مع العقدة المقابلة لها في القائمة الأخرى، M.

ابدأ بالعقدة الرأسية لـ L.

ابدأ بأي عقدة في M.

إذا كانت بيانات العقد الحالية في L وM متساوية، انتقل إلى العقد التالية.

إذا كانت بيانات أي زوج من العقد في L وM غير متساوية، فتابع إلى الخطوة 3.

تتبع عدد العقد التي تمت زيارتها في L. إذا أصبح عدد العقد التي تمت زيارتها في L مساويًا لحجم L، فسيتم إرجاع صحيح حيث تم فحص التسلسل بأكمله.

إذا كان عدد العقد التي تمت زيارتها في L أقل من حجم L ووصلت إلى نهاية M، فسيتم إرجاع خطأ لأن التسلسل في M قد انتهى قبل L.

إذا وصلت إلى هذه الخطوة، فهذا يعني أن التسلسلات ليست متساوية مع نقاط البداية الحالية. للتعامل مع احتمال وجود نقاط بداية مختلفة، كرر الخطوة 2 مع نقاط بداية مختلفة في M حتى يتم التحقق من جميع نقاط البداية.

ابدأ بالعقدة التالية في M من نقطة البداية السابقة.

كرر الخطوة 2 مع نقطة البداية الجديدة في M.

إذا أعادت الخطوة 2 صحيحًا، فارجع صحيحًا لأن التسلسلات متساوية مع نقطة بداية مختلفة.

إذا قامت الخطوة 2 بإرجاع خطأ لجميع نقاط البداية في M، فسيتم إرجاع خطأ نظرًا لعدم ظهور أي من نقاط البداية في تسلسل متساوٍ.

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.

ابحث عن النقطة الوسطى للقائمة المرتبطة دائريًا L. للقيام بذلك، يمكنك استخدام تقنية "المؤشر البطيء والسريع". ابدأ بمؤشرين، بطيء وسريع، وكلاهما يشير في البداية إلى العقدة الرئيسية لـ L. حرك ببطء خطوة واحدة في كل مرة وبسرعة خطوتين في كل مرة حتى يصل سريع إلى نهاية القائمة (على سبيل المثال، fast.getNext() =) = الرأس).

في نهاية هذه الخطوة، سيشير المؤشر البطيء إلى منتصف حرف L.

قم بتعيين مؤشر الرأس للقائمة المرتبطة دائريًا الأولى، على سبيل المثال L1، إلى الرأس الحالي لـ L.

L1.head = L.head

قم بتعيين مؤشر الرأس للقائمة المرتبطة دائريًا الثانية، على سبيل المثال L2، إلى العقدة البطيئة التالية.

L2.head = بطيء.getNext()

اجعل العقدة الأخيرة من L2 تشير إلى رأس L2 لإكمال الارتباط الدائري.

بطيئة.setNext(L2.head)

اجعل العقدة الأخيرة من L تشير إلى رأس L1 لإكمال الارتباط الدائري لـ L1.

L.lastNode.setNext(L1.head)

1. Implement the clone( ) method for the CircularlyLinkedList class.

public CircularlyLinkedList<T> clone() {

CircularlyLinkedList<T> clonedList = new CircularlyLinkedList<>();

if (isEmpty()) {

return clonedList;

}

Node<T> current = head;

do {

clonedList.addLast(current.getData());

current = current.getNext();

} while (current != head);

return clonedList;

}