## Topics

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

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

## 

class Node<E> {

private E element;

private Node<E> next;

public Node(E e, Node<E> n) {

element = e;

next = n;

}

public E getElement() { return element; }

public Node<E> getNext() { return next; }

public void setNext(Node<E> n) { next = n; }

}

class CircularlyLinkedList<E> {

private Node<E> tail = null;

private int size = 0;

public boolean isEmpty() { return size == 0; }

public int size() {

Node<E> temp = tail;

int size = 0;

do {

size++;

temp = temp.getNext();

} while (temp != tail);

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 e) {

if (size == 0) {

tail = new Node<>(e, null);

tail.setNext(tail);

} else {

Node<E> newest = new Node<>(e, tail.getNext());

tail.setNext(newest);

}

size++;

}

public void addLast(E e) {

addFirst(e);

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();

}

// Rotate the first element to the back

public void rotate() {

if (tail != null)

tail = tail.getNext();

}

}

## 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.

class Node<E> {

private E element;

private Node<E> next;

public Node(E e, Node<E> n) {

element = e;

next = n;

}

public E getElement() { return element; }

public Node<E> getNext() { return next; }

public void setNext(Node<E> n) { next = n; }

}

class CircularlyLinkedList<E> {

private Node<E> tail = null;

private int size = 0;

public boolean isEmpty() { return size == 0; }

public int size() {

Node<E> temp = tail;

int size = 0;

do {

size++;

temp = temp.getNext();

} while (temp != tail);

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 e) {

if (size == 0) {

tail = new Node<>(e, null);

tail.setNext(tail);

} else {

tail.setNext(new Node<>(e, tail.getNext()));

}

size++;

}

// Other methods...

}

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

class Node<E> {

private E element;

private Node<E> next;

public Node(E e, Node<E> n) {

element = e;

next = n;

}

public E getElement() { return element; }

public Node<E> getNext() { return next; }

public void setNext(Node<E> n) { next = n; }

}

class CircularlyLinkedList<E> {

private Node<E> tail = null;

public boolean isEmpty() { return tail == null; }

public int size() {

if (isEmpty()) return 0;

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

int size = 0;

do {

size++;

temp = temp.getNext();

} while (temp != tail.getNext());

return size;

}

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.

class Node<E> {

private E element;

private Node<E> next;

public Node(E e, Node<E> n) {

element = e;

next = n;

}

public E getElement() { return element; }

public Node<E> getNext() { return next; }

public void setNext(Node<E> n) { next = n; }

}

class CircularlyLinkedList<E> {

private Node<E> tail = null;

public boolean isEmpty() { return tail == null; }

public int size() {

if (isEmpty()) return 0;

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

int size = 0;

do {

size++;

temp = temp.getNext();

} while (temp != tail.getNext());

return size;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

CircularlyLinkedList<?> that = (CircularlyLinkedList<?>) o;

if (this.size() != that.size()) return false;

Node<E> current1 = this.tail.getNext();

Node<?> current2 = that.tail.getNext();

do {

if (!current1.getElement().equals(current2.getElement())) return false;

current1 = current1.getNext();

current2 = current2.getNext();

} while (current1 != this.tail.getNext());

return true;

}

// Other methods...

}

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).

public boolean compareLists(Node head1, Node head2) {

Node temp1 = head1;

Node temp2 = head2;

while (true) {

// If both lists are at end

if (temp1.next == head1 && temp2.next == head2)

return true;

// If one list is at end and other is not

if (temp1.next == head1 || temp2.next == head2)

return false;

// If both lists are not at end, compare data

if (temp1.data != temp2.data)

return false;

// Move both nodes to next

temp1 = temp1.next;

temp2 = temp2.next;

}

}

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.

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

public class LinkedList {

Node head, head1, head2;

void splitList() {

Node slow\_ptr = head;

Node fast\_ptr = head;

if (head == null) {

return;

}

/\* If there are odd nodes in the circular linked list then

fast\_ptr->next becomes head and for even nodes

fast\_ptr->next->next becomes head \*/

while (fast\_ptr.next != head && fast\_ptr.next.next != head) {

fast\_ptr = fast\_ptr.next.next;

slow\_ptr = slow\_ptr.next;

}

/\* If there are even elements in list then move fast\_ptr \*/

if (fast\_ptr.next.next == head) {

fast\_ptr = fast\_ptr.next;

}

/\* Set the head pointer of first half \*/

head1 = head;

/\* Set the head pointer of second half \*/

if (head.next != head) {

head2 = slow\_ptr.next;

}

/\* Make second half circular \*/

fast\_ptr.next = slow\_ptr.next;

/\* Make first half circular \*/

slow\_ptr.next = head;

}

}

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

public class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

public class CircularLinkedList {

Node head;

// This function returns clone of a given circular

// linked list

public CircularLinkedList clone() {

Node origCurr = this.head, cloneCurr = null;

// Initialize two references, one with original

// list's head.

Node origNext, cloneNext;

// Hash map which contains node to node mapping of

// original and clone linked list.

HashMap<Node, Node> map = new HashMap<Node, Node>();

// Traverse the original list and make a copy of that

// in the clone linked list.

while (origCurr != null) {

cloneCurr = new Node(origCurr.data);

map.put(origCurr, cloneCurr);

origCurr = origCurr.next;

}

// Adjusting the original list reference again.

origCurr = this.head;

// Traversal of original list again to adjust the next

// and random references of clone list using hash map.

while (origCurr != null) {

cloneCurr = map.get(origCurr);

origNext = origCurr.next;

cloneNext = map.get(origNext);

cloneCurr.next = cloneNext;

origCurr = origNext;

}

// Return the head reference of the clone list.

return new CircularLinkedList(map.get(this.head));

}

}