## Topics

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

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

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

Sure! Here's the complete implementation of the Node class, CircularlyLinkedList class, and the methods you requested:

1. Node Class:

java

Copy

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. CircularlyLinkedList Class:

java

Copy

public class CircularlyLinkedList<T> {

private Node<T> head;

private int size;

public CircularlyLinkedList() {

head = null;

size = 0;

}

// Basic Methods

public boolean isEmpty() {

return size == 0;

}

public int size() {

int count = 0;

Node<T> current = head;

if (current != null) {

do {

count++;

current = current.getNext();

} while (current != head);

}

return count;

}

public T first() {

if (isEmpty()) {

throw new NoSuchElementException("The list is empty.");

}

return head.getData();

}

public T last() {

if (isEmpty()) {

throw new NoSuchElementException("The list 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 {

newNode.setNext(head.getNext());

head.setNext(newNode);

T temp = head.getData();

head.setData(newNode.getData());

newNode.setData(temp);

}

size++;

}

public void addLast(T data) {

addFirst(data);

head = head.getNext();

}

public void removeFirst() {

if (isEmpty()) {

throw new NoSuchElementException("The list is empty.");

}

if (size == 1) {

head = null;

} else {

head.setNext(head.getNext().getNext());

}

size--;

}

public void rotate() {

if (!isEmpty()) {

head = head.getNext();

}

}

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (!(obj instanceof CircularlyLinkedList)) {

return false;

}

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

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

return false;

}

if (isEmpty()) {

return true;

}

Node<T> currentNode = head;

Node<T> otherNode = otherList.head;

do {

if (!currentNode.getData().equals(otherNode.getData())) {

return false;

}

currentNode = currentNode.getNext();

otherNode = otherNode.getNext();

} while (currentNode != head);

return true;

}

public CircularlyLinkedList<T> clone() {

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

if (isEmpty()) {

return newList;

}

Node<T> current = head;

do {

newList.addLast(current.getData());

current = current.getNext();

} while (current != head);

return newList;

}

}

1. Describing the algorithm for checking if two circularly linked lists store the same sequence of elements (but perhaps with different starting points):

To determine if two circularly linked lists L and M store the same sequence of elements (with different starting points), you can perform the following steps:

1. Check if both lists are empty. If they are, return true since they both have the same empty sequence.
2. Iterate through List L starting from the head node, and for each node, check if it matches the corresponding node in List M.
3. If at any point during the iteration, the nodes don't match, break the loop and return false.
4. If the iteration completes without finding any unequal nodes, return true, indicating that both lists have the same sequence of elements.

Here's the implementation of the algorithm as a separate method:

java

Copy

public boolean hasSameSequence(CircularlyLinkedList<T> list1, CircularlyLinkedList<T> list2) {

if (list1.isEmpty() && list2.isEmpty()) {

return true; // Both lists are empty, same sequence

}

Node<T> current1 = list1.head;

Node<T> current2 = list2.head;

do {

if (!current1.getData().equals(current2.getData())) {

return false; // Nodes don't match, different sequence

}

current1 = current1.getNext();

current2 = current2.getNext();

} while (current1 != list1.head && current2 != list2.head);

return true; // All nodes match, same sequence

}

1. Splitting a circularly linked list L into two circularly linked lists of half the size:

To split a circularly linked list L into two circularly linked lists of half the size, you can use the following steps:

1. Check if the list L is empty or has an odd number of nodes. If it is empty or has an odd number of nodes, splitting is not possible.
2. If the list L has an even number of nodes, iterate through the list to find the middle node.
3. Once the middle node is found, set it as the last node of the first half of the list and the first node of the second half of the list.
4. Set the next pointer of the last node in the first half to the head of the first half, creating a circular link.
5. Set the next pointer of the last node in the second half to the head of the second half, creating a circular link.
6. Return the two circularly linked lists as a result.

Here's the implementation of the algorithm as a separate method:

java

Copy

public CircularlyLinkedList<T>[] splitIntoHalves(CircularlyLinkedList<T> list) {

if (list.isEmpty() || list.size() % 2 != 0) {

return null; // Splitting not possible, return null

}

int halfSize = list.size() / 2;

Node<T> current = list.head;

CircularlyLinkedList<T> list1 = new CircularlyLinkedList<>();

CircularlyLinkedList<T> list2 = new CircularlyLinkedList<>();

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

list1.addLast(current.getData());

current = current.getNext();

}

list2.head = current;

list1.head.setNext(list2.head);

list2.head.setNext(list.head);

return new CircularlyLinkedList[]{list1, list2};

}

1. Implementing the clone() method for the CircularlyLinkedList class:

Implementing the clone() method allows you to create a deep copy of the circularly linked list. Here's the implementation:

java

Copy

public CircularlyLinkedList<T> clone() {

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

if (isEmpty()) {

return newList; // Return an empty list

}

Node<T> current = head;

do {

newList.addLast(current.getData());

current = current.getNext();

} while (current != head);

return newList;

}

In this implementation, a new CircularlyLinkedList object newList is created. If the original list is empty, the method returns the new empty list.

If the original list is not empty, it iterates through the original list starting from the head node. It adds each element to the new list using the addLast() method.

Finally, it returns the new list, which is a deep copy of the original circularly linked list.