Circular Linked List

EENG212

Algorithms
and

Data Structures
Circular Linked Lists

- In linear linked lists if a list is traversed (all the elements visited) an external pointer to the list must be preserved in order to be able to reference the list again.
- Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node.
Circular Linked Lists

A Linear Linked List
Circular Linked Lists

A circular linked list without a header node.
Circular Linked Lists

A circular linked list with a header node.
Circular Linked Lists

- In a circular linked list there are two methods to know if a node is the first node or not.
  - Either an external pointer, `list`, points the first node or
  - A header node is placed as the first node of the circular list.
- The header node can be separated from the others by either having a sentinel value as the info part or having a dedicated flag variable to specify if the node is a header node or not.
PRIMITIVE FUNCTIONS IN CIRCULAR LISTS

- The structure definition of the circular linked lists and the linear linked list is the same:

```c
struct node{
    int info;
    struct node *next;
};
typedef struct node *NODEPTR;
```
The delete after and insert after functions of the linear lists and the circular lists are almost the same.

The delete after function: \textit{delafter( )}

\begin{verbatim}
void delafter(NODEPTR p, int *px)
{
    NODEPTR q;
    if((p == NULL) || (p == p->next)){ /*the empty list contains a single node and may be pointing itself*/
        printf("void deletion\n");
        exit(1);
    }
    q = p->next;
    *px = q->info; /*the data of the deleted node*/
    p->next = q->next;
    freenode(q);
}
\end{verbatim}
PRIMITIVE FUNCTIONS IN CIRCULAR LISTS

- The insertafter function: `insafter()`
  ```c
  void insafter(NODEPTR p, int x)
  {
    NODEPTR q;
    if(p == NULL){
      printf("void insertion\n");
      exit(1);
    }
    q = getnode();
    q->info = x; /*the data of the inserted node*/
    q->next = p->next;
    p->next = q;
  }
  ```
CIRCULAR LIST with header node

- The header node in a circular list can be specified by a **sentinel value** or a dedicated **flag**:
- **Header Node with Sentinel**: Assume that info part contains positive integers. Therefore the info part of a header node can be -1. The following circular list is an example for a sentinel used to represent the header node:

```c
struct node{
    int info;
    struct node *next;
};
typedef struct node *NODEPTR;
```
CIRCULAR LIST with header node
CIRCULAR LIST with header node

● Header Node with Flag: In this case a extra variable called flag can be used to represent the header node. For example flag in the header node can be 1, where the flag is 0 for the other nodes.

```c
struct node{
    int flag;
    int info;
    struct node *next;
};
typedef struct node *NODEPTR;
```
CIRCULAR LIST with header node
Example

- Consider a circular linked list with a header node, where each node contains the name, account number and the balance of a bank customer. The header node contains a sentinel account number to be -99.

  (a) Write an appropriate node structure definition for the circular linked list.

  (b) Write a function to display the full records of the customers with negative balance.
a) struct node{
    char Name[15]; int AccNo;
    float Balance;
    struct node *next;
};
typedef struct node *NODEPTR;

b) Assume that the list pointer points the header with the sentinel account number -99.

void DispNegBalanca(NODEPTR *plist)
{
    NODEPTR p;
    p=*plist;
    if(p == NULL){
        printf(“There is no list!\n”);
        exit(1);
    }
    p=p->next;
    while(p->AccNo!=-99){
        if(p->Balance < 0.0)
            printf(“The Customer Name:%s\nThe Account No:%d\nThe Balance:%.2f\n”, p->Name, p->AccNo, p->Balance);
        p=p->next;
    }
}
Example

• Write a function that returns the average of the numbers in a circular list. Assume that the following node structure is used, where the flag variable is 1 for the header node and 0 for all the other nodes.

```c
struct node{
    int flag;
    float info;
    struct node *next;
};
typedef struct node *NODEPTR;
```
float avList(NODEPTR *plist)/*assume that plist points the header node*/
{
    int count=0;
    float sum =0.0;
    NODEPTR p;
    p=*plist;
    if((p == NULL)){
        printf(“Empty list\n”);
        exit(1);
    }
    do{
        sum=sum + p->info;
        p =p->next;
        count++;
    }while(p->flag !=1);
    return sum/count;
}
4.1.2.3 Implementing Doubly-Linked Lists

Although the Java programming language has not many composite data structures built-in, they can easily be implemented. In this section we shall discuss an implementation of *doubly-linked lists*. This example of a self-made container class also shows how to implement an enumeration interface for accessing the elements sequentially. If you wish, you may skip this subsection or come back to it later.

1. Overall Structure of Doubly-Linked Lists
2. Implementing the List Items
3. Implementing the Doubly-Linked List
4. Providing the List Enumeration
5. Demonstrating the Doubly-Linked List Implementation

Overall Structure of Doubly-Linked Lists

If you wish to traverse a list both forwards and backwards efficiently, or if you wish, given a list element, to determine the preceding and following elements quickly, then the *doubly-linked list* comes in handy. A list element contains the data plus pointers to the next and previous list items as shown in the picture below.

![Doubly-Linked List](image)

Of course we need a pointer to some *link* in the doubly-linked list to access list elements. It is convenient for doubly-linked lists to use a *list header, or head*, that has the same structure as any other list item and is actually part of the list data structure. The picture below shows an empty and nonempty doubly-linked list. By following arrows it is easy to go from the list header to the first and last list element, respectively.
Insertion and removal of an element in a doubly-linked list is in this implementation rather easy. In the picture below we illustrate the pointer changes for a removal of a list item (old pointers have been drawn solid and new pointers are dashed arrows). We first locate the previous list item using the *previous* field. We make the *next* field of this list item point to the item following the one in cursor position *pos*. Then we make the *previous* field of this following item point to the item preceding the one in the cursor position *pos*. The list item pointed to by the cursor becomes useless and should be automatically garbage collected.

In a pseudo-programming language we could write the following code:

```pseudo
remove(cursor pos)
begin
    if empty list then
        ERROR("empty list.")
    else if cursor points at list header then
        ERROR("cannot remove the list header")
    else
        pos previous next = pos next;
        pos next previous = pos previous;
    endif
```
Implementing the List Items

The basic `ListItem` class is easily defined as follows

```
final class ListItem {
    Object obj;
    ListItem previous, next;

    public ListItem(Object obj) {
        this(null, obj, null);
    }

    public ListItem(ListItem previous, Object obj, ListItem next) {
        this.previous = previous;
        this.obj = obj;
        this.next = next;
    }
}
```

A class definition with only two constructor methods. The keyword `final` ensures that this class has no subclasses nor that a user can derive a class from this one.

Implementing the Doubly-Linked List

The objects of type `ListItem` are placed in a doubly-linked list. For this we have to define a class `LinkedList`. It contains a special `ListItem`, called the head, to give the user a principal handle to the list items. The insertion methods (`insertBefore`, `insertAfter`) and the removal method (`remove`) in the `LinkedList` class need a pointer to the position in the list at which action should take place.

For this purpose we introduce a `ListIterator` class. A `ListIterator` contains the `LinkedList` to which it belongs and a pointer to the `ListItem` that is currently selected. The `ListIterator` class contains methods to move the cursor position. The `LinkedList` class contains the methods

- `Head`, which returns an iterator that points to the head of the linked list;
- `find`, which searches in the linked list for a requested object and, if found, returns an iterator pointing to the object

To step through all current list items, we added a procedure `elements` in the `LinkedList` class; it returns an `Enumeration` object.

Most of the Java code for the two classes below should speak for itself.

```
final class ListIterator {
    LinkedList owner;
    ListItem pos;

    ListIterator(LinkedList owner, ListItem pos) {
        this.owner = owner;
        this.pos = pos;
    }

    /*
     * check whether object owns the iterator
     */
public boolean belongsTo(Object owner) {
    return this.owner == owner;
}

/*
 * move to head position
 */
public void head() {
    pos = owner.head;
}

/*
 * move to next position
 */
public void next() {
    pos = pos.next;
}

/*
 * move to previous position
 */
public void previous() {
    pos = pos.previous;
}

LinkedList.java

import java.util.*;

public class LinkedList {
    ListItem head;
    /*
     * creates an empty list
     */
    public LinkedList() {
        head = new ListItem(null);
        head.next = head.previous = head;
    }

    /*
     * remove all elements in the list
     */
    public final synchronized void clear() {
        head.next = head.previous = head;
    }

    /*
     * returns true if this container is empty.
     */
    public final boolean isEmpty() {
        return head.next == head;
    }

    /*
     * insert element after current position
     */
    public final synchronized void insertAfter(Object obj, ListIterator cursor) {
        ListItem newItem = new ListItem(cursor.pos, obj, cursor.pos.next);
        newItem.next.previous = newItem;
        cursor.pos.next = newItem;
    }

    /*
     * insert element before current position
     */
public final synchronized void insertBefore(Object obj, ListIterator cursor) {
    ListItem newItem = new ListItem(cursor.pos.previous, obj, cursor.pos);
    newItem.previous.next = newItem;
    cursor.pos.previous = newItem;
}

/*
 * remove the element at current position
 */
public final synchronized void remove(ListIterator cursor) {
    if (isEmpty()) {
        throw new IndexOutOfBoundsException("empty list.");
    }
    if (cursor.pos == head) {
        throw new NoSuchElementException("cannot remove the head");
    }
    cursor.pos.previous.next = cursor.pos.next;
    cursor.pos.next.previous = cursor.pos.previous;
}

/*
 * Return an iterator positioned at the head.
 */
public final ListIterator head() {
    return new ListIterator(this, head);
}

/*
 * find the first occurrence of the object in a list
 */
public final synchronized ListIterator find(Object obj) {
    if (isEmpty()) {
        throw new IndexOutOfBoundsException("empty list.");
    }
    ListItem pos = head;
    while (pos.next != head) { // There are still elements to be inspected
        pos = pos.next;
        if (pos.obj == obj) {
            return new ListIterator(this, pos);
        }
    }
    throw new NoSuchElementException("no such object found");
}

/*
 * Returns an enumeration of the elements. Use the Enumeration methods on
 * the returned object to fetch the elements sequentially.
 */
public final synchronized Enumeration elements() {
    return new ListEnumerator(this);
}

The synchronized keyword in many of the above methods indicates that the methods that have this modifier change the internal state of a LinkedList object which is not "thread-safe": for example, insertions and removals of list elements should not be carried out in random order, but in the order in which they are requested. The synchronized keyword takes care of this: when you call a synchronized instance method, Java first locks the instance so that no other threads can modify the object concurrently. See the section on interaction between threads for more details on synchronization.

Providing the List Enumeration

Java provides the Enumeration interface to step through the elements in an instance of a built-in container class such as Vector. You can look up the source code of the Enumeration interface in the development kit; it looks as follows:
public interface Enumeration {
    boolean hasMoreElements();
    Object nextElement();
}

So, these are the two methods that you can use to iterate through the set of values. Two examples of their use.

- Printing the class names of all applets in a Web page:

```java
Enumeration e = getAppletContext().getApplets(); // get all applets
while (e.hasMoreElements()) { // step through all applets
    Object applet = e.nextElement();
    System.out.println(applet.getClass().getName());
}
```

- Working with vectors:

```java
Vector vector = new Vector(); // declaration of vector
vector.addElement(new ...); // with addElement you can add items
Enumeration e = vector.elements(); // get all vector elements
while (e.hasMoreElements()) { // step through all vector elements
    Object obj = e.nextElement();
    // work with the object .....
}
```

Of course we want to offer this enumeration facility for doubly-linked lists as well. The `elements` method of the `LinkedList` class already delivers an `Enumeration`, actually an instance of the `ListEnumerator` class.

The skeleton of this class is as follows:

```java
import java.util.*;

final class ListEnumerator implements Enumeration {
    LinkedList list;
    ListIterator cursor;

    ListEnumerator(LinkedList l) {
    }

    public boolean hasMoreElements() {
    }

    public Object nextElement() {
    }
}
```

The enumerator contains

- a `LinkedList` instance variable that points to the doubly-linked list we want to enumerate;
- a `ListItem` instance variable that serves as a cursor to the doubly-linked list under consideration.

You may wonder why another cursor to step through the data structure, but when you traverse a linked list, maybe a cursor in use is at an important position before traversal and should stay there after traversal.

The constructor `ListEnumerator(LinkedList l)` generates the enumeration from a given doubly-linked list: it positions the cursor at the list header and move it one position further. In Java code:

```java
ListEnumerator(LinkedList l) {
    list = l;
    cursor = list.head();
}
What remains to be done is the implementation of the two promised methods:

- **hasMoreElements**: simply check whether the cursor presently points at the list header.

  ```java
  public boolean hasMoreElements() {
      return cursor.pos != list.head;
  }
  ```

- **nextElement**: return the data of the list item presently pointed at and move the cursor one step further. All this should happen unless the cursor is already pointing at the list header, in which case an exception is thrown. (See the section on exceptions for details about this topic)

  ```java
  public Object nextElement() {
      synchronized (list) {
          if (cursor.pos != list.head) {
              Object object = cursor.pos.obj;
              cursor.next();
              return object;
          }
      }
      throw new NoSuchElementException("ListEnumerator");
  }
  ```

Here you see another use of the `synchronized` keyword:

```java
synchronized (expression) statement,
```

where `expression` is an object or array which is locked during execution of the `statement`. This ensures that no other threads can be executing the program section at the same time.

Collecting the Java code defining the `ListEnumerator` class we get:

**ListEnumerator.java**

```java
import java.util.*;

final class ListEnumerator implements Enumeration {
    LinkedList list;
    ListIterator cursor;

    ListEnumerator(LinkedList l) {
        list = l;
        cursor = list.head();
        cursor.next();
    }

    public boolean hasMoreElements() {
        return cursor.pos != list.head;
    }

    public Object nextElement() {
        synchronized (list) {
            if (cursor.pos != list.head) {
                Object object = cursor.pos.obj;
                cursor.next();
                return object;
            }
        }
        throw new NoSuchElementException("ListEnumerator");
    }
}
```
Demonstrating the Doubly-Linked List Implementation

To show how the doubly-linked lists really work and to test the correctness of the implementation, it is advisable to write a graphical program that allows you to do this. Below, you see a Java applet that simulates a doubly-linked list of integers. You start with an empty list and the first four buttons allow you to insert or remove elements. There is also a cursor belonging to the list. It is always positioned at the list item that is drawn in red. The previous and next buttons allow you to move the cursor. At the right-hand side we have a button to step through all list elements and to find an object in the list.

The source code for this applet is available and can be used for testing other data structures such as queues, stacks, singly-linked (circular) lists, and so on.