3 Doubly Linked Lists and Circular Lists

3.1 Objectives

The learning objectives of this laboratory session are:

- To understand the challenges of implementing circular singly-linked lists (especially when using arrays as for storage).
- To understand the issues which occur when implementing doubly-linked lists.
- To learn to compare the various simple list models and choose the most suitable one for the given problem.
- To get hands-on experience with implementing circular singly-linked lists using arrays as backing storage, and doubly-linked lists using dynamically allocated memory.
- To develop critical thinking concerning implementation decisions for Circular Singly-linked lists (CSLLs), and Doubly-linked Lists (DLLs).

The outcomes for this session are:

- The ability to compare simple list models and choose the most suitable one.
- Improved skills in C programming using pointers and dynamic memory allocation.
- A clear understanding of doubly linked lists and the operations which they support.
- The ability to make implementation decisions concerning the doubly-linked lists.
- The ability to decide when doubly-linked lists are suitable in solving a problem when compared to singly-linked lists.

3.2 Brief Theory Reminder

3.2.1 Circular Singly-linked Lists

A *circular singly linked list* is a singly linked list which has the last element linked to the first element in the list. Being circular it has no ends; thus we may use a single pointer $pNode$ to access the list. That pointer could point to, e.g., the newest element (last element inserted in a list). Figure 3.1 shows a model of such a list.

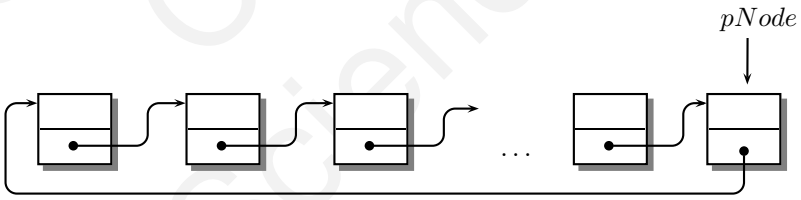

Figure 3.1: A model of a circular singly-linked list.

The structure of a node, when implemented in C, may be the same as the one used with singly-linked lists, as shown by Listing 3.1. Such a record structure is typically used with dynamically allocated list nodes. We will ***not*** focus on this choice in our laboratory session.

Listing 3.1: Sample singly-linked list node type definition

```c
typedef struct node
{
  int key; /* an optional field identifying the data */
  /* other useful data fields */
  struct node *next; /* link to next node */
} NodeT;
```

Another choice is to use an array as backing storage for the list nodes. In that case the structure of a node would be similar to the one shown in Listing 3.2. The list would be stored in an array of nodes. Thus, the next element link is based on a *cursor* — the `next` field of the record is the index of the element succeeding the current element.

Listing 3.2: Sample singly-linked list node type definition for array-based implementation

```
typedef struct node
{
  int key; /* an optional field identifying the data */
  /* other useful data fields */
  int next; /* index of the next node */
} NodeT;
```

3.3 Operations on a Circular Singly-linked List

Creating a Circular Singly-linked List

1. Creating a list involves two steps:

   a) Creating a header cell. A header cell may have a structure similar to the one shown in Listing 3.3. The field `capacity` holds the value of the maximum number of elements which may be stored in this list; the field `first` holds the index of the first element in the dynamically allocated array pointed to by the field `data`; the filed `count` holds the number of elements currently stored in the list; and the filed `available` holds the index of the first unused location in array `data` – i.e. the beginning of the *available* memory list.
   b) Allocating the backing storage for the list, and, possibly, initializing it.

Listing 3.3: Circular singly-linked list header cell structure when an array-based implementation is used.

```
typedef struct node
{
  unsigned int capacity; // the maximum number iof elements this list is able to store
  int available; // index of the next available (unused) location of the data area
  int first; // index of the first element
  int count; // the number of elements currently stored in the list; optional
  NodeT *data; // pointer to array holding the elements
} ArrListT;
```

Figure 3.2 illustrates an empty CSLL using the header implementation shown in Listing 3.3. Figure 3.3 shows a CSLL resulted from a sequence of insertions and deletions. Note that the nodes are chained on the field `next` of the node. The field `next` is also used to chain the unused locations in the array in the list of available space.

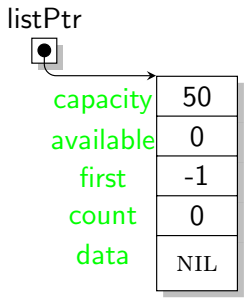

Figure 3.2: Empty circular SLL where data is stored in an array

A C implementation of a function which performs the create operation may by as the one shown inf Listing 3.4. Note that the node structure is the one shown in Listing 3.2.

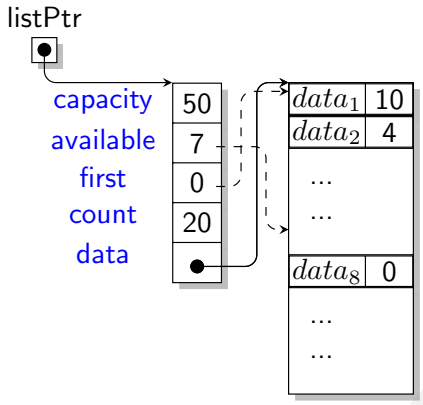

Figure 3.3: Example circular SLL using array-based implementation with elements inserted.

Listing 3.4: Sample list creation for array-based implementation

```
// create a (circular) singly-linked list by allocating space to store its elements
ArrListT *createCSLL(int size)
{
        ArrListT *listPtr = (ArrListT *) malloc(sizeof(ArrListT));
        if (listPtr)
    {   // have header cell; allocate data area
                listPtr->data = (NodeT *) calloc(size, sizeof(NodeT));
                if (listPtr->data)
                {
                        listPtr->capacity = size;
                        listPtr->first = -1;
                        listPtr->count = 0;
                        listPtr->available = 0; // all the data area is available
                        // link all the available storage into the available list
                        // i.e chain to the next element
                        for (int i = 0; i < listPtr->capacity - 1; i++)
                        {
                                listPtr->data[i].next = i + 1;
                        }
                        // mark end of available memory
                        listPtr->data[listPtr->capacity - 1].next = NOT_FOUND;
                }
                else
                {
                        free(listPtr);
                        listPtr = NULL;
                }
        }
        return listPtr;
}
```

2. With this implementation choice, we no longer need to create nodes. Memory for node storage is reserved when creating the list. In our implementation for CSLL, we will assume that the only data stored at the node is a key of type `int`

Finding a Node of a Circular Singly-linked List

Listing 3.5 show a possible implementation of a function to look for a node with a given key in a circular singly linked list. Function *next* implements the discipline to access the next node in such a list.

Listing 3.5: Finding a n node with a given key for array-based implementation.

```
#define NOT_FOUND -1
// returns the index of the next node in list pointed to by listPtr
static int next(ArrListT *listPtr, int current)
{
        return listPtr->data[current].next; // specific implementation
}
// returns the index of the node with key key in in list pointed to by listPtr
// or NOT_FOUND if key key is not found
int find(ArrListT *listPtr, int key)
{
        if (isEmpty(listPtr))
                return NOT_FOUND; // -1 is used for non-existing indexes
        int current = listPtr->first;
        do
        {       // scan the list
                if (listPtr->data[current].key == key)
                        return current; // found
                current = next(listPtr, current);
        }
        while (current != listPtr->first); // till we come back where we started
        return NOT_FOUND;
}
```

Inserting a Node of a Circular Singly-linked List

Inserting at the Front of a List

Listing 3.6: Inserting a node at the front or end of a list for array-based implementation.

```
// Insert a node with key key in in list pointed to by listPtr.
int insert(ArrListT *listPtr, int key)
{
        if (listPtr->available == NOT_FOUND)
                return NOT_FOUND; // list is full
        // copy the data payload to store at the node in the first available cell
        listPtr->data[listPtr->available].key = key;
    int last = listPtr->first;
    // check if the list is not empty
    if (last == NOT_FOUND)
    {
        listPtr->first = listPtr->available;
        listPtr->available = next(listPtr->available);
    }
    else
    {
            while (next(listPtr, last) != listPtr->first)
                {   // scan the list for the last element
                last = next(listPtr, last);
        }

        int aux = next(listPtr, listPtr->available); // save chaining in available list
        listPtr->data[listPtr->available].next = listPtr->first; // make list circular with new
             element
        // chain the last element to the new one
        listPtr->data[last].next = listPtr->available;
        listPtr->available = aux; // change beginning of available slots
    }
        listPtr->count++; // cell added
        return listPtr->first;
}
```

Inserting Before a Node with Key *givenKey*

In order to maintain a list sorted, a node may be inserted *before* a node containing a given key, or *after* it. Both cases imply searching for that key, and, if that key exists, creating the node to insert, and adjusting links accordingly.

There are two steps to execute:

1. Find the node with key $givenKey$, as illustrated by Listing 3.7.

Listing 3.7: Finding a n node with a given key for array-based implementation.

```
// copy the data in the first location available in the backing storage, temporary
if (listPtr->available != NOT_FOUND)
{       // there is available storage for new elements
        // the next statement assumes that only the key is stored in every node
        listPtr->data[listPtr->available].key = givenKey;
}
else
{ // TODO: print a warning message and quit; no more space in the list
  // other approaches exist, although
}
int currentIndex, prevIndex;

prevIndex = -1; /* initialize as out of the list */
currentIndex = listPtr->first;
do
{
  prevIndex = currentIndex;
  currentIndex = next(listPtr, currentIndex);
  if ( listPtr->data[currentIndex].key == givenKey ) break;
}
    while ( currentIndex != listPtr->first );
// now current either holds the index of the node with the given key
// or not
```

2. Chain the node temporary stored in the list as shown in Listing 3.8.

Listing 3.8: Chaining a node for array-based implementation when inserting a node before one with a given key.

```
if ( listPtr->data[currentIndex].key == givenKey )
{ /* node with key givenKey has index currentIndex */
  listPtr->data[prevIndex].next = listPtr->available;
        listPtr->data[listPtr->available].next = currentIndex;
  // adjust available memory
        listPtr->available = next(listPtr, listPtr->available);
}
// else given key not found; no changes to the list needed
```

Inserting After a Node with Key *givenKey*

Again, there are two steps to execute:

1. Find the node with key $givenKey$ as illustrated in Listing 3.9.

Listing 3.9: Finding a n node with a given key for array-based implementation.

```
// copy the data in the first location available in the backing storage, temporary
if (listPtr->available != NOT_FOUND)
{       // there is available storage for new elements
        // the next statement assumes that only the key is stored in every node
        listPtr->data[listPtr->available].key = givenKey;
}
else
{ // TODO: print a warning message and quit; no more space in the list
  // other approaches exist, although
}
int currentIndex

currentIndex = listPtr->first;
do
{
  currentIndex = next(listPtr, currentIndex);
        if ( listPtr->data[currentIndex].key == givenKey ) break;
}
    while ( currentIndex != listPtr->first );
// now current either holds the index of the node with the given key
// or not
```

2. Chain the node temporary stored in the list as shown in Listing 3.10.

Listing 3.10: Chaining a node for array-based implementation for insertion after a node with a given key.

```
if ( listPtr->data[currentIndex].key == givenKey )
{ /* node with key givenKey has index currentIndex */
  listPtr->data[listPtr->available].next = listPtr->data[currentIndex].next;
  listPtr->data[currentIndex].next = listPtr->available;
  // adjust available memory
        listPtr->available = next(listPtr, listPtr->available);
}
// else given key not found; no changes to the list needed
```

Removing a Node from a Circular Singly-linked List

Again there are two steps to take:

1. Find the node with key $givenKey$ as shown before in Listing 3.7.
2. Delete the node indicated by cursor $currentIndex$, and adjust the information stored at the header of the list accordingly (see Listing 3.11).

Listing 3.11: Deleting a node for array-based implementation when deleting a nodewith a given key.

```
if ( listPtr->data[currentIndex].key == givenKey )
{ /* node with key givenKey has index currentIndex */
        // skip node with key
  listPtr->data[prevIndex].next = listPtr->data[currentIndex].next;
  // add node to the front of available list
        listPtr->data[currentIndex].next = listPtr->available;
  listPtr->available = currentIndex;
  // set previous node as first node
  listPtr->first =prevIndex;
  listPtr->count--;
  if (listPtr->count == 0)
         purge(listPtr);
}
// else given key not found; no changes to the list needed
```

Purging a Circular Singly-linked List

Our implementation simplifies this operation a lot. We just have to adjust the available list and some of the fields in the header cell, as shown by Listing 3.12.

Listing 3.12: Purging a CSLL

```
// purge the list. I.e. simply initialize it again.
void purge(ArrListT *listPtr)
{
                        listPtr->first = -1;
                        listPtr->count = 0;
                        listPtr->available = 0; // all the data area is available
                        // link all the available storage into the available list
                        // i.e chain to the next element
                        for (int i = 0; i < listPtr->capacity - 1; i++)
                        {
        listPtr->data[i].next = i + 1;
                        }
                        // mark end of available memory
    listPtr->data[listPtr->capacity - 1].next = NOT_FOUND;
}
```

3.3.1 Doubly-linked Lists

A *doubly-linked* list is a (dynamically allocated) list where the nodes feature two relationships: *successor* and *predecessor*. A model of such a list is given in figure 3.4. The type of a node in a doubly-linked list may be defined as shown in Listing 3.13.

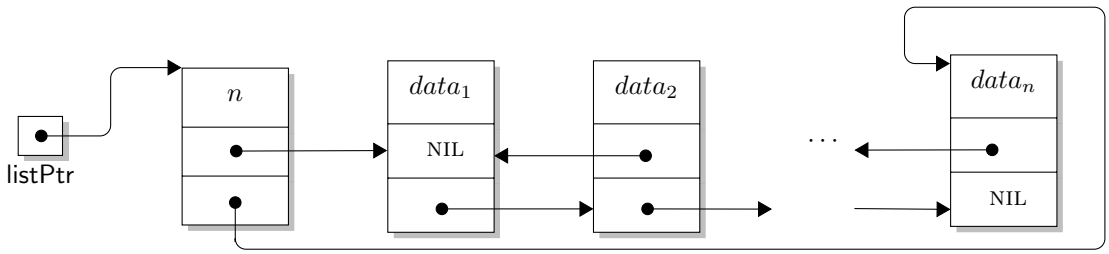

Figure 3.4: A model of a doubly linked list.

Listing 3.13: Example DLL node structure

```
typedef struct node_type
{
  int key; /* we assume here that the data is an integer key alone */
  // a more general implementation would include a statement like the following
  DataT data; /* useful data payload; may contain a field used as a key */
  struct node_type *next; /* pointer to next node */
  struct node_type *prev; /* pointer to previous node */
} NodeT;
```

As we have seen when discussing singly-linked lists, the main operations for a doubly-linked list are:

- creating a cell;
- accessing a cell;
- inserting a new cell;
- deleting a cell;
- purging a list.

3.4 Operations on a Doubly Linked List

In what follows we shall assume that the list is given by a pointer to its header cell. Thus a DLL may have a header cell with a structure similar to the one shown in Listing 3.14.

Listing 3.14: Example of doubly-linked list header cell type definition

```
typedef struct
{
  int count; /* number of elements in this list; an optional field */
  NodeT *first; /* link to the first node in the list */
  NodeT *last; /* link to the last node in the list */
} DLListT;
```

Creating a Doubly Linked List

To create a doubly-linked list with a header we might create a header cell alone, as we did with singly-linked lists (cf. Listing 3.15).

Listing 3.15: Sample code to create an empty singly-linked list

```
/* Create an empty list */
ListT *createEmptyDLL()
{
        ListT *listPtr = (ListT*)malloc(sizeof(ListT));
        if (listPtr)
        {
                listPtr->count = 0; // list empty
                listPtr->first = listPtr->last = NULL;
        }
        return listPtr;
}
```

The list holds nodes. Thus, it is a good choice to have a function (like the one in Listing 3.16) to create a DLL node and fill the data held in the node. We will assume for simplicity that the data payload is an composed of an integer key only.

Listing 3.16: Example code to create and fill a DLL node with data

```
/* Create a node and fill it with data */
NodeT *createDLLNode(int key)
{
    NodeT *p = (NodeT *)malloc(sizeof(NodeT));
    if (p)
    {
        // what is done here depends on the data stored at the node
        p->key = key; // assignment allowed as the key is of a primitive type
        p->next = p->prev = NULL; // initialize links for disconnected node */
    }
    return p;
}
```

Traversing a Doubly Linked List

We can traverse a DLL in a loop, in forward or backward direction as shown by Listing 3.17.

Listing 3.17: Example code traversing a DLL

```
// traverse DLL in forward direction
for ( p = listPtr->first; p != NULL; p = p->next )
{
  /* some operation o current cell */
}
// traverse DLL in backward direction
for ( p = listPtr->last; p != NULL; p->p->prev )
{
  /* some operation o current cell */
}
```

Finding a node in Doubly Linked List

This operation commonly uses the value of a key. Finding a node based on a given key may be achieved as shown in Listing 3.18.

Listing 3.18: Example code for a function to find a node with a given key in a DLL

```
NodeT *find(ListT *listPtr, int givenKey)
{
    NodeT *p;
    p = listPtr->first;
    while ( p != NULL )
        if ( p->key == givenKey ) /* Note. This comparison does work for primitive types only
            */
        {
            return p; /* key found in cell p */
        }
        else
            p = p->next;
    return NULL; /* not found */
}
```

Inserting a Node of a Doubly Linked List

The insert operation may have various forms. Thus, one may:

- **insertAtFront** – insert every new element at the beginning of the list. This is always used with stacks (the *push* operation of a stack) or queues (the *enqueue* operation).
- **insertAtRear** – insert every new element at the end of the list. Also known as an *append*. This is used, e.g. with queues (the *enqueue* operation of a queue).
- **insertInOrder** – every new element is inserted into a sorted list in ascending or descending order of its *key* field.

Listing 3.19 below shows the implementation for **insertAtFront** and **insertAtRear**.

Listing 3.19: Example code for inserting a node at the beginning or end of a DLL

```
// Example code for insertAtFront; does not check for duplicate keys
int insertAtFront(ListT *listPtr, NodeT *p)
{
    if (listPtr)
    { // the list is non null
        if (isEmpty(listPtr))
        {
            // p will be the sole node in the list
            listPtr->first = listPtr->last = p;
            p->prev = p->next = NULL; // p is the first and single node
        }
        else
        {
                p->next = listPtr->first; // chain before the former first node
                listPtr->first->prev = p; // p is before the former first node
                p->prev = NULL; // p is the first node
        }
        listPtr->count++;
        return 1; // success
    }
    return 0; // failure
}
// Example code for insertAtRear; does not check for duplicate keys
int insertAtRear(ListT *listPtr, NodeT *p)
{
        if (listPtr)
        {  // the list is non null
        if (isEmpty(listPtr))
        {
            // p will be the sole node in the list
            listPtr->first = listPtr->last = p;
            p->prev = p->next = NULL; // p is the first and single node
        }
                else
                { // non-empty list
                        listPtr->last->next = p; // chain after the former last node
                        p->prev = listPtr->last; // p is after the former last node
                        p->next = NULL; // p is the last node
                }
                listPtr->count++; // increment number of nodes
                return 1; // success
        }
        return 0; // failure
}
```


- Insertion before a node given by its *key*. There are two steps to execute:

1. Search for the node containing a *givenKey*, as shown by Listing 3.20.

   Listing 3.20: Sample code for searching a node with a given key in a DLL, preceeding an insert operation

   ```
   NodeT *q;
   q = listPtr->first;
   while ( q != NULL )
   {
       if ( q->key == givenKey ) break;
       q = q->next;
   }
   ```

2. Insert the node pointed to by *p*, and adjust links, as Listing 3.21 shows.

Listing 3.21: Sample code for inserting a node with a given key in a DLL

```
if ( q != NULL )
{   /* node with key givenKey has address q */
    if ( q == listPtr->first )
    {   /* insert before first node */
        p->next = listPtr->first; // chain with former first
        listPtr->first = p; // set p as first node
        p->prev = NULL;
    }
    else
    {
        q1->next = p;
        p->prev = q1;
        p->next = q;
        q->prev = p;
    }
    listPtr->count++; // increment number of nodes
    // success
}
// failure;
```

- Insertion after a node given by its *key*. Again, there are two steps to execute:
  1. Search for the node containing the *givenKey*, as shown before in Listing 3.20.
  2. Insert the node pointed to by *p*, and adjust links, as shown by Listing 3.22.

Listing 3.22: Sample code for inserting a node with a given key in a DLL

```
if ( q != NULL )
{   /* node with key givenKey has address q */
    p->next = q->next;
    q->next->prev = p;
    q->next = p;
    p->prev = q;
    if ( q == listPtr->last ) listPtr->last = p;
    listPtr->count++; // increment number of nodes
    //success
}
    // failure
```

Purging a Doubly Linked List

Purging a list means deleting each of its nodes, one by one (cf. Listing 3.23). The operation is very similar to purging a singly-linked list.

Listing 3.23: Purging a DLL

```
NodeT *p;

while ( listPtr->first != NULL )
{
  p = listPtr->first;
  listPtr->first = listPtr->first->next;
  free( p );
}
listPtr->last = NULL;
```

3.5 Laboratory Assignments

Study the provided code and use that code and code derived from it to solve the mandatory problems.

For all proposed problems, input and output data should reside in files whose names are given as command line arguments.

Here are some issues to think about:

- What are the pros and cons for using static vs. dynamically allocated memory for storing a list?
- What are pros and cons for using a doubly-linked lists?
- Why should we try to achieve a more general (abstract) implementation of a list instead of "solving" the problem at hand?

- What are the benefits of using a modular approach instead of a monolithic one in our implementation?
- How can we test our implementation?
- How can we generate test data to use in checking that our program works properly?

3.5.1 Mandatory Assignments

3.1. Simulate the following game: n children sit on a circle; one of the children starts counting the others clockwise. Every n^{th} child leaves the game. The winner is the one who stays in the game till the end. You are supposed to read the input from a file with a name given as a command line argument and print output to another file given as a command line argument, as well. The first line of input contains the number of children, n; the following lines contain child names. .
I/O description. Input:

```
8
John
James
Ann
George
Amy
Fanny
Winston
Bill
```

Output: winner's name.

3.2. Words are read from a file whose name is a given as a command line argument. The output should also be sent to a file whose name is a command line argument. Words are sequences containing only letters, and separated by whitespace. You should insert all the unique words in a doubly-linked list, sorted in ascending order of the words. The list should contain in its data cells every distinct word and its number of occurrences in the input file. The output should be the content of the list in ascending and descending order, as shown in the example below. .
I/O description. Input: a single line of text. (But, note: it may be quite long.) See the example below:

```
Everything LaTeX numbers for you has a counter associated with it.  The name of the counter
is the same as the name of the environment or command that produces the number, except
with no \.  Below is a list of some of the counters used in LaTeX's standard document styles
to control numbering.
```

Output:

```
\.:1
a:2
as:1
associated:1
Below:1
command:1
control:1
counter:2
counters:1
document:1
environment:1
Everything:1
except:1
for:1
has:1
in:1
is:2
it.:1
LaTeX:1
LaTeX's:1
list:1
name:2
no:1
number,:1
numbering.:1
numbers:1
of:5

produces:1
same:1
some:1
standard:1
styles:1
that:1
the:6
The:1
to:1
used:1
with:2
you:1
you:1
with:2
used:1
to:1
the:6
The:1
that:1
styles:1
standard:1
some:1
same:1
produces:1
of:5
numbers:1
numbering.:1
number,:1
no:1
name:2
list:1
LaTeX:1
LaTeX's:1
it.:1
is:2
in:1
has:1
for:1
except:1
Everything:1
environment:1
document:1
counters:1
counter:2
control:1
command:1
Below:1
associated:1
as:1
a:2
\.:1
```


3.5.2 Extra Points Assignment

3.3. Implement a doubly-linked list using the XOR function. Such a list uses a single pointer for chaining in both directions. The method takes advantage of the fact that, for value `A`, `B`, and `C`, the following equations hold:

```
A XOR B = C
C XOR A = B
C XOR B = A
```

Quote from XOR linked list:
"An ordinary doubly linked list stores addresses of the previous and next list items in each list node, requiring two address fields:

```
...  A       B         C         D         E   ...
       ->  next  ->  next  ->  next  ->
       <-  prev  <-  prev  <-  prev  <-
```

An XOR linked list compresses the same information into one address field by storing the bitwise XOR (here denoted by $\oplus$) of the address for previous and the address for next in one field:

$$\ldots \quad A \quad B \quad C \quad D \quad E \quad \ldots$$

i.e.

$$\leftrightarrow \quad A \oplus C \quad \leftrightarrow \quad B \oplus D \leftrightarrow \quad C \oplus E \quad \leftrightarrow$$

More formally:

$$link(B) = addr(A) \oplus addr(C),\ link(C) = addr(B) \oplus addr(D), \ldots$$

When you traverse the list from left to right: supposing you are at C, you can take the address of the previous item, B, and XOR it with the value in the link field ($B \oplus D$). You will then have the address for D and you can continue traversing the list. The same pattern applies in the other direction.
i.e. $addr(D) = link(C) \oplus addr(B)$
where
$link(C) = addr(B) \oplus addr(D)$
so
$addr(D) = addr(B) \oplus addr(D) \oplus addr(B)$
$addr(D) = addr(B) \oplus addr(B) \oplus addr(D)$
since
$X \oplus X = 0 \Rightarrow addr(D) = 0 \oplus addr(D)$ since $X \oplus 0 = X \Rightarrow addr(D) = addr(D)$ The XOR operation cancels $addr(B)$ appearing twice in the equation and all we are left with is the $addr(D)$.

To start traversing the list in either direction from some point, you need the address of two consecutive items, not just one. If the addresses of the two consecutive items are reversed, you will end up traversing the list in the opposite direction."
Input and output is stored in files, with names given as command line arguments.

3.5.3 Optional Assignments

3.4. Define and implement functions for operating on the data structure given below and the model shown in Figure 3.5.

```
typedef struct circularList
{
  int length;
  NodeT *first, *current;
} CircularSLLT;
```

Operations should be coded as: `ins<number>`=insert `<number>` in the list if it is not already there, `del<number>`=delete `<number>` from list (if it is on the list), `dcr`=delete at current position, `icr<number>`=insert at current position, `ist<number>`=insert `number` as first, `dst`=delete first, `prt`=print list. .

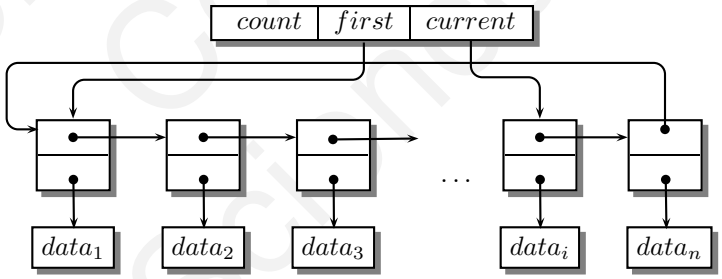

Figure 3.5: Another model of a circular singly-linked list.

I/O description. Input and output is stored in files, with names given as command line arguments.

3.5. Implement a circular buffer (contiguous memory area) to hold records containing student-related data. A producer-consumer principle is to be implemented according to the following:

a) Records are accepted as they are produced.
b) If there are no records in the buffer, consumer is postponed till producer places one.
c) If the buffer fills up, producer is postponed till consumer takes one out.

Use a limit of 1o for queue size, so it fills up fast. .
I/O description. Input and output is stored in files, with names given as command line arguments. Input:

```
p227,Ionescu␣I.␣Ion,␣3071,␣DSA,␣10
p231,Vasilescu␣T.␣Traian,␣3087,␣DSA,␣5
lq
p555,John␣E.␣Doe,␣3031,␣DSA,␣9
p213,King␣K.␣Kong,␣3011,␣DSA,␣2
```

```
p522,Mickey␣M.␣Mouse,␣3122,␣ART,␣10
...␣/*␣'...'␣means␣more␣records␣*/
lq
p573,Curtis␣W.␣Anthony,␣3012,␣ART,␣10
p257,Bugs␣R.␣Bunny,␣3000,␣GYM,␣10
c
c
c
c
lq
```

Output:

```
227,Ionescu␣I.␣Ion,␣3071,␣DSA,␣10
231,Vasilescu␣T.␣Traian,␣3087,␣DSA,␣5
eoq
227,Ionescu␣I.␣Ion,␣3071,␣DSA,␣10
231,Vasilescu␣T.␣Traian,␣3087,␣DSA,␣5
555,John␣E.␣Doe,␣3031,␣DSA,␣9
213,King␣K.␣Kong,␣3011,␣DSA,␣2
522,Mickey␣M.␣Mouse,␣3122,␣ART,␣10
...
eoq
queue␣full.␣postponed␣573
queue␣full,␣postponed␣257
522,Mickey␣M.␣Mouse,␣3122,␣ART,␣10
...
573,Curtis␣W.␣Anthony,␣3012,␣ART,␣10
257,Bugs␣R.␣Bunny,␣3000,␣GYM,␣10
eoq
```

where `227`=student id, `Ionescu I. Ion`=name, `3071`=group id, `DSA`= 3-letter course code, `10`=mark; `eoq`=end-of-queue, `p`=record from producer, `c`=record is consumed, `lq`=list queue contents [command].

3.6. Define and implement the operations for a doubly-0linked list with satellite data. A model for such a list is shown in Figure 3.6. .
I/O description. Input and output is stored in files, with names given as command line arguments. Operations should be coded as: `ins<number>`=insert `<number>` in the list, `del<number>`=delete `<number>` from list, `dcr`=delete at current position, `icr<number>`=insert at current position, `ist<number>`=insert `number` as first, `dst`=delete first, `prt`=print list, `ita<number>`=insert `number` as last, `dta`=delete last node. Here `<number>` represents a numeric value to store as data payload with a node.

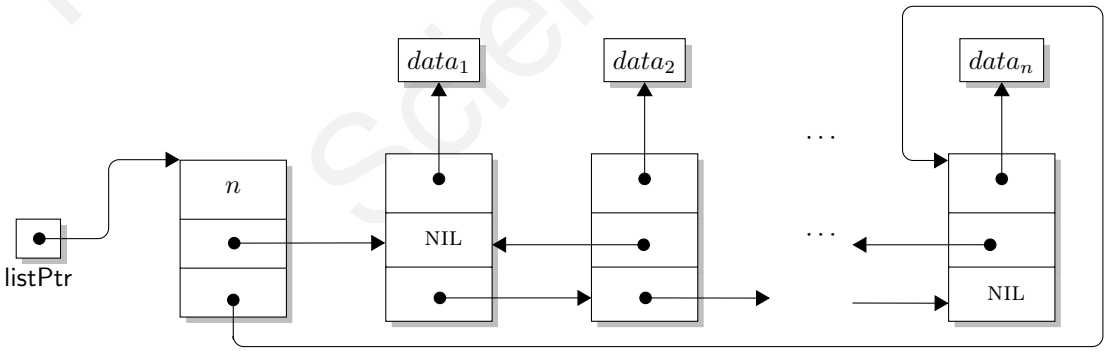

Figure 3.6: A list model for problem 3.6.

3.7. Simulate a game, using a doubly linked list with a sentinel (cf. Figure 3.7). The game involves n children, and you are supposed to read n and their names from the standard input. The children are arranged in a circle. Starting with a child whose name is given on the standard input, children are counted clockwise. Every m^{th} child is eliminated from the game. Counting goes on with the next child. The last child to leave the game wins. The input filename (with data) and the output filename are given as command line arguments. .
I/O description. Input. The first line contains the values for n, and m. Every line which follows contains a name of a child.

```
8␣5
```

```
John
James
Ann
George
Amy
Fanny
Winston
Bill
```

Output:

```
Prime␣[7␣children]:
------
John
James
Ann
Amy
Winston
Alex
Thomas

Non-prime␣[6␣children]:
----------
George
Fanny
Bill
Suzanna
Hannah
Thomas

Starter:␣Bill
Count:␣7

Empty␣list
```

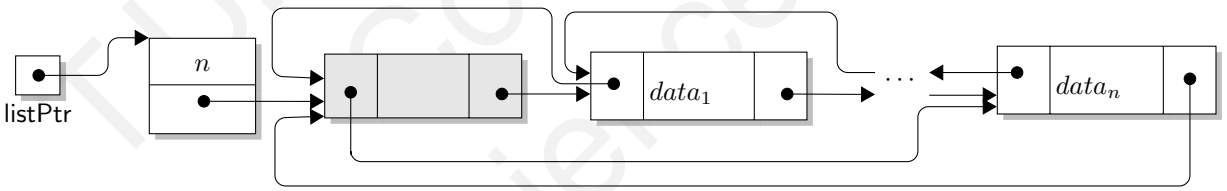

Figure 3.7: A doubly-linked list with a sentinel used for problem 3.7.