CTEC2901 Data Structures and Algorithms 2013-14

Doubly-linked Circular List

David Smallwood

Contents

1 Singly Linked Lists

The data structures we have studied so far are all examples of *linear* data structures. Every node in the structure has (at most) one predecessor and (at most) one successor which means that the nodes can be arranged in a line (hence "linear"). The reason we make reference to "at most" one predecessor/successor is because the definition needs to account for the *terminal* nodes – those at the "ends".

Our linked data strucutre implementations have each made *one* of these relationships explcit while leaving the other implicit. This results in the classic *singly linked list* data structure characterised by the following illustration:

$$\bigcirc \longrightarrow \bigcirc \longrightarrow \bigcirc \longrightarrow \ldots \longrightarrow \bigcirc$$

The "first" node in this illustration might represent the top of a stack, for example, or the front of a queue. Recall the linked implementation of the stack data structure:

```
struct node
{
    any item;
    struct node * next;         // each item points to the next in line
};

struct stack_any_implementation
{
    struct node * top;          // points to the first item in the list
    int size;
};
```

Figure 1: (void*) Singly Linked Stack Implementation

Singly linked structures have the following properties:

1. Only one link (pointer) is saved inside each node. This is obviously less expensive than storing two pointers (a *next* and a *prev*, say).[1]

[1] In our example we have used the `any` data type (i.e. (void*)) but consider a list of `char` items. We

2. It is efficient ($\mathcal{O}(1)$) to add to the front (top) because the new front simply points to the old front.[2]

3. It is easy to move directly from one node to its successor just by following the (*next*) pointer.

4. It is expensive ($\mathcal{O}(\text{n})$) to access nodes near the back of the list because the list can only be traversed in one direction.[3]

5. It is impossible to move directly from one node to its predecessor because there is no pointer (*prev*, say) to follow.

2 Doubly Linked Lists

In light of our previous discusion there may be occasions in which it is more efficient to accept the overhead of an extra (*prev*) pointer in order to speed up navigation. Such *doubly* linked lists can be represented by the following illustration

$$\bigcirc \;\rightleftarrows\; \bigcirc \;\rightleftarrows\; \bigcirc \;\rightleftarrows\; \ldots \;\rightleftarrows\; \bigcirc$$

It is clear to see that navigation between nodes is facilitated in both directions. Such a data structure is useful for general linear data structures that require random access – i.e. ones in which we are not constrained to LIFO or FIFO protocols. This means that we will be able to add and remove nodes from *anywhere* inside the list (including, of course, at the ends).

2.1 How to insert a new node

Suppose we wish to insert a new node *after* a node that already exists within the list: let us trace the process step-by-step:

1. Assume we have located the node, c say, after which insertion will take place:

$$\textcircled{a} \;\rightleftarrows\; \textcircled{b} \;\rightleftarrows\; \overset{\downarrow}{\textcircled{c}} \;\rightleftarrows\; \textcircled{d}$$

could have two 32-bit pointers for every 8-bit item – a pointer/data ratio of 8:1. This is even worse, of course, with 64-bit pointers (16:1)

[2]Note to Haskell programmers: this is as easy as constructing a new list $x : xs$ by joining (:) a new node x to an old list xs: an $\mathcal{O}(1)$ operation.

[3]Once again Haskell programmers will recall that to append to the end of a list an entire list traversal is necessary, an $\mathcal{O}(\text{n})$ operation:
append [] $z = [z]$
append $(x : xs)\ z = x :$ *append* $xs\ z$

2. Create (malloc) the new node, n say:

$$\begin{tikzcd}
 & & {} \arrow[d] & \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{d} \arrow[l, shift left] \\
 & & \textcircled{n} & \\
 & & {} \arrow[u] &
\end{tikzcd}$$

3. Let $n.prev$ point to c:

$$\begin{tikzcd}
 & & {} \arrow[d] & \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{d} \arrow[l, shift left] \\
 & & \textcircled{n} \arrow[u] & \\
 & & {} \arrow[u] &
\end{tikzcd}$$

4. Let $c.next.prev$ point to n:

$$\begin{tikzcd}
 & & {} \arrow[d] & \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[r] & \textcircled{d} \arrow[dl] \\
 & & \textcircled{n} \arrow[u] & \\
 & & {} \arrow[u] &
\end{tikzcd}$$

5. Let $n.next = c.next$:

$$\begin{tikzcd}
 & & {} \arrow[d] & \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[r] & \textcircled{d} \arrow[dl, shift left] \\
 & & \textcircled{n} \arrow[u] \arrow[ur, shift left] & \\
 & & {} \arrow[u] &
\end{tikzcd}$$

6. Let $c.next$ point to n:

$$\begin{tikzcd}
 & & {} \arrow[d] & \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[d, shift left] & \\
 & & \textcircled{n} \arrow[u, shift left] \arrow[r, shift left] & \textcircled{d} \arrow[l, shift left] \\
 & & {} \arrow[u] &
\end{tikzcd}$$

We can redraw the last picture so that it appears on the page as a straight line:

$$\begin{tikzcd}
 & & {} \arrow[d] & & \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{n} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{d} \arrow[l, shift left]
\end{tikzcd}$$

See how the new node, n, has now been inserted *after* node c and *before* node d. You should trace through the sequence of pointer manipulations several times to ensure that you understand them. Note that if you get the sequence of pointer manipulations wrong you can be left with inaccessable data (a space leak) and consequent segmentation errors at runtime are likely. For example, let us redo the insertion wrongly:

2.2 How NOT to insert a new node

This demonstration shows what happens if the order of pointer updates is not managed properly:

1. Assume we have located the node, c say, after which insertion will take place:

$$\begin{tikzcd}
 & & {} \arrow[d] & \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{d} \arrow[l, shift left]
\end{tikzcd}$$

2. Create (malloc) the new node, n say:

$$\begin{tikzcd}
 & & {} \arrow[d] & \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{d} \arrow[l, shift left] \\
 & & \textcircled{n} & \\
 & & {} \arrow[u] &
\end{tikzcd}$$

3. Let $n.prev$ point to c:

$$\begin{tikzcd}
 & & {} \arrow[d] & \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{d} \arrow[l, shift left] \\
 & & \textcircled{n} \arrow[u] & \\
 & & {} \arrow[u] &
\end{tikzcd}$$

4. Let $c.next$ point to n (this is where it goes wrong)

$$\begin{tikzcd}
 & & {} \arrow[d] & \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[d, shift left] & \textcircled{d} \arrow[l] \\
 & & \textcircled{n} \arrow[u, shift left] & \\
 & & {} \arrow[u] &
\end{tikzcd}$$

5. What can we do now? We have lost all pointers to d. Accidentally node d has become garbage. Because we cannot access d, neither can we access d's *prev* pointer, so the linked list is broken!

3 The Problem with End Nodes

In the previous section we looked at inserting a new node into the list and how to rearrange the pointers (in the right order) to maintain the doubly linked list structure. However, things are different at the ends. Suppose we repeat our previous insertion example but this time imagine we want to insert the new node *after* node d – i.e. to append to the end of the list. We could proceed as before:

1. Locate the end node d after which insertion will take place:

$$\begin{tikzcd}
 & & & {} \arrow[d] \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{d} \arrow[l, shift left]
\end{tikzcd}$$

2. Create (malloc) the new node n:

$$\begin{tikzcd}
 & & & {} \arrow[d] & \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{d} \arrow[l, shift left] & \textcircled{n} \\
 & & & & {} \arrow[u]
\end{tikzcd}$$

3. Set $n.prev = d$:

$$\begin{tikzcd}
 & & & {} \arrow[d] & \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{d} \arrow[l, shift left] & \textcircled{n} \arrow[l] \\
 & & & & {} \arrow[u]
\end{tikzcd}$$

4. Set $d.next = n$:

$$\begin{tikzcd}
 & & & {} \arrow[d] & \\
\textcircled{a} \arrow[r, shift left] & \textcircled{b} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{c} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{d} \arrow[l, shift left] \arrow[r, shift left] & \textcircled{n} \arrow[l, shift left]
\end{tikzcd}$$

So what is the problem? This is easy. And, by symmetry, it would be equally easy to add a new node *before* the first node in the list, a. However, there is an important issue that needs to be raised and discussed. The *insert-after* routine will comprise two algorithms ((i) insert in the middle, (ii) insert at the end) and, similarly, the *insert-before* routine will incorporate two algorithms. *Per se* this is not a significant issue although one might be tempted to look for an alternative, *uniform* soltion if one were to exist. However, let us consider a further boundary condition: an empty list. How do any of the insertion routines behave when the list is empty? Once again, each will have to include special case code to handle this (rare) situation. Consider just the *insert-after* routine:

```
if      (the list is empty)
        create a new node and this becomes the list
else if (insert at the end)
        add a new node at the end of the list
else    insert the new node after the current one
```

Thus we have a three-way branching algorithm for insertion-after; the same again for insertion-before; and then we must consider the routine for deletion – does this occur wholly within the list, or at an end, or does the list become empty once the deletion has taken place? All of these situations require different combinations of pointer updates and will lead to fairly lengthy algorithms. The next implementation strategy allows us to eliminate all of these special cases and to handle insertion and deltion consistently regardless of whether or not the list is empty, or whether or not the activity occurs inside the list or at one of the "ends". Such a list is a *doubly-linked, circular* list with a *sentinel* node.

The main advantage of our new list implementation strategy is the *simplicity* of the source code. The algorithms will be shorter and consistent and, as a result, we expect them to be more reliable.

4 A Doubly-linked, Circular List with Sentinel Node

The following list implementation does not have any "end points" because it wraps around on itself. The arrows going off to the left of the picture connect with the corresponding arrows coming in on the right and vice versa. One of the nodes, the *sentinel*, has been marked $\otimes$ on the picture and this special node does not contain any item data. Its sole purpose is to provide a way in to the list.

$$\overset{\hookrightarrow}{\cdots} \quad \textcircled{c} \quad \rightleftarrows \quad \overset{\downarrow}{\bigotimes} \quad \rightleftarrows \quad \textcircled{a} \quad \rightleftarrows \quad \textcircled{b} \quad \underset{\hookleftarrow}{\cdots}$$

Thus, beginning with the sentinal and moving right (*next*), we follow the sequence

$$\otimes, a, b, c, \otimes, a, b, c, \ldots$$

We represent an *empty list* using the sentinel alone:

$$\overset{\hookrightarrow}{\cdots} \quad \overset{\downarrow}{\bigotimes} \quad \underset{\hookleftarrow}{\cdots}$$

In this case note that *sentinel.next* == *sentinel* and *sentinel.prev* == *sentinel*.

4.1 Adding a Point of Interest (a cursor)

The finishing touch to our data structure is achieved by incorporating a *cursor* that can navigate freely along the list. The idea is that if the cursor is at the sentinel node we consider it to be *not in* the list. Alternatively, if the cursor is pointing to any other node then we consider it to be *in* the list.

The cursor allows operations on the list to be written with respect to a point of interest. For example:

move-next the cursor advances to the *next* node

move-prev the cursor advances to the *previous* node

ins-after a new node is created and inserted *after* the current cursor position

ins-before a new node is created and inserted *before* the current cursor position

find-p move the cursor forward and locate the next node whose item satisfies p

We provide the full specification of the *clist* operators in the next section.

5 Cursor List (`clist`) Specification

We discuss a particular implementation of a list data structure: a list with a cursor. We will use C for the presentation.

Firstly we introduce the type of a node in the linked list. The structure is fairly standard with *generic* data stored by using a `void *` data type. Each node points to its successor and its predecessor. Notice, for reasons that are explained shortly, these `next` and `prev` pointers are never `NULL`.

```
struct node
{
    any item;
    struct node * next;
    struct node * prev;
};
```

The point about a *circular* implementation is that the last node points back to the beginning of the list and the first node points back to the end of the list. However, there is one other special node called the `sentinel` which is *always* present as long as the list exists. This sentinel node does not store any useful data and its `data` field is set to `NULL`. The struct representing a list object contains two other fields: the `size` of the list (initially zero); and another node pointer called the `cursor`. The cursor may be *moved* around the list rather like a cursor moves around the characters of a document in a text editor. The cursor may move to the left or to the right in single steps. Initially the cursor points to the sentinel node.

```
struct clist_implementation
{
    struct node * sentinel;
    struct node * cursor;
    int size;
};
```

Note that as the list grows we can consider that the sentinel node lies in between the first and last nodes in the list. So, if the cursor is at the last node and moves to the next position it lands on the sentinel. Similarly, if the cursor is at the first node and moves to the previous position it lands on the sentinel. If the cursor is at the sentinel and moves to the next node it moves to the first node in the list, and if the cursor is at the sentinel and moves to the previous node it moves to the last node in the list.

There are, of course, two special cases to note: (1) when the list is "empty" when the sentinel points to itself in both directions thus any *move* simply lands back on the sentinel; and, (2), when the list has one element then there is only one node apart from the sentinel and it is coincidentally the first and last node in the list.

5.1 Initialisation

```
clist * new_clist ();
```

Whenever a new clist object is requested the following sequence of statements must take place:

1. Allocate space for a (new) sentinel node and let `sentinel` point to it
2. Initialise the fields of the sentinel node: set the `data` field to `NULL`; set the `next` and `prev` fields to point to the newly created sentinel itself. Thus the list contains one (sentinel) node with both pointers pointing back to itself.
3. Initialise the `size` of the list to zero.
4. Initialise the `cursor` so that it points to the sentinel.

5.2 Returning the size of the list

```
int clist_size (clist *c)
```

This function simply returns the current size of the list. This function must not modify any of the fields of the list.

5.3 Is the cursor in the list?

```
int clist_cursor_inlist (clist *c)
```

It is possible for the cursor to be *at* the sentinel node. When this happens we say that the cursor *is not in the list.*

Whenever the cursor is pointing to a node in the list that is not the sentinel then we say that the cursor is *in the list.*

Thus it is easy to inspect the `cursor` and `sentinel` fields in the list structure and see if they are different (*in the list*) or the same (*not in the list*).

This function must not modify any of the fields of the list.

5.4 Is the list empty?

```
int clist_isempty (clist *c)
```

An empty list can be identified in a number of ways. Firstly, if the `size` is zero the the list is empty. Secondly, if the `sentinel` node points to itself in both directions then the list is empty. These states must be consistent. It is clearly easier to check the `size` field to implement the function.

However, it would be useful to use a C assertion to insist on the *invariant* property just described.

This function must not modify any of the fields of the list.

5.5 Returning the element at the cursor position

```
any clist_get_item (clist *c)
```

This is where the cursor becomes useful. It is the point at which actions take place. The first action we consider is one that returns the element that the cursor currently points to. Note that if the cursor is at the sentinel position then it is considered to be *not in the list* – if this function is called when the cursor is at the sentinel position then the value `NULL` must be returned. Otherwise a copy of the `data` pointer is returned.

This function must not modify any of the fields of the list.

5.6 Inserting an element

```
void clist_ins_before (clist *c, any  item);
void clist_ins_after  (clist *c, any  item);
```

All actions occur at the cursor position. An insertion operation extends the list. However, there are two possible places to insert the new node – either *before* or *after* the cursor position. Both functions are provided so that the client can choose which behaviour is required.

The operation must allocate a new node to store the new data item and link it in to the existing list structure correctly.

Note that *after* the operation has taken place the cursor should still point to the node that it was pointing to when the operation began. The `size` field should also be updated correctly.

5.7 Deleting an element

```
int clist_delete (clist *c);
```

The element at the cursor position is removed from the list and the node containing it is deallocated. The `size` is decremented appropriately and the deleted item is returned.

If the operation is called when the cursor is at the sentinel (i.e. *not in the list*) then nothing is removed and the list is not changed at all. In this case the value `NULL` is returned as the data reference.

If the operation is called when the cursor is *in the list* then the node at the current cursor position is deleted. However, successful deletion leaves a question: where does the cursor point to afterwards? We specify that the cursor is left pointing at the `next` node (i.e. the one that followed the node that was deleted). Note, a consequence of this is that when the *last* element in the list is deleted the cursor ends up on the sentinel.

5.8 Cursor movement operations

```
void clist_goto_head (clist *c);
void clist_goto_last (clist *c);
void clist_goto_next (clist *c);
void clist_goto_prev (clist *c);
```

These operations must not change the structure of the list in any way apart from the cursor which may move.

If the list is empty then `clist_goto_head` and `clist_goto_last` do not crash, but simply leave the cursor at the sentinel node. If the list is not empty then `clist_goto_head` and `clist_goto_last` move the cursor to the *first* and *last* elements in the list respectively. (Remember that the sentinel is considered to lie *between* the first and last nodes.)

The functions `clist_goto_next` and `clist_goto_prev` simply move the cursor forwards or backwards one position as indicated. Note that there are no special cases to consider for these operations – they will work consistently given any initial list configuration.

5.9 Apply a function to every element in the list

```
void clist_iterate (clist *c, modify f);
```

The iterator operation applies a function to each element in the list. The structure of the list must not be modified in any way (only the data items are allowed to change). At the end of the operation the cursor must be in the same position that it was in before the operation.

5.10 Locate an element in the list

```
int clist_find      (clist *c, pred p);
int clist_find_next (clist *c, pred p);
```

The user supplies a function that specifies a condition that the required element must satisfy. The search begins *following* the current cursor position. If no value is found that satisfies the condition then the cursor is set to the sentinel position.

The search does not "wrap around" so once the sentinel is reached the searching stops.

If the search is successful then the cursor is left at the *first node to satisfy* the condition.

Apart from the cursor, the rest of the list structure must not be modified in any way by this operation.

5.11 Print the list on the standard output stream

```
void clist_print (clist *c, void (* item_print)(any item));
```

The list should be enclosed in a pair of square brackets: "[" and "]". The individual items in the list should be separated by commas: "," as demonstrated by the following example:

```
[ 10,  28,  39,  44,  56,  64,  73,  82,  91]
```

NB: The spacing of the individual elements is determined by the printer function that is passed in. The example above used the following print function.

```
void show_int( void * d ) {
    printf("%3i", *(int *)d);
}
```

5.12 Release the list memory when the structure is no longer required

```
void clist_release (clist *c);
```

6 C Header File for the clist

```
#ifndef CLIST_H
#define CLIST_H

#include "any.h"

typedef struct clist_implementation clist;
typedef void (*modify)(any a);
typedef int (*pred)(any a);

clist * new_clist           ();
int     clist_size          (clist *c);
int     clist_isempty       (clist *c);
void    clist_goto_head     (clist *c);
void    clist_goto_last     (clist *c);
void    clist_goto_next     (clist *c);
void    clist_goto_prev     (clist *c);
int     clist_cursor_inlist (clist *c);
any     clist_get_item      (clist *c);
void    clist_ins_before    (clist *c, any  item);
void    clist_ins_after     (clist *c, any  item);
int     clist_delete        (clist *c);
void    clist_iterate       (clist *c, modify f);
int     clist_find          (clist *c, pred p);
int     clist_find_next     (clist *c, pred p);
void    clist_print         (clist *c, void (* item_print)(any item));
void    clist_release       (clist *c);

#endif
```

7 Demonstration Program

The following demonstration shows the clist operations being exercised. Look carefully at the code and see how the components have been assembled.

```
/* Program:  cl_demo1
 * Function: Demonstrating the use of the clist data structure
 * Author:   drs
 */

#include <stdlib.h>
#include <stdio.h>
#include "any.h"
#include "clist.h"

#define MAX_INTS 100
#define NEWLINE  printf("\n");

/************************************************************************************
 * Functions that operate on references to int values
 ************************************************************************************/

int even_int( any d )                    // even(d)      ?
{
    return ((*(int*)d) % 2) == 0;
}
void double_int( any d )                 // d *= 2
{
    *(int *)d = *(int *)d * 2;
}

/************************************************************************************
 * Functions to display the contents of a list
 ************************************************************************************/

void show_int( any d )
{
    printf("%3i", *(int *)d);
}
void show_list(clist * cl, char * caption) {
    any d;
    printf("\nCAPTION: \"%s\"",caption);
    printf("\n clist_size:          %3i", clist_size(cl));
    printf("\n clist_isempty:       %3i", clist_isempty(cl));
    printf("\n clist_cursor_inlist: %3i", clist_cursor_inlist(cl));
    printf("\n item at cursor:      ");
    d = clist_get_item(cl);
    if (d==NULL)
        printf("  NULL");
    else
        show_int( d );
    printf("\n clist_print:           ");
    clist_print(cl, &show_int);
    NEWLINE
}
```

```
int main()
{
    clist *xs;
    any j;

    /***********************************************************************************
     * A pool of data values is needed to be placed into the list during various
     * test cases.  A simple approach is to create an array of ints in which the
     * index is equal to the value.  The required "pointer to an integer" can then
     * be obtained by using the address of a particular array element.
     * Be careful though: these are mutable objects (having reference semantics)
     * and each new test case should re-initialise the array to avoid previous
     * modifications affecting behaviour (e.g. as would happen with a call to the
     * iterate function).
     ***********************************************************************************/
    int i, d[MAX_INTS];
    for (i=0;i<MAX_INTS;i++) d[i]=i;

    xs = new_clist();
    show_list(xs, "Empty");           // print out the empty list
    NEWLINE

    clist_ins_before( xs, &d[1] );
    clist_ins_before( xs, &d[2] );
    clist_ins_before( xs, &d[3] );
    clist_ins_before( xs, &d[4] );
    show_list(xs, "Inserted four integers before cursor");
    NEWLINE
    clist_ins_after( xs, &d[5] );
    clist_ins_after( xs, &d[6] );
    clist_ins_after( xs, &d[7] );
    clist_ins_after( xs, &d[8] );
    show_list(xs, "Inserted four integers after cursor");
    NEWLINE

    clist_find( xs, even_int );
    clist_delete( xs );
    clist_find_next( xs, even_int );
    clist_delete( xs );
    clist_find_next( xs, even_int );
    clist_delete( xs );
    clist_find_next( xs, even_int );
    clist_delete( xs );
    show_list(xs, "Located and deleted the even numbers");
    NEWLINE

    clist_iterate( xs, double_int );
    show_list(xs, "Doubled all the numbers in the list");
    NEWLINE

    clist_goto_last( xs );
    clist_goto_prev( xs );
    printf("Deleting value %i\n", clist_delete( xs ));
    show_list(xs, "Deleted penultimate value in the list");
    NEWLINE

    clist_goto_head( xs );
    clist_delete( xs );
    clist_delete( xs );
    clist_delete( xs );
    show_list(xs, "Emptied the list");
    NEWLINE

    clist_release( xs );

    return 0;
}
```

8 Exercises

Practical Exercise 1

The following instructions will install the `clist` libraries within your environment.

1. Download from Blackboard the file `ds.clist.tar.gz` and save it in your `ctec2901` directory. It is important to ensure that you download this into the `ctec2901` so double check that it downloaded into the correct place – if not, then move it into this directory.
2. (You should still be within `ctec2901`). Unzip and untar the compressed archive file that you just downloaded:

   ```
   tar xvzf ds.clist.tar.gz
   ```

   This will create the subdirectory `ds/clist`. Navigate to the `ds/clist` directory:

   ```
   cd ds/clist
   ```

3. (You should still be within `ctec2901/ds/clist`). You can type 'ls' and see the files that have been unpacked. Feel free to look at these *but be careful not to edit them.* To install all of the clist libraries type:

   ```
   make install
   ```

   `$HOME/include` will be populated with the clist header files and `$HOME/lib` will be populated with the clist library archive files. Normally you will leave these files where they have been placed.

Practical Exercise 2

Enter the example program from section 7 as program `cl_demo1.c`.

1. Study the code carefully (i.e. perform a *static analysis*)
2. Compile the program:

   ```
   gcc cl_demo1.c -o cl_demo -I$HOME/include -L$HOME/lib -llinked_clists
   ```

3. Run the program and check your understanding of the behaviour
4. Modify the program so that it exercises the clist functions in similar, but different ways

Portfolio Exercise (Doubly-linked Circular List)

For this portfolio exercise we want you to modify the actual implementation of the data structure by adding new functionality.

1. Add a function, `int clist_backspace(clist *c)`, that deletes the item *to the left* of (i.e. previous to) the current cursor position. You will need to modify the actual `clist.c` code in your `ds/clist` directory. Make sure you re-make and re-install the `linked_clists` library each time you modify it. Compilation, linking, and installing is all simply achieved by typing:

   ```
   make install
   ```

   Note that:

   (a) The backspace function does nothing if the node immediately previous to the cursor is the sentinel node.

   (b) The cursor position does not change following the operation.

2. Write a demo program, `pfclist.c`, that demonstrates your backspace function in operation. Use the `clist_print` function to show how your example list grows and then shrinks following insertions and backspace operations

9 Implementation of the `clist` in C

9.1 `clist.c`

```
// Linked implementation of an circular list with a sentinel node
// Author: drs
```

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "clist.h"

struct node
{
    any item;
    struct node * next;
    struct node * prev;
};

struct clist_implementation
{
    struct node * sentinel;
    struct node * cursor;
    int size;
};

clist * new_clist()
{
    clist * c = (clist *)malloc(sizeof(clist));
    if (c==NULL) {
        printf("new_clist: malloc failure.\n");
        exit(1);
    }
    c->size = 0;
    c->sentinel = (struct node *) malloc(sizeof(struct node));
    if (c->sentinel==NULL) {
        printf("new_clist(sentinel): malloc failure.\n");
        exit(1);
    }
    c->sentinel->item = NULL;
    c->sentinel->next = c->sentinel;
    c->sentinel->prev = c->sentinel;
    c->cursor = c->sentinel;
}

int clist_size(clist *c)
{
    assert(c!=NULL);
    return c->size;
}

int clist_isempty(clist *c)
{
    assert(c!=NULL);
    return c->size == 0;
}

void clist_goto_head(clist *c)
```

```
{
    assert(c!=NULL);
    c->cursor = c->sentinel->next;
}

void clist_goto_last(clist *c)
{
    assert(c!=NULL);
    c->cursor = c->sentinel->prev;
}

void clist_goto_next(clist *c)
{
    assert(c!=NULL);
    c->cursor = c->cursor->next;
}

void clist_goto_prev(clist *c)
{
    assert(c!=NULL);
    c->cursor = c->cursor->prev;
}

int clist_cursor_inlist(clist *c)
{
    assert(c!=NULL);
    return c->cursor != c->sentinel;
}

any clist_get_item(clist *c)
{
    assert(c!=NULL);
    return (clist_cursor_inlist(c)) ? c->cursor->item : NULL;
}

void clist_ins_before(clist *c, any  item)
{
    assert(c!=NULL);
    struct node * n = (struct node *) malloc(sizeof(struct node));
    if (n==NULL) {
        printf("clist_ins_before: malloc failure.\n");
        exit(1);
    }
    n->item = item;
    n->next = c->cursor;
    n->prev = c->cursor->prev;
    c->cursor->prev->next = n;
    c->cursor->prev = n;
    c->size++;
}
```

```
void clist_ins_after(clist *c, any  item)
{
    assert(c!=NULL);
    struct node * n = (struct node *) malloc(sizeof(struct node));
    if (n==NULL) {
        printf("clist_ins_after: malloc failure.\n");
        exit(1);
    }
    n->item = item;
    n->prev = c->cursor;
    n->next = c->cursor->next;
    c->cursor->next->prev = n;
    c->cursor->next = n;
    c->size++;
}

int clist_delete(clist *c)
{
    assert(c!=NULL);
    struct node * p = c->cursor;
    if (clist_cursor_inlist(c))
    {
        c->cursor = c->cursor->next;
        p->prev->next = p->next;
        p->next->prev = p->prev;
        free(p);
        c->size--;
        return 1;
    }
    else
        return 0;
}

void clist_iterate(clist *c, modify f)
{
    assert(c!=NULL);
    struct node * n = c->cursor;
    clist_goto_head(c);
    while (clist_cursor_inlist(c))
    {
        f(c->cursor->item);
        clist_goto_next(c);
    }
    c->cursor = n;
}

int clist_find(clist *c, pred p)
{
    assert(c!=NULL);
    clist_goto_head(c);
    while (clist_cursor_inlist(c) && (!p(c->cursor->item)))
```

```
        clist_goto_next(c);
    return clist_cursor_inlist(c);
}

int clist_find_next(clist *c, pred p)
{
    assert(c!=NULL);
    struct node * n = c->cursor;
    while (clist_cursor_inlist(c) && (!p(c->cursor->item)))
        clist_goto_next(c);
    if (clist_cursor_inlist(c))
        return 1;
    else {
        c->cursor = n;
        return 0;
    }
}

void clist_print(clist *c, void (* item_print)(any item))
{
    assert(c!=NULL);
    struct node * n = c->cursor;
    printf("CL[");
    clist_goto_head(c);
    if (clist_cursor_inlist(c)) {
        item_print(c->cursor->item);
        clist_goto_next(c);
        while (clist_cursor_inlist(c)) {
            printf(", ");
            item_print(c->cursor->item);
            clist_goto_next(c);
        }
    }
    printf("]");
    c->cursor = n;
}

void clist_release(clist *c)
{
    assert(c!=NULL);
    free(c->sentinel);
    free(c);
}
```