Cursor Implementation of Linked Lists
from Data Structures and Algorithm Analysis in C++, 2nd ed. by Mark Weiss

Many languages, such as `BASIC` and `FORTRAN`, do not support pointers. If linked lists are required and pointers are not available, then an alternate implementation must be used. The alternate method we will describe is known as a *cursor* implementation.

The two important items present in a pointer implementation of linked lists are

1. The data is stored in a collection of structures. Each structure contains the data and a pointer to the next structure.

2. A new structure can be obtained from the system's global memory by a call to *malloc* and released by a call to *free.*

Our cursor implementation must be able to simulate this. The logical way to satisfy condition 1 is to have a global array of structures. For any cell in the array, its array index can be used in place of an address. Figure 3.28 gives the type declarations for a cursor implementation of linked lists.

We must now simulate condition 2 by allowing the equivalent of *malloc* and *free* for cells in the *CURSOR_SPACE* array. To do this, we will keep a list (the *freelist*) of cells that are not in any list. The list will use cell 0 as a header. The initial configuration is shown in Figure 3.29.

A value of 0 for *next* is the equivalent of a *pointer*. The initialization of *CURSOR_SPACE* is a straightforward loop, which we leave as an exercise. To perform an *malloc*, the first element (after the header) is removed from the freelist.

```
typedef unsigned int node_ptr;

struct node
{
  element_type element;
  node_ptr next;
};

typedef node_ptr LIST;
typedef node_ptr position;
struct node CURSOR_SPACE[ SPACE_SIZE ];
```

Figure 3.28 Declarations for cursor implementation of linked lists

```
Slot Element Next
----------------------
  0      ?      1
  1      ?      2
  2      ?      3
  3      ?      4
  4      ?      5
  5      ?      6
  6      ?      7
  7      ?      8
  8      ?      9
  9      ?     10
 10      ?      0
```

Figure 3.29 An initialized CURSOR_SPACE

To perform a *free*, we place the cell at the front of the freelist. Figure 3.30 shows the cursor implementation of *malloc* and *free*. Notice that if there is no space available, our routine does the correct thing by setting $p = 0$. This indicates that there are no more cells left, and also makes the second line of *cursor_new* a nonoperation (no-op).

Given this, the cursor implementation of linked lists is straightforward. For consistency, we will implement our lists with a header node. As an example, in Figure 3.31, if the value of *L* is 5 and the value of *M* is 3, then *L* represents the list *a*, *b*, *e*, and *M* represents the list *c*, *d*, *f*.

```
position cursor_alloc( void )
{
  position p;
  p = CURSOR_SPACE[O].next;
  CURSOR_SPACE[0].next = CURSOR_SPACE[p].next;
  return p;
}

void cursor_free( position p)
{
  CURSOR_SPACE[p].next = CURSOR_SPACE[O].next;
  CURSOR_SPACE[O].next = p;
}
```

Figure 3.30 Routines: cursor-alloc and cursor-free

Slot	Element	Next
0	-	6
1	b	9
2	f	0
3	header	7
4	-	0
5	header	10
6	-	4
7	c	8
8	d	2
9	e	0
10	a	1

Figure 3.31 Example of a cursor implementation of linked lists

To write the functions for a cursor implementation of linked lists, we must pass and return the identical parameters as the pointer implementation. The routines are straightforward. Figure 3.32 implements a function to test whether a list is empty. Figure 3.33 implements the test of whether the current position is the last in a linked list.

The function *find* in Figure 3.34 returns the position of *x* in list *L*.

The code to implement deletion is shown in Figure 3.35. Again, the interface for the cursor implementation is identical to the pointer implementation. Finally, Figure 3.36 shows a cursor implementation of *insert*.

The rest of the routines are similarly coded. The crucial point is that these routines follow the ADT specification. They take specific arguments and perform specific operations. The implementation is transparent to the user. The cursor implementation could be used instead of the linked list implementation, with virtually no change required in the rest of the code.

```
int is_empty( LIST L ) /* using a header node */
{
  return( CURSOR_SPACE[L].next == 0
}
```

Figure 3.32 Function to test whether a linked list is empty--cursor implementation

```
int is_last( position p, LIST L) /* using a header node */
{
  return( CURSOR_SPACE[p].next == 0
}
```

Figure 3.33 Function to test whether p is last in a linked list--cursor implementation

```
position find( element_type x, LIST L) /* using a header node */
{
  position p;
  /*1*/ p = CURSOR_SPACE[L].next;
  /*2*/ while( p && CURSOR_SPACE[p].element != x )
  /*3*/ p = CURSOR_SPACE[p].next;
  /*4*/ return p;
}
```

Figure 3.34 Find routine--cursor implementation

```
void delete( element_type x, LIST L )
{
  position p, tmp_cell;
  p = find_previous( x, L );

  if( !is_last( p, L) )
  {
    tmp_cell = CURSOR_SPACE[p].next;
    CURSOR_SPACE[p].next = CURSOR_SPACE[tmp_cell].next;
    cursor_free( tmp_cell );
  }
}
```

Figure 3.35 Deletion routine for linked lists--cursor implementation

```
/* Insert (after legal position p); */
/* header implementation assumed */

void insert( element_type x, LIST L, position p )
{
  position tmp_cell;
  /*1*/ tmp_cell = cursor_alloc( )
  /*2*/ if( tmp_cell ==0 )
  /*3*/ fatal_error("Out of space!!!");
  else
  {
    /*4*/ CURSOR_SPACE[tmp_cell].element = x;
    /*5*/ CURSOR_SPACE[tmp_cell].next = CURSOR_SPACE[p].next;
    /*6*/ CURSOR_SPACE[p].next = tmp_cell;
  }
}
```

Figure 3.36 Insertion routine for linked lists--cursor implementation

The freelist represents an interesting data structure in its own right. The cell that is removed from the freelist is the one that was most recently placed there by virtue of *free*. Thus, the last cell placed on the freelist is the first cell taken off. The data structure that also has this property is known as a *stack*.