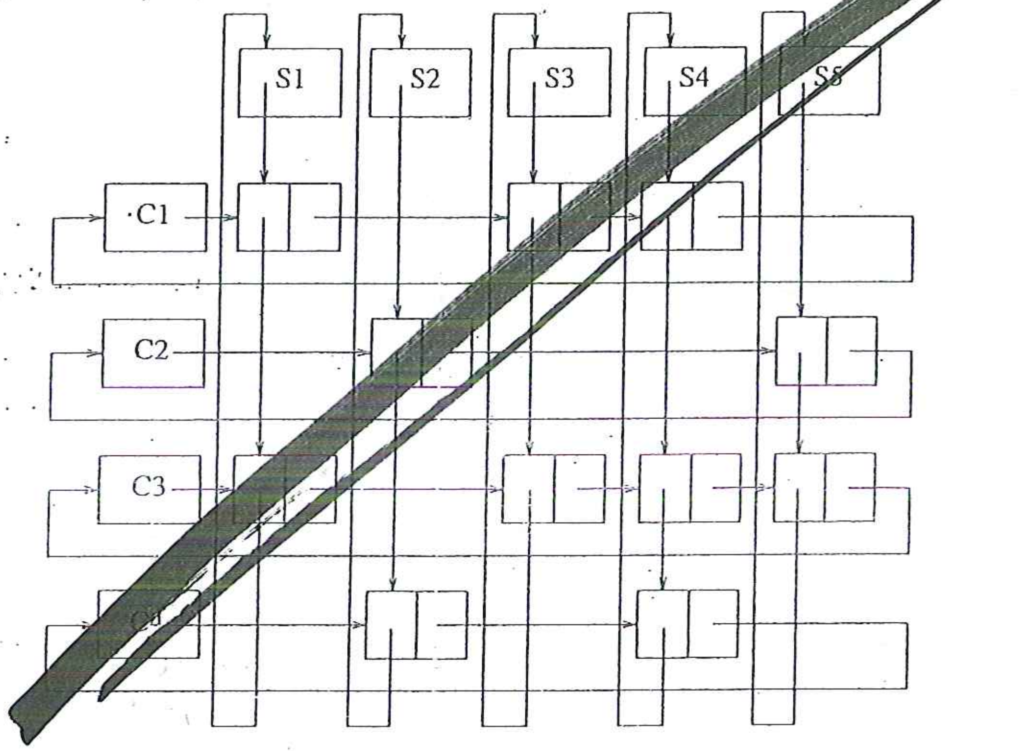

Figure 3.27 Multilist implementation for registration problem

3.2.8. *Cursor Implementation of Linked Lists*

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

The two important features present in a pointer implementation of linked lists are as follows:

1. The data are stored in a collection of structures. Each structure contains 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 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 *CursorSpace* 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 *NULL* pointer. The initialization of *CursorSpace* is a straightforward loop, which we leave as an exercise. To perform a *malloc*, the first element (after the header) is removed from the freelist. To perform

```
#ifndef _Cursor_H

typedef int PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;

void InitializeCursorSpace( void );

List MakeEmpty( List L );
int IsEmpty( const List L );
int IsLast( const Position P, const List L );
Position Find( ElementType X, const List L );
void Delete( ElementType X, List L );
Position FindPrevious( ElementType X, const List L );
void Insert( ElementType X, List L, Position P );
void DeleteList( List L );
Position Header( const List L );
Position First( const List L );
Position Advance( const Position P );
ElementType Retrieve( const Position P );

#endif      /* _Cursor_H */


/* Place in the implementation file */
struct Node
{
    ElementType Element;
    Position    Next;
};

struct Node CursorSpace[ SpaceSize ];
```

Figure 3.28 Declarations for cursor implementation of linked lists

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 *CursorAlloc* 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.

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

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 *CursorSpace*

```
static Position
CursorAlloc( void )
{
    Position P;

    P = CursorSpace[ 0 ].Next;
    CursorSpace[ 0 ].Next = CursorSpace[ P ].Next;

    return P;
}

static void
CursorFree( Position P )
{
    CursorSpace[ P ].Next = CursorSpace[ 0 ].Next;
    CursorSpace[ 0 ].Next = P;
}
```

Figure 3.30 Routines: *CursorAlloc* and *CursorFree*

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. If relatively few *Finds* are header 1

$c \to d \to f \to 0$

header 2

$a \to b \to e \to 0$

free $\boxed{6,4}$

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

Figure 3.31 Example of a cursor implementation of linked lists

```
/* Return true if L is empty */

int
IsEmpty( List L )
{
    return CursorSpace[ L ].Next == 0;
}
```

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

```
/* Return true if P is the last position in list L */
/* Parameter L is unused in this implementation */

int
IsLast( Position P, List L )
{
    return CursorSpace[ P ].Next == 0;
}
```

Figure 3.33 Function to test whether *P* is last in a linked list—cursor implementation

```
/* Return Position of X in L; 0 if not found */
/* Uses a header node */

Position
Find( ElementType X, List L )
{
    Position P;

    P = CursorSpace[ L ].Next;
    while( P && CursorSpace[ P ].Element != X )
        P = CursorSpace[ P ].Next;

    return P;
}
```

Figure 3.34 *Find* routine—cursor implementation

```
/* Delete first occurrence of X from a list */
/* Assume use of a header node */

void
Delete( ElementType X, List L )
{
    Position P, TmpCell;

    P = FindPrevious( X, L );

    if( !IsLast( P, L ) )   /* Assumption of header use */
    {                       /* X is found; delete it */
        TmpCell = CursorSpace[ P ].Next;
        CursorSpace[ P ].Next = CursorSpace[ TmpCell ].Next;
        CursorFree( TmpCell );
    }
}
```

Figure 3.35 Deletion routine for linked lists—cursor implementation

performed, the cursor implementation could be significantly faster because of the lack of memory management routines.

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

```
        /* Insert (after legal position P) */
        /* Header implementation assumed */
        /* Parameter L is unused in this implementation */

        void
        Insert( ElementType X, List L, Position P )
        {
            Position TmpCell;

/* 1*/      TmpCell = CursorAlloc( );
/* 2*/      if( TmpCell == 0 )
/* 3*/          FatalError( "Out of space!!!" );

/* 4*/      CursorSpace[ TmpCell ].Element = X;
/* 5*/      CursorSpace[ TmpCell ].Next = CursorSpace[ P ].Next;
/* 6*/      CursorSpace[ P ].Next = TmpCell;
        }
```

Figure 3.36 Insertion routine for linked lists—cursor implementation

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*, and is the topic of the next section.

3.3. The Stack ADT

3.3.1. *Stack Model*

A *stack* is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the *top*. The fundamental operations on a stack are *Push*, which is equivalent to an insert, and *Pop*, which deletes the most recently inserted element. The most recently inserted element can be examined prior to performing a *Pop* by use of the *Top* routine. A *Pop* or *Top* on an empty stack is generally considered an error in the stack ADT. On the other hand, running out of space when performing a *Push* is an implementation error but not an ADT error.

Stacks are sometimes known as LIFO (last in, first out) lists. The model depicted in Figure 3.37 signifies only that *Push*es are input operations and *Pop*s and *Top*s are output. The usual operations to make empty stacks and test for emptiness are part of the repertoire, but essentially all that you can do to a stack is *Push* and *Pop*.

Figure 3.38 shows an abstract stack after several operations. The general model is that there is some element that is at the top of the stack, and it is the only element that is visible.