COMP2421—DATA STRUCTURES AND ALGORITHMS

Cursor Implementation of Linked Lists

Dr. Radi Jarrar
Department of Computer Science
Birzeit University

Cursor Implementation

- Cursor Implementation of Linked Lists is used in cases where lists are required and pointers are not available as the case of using programming languages like Fortran and Basic.
- 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.

Cursor Space

- This can be done by creating a global array of structures. For any cell in the array, its array index can be used in place of an address.
- This array is called CURSOR_SPACE

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

Cursor – `malloc` and `free`

- Now we want to simulate the malloc and free processes in the CURSOR_SPACE.
- 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.
- A value of 0 for next is the equivalent of a pointer. The initialization of CURSOR_SPACE is a straightforward loop. To perform a malloc, the first element (after the header) is removed from the freelist.

Cursor – `malloc` and `free`

- To perform a free operation, we place the cell at the front of the freelist.
- Notice that if there is no space available, the routine does the correct thing by setting $p = 0$. This indicates that there are no more cells left.

Cursor

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

Cursor – Struct & Cursor Space

```
typedef int List;
typedef int Position;

struct node {
    int element;
    int next;
};

struct node CURSOR_SPACE[ 11 ];
```

Cursor – Initialise Cursor Space

```
void InitializeCursorSpace( void )
{
    int i;

    for( i = 0; i < SpaceSize - 1; i++ )
    {
        CursorSpace[ i ].Next = i + 1;
     }

    CursorSpace[ SpaceSize - 1 ].Next = 0;
}
```

Cursor – Cursor List

```
List CursorList()
{
    List L = CursorAlloc();

    CursorSpace[ L ].Next = 0;

    return L;
}
```

Cursor – Cursor Allocation

```
Position CursorAlloc()
{
    Position P;

    //get the position of the first position in the free list
    P = CursorSpace[ 0 ].Next;
    CursorSpace[ 0 ].Next = CursorSpace[ P ].Next;

    if( P == 0 )
    {
     printf("Out of space!\n");
     return;
    }

    return P;
}
```

Cursor – Make Empty

```
// empties the list that is sent here (the header of the list)
List MakeEmpty( List L )
{
    if( L )
        DeleteList( L );

    L = CursorAlloc();
    if( L == 0 )
        printf( "Out of memory!" );

    CursorSpace[ L ].Next = 0;

    return L;
}
```

Cursor – IsEmpty

```
//returns true if the list is empty
int IsEmpty( List L )
{
    return CursorSpace[ L ].Next == 0; //header
refers to zero
}
```

Cursor – IsLast

```
//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;
}
```

Cursor

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

Cursor – Insert

```
// Insert (after legal position P)
void Insert( int X, List L )
{
    Position TmpCell, P = L;

    TmpCell = CursorAlloc( ); //will allocate the second node
(after the header) in the array. It also makes sure the
CursorSpace[0] to point to a new empty location

    while( P && CursorSpace[P].Next != 0)
      P = CursorSpace[P].Next;

    CursorSpace[ TmpCell ].Element = X;
    CursorSpace[ TmpCell ].Next = CursorSpace[ P ].Next;
    CursorSpace[ P ].Next = TmpCell;
}
```

Cursor – Find

```
//Return Position of X in L; 0 if not found

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

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

    return P;
}
```

Cursor – Find Previous

```
// If X is not found, then Next field of returned value is 0

Position FindPrevious( int X, List L )
{
    Position P;

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

    return P;
}
```

Cursor – Delete

```
/* Assume that the position is legal */

void Delete( int X, List L )
{
    Position P, TmpCell;
    P = FindPrevious( X, L );

    if( !IsLast( P, L ) )
    {
        TmpCell = CursorSpace[ P ].Next;
        CursorSpace[ P ].Next = CursorSpace[ TmpCell ].Next;
        CursorFree( TmpCell );
    }
}
```

Cursor – Cursor Free

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

Cursor – DeleteList

```
void DeleteList( List L )
{
    Position P, Tmp;

    P = CursorSpace[ L ].Next;  /* Header assumed */
    CursorSpace[ L ].Next = 0;
    while( P != 0 )
    {
        Tmp = CursorSpace[ P ].Next;
        CursorFree( P );
        P = Tmp;
    }
}
```

Cursor – PrintList

```
void PrintList( List L )
{
     Position P = L;

     printf("\nSlot \t Element \t Next\n");

     while( P != 0 )
    {
          printf("%3d \t %3d \t %3d\n", P,
CursorSpace[P].Element, CursorSpace[P].Next);
          P = CursorSpace[P].Next; }
}
```