Rajalakshmi Engineering College

Rajalakshmi Nagar, Thandalam, Chennai - 602 105

Department of Computer Science and Engineering

CS19241 - Data Structures

Unit - I

Lecture Notes

(Regulations - 2019)

Prepared by:

B.BHUVANESWARAN

Assistant Professor (SG) / CSE / REC

bhuvaneswaran@rajalakshmi.edu.in

CS19241 - DATA STRUCTURES

Course Pre Requisites:

Students should have knowledge of Fundamentals of data types and programming concepts.

Course Objective:

- To apply the concepts of List ADT in the applications of various linear and nonlinear data structures.
- To demonstrate the understanding of stacks, queues and their applications.
- To analyse the concepts of tree data structure.
- To understand the implementation of graphs and their applications.
- To be able to incorporate various searching and sorting techniques in real time scenarios.

Unit - I - Linear Data Structures – List 9

Abstract Data Types (ADTs) - List ADT - array-based implementation - linked list implementation - singly linked lists - circularly linked lists - doubly-linked lists - applications of lists - Polynomial Manipulation - All operations (Insertion, Deletion, Merge, Traversal).

Unit - II - Linear Data Structure - Stacks, Queues 9

Stack ADT - Operations - Applications - Evaluating arithmetic expressions - Conversion of infix to postfix expression - Queue ADT - Operations - Circular Queue - DEQUE - applications of queues.

Unit - III - Non-linear Data Structure - Trees 9

Tree Terminologies- Binary Tree – Representation - Tree traversals - Expression trees - Binary Search Tree - AVL Trees - Splay Trees - Binary Heap - Applications.

Unit - IV - Non-linear Data Structure - Graphs 9

Graph Terminologies - Representation of Graph - Types of graph - Breadth-first traversal - Depth-first traversal -Topological Sort - Shortest path - Dijkstra's Algorithm - Minimum Spanning Tree - Prim's Algorithm.

Unit - V - Searching, Sorting and Hashing Techniques 9

Searching - Linear Search - Binary Search. Sorting - Bubble sort - Selection sort - Insertion sort - Shell sort - Quick sort - Merge Sort. Hashing - Hash Functions - Collision resolution strategies - Separate Chaining - Open Addressing - Rehashing.

Course Outcomes:

On completion of the course, the students will be able to:

- Analyse the various data structure concepts.
- Implement Stacks and Queue concepts for solving real-world problems.
- Analyse and structure the linear data structure using tree concepts.
- Critically Analyse various non-linear data structures algorithms.
- Apply different Sorting, Searching and Hashing algorithms.

Text Books:

1. Mark Allen Weiss, "Data Structures and Algorithm Analysis in C", 2nd Edition, Pearson Education, 2002.
2. Reema Thareja, "Data Structures Using C", Second Edition, Oxford University Press, 2014.

Reference Books:

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest and Clifford Stein, "Introduction to Algorithms", Second Edition, McGraw Hill, 2002.
2. Aho, Hopcroft and Ullman, "Data Structures and Algorithms", Pearson Education, 1983.
3. Stephen G. Kochan, "Programming in C", 3rd edition, Pearson Education.
4. Ellis Horowitz, Sartaj Sahni and Susan Anderson Freed, "Fundamentals of Data Structures in C", 2ndEdition, University Press, 2008.

Web Links for Virtual Lab:

1. http://vlabs.iitb.ac.in/vlab/labscse.html

CHAPTER - 1 - INTRODUCTION TO DATA STRUCTURES

1.1 INTRODUCTION

Data structure is the way of organizing and storing data in a computer system so that it can be used efficiently.

1.2 CHARACTERISTICS OF DATA STRUCTURES

The following points highlight the need of data structures in computer science:

1. It depicts the logical representation of data in computer memory.
2. It represents the logical relationship between the various data elements.
3. It helps in efficient manipulation of stored data elements.
4. It allows the programs to process the data in an efficient manner.

1.3 APPLICATIONS OF DATA STRUCTURES

Some of the applications of data structures are:

1. Compiler design
2. Operating system
3. Database management system
4. Statistical analysis package
5. Numerical analysis
6. Graphics
7. Artificial intelligence
8. Simulation

1.4 CLASSIFICATION OF DATA STRUCTURES

Data structures are primarily divided into two classes:

- Primitive
- Non-primitive

1.4.1 Primitive Data Structures

Primitive data structures include all the fundamental data structures that can be directly manipulated by machine level instructions.

Some of the common primitive data structures include:

- Integer
- Character
- Real
- Boolean, etc.

1.4.2 Non-primitive Data Structures

Non-primitive data structures refer to all those data structures that are derived from one or more primitive data structures. The objective of creating non-primitive data structures is to form sets of homogeneous or heterogeneous data elements.

Non-primitive data structures are further categorized into two types:

- Linear
- Non-linear

1.4.3 Linear Data Structures

In linear data structures, all the data elements are arranged in a linear or sequential fashion. Examples of linear data structures include:

- Arrays
- Stacks
- Queues
- Linked lists, etc.

1.4.4 Non-linear Data Structures

In non-linear data structures, there is no definite order or sequence in which data elements are arranged. For instance, a non-linear data structure could arrange data elements in a hierarchical fashion. Examples of non-linear data structures are:

- Trees
- Graphs

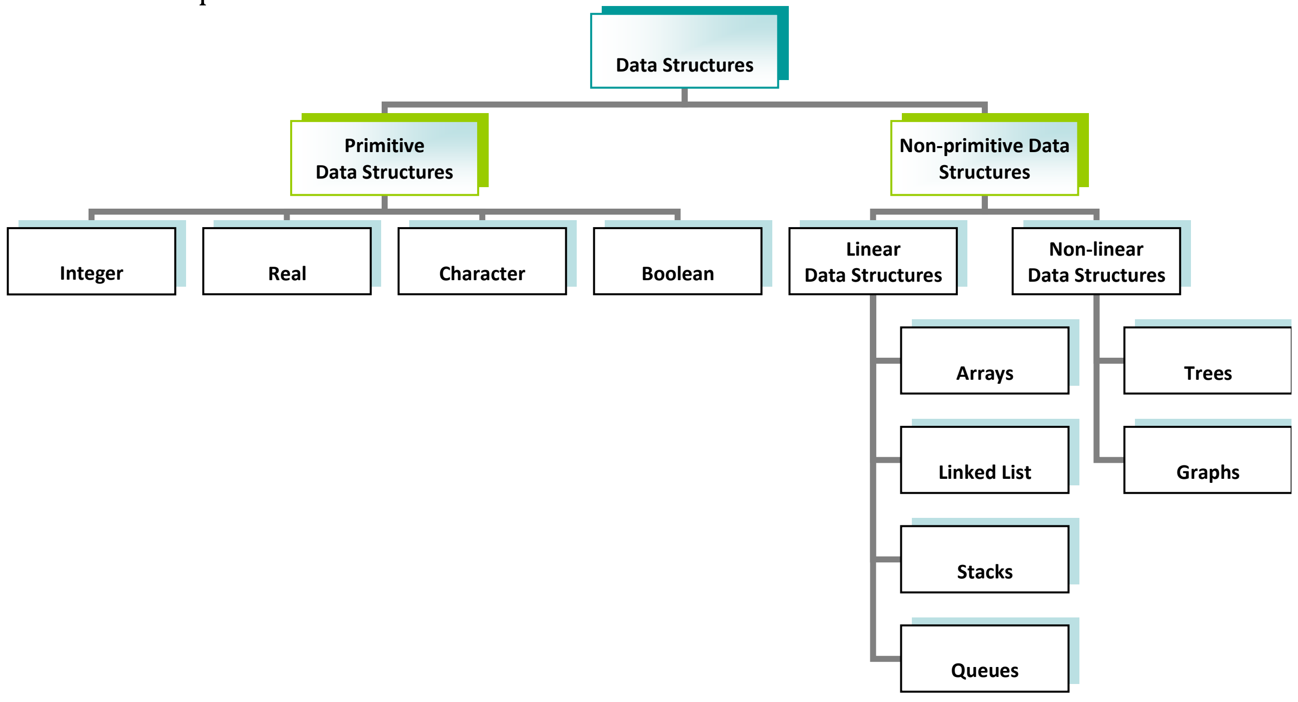


Fig. 1.1 Classification of Data Structures

REVIEW QUESTIONS

1. What is data structure? How is it classified?
2. What are the various characteristics of a data structure?
3. What are primitive data structures?
4. What are non-primitive data structures?
5. What is a linear data structure?
6. What is anon-linear data structure?
7. List out the characteristics of data structure?
8. List out the areas in which data structures are applied.
9. What are the major data structures used in the following areas: RDBMS, Network data model and Hierarchical data model.
10. Write any two data structures used in Operating System?

2.1 ABSTRACT DATA TYPE (ADT)

An abstract data type (ADT) is a set of objects together with a set of operations. Abstract data types are mathematical abstractions. Objects such as lists, sets, and graphs, along with their operations, can be viewed as abstract data types.

2.2 LIST ADT

List is an ordered set of elements. For any list except the empty list, we say that A_{i+1} follows (or succeeds) A_i ($i<N$) and that A_{i-1} precedes A_i ($i>1$). The first element of the list is A_1, and the last element is A_N.

Some popular operations are printList, makeEmpty, find, insert, remove, findKth, next, previous.

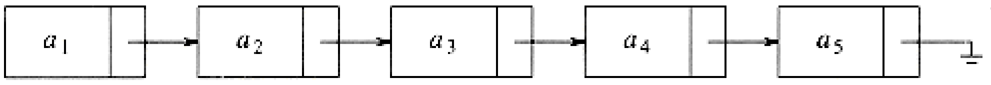


Fig. 2.1. A linked list

2.3 OPERATIONS OF THE LIST ADT

- printList – contents of the list is displayed
- makeEmpty – makes the list empty
- find – returns the position of the first occurrence of an item
- insert – insert some element from some position in the list
- remove – remove some element from some position in the list
- findKth – returns the element in some position (specified as an argument)
- next – return the position of the successor
- previous – return the position of the predecessor

2.4 DIFFERENT WAYS TO IMPLEMENT LIST

Lists are implemented using:

- Array
- Linked list

2.5 ARRAYS

An array implementation allows print_list and find to be carried out in linear time, which is as good as can be expected, and the find_kth operation takes constant time. However, insertion and deletion are expensive. For example, inserting at position 0 (which amounts to making a new first element) requires first pushing the entire array down one spot to make room, whereas deleting the first element requires shifting all the elements in the list up one, so the worst case of these operations is O(n). On average, half the list needs to be moved for either operation, so linear time is still required. Merely building a list by n successive inserts would require quadratic time.

Because the running time for insertions and deletions is so slow and the list size must be known in advance, simple arrays are generally not used to implement lists.

2.6 LINKED LIST

The linked list consists of a series of structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a structure containing its successor. We call this the next pointer. The last cell's next pointer points to NULL.

Data Element	Next Pointer

Node

2.7 ADVANTAGES OF LINKED LISTS

Some of the key advantages of linked lists are:

1. Linked lists facilitate dynamic memory management by allowing elements to be added or deleted at any time during program execution.
2. The use of linked lists ensures efficient utilization of memory space as only that much amount of memory space is reserved as is required for storing the list elements.
3. It is easy to insert or delete elements in a linked list, unlike arrays, which require shuffling of other elements with each insert and delete operation.

2.8 DISADVANTAGES OF LINKED LISTS

Apart from the advantages, linked lists also possess certain limitations, which are:

1. A linked list element requires more memory space in comparison to an array element because it has to also store the address of the next element in the list.
2. Accessing an element is a little more difficult in linked lists than arrays because unlike arrays, there is no index identifier associated with each list element. Thus, to access a linked list element, it is mandatory to traverse all the preceding elements.

2.9 TYPES OF LINKED LIST

Depending on the manner in which its nodes are interconnected with each other, linked lists are categorized into the following types:

- Singly linked list
- Doubly linked list
- Circular linked list

REVIEW QUESTIONS

1. What is an Abstract Data Type (ADT)? (or) Define Abstract Data Type (ADT). (or) Define ADT and give an example. (or) What do you mean by abstract data type? (or) What is an abstract data type? Give any two examples. (or) Give an example for abstract data type. (or) Define abstract data type. List out few.
2. What is advantage of an ADT?
3. Define a 'list'; Mention any two operations that are performed on a list. (or) Define List Abstract Data Type with example.
4. List out the operations of the list ADT. (or) Which operations are supported by the list ADT?
5. What are the different ways to implement list?
6. What are the advantages of linked list over arrays?
7. What are the disadvantages of linked list?
8. List the various types of linked lists.

3.1 INTRODUCTION

Array is a collection of specific number of data stored in consecutive memory locations.

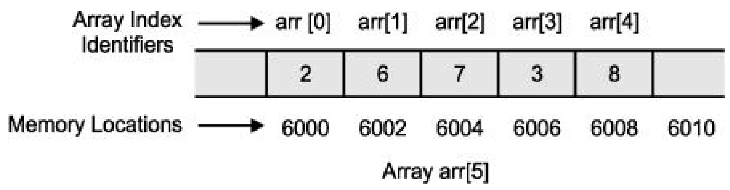


Fig. 3.1. Array example

3.2 OPERATIONS ON ARRAY

- Insert
- Delete
- Traversal
- Sorting
- Searching

3.3 INSERTION

Insertion is the task of adding an element into an existing array. If an element is to be inserted at the end of the array, then it can be simply achieved by storing the new element one position to the right of the last element. Alternatively, if an element is required to be inserted at the middle, then this will require all the subsequent elements to be moved one place to the right.

Example

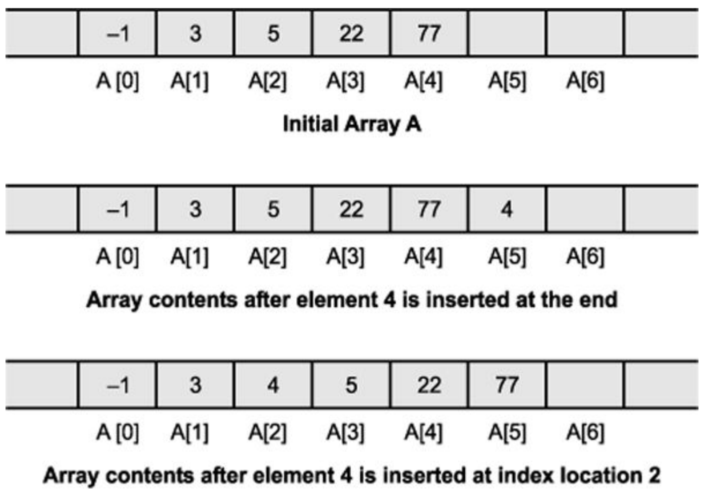


Fig. 3.2. Array insertion

Algorithm to Insert an Element in an Array

```
Insert(a[], p, e)

Step 1 : Start.
Step 2 : Set i = n.
Step 3 : Repeat Steps 4 to 5 while i >= p.
Step 4 : Set a[i+1] = a[i].
Step 5 : Set i  = i – 1.
Step 6 : Set a[p] = e.
Step 7 : Set n = n + 1.
Step 8 : Stop.
```

Routine to Insert an Element in an Array

```
void Insert(int a[], int p, int e)
{
	int i;
	for(i = n; i >= p; i--)
		a[i + 1] = a[i];
	a[p] = e;
	n = n + 1;
}
```

3.4 DELETION

Deletion is the task of removing an element from the array. The deletion of element from the end is quite simple and can be achieved by mere updation of index identifier. However, to remove an element from the middle, one must move all the elements present to the right of the point of deletion, one position to the left.

Example

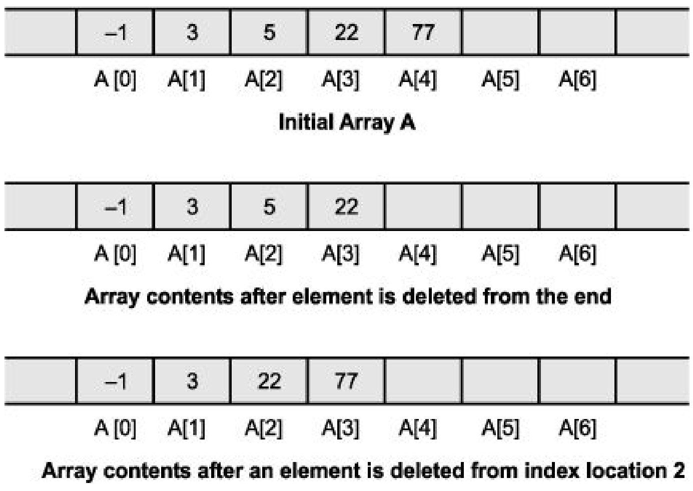


Fig. 3.3. Array deletion

Algorithm to Delete an Element from an Array

```
Delete(a[], p)
Step 1 : Start.
Step 2 : Set i = p.
Step 3 : Repeat Steps 4 to 5 while i < n.
Step 4 : Set a[i] = a[i+1].
Step 5 : Set i  = i + 1.
Step 6 : Set n = n - 1.
Step 7 : Stop.
```

Routine to Delete an Element from an Array

```c
void Delete(int a[], int p)
{
      int i;
      for(i = p; i < n; i++)
            a[i] = a[i + 1];
      n = n - 1;
}
```

3.5 TRAVERSAL

While working with arrays, it is often required to access the array elements; that is, reading values from the array. This is achieved with the help of array traversal. It involves visiting the array elements and storing or retrieving values from it.

Some of the typical situations where array traversal may be required are:

- Printing array elements
- Searching an element in the array
- Sorting an array

Algorithm to Traverse an Array

```
Traverse(a[])
Step 1 : Start.
Step 2 : Set i = 0.
Step 3 : Repeat Steps 4 to 5 while i < n.
Step 4 : Access a[i].
Step 5 : Set i  = i + 1.
Step 6 : Stop.
```

Routine to Traverse an Array

```c
void Traverse(int a[])
{
      int i;
      for(i = 0; i < n; i++)
            printf("%d\t", a[i]);
}
```

3.6 SEARCHING

Searching is the process of traversing an array to find out if a specific element is present in the array or not. If the search is successful, the index location of the element is returned.

Example

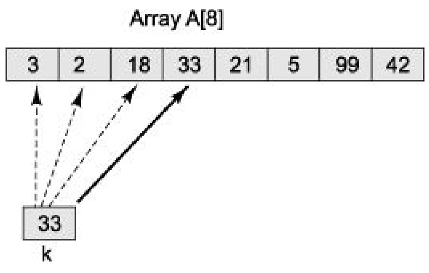


Fig. 3.4. Array Searching

Algorithm to Search an element in an Array

```
Search(a[], e)

Step 1  : Start.
Step 2  : Set i = 0.
Step 3  : Repeat Steps 4 to 6 while i < n.
Step 4  : if e = a[i] goto Step 5 else goto Step 6.
Step 5  : Set flag = 1 and goto Step 7.
Step 6  : Set i = i + 1.
Step 7  : if flag = 1 goto Step 8 else goto Step 9.
Step 8  : Print i and goto step 10.
Step 9  : Print i.
Step 10: Print "Unsuccesful.".
Step 11: Stop.
```

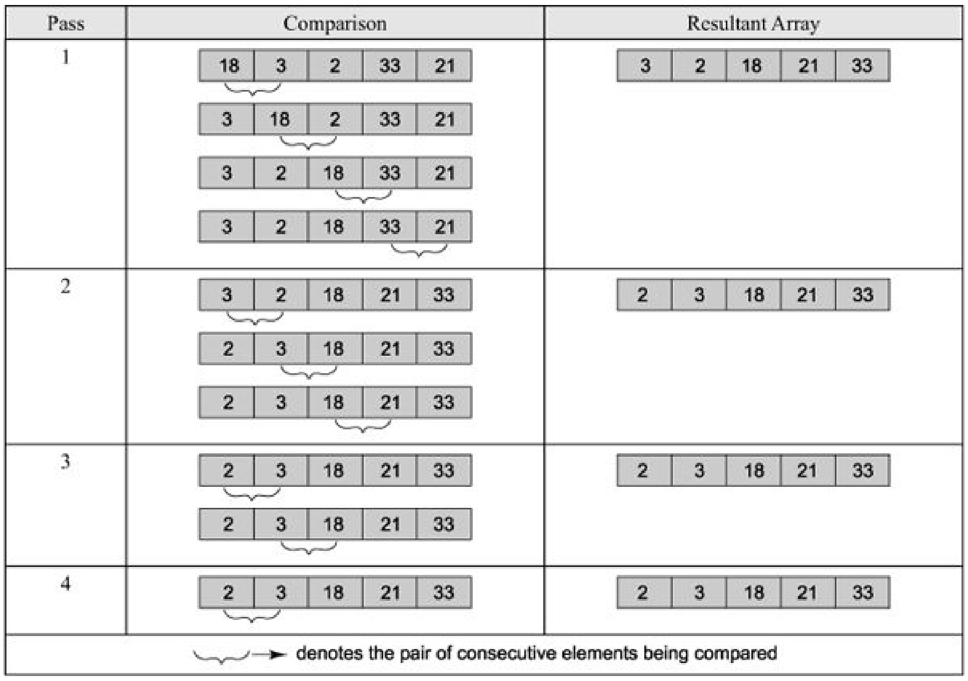
Routine to Search an element in an Array

```
void Search(int a[], int e)
{
      int i, flag = 0;
      for(i = 0; i < n; i++)
      {
            if(e == a[i])
            {
                  flag = 1;
                  break;
            }
      }
      if(flag == 1)
            printf("Successful. Element %d is at location %d",e,i);
      else
            printf("Unsuccessful.");
}
```

3.7 SORTING

The sorting operation arranges the elements of an array in a specific order or sequence. Sorting involves comparing the array elements with each other and shuffling them until all the elements are sorted.

Example



Pass	Comparison	Resultant Array
1	18 | 3 | 2 | 33 | 21	3 | 2 | 18 | 21 | 33
	3 | 18 | 2 | 33 | 21	
	3 | 2 | 18 | 33 | 21	
	3 | 2 | 18 | 33 | 21	
2	3 | 2 | 18 | 21 | 33	2 | 3 | 18 | 21 | 33
	2 | 3 | 18 | 21 | 33	
	2 | 3 | 18 | 21 | 33	
3	2 | 3 | 18 | 21 | 33	2 | 3 | 18 | 21 | 33
	2 | 3 | 18 | 21 | 33	
4	2 | 3 | 18 | 21 | 33	2 | 3 | 18 | 21 | 33

denotes the pair of consecutive elements being compared

Fig. 3.5. Array Sorting

Algorithm to Sort the elements in an Array

```
Sort(a[])

Step 1  : Start.
Step 2  : Set i = 0.
Step 3  : Repeat Steps 4 to 11 while i < n-1.
Step 4  : Set j = i+1.
Step 5  : Repeat Steps 6 to 10 while j < n.
Step 6  : if a[i] > a[j] goto Step 7 else goto Step 10.
Step 7  : Set t = a[i].
Step 8  : Set a[i] = a[j].
Step 9  : Set a[j] = t.
Step 10: Set j = j + 1.
Step 11: Set i = i + 1.
Step 12: Stop.
```

Routine to Sort the elements in an Array

```
void Sort(int a[])
{
      int i, j, t;
      for(i = 0; i < n-1; i++)
      {
            for(j = i + 1; j < n; j++)
            {
                  if(a[i] > a[j])
                  {
                        t = a[i];
                        a[i] = a[j];
                        a[j] = t;
                  }
            }
      }
}
```

3.8 PROGRAM

```
#include <stdio.h>

int n = 0;

void Insert(int a[], int p, int e);
void Delete(int a[], int p);
void Search(int a[], int e);
void Traverse(int a[]);
void Sort(int a[]);

int main()
{
      int a[5], ch, e, p;
      printf("1.Insert \n2.Delete \n3.Search");
      printf("\n4.Traverse \n5.Sort \n6.Exit\n");
      do
      {
            printf("\nEnter your choice : ");
            scanf("%d", &ch);
            switch(ch)
            {
                  case 1:
                        printf("Enter the position : ");
                        scanf("%d", &p);
                        printf("Enter the element : ");
                        scanf("%d", &e);
                        Insert(a, p, e);
                        break;
```

```
                case 2:
                     printf("Enter the position : ");
                     scanf("%d", &p);
                     Delete(a, p);
                     break;
                case 3:
                     printf("Enter the element : ");
                     scanf("%d", &e);
                     Search(a, e);
                     break;
                case 4:
                     printf("The elements are : ");
                     Traverse(a);
                     break;
                case 5:
                     Sort(a);
                     break;
           }
      } while(ch <= 5);
      return 0;
}

void Insert(int a[], int p, int e)
{
      int i;
      for(i = n; i >= p; i--)
           a[i + 1] = a[i];
      a[p] = e;
      n = n + 1;
}

void Delete(int a[], int p)
{
      int i;
      for(i = p; i < n; i++)
           a[i] = a[i + 1];
      n = n - 1;
}

void Search(int a[], int e)
{
      int i, flag = 0;
      for(i = 0; i < n; i++)
      {
           if(e == a[i])
           {
                flag = 1;
                break;
           }
      }
```

```
        if(flag == 1)
                printf("Successful. Element %d is at location %d", e, i);
        else
                printf("Unsuccessful.");
}

void Traverse(int a[])
{
        int i;
        for(i = 0; i < n; i++)
                printf("%d\t", a[i]);
}

void Sort(int a[])
{
        int i, j, t;
        for(i = 0; i < n-1; i++)
        {
                for(j = i + 1; j < n; j++)
                {
                        if(a[i] > a[j])
                        {
                                t = a[i];
                                a[i] = a[j];
                                a[j] = t;
                        }
                }
        }
}
```

OUTPUT

```
1.Insert
2.Delete
3.Search
4.Traverse
5.Sort
6.Exit

Enter your choice : 1
Enter the position : 0
Enter the element : 10

Enter your choice : 4
The elements are : 10
Enter your choice : 1
Enter the position : 0
Enter the element : 20
Enter your choice : 4
The elements are : 20    10

Enter your choice : 1
Enter the position : 1
Enter the element : 25

Enter your choice : 4
The elements are : 20    25      10
Enter your choice : 2
Enter the position : 1

Enter your choice : 4
The elements are : 20    10
Enter your choice : 3
Enter the element : 10
Successful. Element 10 is at location 1

Enter your choice : 3
Enter the element : 25
Unsuccessful.
Enter your choice : 5

Enter your choice : 4
The elements are : 10    20
Enter your choice : 6
```

REVIEW QUESTIONS

1. What is an array?
2. What are the two basic operations on arrays?
3. What are the limitations of list implemented by array?

4.1 INTRODUCTION

A singly linked list is a linked list in which each node contains only one link field pointing to the next node in the list.

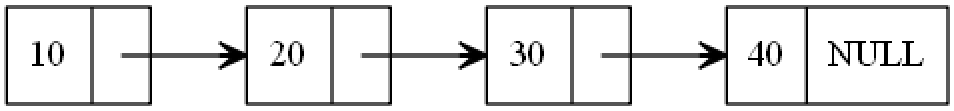


Fig. 4.1 Singly Linked List

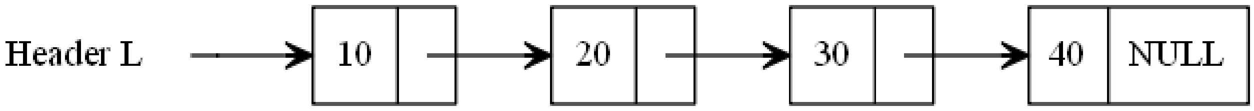


Fig. 4.2 Singly Linked List with Header

4.2 TYPE DECLARATIONS FOR SINGLY LINKED LIST

```
struct node
{
	int Element;
	struct node *Next;
};
typedef struct node Node;
```

4.3 EMPTY LIST WITH HEADER

Example

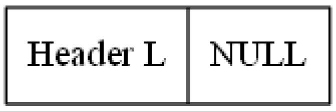


Fig. 4.3 Empty List with Header

Algorithm

```
IsEmpty(List)

Step 1  : Start.
Step 2  : If List→Next = NULL goto Step 3 else goto Step 4.
Step 3  : Return 1 and Stop.
Step 4  : Return 0.
Step 5  : Stop.
```

Routine

```
int IsEmpty(Node *List)
{
	if(List->Next == NULL)
		return 1;
	else
		return 0;
}
```

4.4 CURRENT POSITION IS LAST

Example

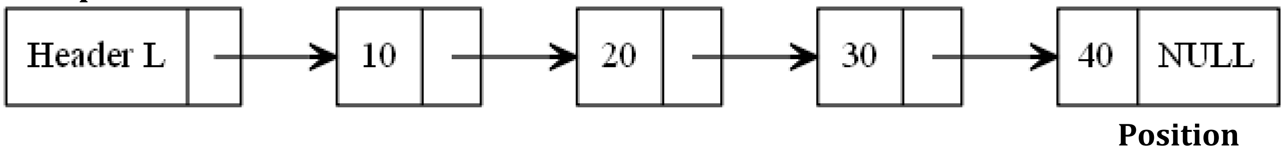


Fig. 4.4 Current Position is the Last in a Linked List

Algorithm

```
IsLast(Position)

Step 1 : Start.
Step 2 : If Position→Next = NULL goto Step 3 else goto Step 4.
Step 3 : Return 1 and Stop.
Step 4 : Return 0.
Step 5 : Stop.
```

Routine to Check whether the Current Position is Last

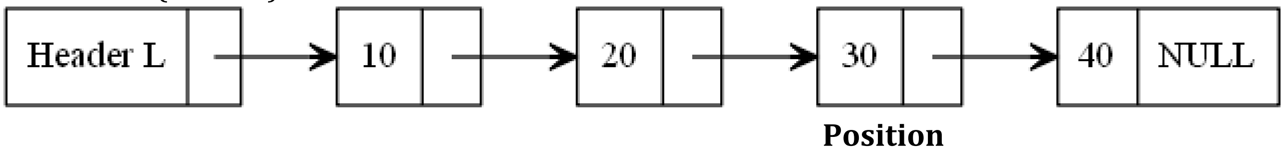
```
int IsLast(Node *Position)
{
	if(Position->Next == NULL)
		return 1;
	else
		return 0;
}
```

4.5 FIND

4.5.1 Find

Example

Find(List, 30)



Algorithm

```
Find(List, x)

Step 1 : Start.
Step 2 : Set Position = List→Next.
Step 3 : Repeat the Step 4 until Position != NULL and Position→Element != x.
Step 4 : Set Position = Position→Next.
Step 5 : Return Position.
Step 6 : Stop.
```

Routine

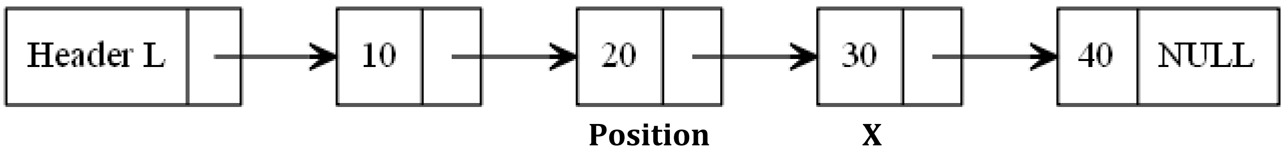
```
Node *Find(Node *List, int x)
{
      Node *Position;
      Position = List->Next;
      while(Position != NULL && Position->Element != x)
                  Position = Position->Next;
      return Position;
}
```

4.5.2 FindPrevious

To avoid the problems associated with deletions, we need to write a routine FindPrevious, which will return the position of the predecessor of the cell we wish to delete. If we use a header, then if we wish to delete the first element in the list, FindPrevious will return the position of the header.

Example

FindPrevious(List, 30)



Algorithm

```
FindPrevious(List, x)

Step 1 : Start.
Step 2 : Set Position = List.
Step 3 : Repeat the Step 4 until Position→Next != NULL and Position→Next→Element
!= x.
Step 4 : Set Position = Position→Next.
Step 5 : Return Position.
Step 6 : Stop.
```

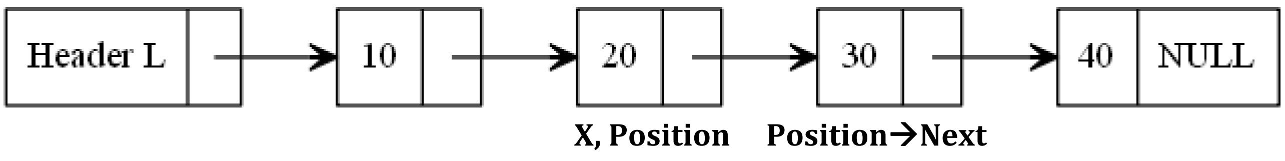
Routine

```
Node *FindPrevious(Node *List, int x)
{
      Node *Position;
      Position = List;
      while(Position->Next != NULL && Position->Next->Element != x)
                  Position = Position->Next;
      return Position;
}
```

4.5.3 FindNext

Example

FindNext(List, 20)



Algorithm

FindNext(List, x)

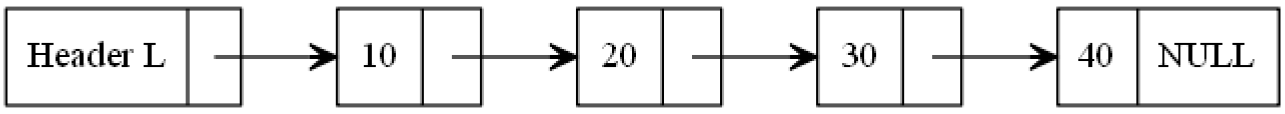
```
Step 1 : Start.
Step 2 : Set Position = Find(List, x).
Step 3 : Return Position→Next.
Step 4 : Stop.
```

Routine

```
Node *FindNext(Node *List, int x)
{
      Node *Position;
      Position = Find(List, x);
      return Position->Next;
}
```

4.6 TRAVERSE THE LIST

Example



Algorithm

```
Traverse(List)

Step 1 : Start.
Step 2 : If !IsEmpty = TRUE goto Step 3 else goto Step 8.
Step 3 : Set Position = List.
Step 4 : Repeat the Steps 5-6 until Position→Next != NULL.
Step 5 : Set Position = Position→Next.
Step 6 : Display Position→Element.
Step 7 : Goto Step 9.
Step 8 : Display "List is empty".
Step 9 : Stop.
```

Routine

```c
void Traverse(Node *List)
{
	if(!IsEmpty(List))
	{
		Node *Position;
		Position = List;
		while(Position->Next != NULL)
		{
			Position = Position->Next;
			printf("%d\t", Position->Element);
		}
		printf("\n");
	}
	else
		printf("List is empty...!");
}
```

4.7 INSERT

The insert command requires obtaining a new cell from the system by using an malloc call and then executing two pointer maneuvers.

We will pass an element to be inserted along with the list L and a position P. Our particular insertion routine will insert an element after the position implied by P. This decision is arbitrary and meant to show that there are no set rules for what insertion does. It is quite possible to insert the new element into position P (which means before the element currently in position p), but doing this requires knowledge of the element before position P. This could be obtained by a call to Find.

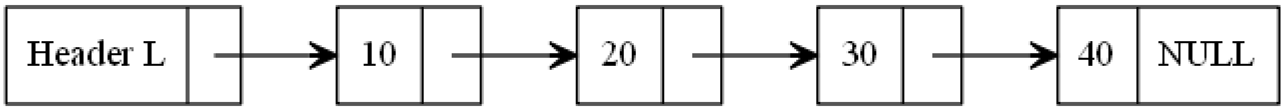
Insertion

- Insert an element at the beginning
- Insert an element at the end
- Insert an element in the middle

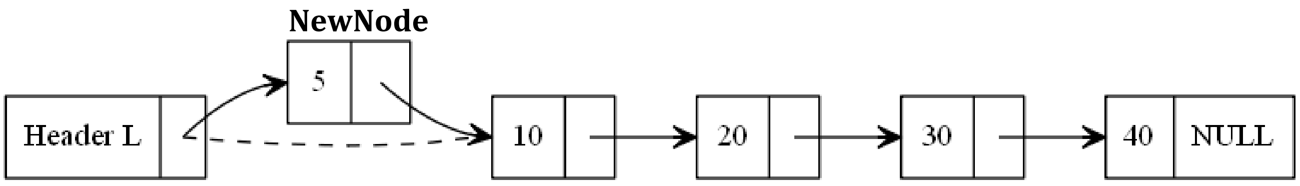
4.7.1 Insert an Element at the Beginning

Example

The general idea is shown in Figure. The dashed line represents the old pointer.



InsertBeg(List, 5)



Algorithm

InsertBeg(List, e)

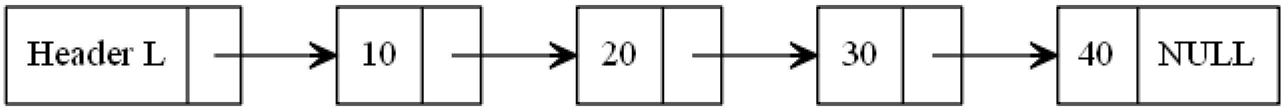
```
Step 1  : Start.
Step 2  : Set NewNode = addressof(Node).
Step 3  : Set NewNode→Element = e.
Step 4  : If List = NULL, then goto Step 5 else goto Step 6.
Step 5  : Set NewNode→Next = NULL and goto Step 7.
Step 6  : Set NewNode→Next = List→Next.
Step 7  : Set List→Next = NewNode.
Step 8: Stop.
```

Routine

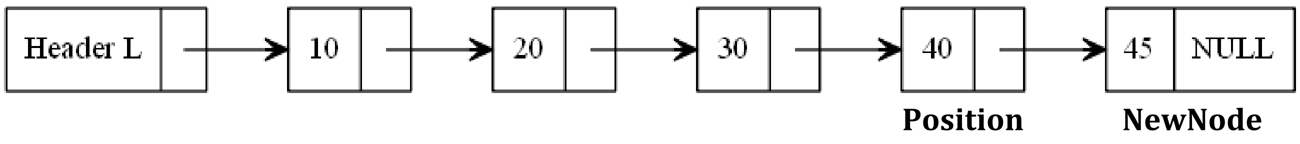
```c
void InsertBeg(Node *List, int e)
{
      Node *NewNode = malloc(sizeof(Node));
      NewNode->Element = e;
      if(IsEmpty(List))
            NewNode->Next = NULL;
      else
            NewNode->Next = List->Next;
      List->Next = NewNode;
}
```

4.7.2 Insert an Element at the End

Example



InsertLast(List, 45)



Algorithm

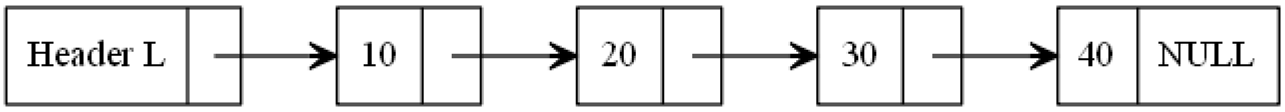
```
InsertLast(List, e)
Step 1  : Start.
Step 2  : Set NewNode = addressof(Node).
Step 3  : Set NewNode→Element = e.
Step 4  : Set NewNode→Next = NULL.
Step 5  : If List = NULL, then goto Step 6 else goto Step 7.
Step 6  : Set List→Next = NewNode and goto Step 11.
Step 7  : Set Position = List.
Step 8  : Repeat the Step 9 until Position→Next != NULL.
Step 9  : Set Position = Position→Next.
Step 10: Set Position→Next = NewNode.
Step 11: Stop.
```

Routine

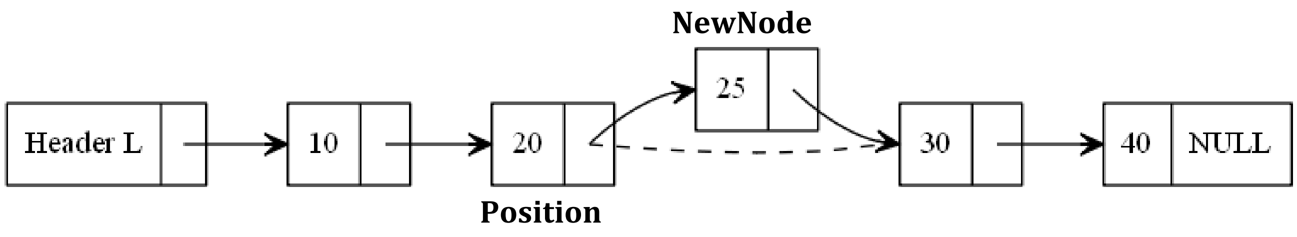
```
void InsertLast(Node *List, int e)
{
      Node *NewNode = malloc(sizeof(Node));
      Node *Position;
      NewNode->Element = e;
      NewNode->Next = NULL;
      if(IsEmpty(List))
            List->Next = NewNode;
      else
      {
            Position = List;
            while(Position->Next != NULL)
                  Position = Position->Next;
            Position->Next = NewNode;
      }
}
```

4.7.3 Insert an Element in the Middle

Example



InsertMid(List, 20, 25)



Algorithm

```
InsertMid(List, p, e)

Step 1 : Start.
Step 2 : Set NewNode = addressof(Node).
Step 3 : Set Position = Find(List, p).
Step 4 : Set NewNode→Element = e.
Step 5 : Set NewNode→Next = Position→Next.
Step 6 : Set Position→Next = NewNode.
Step 7 : Stop.
```

Routine

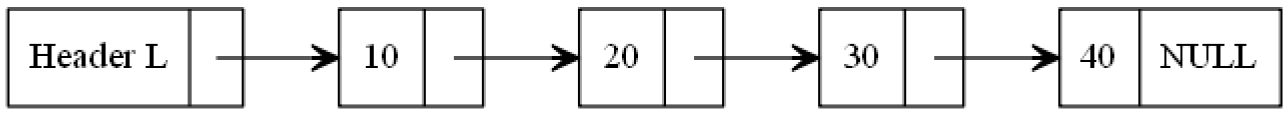
```
void InsertMid(Node *List, int p, int e)
{
     Node *NewNode = malloc(sizeof(Node));
     Node *Position;
     Position = Find(List, p);
     NewNode->Element = e;
     NewNode->Next = Position->Next;
     Position->Next = NewNode;
}
```

4.8 DELETE

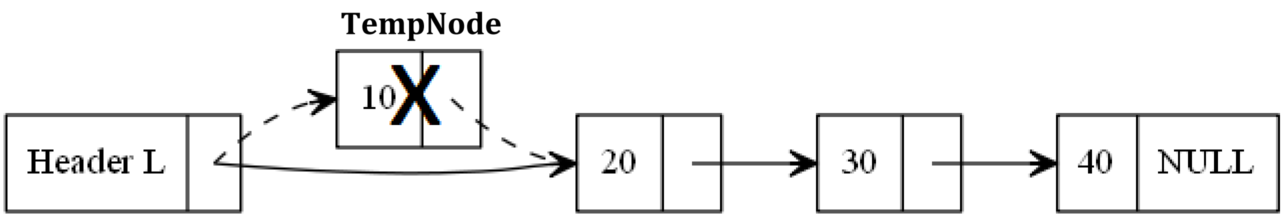
The delete command can be executed in one pointer change. Our routine will delete some element X in list L. We need to decide what to do if x occurs more than once or not at all. Our routine deletes the first occurrence of x and does nothing if x is not in the list. To do this, we find p, which is the cell prior to the one containing x, via a call to FindPrevious.

4.8.1 Delete an Element from the Beginning

Example



DeleteBeg(List)



Algorithm

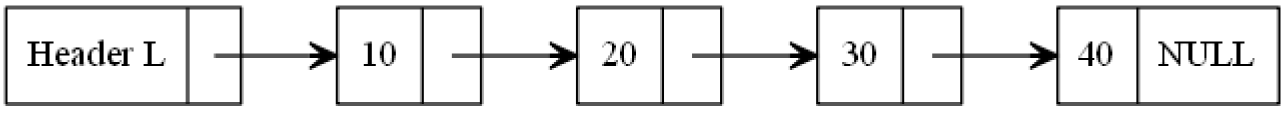
```
DeleteBeg(List, e)
Step 1 : Start.
Step 2 : If !IsEmpty = True, then goto Step 3 else goto Step 7.
Step 3 : Set TempNode = List→Next.
Step 4 : Set List→Next = TempNode→Next.
Step 5 : Display the TempNode→Element.
Step 6 : Delete TempNode and goto Step 8.
Step 7 : Display "List is Empty".
Step 8: Stop.
```

Routine

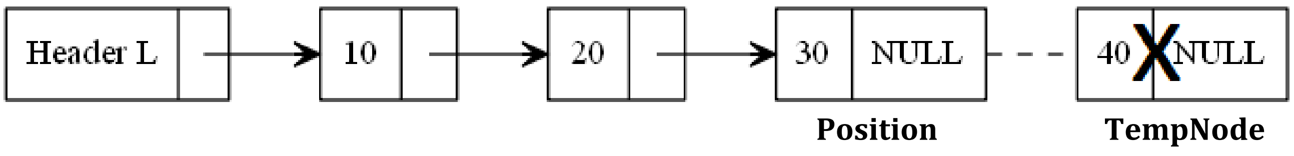
```c
void DeleteBeg(Node *List)
{
	if(!IsEmpty(List))
	{
		Node *TempNode;
		TempNode = List->Next;
		List->Next = TempNode->Next;
		printf("The deleted item is %d\n", TempNode->Element);
		free(TempNode);
	}
	else
		printf("List is empty...!\n");
}
```

4.8.2 Delete an Element from the End

Example



DeleteEnd(List)



Algorithm

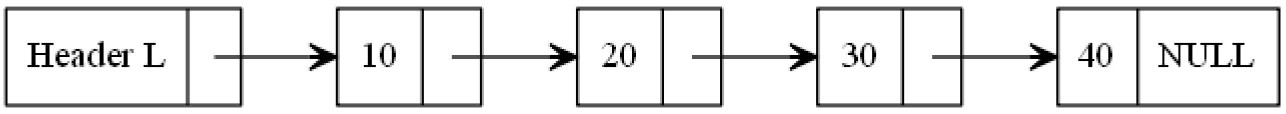
```
DeleteEnd(List)

Step 1 : Start.
Step 2 : If !IsEmpty = True, then goto Step 3 else goto Step 10.
Step 3 : Set Position = List.
Step 4 : Repeat the Step 5 until Position→Next != NULL.
Step 5 : Set Position = Position→Next.
Step 6 : Set TempNode = Position→Next.
Step 7 : Set Position→Next = NULL.
Step 8 : Display TempNode→Element.
Step 9 : Delete TempNode and goto Step 11.
Step 10: Display "List is Empty".
Step 11: Stop.
```

Routine

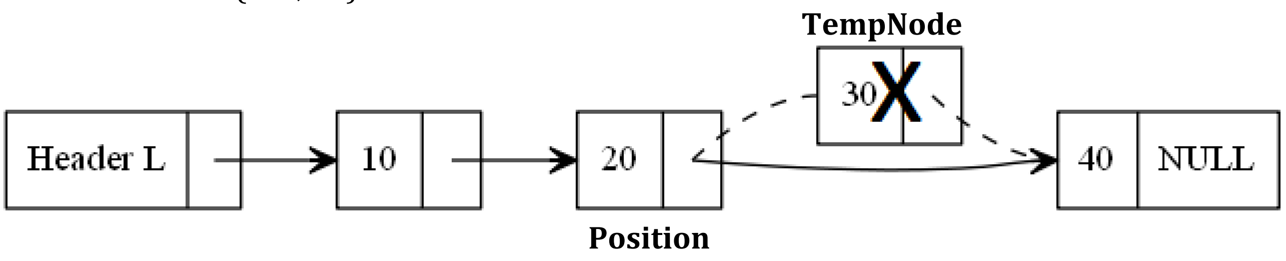
```c
void DeleteEnd(Node *List)
{
      if(!IsEmpty(List))
      {
            Node *Position;
            Node *TempNode;
            Position = List;
            while(Position->Next->Next != NULL)
                  Position = Position->Next;
            TempNode = Position->Next;
            Position->Next = NULL;
            printf("The deleted item is %d\n", TempNode->Element);
            free(TempNode);
      }
      else
            printf("List is empty...!\n");
}
```

4.8.3 Delete an Element from the Middle

Example



DeleteMid(List, 30)



Algorithm

```
DeleteMid(List, e)
Step 1  : Start.
Step 2  : If !IsEmpty = True, then goto Step 3 else goto Step 9.
Step 3  : Set Position = FindPrevious(List, e).
Step 4  : If !Islast(Position) = True, then goto Step 5 else goto Step 10.
Step 5  : Set TempNode = Position→Next.
Step 6  : Set Position→Next = TempNode→Next.
Step 7  : Display the TempNode→Element.
Step 8  : Delete TempNode and goto Step 10.
Step 9  : Display "List is Empty".
Step 10: Stop.
```

Routine

```
void DeleteMid(Node *List, int e)
{
      if(!IsEmpty(List))
      {
            Node *Position;
            Node *TempNode;
            Position = FindPrevious(List, e);
            if(!IsLast(Position))
            {
                  TempNode = Position->Next;
                  Position->Next = TempNode->Next;
                  printf("The deleted item is %d\n", TempNode->Element);
                  free(TempNode);
            }
      }
      else
            printf("List is empty...!\n");
}
```

4.9. PROGRAM

```
/* Implementation of singly linked list - SLL.C */

#include <stdio.h>
#include <stdlib.h>

struct node
{
      int Element;
      struct node *Next;
};
typedef struct node Node;

int IsEmpty(Node *List);
int IsLast(Node *Position);
```

```
Node *Find(Node *List, int x);
Node *FindPrevious(Node *List, int x);
Node *FindNext(Node *List, int x);
void InsertBeg(Node *List, int e);
void InsertLast(Node *List, int e);
void InsertMid(Node *List, int p, int e);
void DeleteBeg(Node *List);
void DeleteEnd(Node *List);
void DeleteMid(Node *List, int e);
void Traverse(Node *List);

int main()
{
	Node *List = malloc(sizeof(Node));
	List->Next = NULL;
	Node *Position;
	int ch, e, p;
	printf("1.Insert Beg \n2.Insert Middle \n3.Insert End");
	printf("\n4.Delete Beg \n5.Delete Middle \n6.Delete End");
	printf("\n7.Find \n8.Traverse \n9.Exit\n");
	do
	{
		printf("Enter your choice : ");
		scanf("%d", &ch);
		switch(ch)
		{
			case 1:
				printf("Enter the element : ");
				scanf("%d", &e);
				InsertBeg(List, e);
				break;
			case 2:
				printf("Enter the position element : ");
				scanf("%d", &p);
				printf("Enter the element : ");
				scanf("%d", &e);
				InsertMid(List, p, e);
				break;
			case 3:
				printf("Enter the element : ");
				scanf("%d", &e);
				InsertLast(List, e);
				break;
			case 4:
				DeleteBeg(List);
				break;
			case 5:
				printf("Enter the element : ");
				scanf("%d", &e);
				DeleteMid(List, e);
				break;
```

```
                case 6:
                     DeleteEnd(List);
                     break;
                case 7:
                     printf("Enter the element : ");
                     scanf("%d", &e);
                     Position = Find(List, e);
                     if(Position != NULL)
                          printf("Element found...!\n");
                     else
                          printf("Element not found...!\n");
                     break;
                case 8:
                     Traverse(List);
                     break;
           }
      } while(ch <= 8);

      return 0;
}

int IsEmpty(Node *List)
{
      if(List->Next == NULL)
           return 1;
      else
           return 0;
}
int IsLast(Node *Position)
{
      if(Position->Next == NULL)
           return 1;
      else
           return 0;
}
Node *Find(Node *List, int x)
{
      Node *Position;
      Position = List->Next;
      while(Position != NULL && Position->Element != x)
                Position = Position->Next;
      return Position;
}
Node *FindPrevious(Node *List, int x)
{
      Node *Position;
      Position = List;
      while(Position->Next != NULL && Position->Next->Element != x)
                Position = Position->Next;
      return Position;
}
```

```
Node *FindNext(Node *List, int x)
{
	Node *Position;
	Position = Find(List, x);
	return Position->Next;
}

void InsertBeg(Node *List, int e)
{
	Node *NewNode = malloc(sizeof(Node));
	NewNode->Element = e;

	if(IsEmpty(List))
		NewNode->Next = NULL;
	else
		NewNode->Next = List->Next;
	List->Next = NewNode;
}

void InsertLast(Node *List, int e)
{
	Node *NewNode = malloc(sizeof(Node));
	Node *Position;
	NewNode->Element = e;
	NewNode->Next = NULL;

	if(IsEmpty(List))
		List->Next = NewNode;
	else
	{
		Position = List;
		while(Position->Next != NULL)
			Position = Position->Next;
		Position->Next = NewNode;
	}
}

void InsertMid(Node *List, int p, int e)
{
	Node *NewNode = malloc(sizeof(Node));
	Node *Position;
	Position = Find(List, p);
	NewNode->Element = e;
	NewNode->Next = Position->Next;
	Position->Next = NewNode;
}
```

```
void DeleteBeg(Node *List)
{
	if(!IsEmpty(List))
	{
		Node *TempNode;
		TempNode = List->Next;
		List->Next = TempNode->Next;
		printf("The deleted item is %d\n", TempNode->Element);
		free(TempNode);
	}
	else
		printf("List is empty...!\n");
}

void DeleteEnd(Node *List)
{
	if(!IsEmpty(List))
	{
		Node *Position;
		Node *TempNode;
		Position = List;
		while(Position->Next->Next != NULL)
			Position = Position->Next;
		TempNode = Position->Next;
		Position->Next = NULL;
		printf("The deleted item is %d\n", TempNode->Element);
		free(TempNode);
	}
	else
		printf("List is empty...!\n");
}

void DeleteMid(Node *List, int e)
{
	if(!IsEmpty(List))
	{
		Node *Position;
		Node *TempNode;
		Position = FindPrevious(List, e);
		if(!IsLast(Position))
		{
			TempNode = Position->Next;
			Position->Next = TempNode->Next;
			printf("The deleted item is %d\n", TempNode->Element);
			free(TempNode);
		}
	}
	else
		printf("List is empty...!\n");
}
```

```
void Traverse(Node *List)
{
      if(!IsEmpty(List))
      {
            Node *Position;
            Position = List;
            while(Position->Next != NULL)
            {
                  Position = Position->Next;
                  printf("%d\t", Position->Element);
            }
            printf("\n");
      }
      else
            printf("List is empty...!\n");
}
```

OUTPUT

```
1.Insert Beg
2.Insert Middle
3.Insert End
4.Delete Beg
5.Delete Middle
6.Delete End
7.Find
8.Traverse
9.Exit
Enter your choice : 1
Enter the element : 40
Enter your choice : 1
Enter the element : 30
Enter your choice : 1
Enter the element : 20
Enter your choice : 1
Enter the element : 10
Enter your choice : 8
10      20      30      40
Enter your choice : 7
Enter the element : 30
Element found...!
Enter your choice : 1
Enter the element : 5
Enter your choice : 8
5       10      20      30      40
Enter your choice : 3
Enter the element : 45
Enter your choice : 8
5       10      20      30      40      45
Enter your choice : 2
Enter the position element : 20
Enter the element : 25
Enter your choice : 8
5       10      20      25      30      40      45
Enter your choice : 4
The deleted item is 5
Enter your choice : 8
10      20      25      30      40      45
Enter your choice : 6
The deleted item is 45
Enter your choice : 8
10      20      25      30      40
Enter your choice : 5
Enter the element : 30
The deleted item is 30
Enter your choice : 8
10      20      25      40
Enter your choice : 9
```

REVIEW QUESTIONS

1. What is a singly linked list?
2. Should arrays or linked list be used for the following types of applications? Justify your answer.
   (a) Many search operations in sorted list.
   (b) Many search operations in unsorted list.

CHAPTER - 5 - CIRCULAR LINKED LISTS

5.1 INTRODUCTION

A popular convention is to have the last cell keep a pointer back to the first. This can be done with or without a header (if the header is present, the last cell points to it), and can also be done with doubly linked lists (the first cell's previous pointer points to the last cell). This clearly affects some of the tests, but the structure is popular in some applications.

In circular linked list the pointer of the last node points to the first node. Circular linked list can be implemented as singly linked list and doubly linked list with or without headers.

5.2 IMPLEMENTATION

A singly linked circular list is a linked list in which the last node of the list points to the first node.

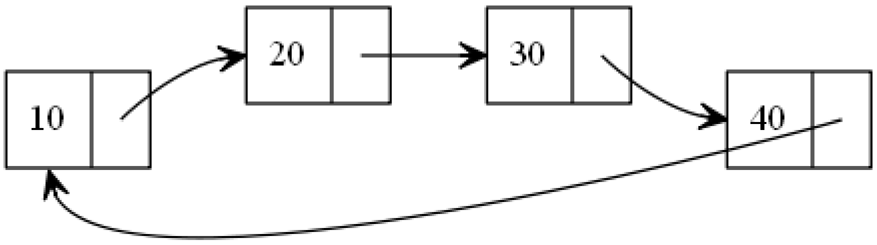


Fig. 5.1 Singly Linked Circular List

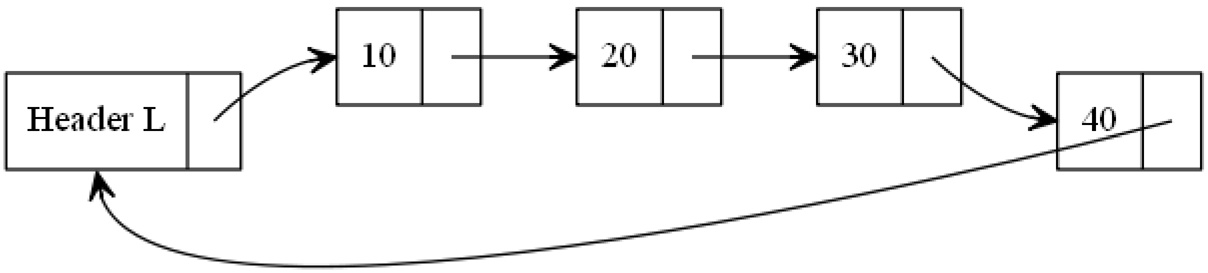


Fig. 5.2. Singly Linked Circular List with Header

5.3 TYPE DECLARATIONS FOR LINKED LIST

```
struct node
{
	int Element;
	struct node *Next;
};
typedef struct node Node;
```

5.4. EMPTY SINGLY LINKED CIRCULAR LIST WITH HEADER

Example

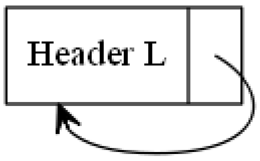


Fig. 5.3 Empty Singly Linked Circular List with Header

Algorithm

```
IsEmpty(List)

Step 1  : Start.
Step 2  : If List→Next = LIST goto Step 3 else goto Step 4.
Step 3  : Return 1 and Stop.
Step 4  : Return 0.
Step 5  : Stop.
```

Routine

```c
int IsEmpty(Node *List)
{
	if(List->Next == List)
		return 1;
	else
		return 0;
}
```

5.5 CURRENT POSITION IS LAST

Example

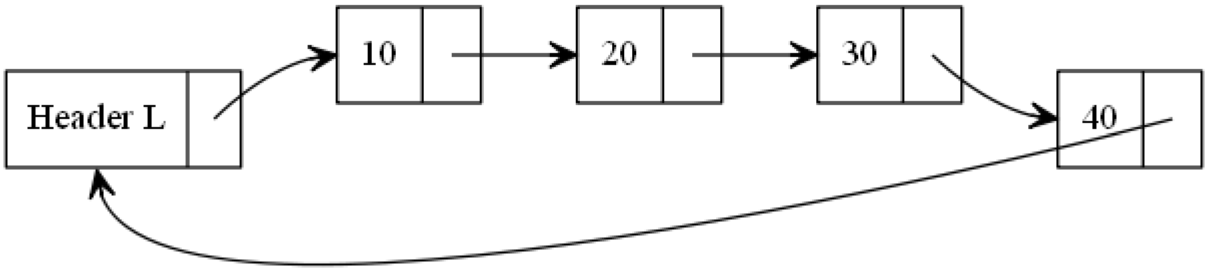


Fig. 5.4 Current Position is the Last in a Singly Linked Circular List

Algorithm

```
IsLast(Position, List)

Step 1  : Start.
Step 2  : If Position→Next = List goto Step 3 else goto Step 4.
Step 3  : Return 1 and Stop.
Step 4 : Return 0.
Step 5 : Stop.
```

Routine to Check whether the Current Position is Last

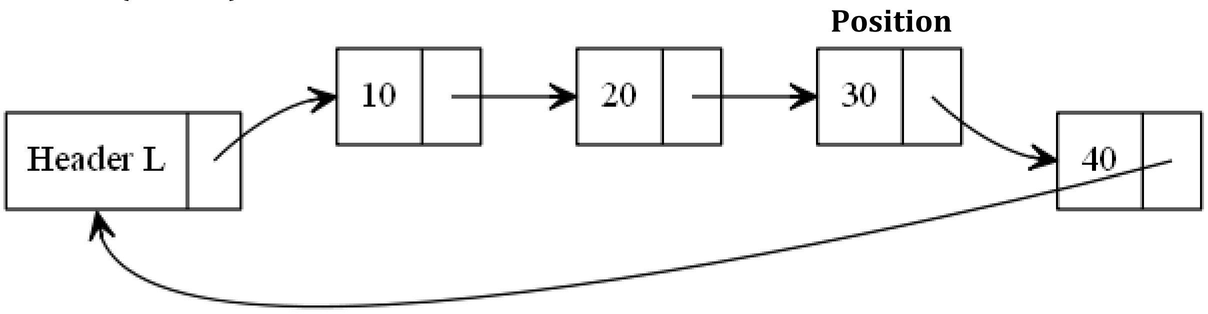
```
int IsLast(Node *Position, Node *List)
{
      if(Position->Next == List)
            return 1;
      else
            return 0;
}
```

5.6 FIND

5.6.1 Find

Example

Find(List, 30)



Algorithm

Find(List, x)

```
Step 1 : Start.
Step 2 : Set Position = List→Next.
Step 3 : Repeat the Step 4 until Position != List and Position→Element != x.
Step 4 : Set Position = Position→Next.
Step 5 : Return Position.
Step 6 : Stop.
```

Routine

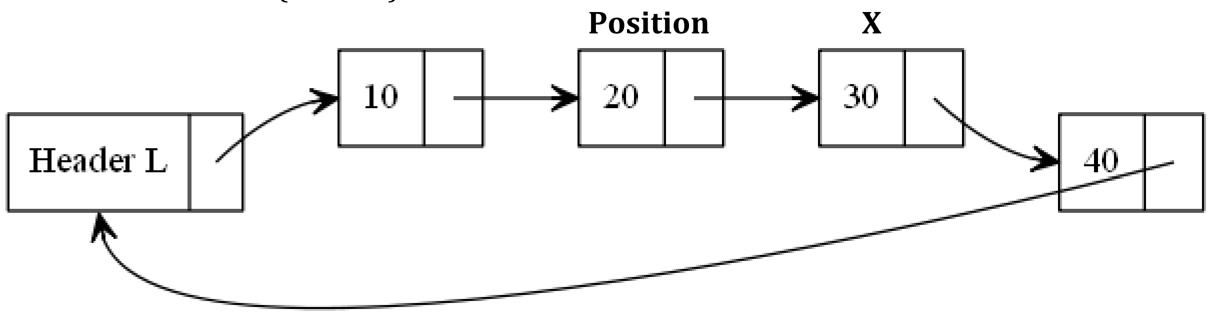
```
Node *Find(Node *List, int x)
{
      Node *Position;
      Position = List->Next;
      while(Position != List && Position->Element != x)
                  Position = Position->Next;
      return Position;
}
```

5.6.2 FindPrevious

To avoid the problems associated with deletions, we need to write a routine FindPrevious, which will return the position of the predecessor of the cell we wish to delete. If we use a header, then if we wish to delete the first element in the list, FindPrevious will return the position of the header.

Example

FindPrevious(List, 30)



Algorithm

FindPrevious(List, x)

```
Step 1 : Start.
Step 2 : Set Position = List.
Step 3 : Repeat the Step 4 until Position→Next != List and
Position→Next→Element!=x.
Step 4 : Set Position = Position→Next.
Step 5 : Return Position.
Step 6 : Stop.
```

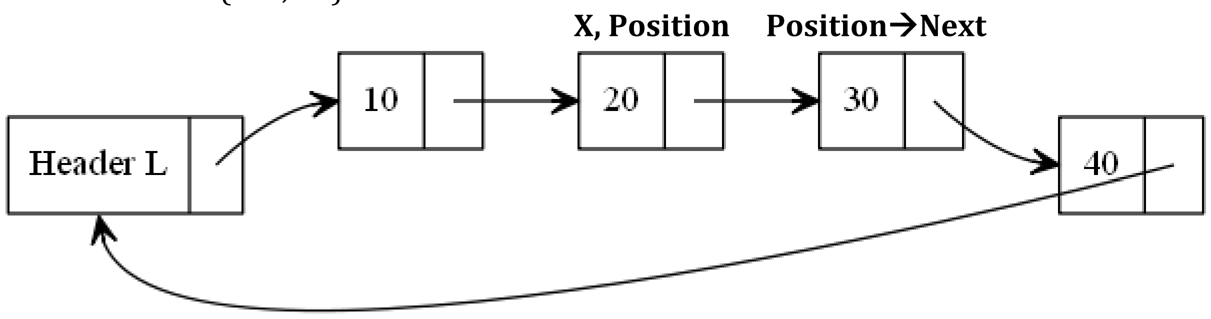
Routine

```c
Node *FindPrevious(Node *List, int x)
{
     Node *Position;
     Position = List;
     while(Position->Next != List && Position->Next->Element != x)
               Position = Position->Next;
     return Position;
}
```

5.6.3 FindNext

Example

FindNext(List, 20)



Algorithm

FindNext(List, x)

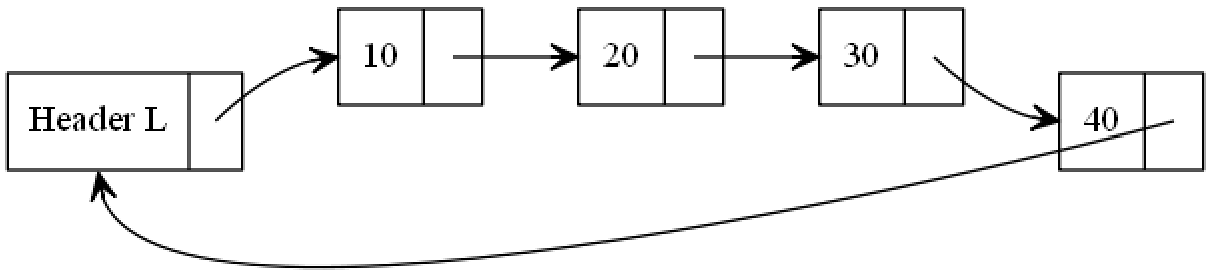
```
Step 1 : Start.
Step 2 : Set Position = Find(List, x).
Step 3 : Return Position→Next.
Step 4 : Stop.
```

Routine

```
Node *FindNext(Node *List, int x)
{
      Node *Position;
      Position = Find(List, x);
      return Position->Next;
}
```

5.7 TRAVERSE THE LIST

Example



Algorithm

Traverse(List)

```
Step 1 : Start.
Step 2 : If !IsEmpty = TRUE goto Step 3 else goto Step 8.
Step 3 : Set Position = List.
Step 4 : Repeat the Steps 5-6 until Position→Next != List.
Step 5 : Set Position = Position→Next.
Step 6 : Display Position→Element.
Step 7 : Goto Step 9.
Step 8 : Display "List is empty".
Step 9 : Stop.
```

Routine

```
void Traverse(Node *List)
{
	if(!IsEmpty(List))
	{
		Node *Position;
		Position = List;
		while(Position->Next != List)
		{
			Position = Position->Next;
			printf("%d\t", Position->Element);
		}
		printf("\n");
	}
	else
		printf("List is empty...!\n");
}
```

5.8 INSERT

The insert command requires obtaining a new cell from the system by using an malloc call and then executing two pointer maneuvers.

We will pass an element to be inserted along with the list L and a position P. Our particular insertion routine will insert an element after the position implied by P. This decision is arbitrary and meant to show that there are no set rules for what insertion does. It is quite possible to insert the new element into position P (which means before the element currently in position p), but doing this requires knowledge of the element before position P. This could be obtained by a call to Find.

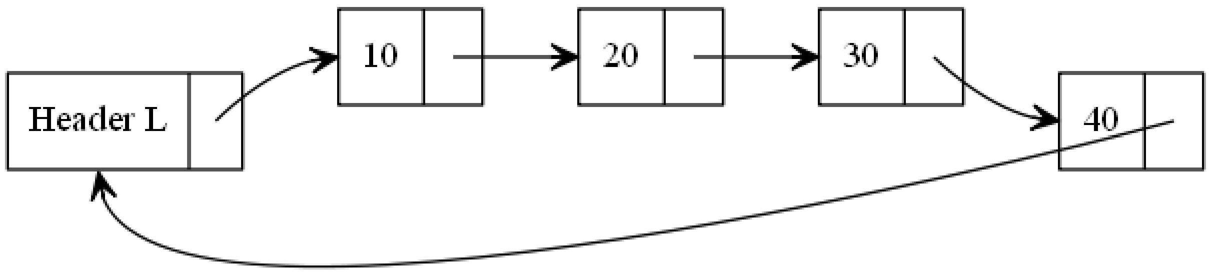
Insertion

- Insert an element at the beginning
- Insert an element at the end
- Insert an element in the middle

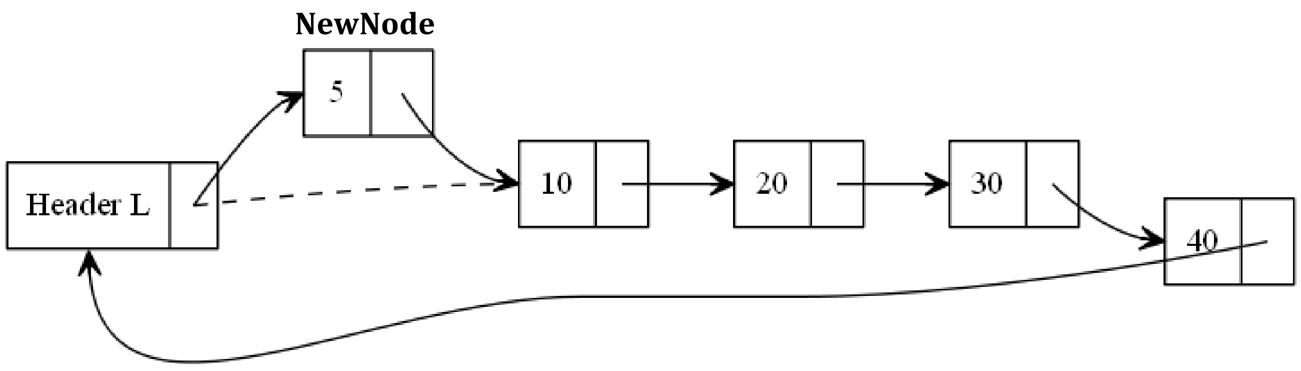
5.8.1 Insert an Element at the Beginning

Example

The general idea is shown in Figure. The dashed line represents the old pointer.



InsertBeg(List, 5)



Algorithm

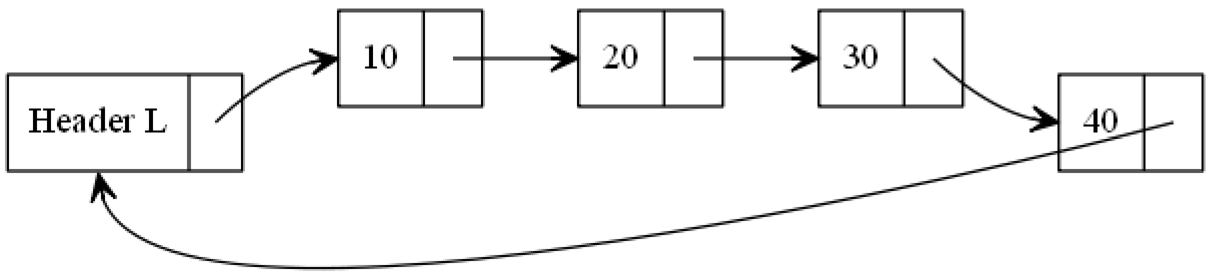
```
InsertBeg(List, e)

Step 1  : Start.
Step 2  : Set NewNode = addressof(Node).
Step 4  : Set NewNode→Element = e.
Step 5  : Set NewNode→Next = List→Next.
Step 6  : Set List→Next = NewNode.
Step 7: Stop.
```

Routine

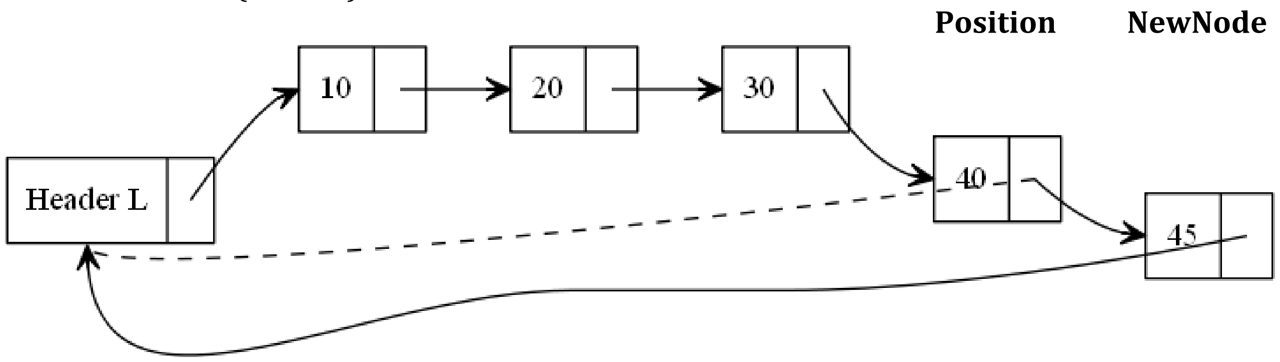
```c
void InsertBeg(Node *List, int e)
{
	Node *NewNode = malloc(sizeof(Node));
	NewNode->Element = e;
	NewNode->Next = List->Next;
	List->Next = NewNode;
}
```

5.8.2 Insert an Element at the End

Example



InsertLast(List, 45)



Algorithm

```
InsertLast(List, e)
Step 1  : Start.
Step 2  : Set NewNode = addressof(Node).
Step 3  : Set NewNode→Element = e.
Step 4  : If List = NULL, then goto Step 5 else goto Step 7.
Step 5  : Set NewNode→Next = List.
Step 6  : Set List→Next = NewNode and goto Step 12.
Step 7  : Set Position = List.
Step 8  : Repeat the Step 9 until Position→Next != List.
Step 9  : Set Position = Position→Next.
Step 10: Set Position→Next = NewNode.
Step 11: Set NewNode→Next = List.
Step 12: Stop.
```

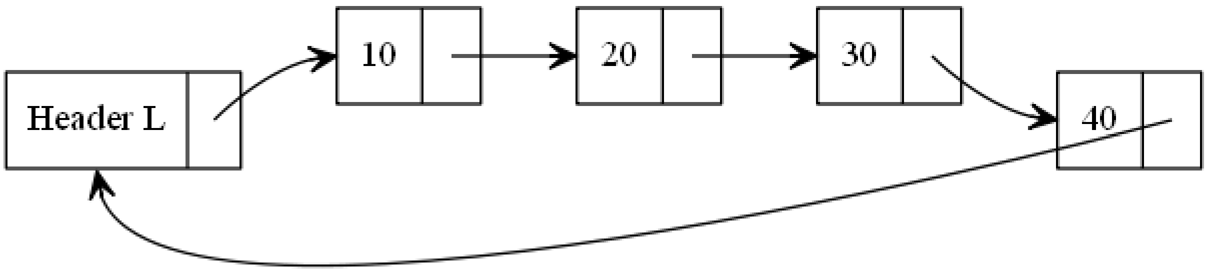
Routine

```
void InsertLast(Node *List, int e)
{
        Node *NewNode = malloc(sizeof(Node));
        Node *Position;
        NewNode->Element = e;
        if(IsEmpty(List))
        {
                NewNode->Next = List;
                List->Next = NewNode;
        }
```

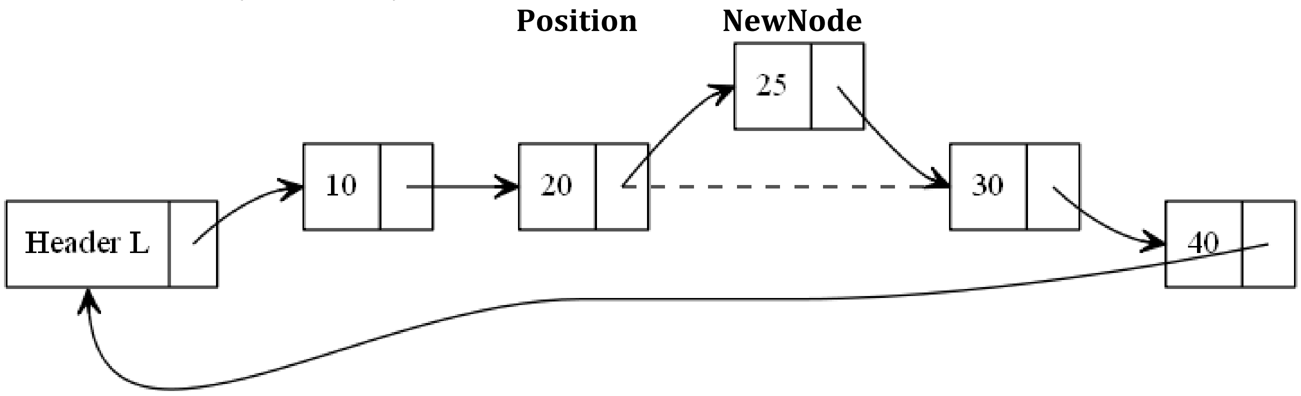
```
          else
          {
                Position = List;
                while(Position->Next != List)
                      Position = Position->Next;
                Position->Next = NewNode;
                NewNode->Next = List;
          }
    }
```

5.8.3 Insert an Element in the Middle

Example



InsertMid(List, 20, 25)



Algorithm

InsertMid(List, p, e)

```
Step 1  : Start.
Step 2  : Set NewNode = addressof(Node).
Step 3  : Set Position = Find(List, p).
Step 4  : Set NewNode→Element = e.
Step 5  : Set NewNode→Next = Position→Next.
Step 6  : Set Position→Next = NewNode.
Step 7  : Stop.
```

Routine

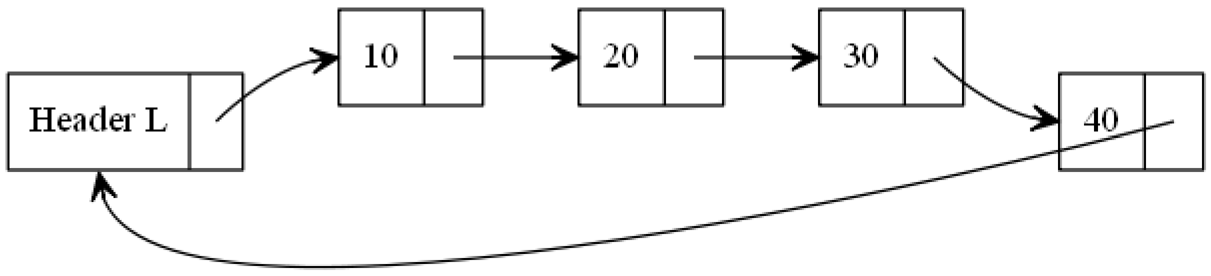
```
void InsertMid(Node *List, int p, int e)
{
      Node *NewNode = malloc(sizeof(Node));
      Node *Position;
      Position = Find(List, p);
      NewNode->Element = e;
      NewNode->Next = Position->Next;
      Position->Next = NewNode;
}
```

5.9 DELETE

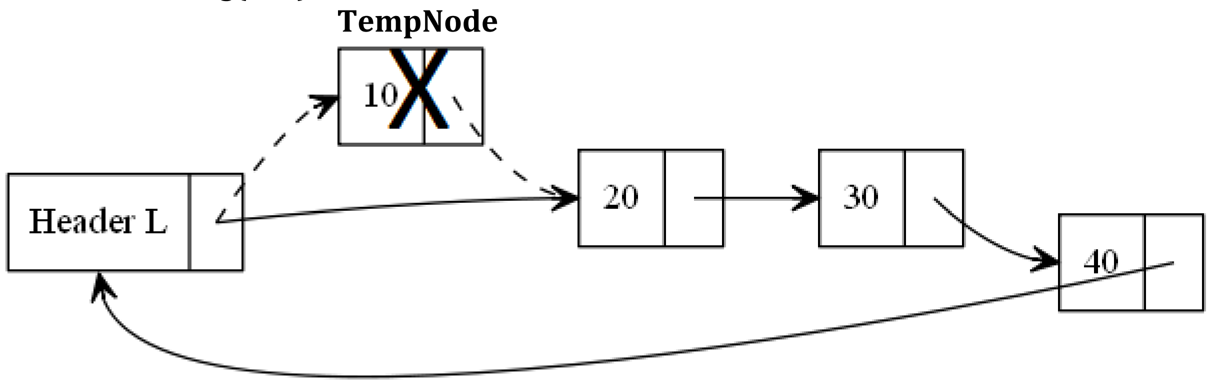
The delete command can be executed in one pointer change. Our routine will delete some element X in list L. We need to decide what to do if x occurs more than once or not at all. Our routine deletes the first occurrence of x and does nothing if x is not in the list. To do this, we find p, which is the cell prior to the one containing x, via a call to FindPrevious.

5.9.1 Delete an Element from the Beginning

Example



DeleteBeg(List)



Algorithm

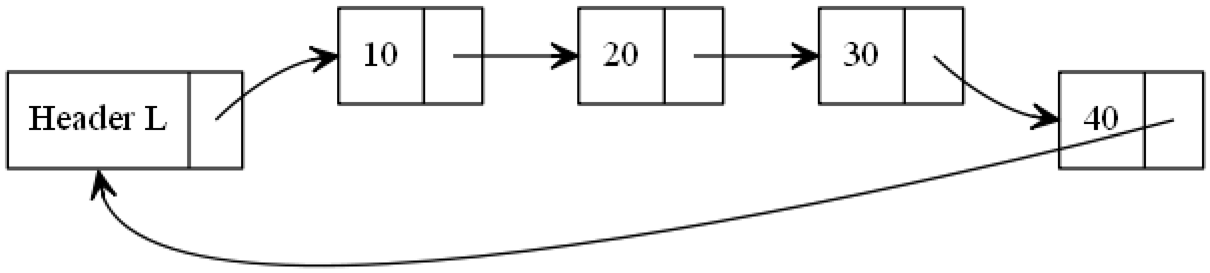
```
DeleteBeg(List, e)
Step 1  : Start.
Step 2  : If !IsEmpty = True, then goto Step 3 else goto Step 7.
Step 3  : Set TempNode = List→Next.
Step 4  : Set List→Next = TempNode→Next.
Step 5  : Display the TempNode→Element.
Step 6  : Delete TempNode and goto Step 8.
Step 7  : Display "List is Empty".
Step 8: Stop.
```

Routine

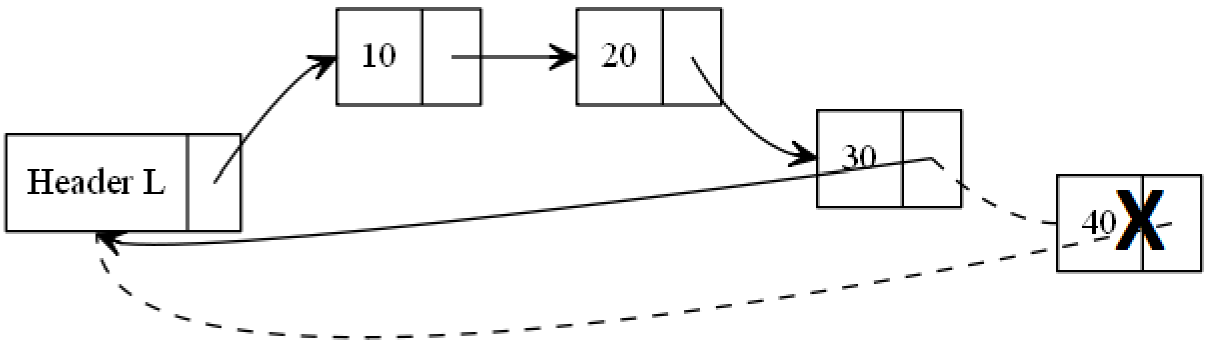
```
void DeleteBeg(Node *List)
{
      if(!IsEmpty(List))
      {
            Node *TempNode;
            TempNode = List->Next;
            List->Next = TempNode->Next;
            printf("The deleted item is %d\n", TempNode->Element);
            free(TempNode);
      }
      else
            printf("List is empty...!\n");
}
```

5.9.2 Delete an Element from the End

Example



DeleteEnd(List)



Position **TempNode**

Algorithm

DeleteEnd(List)

```
Step 1 : Start.
Step 2 : If !IsEmpty = True, then goto Step 3 else goto Step 10.
Step 3 : Set Position = List.
Step 4 : Repeat the Step 5 until Position→Next != List.
Step 5 : Set Position = Position→Next.
Step 6 : Set TempNode = Position→Next.
```

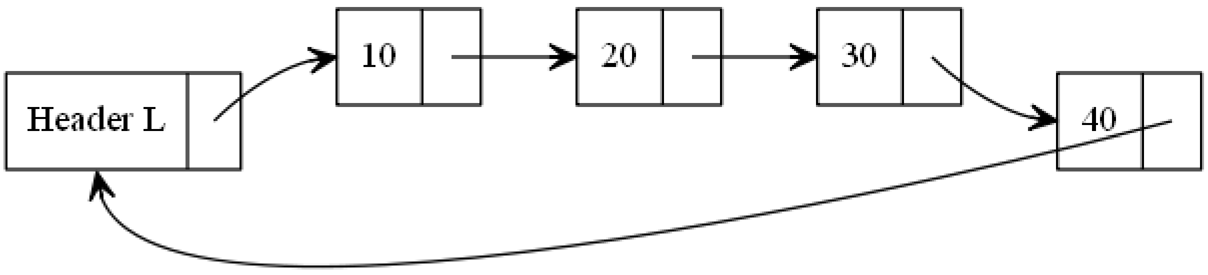
Step 7 : Set Position→Next = List.
Step 8 : Display TempNode→Element.
Step 9 : Delete TempNode and goto Step 11.
Step 10: Display "List is Empty".
Step 11: Stop.

Routine

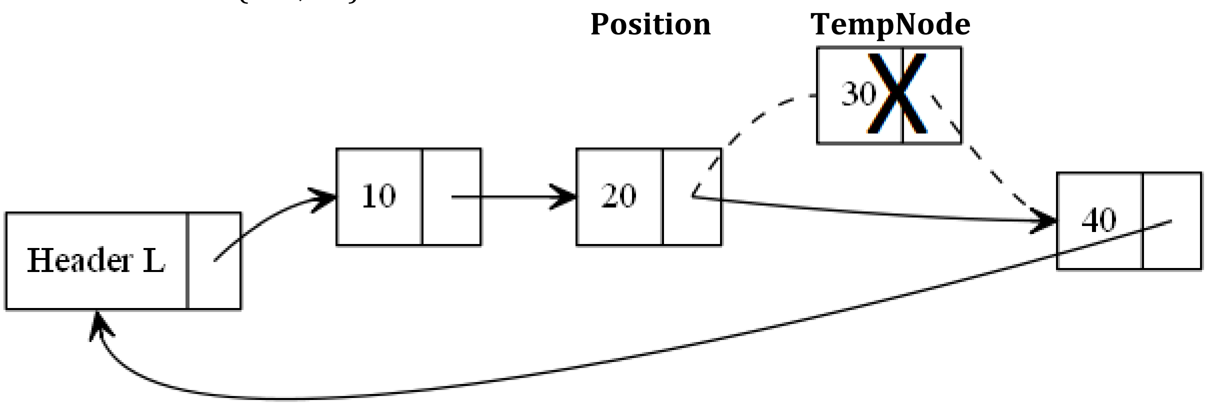
```
void DeleteEnd(Node *List)
{
	if(!IsEmpty(List))
	{
		Node *Position;
		Node *TempNode;
		Position = List;
		while(Position->Next->Next != List)
			Position = Position->Next;
		TempNode = Position->Next;
		Position->Next = List;
		printf("The deleted item is %d\n", TempNode->Element);
		free(TempNode);
	}
	else
		printf("List is empty...!\n");
}
```

5.9.3 Delete an Element from the Middle

Example



DeleteMid(List, 30)



Algorithm

```
DeleteMid(List, e)
Step 1  : Start.
Step 2  : If !IsEmpty = True, then goto Step 3 else goto Step 9.
Step 3  : Set Position = FindPrevious(List, e).
Step 4  : If !Islast(Position, List) = True, then goto Step 5 else goto Step 10.
Step 5  : Set TempNode = Position→Next.
Step 6  : Set Position→Next = TempNode→Next.
Step 7  : Display the TempNode→Element.
Step 8  : Delete TempNode and goto Step 10.
Step 9  : Display "List is Empty".
Step 10: Stop.
```

Routine

```
void DeleteMid(Node *List, int e)
{
	if(!IsEmpty(List))
	{
		Node *Position;
		Node *TempNode;
		Position = FindPrevious(List, e);
		if(!IsLast(Position, List))
		{
			TempNode = Position->Next;
			Position->Next = TempNode->Next;
			printf("The deleted item is %d\n", TempNode->Element);
			free(TempNode);
		}
	}
	else
		printf("List is empty...!\n");
}
```

5.10 PROGRAM

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
	int Element;
	struct node *Next;
};
typedef struct node Node;

int IsEmpty(Node *List);
int IsLast(Node *Position, Node *List);
Node *Find(Node *List, int x);
Node *FindPrevious(Node *List, int x);
Node *FindNext(Node *List, int x);
void InsertBeg(Node *List, int e);
void InsertLast(Node *List, int e);
void InsertMid(Node *List, int p, int e);
void DeleteBeg(Node *List);
void DeleteEnd(Node *List);
void DeleteMid(Node *List, int e);
void Traverse(Node *List);

int main()
{
	Node *List = malloc(sizeof(Node));
	List->Next = List;
	Node *Position;
	int ch, e, p;
	printf("1.Insert Beg \n2.Insert Middle \n3.Insert End");
	printf("\n4.Delete Beg \n5.Delete Middle \n6.Delete End");
	printf("\n7.Find \n8.Traverse \n9.Exit\n");
	do
	{
		printf("Enter your choice : ");
		scanf("%d", &ch);
		switch(ch)
		{
			case 1:
				printf("Enter the element : ");
				scanf("%d", &e);
				InsertBeg(List, e);
				break;
			case 2:
				printf("Enter the position element : ");
				scanf("%d", &p);
				printf("Enter the element : ");
				scanf("%d", &e);
				InsertMid(List, p, e);
				break;
			case 3:
				printf("Enter the element : ");
				scanf("%d", &e);
				InsertLast(List, e);
				break;
			case 4:
				DeleteBeg(List);
				break;
			case 5:
				printf("Enter the element : ");
				scanf("%d", &e);
				DeleteMid(List, e);
				break;
			case 6:
				DeleteEnd(List);
```

```
                        break;
                case 7:
                        printf("Enter the element : ");
                        scanf("%d", &e);
                        Position = Find(List, e);
                        if(Position != List)
                                printf("Element found...!\n");
                        else
                                printf("Element not found...!\n");
                        break;
                case 8:
                        Traverse(List);
                        break;
                }
        } while(ch <= 8);

        return 0;
}

int IsEmpty(Node *List)
{
        if(List->Next == List)
                return 1;
        else
                return 0;
}

int IsLast(Node *Position, Node *List)
{
        if(Position->Next == List)
                return 1;
        else
                return 0;
}

Node *Find(Node *List, int x)
{
        Node *Position;
        Position = List->Next;
        while(Position != List && Position->Element != x)
                Position = Position->Next;
        return Position;
}

Node *FindPrevious(Node *List, int x)
{
        Node *Position;
        Position = List;
        while(Position->Next != List && Position->Next->Element != x)
                Position = Position->Next;
        return Position;
```

```
}

Node *FindNext(Node *List, int x)
{
	Node *Position;
	Position = Find(List, x);
	return Position->Next;
}

void InsertBeg(Node *List, int e)
{
	Node *NewNode = malloc(sizeof(Node));
	NewNode->Element = e;
	NewNode->Next = List->Next;
	List->Next = NewNode;
}

void InsertLast(Node *List, int e)
{
	Node *NewNode = malloc(sizeof(Node));
	Node *Position;
	NewNode->Element = e;
	if(IsEmpty(List))
	{
		NewNode->Next = List;
		List->Next = NewNode;
	}
	else
	{
		Position = List;
		while(Position->Next != List)
			Position = Position->Next;
		Position->Next = NewNode;
		NewNode->Next = List;
	}
}

void InsertMid(Node *List, int p, int e)
{
	Node *NewNode = malloc(sizeof(Node));
	Node *Position;
	Position = Find(List, p);
	NewNode->Element = e;
	NewNode->Next = Position->Next;
	Position->Next = NewNode;
}

void DeleteBeg(Node *List)
{
	if(!IsEmpty(List))
	{
```

```
                Node *TempNode;
                TempNode = List->Next;
                List->Next = TempNode->Next;
                printf("The deleted item is %d\n", TempNode->Element);
                free(TempNode);
        }
        else
                printf("List is empty...!\n");
}

void DeleteEnd(Node *List)
{
        if(!IsEmpty(List))
        {
                Node *Position;
                Node *TempNode;
                Position = List;
                while(Position->Next->Next != List)
                        Position = Position->Next;
                TempNode = Position->Next;
                Position->Next = List;
                printf("The deleted item is %d\n", TempNode->Element);
                free(TempNode);
        }
        else
                printf("List is empty...!\n");
}

void DeleteMid(Node *List, int e)
{
        if(!IsEmpty(List))
        {
                Node *Position;
                Node *TempNode;
                Position = FindPrevious(List, e);
                if(!IsLast(Position, List))
                {
                        TempNode = Position->Next;
                        Position->Next = TempNode->Next;
                        printf("The deleted item is %d\n", TempNode->Element);
                        free(TempNode);
                }
        }
        else
                printf("List is empty...!\n");
}
```

```
void Traverse(Node *List)
{
	if(!IsEmpty(List))
	{
		Node *Position;
		Position = List;
		while(Position->Next != List)
		{
			Position = Position->Next;
			printf("%d\t", Position->Element);
		}
		printf("\n");
	}
	else
		printf("List is empty...!\n");
}
```

OUTPUT

```
1.Insert Beg
2.Insert Middle
3.Insert End
4.Delete Beg
5.Delete Middle
6.Delete End
7.Find
8.Traverse
9.Exit
Enter your choice : 1
Enter the element : 40
Enter your choice : 1
Enter the element : 30
Enter your choice : 1
Enter the element : 20
Enter your choice : 1
Enter the element : 10
Enter your choice : 8
10      20      30      40
Enter your choice : 7
Enter the element : 30
Element found...!
Enter your choice : 1
Enter the element : 5
Enter your choice : 8
5       10      20      30      40
Enter your choice : 3
Enter the element : 45
Enter your choice : 8
5       10      20      30      40      45
Enter your choice : 2
Enter the position element : 20
Enter the element : 25
Enter your choice : 8
5       10      20      25      30      40      45
Enter your choice : 4
The deleted item is 5
Enter your choice : 8
10      20      25      30      40      45
Enter your choice : 6
The deleted item is 45
Enter your choice : 8
10      20      25      30      40
Enter your choice : 5
Enter the element : 30
The deleted item is 30
Enter your choice : 8
10      20      25      40
Enter your choice : 9
```

REVIEW QUESTIONS

1. What is circular linked list?
2. What is a singly linked circular list?

CHAPTER - 6 - DOUBLY LINKED LIST IMPLEMENTATION

6.1 INTRODUCTION

Sometimes it is convenient to traverse lists backwards. The standard implementation does not help here, but the solution is simple. Merely add an extra field to the data structure, containing a pointer to the previous cell. The cost of this is an extra link, which adds to the space requirement and also doubles the cost of insertions and deletions because there are more pointers to fix. On the other hand, it simplifies deletion, because you no longer have to refer to a key by using a pointer to the previous cell; this information is now at hand. Figure shows a doubly linked list.

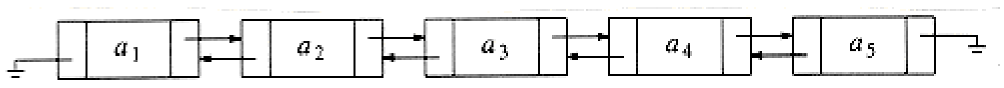


Fig. 6.1 A Doubly Linked List

A doubly linked list is a linked list in which each node has three fields namely data field, next link (Next) and previous link (Prev). Next points to the successor node in the list whereas Prev points to the predecessor node.

Previous Pointer	Data Element	Next Pointer

Node

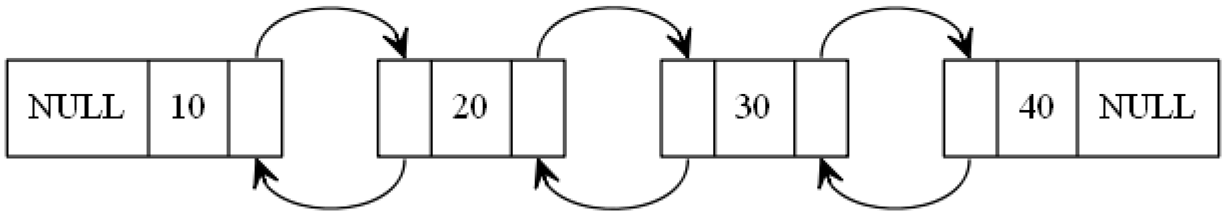


Fig. 6.2 Doubly Linked List

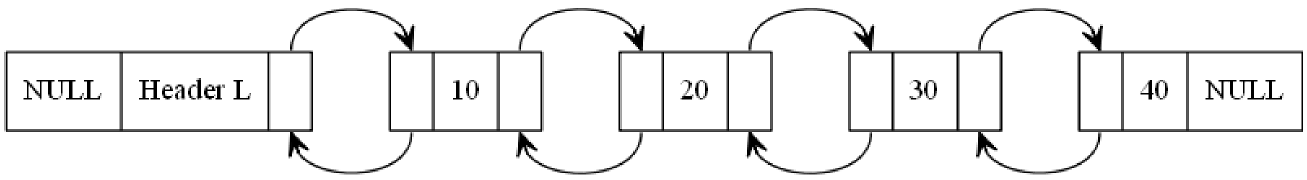


Fig. 6.3 Doubly Linked List with Header

6.2 TYPE DECLARATIONS FOR DOUBLY LINKED LIST

```
struct node
{
        struct node *Prev;
        int Element;
        struct node *Next;

};
typedef struct node Node;
```

6.3 EMPTY LIST WITH HEADER

Example

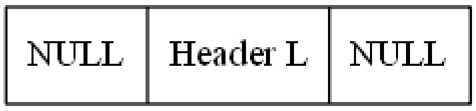


Fig. 6.4 Empty Doubly Linked List with Header

Algorithm

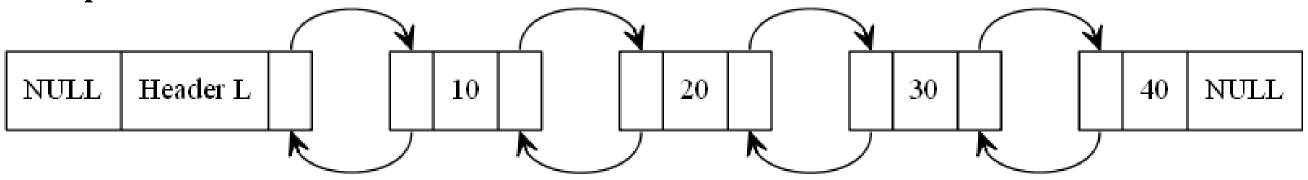
```
IsEmpty(List)

Step 1 : Start.
Step 2 : If List→Next = NULL goto Step 3 else goto Step 4.
Step 3 : Return 1 and Stop.
Step 4 : Return 0 and Stop.
Step 5 : Stop.
```

Routine

```
int IsEmpty(Node *List)
{
        if(List->Next == NULL)
                return 1;
        else
                return 0;
}
```

6.4 CURRENT POSITION IS LAST

Example



Position

Fig. 6.5. Current Position is the Last in a Doubly Linked List

Algorithm

```
IsLast(Position)

Step 1  : Start.
Step 2  : If Position→Next = NULL goto Step 3 else goto Step 4.
Step 3  : Return 1 and Stop.
Step 4  : Return 0 and Stop.
Step 5  : Stop.
```

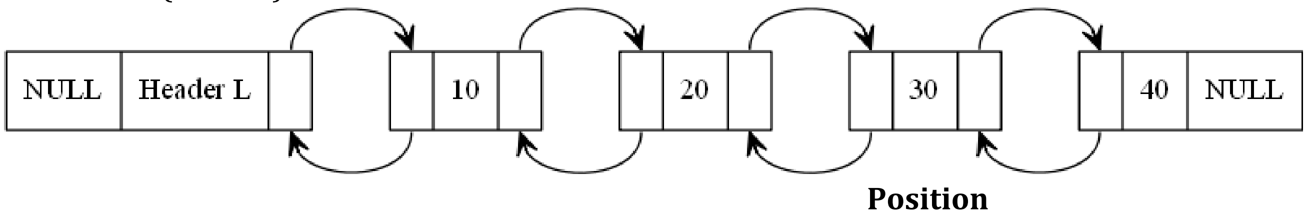
Routine to Check Whether the Current Position is Last

```
int IsLast(Node *Position)
{
      if(Position->Next == NULL)
            return 1;
      else
            return 0;
}
```

6.5 FIND

Example

Find(List, 30)



Algorithm

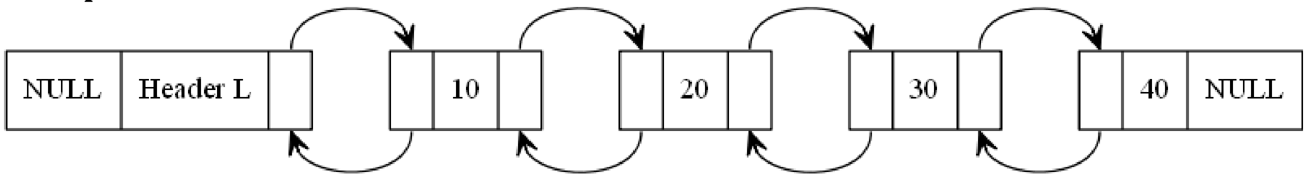
```
Find(List, x)

Step 1  : Start.
Step 2  : Set Position = List→Next.
Step 3  : Repeat the Step 4 until Position != NULL and Position→Element != x.
Step 4  : Set Position = Position→Next.
Step 5  : Return Position.
Step 6  : Stop.
```

Routine

```
Node *Find(Node *List, int x)
{
      Node *Position;
      Position = List->Next;
      while(Position != NULL && Position->Element != x)
                  Position = Position->Next;
      return Position;
}
```

6.6 TRAVERSE THE LIST

Example



Algorithm

```
Traverse(List)

Step 1  : Start.
Step 2  : If !IsEmpty = TRUE goto Step 3 else goto Step 8.
Step 3  : Set Position = List.
Step 4  : Repeat the Steps 5-6 until Position→Next != NULL.
Step 5  : Set Position = Position→Next.
Step 6  : Display Position→Element.
Step 7  : Goto Step 9.
Step 8  : Display "List is empty".
Step 9  : Stop.
```

Routine

```c
void Traverse(Node *List)
{
      if(!IsEmpty(List))
      {
            Node *Position;
            Position = List;
            while(Position->Next != NULL)
            {
                  Position = Position->Next;
                  printf("%d\t", Position->Element);
            }
            printf("\n");
      }
      else
      {
            printf("List is empty...!\n");
      }
}
```

6.7 INSERT

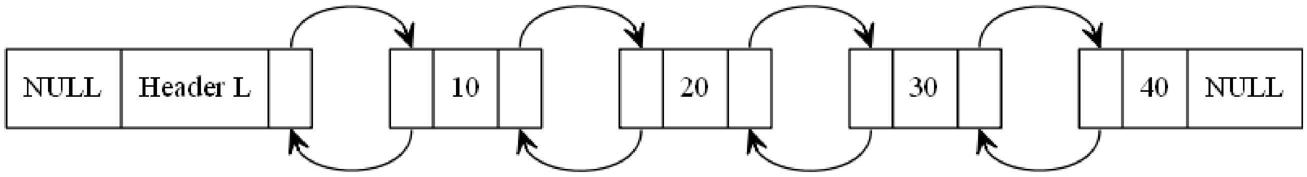
Insertion

- Insert an element at the beginning
- Insert an element at the end
- Insert an element in the middle

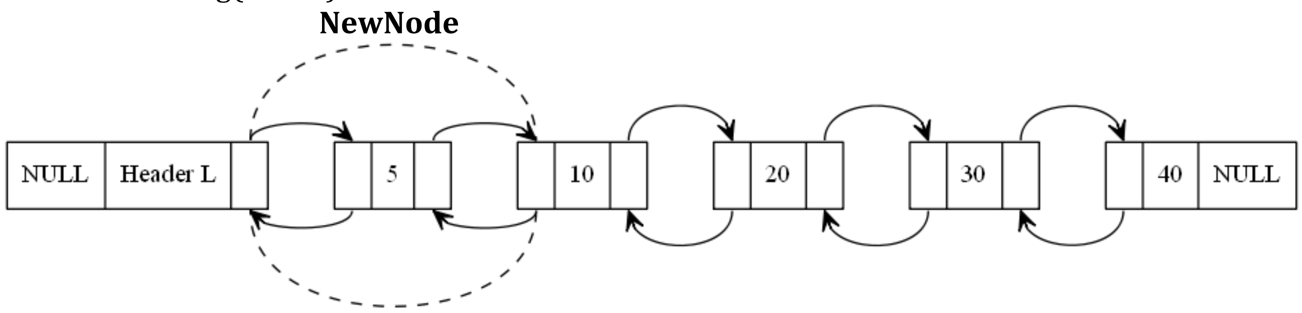
6.7.1 Insert an Element at the Beginning

Example

The general idea is shown in Figure. The dashed line represents the old pointer.



InsertBeg(List, 5)



Algorithm

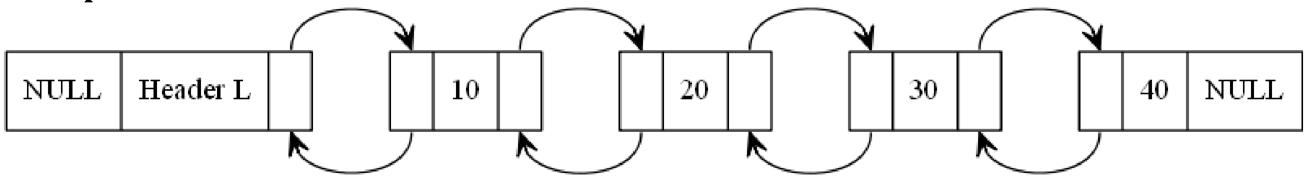
```
InsertBeg(List, e)

Step 1  : Start.
Step 2  : Set NewNode = addressof(Node).
Step 3  : If List = NULL, then goto Step 4 else goto Step 8.
Step 4  : Set NewNode→Element = e.
Step 5  : Set NewNode→Next = NULL.
Step 6  : Set NewNode→Prev = List.
Step 7  : Set List→Next = NewNode and goto Step 13.
Step 8  : Set NewNode→Element = e.
Step 9  : Set NewNode→Next = List→Next.
Step 10: Set NewNode→Next→Prev = NewNode.
Step 11: Set NewNode→Prev = List.
Step 12: Set List→Next = NewNode.
Step 13: Stop.
```

Routine

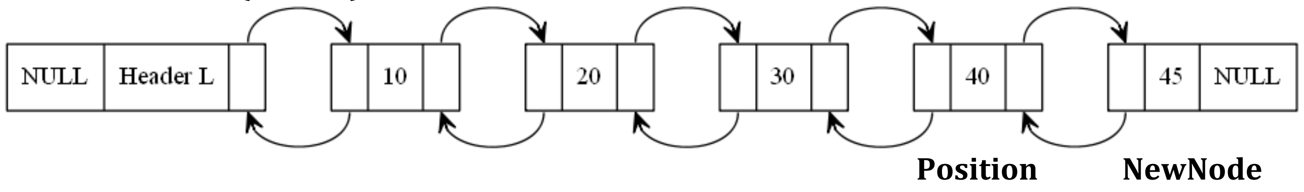
```
void InsertBeg(Node *List, int e)
{
        Node *NewNode = malloc(sizeof(Node));
        NewNode->Element = e;
        if(IsEmpty(List))
                NewNode->Next = NULL;
        else
        {
                NewNode->Next = List->Next;
                NewNode->Next->Prev = NewNode;
        }
        NewNode->Prev = List;
        List->Next = NewNode;
    }
```

6.7.2 Insert an Element at the End

Example



InsertLast(List, 45)



Algorithm

```
InsertLast(List, e)

Step 1  : Start.
Step 2  : Set NewNode = addressof(Node).
Step 3  : If List = NULL, then goto Step 4 else goto Step 8.
Step 4  : Set NewNode→Element = e.
Step 5  : Set NewNode→Next = NULL.
Step 6  : Set NewNode→Prev = List.
Step 7  : Set List→Next = NewNode and goto Step 15.
Step 8  : Set Position = List.
Step 9  : Repeat the Step 10 until Position→Next != NULL.
Step 10: Set Position = Position→Next.
Step 11: Set NewNode→Element = e.
Step 12: Set Position→Next = NewNode.
Step 13: Set NewNode→Prev = Position.
Step 14: Set NewNode→Next = NULL.
Step 15: Stop.
```

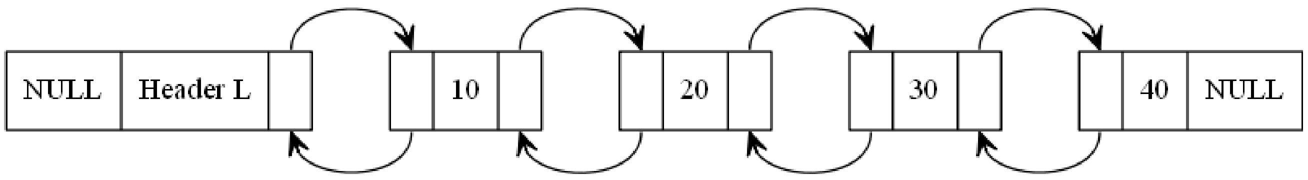
Routine

```
void InsertLast(Node *List, int e)
{
    Node *NewNode = malloc(sizeof(Node));
    Node *Position;
    NewNode->Element = e;
    NewNode->Next = NULL;
    if(IsEmpty(List))
    {
        NewNode->Prev = List;
        List->Next = NewNode;
    }
```

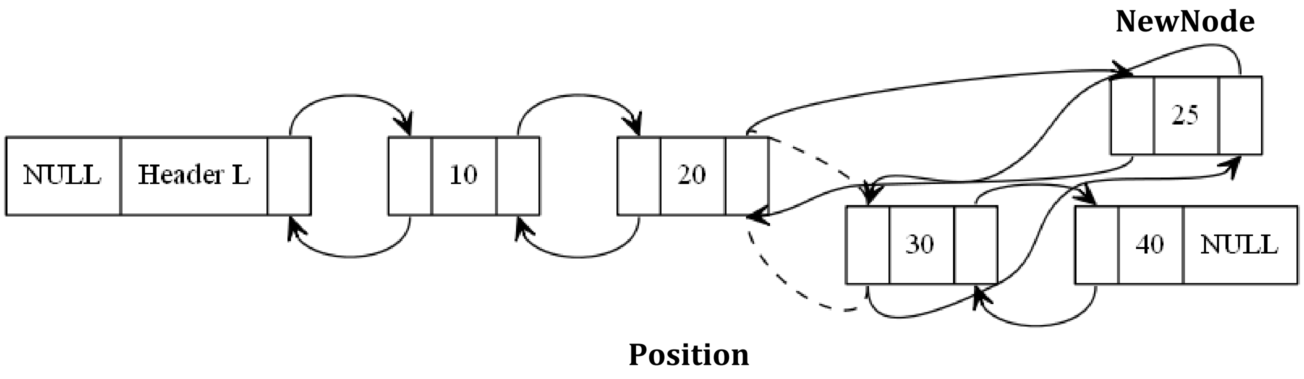
```
          else
          {
               Position = List;
               while(Position->Next != NULL)
                    Position = Position->Next;
               Position->Next = NewNode;
               NewNode->Prev = Position;
          }
     }
```

6.7.3 Insert an Element in the Middle

Example



InsertMid(List, 20, 25)



Algorithm

InsertMid(List, p, e)

```
Step 1 : Start.
Step 2 : Set NewNode = addressof(Node).
Step 3 : Set Position = Find(List, p).
Step 4 : Set NewNode→Element = e.
Step 5 : Set NewNode→Next = Position→Next.
Step 6 : Set Position→Next→Prev = NewNode.
Step 7 : Set Position→Next = NewNode.
Step 8 : Set NewNode→Prev = Position.
Step 9 : Stop.
```

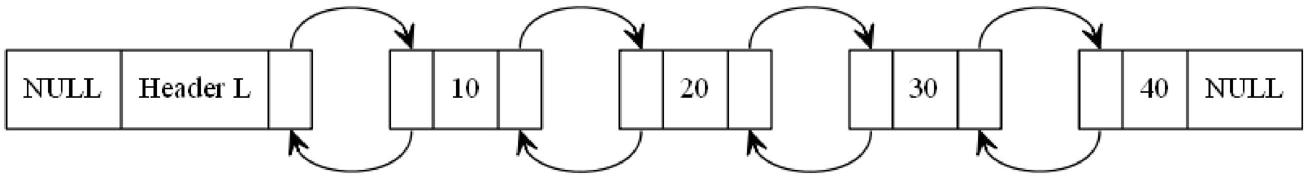
Routine

```
void InsertMid(Node *List, int p, int e)
{
      Node *NewNode = malloc(sizeof(Node));
      Node *Position;
      Position = Find(List, p);
      NewNode->Element = e;
      NewNode->Next = Position->Next;
      Position->Next->Prev = NewNode;
      Position->Next = NewNode;
      NewNode->Prev = Position;
}
```

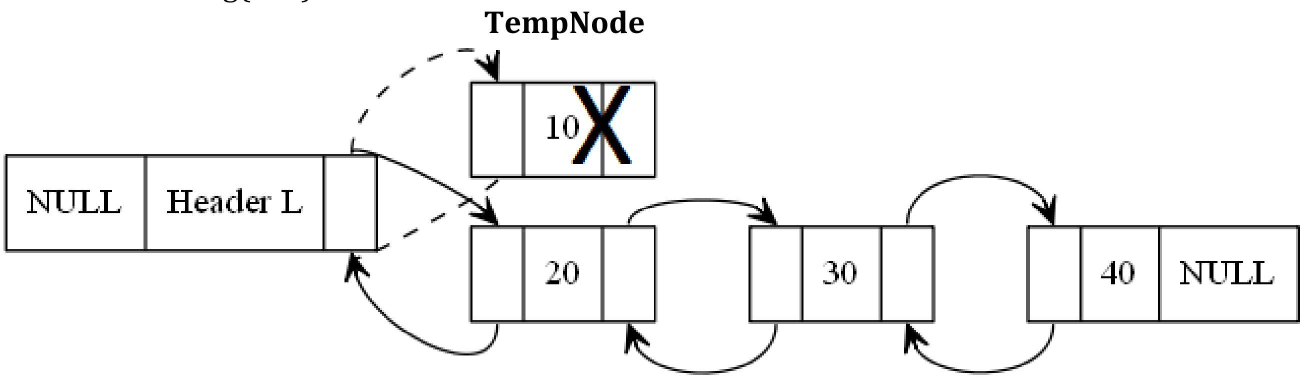
6.8 DELETE

6.8.1 Delete an Element from the Beginning

Example



DeleteBeg(List)



Algorithm

DelBeg(List, e)

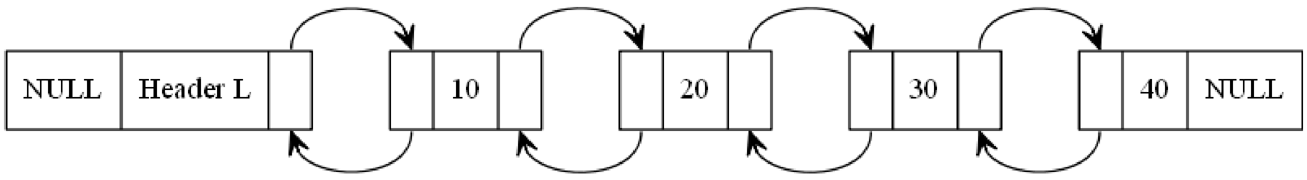
```
Step 1  : Start.
Step 2  : If !IsEmpty = True, then goto Step 3 else goto Step 9.
Step 3  : Set TempNode = List→Next.
Step 4  : Set List→Next = TempNode→Next.
Step 5  : If List→Next != NULL, then goto Step 6 else goto Step 7.
Step 6  : Set TempNode→Next→Prev = List.
Step 7  : Display the TempNode→Element.
Step 8  : Delete TempNode and goto Step 10.
Step 9  : Display "List is Empty".
Step 10: Stop.
```

Routine

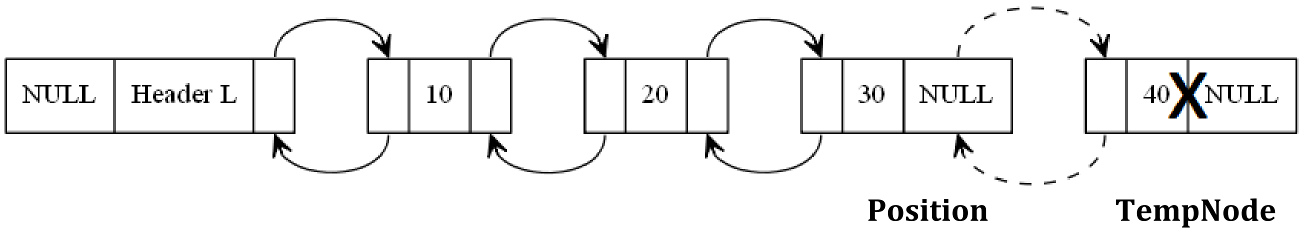
```
void DeleteBeg(Node *List)
{
      if(!IsEmpty(List))
      {
            Node *TempNode;
            TempNode = List->Next;
            List->Next = TempNode->Next;
            if(List->Next != NULL)
                  TempNode->Next->Prev = List;
            printf("The deleted item is %d\n", TempNode->Element);
            free(TempNode);
      }
      else
            printf("List is empty...!\n");
}
```

6.8.2 Delete an Element from the End

Example



DeleteEnd(List)



Algorithm

DeleteEnd(List)

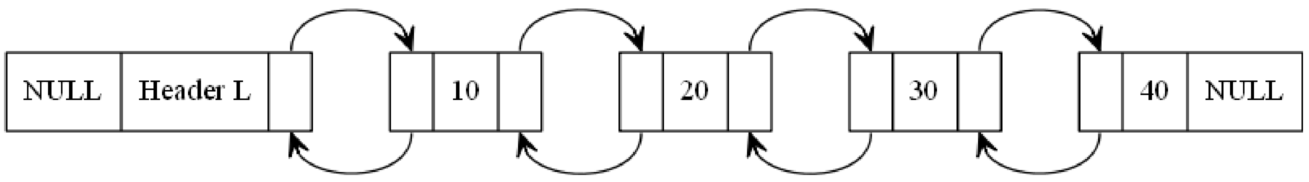
```
Step 1  : Start.
Step 2  : If !IsEmpty = True, then goto Step 3 else goto Step 10.
Step 3  : Set Position = List.
Step 4  : Repeat the Step 5 until Position→Next != NULL.
Step 5  : Set Position = Position→Next.
Step 6  : Set TempNode = Position.
Step 7  : Set Position→Prev→Next = NULL.
Step 8  : Display the TempNode→Element.
Step 9  : Delete TempNode and goto Step 11.
Step 10: Display “List is Empty”.
Step 11: Stop.
```

Routine

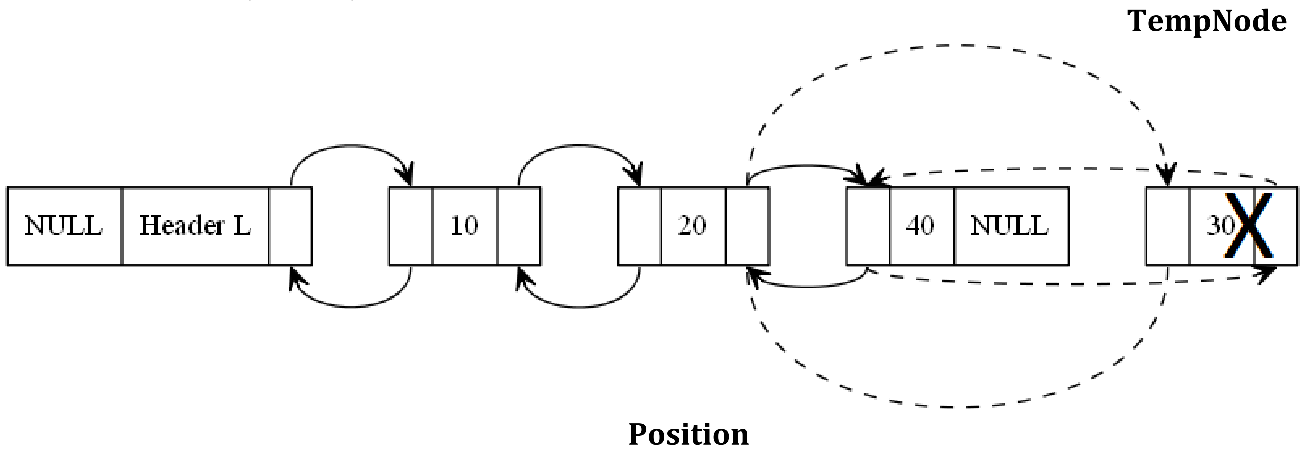
```
void DeleteEnd(Node *List)
{
	if(!IsEmpty(List))
	{
		Node *Position;
		Node *TempNode;
		Position = List;
		while(Position->Next != NULL)
			Position = Position->Next;
		TempNode = Position;
		Position->Prev->Next = NULL;
		printf("The deleted item is %d\n", TempNode->Element);
		free(TempNode);
	}
	else
		printf("List is empty...!\n");
}
```

6.8.3 Delete an Element from the Middle

Example



DeleteMid(List, 30)



Algorithm

DeleteMid(List, e)

```
Step 1  : Start.
Step 2  : If !IsEmpty = True, then goto Step 3 else goto Step 10.
Step 3  : Set Position = Find (List, e).
Step 4  : If !Islast(Position) = True, then goto Step 5 else goto Step 11.
Step 5  : Set TempNode = Position.
Step 6  : Set Position→Prev→Next = Position→Next.
Step 7  : Set Position→Next→Prev = Position→Prev.
Step 8  : Display the TempNode→Element.
Step 9  : Delete TempNode and goto Step 11.
Step 10: Display "List is Empty".
Step 11: Stop.
```

Routine

```
void DeleteMid(Node *List, int e)
{
	if(!IsEmpty(List))
	{
		Node *Position;
		Node *TempNode;
		Position = Find(List, e);
		if(!IsLast(Position))
		{
			TempNode = Position;
			Position->Prev->Next = Position->Next;
			Position->Next->Prev = Position->Prev;
			printf("The deleted item is %d\n", TempNode->Element);
			free(TempNode);
		}
	}
	else
		printf("List is empty...!\n");
}
```

6.9 PROGRAM

```
/* Implementation of doubly linked list - DLL.C */

#include <stdio.h>
#include <stdlib.h>

struct node
{
	struct node *Prev;
	int Element;
	struct node *Next;

};
typedef struct node Node;

int IsEmpty(Node *List);
int IsLast(Node *Position);
Node *Find(Node *List, int x);
void InsertBeg(Node *List, int e);
void InsertLast(Node *List, int e);
```

```c
void InsertMid(Node *List, int p, int e);
void DeleteBeg(Node *List);
void DeleteEnd(Node *List);
void DeleteMid(Node *List, int e);
void Traverse(Node *List);

int main()
{
	Node *List = malloc(sizeof(Node));
	List->Prev = NULL;
	List->Next = NULL;
	Node *Position;
	int ch, e, p;
	printf("1.Insert Beg \n2.Insert Middle \n3.Insert End");
	printf("\n4.Delete Beg \n5.Delete Middle \n6.Delete End");
	printf("\n7.Find \n8.Traverse \n9.Exit\n");
	do
	{
		printf("Enter your choice : ");
		scanf("%d", &ch);
		switch(ch)
		{
			case 1:
				printf("Enter the element : ");
				scanf("%d", &e);
				InsertBeg(List, e);
				break;
			case 2:
				printf("Enter the position element : ");
				scanf("%d", &p);
				printf("Enter the element : ");
				scanf("%d", &e);
				InsertMid(List, p, e);
				break;
			case 3:
				printf("Enter the element : ");
				scanf("%d", &e);
				InsertLast(List, e);
				break;
			case 4:
				DeleteBeg(List);
				break;
			case 5:
				printf("Enter the element : ");
				scanf("%d", &e);
				DeleteMid(List, e);
				break;
			case 6:
				DeleteEnd(List);
				break;
```

```
                case 7:
                     printf("Enter the element : ");
                     scanf("%d", &e);
                     Position = Find(List, e);
                     if(Position != NULL)
                          printf("Element found...!\n");
                     else
                          printf("Element not found...!\n");
                     break;
                case 8:
                     Traverse(List);
                     break;
           }
      } while(ch <= 8);

      return 0;
}

int IsEmpty(Node *List)
{
      if(List->Next == NULL)
           return 1;
      else
           return 0;
}

int IsLast(Node *Position)
{
      if(Position->Next == NULL)
           return 1;
      else
           return 0;
}

Node *Find(Node *List, int x)
{
      Node *Position;
      Position = List->Next;
      while(Position != NULL && Position->Element != x)
                Position = Position->Next;
      return Position;
}

void InsertBeg(Node *List, int e)
{
      Node *NewNode = malloc(sizeof(Node));
      NewNode->Element = e;
      if(IsEmpty(List))
           NewNode->Next = NULL;
```

```c
	else
	{
		NewNode->Next = List->Next;
		NewNode->Next->Prev = NewNode;
	}
	NewNode->Prev = List;
	List->Next = NewNode;
}

void InsertLast(Node *List, int e)
{
	Node *NewNode = malloc(sizeof(Node));
	Node *Position;
	NewNode->Element = e;
	NewNode->Next = NULL;
	if(IsEmpty(List))
	{
		NewNode->Prev = List;
		List->Next = NewNode;
	}
	else
	{
		Position = List;
		while(Position->Next != NULL)
			Position = Position->Next;
		Position->Next = NewNode;
		NewNode->Prev = Position;
	}
}

void InsertMid(Node *List, int p, int e)
{
	Node *NewNode = malloc(sizeof(Node));
	Node *Position;
	Position = Find(List, p);
	NewNode->Element = e;
	NewNode->Next = Position->Next;
	Position->Next->Prev = NewNode;
	Position->Next = NewNode;
	NewNode->Prev = Position;
}

void DeleteBeg(Node *List)
{
	if(!IsEmpty(List))
	{
		Node *TempNode;
		TempNode = List->Next;
		List->Next = TempNode->Next;
		if(List->Next != NULL)
			TempNode->Next->Prev = List;
```

```
            printf("The deleted item is %d\n", TempNode->Element);
            free(TempNode);
      }
      else
            printf("List is empty...!\n");
}
void DeleteEnd(Node *List)
{
      if(!IsEmpty(List))
      {
            Node *Position;
            Node *TempNode;
            Position = List;
            while(Position->Next != NULL)
                  Position = Position->Next;
            TempNode = Position;
            Position->Prev->Next = NULL;
            printf("The deleted item is %d\n", TempNode->Element);
            free(TempNode);
      }
      else
            printf("List is empty...!\n");
}
void DeleteMid(Node *List, int e)
{
      if(!IsEmpty(List))
      {
            Node *Position;
            Node *TempNode;
            Position = Find(List, e);
            if(!IsLast(Position))
            {
                  TempNode = Position;
                  Position->Prev->Next = Position->Next;
                  Position->Next->Prev = Position->Prev;
                  printf("The deleted item is %d\n", TempNode->Element);
                  free(TempNode);
            }
      }
      else
            printf("List is empty...!\n");
}
void Traverse(Node *List)
{
      if(!IsEmpty(List))
      {
            Node *Position;
            Position = List;
```

```
                while(Position->Next != NULL)
                {
                        Position = Position->Next;
                        printf("%d\t", Position->Element);
                }
                printf("\n");
        }
        else
                printf("List is empty...!\n");
}
```

OUTPUT

```
1.Insert Beg
2.Insert Middle
3.Insert End
4.Delete Beg
5.Delete Middle
6.Delete End
7.Find
8.Traverse
9.Exit
Enter your choice : 1
Enter the element : 40
Enter your choice : 1
Enter the element : 30
Enter your choice : 1
Enter the element : 20
Enter your choice : 1
Enter the element : 10
Enter your choice : 8
10      20      30      40
Enter your choice : 7
Enter the element : 30
Element found...!
Enter your choice : 1
Enter the element : 5
Enter your choice : 8
5       10      20      30      40
Enter your choice : 3
Enter the element : 45
Enter your choice : 8
5       10      20      30      40      45
Enter your choice : 2
Enter the position element : 20
Enter the element : 25
Enter your choice : 8
5       10      20      25      30      40      45
Enter your choice : 4
The deleted item is 5
Enter your choice : 8
```

```
10      20      25      30      40      45
Enter your choice : 6
The deleted item is 45
Enter your choice : 8
10      20      25      30      40
Enter your choice : 5
Enter the element : 30
The deleted item is 30
Enter your choice : 8
10      20      25      40
Enter your choice : 9
```

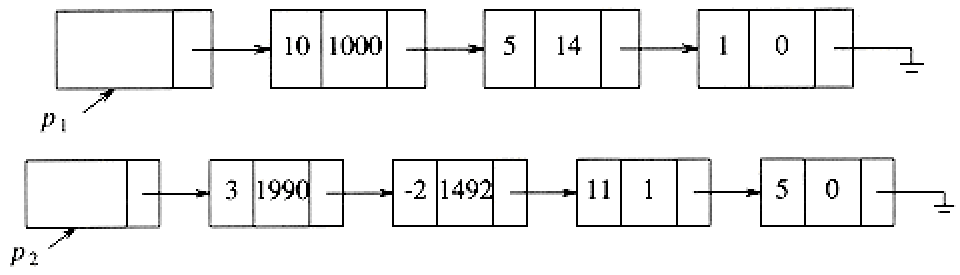
REVIEW QUESTIONS

1. What is a doubly linked list?
2. What are the advantages of doubly linked list over singly linked list? (or) What is the advantage of using doubly linked list over single linked list?
3. Differentiate doubly and circular linked list.

7.1 INTRODUCTION

We can define an abstract data type for single-variable polynomials (with nonnegative exponents) by using a list. We could then write routines to perform addition, subtraction, multiplication, differentiation, and other operations on these polynomials.

7.2 LINKED LIST REPRESENTATIONS OF TWO POLYNOMIALS



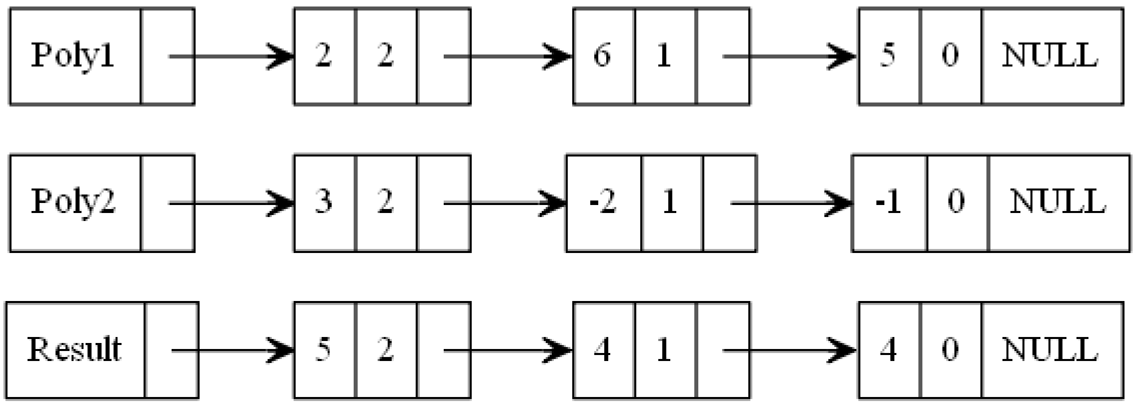
7.3 DECLARATION FOR LINKED LIST IMPLEMENTATION OF POLYNOMIAL ADT

```
struct poly
{
      int coeff;
      int pow;
      struct poly *Next;
};
typedef struct poly Poly;
```

REVIEW QUESTIONS

1. Why is linked list used for polynomial arithmetic?

8.1 EXAMPLE



8.2 ROUTINE

```
void Addition(Poly *Poly1, Poly *Poly2, Poly *Result)
{
      Poly *Position;
      Poly *NewNode;
      Poly1 = Poly1->Next;
      Poly2 = Poly2->Next;
      Result->Next = NULL;
      Position = Result;
      while(Poly1 != NULL && Poly2 != NULL)
      {
             NewNode = malloc(sizeof(Poly));
             if(Poly1->pow == Poly2->pow)
             {
                   NewNode->coeff = Poly1->coeff + Poly2->coeff;
                   NewNode->pow = Poly1->pow;
                   Poly1 = Poly1->Next;
                   Poly2 = Poly2->Next;
             }
             else if(Poly1->pow > Poly2->pow)
             {
                   NewNode->coeff = Poly1->coeff;
                   NewNode->pow = Poly1->pow;
                   Poly1 = Poly1->Next;
             }
             else if(Poly1->pow < Poly2->pow)
             {
                   NewNode->coeff = Poly2->coeff;
                   NewNode->pow = Poly2->pow;
                   Poly2 = Poly2->Next;
             }
```

```
			NewNode->Next = NULL;
			Position->Next = NewNode;
			Position = NewNode;
		}

		while(Poly1 != NULL || Poly2 != NULL)
		{
			NewNode = malloc(sizeof(Poly));
			if(Poly1 != NULL)
			{
				NewNode->coeff = Poly1->coeff;
				NewNode->pow = Poly1->pow;
				Poly1 = Poly1->Next;
			}
			if(Poly2 != NULL)
			{
				NewNode->coeff = Poly2->coeff;
				NewNode->pow = Poly2->pow;
				Poly2 = Poly2->Next;
			}
			NewNode->Next = NULL;
			Position->Next = NewNode;
			Position = NewNode;
		}
}
```

8.3 PROGRAM

```
#include <stdio.h>
#include <stdlib.h>

struct poly
{
	int coeff;
	int pow;
	struct poly *Next;
};
typedef struct poly Poly;
void Create(Poly *List);
void Display(Poly *List);
void Addition(Poly *Poly1, Poly *Poly2, Poly *Result);

int main()
{
	Poly *Poly1 = malloc(sizeof(Poly));
	Poly *Poly2 = malloc(sizeof(Poly));
	Poly *Result = malloc(sizeof(Poly));
	Poly1->Next = NULL;
	Poly2->Next = NULL;
	printf("Enter the values for first polynomial :\n");
	Create(Poly1);
```

```c
	printf("The polynomial equation is : ");
	Display(Poly1);
	printf("\nEnter the values for second polynomial :\n");
	Create(Poly2);
	printf("The polynomial equation is : ");
	Display(Poly2);
	Addition(Poly1, Poly2, Result);
	printf("\nThe polynomial equation addition result is : ");
	Display(Result);
	return 0;
}

void Create(Poly *List)
{
	int choice;
	Poly *Position, *NewNode;
	Position = List;
	do
	{
		NewNode = malloc(sizeof(Poly));
		printf("Enter the coefficient : ");
		scanf("%d", &NewNode->coeff);
		printf("Enter the power : ");
		scanf("%d", &NewNode->pow);
		NewNode->Next = NULL;
		Position->Next = NewNode;
		Position = NewNode;
		printf("Enter 1 to continue : ");
		scanf("%d", &choice);
	} while(choice == 1);
}

void Display(Poly *List)
{
	Poly *Position;
	Position = List->Next;
	while(Position != NULL)
	{
		printf("%dx^%d", Position->coeff, Position->pow);
		Position = Position->Next;
		if(Position != NULL && Position->coeff > 0)
		{
			printf("+");
		}
	}
}

void Addition(Poly *Poly1, Poly *Poly2, Poly *Result)
{
	Poly *Position;
	Poly *NewNode;
	Poly1 = Poly1->Next;
	Poly2 = Poly2->Next;
	Result->Next = NULL;
	Position = Result;
	while(Poly1 != NULL && Poly2 != NULL)
	{
		NewNode = malloc(sizeof(Poly));
		if(Poly1->pow == Poly2->pow)
		{
			NewNode->coeff = Poly1->coeff + Poly2->coeff;
			NewNode->pow = Poly1->pow;
			Poly1 = Poly1->Next;
			Poly2 = Poly2->Next;
		}
		else if(Poly1->pow > Poly2->pow)
		{
			NewNode->coeff = Poly1->coeff;
			NewNode->pow = Poly1->pow;
			Poly1 = Poly1->Next;
		}
		else if(Poly1->pow < Poly2->pow)
		{
			NewNode->coeff = Poly2->coeff;
			NewNode->pow = Poly2->pow;
			Poly2 = Poly2->Next;
		}
		NewNode->Next = NULL;
		Position->Next = NewNode;
		Position = NewNode;
	}

	while(Poly1 != NULL || Poly2 != NULL)
	{
		NewNode = malloc(sizeof(Poly));
		if(Poly1 != NULL)
		{
			NewNode->coeff = Poly1->coeff;
			NewNode->pow = Poly1->pow;
			Poly1 = Poly1->Next;
		}
		if(Poly2 != NULL)
		{
			NewNode->coeff = Poly2->coeff;
			NewNode->pow = Poly2->pow;
			Poly2 = Poly2->Next;
		}
		NewNode->Next = NULL;
		Position->Next = NewNode;
		Position = NewNode;
	}
}
```

OUTPUT

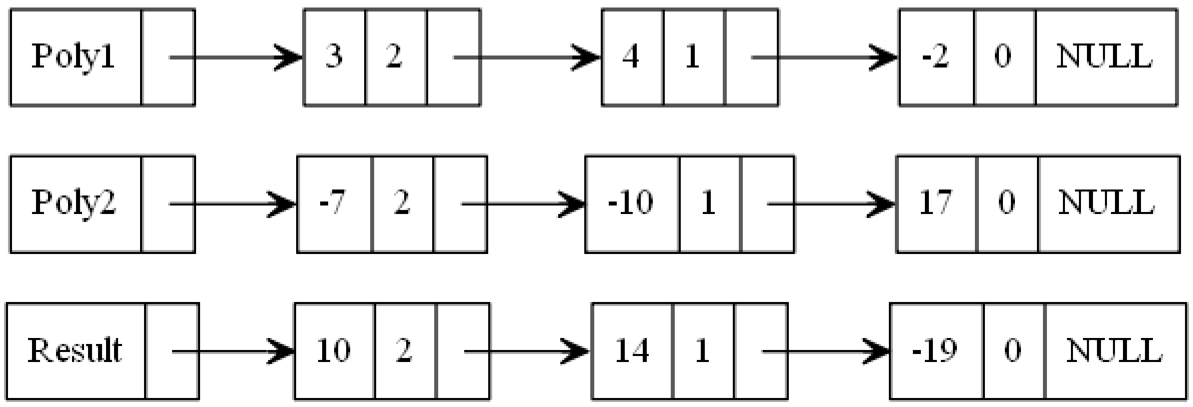
```
Enter the values for first polynomial :
Enter the coefficient : 2
Enter the power : 2
Enter 1 to continue : 1
Enter the coefficient : 6
Enter the power : 1
Enter 1 to continue : 1
Enter the coefficient : 5
Enter the power : 0
Enter 1 to continue : 0
The polynomial equation is : 2x^2+6x^1+5x^0
Enter the values for second polynomial :
Enter the coefficient : 3
Enter the power : 2
Enter 1 to continue : 1
Enter the coefficient : -2
Enter the power : 1
Enter 1 to continue : 1
Enter the coefficient : -1
Enter the power : 0
Enter 1 to continue : 0
The polynomial equation is : 3x^2-2x^1-1x^0
The polynomial equation addition result is : 5x^2+4x^1+4x^0
```

REVIEW QUESTIONS

1. Write the routine for polynomial addition.

CHAPTER - 9 - POLYNOMIAL SUBTRACTION

9.1 EXAMPLE



9.2 ROUTINE

```
void Subtraction(Poly *Poly1, Poly *Poly2, Poly *Result)
{
        Poly *Position;
        Poly *NewNode;
        Poly1 = Poly1->Next;
        Poly2 = Poly2->Next;
        Result->Next = NULL;
        Position = Result;
        while(Poly1 != NULL && Poly2 != NULL)
        {
                NewNode = malloc(sizeof(Poly));
                if(Poly1->pow == Poly2->pow)
                {
                        NewNode->coeff = Poly1->coeff - Poly2->coeff;
                        NewNode->pow = Poly1->pow;
                        Poly1 = Poly1->Next;
                        Poly2 = Poly2->Next;
                }
                else if(Poly1->pow > Poly2->pow)
                {
                        NewNode->coeff = Poly1->coeff;
                        NewNode->pow = Poly1->pow;
                        Poly1 = Poly1->Next;
                }
                else if(Poly1->pow < Poly2->pow)
                {
                        NewNode->coeff = -(Poly2->coeff);
                        NewNode->pow = Poly2->pow;
                        Poly2 = Poly2->Next;
                }
```

```
            NewNode->Next = NULL;
            Position->Next = NewNode;
            Position = NewNode;
      }

      while(Poly1 != NULL || Poly2 != NULL)
      {
            NewNode = malloc(sizeof(Poly));
            if(Poly1 != NULL)
            {
                  NewNode->coeff = Poly1->coeff;
                  NewNode->pow = Poly1->pow;
                  Poly1 = Poly1->Next;
            }
            if(Poly2 != NULL)
            {
                  NewNode->coeff = -(Poly2->coeff);
                  NewNode->pow = Poly2->pow;
                  Poly2 = Poly2->Next;
            }
            NewNode->Next = NULL;
            Position->Next = NewNode;
            Position = NewNode;
      }
}
```

9.3 PROGRAM

```
#include <stdio.h>
#include <stdlib.h>

struct poly
{
      int coeff;
      int pow;
      struct poly *Next;
};
typedef struct poly Poly;
void Create(Poly *List);
void Display(Poly *List);
void Subtraction(Poly *Poly1, Poly *Poly2, Poly *Result);

int main()
{
      Poly *Poly1 = malloc(sizeof(Poly));
      Poly *Poly2 = malloc(sizeof(Poly));
      Poly *Result = malloc(sizeof(Poly));
      Poly1->Next = NULL;
      Poly2->Next = NULL;
      printf("Enter the values for first polynomial :\n");
      Create(Poly1);
```

```c
	printf("The polynomial equation is : ");
	Display(Poly1);
	printf("\nEnter the values for second polynomial :\n");
	Create(Poly2);
	printf("The polynomial equation is : ");
	Display(Poly2);
	Subtraction(Poly1, Poly2, Result);
	printf("\nThe polynomial equation subtraction result is : ");
	Display(Result);
	return 0;
}

void Create(Poly *List)
{
	int choice;
	Poly *Position, *NewNode;
	Position = List;
	do
	{
		NewNode = malloc(sizeof(Poly));
		printf("Enter the coefficient : ");
		scanf("%d", &NewNode->coeff);
		printf("Enter the power : ");
		scanf("%d", &NewNode->pow);
		NewNode->Next = NULL;
		Position->Next = NewNode;
		Position = NewNode;
		printf("Enter 1 to continue : ");
		scanf("%d", &choice);
	} while(choice == 1);
}

void Display(Poly *List)
{
	Poly *Position;
	Position = List->Next;
	while(Position != NULL)
	{
		printf("%dx^%d", Position->coeff, Position->pow);
		Position = Position->Next;
		if(Position != NULL && Position->coeff > 0)
		{
			printf("+");
		}
	}
}

void Subtraction(Poly *Poly1, Poly *Poly2, Poly *Result)
{
	Poly *Position;
	Poly *NewNode;
```

```
Poly1 = Poly1->Next;
Poly2 = Poly2->Next;
Result->Next = NULL;
Position = Result;
while(Poly1 != NULL && Poly2 != NULL)
{
      NewNode = malloc(sizeof(Poly));
      if(Poly1->pow == Poly2->pow)
      {
           NewNode->coeff = Poly1->coeff - Poly2->coeff;
           NewNode->pow = Poly1->pow;
           Poly1 = Poly1->Next;
           Poly2 = Poly2->Next;
      }
      else if(Poly1->pow > Poly2->pow)
      {
           NewNode->coeff = Poly1->coeff;
           NewNode->pow = Poly1->pow;
           Poly1 = Poly1->Next;
      }
      else if(Poly1->pow < Poly2->pow)
      {
           NewNode->coeff = -(Poly2->coeff);
           NewNode->pow = Poly2->pow;
           Poly2 = Poly2->Next;
      }

      NewNode->Next = NULL;
      Position->Next = NewNode;
      Position = NewNode;
}

while(Poly1 != NULL || Poly2 != NULL)
{
      NewNode = malloc(sizeof(Poly));
      if(Poly1 != NULL)
      {
           NewNode->coeff = Poly1->coeff;
           NewNode->pow = Poly1->pow;
           Poly1 = Poly1->Next;
      }
      if(Poly2 != NULL)
      {
           NewNode->coeff = -(Poly2->coeff);
           NewNode->pow = Poly2->pow;
           Poly2 = Poly2->Next;
      }
      NewNode->Next = NULL;
      Position->Next = NewNode;
      Position = NewNode;
}
```

```
}
```

OUTPUT

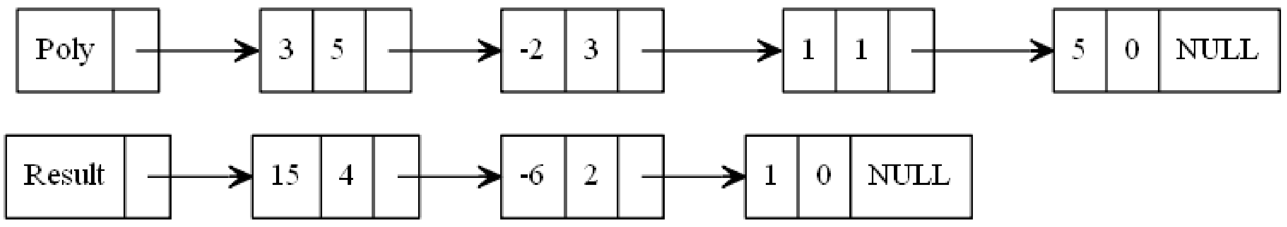
```
Enter the values for first polynomial :
Enter the coefficient : 3
Enter the power : 2
Enter 1 to continue : 1
Enter the coefficient : 4
Enter the power : 1
Enter 1 to continue : 1
Enter the coefficient : -2
Enter the power : 0
Enter 1 to continue : 0
The polynomial equation is : 3x^2+4x^1-2x^0
Enter the values for second polynomial :
Enter the coefficient : -7
Enter the power : 2
Enter 1 to continue : 1
Enter the coefficient : -10
Enter the power : 1
Enter 1 to continue : 1
Enter the coefficient : 17
Enter the power : 0
Enter 1 to continue : 0
The polynomial equation is : -7x^2-10x^1+17x^0
The polynomial equation subtraction result is : 10x^2+14x^1-19x^0
```

REVIEW QUESTIONS

1. Write the routine for polynomial subtraction.

CHAPTER - 10 - POLYNOMIAL DIFFERENTIATION

10.1 EXAMPLE



10.2 ROUTINE

```
void Differentiation(Poly *Poly1, Poly *Result)
{
        Poly *Position;
        Poly *NewNode;
        Poly1 = Poly1->Next;
        Result->Next = NULL;
        Position = Result;
        while(Poly1 != NULL)
        {
                NewNode = malloc(sizeof(Poly));
                NewNode->coeff = Poly1->coeff * Poly1->pow;
                NewNode->pow = Poly1->pow - 1;
                Poly1 = Poly1->Next;
                NewNode->Next = NULL;
                Position->Next = NewNode;
                Position = NewNode;
        }
}
```

10.3 PROGRAM

```
#include <stdio.h>
#include <stdlib.h>

struct poly
{
        int coeff;
        int pow;
        struct poly *Next;
};
typedef struct poly Poly;

void Create(Poly *List);
void Display(Poly *List);
void Differentiation(Poly *Poly1, Poly *Result);
int main()
{
        Poly *Poly1 = malloc(sizeof(Poly));
        Poly *Result = malloc(sizeof(Poly));
```

```
	Poly1->Next = NULL;
	printf("Enter the values for polynomial :\n");
	Create(Poly1);
	printf("The polynomial equation is : ");
	Display(Poly1);
	Differentiation(Poly1, Result);
	printf("\nThe polynomial differentiation equation is : ");
	Display(Result);
	return 0;
}

void Create(Poly *List)
{
	int choice;
	Poly *Position, *NewNode;
	Position = List;
	do
	{
		NewNode = malloc(sizeof(Poly));
		printf("Enter the coefficient : ");
		scanf("%d", &NewNode->coeff);
		printf("Enter the power : ");
		scanf("%d", &NewNode->pow);
		NewNode->Next = NULL;
		Position->Next = NewNode;
		Position = NewNode;
		printf("Enter 1 to continue : ");
		scanf("%d", &choice);
	} while(choice == 1);
}

void Display(Poly *List)
{
	Poly *Position;
	Position = List->Next;
	while(Position != NULL && Position->pow >= 0)
	{
		printf("%dx^%d", Position->coeff, Position->pow);
		Position = Position->Next;
		if(Position != NULL && Position->coeff > 0)
		{
			printf("+");
		}
	}
}
```

```c
void Differentiation(Poly *Poly1, Poly *Result)
{
	Poly *Position;
	Poly *NewNode;
	Poly1 = Poly1->Next;
	Result->Next = NULL;
	Position = Result;
	while(Poly1 != NULL)
	{
		NewNode = malloc(sizeof(Poly));
		NewNode->coeff = Poly1->coeff * Poly1->pow;
		NewNode->pow = Poly1->pow - 1;
		Poly1 = Poly1->Next;
		NewNode->Next = NULL;
		Position->Next = NewNode;
		Position = NewNode;
	}
}
```

Output

```
Enter the values for polynomial :
Enter the coefficient : 3
Enter the power : 5
Enter 1 to continue : 1
Enter the coefficient : -2
Enter the power : 3
Enter 1 to continue : 1
Enter the coefficient : 1
Enter the power : 1
Enter 1 to continue : 1
Enter the coefficient : 5
Enter the power : 0
Enter 1 to continue : 0
The polynomial equation is : 3x^5-2x^3+1x^1+5x^0
The polynomial differentiation equation is : 15x^4-6x^2+1x^0
```

REVIEW QUESTIONS

1. Write the routine for polynomial differentiation.

11.1 INTRODUCTION

A university with 40,000 students and 2,500 courses needs to be able to generate two types of reports. The first report lists the class registration for each class, and the second report lists, by student, the classes that each student is registered for.

The obvious implementation might be to use a two-dimensional array. Such an array would have 100 million entries. The average student registers for about three courses, so only 120,000 of these entries, or roughly 0.1 percent, would actually have meaningful data.

What is needed is a list for each class, which contains the students in the class. We also need a list for each student, which contains the classes the student is registered for. Figure shows our implementation.

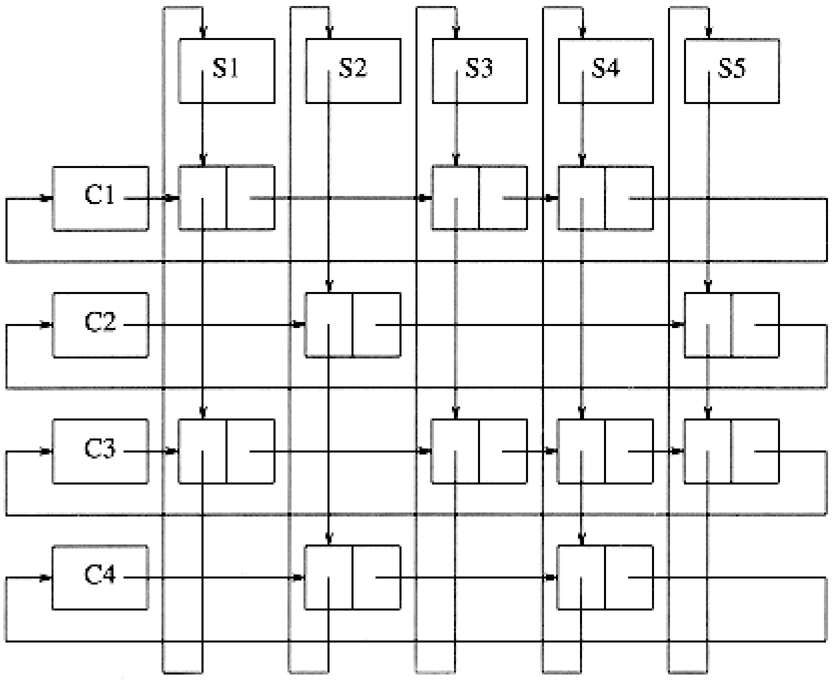


Fig. 11.1 Multilist implementation for registration problem

As the figure shows, we have combined two lists into one. All lists use a header and are circular. To list all of the students in class C3, we start at C3 and traverse its list (by going right). The first cell belongs to student S1. Although there is no explicit information to this effect, this can be determined by following the student's linked list until the header is reached. Once this is done, we return to C3's list (we stored the position we were at in the course list before we traversed the student's list) and find another cell, which can be determined to belong to S3. We can continue and find that S4 and S5 are also in this class. In a similar manner, we can determine, for any student, all of the classes in which the student is registered.

REVIEW QUESTIONS

1. What is multilist?

RAJALAKSHMI
ENGINEERING COLLEGE

Website : www.rajalakshmi.org

Elearning : www.rajalakshmicolleges.net/moodle

Give a man a fish and you feed him for a day.

Teach him how to fish and you feed him for a lifetime.