UNIT II

Data Structure:

Data structure is basically a group of data elements that are put together under one name. It also defines a particular way of storing and organizing data in a computer, so that it can be used efficiently.

Data Structure is the structural representation of logical relationships between data or element. It represents the way of storing, organizing and retrieving data.

It is classified as

- **Linear data structure**
- **Non Linear data structure**

Examples:

Linear : Lists, Stacks, Queues etc.

Non Linear : Trees, graphs etc.

Normally all the data structures can be implemented using 2 methods

- Array implementation
- Linked List implementation

Applications of data structure:

- Compiler design
- Statistical analysis package
- Numerical Analysis
- Artificial Intelligence
- Operating System
- Data base management system
- Simulation
- Graphics

Abstract Data Type:

Abstract data type is a set of operations of data. It also specifies the logical and mathematical model of the data type . ADT specification includes description of the data, list of operations that can be carried out on the data and instructions how to use these instructions. But ADT does not mention how the set of operations is implemented.

Examples for ADT: lists, sets and graphs along with their operations for ADT.

Examples for data type: Integers, reals and Booleans.

List ADT:

A list is a collection of elements. It can be represented as A_1, A2, A3…...A_N and its size is N. A list with size 0 is called as an empty list.

For any list except the empty list, we say that A_{i+1} follows A_i (i<N) and A_{i-1} precedes A_i(i>1). The first element of the list is A_1 and the last element is A_N. the position of A_i in a list is i.

Some popular operations on the list are

- **Find** which returns the position of the first occurrence of an element or key.
- **Insert** which inserts an element in a given position.
- **Delete** which deletes an element from the list.

- **FindKth** which returns the element in K^{th} position.

Example: Consider 34,12,52,16,12 as a list , then

Find (52) returns 3

After **Insert (70,3)** the list is 34,12,70,52,16,12

After **Delete (52)** the list is 34,12,70,16,12

Implementation of list ADT:

There are 2 ways of implementation

- Using Arrays
- Using Linked list

List ADT –Array based Implementation:

A list can be implemented using an array. An array is a collection of similar data items or elements. In the array implementation maximum size of the array is required earlier. That is the maximum number of elements in the array is fixed and it can be modified only through the code.

Since the maximum size should be known earlier, required high estimate, which wastes considerable space.

- An Array implementation allows **printlist** and **Find** to be carried out in **linear time**.
- **Findkth** takes **constant time**.
- **Insertion** and **Deletion** are expensive.

For Example inserting an element at position 0 (i.e) first positions requires shifting the entire elements one position to right.

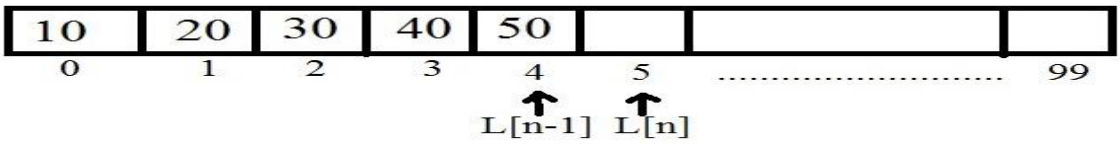
For deleting the first element requires shifting all the elements one position to the left.

Operations:

- Insertion -This inserts a given element X in to the List.
  - Insertion at First.
  - Insertion at any position.
  - Insertion at last.
- Deletion -> deletes a given element from the list
- Find - This returns the position of the first occurrence of an element or key.
- Findkth -returns the element in the k^{th} position.

Consider an array L[100] and insert 'n' elements into the array.

Assume n is 5 and elements are 10,20,30,40,50

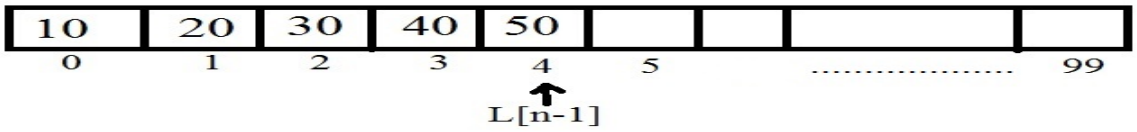


The elements are placed as above and the last element present at location L[n-1] (i.e) L[4] .

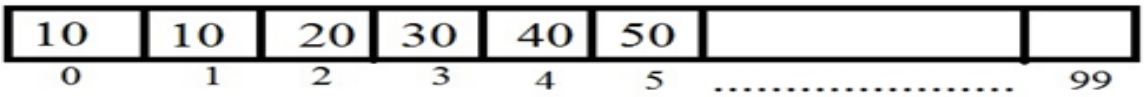
INSERTION

a) **Insertion at First :**

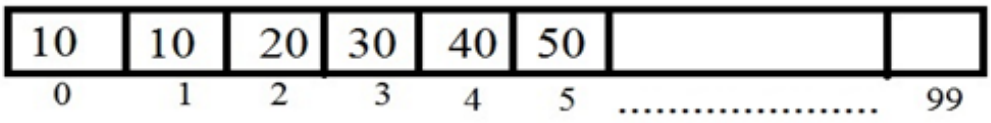
Consider a new element 'X' is to be inserted at the first position (i.e) at L[0]. For this ,shift all the elements to the right one bit position. Let us start shifting from the last location (i.e) L[n-1].



After shifting the array looks like as below.



Now inserts the new element in to first location that is L[0]=X; Assume X=100,Then



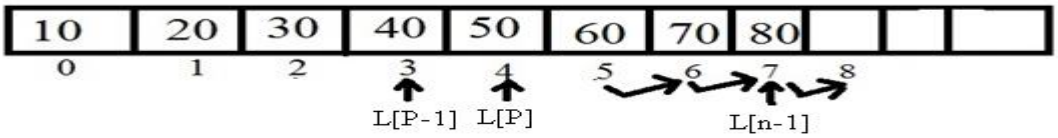
After insertion the no of elements in the list will be increased by one. Therefore n will be updated as n+1 .

Routine:

```
void InsertFirst(int L[],int x,int n)
{
     int n;
     for(i=n-1;i>=0;i--)  //shifting the
           L[i+1]=L[i];    //   elements
     L[0]=x;
    n=n+1;
}
```

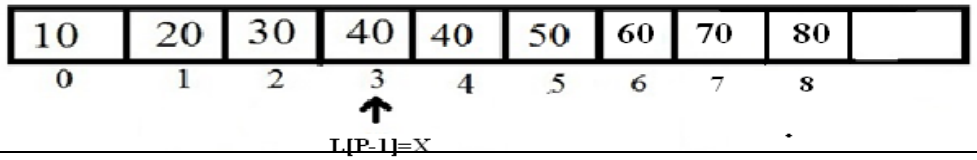
b) Inserting at any position :

Insert an element 'X' at position 'p' , the elements from the next position will be shifted to the right and then the new element will be inserted. Assume p=4 (i.e) 4th position. In array 4th is L[3] .

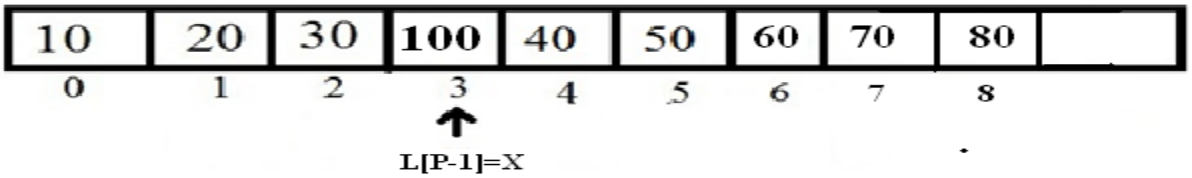


Shift all the elements from p-1th location upto L[n-1].

After shifting, the array looks like as below



Now X can be inserted at position L[p-1] i.e. L[p-1]=X and update the n value i.e. n+1 .Assume the value of X=100,Then the array is

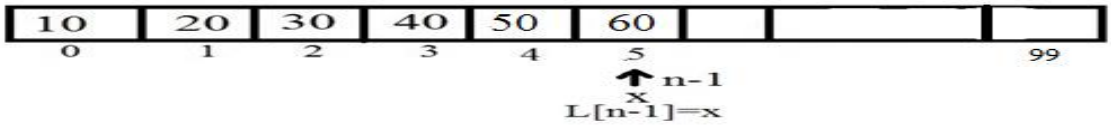


Routine :

```
void InsertAtAny(int L[] , int x , int p , int n)
{
   for(i=n-1;i>p-1;i--)
        L[i+1]=L[i];
   L[p-1]=x;
   n=n+1;
}
```

c) Insertion at Last :

To insert an element at last no shifting is required. Just assign the new element to the position L[n-1].

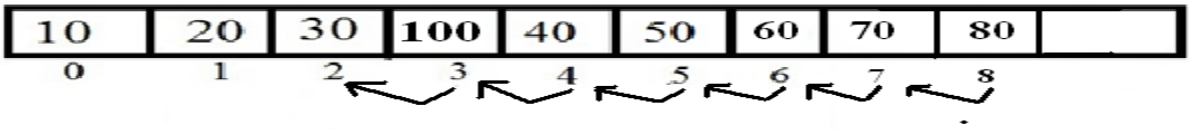


Routine:

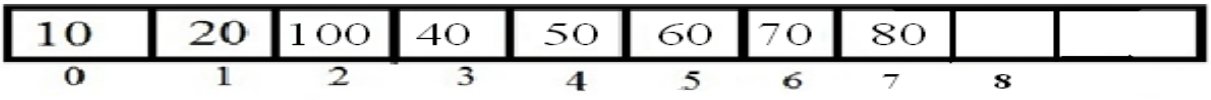
```
void InsertLast(int L[] , int x , int n )
{
   L[n-1]=x;
   n=n+1;
}
```

DELETION:

In deletion a given element will be removed from the list. For this initially the element is searched and the location of the element is identified, then the element is removed by shifting all the elements to the left from next location of the element to be deleted to the next location of the element to be deleted. Assume the element to be deleted 30,



After deletion,



Update the value of n (i.e) n-1.

```
void   Delete(int L[] , int x , int n)
{
   for(i=0;i<n;i++)
       if(L[i]==x)
            break;
  for(j=i+1;j<n;j++)
        L[j-1]=L[j];
  n=n-1;
}
```

FIND:

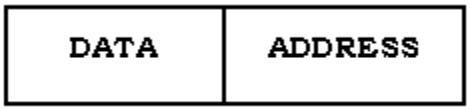
This operation checks whether the given element is present or not .

```
void Find(int L[] , int x , int n)
{
int flag=0;
for(i=0;i<n;i++)
        if(L[i]==x)
  flag=1;
if(flag==1)
  printf("The element is present");
else
  printf("The element is Not Present");

}
```

LINKED LIST:

Linked List is a list, which is implemented using structures and pointers. It can also be defined as group nodes or series of structures where each node has minimum 2 fields.

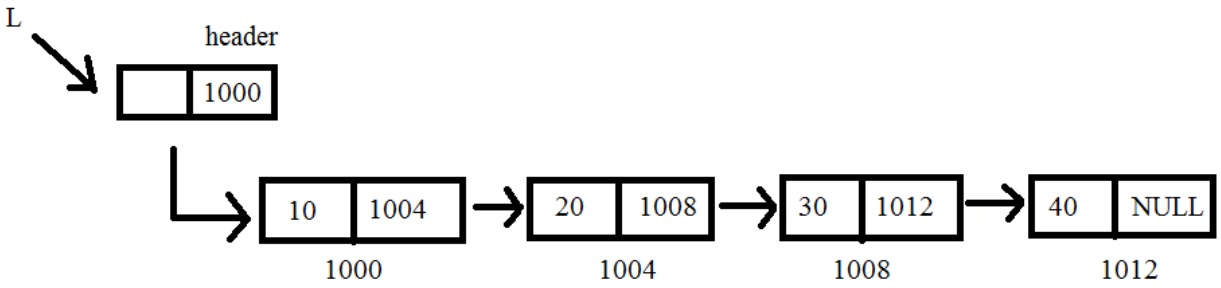
- **Data field :** Stores the actual information
- **Address field :** Points the next node



Node Declaration

```
struct Node
{
    ElementType Element;
    Position Next;
}
```

Example: Singly Linked list



The link field of the last node points to NULL which indicates end of linked list. In order to avoid the linear cost of insertion and deletion, the elements are not stored contiguously in linked list.

Operations

- Insertion
- Deletion
- Find
- FindPrevious
- DeleteList

Advantages:

- Linked lists allow dynamic memory management by allowing elements to be added or deleted at any time during program execution.
- Efficient utilization of memory space. Memory space is reserved only for the elements in a linked list.
- Easy to insert or delete elements in a linked list.

Disadvantages:

- Each element in the linked list requires more space when compared with array.
- Accessing an element is more difficult, Since to access an element it is mandatory to traverse all the preceding elements.

Types of Linked List:

3 types

- Singly Linked List
- Doubly Linked List
- Circularly Linked List

SINGLY LINKED LIST

Singly linked list consists of a series of structures or nodes. Each node contains only 2 fields.

Data field: Actual information

Address field: Address of next node

DATA	ADDRESS

NODE DECLARATION

```
struct Node
{
```

```
    ElementType Element;
    Position Next;
}
```

Type Declarations:

```
    struct Node;
    typedef struct Node *ptrToNode;
    typedef ptrToNode List;
    typedef ptrToNode Position;

    List MakeEmpty(List L);
    int IsEmpty(List L);
    int IsLast(Position P,List L);
    Position Find(ElementType x,List L);
    void Delete(ElementTypr x,List L) ;
    Position FindPrevious(ElementType x.List L);
    void Insert(ElementType x,List L,Position P);
    void DeleteList(List L);
```

OPERATIONS

- Insertion
- Deletion
- Find
- FindPrevious
- DeleteList

INSERTION

Insertion is the process of inserting node at the given position in the linked list. A new node can be inserted after node P. Here, P is the address of the node after which the new node is to be inserted.

For insertion, create a node first from Node structure and assign the value X to its data field.

Newnode = (struct Node *) malloc (sizeof(struct Node))

Newnode → Element = X

- Then perform the following

  Newnode →Next = P → Next

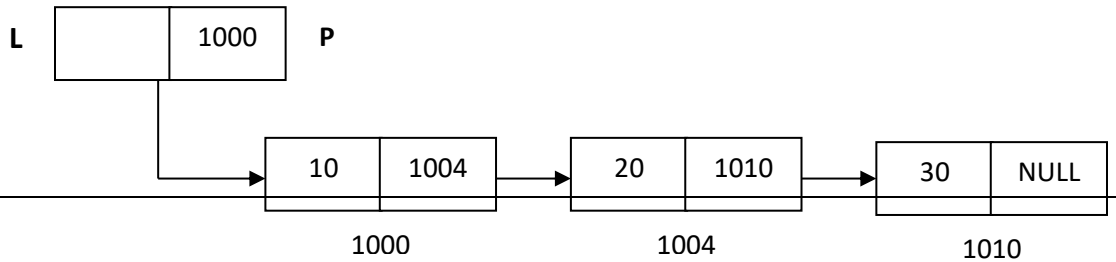
  P → Next= Newnode

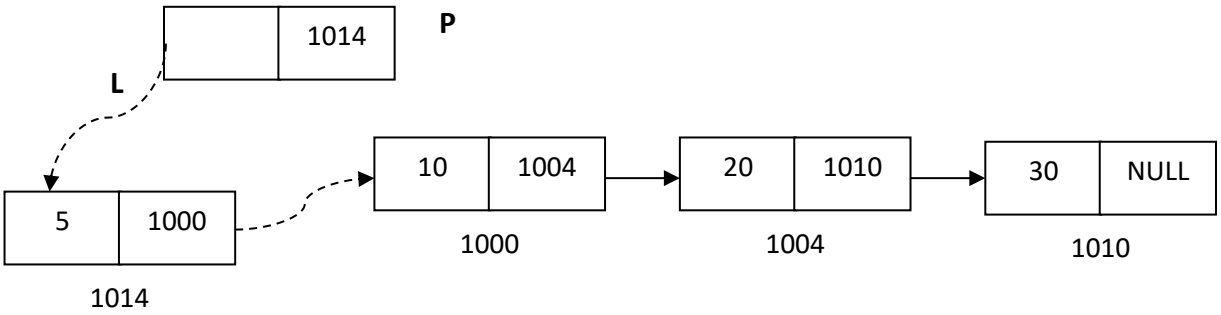
Insertion can be done at any position

a. Insertion at first
b. Insertion at any position
c. Insertion at last

a) INSERTION AT FIRST

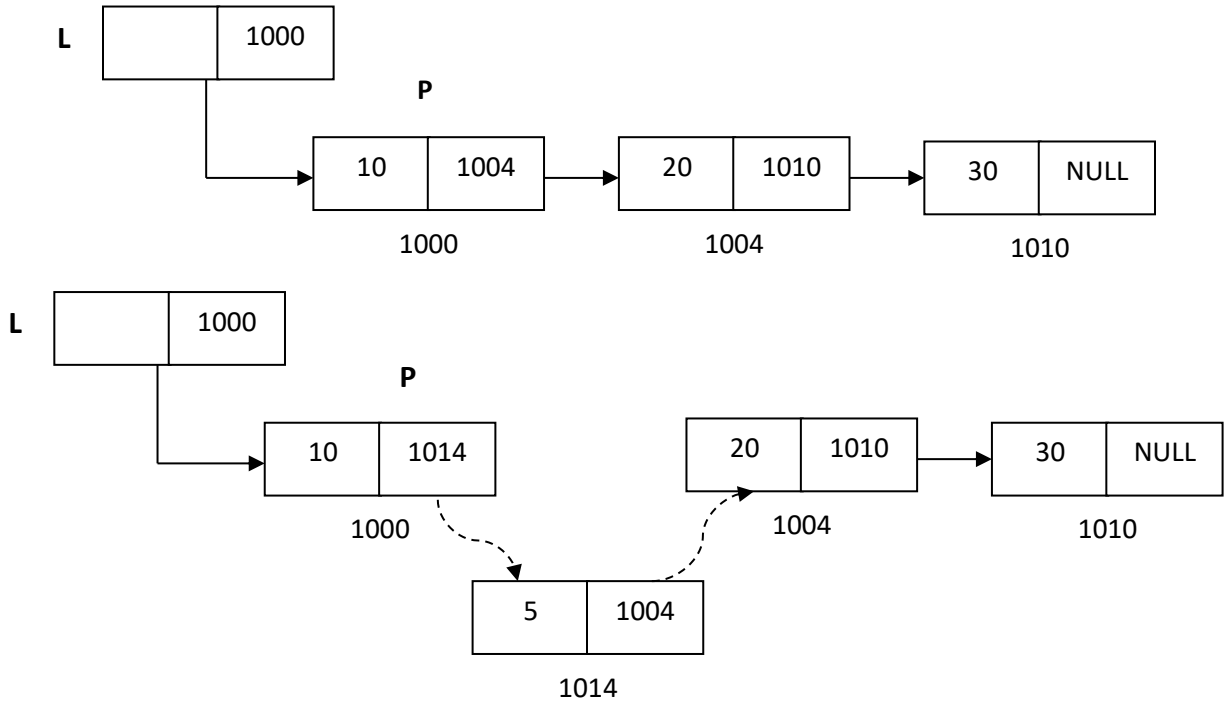
Insert(5,P)





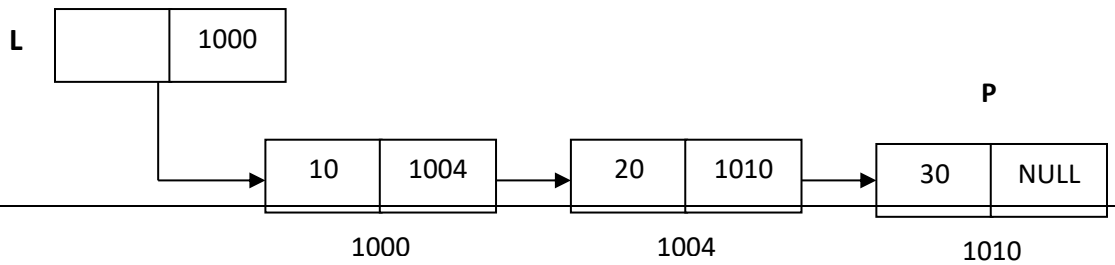
b) INSERTION AT MIDDLE

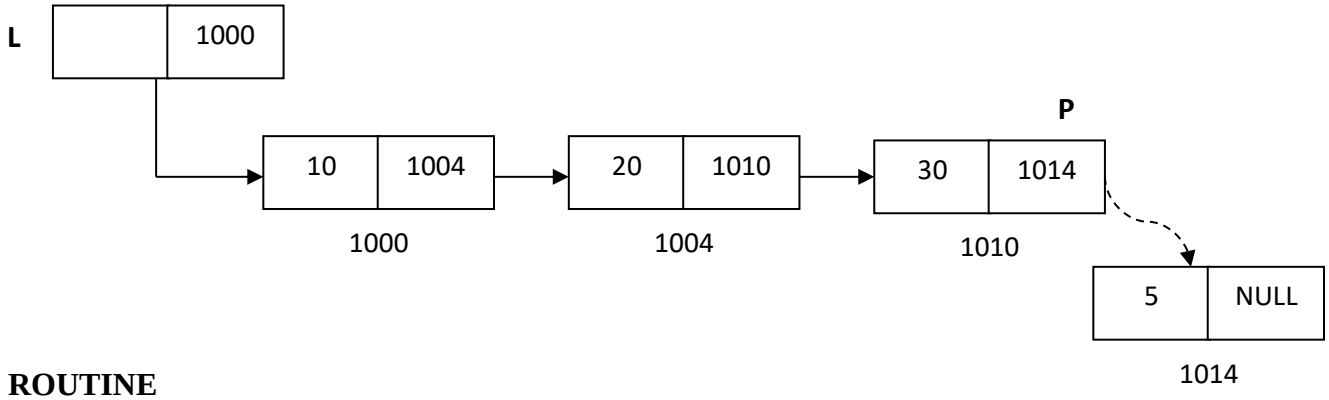
Insert(5,P)



c) INSERTION AT LAST

Insert(5,P)





ROUTINE

Routine to Insert at First	Routine to Insert at Last	Routine to insert at Any Position

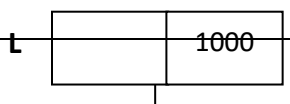
void **InsertFirst**(ElementType X, List L)
{
Position NewNode;
NewNode =malloc (sizeof (struct Node));
if(NewNode==NULL)
FatalError("Out of Space");
else
{
NewNode → Element = X;
NewNode → Next = L → Next;
L → Next=NewNode;
 	
}
}
	void **InsertLast**(ElementType X, List L)
	{
	Position NewNode,P;
	NewNode =malloc (sizeof (struct Node));
	if(NewNode ==NULL)
	FatalError("Out ofSpace");
	else
	{
	P = L → Next;
	while(Temp → Next != NULL)
	P = P → next;
	NewNode → Next=X;
	NewNode → Next=NULL;
	P → Next = NewNode;
	}
	}
		void **Insert** (ElementType X, List L, Position P)
		{
		Position NewNode;
		NewNode =malloc (sizeof (struct Node));
		if(NewNode==NULL)
		FatalError("Out of Space");
		else
		{
		NewNode → Element = x;
		NewNode ->Next = P → Next;
		P → Next = NewNode;
		}
		}

DELETION

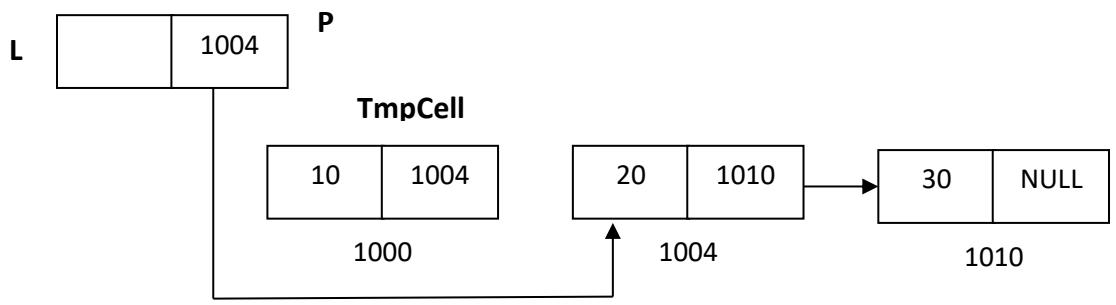
Deletion is the process of removing an element from the linked list. A node which is having the given element X will be removed from the list. For that, address of the previous node which contains X is needed. Address of the previous node of X is identified using the FindPrevious routine and then assign it to P.

- Assign the address of the node to be deleted to TmpCell.
  TmpCell=P→ Next;
- Then do the following
  P→ Next = TmpCell→Next
- After the above assignments the node having X will be removed from the list.
- Deallocate the memory of removed node using **free(TmpCell),**which is a predefined function in C.
- Element from any position can be deleted.
  - At first
  - At last
  - At any position

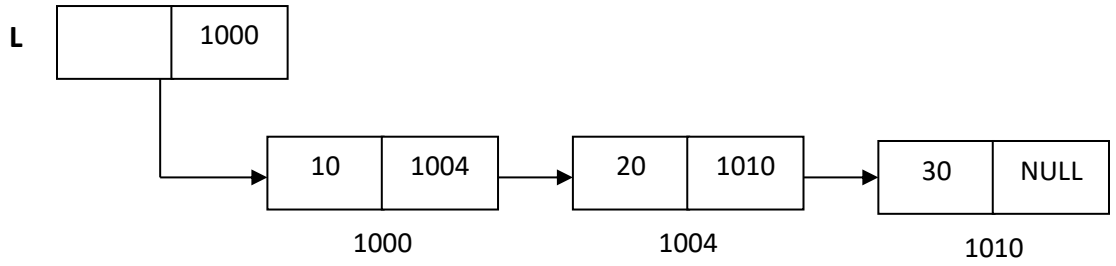
a) DELETION AT FIRST



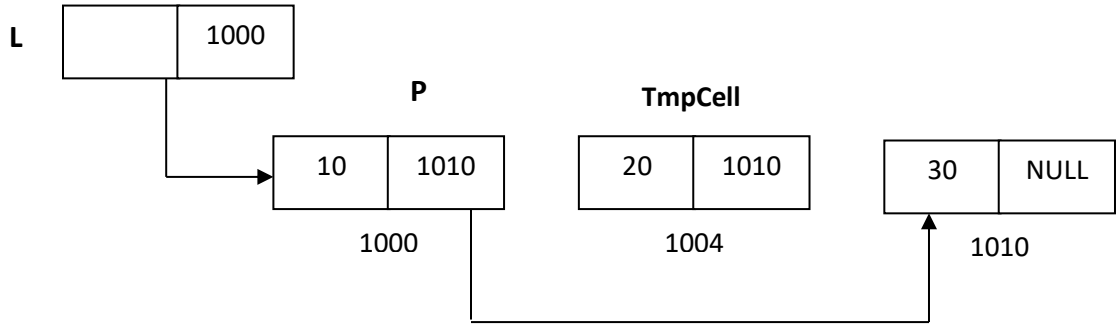
Delete(10)



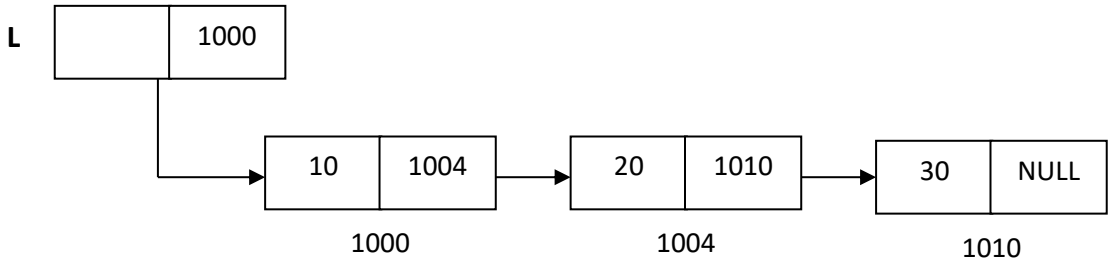
b) DELETION AT ANY POSITION



Delete(20)



c) DELETION AT LAST



Delete(30)

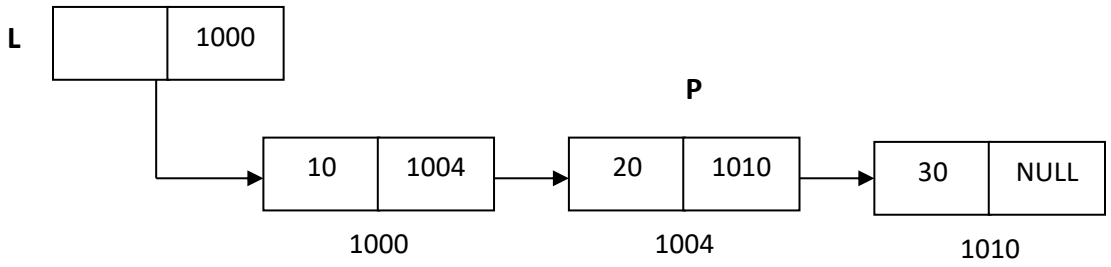
Routine to Delete from First	**Routine to Delete from Last**	Routine to **Delete from** Any Position

void deleteBeginning(List L)
{
Position TmpCell;
if(L → Next!=NULL)
{
TmpCell = L → Next;
L → next = TmpCell → Next;
free(TmpCell);
}
}
	void deleteEnd(List L)
	{
	Position P,TmpCell;
	P=L → **Next;**
	while(P → **Next** → **Next!=NULL**)
	P=P → **Next**;
	TmpCell = P → Next;
	P → **N**ext=NULL;
	free(TmpCell);
	}
		void Delete (ElementType X, List L)
		{
		Position P,TmpCell;
		P = FindPrevious(X,L);
		if(P → Next!=NULL)
		{
		TmpCell = P → Next;
		P → next = TmpCell->Next;
		free(TmpCell);
		}
		}

FIND

Find is the process of identifying the location of particular element in the linked list. This function will return the position of the given element.

Find(20)



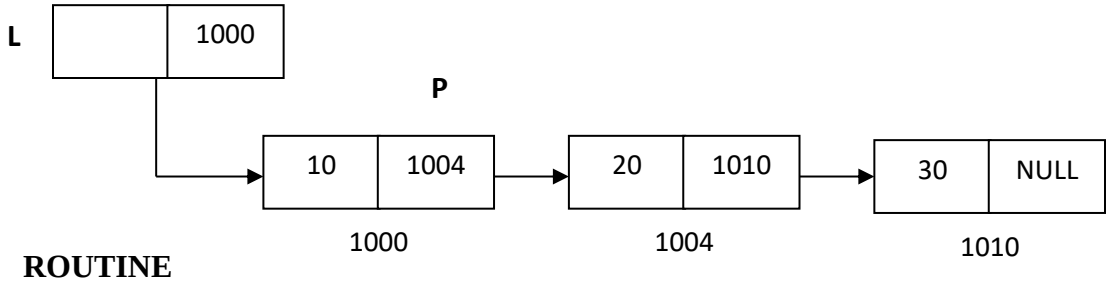
ROUTINE

```
Position Find ( ElementType X, List L )
{
    Position P;
    P = L → next;
    while( P != NULL && P → Element != x) )
        P = P → next;
     return P;
}
```

FIND PREVIOUS

Find is the process of identifying the previous location of particular element in the linked list. This function will return the previous position of the given element.

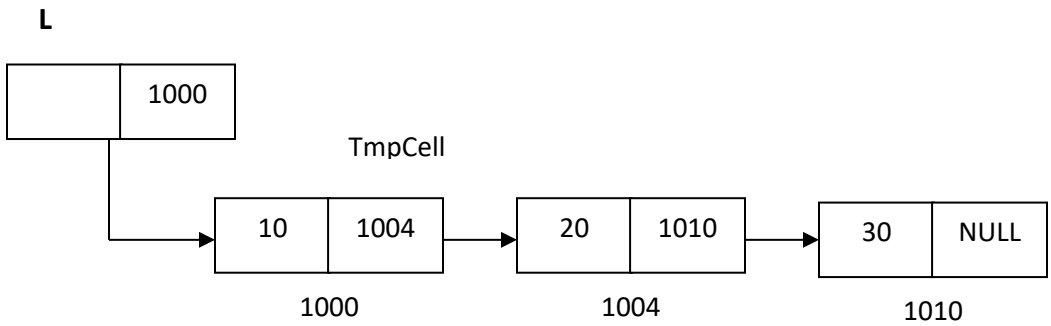
FindPrevious(20)

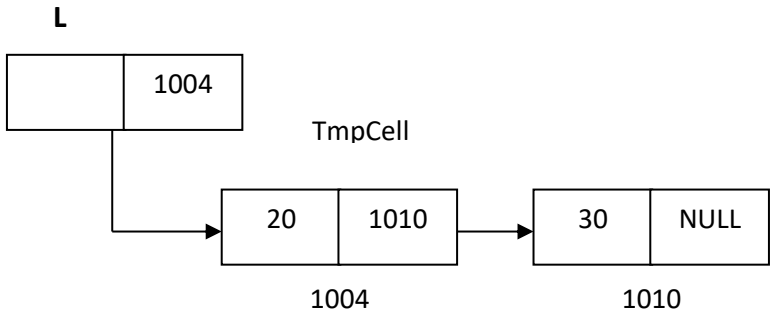


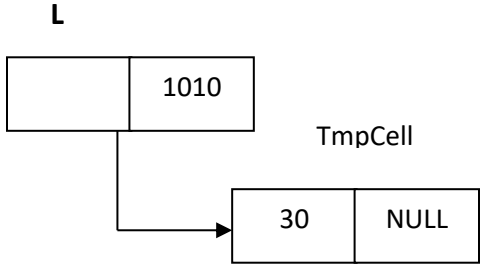
ROUTINE

```
Position FindPrevious ( ElementType X, List L )
{
        Position P;
        P = L;
        while( P → Next != NULL && P → Next → Element != x) )
                P = P → next;
        return P;
}
```

DELETINING ALL THE ELEMENTS IN THE LIST





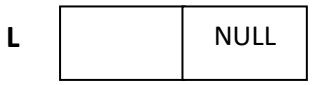


ROUTINE

```
void DeleteList( List L )
```

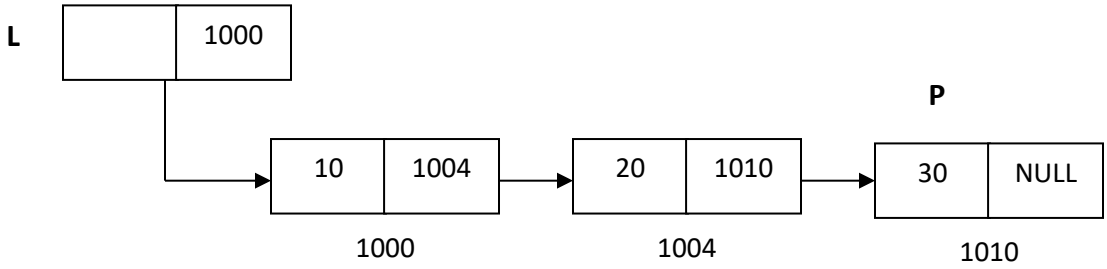
```
{
        position P, tmp;
        P = L → next; /* header assumed */
        L → next = NULL;
        while( p != NULL )
        {
                TmpCell = p → next;
                free( p );
                p = tmp;
        }
}
```

CHECKING WHETHER THE SINGLY LINKED LIST IS EMPTY




```
int  IsEmpty(List L)
{
        return L → Next==NULL;
}
```

CHECKING WHETHER THE GIVEN NODE P IS LAST



ROUTINE

```
int IsLast( Position P , List L )
{
        return( P → Next == NULL );
}
```

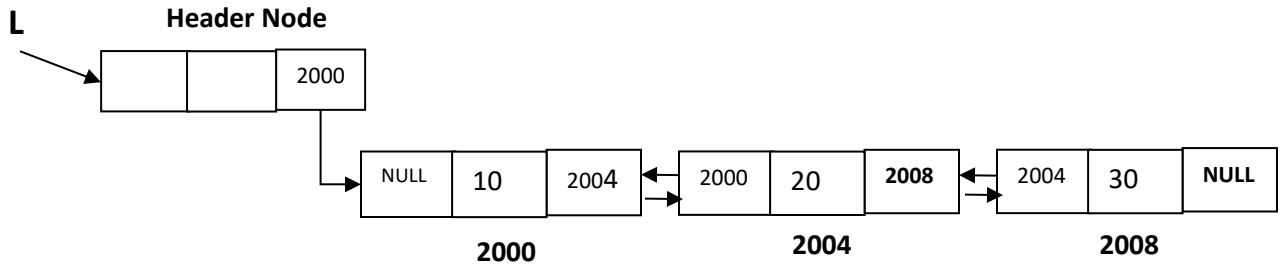
DOUBLY LINKED LIST

Doubly linked list is a list, which is implemented using structures and pointers. Each structure has three fields

- **Data :** Actual Information
- **Pointer to previous structure:** Points the previous node in the list.
- **Pointer to next structure:** Points the next node in the list.

It can be represented as

Example:



Advantages:

Singly linked list cannot be traversed in the backward direction. But it is convenient to traverse the doubly linked list backwards. To do this, an extra field to point the previous structure is added.

Disadvantages:

This requires an extra link to point the previous node which doubles the cost of insertions and deletions because there are more pointers to fix. But it simplifies deletion.

Operations:

The following operations can be performed on a doubly linked list

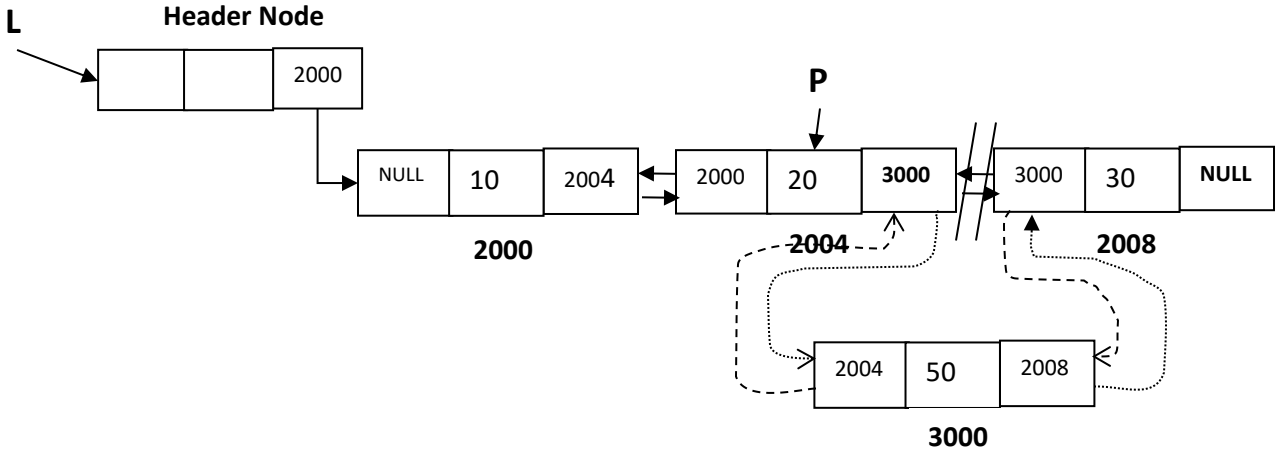
- **Insertion:** Inserts an element in a given position.
- **Deletion:** Deletes an element along with it node.
- **Find:** Finds a given element and returns the address of its node.

INSERTION:

- A new node can be inserted after node P.
- P is the address of the node after which the new node is to be inserted.
- For insertion create a new node in **TmpCell** and assign the ne value X to the node.

  TmpCell = (struct Node *)malloc(sizeof(struct Node))
- Then perform the following

$$\mathbf{TmpCell \rightarrow Next = P \rightarrow Next}$$
$$\mathbf{TmpCell \rightarrow Prev = P}$$
$$\mathbf{P \rightarrow Next = TmpCell}$$
$$\mathbf{P \rightarrow Next \rightarrow Prev = TmpCell}$$

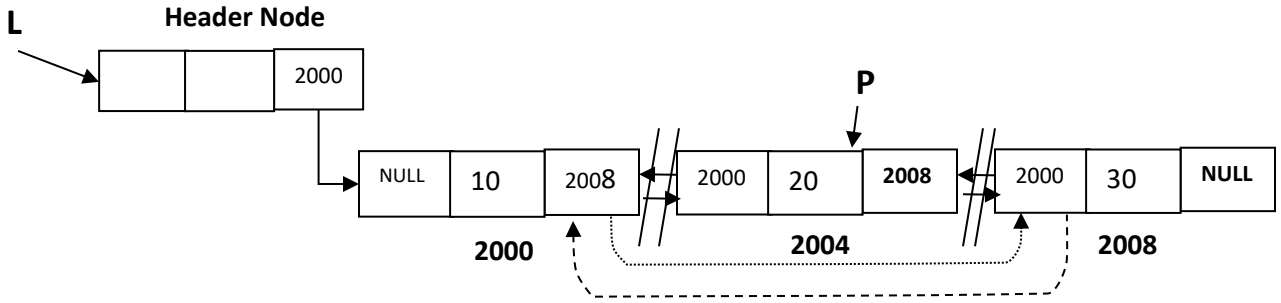


DELETION:

- A node which is having the given element X will be removed from the list.
- Find the address of the node to be deleted by traversing the list and assign the address to P.
- Then do the following

  $P \rightarrow$ **Prev** $\rightarrow$ **Next = P** $\rightarrow$ **Next**

  $P \rightarrow$ **Next** $\rightarrow$ **Prev=P** $\rightarrow$ **Next**
- After the above assignments the node P will be removed from the list.
- Deallocate the memory of removed node using **free(P),**which is a predefined function in C.



FIND:

- Finds a given element X from the list by traversing and returns the address of it.

Programming Details (Routines):

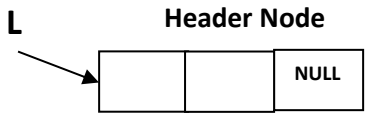
```
struct Node;
typedef struct Node *PtrToNode;
typedef PtrToNode DList;
typedef PtrToNode Position;

int IsEmpty(DList L);
int IsLast(Position P, DList L);
Position Find (Element Type X, DList L);
void Delete (Element Type X, DList L);
Position FindPrevious (Element Type X, DList L);
void Insert (Element Type X, DList L, Position P);
void DeleteList (DList L);

struct Node
{
        ElementType Element;
        Position Next;
        Position Prev;
};
```

Function to test whether the doubly linked list is empty

```
Int  IsEmpty(DList L)
{
        return L → Next==NULL;
}
```

```
if(L → Next==NULL)
        return 1;
else
        return 0;
```

This function checks whether the given list is empty or not. If it is empty it return 1 otherwise 0.

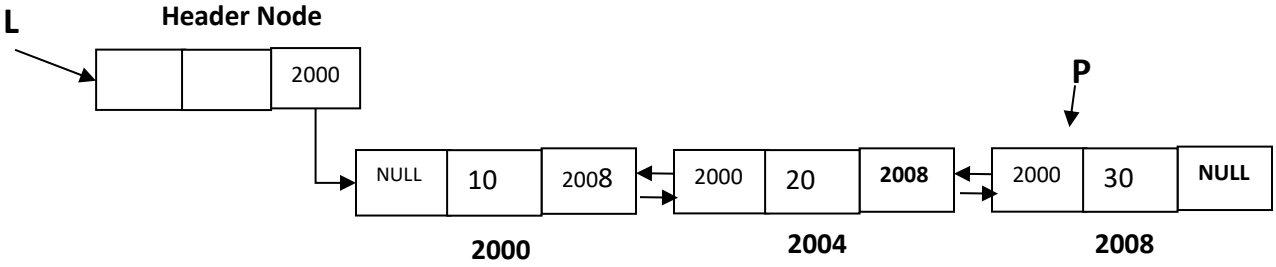


It is done by checking the Next pointer of the header. If it is NULL, then the list is empty

Function to test whether the given node is the last:

```
Int  IsLast(Position P, DList L)
{
        return P → Next==NULL;
}
```

```
if(P → Next==NULL)
        return 1;
else
        return 0;
```



Function for Find operation:

```
Position Find (ElementType X, DList L)
{
        Position P;
        P=L → Next;
        while(P!=NULL && P → Element!=X)
                P=P → Next;
        return P;
}
```

This function returns the address of the node in which the X present

Function for Deletion:

```
Position Delete (ElementType X, DList L)
{
   Position P;
   P=L → Next;
   while(P!=NULL && P → Element!=X)
         P=P → Next;
   P → Prev → Next=P → Next;
    P → Next → Prev=P → Prev;
    free(P)
          }
```

Function for Insertion:

```
void Insert ( ElementType X, List L, Position P )
{
Position NewNode;
NewNode =malloc (sizeof (struct Node));
if(NewNode==NULL)
        FatalError("Out of Space");
else
{
        NewNode  → Element = x;
        NewNode → Prev=P;
        NewNode → Next=P → Next;
        P->Next=NewNode;
        NewNode->Next->Prev=NewNode;
```

Function for DeleteList:

This function deletes all the elements of a list one by one

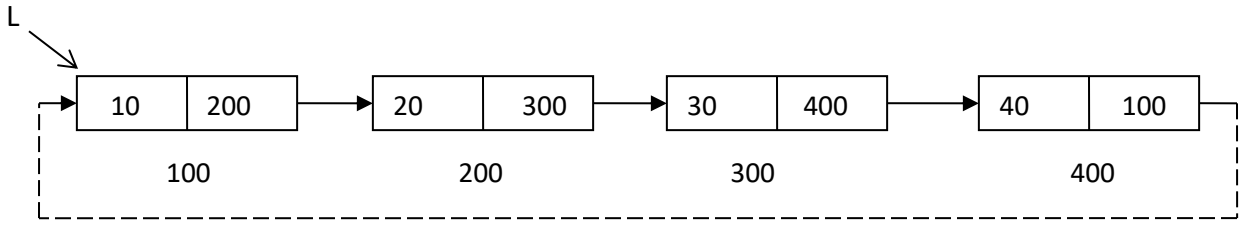
CIRCULAR LINKED LIST

Circular linked list a linked list, in which the next field of the last node points the first node. It can also be done with doubly linked list. It can be done with or without header node

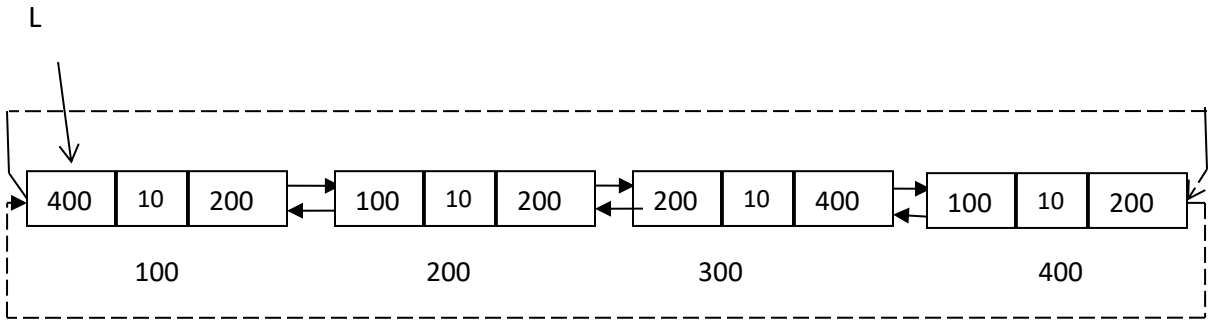
Circular linked lists can be used to traverse the same list again and again if needed. In a circular linked list there are two methods to know if a node is the first node or not.

- Either an external pointer, **List L**, points the first node or
- A header node is placed as the first node of the circular list.

A circularly singly linked list without header

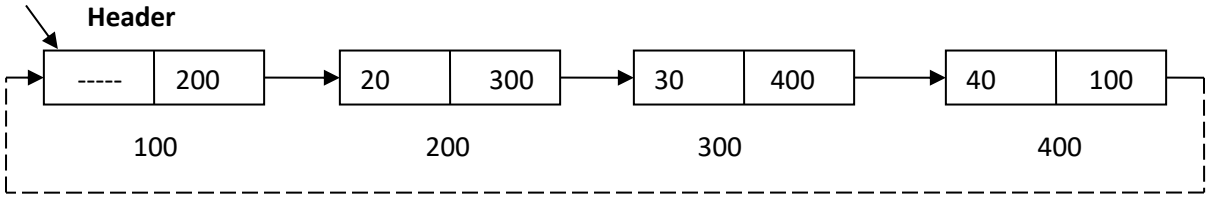


A double circularly linked list without header



A circularly singly linked list with header

L



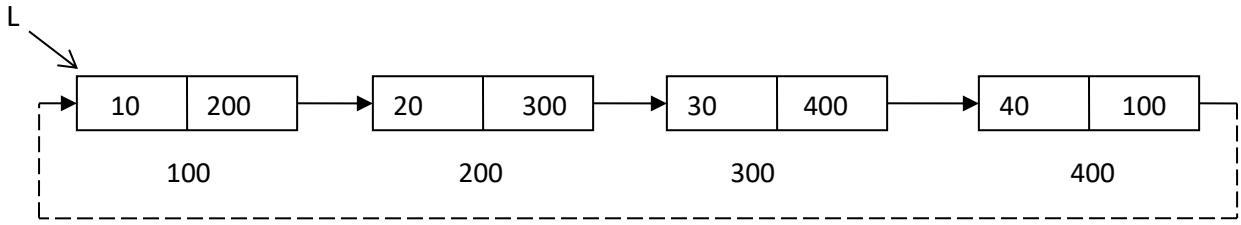
Various Operations

- Insertion
- Deletion
- Find
- Display all the elements

SINGLY CIRCULAR LINKED LIST:

In a normal singly linked list, for accessing any node of linked list, we start traversing from the first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of singly linked list. In a singly linked list, next part (pointer to next node) is NULL, if we utilize this link to point to the first node then we can reach preceding nodes.

In circularly linked list the pointer of last node points the address of the first node. It allows quick access to the first and last node. It can be implemented using singly linked list or doubly linked list.



OPERATIONS

- Insertion
- Deletion
- Find
- FindPrevious
- DeleteList

Routine to Insert at First	Routine to Insert at Last	Routine to insert at Any Position

```
void InsertFirst( ElementType X, List L)
{
Position NewNode,P;
NewNode=malloc(sizeof(struct Node));
if(TmpCell==NULL)
        FatalError("Out of Space");
else
{
NewNode → Element = x;
NewNode ->Next = L;
L=NewNode;
while(Temp -> Next != L)
    P = P -> next;
P->Next=NewNode;
 }
}
```

```
void InsertLast( ElementType X, List L )
{
Position NewNode ,Temp;
NewNode=malloc (sizeof (struct Node));
if(NewNode ==NULL)
        FatalError("Out of Space");
 else
 {
   Temp = L;
  while(Temp -> Next != L)
     Temp = Temp -> next;
   Temp -> Next = NewNode ;
 NewNode -> next = L;
 }
}
```

```
void Insert ( ElementType X, List L, Position P )
{
Position NewNode;
NewNode =malloc (sizeof (struct Node));
if(NewNode ==NULL)
        FatalError("Out of Space");
else
{
NewNode → Element = x;
NewNode ->Next = P → Next;
P → Next = NewNode;
}
 }
```

Routine to Delete from First

```
void deleteBeginning(List L)
{
    Position TmpCell;
    if(L!=NULL)
    {
        TmpCell = L;
        L = TmpCell → Next;

         P=L;
         while(P → Next!=L)
             P=P → Next
         P->Next=TmpCell->Next;
         free( TmpCell);

    }
}
```

Routine to Delete from Last

```
void deleteEnd(List L)
{
 Position P,TmpCell;
 P=L;
 while(P → Next → Next!=L)
     P=P → Next;
 TmpCell = P → Next;
 P → Next = TmpCell → Next;
 free( TmpCell);
}
```

Routine to **Delete from** Any Position

```
void Delete ( ElementType X, List L )
{
        Position P,TmpCell;
        P = FindPrevious(X,L);
        if(P → Next!=L)
        {
          TmpCell = P → Next;
           P → next= TmpCell->Next;
          free( TmpCell);
        }
}
```

APPLICATIONS OF LIST OR LINKED LIST

1. Polynomial Addition
2. Radix Sort
3. Multi list
4. Buddy system
5. Dynamic memory allocation
6. Garbage Collection

POLYNOMIAL OPERATIONS

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + \ldots. + jx + k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

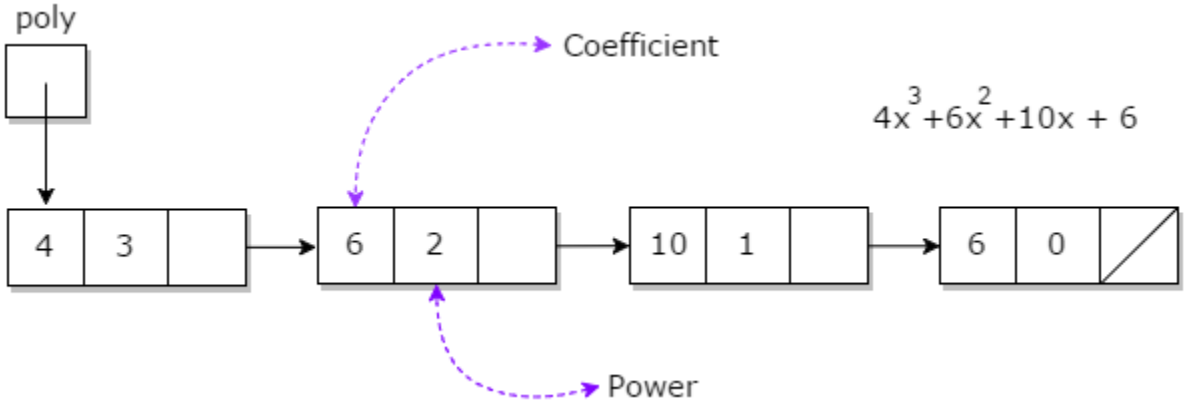
An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

- one is the coefficient
- other is the exponent

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- Additional terms having equal exponent is possible one
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent



Polynomial can be represented in the various ways. These are:

- By the use of arrays
- By the use of Linked List

Addition:

```
void polyadd(struct Node *poly1, struct Node *poly2, struct Node *poly)
{
while(poly1 && poly2)
   {
      // If power of 1st polynomial is greater then 2nd, then store 1st as it is and move its pointer
      if(poly1->pow > poly2->pow)
      {


         poly->pow = poly1->pow;
         poly->coeff = poly1->coeff;
         poly1 = poly1->next;
```

```
        }

        // If power of 2nd polynomial is greater then 1st, then store 2nd as it is
        // and move its pointer
        else if(poly1->pow < poly2->pow)
        {
            poly->pow = poly2->pow;
            poly->coeff = poly2->coeff;
            poly2 = poly2->next;
        }

        // If power of both polynomial numbers is same then add their coefficients
        else
        {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff+poly2->coeff;
            poly1 = poly1->next;
            poly2 = poly2->next;
        }

       // Dynamically create new node
        poly->next = (struct Node *)malloc(sizeof(struct Node));
        poly = poly->next;
        poly->next = NULL;
    }
while(poly1 || poly2)
    {
        if(poly1)
        {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff;
            poly1 = poly1->next;
        }
        if(poly2)
        {
            poly->pow = poly2->pow;
            poly->coeff = poly2->coeff;
            poly2 = poly2->next;
        }
        poly->next = (struct Node *)malloc(sizeof(struct Node));
        poly = poly->next;
        poly->next = NULL;
    }
}
```

Multiplication:

```
Node* multiply(Node* poly1, Node* poly2,
               Node* poly3)
{
```

```
    // Create two pointer and store the
    // address of 1st and 2nd polynomials
    Node *ptr1, *ptr2;
    ptr1 = poly1;
    ptr2 = poly2;
    while (ptr1 != NULL) {
       while (ptr2 != NULL) {
          int coeff, power;

          // Multiply the coefficient of both
          // polynomials and store it in coeff
          coeff = ptr1->coeff * ptr2->coeff;

          // Add the powerer of both polynomials
          // and store it in power
          power = ptr1->power + ptr2->power;

          // Invoke addnode function to create
          // a newnode by passing three parameters
          poly3 = addnode(poly3, coeff, power);

          // move the pointer of 2nd polynomial
          // two get its next term
          ptr2 = ptr2->next;
       }

             ptr2 = poly2;
        ptr1 = ptr1->next;
    }

       removeDuplicates(poly3);
    return poly3;
}
```