

CS3351 – DATA STRUCTURES

UNIT I LISTS

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 ADT – Radix Sort – Multilists.

Data:

A collection of facts, concepts, figures, observations, occurrences or instructions in a formalized manner.

Information:

The meaning that is currently assigned to data by means of the conventions applied to those data (i.e., processed data)

Record:

Collection of related fields.

Data type:

Set of elements that share common set of properties used to solve a program.

Data Structures:

Data Structure is the way of organizing, storing, and retrieving data and their relationship with each other.

Characteristics of data structures:

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.

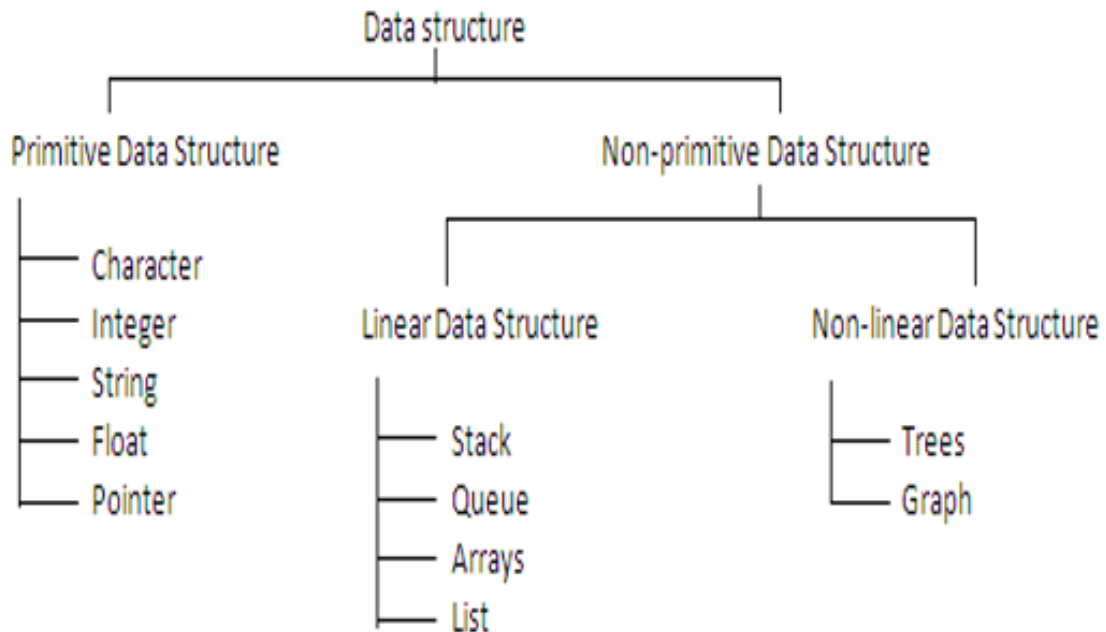
Operations on Data Structures:

- 1.Traversal
2. Search

3. Insertion

4. Deletion

CLASSIFICATION OF DATA STRUCTURES



Primary Data Structures/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

Secondary Data Structures/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.

Linear Data Structures:

Linear data structures are data structures in which, all the data elements are arranged in a linear or sequential fashion. Examples of data structures include arrays, stacks, queues, linked lists, etc.

Non-Linear Structures:

In non-linear data structures, there is 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 and graphs.

Static and dynamic data structure:

Static Data Structure:

If a data structure is created using static memory allocation, ie. data structure formed when the number of data items are known in advance, it is known as static data static data structure or fixed size data structure.

Dynamic Data Structure:

If the data structure is created using dynamic memory allocation i.e data structure formed when the number of data items are not known in advance is known as dynamic data structure or variable size data structure.

Application of data structures:

Data structures are widely applied in the following areas:

- / Compiler design
- / Operating system
- / Statistical analysis package
- / DBMS
- / Numerical analysis
- / Simulation
- / Artificial intelligence
- / Graphics

ABSTRACT DATA TYPES (ADTS):

An abstract Data type (ADT) is defined as a mathematical model with a collection of operations defined on that model. Set of integers, together with the operations of union, intersection and set difference form an example of an ADT. An ADT consists of data together with functions that operate on that data.

ADTs are like user defined data types which defines operation on values using functions without specifying what is there inside the function and how the operations are performed.

Advantages/Benefits of ADT:

- 1.Modularity
- 2.Reuse
- 3.code is easier to understand

4. Implementation of ADTs can be changed without requiring changes to the program that use the ADTs.

THE LIST ADT:

List is an ordered set of elements.

The general form of the list is A_1, A_2, \dots, A_N

A_1 - First element of the list

A_2 - 1st element of the list

N - Size of the list

If the element at position i is A_i , then its successor is A_{i+1} and its predecessor is A_{i-1}

Various operations performed on List

1. Insert ($X, 5$)- Insert the element X after the position 5.
2. Delete (X) - The element X is deleted
3. Find (X) - Returns the position of X .
4. Print list - Contents of the list is displayed.
5. Make empty- Makes the list empty.

Implementation of list ADT:

1. Array based Implementation
2. Linked List based implementation

Array Implementation of list:

Array is a collection of specific number of same type of data stored in consecutive memory locations. Array is a static data structure i.e., the memory should be allocated in advance and the size is fixed. This will waste the memory space when used space is less than the allocated space.

Insertion and Deletion operation are expensive as it requires more data movements. Find and Print list operations take constant time.

20	10	30	40	50	60
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

The basic operations performed on a list of elements are

- Creation of List.
- Insertion of data in the List
- Deletion of data from the List
- Display all data's in the List
- Searching for a data in the list

Declaration of Array:

```
#define maxsize 10
```

```
int list[maxsize], n;
```

Create Operation:

Create operation is used to create the list with „ n „, number of elements. If „ n „, exceeds the array's maxsize, then elements cannot be inserted into the list. Otherwise, the array elements are stored in the consecutive array locations (i.e.) list [0], list [1] and so on.

```
#define maxsize 10
```

```
int main()
```

```
{
```

```
    int list[maxsize],n,i;
```

```
    printf("\nEnter the number of elements to be added in the list:");
```

```
    scanf("%d",&n);
```

```
    printf("\nEnter the array elements:\t");
```

```
    for(i=0;i<n;i++)
```

```
    {
```

```
        scanf("%d",&list[i]);
```

```
    }
```

```
    printf("Array elements in the list:");
```

```
    for(i=0;i<n;i++)
```

```
    {
```

```
        printf("%d ",list[i]);
```

```
    }
```

```

    return 0;
}

```

If n=6, the output of creation is as follows:

list[6]

20	10	30	40	50	60
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

Insert Operation:

Insert operation is used to insert an element at particular position in the existing list. Inserting the element in the last position of an array is easy. But inserting the element at a particular position in an array is quite difficult since it involves all the subsequent elements to be shifted one position to the right.

Insert an element in the array:

```

#define maxsize 10
int main()
{
    int i, data, pos, list[maxsize], n;
    printf("\nEnter the number of elements to be added in the list:");
    scanf("%d", &n);
    printf("\nEnter the array elements:\t");
    for(i=0; i<n; i++)
    {
        scanf("%d", &list[i]);
    }
    printf("\nEnter the data to be inserted:");
    scanf("%d", &data);
    printf("\nEnter the position at which element to be inserted:");
    scanf("%d", &pos);
    if(pos==n)
        printf("Array overflow");
    for(i=n; i>=pos-1; i--)

```

```

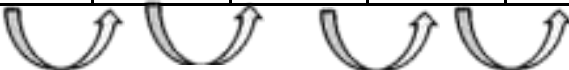
{
    list[i+1]=list[i];
}
list[pos]=data;
n=n+1;
printf("Array elements in the list:");
for(i=0;i<n;i++)
{
    printf("%d ",list[i]);
}
return 0;
}
  
```

Consider an array with 5 elements [max elements = 10]

10	20	30	40	50					
----	----	----	----	----	--	--	--	--	--

If data 15 is to be inserted in the 2nd position then 50 has to be moved to next index position, 40 has to be moved to 50 position, 30 has to be moved to 40 position and 20 has to be moved to 30 position.

10	20	30	40	50					
----	----	----	----	----	--	--	--	--	--



10		20	30	40	50				
----	--	----	----	----	----	--	--	--	--

After this four data movement, 15 is inserted in the 2nd position of the array.

10	15	20	30	40	50				
----	----	----	----	----	----	--	--	--	--

Deletion Operation:

Deletion is the process of removing an element from the array at any position.

Deleting an element from the end is easy. If an element is to be deleted from any particular position, it requires all subsequent element from that position is shifted one position towards left.

Delete an element in the array:

```
#define maxsize 10

int main()
{
    int i, data, pos, list[maxsize], n;
    printf("\nEnter the number of elements to be added in the list:");
    scanf("%d", &n);
    printf("\nEnter the array elements: \t");
    for(i=0; i<n; i++)
    {
        scanf("%d", &list[i]);
    }
    printf("\nEnter the position to be deleted:");
    scanf("%d", &pos);
    printf("\nDeleted data is %d", list[pos-1]);
    for(i=pos-1; i<n-1; i++)
    {
        list[i]=list[i+1];
    }
    n=n-1;
    printf("\nArray elements in the list:");
    for(i=0; i<n; i++)
    {
        printf("%d ", list[i]);
    }
}
```



```

return 0;
}

```

Consider an array with 5 elements [max elements = 10]

10	20	30	40	50					
----	----	----	----	----	--	--	--	--	--

If data 20 is to be deleted from the array, then 30 has to be moved to data 20 position, 40 has to be moved to data 30 position and 50 has to be moved to data 40 position.

1 0	2 0	3 0	4 0	5 0					
--------	--------	--------	--------	--------	--	--	--	--	--

After these 3 data movements, data 20 is deleted from the 2nd position of the array.

10	30	40	50						
----	----	----	----	--	--	--	--	--	--

Display Operation/Traversing a list

Traversal is the process of visiting the elements in an array.

Display() operation is used to display all the elements stored in the list.

Display Operation:

```

#define maxsize 10
int main()
{
    int list[maxsize],n,i;
    printf("\nEnter the number of elements to be added in the list:");
    scanf("%d",&n);
    printf("\nEnter the array elements:\t");
    for(i=0;i<n;i++)
    {
        scanf("%d",&list[i]);
    }
    printf("Array elements in the list:");
    for(i=0;i<n;i++)
    {

```

```
    printf("%d ",list[i]);  
}  
return 0;  
}
```

Search Operation:

Search() operation is used to determine whether a particular element is present in the list or not.
Input the search element to be checked in the list.

To search an element in the array:

```
#define maxsize 10  
int main()  
{  
    int list[maxsize],n,i, search, count=0;  
    printf("\nEnter the number of elements to be added in the list:");  
    scanf("%d",&n);  
    printf("\nEnter the array elements:\t");  
    for(i=0;i<n;i++)  
    {  
        scanf("%d",&list[i]);  
    }  
    printf("\nEnter the elements to be searched:");  
    scanf("%d",&search);  
    for(i=0;i<n;i++)  
    {  
        if(search == list[i])  
            count++;  
    }  
    if(count==0)  
        printf("\nElement not present in the list");  
    else  
        printf("\nElement present in the test");  
    return 0;  
}
```

Program for array implementation of List

```
/* ARRAY BASED IMPLEMENTATION OF LIST */  
  
#include<stdio.h>  
#define maxsize 10
```

```
int list[maxsize],n;
int main()
{
    int choice;
    while(1)
    {
        printf("\n\n***ARRAY IMPLEMENTATION OF LIST***\n\n");
        printf("1.Create\n");
        printf("2.Display\n");
        printf("3.Insert\n");
        printf("4.Delete\n");
        printf("5.Search\n");
        printf("6.Exit\n");
        printf("Enter your choice (1,2,3,4,5,6): ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                create();
                break;
            case 2:
                display();
                break;
            case 3:
                insert();
                break;
            case 4:
                delete();
                break;
            case 5:
                search();
                break;
            case 6:
                exit(0);
                break;
            default:
                printf("Invalid choice");
                break;
        }
    }
    return 0;
}

void create()
{
    int i;
    printf("\nEnter the number of elements to be added in the list:");
    scanf("%d",&n);
```

```
printf("\nEnter the array elements:\t");
for(i=0;i<n;i++)
{
    scanf("%d",&list[i]);
}
}
void display()
{
    int i;
    printf("Array elements in the list:");
    for(i=0;i<n;i++)
    {
        printf("%d ",list[i]);
    }
}
void insert()
{
    int i, data, pos;
    printf("\nEnter the data to be inserted:");
    scanf("%d",&data);
    printf("\nEnter the position at which element to be inserted:");
    scanf("%d",&pos);
    if(pos==n)
        printf("\nArray overflow");
    for(i=n;i>=pos-1;i--)
    {
        list[i+1]=list[i];
    }
    list[pos]=data;
    n=n+1;
    printf("\nAFTER INSERTION");
}
void delete()
{
    int i, data, pos;
    printf("\nEnter the position to be deleted:");
    scanf("%d",&pos);
    printf("\nDeleted data is %d",list[pos-1]);
    for(i=pos-1;i<n-1;i++)
    {
        list[i]=list[i+1];
    }
    n=n-1;
    printf("\nAFTER DELETION");
}
void search()
{
```

```
int i, search, count=0;
printf("\nEnter the elements to be searched:");
scanf("%d",&search);
for(i=0;i<n;i++)
{
    if(search == list[i])
        count++;
}
if(count==0)
    printf("\nElement not present in the list");
else
    printf("\nElement present in the test");
}
```

Output

ARRAY IMPLEMENTATION OF LIST

- 1.Create
- 2.Display
- 3.Insert
- 4.Delete
- 5.Search
- 6.Exit

Enter your choice (1,2,3,4,5,6): 1

Enter the number of elements to be added in the list:5

Enter the array elements: 10

20

30

40

50

ARRAY IMPLEMENTATION OF LIST

- 1.Create
- 2.Display
- 3.Insert
- 4.Delete
- 5.Search
- 6.Exit

Enter your choice (1,2,3,4,5,6): 2

Array elements in the list:10 20 30 40 50

ARRAY IMPLEMENTATION OF LIST

- 1.Create
- 2.Display
- 3.Insert
- 4.Delete
- 5.Search
- 6.Exit

Enter your choice (1,2,3,4,5,6): 3

Enter the data to be inserted:60

Enter the position at which element to be inserted:2

AFTER INSERTION

ARRAY IMPLEMENTATION OF LIST

- 1.Create
- 2.Display
- 3.Insert
- 4.Delete
- 5.Search
- 6.Exit

Enter your choice (1,2,3,4,5,6): 2

Array elements in the list:10 20 60 30 40 50

ARRAY IMPLEMENTATION OF LIST

- 1.Create
- 2.Display
- 3.Insert
- 4.Delete
- 5.Search
- 6.Exit

Enter your choice (1,2,3,4,5,6): 4

Enter the position to be deleted:2

Deleted data is 20

AFTER DELETION

ARRAY IMPLEMENTATION OF LIST

- 1.Create
- 2.Display
- 3.Insert
- 4.Delete
- 5.Search
- 6.Exit

Enter your choice (1,2,3,4,5,6): 2

Array elements in the list:10 60 30 40 50

ARRAY IMPLEMENTATION OF LIST

- 1.Create
- 2.Display
- 3.Insert
- 4.Delete
- 5.Search
- 6.Exit

Enter your choice (1,2,3,4,5,6): 6

Advantages of array implementation:

- 1.The elements are faster to access using random access
- 2.Searching an element is easier

Limitation of array implementation

- An array store its nodes in consecutive memory locations.
- The number of elements in the array is fixed and it is not possible to change the number of elements .
- Insertion and deletion operation in array are expensive. Since insertion is performed by pushing the entire array one position down and deletion is performed by shifting the entire array one position up.

Applications of arrays:

Arrays are particularly used in programs that require storing large collection of similar type data elements.

Differences between Array based and Linked based implementation

	Array	Linked List
Definition	Array is a collection of elements having same data type with common name	Linked list is an ordered collection of elements which are connected by links/pointers
Access	Elements can be accessed using index/subscript, random access	Sequential access
Memory structure	Elements are stored in contiguous memory locations	Elements are stored at available memory space
Insertion & Deletion	Insertion and deletion takes more time in array	Insertion and deletion are fast and easy
Memory Allocation	Memory is allocated at compile time i.e static memory allocation	Memory is allocated at run time i.e dynamic memory allocation
Types	1D,2D, multi-dimensional	SLL, DLL circular linked list
Dependency	Each elements is independent	Each node is dependent on each other as address part contains address of next node in the list

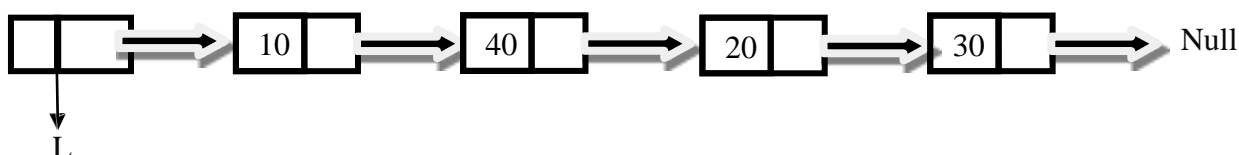
Linked list-based implementation:

Linked Lists:

A Linked list is an ordered collection of elements. Each element in the list is referred as a node. Each node contains two fields namely,
 Data field-The data field contains the actual data of the elements to be stored in the list
 Next field- The next field contains the address of the next node in the list

DATA	NEXT
-------------	-------------

A Linked list node



Advantages of Linked list:

1. Insertion and deletion of elements can be done efficiently
2. It uses dynamic memory allocation
3. Memory utilization is efficient compared to arrays

Disadvantages of linked list:

1. Linked list does not support random access
2. Memory is required to store next field
3. Searching takes time compared to arrays

Types of Linked List

1. Singly Linked List or One Way List
2. Doubly Linked List or Two-Way Linked List
3. Circular Linked List

Dynamic allocation

The process of allocating memory to the variables during execution of the program or at run time is known as dynamic memory allocation. C language has four library routines which allow this function.

Dynamic memory allocation gives best performance in situations in which we do not know memory requirements in advance. C provides four library routines to automatically allocate memory at the run time.

Memory allocation/de-allocation functions

Function	Task
<code>malloc()</code>	Allocates memory and returns a pointer to the first byte of allocated space
<code>calloc()</code>	Allocates space for an array of elements, initializes them to zero and returns a pointer to the memory
<code>free()</code>	Frees previously allocated memory
<code>realloc()</code>	Alters the size of previously allocated memory

To use dynamic memory allocation functions, you must include the header file `stdlib.h`.

malloc()

The `malloc` function reserves a block of memory of specified size and returns a pointer of type `void`. This means that we can assign it to any type of pointer.

The general syntax of `malloc()` is

`ptr=(cast-type*)malloc(byte-size);`

where `ptr` is a pointer of type `cast-type`. `malloc()` returns a pointer (of cast type) to an area of memory with size `byte-size`.

calloc():

`calloc()` function is another function that reserves memory at the run time. It is normally used to request multiple blocks of storage each of the same size and then sets all bytes to zero. `calloc()` stands for contiguous memory allocation and is primarily used to allocate memory for arrays. The syntax of `calloc()` can be given as:

`ptr=(cast-type*) calloc(n,elem-size);`

The above statement allocates contiguous space for `n` blocks each of size `elem-size` bytes.

The only difference between `malloc()` and `calloc()` is that when we use `calloc()`, all bytes are initialized to zero. `calloc()` returns a pointer to the first byte of the allocated region.

free():

The `free()` is used to release the block of memory.

Syntax:

The general syntax of the `free()` function is, `free(ptr);`

where `ptr` is a pointer that has been created by using `malloc()` or `calloc()`. When memory is deallocated using `free()`, it is returned back to the free list within the heap.

realloc():

At times the memory allocated by using `calloc()` or `malloc()` might be insufficient or in excess. In both the situations we can always use `realloc()` to change the memory size already allocated by `calloc()` and `malloc()`. This process is called *reallocation of memory*.

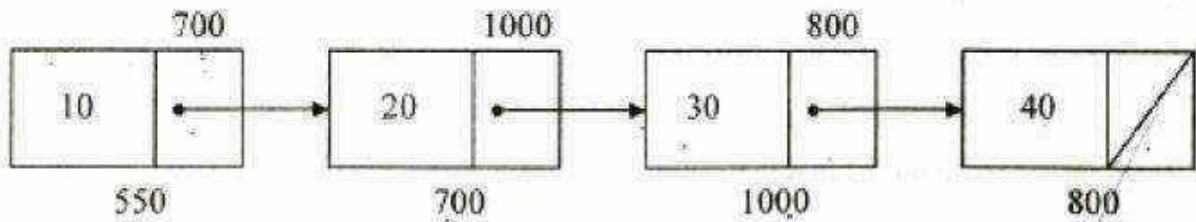
The general syntax for `realloc()` can be given as,

`ptr = realloc(ptr,newsize);`

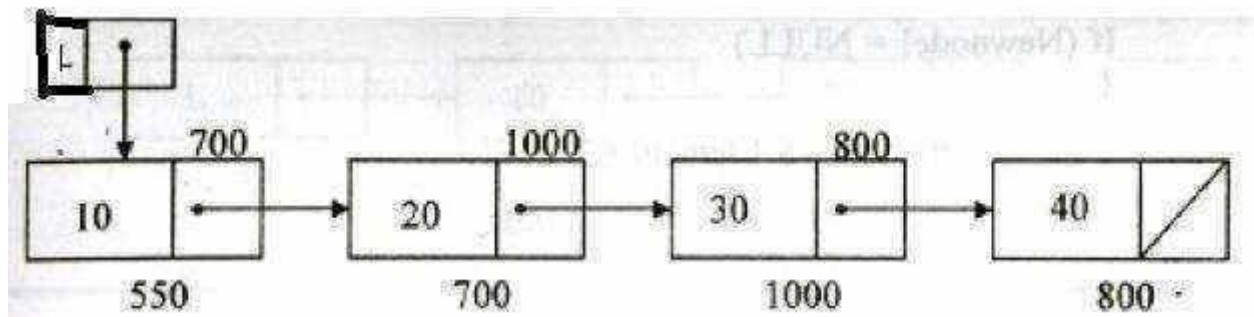
new memory space of size specified by newsize to the pointer variable ptr. It returns a pointer to the first byte of the memory block. The allocated new block may be or may not be at the same region. Thus, we see that realloc() takes two arguments. The first is the pointer referencing the memory and the second is the total number of bytes you want to reallocate.

Singly Linked List

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



SLL



SLL with a Header

Basic operations on a singly-linked list are:

1. Insert – Inserts a new node in the list.
2. Delete – Deletes any node from the list.
3. Find – Finds the position(address) of any node in the list.
4. FindPrevious - Finds the position(address) of the previous node in the list.
5. FindNext- Finds the position(address) of the next node in the list.
6. Display-display the data in the list
7. Search-find whether a element is present in the list or not

Declaration of Linked List

```
struct node
{
    int data;
    struct node *
    next;
};
```

Creation of the list:

This routine creates a list by getting the number of nodes from the user. Assume n=4
for this example.

```
void create()
{
    int n,i, count=0;
    printf("Enter the size: ");
    scanf("%d",&n);
    for(i=0; i<n; i++)
    {
        newnode=(struct node *)malloc(sizeof(struct node));
        printf("Enter data:");
        scanf("%d",&newnode->data);
        newnode->next=NULL;
        if(head==NULL)
        {
            head=newnode;
            temp=newnode;
        }
        else
        {
            temp->next=newnode;
            temp=newnode;
        }
    }
}
```

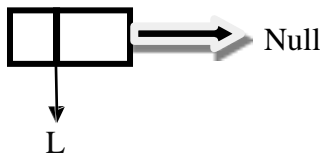
```

}
temp=head;
while(temp!=NULL)
{
    printf("%d ",temp->data);
    temp=temp->next;
    count++;
}

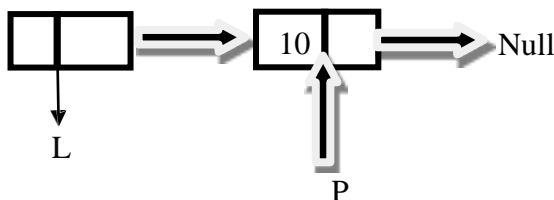
printf("\nTotal number of elements in linked list is %d",count);
}
  
```

Initially the list is empty

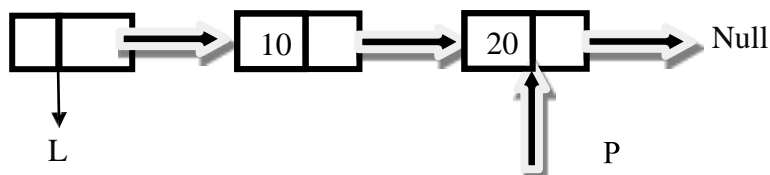
List L



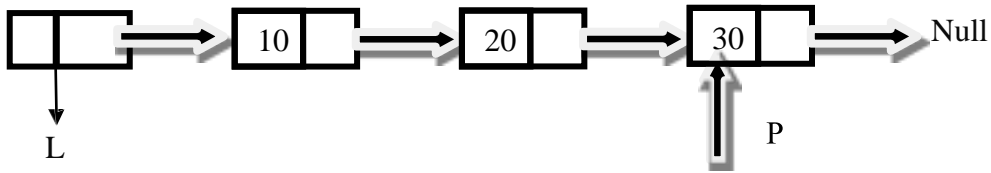
Insert(10,List L)- A new node with data 10 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.



Insert(20,L) - A new node with data 20 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.



Insert(30,L) - A new node with data 30 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.



Case 1: Routine to insert an element in list at the beginning

void insert()

```

{
    int n,i, count=0;
    printf("Enter the size: ");
    scanf("%d",&n);
    for(i=0; i<n; i++)
    {
        newnode=(struct node *)malloc(sizeof(struct node));
        printf("Enter data:");
        scanf("%d",&newnode->data);
        newnode->next=NULL;
        if(head==NULL)
        {
            head=newnode;
            temp=newnode;
        }
        else
        {
            temp->next=newnode;
            temp=newnode;
        }
    }
    temp=head;
    while(temp!=NULL)
    {
        printf("%d ",temp->data);
        temp=temp->next;
    }
    printf("\nINSERTION AT THE BEGINNING");
    newnode=(struct node *)malloc(sizeof(struct node));
    printf("\nEnter data that you want to insert:");
    scanf("%d",&newnode->data);
    newnode->next=head;

```

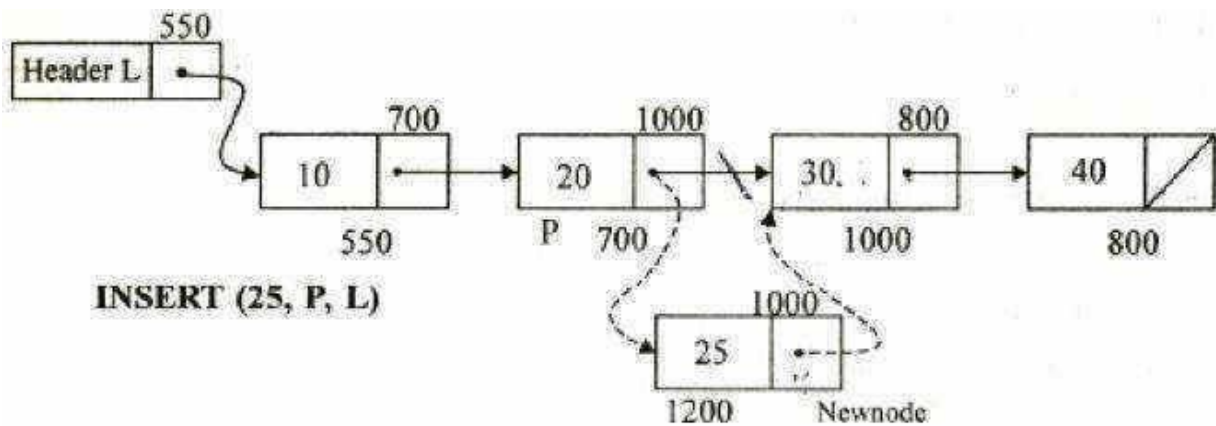
```
    head=newnode;  
}
```

Case 2:Routine to insert an element in list at Position

This routine inserts an element X after the position P.

```
Void Insert( )  
{  
    int pos;  
    newnode=(struct node *)malloc(sizeof(struct node));  
    printf("\nEnter the position:");  
    scanf("%d",&pos);  
    if(pos>count)  
    {  
        printf("\nINVALID POSITION");  
    }  
    else  
    {  
        temp=head;  
        int i=1, value;  
        while(i<pos)  
        {  
            temp=temp->next;  
            i++;  
        }  
        printf("\nEnter the data that you want to insert: ");  
        scanf("%d",&newnode->data);  
        newnode->next=temp->next;  
        temp->next=newnode;  
    }  
}
```

A new node with data 25 is inserted after the position P and the next field is updated to NULL.
The next field of previous node is updated to store the address of new node.



Case 3: Routine to insert an element in list at the end of the list

```

void insert( )
{
    newnode=(struct node *)malloc(sizeof(struct node));
    printf("\nEnter data that you want to insert:");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    temp=head;
    while(temp->next!=NULL)
    {
        temp=temp->next;
    }
    temp->next = newnode;
}
    
```

Routine to check whether a list is Empty

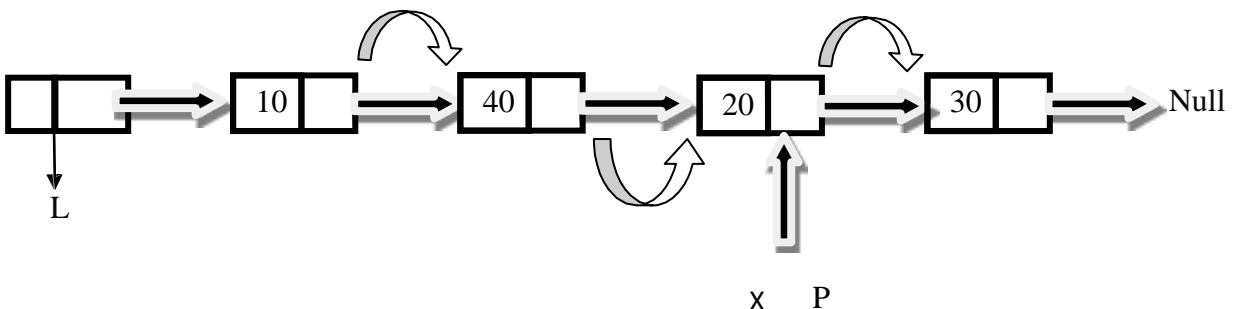
These routine checks whether the list is empty. If the list is empty, it returns 1


```
if(head==NULL)
{
    printf("\nList is empty");
}
```

Routine to Find the Element in the List

```
int search, count=0;
printf("Enter an element to search in the list:");
scanf("%d",&search);
temp=head;
while(temp!=NULL)
{
    if(temp->data == search)
    {
        count++;
    }
    temp=temp->next;
}
if(count==1)
{
    printf("\nElement is there");
}
else
{
    printf("\nElement is not there");
}
```

Find(List L, 20) - To find an element X traverse from the first node of the list and move to the next with the help of the address stored in the next field until data is equal to X or till the end of the list



Routine to Count the Element in the List:

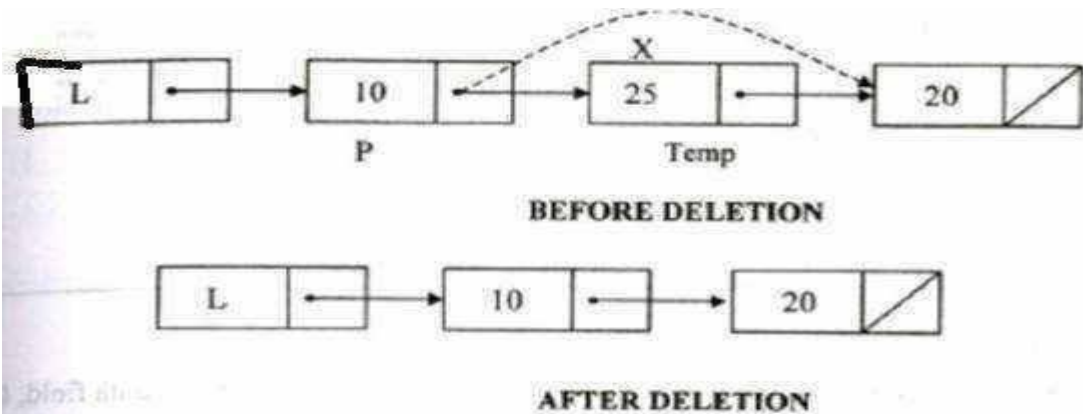
This routine counts the number of elements in the list

```
void count ( )

int count=0;
temp=head;
while(temp!=NULL)
{
    printf("%d ",temp->data);
    temp=temp->next;
    count++;
}
printf("\nTotal number of elements in linked list is %d", count);
```

Routine to Delete an Element in the beginning of the List:

```
void delete( )
{
    printf("\nDELETION AT THE BEGINNING");
    if(head==NULL)
    {
        printf("\nList is empty");
    }
    else
    {
        temp=head;
        head=head->next;
        free(temp);
    }
}
```



Routine to Delete the element at the end of the list

```
void delete( )
{
    printf("\nDELETION AT THE END");
    temp=head;
    while(temp->next != NULL)
    {
        prev=temp;
        temp=temp->next;
    }
    if(temp==head)
    {
        head=NULL;
    }
    else
    {
        prev->next=NULL;
    }
    free(temp);
}
```

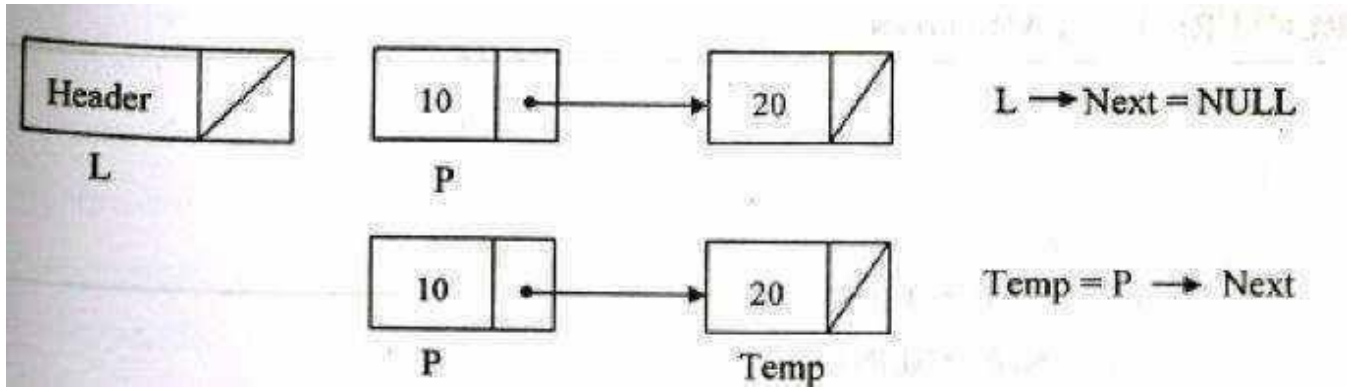
Routine to Delete the element in the middle of the list

```
void delete( )
{
    printf("\nDELETION AT THE MIDDLE");
    int pos, j=1;
    printf("\nEnter the position:");
    scanf("%d",&pos);
    temp=head;
    while(j<pos-1)
    {
        temp=temp->next;
        j++;
    }
```

```

    }
    nextnode=temp->next;
    temp->next=nextnode->next;
    free(nextnode);
}

```



Implementation of Singly linked List

```

/** IMPLEMENTATION OF SINGLY LINKED LIST**/
#include<stdio.h>
#include<stdlib.h>
void create();
void display();
void insertfront();
void insertend();
void insertmid();
void deletefront();
void deleteend();
void deletemid();
void count();
void search();
struct node
{
    int data;
    struct node *next;
}*head, *temp, *newnode, *nextnode, *prev;
int main()
{
    int option;
    while(1)
    {
        printf("\n\n***SINGLY LINKED LIST***\n\n");
        printf("1.Create\n");
        printf("2.Display\n");
    }
}

```

```
printf("3.Insert at front\n");
printf("4.Insert at end\n");
printf("5.Insert at middle\n");
printf("6.Deletion at front\n");
printf("7.Deletion at end\n");
printf("8.Deletion at middle\n");
printf("9.Count the elements\n");
printf("10.Search the element\n");
printf("11.Exit\n");
printf("Enter your choice (1,2,3,4,5,6,7,8,9): ");
scanf("%d",&option);
switch(option)
{
    case 1: create();
            break;
    case 2: display();
            break;
    case 3: insertfront();
            break;
    case 4: insertend();
            break;
    case 5: insertmid();
            break;
    case 6: deletefront();
            break;
    case 7: deleteend();
            break;
    case 8: deletemid();
            break;
    case 9: count();
            break;
    case 10: search();
            break;
    case 11: exit(0);
    default: printf("Invalid choice");
            break;
}
}
return 0;
}
void create()
{
    int size,i;
    printf("\nEnter the size of the list:");
    scanf("%d",&size);
    for(i=1;i<=size;i++)
    {
        newnode=(struct node*)malloc(sizeof(struct node));
```

```
scanf("%d",&newnode->data);
newnode->next=NULL;
if(head==NULL)
{
    head=temp=newnode;
}
else
{
    temp->next=newnode;
    temp=newnode;
}
}
}
void display()
{
    printf("\nDisplayed the elements in the list:");
    temp=head;
    while(temp!=NULL)
    {
        printf("%d ",temp->data);
        temp=temp->next;
    }
}
void insertfront()
{
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter the data that you want to insert:");
    scanf("%d",&newnode->data);
    newnode->next=head;
    head=newnode;
}
void insertend()
{
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter the data that you want to insert:");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    temp=head;
    while(temp->next!=NULL)
    {
        temp=temp->next;
    }
    temp->next = newnode;
}
void insertmid()
{
    int pos,j;
    newnode=(struct node*)malloc(sizeof(struct node));
```

```
printf("\nEnter the data that you want to insert:");
scanf("%d",&newnode->data);
printf("\nEnter the position:");
scanf("%d",&pos);
temp=head;
for(j=1;j<pos;j++)
{
    temp=temp->next;
    newnode->next=temp->next;
    temp->next=newnode;
}
}
void deletefront()
{
    int pos, j=1;
    printf("\nEnter the position:");
    scanf("%d",&pos);
    temp=head;
    while(j<pos-1)
    {

        temp=temp->next;
        j++;
    }
    nextnode=temp->next;
    temp->next=nextnode->next;
    free(nextnode);
}
void deleteend()
{
    temp=head;
    while(temp->next != NULL)
    {
        prev=temp;
        temp=temp->next;
    }
    if(temp==head)
    {
        head=NULL;
    }
    else
    {
        prev->next=NULL;
    }
    free(temp);
}
void deletemid()
```

```
{
    int position, k=1;
    printf("\nEnter the position:");
    scanf("%d",&position);
    temp=head;
    while(k<position-1)
    {

        temp=temp->next;
        k++;
    }
    nextnode=temp->next;
    temp->next=nextnode->next;
    free(nextnode);
}
void count()
{
    int count=0;
    temp=head;
    while(temp!=NULL)
    {
        count++;
        temp=temp->next;
    }
    printf("\nTotal number of elements in the list: %d", count);
}
void search()
{
    int find, index=0;
    printf("Enter an element to search in the list:");
    scanf("%d",&find);
    temp=head;
    while(temp!=NULL)
    {
        if(temp->data == find)
        {
            index++;
        }
        temp=temp->next;
    }
    if(index==1)
    {
        printf("\nElement is there");
    }
    else
    {
        printf("\nElement is not there");
    }
}
```


}
}

Advantages of SLL

1. The elements can be accessed using the next link
2. Occupies less memory than DLL as it has only one next field.

Disadvantages of SLL

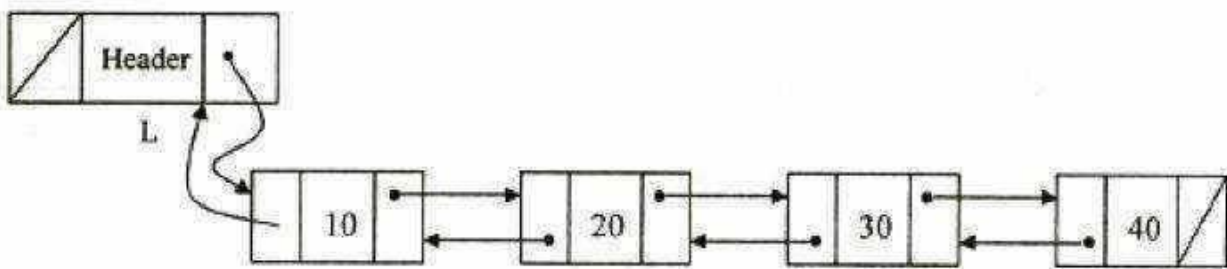
1. Traversal in the backwards is not possible
2. Less efficient to for insertion and deletion.

Doubly-Linked List

A doubly linked list is a linked list in which each node has three fields namely Data, Next, Prev. Data-This field stores the value of the element
Next-This field points to the successor node in the list
Prev-This field points to the predecessor node in the list

PREV	DATA	NEXT

DLL NODE



DOUBLY LINKED LIST

Basic operations of a doubly -linked list are:

1. Insert – Inserts a new element at the end of the list.

2. Delete – Deletes any node from the list.
3. Find – Finds any node in the list.
4. Print – Prints the list

Declaration of DLL Node

struct

node

{

int data;

s

t

r

u

c

t

n

o

d

e

*

n

e

x

t

;

s

t

r

u

c

t

n

PREV	DATA	NEXT

```
o  
d  
e  
*  
p  
r  
e  
v  
;  
};
```

Creation of list in DLL:

```
Void create( )  
{  
    int n,i;  
    printf("Enter the size: ");  
    scanf("%d",&n);  
    for(i=0;i<n;i++)  
    {  
        newnode=(struct  
node*)malloc(sizeof(struct node));  
        printf("Enter the data:");  
        scanf("%d",&newnode->data);  
        newnode->next=NULL;  
        newnode->prev=NULL;  
        if(head==NULL)  
        {  
            head=tail=newnode;  
        }  
        else  
        {  
            tail->next=newnode;  
            newnode->prev=tail;  
            tail=newnode;  
        }  
    }  
    temp=head;  
    while(temp!=NULL)
```

```
    {  
        printf("%d ",temp->data);  
        temp=temp->next;  
    }  
    return 0;  
}
```

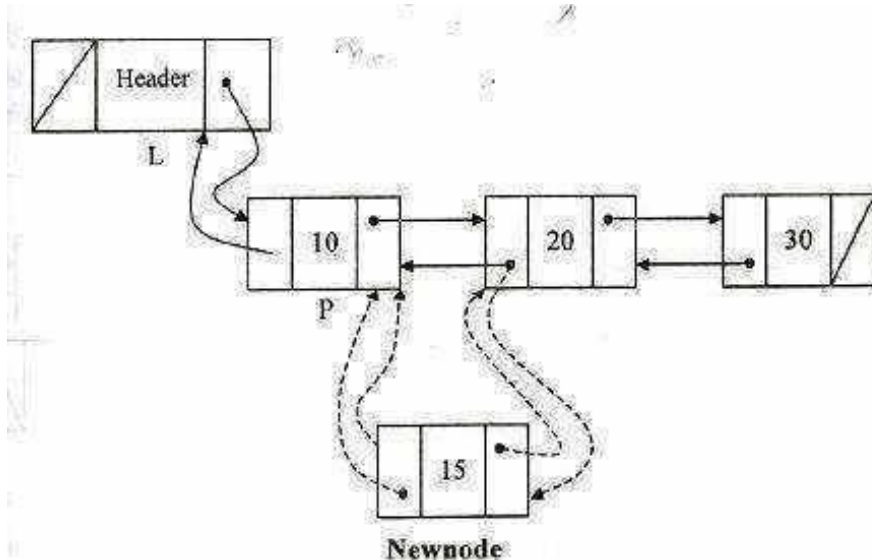
Routine to insert an element in a DLL at the beginning

```
void Insert ( )  
{  
    newnode=(struct node *)malloc(sizeof(struct node));  
    printf("\nEnter data that you want to insert:");  
    scanf("%d",&newnode->data);  
    newnode->next=NULL;  
    newnode->prev=NULL;  
    head->prev=newnode;  
    newnode->next=head;  
    head=newnode;  
}
```

Routine to insert an element in a DLL any position:

```
Void insertmid( )  
{  
    int i=1, pos;  
    temp=head;  
    newnode=(struct node *)malloc(sizeof(struct node));  
    printf("\nEnter the position:");  
    scanf("%d",&pos);  
    printf("\nEnter data that you want to insert:");  
    scanf("%d",&newnode->data);  
    newnode->next=NULL;  
    newnode->prev=NULL;  
    while(i<pos-1)  
    {  
        temp=temp->next;  
        i++;  
    }  
    newnode->prev=temp;
```

```
newnode->next=temp->next;  
temp->next=newnode;  
newnode->next->prev=newnode;  
}
```



Routine to insert an element in a DLL at the end:

```
Void insertend( )
```

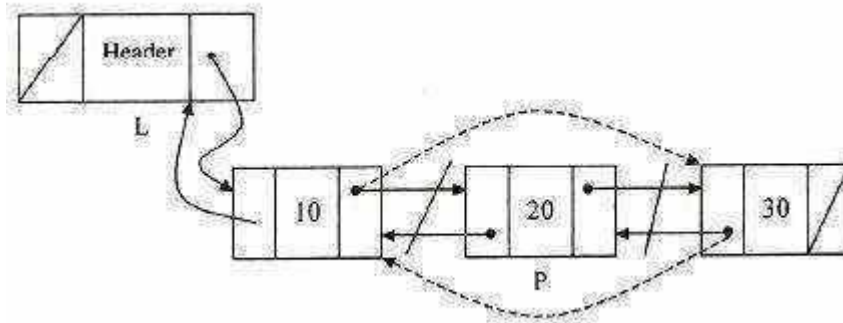
```
{  
    newnode=(struct node *)malloc(sizeof(struct node));  
    printf("\nEnter data that you want to insert:");  
    scanf("%d",&newnode->data);  
    newnode->next=NULL;  
    newnode->prev=NULL;  
    tail->next=newnode;  
    newnode->prev=tail;  
    tail=newnode;  
}
```

Routine for deleting an element at the beginning:

```
void Delete ( )
```

```
{  
    temp=head;
```

```
head=temp->next;  
temp->next=NULL;  
head->prev=NULL;  
free(temp);  
}
```



Routine for deleting an element at the end:

```
void Delete ( )  
{  
    tail->prev->next=NULL;  
    free(tail);  
}
```

Routine for deleting an element at the middle:

```
void Delete ( )  
{  
    int j=1, pos;  
    temp=head;  
    printf("\nEnter the position:");  
    scanf("%d",&pos);  
    while(j<pos-1)  
    {  
        temp=temp->next;  
        j++;  
    }  
    nextnode=temp->next;  
    temp->next=nextnode->next;  
    free(nextnode);  
}
```

Routine to display the elements in the list:

```
void Display()
{
    temp=head;
    while(temp!=NULL)
    {
        printf("%d ",temp->data);
        temp=temp->next;
    }
}
```

Routine to search whether an element is present in the list

```
void find()
{
    int search, count=0;
    printf("Enter an element to search in the list:");
    scanf("%d",&search);
    temp=head;
    while(temp!=NULL)
    {
        if(temp->data == search)
        {
            count++;
        }
        temp=temp->next;
    }
    if(count==1)
    {
        printf("\nElement is there");
    }
    else
    {
        printf("\nElement is not there");
    }
}
```

Implementation of Doubly linked list

```
// DOUBLY LINKED LIST
```

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

```
{
    int data;
    struct node*next;
    struct node*prev;
}*head,*tail,*temp,*newnode,*nextnode;
int main()
{
    int option;
    while(1)
    {
        printf("\n\n***DOUBLY LINKED LIST***\n\n");

        printf("1.Creation\n");
        printf("2.Display\n");
        printf("3.Count\n");
        printf("4.Insert at front\n");
        printf("5.Insert at end\n");
        printf("6.Insert at middle\n");
        printf("7.Delete at front\n");
        printf("8.Delete at end\n");
        printf("9.Delete at middle\n");
        printf("10.Search an element\n");
        printf("11.Exit\n");
        printf("Enter your option(1,2,3,4,5,6,7,8,9,10,11): ");
        scanf("%d",&option);
        switch(option)
        {
            case 1: create();
                    break;
            case 2: display();
                    break;
            case 3: count();
                    break;
            case 4: insertfront();
                    break;
            case 5: insertend();
                    break;
            case 6: insertmid();
                    break;
            case 7: deletefront();
                    break;
            case 8: deleteend();
                    break;
            case 9: deletemid();
                    break;
            case 10: search();
                    break;
```



```
        case 11: exit(0);
        default: printf("Invalid option\n");
                break;
    }
}
return 0;
}
void create()
{
    int n,i;
    printf("Enter the size: ");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        newnode=(struct node*)malloc(sizeof(struct node));
        printf("Enter the elements: ");
        scanf("%d",&newnode->data);
        newnode->prev=NULL;
        newnode->next=NULL;
        if(head==NULL)
        {
            head=tail=newnode;
        }
        else
        {
            tail->next=newnode;
            newnode->prev=tail;
            tail=newnode;
        }
    }
}
void display()
{
    temp=head;
    printf("Displayed the elements:\n");
    while(temp!=NULL)
    {
        printf("%d\n",temp->data);
        temp=temp->next;
    }
}
void count()
{
    int count=0;
    temp=head;
    printf("Displayed the elements:\n");
    while(temp!=NULL)
    {
        printf("%d\n",temp->data);
```

```
        temp=temp->next;
        count++;
    }
    printf("Total no.of elements in the list: %d",count);
}
void insertfront()
{
    newnode=(struct node *)malloc(sizeof(struct node));
    printf("\nEnter data that you want to insert:");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    newnode->prev=NULL;

    head->prev=newnode;
    newnode->next=head;
    head=newnode;
}
void insertend()
{
    newnode=(struct node *)malloc(sizeof(struct node));
    printf("\nEnter data that you want to insert:");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    newnode->prev=NULL;

    tail->next=newnode;
    newnode->prev=tail;
    tail=newnode;
}
void insertmid()
{
    int i=1, pos;
    temp=head;
    newnode=(struct node *)malloc(sizeof(struct node));
    printf("\nEnter the position:");
    scanf("%d",&pos);
    printf("\nEnter data that you want to insert:");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    newnode->prev=NULL;
    while(i<pos-1)
    {
        temp=temp->next;
        i++;
    }
    newnode->prev=temp
    newnode->next=temp->next;
    temp->next=newnode;
    newnode->next->prev=newnode;
```

```
}  
void deletefront()  
{  
    temp=head;  
    head=temp->next;  
    temp->next=NULL;  
    head->prev=NULL;  
    free(temp);  
}  
void deleteend()  
{  
    tail->prev->next=NULL;  
    free(tail);  
}  
void deletemid()  
{  
    int j=1,pos;  
    temp=head;  
    printf("\nEnter the position:");  
    scanf("%d",&pos);  
    while(j<pos-1)  
    {  
        temp=temp->next;  
        j++;  
    }  
    nextnode=temp->next;  
    temp->next=nextnode->next;  
    free(nextnode);  
}  
void search()  
{  
    int search, index=0;  
    printf("Enter an element to search in the list:");  
    scanf("%d",&search);  
    temp=head;  
    while(temp!=NULL)  
    {  
        if(temp->data == search)  
        {  
            index++;  
        }  
        temp=temp->next;  
    }  
    if(index==1)  
    {  
        printf("\nElement is there");  
    }  
    else  
    {  

```

```
        printf("\nElement is not there");  
    }  
}
```

OUTPUT:

DOUBLY LINKED LIST

```
1.Creation  
2.Display  
3.Count  
4.Insert at front  
5.Insert at end  
6.Insert at middle  
7.Delete at front  
8.Delete at end  
9.Delete at middle  
10.Search an element  
11.Exit  
Enter your option(1,2,3,4,5,6,7,8,9,10,11): 1  
Enter the size: 3  
Enter the elements: 10  
Enter the elements: 20  
Enter the elements: 30
```

DOUBLY LINKED LIST

```
1.Creation  
2.Display  
3.Count  
4.Insert at front  
5.Insert at end  
6.Insert at middle  
7.Delete at front  
8.Delete at end  
9.Delete at middle  
10.Search an element  
11.Exit  
Enter your option(1,2,3,4,5,6,7,8,9,10,11): 2  
Displayed the elements:  
10
```

20

30

DOUBLY LINKED LIST

1.Creation

2.Display

3.Count

4.Insert at front

5.Insert at end

6.Insert at middle

7.Delete at front

8.Delete at end

9.Delete at middle

10.Search an element

11.Exit

Enter your option(1,2,3,4,5,6,7,8,9,10,11): 3

Displayed the elements:

10

20

30

Total no.of elements in the list: 3

DOUBLY LINKED LIST

1.Creation

2.Display

3.Count

4.Insert at front

5.Insert at end

6.Insert at middle

7.Delete at front

8.Delete at end

9.Delete at middle

10.Search an element

11.Exit

Enter your option(1,2,3,4,5,6,7,8,9,10,11): 4

Enter data that you want to insert:40

DOUBLY LINKED LIST

1.Creation
2.Display
3.Count
4.Insert at front
5.Insert at end
6.Insert at middle
7.Delete at front
8.Delete at end
9.Delete at middle
10.Search an element
11.Exit
Enter your option(1,2,3,4,5,6,7,8,9,10,11): 2
Displayed the elements:
40
10
20
30

DOUBLY LINKED LIST

1.Creation
2.Display
3.Count
4.Insert at front
5.Insert at end
6.Insert at middle
7.Delete at front
8.Delete at end
9.Delete at middle
10.Search an element
11.Exit
Enter your option(1,2,3,4,5,6,7,8,9,10,11): 5

Enter data that you want to insert:50

DOUBLY LINKED LIST

- 1.Creation
- 2.Display
- 3.Count
- 4.Insert at front
- 5.Insert at end
- 6.Insert at middle
- 7.Delete at front
- 8.Delete at end
- 9.Delete at middle
- 10.Search an element
- 11.Exit

Enter your option(1,2,3,4,5,6,7,8,9,10,11): 2

Displayed the elements:

40

10

20

30

50

DOUBLY LINKED LIST

- 1.Creation
- 2.Display
- 3.Count
- 4.Insert at front
- 5.Insert at end
- 6.Insert at middle
- 7.Delete at front
- 8.Delete at end
- 9.Delete at middle
- 10.Search an element
- 11.Exit

Enter your option(1,2,3,4,5,6,7,8,9,10,11): 6

Enter the position:3

Enter data that you want to insert:70

DOUBLY LINKED LIST

- 1.Creation
- 2.Display
- 3.Count
- 4.Insert at front
- 5.Insert at end
- 6.Insert at middle
- 7.Delete at front
- 8.Delete at end
- 9.Delete at middle
- 10.Search an element
- 11.Exit

Enter your option(1,2,3,4,5,6,7,8,9,10,11): 2

Displayed the elements:

40
10
70
20
30
50

DOUBLY LINKED LIST

- 1.Creation
- 2.Display
- 3.Count
- 4.Insert at front
- 5.Insert at end
- 6.Insert at middle
- 7.Delete at front
- 8.Delete at end
- 9.Delete at middle
- 10.Search an element
- 11.Exit

Enter your option(1,2,3,4,5,6,7,8,9,10,11): 7

DOUBLY LINKED LIST

1.Creation
2.Display
3.Count
4.Insert at front
5.Insert at end
6.Insert at middle
7.Delete at front
8.Delete at end
9.Delete at middle
10.Search an element
11.Exit
Enter your option(1,2,3,4,5,6,7,8,9,10,11): 2
Displayed the elements:
10
70
20
30
50

DOUBLY LINKED LIST

1.Creation
2.Display
3.Count
4.Insert at front
5.Insert at end
6.Insert at middle
7.Delete at front
8.Delete at end
9.Delete at middle
10.Search an element
11.Exit
Enter your option(1,2,3,4,5,6,7,8,9,10,11): 8

DOUBLY LINKED LIST

1.Creation
2.Display
3.Count
4.Insert at front
5.Insert at end
6.Insert at middle
7.Delete at front
8.Delete at end
9.Delete at middle
10.Search an element
11.Exit
Enter your option(1,2,3,4,5,6,7,8,9,10,11): 2
Displayed the elements:
10
70
20
30

DOUBLY LINKED LIST

1.Creation
2.Display
3.Count
4.Insert at front
5.Insert at end
6.Insert at middle
7.Delete at front
8.Delete at end
9.Delete at middle
10.Search an element
11.Exit
Enter your option(1,2,3,4,5,6,7,8,9,10,11): 9

Enter the position:2

DOUBLY LINKED LIST

```
1.Creation
2.Display
3.Count
4.Insert at front
5.Insert at end
6.Insert at middle
7.Delete at front
8.Delete at end
9.Delete at middle
10.Search an element
11.Exit
Enter your option(1,2,3,4,5,6,7,8,9,10,11): 2
Displayed the elements:
10
20
30
```

DOUBLY LINKED LIST

```
1.Creation
2.Display
3.Count
4.Insert at front
5.Insert at end
6.Insert at middle
7.Delete at front
8.Delete at end
9.Delete at middle
10.Search an element
11.Exit
Enter your option(1,2,3,4,5,6,7,8,9,10,11):11
```

Advantages of DLL:

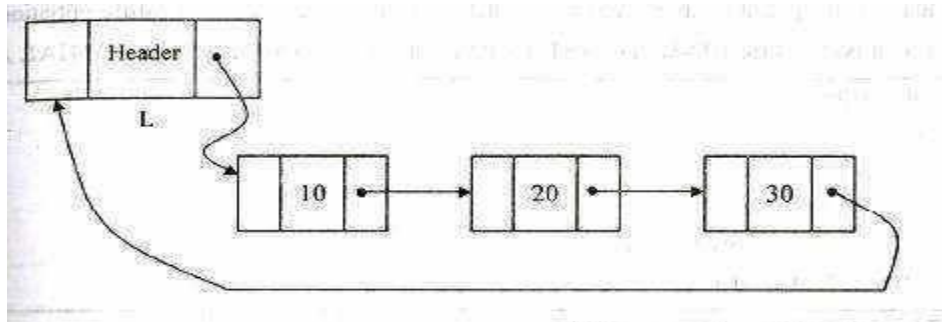
The DLL has two pointer fields. One field is prev link field and another is next link field. Because of these two pointer fields we can access any node efficiently whereas in SLL only onelink field is there which stores next node which makes accessing of any node difficult.

Disadvantages of DLL:

The DLL has two pointer fields. One field is prev link field and another is next link field. Because of these two pointer fields, more memory space is used by DLL compared to SLL

CIRCULAR LINKED LIST:

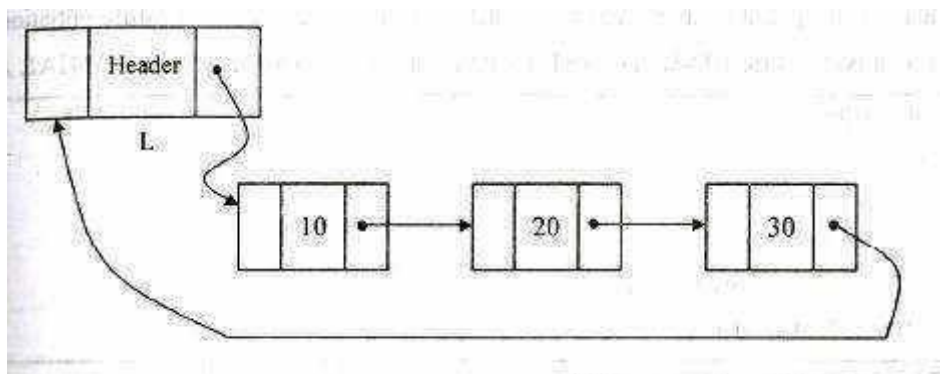
Circular Linked list is a linked list in which the pointer of the last node points to the first node.

**Types of CLL:**

CLL can be implemented as circular singly linked list and circular doubly linked list.

Singly linked circular list:

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

**Declaration of node:**

```
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};
```

Creation of node:

```
void create( )  
{  
    int n,i;  
    printf("Enter the size: ");  
    scanf("%d",&n);  
    for(i=0;i<n;i++)  
    {  
        newnode=(struct node*)malloc(sizeof(struct node));  
        printf("Enter the data:");  
        scanf("%d",&newnode->data);  
        newnode->next=NULL;  
        if(head==NULL)  
        {  
            head=temp=newnode;  
        }  
        else  
        {  
            temp->next=newnode;  
            temp=newnode;  
            newnode->next=head;  
        }  
    }  
}
```

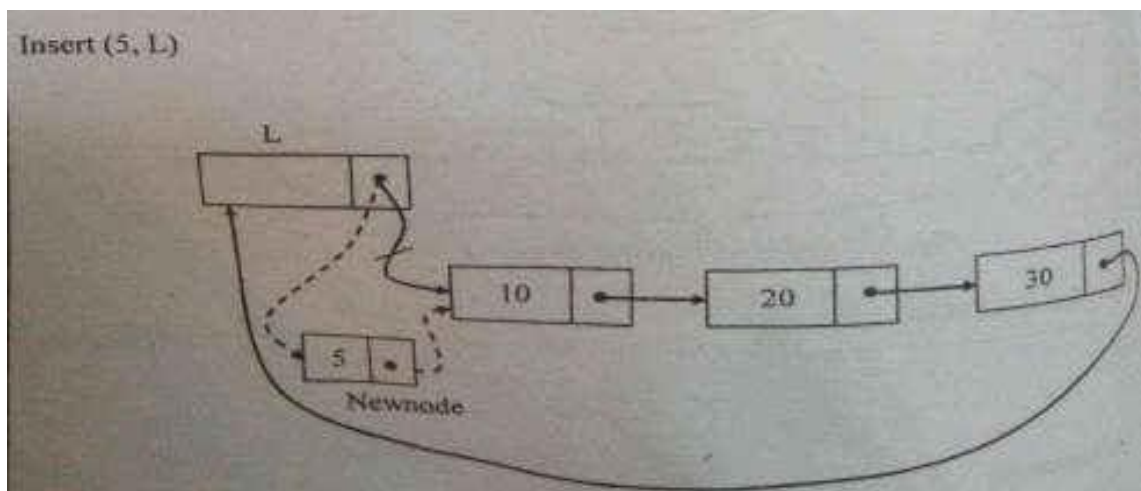
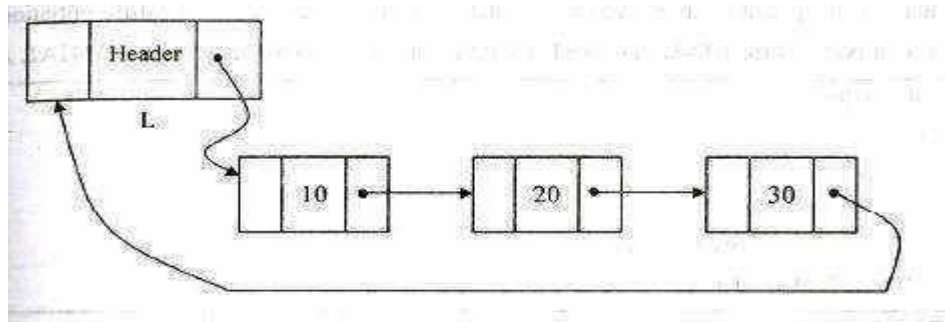
Display the node:

```
temp=head;  
printf("Displayed the elements:\n");
```

```
while(temp->next!=head)
{
    printf("%d\n",temp->data);
    temp=temp->next;
}
printf("%d\t",temp->data);
```

Routine to insert an element in the beginning

```
void insert_beg( )
{
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter the data that you want to insert:");
    scanf("%d", &newnode->data);
    newnode->next=head;
    tail->next=newnode;
    head=newnode;
}
```



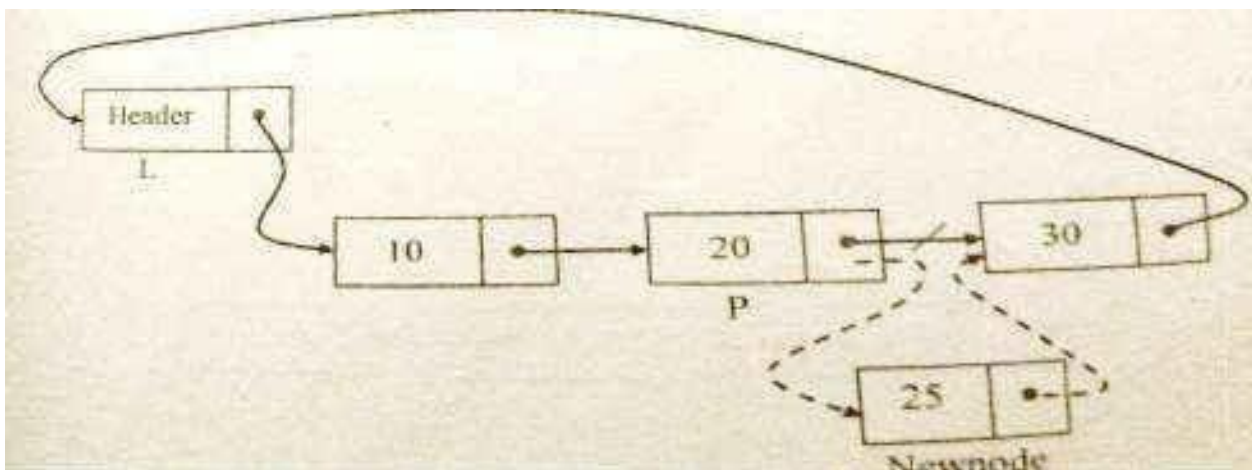
Routine to insert an element in the end

```
void insert_end( )
{
    newnode=(struct node*)malloc(sizeof(struct node));
```

```
printf("\nInsert the end element: ");
int value;
scanf("%d",&value);
newnode->data=value;
tail->next=newnode;
newnode->next=head;
tail=newnode;
}
```

Routine to insert an element in the middle

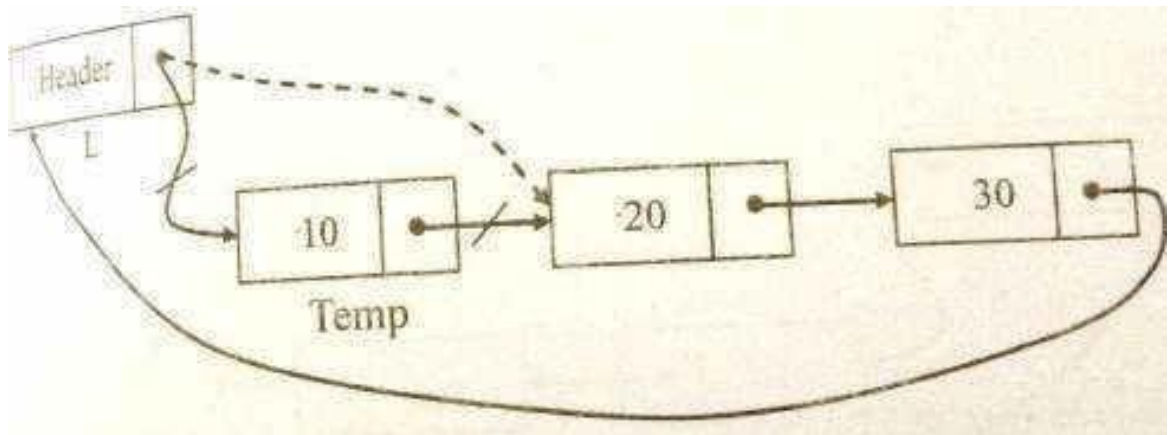
```
void insert_end( )
{
    int pos,j;
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nInsert the middle element: ");
    scanf("%d",&newnode->data);
    printf("Position= ");
    scanf("%d",&pos);
    temp=head;
    for(j=0; j<pos-1; j++)
    {
        temp=temp->next;
        newnode->next=temp->next;
        temp->next=newnode;
    }
```



Routine to delete an element from the beginning

```
void del_first()
```

```
{  
    temp=head;  
    head=head->next;  
    temp=tail;  
    tail->next=head;  
}
```

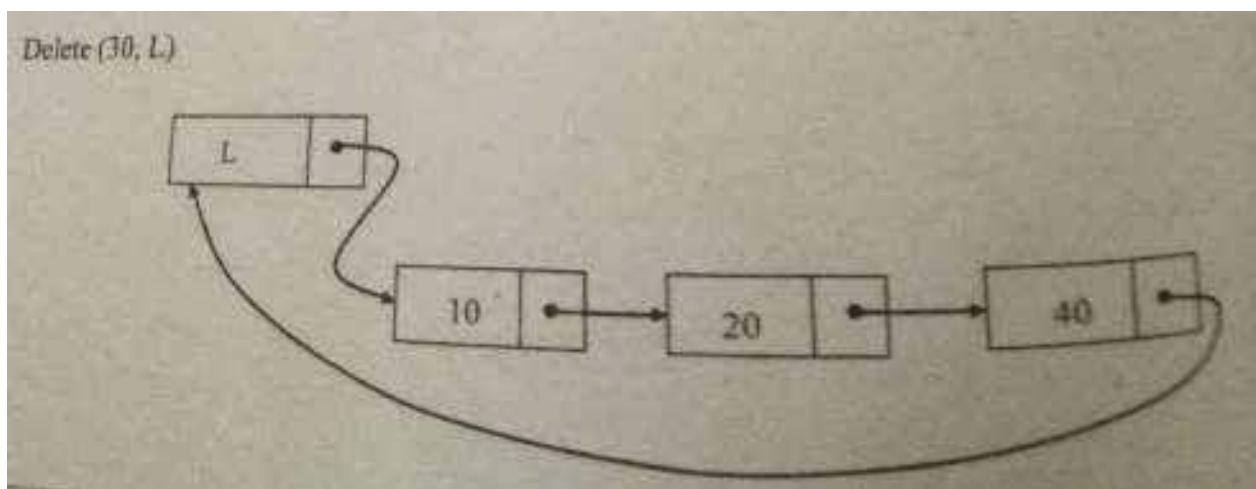
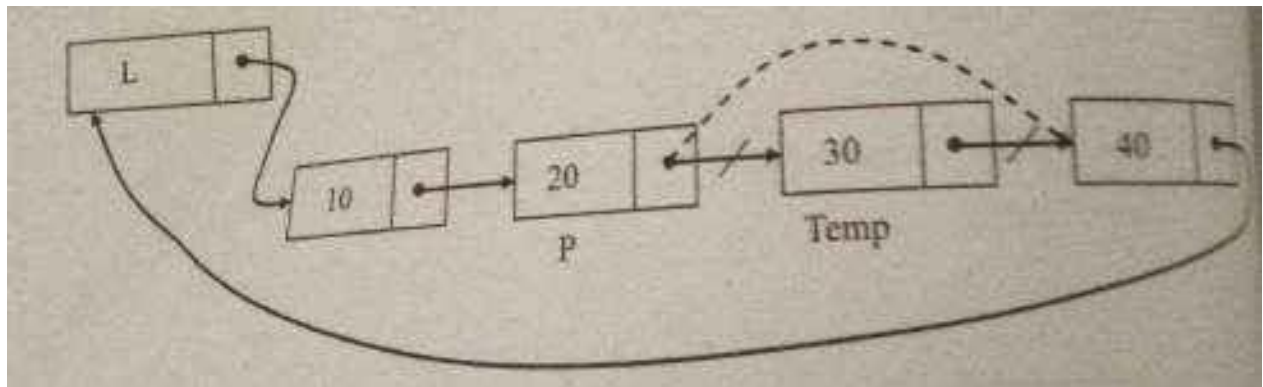


Routine to delete an element from the end

```
void del_end( )  
{  
    temp=head;  
    while(temp->next!=tail)  
    {  
        temp=temp->next;  
    }  
    tail=temp;  
    tail->next=head;  
}
```

Routine to delete an element from the middle

```
void del_mid( )  
{  
    int pos,j;  
    printf("Position= ");  
    scanf("%d",&pos);  
    temp=head;  
    for(j=0; j<pos-1; j++)  
    {  
        temp=temp->next;  
        temp->next=temp->next->next;  
    }  
}
```

Advantages of Circular linked List

- It allows to traverse the list starting at any point. It allows quick access
- to the first and last records.
- Circularly doubly linked list allows to traverse the list in either direction.

Applications of List:

1. Polynomial ADT
2. Radix sort
3. Multilists