

Abstract and Concrete Data Structures- Basic data structures – vectors and arrays. Applications, Linked lists:- singly linked list, doubly linked list, Circular linked list, operations on linked list, linked list with header nodes, applications of linked list: polynomials,.

2.1. ABSTRACT AND CONCRETE DATA STRUCTURE:

- All data types which are absolutely defined are called as concrete data types
- Abstract data type can be constructed from known –or unknown data.

For example: Boolean, Integer, Floating point, String are examples of concrete data types as they are very strictly defined to contain the specified data type values

Array can be an example of an abstract data type as an array can consist of a number of Booleans, integers, Alphanumeric, Text, or even arrays (A program can (if properly programmed) seek out a particular cell in an array, and return the data there, no matter what data type.)

Concrete data type is reverse of abstract data type.

Differences are:

- The concrete data type is defined for certain inputs and outputs, whereas abstract is defined for all kind of inputs and outputs.
- A concrete data type is rarely reusable, whereas abstract data types are reusable repetitively
- Arrays, lists and trees are concrete data types whereas stacks, queues and heaps are abstract data types.

For implementing abstract data type, we need to choose a suitable concrete data type.

ABSTRACT AND CONCRETE DATA STRUCTURES

Abstract Data Structures	Concrete Data Structures
Concrete data types or structures (CDT's) are direct implementations of a relatively simple concept	Abstract Data Types (ADT's) offer a high level view (and use) of a concept independent of its implementation
Data types which are absolutely defined	constructed from known –or unknown data
Array, records, linked lists, trees, graphs	stacks, queues and heaps
	Usually there are many ways to implement the same ADT, using several different concrete data structures. An abstract stack can be implemented by a linked list or by an array

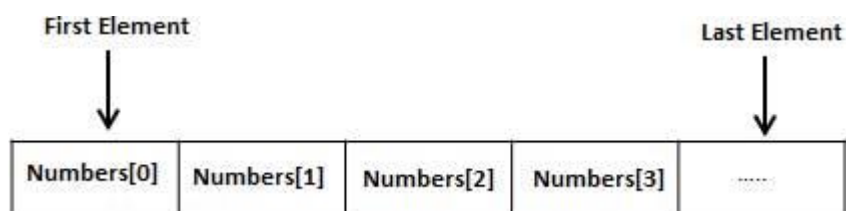
2.2 BASIC DATA STRUCTURE

2.2.1. Array

Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows –

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement –

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows –

```
double balance[5] = { 1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write –

```
double balance[] = { 1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array –

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above –

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Input data into array

```

for (x=0; x<=n;x++)
{
    printf("enter the integer number %d\n", x);
    scanf("%d", &num[x]);
}

```

Reading out data from an array

```

for (int i=0; i<n; i++)
{
    printf("%d\n", mydata[x]);
}

```

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example –

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays –

```

#include <stdio.h>

int main ()
{
    int n[ 10 ]; /* n is an array of 10 integers */
    int i,j;
    /* initialize elements of array n to 0 */
    for ( i = 0; i < 10; i++ )
    {
        n[ i ] = i + 100; /* set element at location i to i + 100 */
    }

    /* output each array element's value */
    for (j = 0; j < 10; j++ )

```

```

        {
            printf("Element[%d] = %d\n", j, n[j] );
        }
    return 0;
}

```

When the above code is compiled and executed, it produces the following result –

```

Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

```

Write a C Program to insert an element into the beginning of an array

```

#include<stdio.h>
#include<conio.h>
int main()
{
    int a[10],i,n,p;
    printf("Enter the number of elements\n");
    scanf("%d",&n);
    printf("Enter the Elements\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    if(n==10)
    {
        printf("Array is full");
    }
    else

```

```

{
    printf("Enter the element to be inserted")
    scanf("%d",&p);
    for(i=n-1;i>=0;i--)
    {
        a[i+1]=a[i];
    }
    n=n+1
    a[0]=p;
    printf("After insertion elements are:\n");
    for(i=0;i<n;i++)
    {
        printf("%d\t",a[i]);
    }
}
getch();
}

```

Write a C Program to insert an element into the end of an array

```

#include<stdio.h>
#include<conio.h>
int main()
{
    int a[10],i,n,p;
    printf("Enter the number of elements\n");
    scanf("%d",&n);
    printf("Enter the Elements\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    if(n==10)
    {
        printf("Array is full");
    }
    else

```

```

{
    printf("Enter the element to be inserted")
    scanf("%d",&p);
    a[n+1]=p;
    n=n+1;
    printf("After insertion elements are:\n");
    for(i=0;i<n;i++)
    {
        printf("%d\t",a[i]);
    }
}
getch();
}

```

Write a C Program to insert an element into the particular position of an array

```

#include <stdio.h>

int main()
{
    int array[100], position, c, n, value;
    printf("Enter number of elements in array\n");
    scanf("%d", &n);
    printf("Enter %d elements\n", n);
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
    printf("Enter the location where you wish to insert an element\n");
    scanf("%d", &position);

    if(position>=100)
        printf("Array is full")
    else
    {
        printf("Enter the value to insert\n");
        scanf("%d", &value);

        for (c = n - 1; c >= position - 1; c--)

```

```
array[c+1] = array[c];
```

```
array[position-1] = value;
```

```
n=n+1;
```

```
printf("Resultant array is\n");
```

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

```
    printf("%d\n", array[c]);
```

```
}
```

```
getch();
```

```
}
```

Write a C Program to delete an element from the beginning of an array

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

```
#include<conio.h>
```

```
int main()
```

```
{
```

```
    int a[10],i,n,p;
```

```
    printf("Enter the number of elements\n");
```

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

```
    printf("Enter the Elements\n");
```

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

```
{
```

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

```
}
```

```
        p=a[0];
```

```
        for(i=1;i<n;i++)
```

```
{
```

```
            a[i-1]=a[i];
```

```
}
```

```
        n=n-1;
```

```
        printf("Deleted elements are %d",p);
```

```
        if(n==0)
```

```
            printf("Array is empty");
```

```
    else
```



```

{
    printf("After deletion elements are:\n");
    for(i=0;i<n;i++)
    {
        printf("%d\t",a[i]);
    }
}
getch();
}

```

Write a C Program to delete an element from the end of an array

```

#include<stdio.h>
#include<conio.h>
int main()
{
    int a[10],i,n;
    printf("Enter the number of elements\n");
    scanf("%d",&n);
    printf("Enter the Elements\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    p=a[n];
    printf("Deleted element is %d",p);
    n=n-1;
    if(n==0)
        printf("Array is empty");
    else
    {
        printf("After deletion elements are:\n");
        for(i=0;i<n;i++)
        {
            printf("%d\t",a[i]);
        }
    }
}

```

```

    getch();
}

```

Write a C Program to delete an element from the particular position of an array

```

#include <stdio.h>

int main()
{
    int array[100], position, c, n, value;
    printf("Enter number of elements in array\n");
    scanf("%d", &n);
    printf("Enter %d elements\n", n);
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
    printf("Enter the location where you wish to delete an element\n");
    scanf("%d", &position);
    value=array[position-1];
    for (c = position; c >n; c--)
        array[c-1] = array[c];
    n=n-1;
    printf("Deleted element is %d\n",value);
    if(==0)
        printf("Array is empty");
else
{
    printf("Resultant array is\n");

    for (c = 0; c <=n; c++)
        printf("%d\n", array[c]);
}
    getch();
}

```

Two dimensional array

Ordered in number of rows and columns .Array $m*n$ denotes m number of row and n number of columns

$a_{11}, a_{12}, a_{13} \dots a_{1n}$

$a_{21}, a_{22}, a_{23} \dots a_{2n}$

.....

.....

$a_{m1}, a_{m2}, a_{m3} \dots a_{mn}$

Subscript $a[i][j]$ represents the i th row and j th column.

Memory representation of matrix

It is also stored in contiguous memory location. There are two conventions of storing matrix in memory

1. Row-major representation
2. Column-major representation

1. Row major representation

Elements of the matrix are stored in row by row basis. Assume that base address is the first location of the memory, address of $a[i][j]$ is Address $a[i][j]$ = storing all elements in the first $(i-1)$ rows + the number of elements in the i -th row up to j -th column $= (i-1)*n + j$, where n is the number of columns (Assume base address is 1)

If the base address is M Address $a[i][j] = M + (i-1)*n + j - 1$

2. Column major representation

All elements in the first column is stored first, then second column and so on. Address of $a[i][j]$ = storing all elements in the first $(j-1)$ columns elements + The number of elements in j -th column up to i -th row $= (j-1)*m + i$, where m is the number of rows (Assume base address is 1) If the base address is M Address of $a[i][j] = M + (j-1)*m + i - 1$

2.2.2. VECTOR

- Vector is an array with a dynamic size.
- Instead of having a predefined size to the structure it increases, decreases its size as you add/remove elements from/ to it.
- Which together gives as some other advantages as adding elements at a specific index and removing from a specific index.
- Vector is not as fast as the array, but altogether efficient. Operations on a vector offer the same big O as their counterparts on an array.
- Like arrays, vector data is allocated in contiguous memory. This can be done either explicitly or by adding more data.
- In order to do this efficiently, the typical vector implementation grows by doubling its allocated space (rather than incrementing it) and often has more space allocated to it at any one

time than it needs. This is because reallocating memory can sometimes be an expensive operation.

APPLICATIONS

1. Stores Elements of Same Data Type
2. Used for Maintaining multiple variable names using single name
3. Can be Used for Sorting Elements
4. Can Perform Matrix Operation
5. Can be Used in CPU Scheduling
6. Can be Used in Recursive Function
7. Can be used to implement abstract data structures like stack, queue etc

2.3 LINKED LIST

If the memory is allocated before the execution of a program, it is fixed and cannot be changed. We have to adopt an alternative strategy to allocated memory only when it is required. There is a special data structure called linked list that provides a more flexible storage system and it does not require the use of array.

Linked lists are special list of some data elements linked to one another. The logical ordering is represented by having each element pointing to the next element. Each element is called a node, which has two parts, INFO part which stores the information and LINK which points to the next element.

Advantages

Linked list have many advantages. Some of the very important advantages are:

- ***Linked Lists are dynamic data structure:*** That is, they can grow or shrink during the execution of a program.
- ***Efficient memory utilization:*** Here, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated when it is no longer needed.
- ***Insertion and deletions are easier and efficient:*** Linked lists provide flexibility in inserting data item at a specified position and deletion of a data item from the given position.
- ***Many complex applications can be easily carried out with linked lists.***

Disadvantages

- ***More Memory:*** If the numbers of fields are more, then more memory space is needed.
- Access to an arbitrary data item is little bit cumbersome and also time consuming.

Types of Linked List

Following are the various flavours of linked list.

- Simple Linked List – Item Navigation is forward only.
- Doubly Linked List – Items can be navigated forward and backward way.
- Circular Linked List – Last item contains link of the first element as next and first element has link to last element as prev.

Basic Operations

- Insertion – add an element at the beginning of the list.
- Display – displaying complete list.
- Search – search an element using given key.
- Delete – delete an element using given key

2.3.1 Singly Linked List

A linked list is a non-sequential collection of data items called nodes. These nodes in principles are structures containing fields. Each node in a linked list has basically two fields.

1. DATA field
2. NEXT field

The DATA field contains an actual value to be stored and processed. And, the NEXT field contains the address of the next data item in the linked list. The address used to access a particular node is known as a pointer. Therefore, the elements in a linked list are ordered not by their physical placement in memory but their logical links stored as part of the data within the node itself.

Note that, the link field of the last node contains NULL rather than a valid address. It is a NULL pointer and indicates the end of the list. External pointer(HEAD) is a pointer to the very first node in the linked list, it enables us to access the entire linked list.



Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – A new node may be inserted
 - At the beginning of the linked list
 - At the end of the linked list
 - At the specific position of the linked list
- **Deletion** – delete an element from the
 - Beginning of the linked list
 - End of the linked list
 - Specific position of the linked list
- **Display** – This operation is used to print each and every node's information.
- **Traversing** – It is a process of going through all the nodes of a linked list from one end to the other end. If we start traversing from the very first node towards the last node, it is called forward traversing. If the desired element is found, we signal operation "SUCCESSFUL". Otherwise, we signal it as "UNSUCCESSFUL".

Steps to create a linked list

Step 1 : Include alloc.h Header File

```
#include<alloc.h>
```

1. We don't know, how many nodes user is going to create once he execute the program.
2. In this case we are going to allocate memory using Dynamic Memory Allocation functions malloc.
3. Dynamic memory allocation functions are included in alloc.h

Step 2 : Define Node Structure

We are now defining the new global node which can be accessible through any of the function.

```
struct node
```

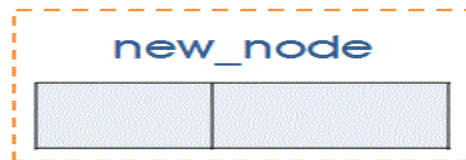
```
{
```

```
int data;
struct node *next;
}*start=NULL;
```

Step 3 : Create Node using Dynamic Memory Allocation

Now we are creating one node dynamically using malloc function. We don't have prior knowledge about number of nodes, so we are calling malloc function to create node at run time.

```
new_node=(struct node *)malloc(sizeof(struct node));
```

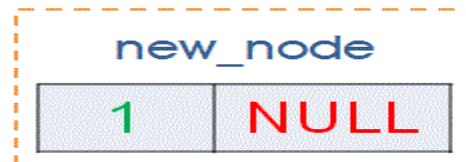



 Created New Node using Dynamic memory allocation

Step 4 : Fill Information in newly Created Node

Now we are accepting value from the user using scanf. Accepted Integer value is stored in the data field. Whenever we create new node, Make its Next Field as NULL.

```
printf("Enter the data : ");
scanf("%d",&new_node->data);
new_node->next=NULL;
```



 Fill Node with the data provided by user

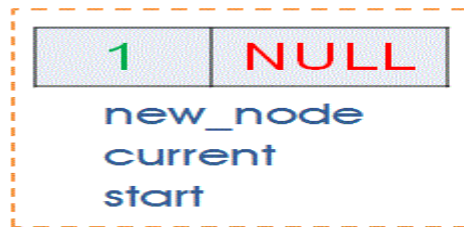
Step 5 : Creating Very First Node

If node created in the above step is very first node then we need to assign it as starting node. If start is equal to null then we can identify node as first node.

```
start = NULL
```

First node has 3 names : new_node, current, start

```
if(start == NULL) {
    start = new_node;
    curr = new_node;
}
```



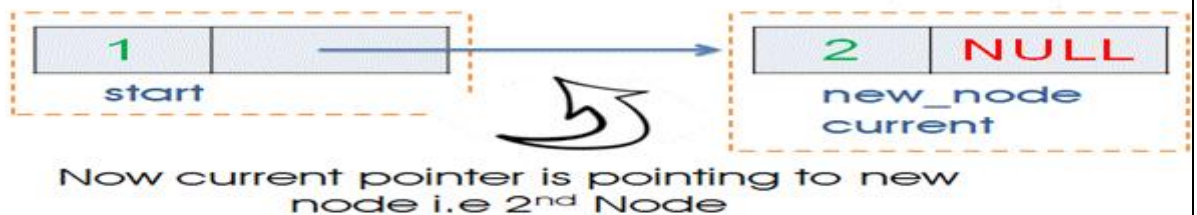
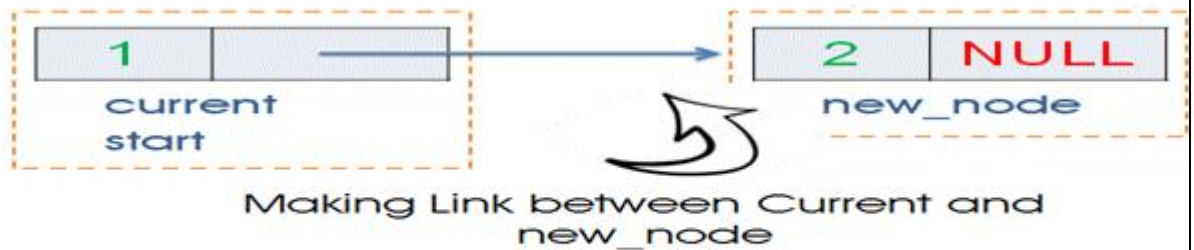
Below are the some of the alternate names given to the Created Linked List node

Step 6: Creating Second or nth node

1. Let's assume we have 1 node already created i.e we have first node. First node can be referred as "new_node", "curr", "start".
2. Now we have called create() function again
Now we already have starting node so control will be in the else block –

else

```
{
current->next = new_node; // link between new_node and current node
current = new_node; // move current pointer to next node
}
```



Insert node at Start/First Position in Singly Linked List

1. Allocate a memory for the new_node
2. Insert Data into the "Data Field" of new_node
3. If(start=NULL) list is empty then goto step 4 otherwise goto step 5
4. Start point to the new_node and set linked field of new_node to NULL

5. Linked field of the new_node pointed to start, then set start to new_node.



```

#include<stdio.h>
#include<conio.h>

struct node
{
    int data;
    struct node *ptr;
};

int main()
{
    typedef struct node NODE;
    NODE *start=NULL, *temp;
    int item;
    printf("\n\n\tSingle Linked List ");
    temp=(NODE*)malloc(sizeof(NODE));
    printf("Enter the data to be inserted: ");
    scanf("%d", &temp->data);

    if(start==NULL)
    {
        temp->ptr=NULL;
        start=temp;
    }

    else
    {
        temp->ptr=start;
        start=temp;
    }

    getch();
}

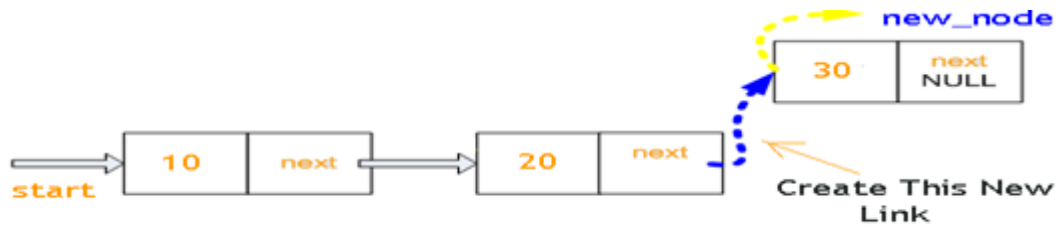
```

Insert node at Start/First Position in Singly Linked List

Algorithm

1. Allocate a memory for the new_node
2. Insert Data into the “Data Field“ of new_node and set “Linked field of new_node to NULL.
3. If(start=NULL) list is empty then goto step 4 otherwise goto step 5

4. Start point to the new_node and set linked field of new_node to NULL
5. Node is to be inserted at Last Position so we need to traverse SLL upto Last Node.
6. Linked field of last node pointed to the new node.



Program

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

struct node
{
    int data;
    struct node *ptr;
};

int main()
{
    typedef struct node NODE;
    NODE *start=NULL, *temp, *p;
    int item;

    printf("\n\n\tSingle Linked List ");
    printf("\n Insert into end");
    temp=(NODE*)malloc(sizeof(NODE));
    printf("Enter the data to be inserted: ");
    scanf("%d", &item);
    item=temp->data;

    if(start==NULL)
    {
        temp->ptr=NULL;
        start=temp;
    }

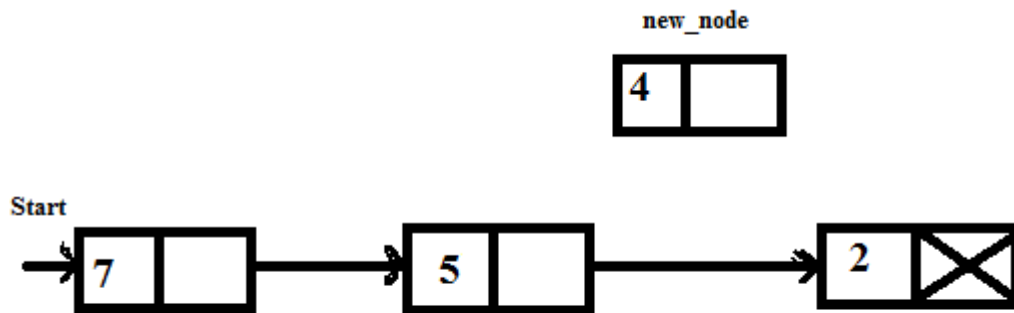
    else
    {
        p=start;
        while(p->ptr!=NULL)
        {
            p=p->ptr;
        }
        p->ptr=temp;
        temp->ptr=NULL;
    }
    getch();
}
```

}

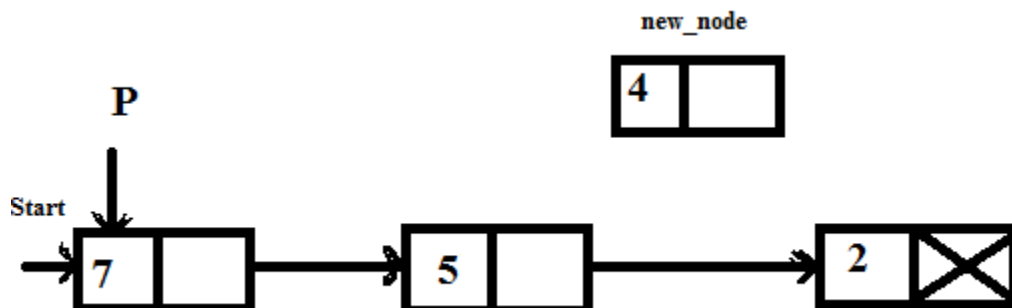
Insert node at Particular Position in Singly Linked List

Algorithm

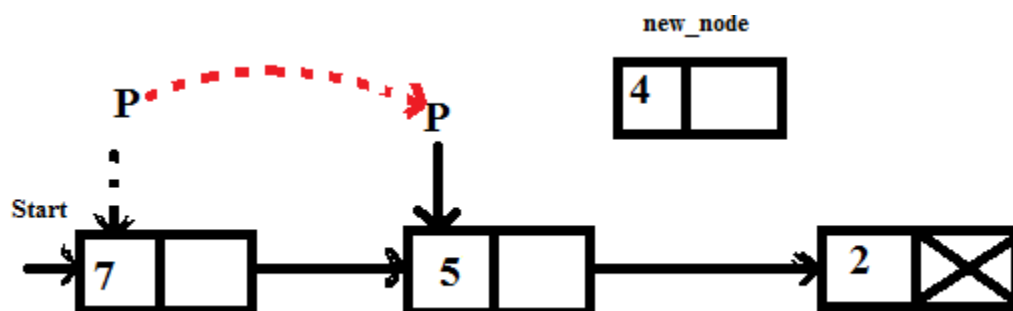
1. Allocate memory for the new node
2. Assign value to the data field of the new node
3. Find the position to insert
4. Suppose new node insert between node A and node B, make the link field of the new node point to the node B and make the link field of node A point to the new node

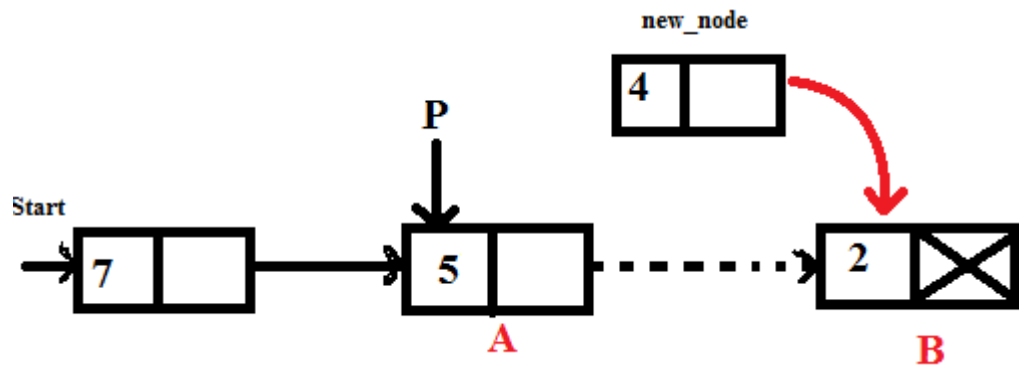
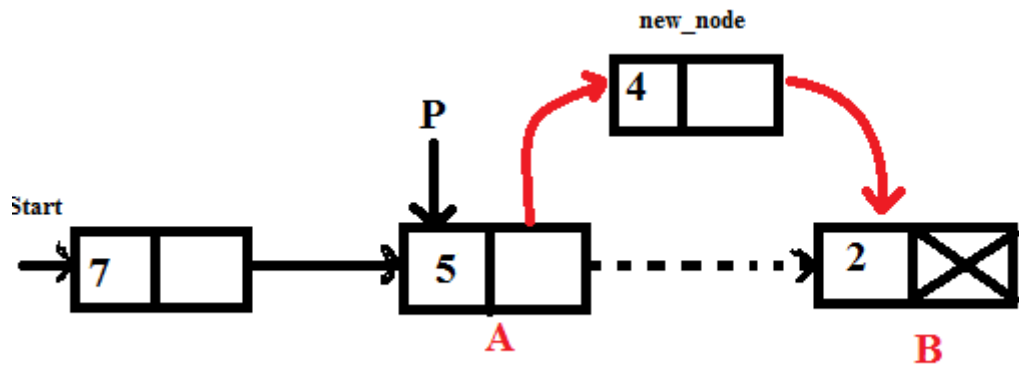


(a) Create a new node and insert the data into the data field



(b) p=start



(c) $p = p \rightarrow \text{ptr}$ (d) $\text{new_node} \rightarrow \text{ptr} = p \rightarrow \text{ptr}$ (d) $p \rightarrow \text{ptr} = \text{new_node}$ Program

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node*ptr;
};

void main()
{
    typedef struct node NODE;
    NODE *start=NULL,*temp,*p,*t;

```

```

int ch,item,pos,i;
printf("\nEnter the number: ");
scanf("%d",&item);
temp=(NODE*)malloc(sizeof(NODE));
temp->data=item;
printf("\nEnter the position: ");
scanf("%d",&pos);
p=start;
for(i=1;i<pos-1;i++)
{
    p=p->ptr;
}
temp->ptr=p->ptr;
p->ptr=temp;
getch();
}

```

Display the elements in the linked list

1. If start=NULL, Print 'No elements'
2. Otherwise print elements from start to NULL

```

if(start==NULL)
    printf("\nList is empty");
else
{
    printf("\nElements are:");
    for(p=start;p!=NULL;p=p->ptr)
        printf(" %d",p->data);
}

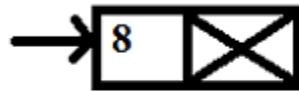
```

Delete node at Start/First Position in Singly Linked List

Algorithm

1. If list is empty, deletion is not possible.
2. If the list contain only one element, set external pointer to NULL. Otherwise move the external pointer point to the second node and delete the first node

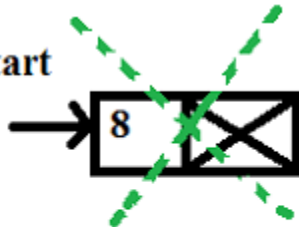
Start



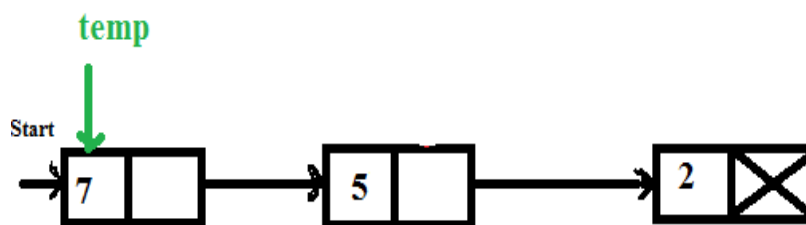
start->ptr=NULL

start->ptr=NULL
(Only one element)

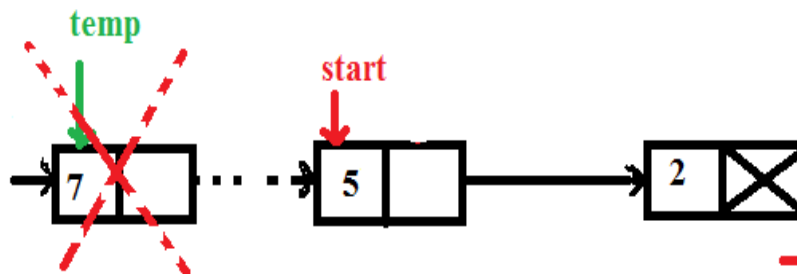
Start



start=NULL



More than one
elements



Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node*ptr;
};
```

```

void main()
{
    typedef struct node NODE;
    NODE *start=NULL,*temp;
    int item;
    if(start==NULL)
        printf("\nDeletion is not possible");
    else if(start->ptr==NULL)
    {
        temp=start;
        start=NULL;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    else
    {
        temp=start;
        start=start->ptr;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    getch();
}

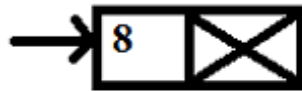
```

Delete node at End/Last Position in Singly Linked List

Algorithm

1. If the list is empty, deletion is not possible
2. If the list contain only one element, set external pointer to NULL
3. Otherwise go on traversing the second last node and set the link field point to NULL

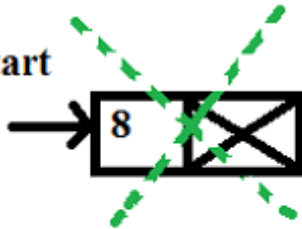
Start



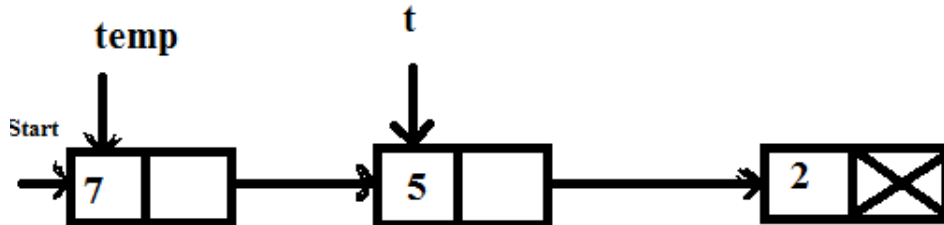
start->ptr=NULL

start->ptr=NULL
(Only one element)

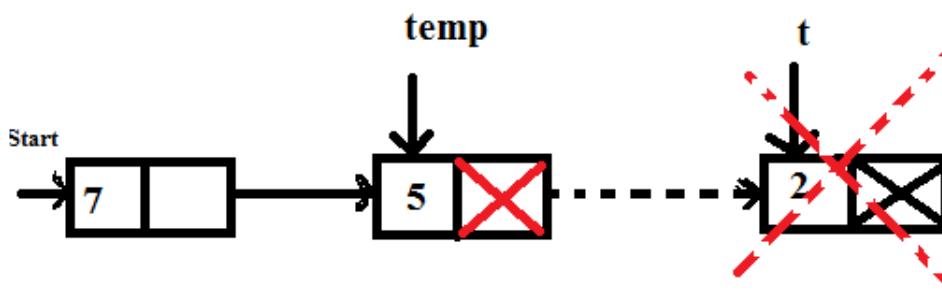
Start



start=NULL



more than one
node



Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node*ptr;
};

void main()
```



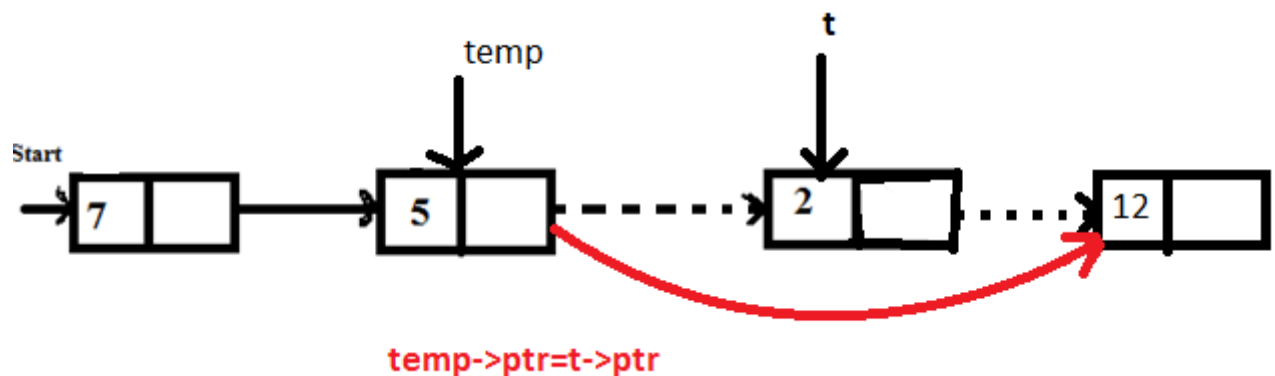
```

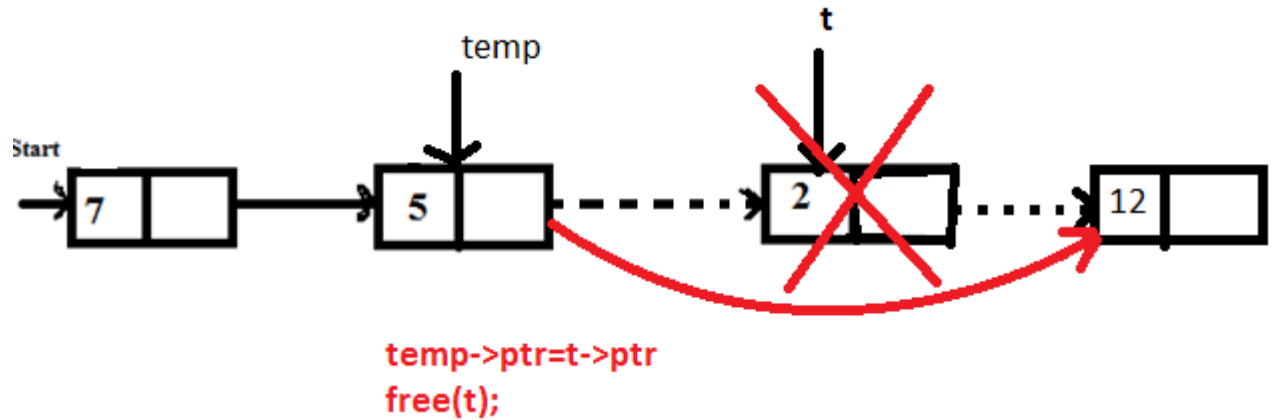
{
    typedef struct node NODE;
    NODE *start=NULL,*temp,*p,*t;
    int ch,item,pos,i;
    if(start==NULL)
        printf("\nDeletion is not possible");
    else if(start->ptr==NULL)
    {
        temp=start;
        start=NULL;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    else
    {
        temp=start;
        t=start->ptr;
        while(t->ptr!=NULL)
        {
            t=t->ptr;
            temp=temp->ptr;
        }
        temp->ptr=NULL;
        printf("\nDeleted item is %d",t->data);
        free(t);
    }
    getch();
}

```

Delete node at Particular Position in Singly Linked List

1. Find the position to delete.
2. Suppose we delete node between node A & node B, set the link field of node A point to the node B





Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node*ptr;
};

void main()
{
    typedef struct node NODE;
    NODE *start=NULL,*temp,*p,*t;
    int ch,item,pos,i;

    printf("\nEnter the position: ");
    scanf("%d",&pos);
    temp=start;
    if(pos==1)
    {
        temp=start;
        start=start->ptr;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    else
    {
        for(i=1;i<pos-1;i++)
            temp=temp->ptr;
        t=temp->ptr;
        temp->ptr=t->ptr;
        printf("\nDeleted item is %d",t->data);
        free(t);
    }
}
```

```

    }
    getch()
}

```

To write a program to implement singly linked list.

- 1. Insert at beginning**
- 2. Insert at particular position**
- 3. Insert at end**
- 4. Delete from beginning**
- 5. Delete from particular position**
- 6. Delete from end**
- 7. Display**

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node*ptr;
};

void main()
{
    typedef struct node NODE;
    NODE *start=NULL,*temp,*p,*t;
    int ch,item,pos,i;
    clrscr();
    while(1)
    {
        printf("\nMENU: \n1.Insert at beginning\n2.Insert at particular position\n3.Insert at end\n4.Delete
from beginning\n5.Delete from particular position\n6.Delete from end\n7.Display\n8.Exit\nEnter your
choice: ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                printf("\nEnter the number: ");
                scanf("%d",&item);
                temp=(NODE*)malloc(sizeof(NODE));
                temp->data=item;

```

```

    if(start==NULL)
    {
        temp->ptr=NULL;
        start=temp;
    }
    else
    {
        temp->ptr=start;
        start=temp;
    }
    break;
case 2:
    printf("\nEnter the number: ");
    scanf("%d",&item);
    temp=(NODE*)malloc(sizeof(NODE));
    temp->data=item;
    printf("\nEnter the position: ");
    scanf("%d",&pos);
    p=start;
    for(i=1;i<pos-1;i++)
        p=p->ptr;
    temp->ptr=p->ptr;
    p->ptr=temp;

    break;
case 3:
    printf("\nEnter the number: ");
    scanf("%d",&item);
    temp=(NODE*)malloc(sizeof(NODE));
    temp->data=item;
    temp->ptr=NULL;
    if(start==NULL)
        start=temp;
    else
    {
        p=start;
        while(p->ptr!=NULL)
            p=p->ptr;
        p->ptr=temp;
    }
    break;
case 4:
    if(start==NULL)
        printf("\nDeletion is not possible");
    else if(start->ptr==NULL)
    {

```

```

        temp=start;
        start=NULL;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    else
    {
        temp=start;
        start=start->ptr;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    break;
case 5:

```

```

    printf("\nEnter the position: ");
    scanf("%d",&pos);
    temp=start;

    for(i=1;i<pos-1;i++)
        temp=temp->ptr;
    t=temp->ptr;
    temp->ptr=t->ptr;
    printf("\nDeleted item is %d",t->data);
    free(t);
    break;

```

```

case 6:
    if(start==NULL)
        printf("\nDeletion is not possible");
    else if(start->ptr==NULL)
    {
        temp=start;
        start=NULL;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    else
    {
        temp=start;
        t=start->ptr;
        while(t->ptr!=NULL)
        {
            t=t->ptr;
            temp=temp->ptr;
        }
        temp->ptr=NULL;
    }

```

```

        printf("\nDeleted item is %d",t->data);
        free(t);
    }
    break;
case 7:
    if(start==NULL)
        printf("\nList is empty");
    else
    {
        printf("\nElements are:");
        for(p=start;p!=NULL;p=p->ptr)
            printf(" %d",p->data);
    }
    break;
case 8:
    exit(0);
default:
    printf("\nWrong Choice");
    break;
}
getch();
}
}

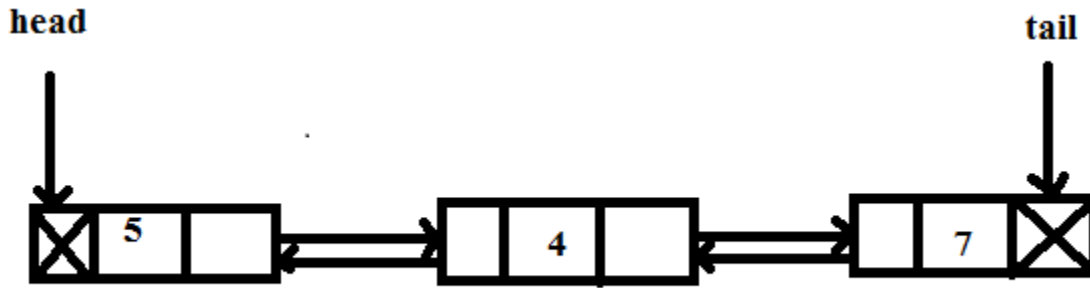
```

DOUBLY LINKED LIST

A more sophisticated kind of linked list is a doubly-linked list or a two-way linked list. In a doubly linked list, each node has two links: one pointing to the previous node and one pointing to the next node.

Adress of the pervious node	Data filed	Adress of the Next node
--	-------------------	------------------------------------

Example:



head is the external pointer ,used to point starting of the doubly linked list. Tail is used to denote end of the doubly linked list.

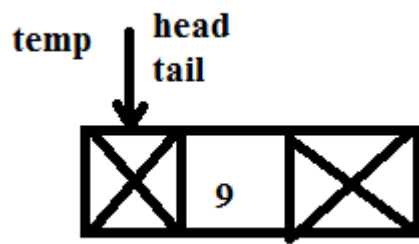
Representation of doubly linked list

```
struct node
```

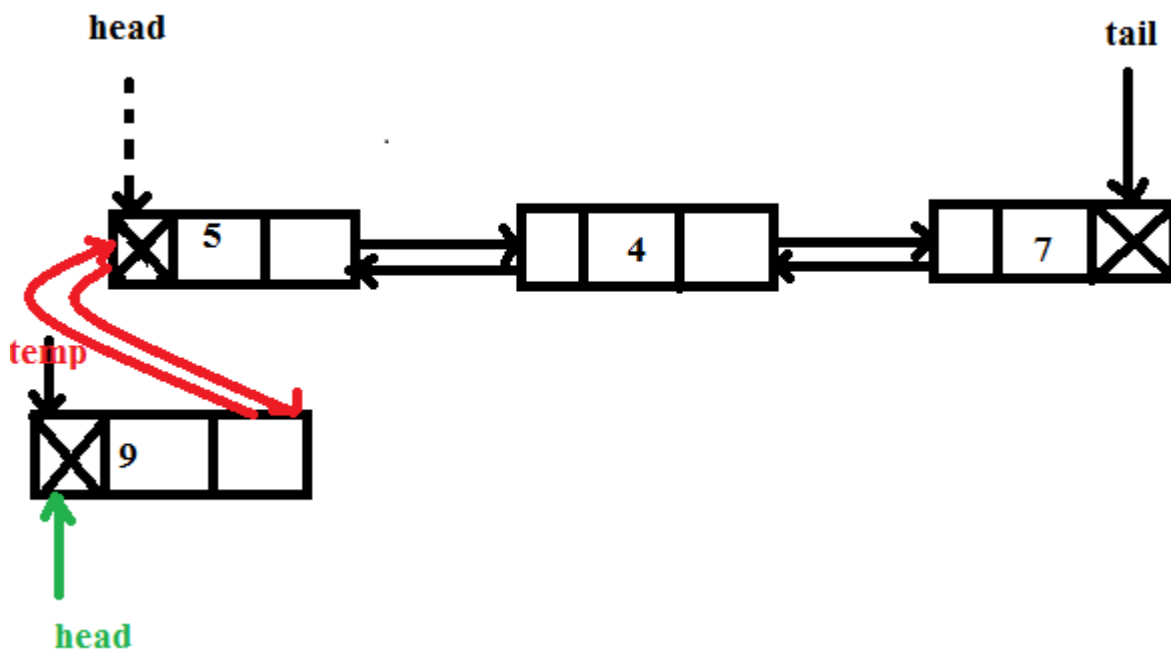
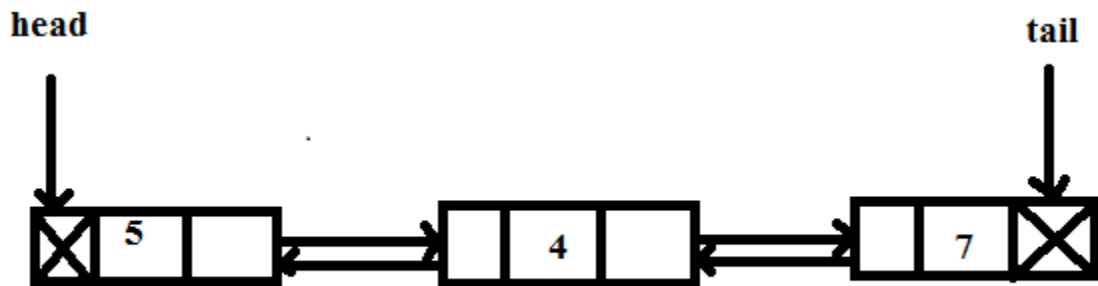
```
{
    int data;
    struct node *prev,*next;
};
```

Insert node at Start/First Position in Doubly Linked List

1. Allocate memory for new node
2. Assign value to the data field of the new node
3. If head=NULL then,
Set head and tail pointer point to the new node, set previous node is NULL and next node is NULL
4. Otherwise
Set temp→next=head and head→prev=temp and temp→prev=NULL
Set external pointer head point to the new node



head=NULL



head!=NULL

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
struct node
{
    int data;
    struct node *prev,*next;
```



```

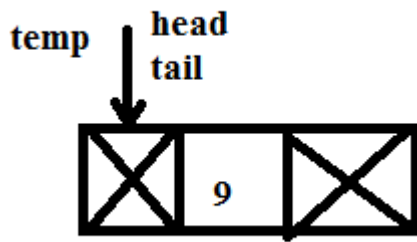
};
void main()
{
    typedef struct node NODE;
    NODE *head=NULL,tail=NULL,*temp;
    int no;
    temp=(NODE*)malloc(sizeof(NODE));
    printf("Enter the no: ");
    scanf("%d",&no);
    temp->data=no;
    if(start==NULL)
    {
        temp->prev=NULL;
        temp->next=NULL;
        head=tail=temp;
    }
    else
    {
        temp->next=head;
        head->prev=temp;
        temp->prev=NULL;
        head=temp;
    }
    if(head==NULL)
    {
        printf("No elements");
    }
    else
    {
        printf("\nElements are:");
        for(p= head;p!=NULL;p=p->next)
        {
            printf(" %d",p->data);
        }
    }
    getch();
}

```

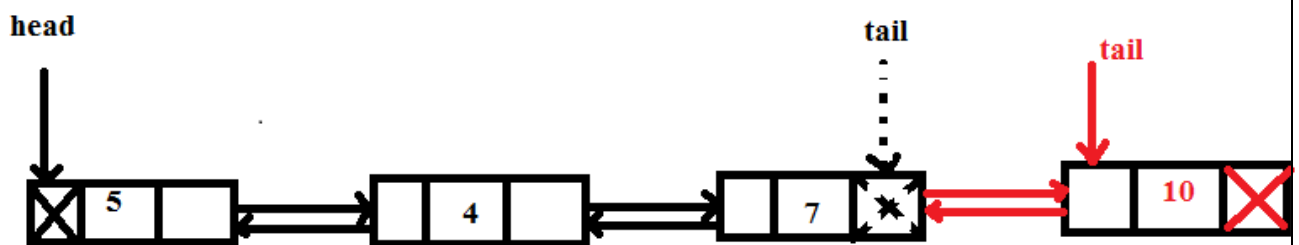
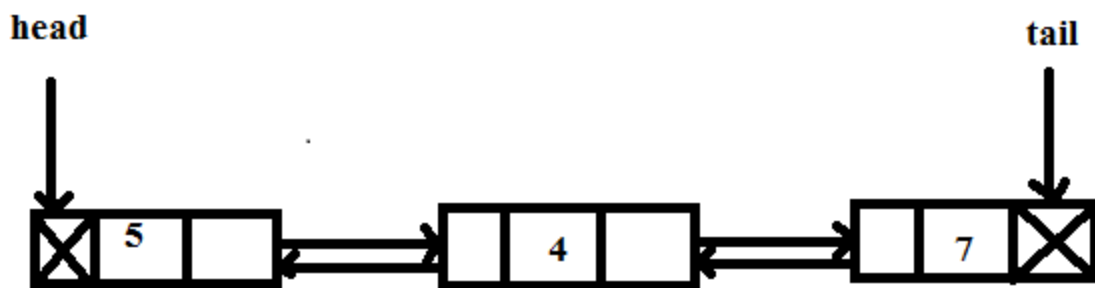
Insert node at End/Last Position in Doubly Linked List

1. Allocate memory for new node
2. Assign value to the new node
3. If head=NULL then,Set external pointer to the new node, set previous node is NULL and next node is NULL

4. Otherwise, $\text{tail} \rightarrow \text{next} = \text{temp}$, $\text{temp} \rightarrow \text{prev} = \text{tail}$ and next pointer of new node set to NULL. tail point to the new node



$\text{head} = \text{NULL}$



$\text{head} \neq \text{NULL}$

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
struct node
{
    int data;
    struct node *prev,*next;
};
void main()
{
```

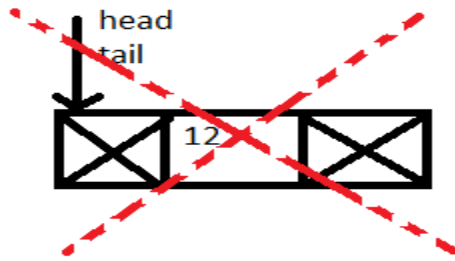
```

int ch,no;
typedef struct node NODE;
NODE *head=NULL,*tail=NULL,*temp;
temp=(NODE*)malloc(sizeof(NODE));
printf("Enter the no: ");
scanf("%d",&no);
temp->data=no;
    if(start==NULL)
    {
        temp->prev=NULL;
        temp->next=NULL;
        head=tail=temp;
    }
    else
    {
        tail->next=temp;
        temp->prev=tail;
        temp->next=NULL;
        tail=temp;
    }
    if(head==NULL)
    {
        printf("No elements");
    }
    else
    {
        printf("\nElements are:");
        for(p= head;p!=NULL;p=p->next)
        {
            printf(" %d",p->data);
        }
    }
getch();
}

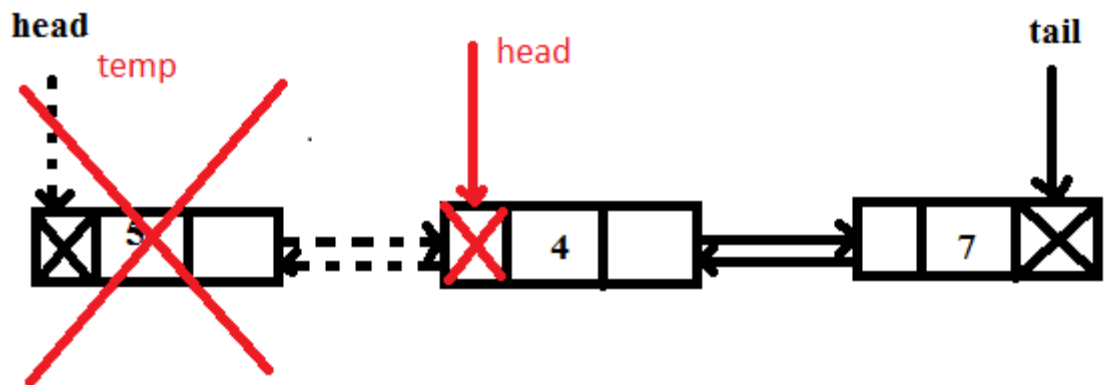
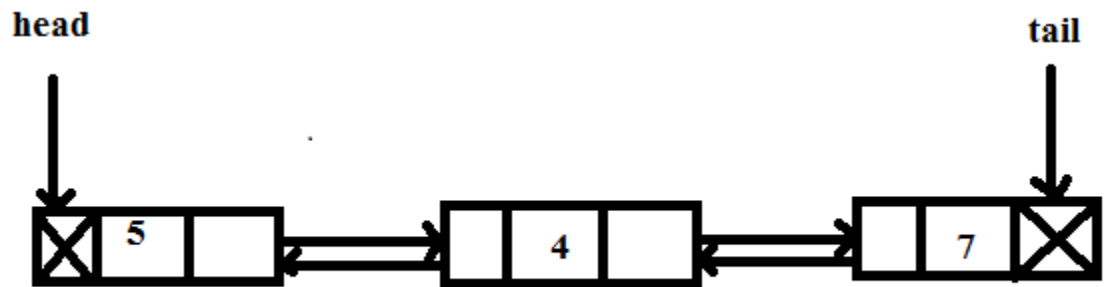
```

Delete a node at Start/First Position in Doubly Linked List

1. If start=NULL then,
Print deletion is not possible
2. If the list contain only one element, set external pointer to NULL
3. Otherwise move the external pointer point to the second node and delete first node



head=tail=NULL



```
#include<stdio.h>
#include<conio.h>
struct node
{
    int data;
    struct node *prev,*next;
};
void main()
{
    int no;
    typedef struct node NODE;
    NODE *head=NULL,*temp;
    if(head==NULL)
    {
        printf("Deletion is not possible");
    }
}
```

```

else if(head->next==NULL)
{
    temp=start;
    head=tail=NULL;
    printf("Deleted element is: %d",temp->data);
    free(temp);
}
else
{
    temp=head;
    head=temp->next;
    head->prev=NULL;
    printf("Deleted element is: %d",temp->data);
    free(temp);
}
if(head==NULL)
{
    printf("No elements");
}
else
{
    printf("\nElements are:");
    for(p=head;p!=NULL;p=p->next)
    {
        printf(" %d",p->data);
    }
}

```

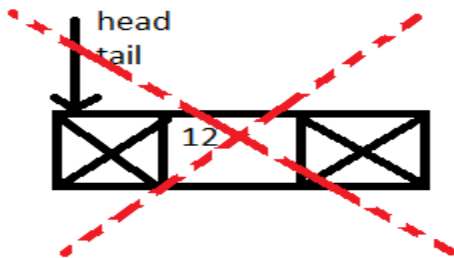
```

getch();
}

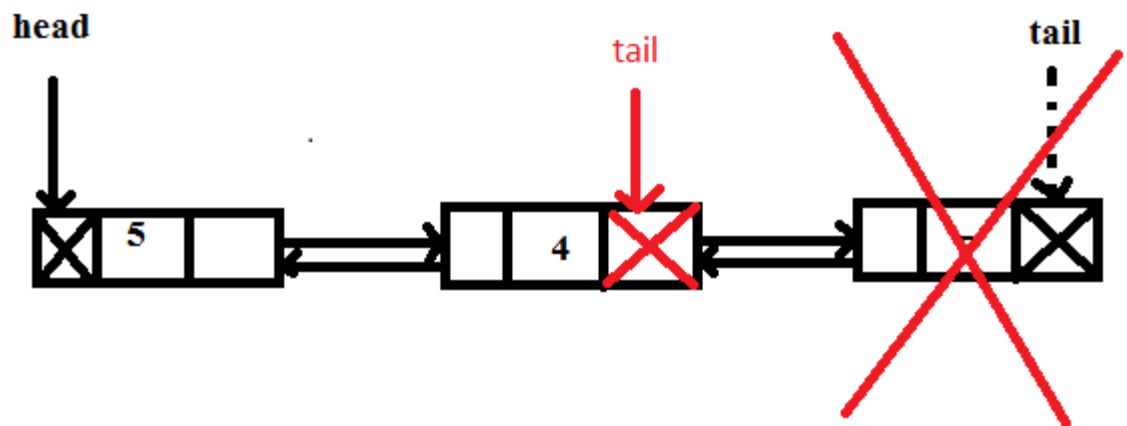
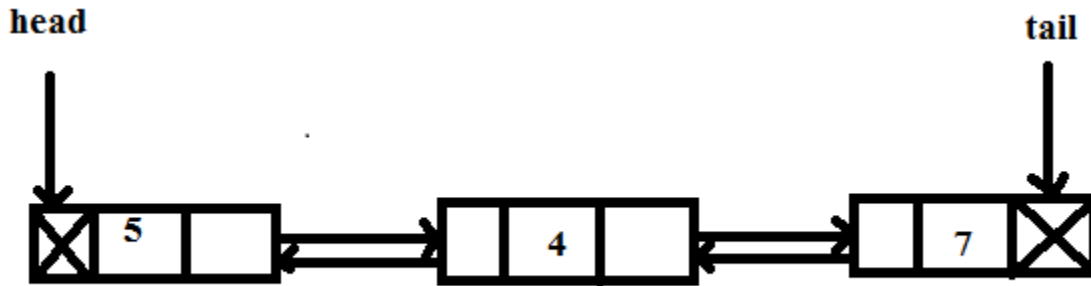
```

Delete a node at End/Last Position in Doubly Linked List

1. If the list is empty, deletion is not possible
2. If the list contain only one element, set external pointer to NULL
1. Otherwise go on traversing the last and set next field of second last node to NULL



head=tail=NULL



```

#include<stdio.h>
#include<conio.h>
struct node
{
    int data;
    struct node *prev,*next;
};
void main()
{
    int no;
    typedef struct node NODE;
    NODE *head=NULL,tail=NULL,*temp;
    if(head==NULL)
    {
        printf("Deletion is not possible");
    }
    else if(head->next==NULL)
    {
        temp=start;
        head=tail=NULL;
    }
  
```

```

        printf("Deleted element is: %d",temp->data);
        free(temp);
    }
    else
    {
        temp=tail;
        tail=tail->prev;
        free(temp);
    }
    if(head==NULL)
    {
        printf("No elements");
    }
    else
    {
        printf("\nElements are:");
        for(p=head;p!=NULL;p=p->next)
        {
            printf(" %d",p->data);
        }
    }

    getch();
}

```

To write a program to implement doubly linked list.

- 1. Insert at beginning**
- 2. Insert at end**
- 3. Delete from beginning**
- 4. Delete from end**
- 5. Display**

```

#include<stdio.h>
#include<conio.h>
struct node
{
    int data;
    struct node *prev,*next;
};
void main()
{
    int ch,no;
    typedef struct node NODE;
    NODE *head=NULL,tail=NULL,*temp,*p,*t;
    clrscr();
    while(1)
    {

```

```

printf("\nMENU\n1.Insert at beginning\n2.Insert at end"
      "\n3.Delete from beginning\n4.Delete from end\n5.Display\n6.Exit");
printf("\nEnter your choice: ");
scanf("%d",&ch);
switch(ch)
{
    case 1:
        temp=(NODE*)malloc(sizeof(NODE));
printf("Enter the no: ");
scanf("%d",&no);
temp->data=no;
        if(start==NULL)
        {
            temp->prev=NULL;
            temp->next=NULL;
            head=temp;
        }
        else
        {
            temp->next=head;
            head->prev=temp;
            temp->prev=NULL;
            head=temp;
        }
        break;
    case 2:
        temp=(NODE*)malloc(sizeof(NODE));
printf("Enter the no: ");
scanf("%d",&no);
temp->data=no;
        if(start==NULL)
        {
            temp->prev=NULL;
            temp->next=NULL;
            head=temp;
        }
        else
        {
            tail->next=temp;
            temp->prev=tail;
            temp->next=NULL;
            tail=temp;
        }
        break;
    case 3:
        if(head==NULL)

```



```

        {
            printf("Deletion is not possible");
        }
    else if(head->next==NULL)
    {
        temp=start;
        head=tail=NULL;
        printf("Deleted element is: %d",temp->data);
        free(temp);
    }
    else
    {
        temp=head;
        head=temp->next;
        head->prev=NULL;
        printf("Deleted element is: %d",temp->data);
        free(temp);
    }
    break;
case 4:
if(head==NULL)
{
    printf("Deletion is not possible");
}
else if(head->next==NULL)
{
    temp=start;
    head=tail=NULL;
    printf("Deleted element is: %d",temp->data);
    free(temp);
}
else
{
    temp=tail;
    tail=tail->prev;
    free(temp);
}

    break;
case 5:
    if(start==NULL)
    {
        printf("No elements");
    }
    else
    {

```

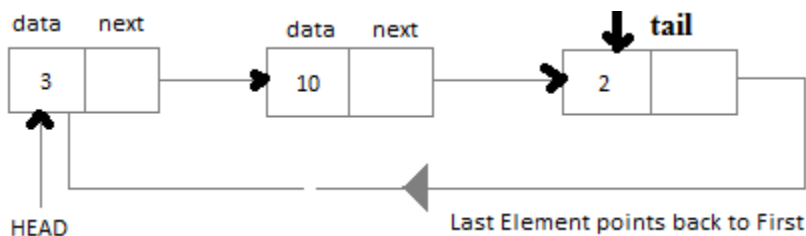
```

        printf("\nElements are:");
        for(p=head;p!=NULL;p=p->next)
        {
            printf(" %d",p->data);
        }
    }
    break;
case 6:
    exit(0);
}
getch();
}
}

```

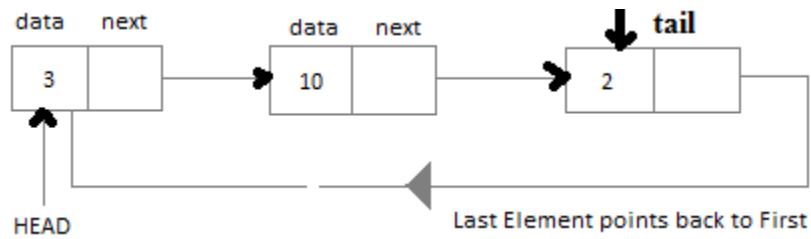
CIRCULAR LINKED LIST

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list where as in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required. A circular linked list has no end. Therefore, it is necessary to establish the **head** and **tail** nodes in such a linked list,

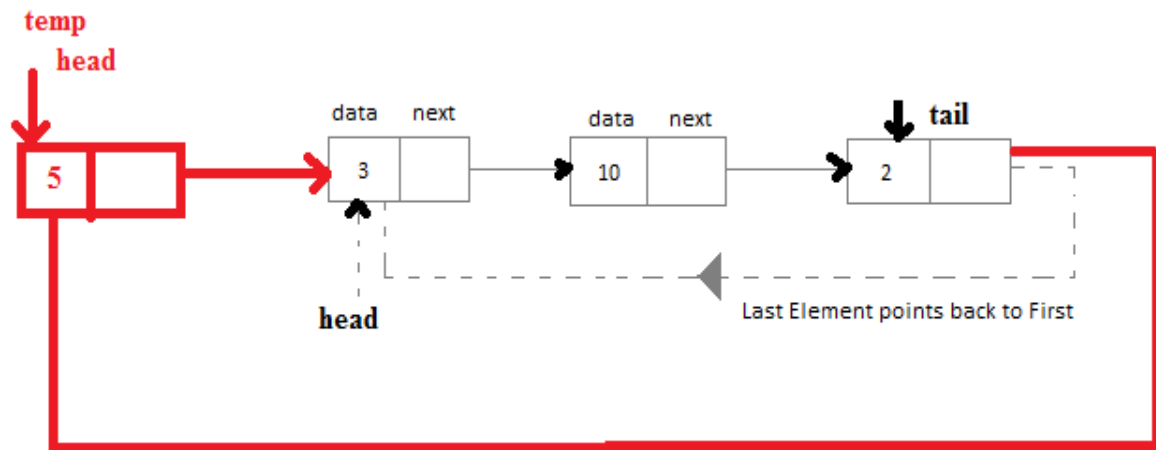


Inserting a node at the beginning

1. Allocate a memory for new node
2. If list is empty then head and tail point to the new node. And linked field of tail pointed to head.
3. If the list is not empty then
 - a) New node pointed to head
 - b) Head point to new node
 - c) Linked field of tail pointed to head



After insertion



We declare the structure for the circular linked list in the same way as we declare it for the linear linked lists

```
struct node
{
int data;
struct node *next;
};
typedef struct node NODE;
NODE *head=NULL;
NODE *tail=NULL;
```

```
-----
NODE *temp;
temp=(struct NODE*)malloc(sizeof(NODE));
printf("Enter the element to be inserted")
scanf("%d",&item);
```

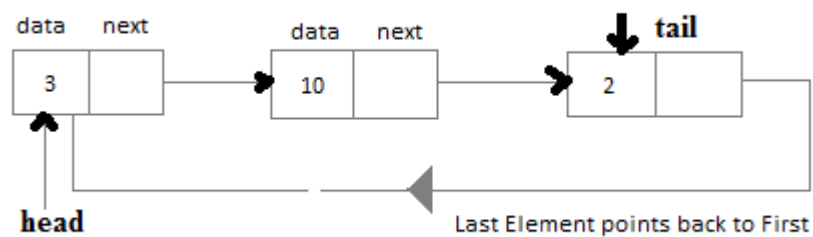
```

temp->data=item;
if(head==NULL)
{
    head=tail=temp;
    tail->next=head;
}
else
{
    temp->next=head;
    head=temp;
    tail->next=head;
}

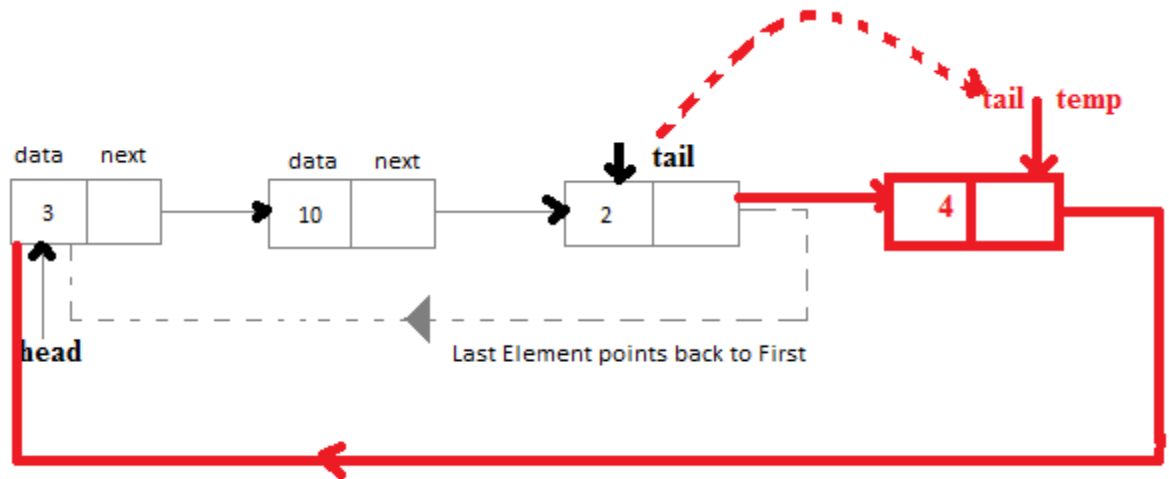
```

Inserting a node at the End

1. Allocate a memory for new node
2. If list is empty then head and tail point to the new node. And linked field of tail pointed to head.
3. If the list is not empty then
 - a) Linked field of tail point to the new node
 - b) tail point to new node
 - c) Linked field of tail pointed to head



After insertion



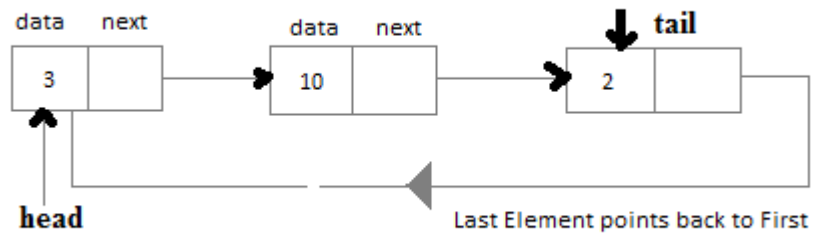
```

NODE *temp;
temp=(struct NODE*)malloc(sizeof(NODE));
printf("Enter the element to be inserted")
scanf("%d",&item);
temp->data=item;
if(head==NULL)
{
    head=tail=temp;
    tail->next=head;
}
else
{
    tail->next=temp;
    tail=temp;
    tail->next=head;
}

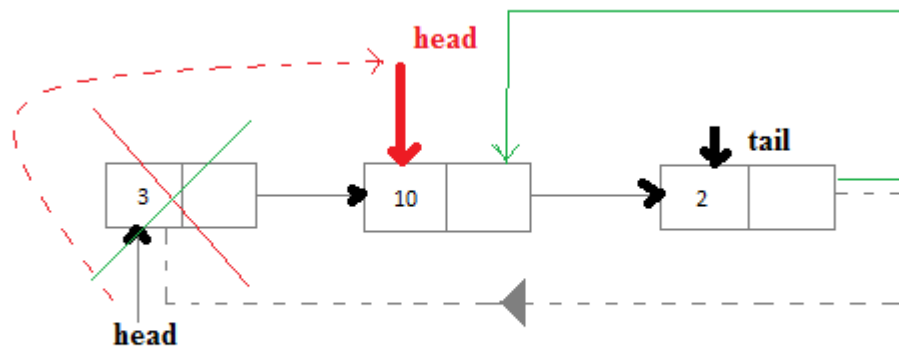
```

Delete a node from the beginning

1. If list is empty then print deletion is not possible
2. If the list contain only on element the set head and tail to NULL
3. If the list contain more than one element then
 - a) temp pointer point to the head node
 - b) head move to the next node
 - c) Linked field of tail pointed to head



After Deletion



```

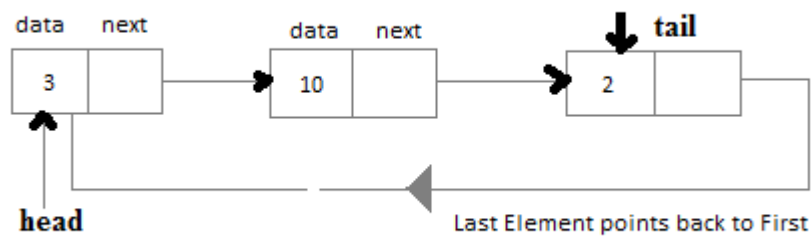
if(head==NULL)    //List is empty
{
    printf("Deletion is not possible\n");
}
else if(head==tail)    //List contains only one node
{
    free(head);
    head=tail=NULL;
}
else
{
    temp=head;
    head=head->next;
    tail->next=head;
    free(temp);
}

```

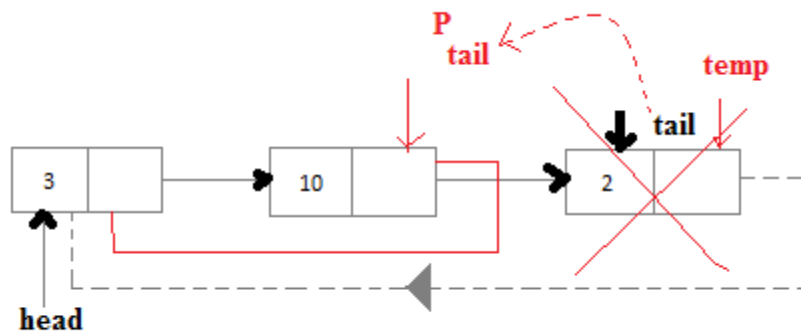
}

Delete a node from the beginning

1. If list is empty then print deletion is not possible
2. If the list contain only on element the set head and tail to NULL
3. If the list contain more than one element then
 - a) 'p' pointer point to the second last node and temp pointer points to the last node
 - b) tail move to the p
 - c) Linked field of tail pointed to head
 - d) Delete the last node



After Deletion



```

if(head==NULL)    //List is empty
{
printf("Deletion is not possible\n");
}
else if(head==tail)    //List contains only one node
{
free(head);

```

```

    head=tail=NULL;
}
else
{
    p=head;
    while(p->next!=tail)
    {
        p=p->next;
    }
    temp=p->next;
    tail=p;
    tail->next=head;
    free(temp);
}

```

Application of linked list-Polynomial

Polynomial Addition

Linked list are widely used to represent and manipulate polynomials. Polynomials are the expressions containing number of terms with nonzero coefficient and exponents. In the linked representation of polynomials, each term is considered as a node. And such a node contains three fields

- Coefficient field
- Exponent field
- Link field

The coefficient field holds the value of the coefficient of a term and the exponent field contains the exponent value of the term. And the link field contains the address of the next term in the polynomial. The polynomial node structure is

Coefficient(coeff)	Exponent(expo)	Address of the next node(next)

Algorithm

Two polynomials can be added. And the steps involved in adding two polynomials are given below

1. Read the number of terms in the first polynomial P

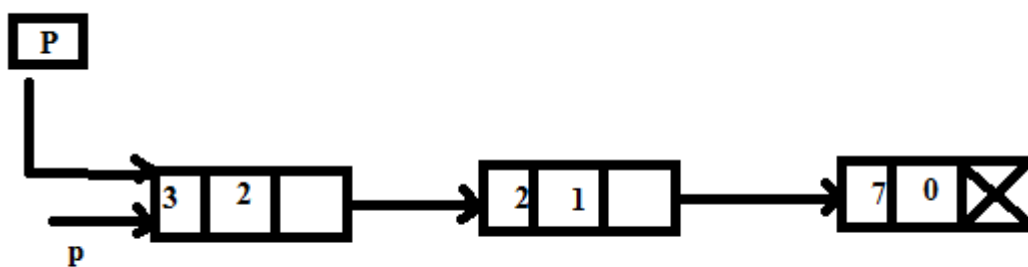
2. Read the coefficient and exponent of the first polynomial
3. Read the number of terms in the second polynomial Q
4. Read the coefficient and exponent of the second polynomial
5. Set the temporary pointers p and q to travers the two polynomials respectively
6. Compare the exponents of two polynomials starting from the first nodes
 - a) If both exponents are equal then add the coefficient and store it in the resultant linked list
 - b) If the exponent of the current term in the first polynomial P is less than the exponent of the current term of the second polynomial then added the second term to the resultant linked list. And, move the pointer q to point to the next node in the second polynomial Q.
 - c) If the exponent of the current term in the first polynomial P is greater than the exponent of the current term in the second polynomial Q, then the current term of the first polynomial is added to the resultant linked list. And move the pointer p to the next node.
 - d) Append the remaining nodes of either of the polynomials to the resultant linked list.

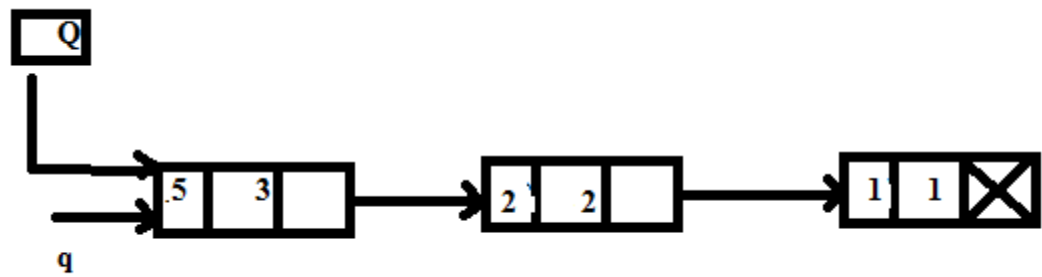
Let us illustrate the way the two polynomials are added. Let p and q be two polynomials having three terms each.

$$P=3x^2+2x+7$$

$$Q=5x^3+2x^2+x$$

These two polynomial can be represented as



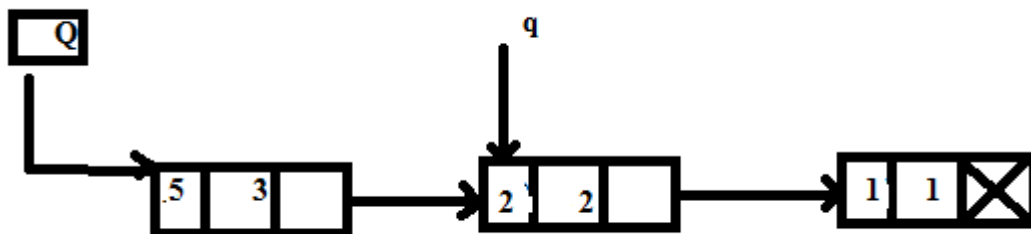
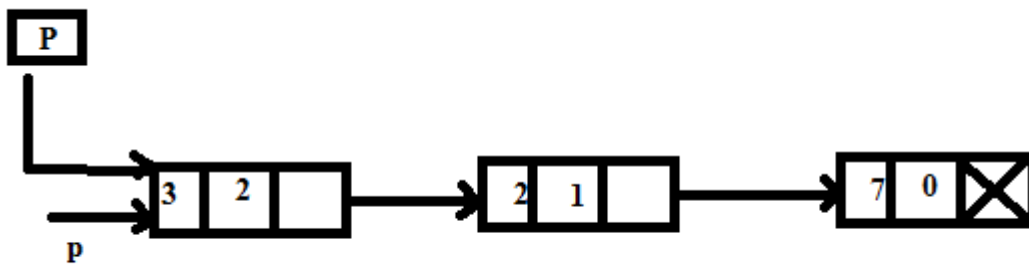


Step 1. Compare the exponent of p and the corresponding exponent of q. Here,

$$\text{expo}(p) < \text{expo}(q)$$

So, add the terms pointed to by q to the resultant list. And now advance the q pointer.

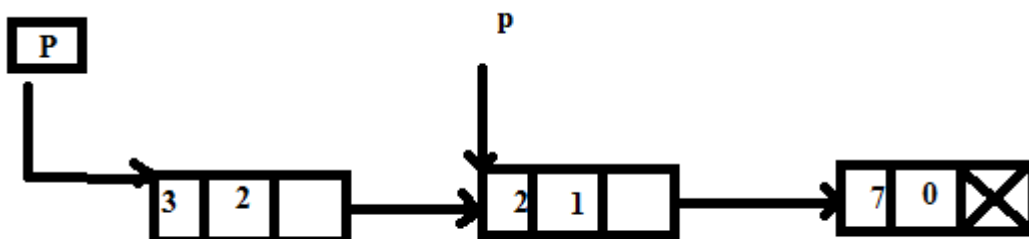
Step 2.

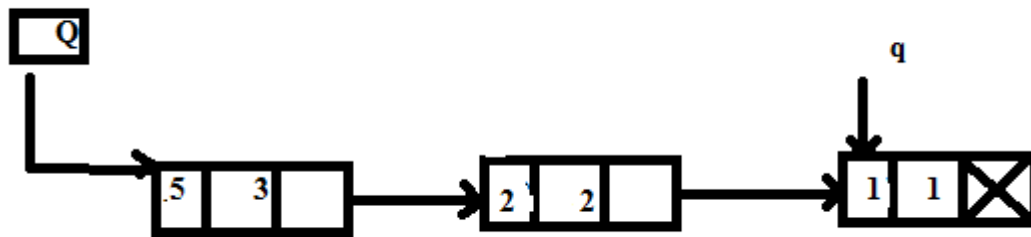


Compare the exponent of the current terms. Here,

$$\text{expo}(p) = \text{expo}(q)$$

So, add the coefficients of these two terms and link this to the resultant list. And, advance the pointers p and q to their next nodes.



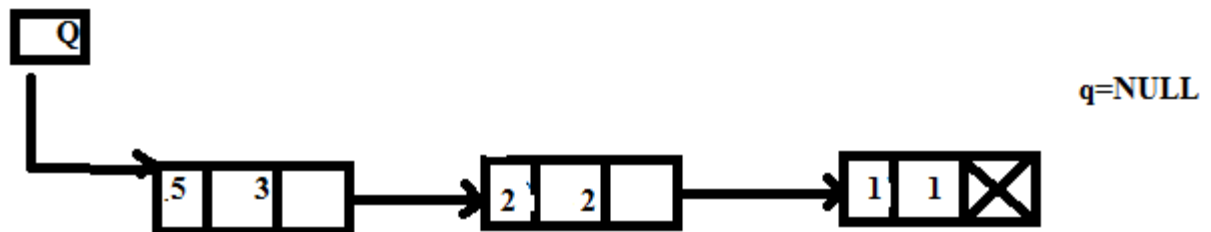
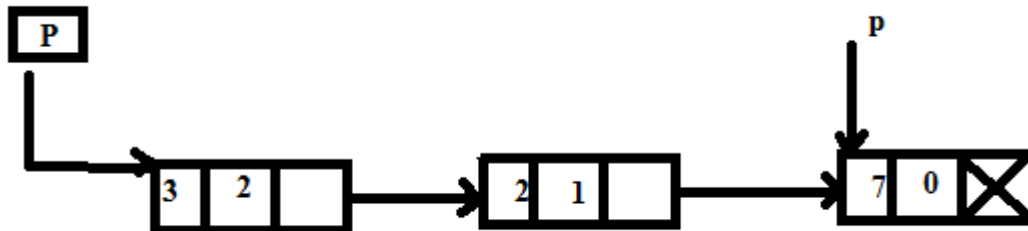


Compare the exponents of the current terms again

$$\text{expo}(p) = \text{expo}(q)$$

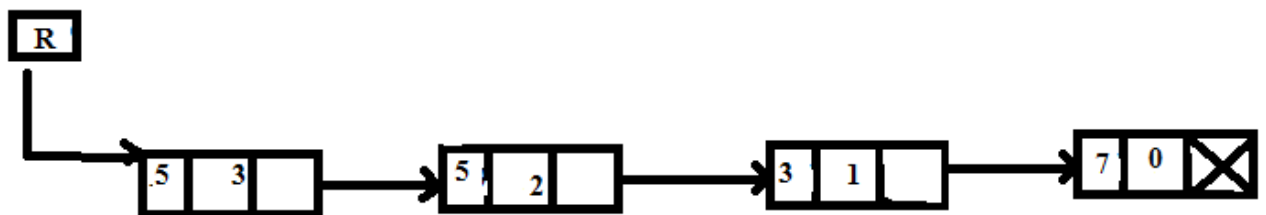
So, add the coefficients of these two terms and link this to the resultant linked list. And, advance the pointers to their next nodes. Q reaches the NULL and p points the last node.

Step 4.



There is no node in the second polynomial to compare with. So, the last node in the first polynomial is added to the end of the resultant linked list.

Step 5. Display the resultant linked list. The resultant linked list is pointed to by the pointer R



Algorithm for Polynomial Multiplication

1. Read the number of terms in the first polynomial
2. Read the coefficient and exponent of the first polynomial
3. Read the number of terms in the second polynomial
4. Read the coefficient and exponent of the second polynomial
5. if one of the list is empty then the nonempty linked list is added to the resultant linked list
Otherwise goto step 6.
6. for each term of the first list
 - a) multiply each term of the second linked list with a term of the first linked list
 - b) add the new term to the resultant polynomial
 - c) reposition the pointer to the starting of the second linked list
 - d) go to the next node
 - e) adds a term to the polynomial in the descending order of the exponent
7. Display the resultant linked list