# CHAPTER 2
# LINK LISTS

## 2.1 INTRODUCTION

Link list is nothing but the collection of numerous nodes. A node consists of INFO (Data) and an address where address of next node is stored .

A node can be shown as in figure 2.1 it consists of two parts

a) INFO part or Data
b) Address part also known as next or LINK that contain address of next node.

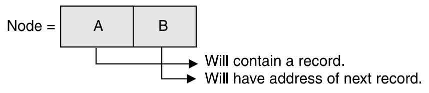


**Figure 2.1**

Let's store the following Data in our computer

2 4 6 8

Till now what we have learnt is that above Data can be stored either as individual variables like a = 2 ; b = 4 etc. or as an array i.e. int a [4]' (i.e. array having 4 integer elements).

While using array the above Data is stored in the following way in the computer memory (figure 2.2)

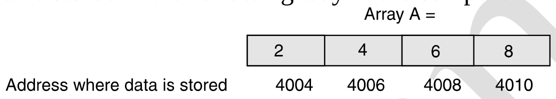


Figure 2.2

As clear from the fig.2.2 in an array the elements are stored in continuous locations in the memory.

**STORING DATA USING LINK LIST**

The above Data can be stored using link list also.

Since there are four Data items to be stored, so we require four nodes, each having its INFO(Data) and the address to next node.

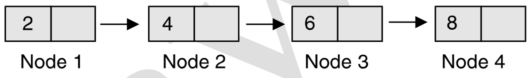


Figure 2.3

The arrows lead us to the LINK node in the link list. Now since node 2 having INFO 4 comes after node1, so there is an arrow after node 1 to point that it's this node which comes after first node and similarly for rest of the arrows. How this arrow or link can be shown? For that we require the address of each node.

In link list, unlike in case of arrays, the nodes can be present anywhere in the memory.

Let the addresses where the above nodes are stored, be as follows

| Node | INFO | Address or LINK |
|---|---|---|
| 1 | 2 | 2025 |
| 2 | 4 | 4039 |
| 3 | 6 | 7132 |
| 4 | 8 | 5535 |

Then the link list will look as shown in Figure 2.4

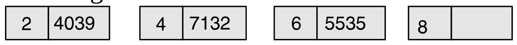


Figure 2.4

A link is created which is responsible for traversing from one node to the other.

Logically the above link list looks as follows.

START = 2025

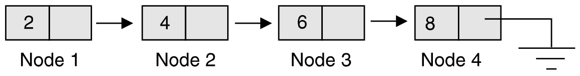


Figure 2.5

START or HEAD represents the starting of the link list. It always points to the first node of the link list, i.e. START or HEAD contains the address of the first node (=2025 in figure 2.5).

You must have noted that the last node in the above diagram is shown grounded, because after last node we don't require to move further (that is no more data is there) so in the address part of the last node we can have any special character to mark the end of the link list. We can use any character like 0 (zero) of '\0' or NULL etc. (anything that can never be a valid address) to mark the end of the link list. In C we use NULL to mark the end of the link list.

**WHY TO USE LINK LISTS?**

A very important question that remained unanswered is "Why to use link lists when the same job can be done using arrays?''

Link lists are required due to the fact that it allows us to dynamically allocate memory.

**Notations used in the Algorithms**

The following notations are used in the algorithms:

1. p = newnode ( ) New node is created with starting address as p. p will point to the new node.
2. p = LINK (x) This means that p will point where LINK of the node x is pointing. See figure 2.6, x represents node 1 and p will points to node 2. i.e. p is pointing where LINK of node x is pointing.

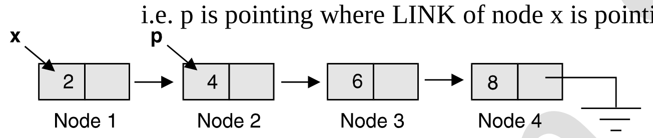


Figure 2.6

3. i = INFO (x) INFO of node x i would be 2 in case in figure 2.6
4. delete (p) To permanently delete a node from a link list.

**2.2 OPERATIONS ON THE LINK LISTS**

There can be many operations that can be performed on a link list.

Following are few commonly used operations.

1. Traversing a Link List.
2. Insertion of a New Node.
3. Deletion of a Node.

**1. Traversing a Link List**

Traversing means to travel. While traversing a link list we travel through all the nodes of the link list that may be required to find a particular INFO, to count the number of records etc.

Traversing is basically to go through each and every node until the end of link list is reached.

**Algorithm 2.1 (To traverse the given Link List.)**

Let's traverse the link list shown in figure 2.6

Algorithm To Traverse the given Link List

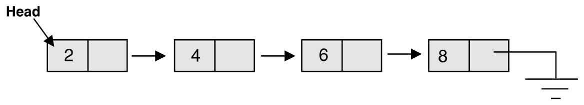
```
Algorithm Trav(START) //START points to the first node.
        {
              p= START
              while(p!=NULL)
              {
                    write (INFO(p));
                    p=LINK(p);
              }
        }
```

By replacing **P by PTR** , **START by START** ,**INFO by INFO** and **LINK by LINK**.

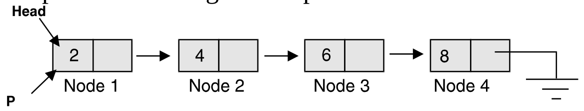
We get another form of algorithm shown in figure below.

```
Algorithm Trave(START)
{
      PTR = START
      While (PTR! = NULL)
      {
            Write (INFO[PTR]);
            PTR = LINK [PTR] ;
            }
    }
```

**Explanation of Algorithm 2.1**



**Step 1.** The above link list is passed to the algorithm. p = START or HEAD

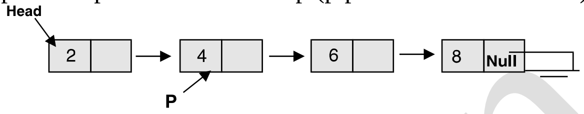


p will now point where START or HEAD is pointing.

**Step 2**. p ! = null so the we enter into the while loop.

**Step 3**. Write (INFO(p)) **Output 2**

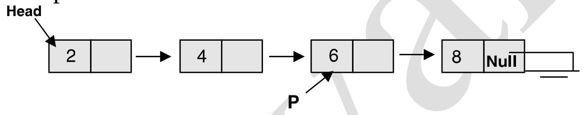
**Step 4.** p = LINK (p) the p would point to LINK of p (p points to its next node) as shown.



**Step 5.** While loop continuous. Still p ! = null

**Step 6.** Write (INFO(p) **Output 4**

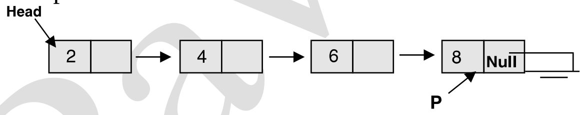
**Step 7**. p = LINK (p) And the p would transfer to LINK node as shown.



**Step 8.** Again while loop continuous. p ! = null, but is pointing to node 3 as shown.

**Step 9**. Write (INFO(p)) **Output 6**

**Step 10.** p = LINK (p) And the p would transfer to LINK node i.e. node 4.



**Step 11.** Write (INFO(p)) **Output 8**

**Step 12.** p= LINK (p) which is equal to NULL. So p becomes NULL.

**Step 13.** Since p becomes NULL, condition is not satisfied and while loop is terminates But before that all the nodes have been traversed.

***Important:*** *START is still at node 1*
And we get the full output as
2 4 6 8

## 2.3 INSERTION OF THE NEW NODE

Insertion of node means to add a node in the link list. Addition of node in a link list can be done at any of the following positions in the list.

a) At the beginning.
b) At the end.
c) In between two nodes.

There are three cases because insertion in a link list requires different actions with different positions.

**a) To add a Node at the Beginning**

**Algorithm 2.2 (To insert a Node at the beginning of a Link List)**

To add a node at the beginning we require to change the START or HEAD to the new node created. The algorithm to insert the node at beginning is

Algorithm for insertion of a node at beginning of Link List.

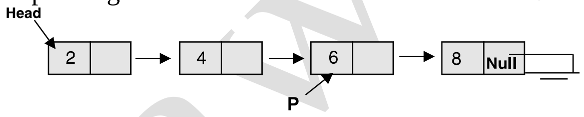
```
Algorithm insert_at_Begin (d.START)
        {
                p=newnode();
                INFO(p)=d ;
                LINK(p) = START ;
                START = p ;
        }
```

By replacing
P by NEW ,newnode( ) by AVAIL,INFO by INFO,START by START and d by ITEM
we get another form of the same algorithm shown below.

```
Algorithm Insert_at_Begin(ITEM,START)
{
    NEW = AVAIL
    INFO[NEW] = ITEM
    LINK [NEW] = START
    START = NEW
}
```

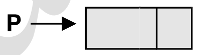
**Explanation of Algorithm 2.2**

Consider a link list with START pointing to first node that looks like as follows :

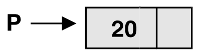


Let INFO 20 needs to be added in the link list i.e. d=20

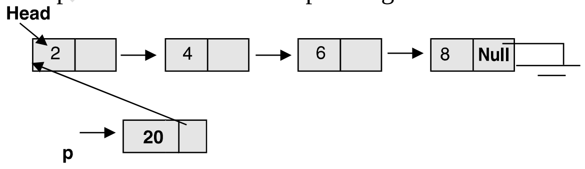
**Step 1**. p= newnode () i.e. A new node will be created and node will have the address in p.



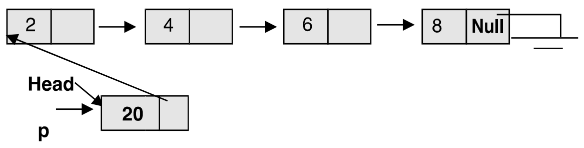
**Step 2.** INFO (p) = d. In the INFO part of the node d is stored.



**Step 3.** LINK (p) = START. That means, whatever be the START, that is stored in the address part of the node p means LINK of P will point where START is pointing. And the link list looks like as follows.



**Step 4.** START = p ; Now the value of START is changed. Which is equal to p.i.e. START will now point where P is pointing



When START becomes equal to p, the link list will start from the node with INFO 20 that has just been added. And hence the node is actually added to the link list at the beginning.

**b) To add a Node at the End**

**Algorithm 2.3 (To insert a Node in a Link List at the end of the Link List)**

To add a node at the end of the link list requires us to travel to the last node and put a new node address in its address part. And new node created must have terminator that is NULL in this case.
The algorithm to insert the node at end is.

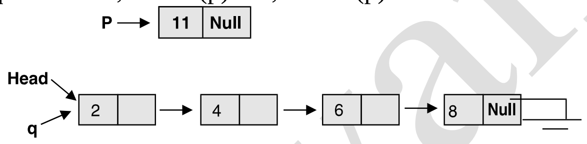
Algorithm for insertion of a node at end of Link List.

```
Algorithm Insert_at_end (d.START)
        {
                q=START ;
                p= newnode();
                INFO (p)=d;
                LINK (p) = NULL ;
                while (LINK (q)!=NULL)
                q = LINK (q);
                LINK (q) = p ;
        }
```

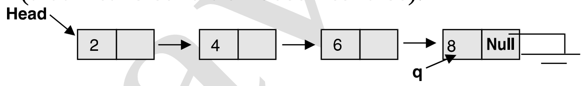
**Explanation of Algorithm 2.3**

Let a node with INFO 11 require to be entered in a link list starting with START. Let us take the same link list which we took in algorithm 2.2

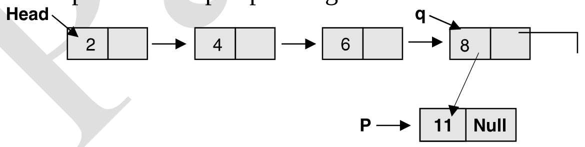
**Step 1.** The following instruction in algorithm will give us a following link list.
p = newnode () q = START, INFO (p) = d, LINK (p) = NULL



**Step 2.** The condition while (LINK (q) ! = NULL ) will be executed till the last of the node is not reached. That is after the execution of this, q will be pointing to the last node because last node is the condition having LINK (q) = NULL (that means condition becomes false).



**Step 3.** Now the only thing required to be done is : LINK (q) = p And after which the link list changes to i.e. LINK of q will now point where p is pointing



Hence the link list is modified with the required addition at the end.

**C) To insert a Node in between Two Nodes.**

**Algorithm 2.4 (To insert a Node in between a Link List)**

In this algorithm either of the following conditions should be known to us before insertion.

1. Position where node is to be inserted or
2. After which INFO the node is to be inserted.

If either of the above condition is known to us then only we can insert the node at that particular position.

The following algorithm is for inserting node after a particular INFO. START is the start of the link list. d is the INFO to be added in new node and n is INFO of the node after which node is to be inserted.

Algorithm To insert a Node in between a Link List.

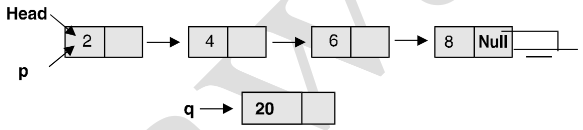
```
Algorithm Insert_at_mid (START, d, n)
        {
              p=START ;
              q = newnode() ;
              INFO(q) = d;
              while (INFO (p) ! = n )
              p= LINK (p) ;
              LINK (q) = LINK (p) ;
              LINK (p) = q ;
         }
```

**Explanation of Algorithm 2.4**

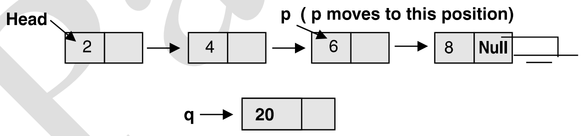
Let's take the same example as that of algorithm 2.2

START is start, d = 20 and n = 6. That means new node with INFO 20 needs to be created and inserted after node having INFO 6.

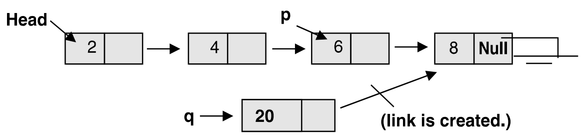
**Step 1.** After executing the instructions p = START and q = newnode () INFO (q) = 20, we will have the following link list and variables.



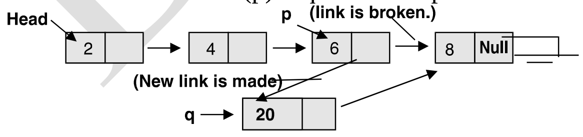
**Step 2.** It includes the while loop till the INFO (q) ! = n.i.e. 6. The Link list at this stage looks as follows with the variables at the position shown.



**Step 3.** Now the important step is to create a link of q with the main link list ; LINK (q) = LINK (p). Since the LINK of p contains the address of node where 8 is stored, hence that address would be stored in the address part of q.



**Step 4.** Once the link with main link list is created LINK step would be to put this created node in the main list . i.e. LINK(p) = q. This will put the value of q (i.e.address of new node ) in address part of p.



The above link list shows the insertion of node with INFO 20 after node with INFO 6. And hence the new link list will be having the following sequences.

**2 ,4 ,6 ,20 ,8**

## 2.4 DELETION OF NODE

Deletion of node means just to delete the node from the link list. First thing is to determine the position of the node in the link list. Once the position is known, the only thing required to be done is rearrangement of the links.

Deleting node from the link list can be done at any of the following positions in the list :

a) First Node.
b) Last Node.
c) Any other Node.

The above three can basically work on the same algorithm except for the first one where START is required to be changed. So a condition related to the change of the START can be included in the algorithm.

**Algorithm 2.5 (To Delete a Node.)**

The following algorithm is for deleting a node. Let START be the beginning of the link list and the node to be deleted should have the record d.

Algorithm for deleting a node from Link List

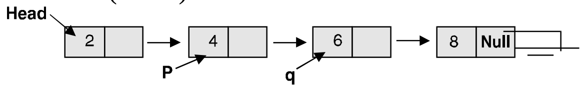
```
Algorithm (START. d)
    {
            q= START :
            if (INFO (q) = =d )    (This means first node to be deleted)
            {
                    START = LINK(START) ;
                    delete (q) ;
                    return ;
            }
            while(INFO (q) ! = d)
            {
                    p=q ;      ( p will have the address of previous node)
                    q= LINK (q) ;
            }
            LINK (p) = LINK (q) ;
            delete(q) ;
    }
```

**Explanation of the Algorithm 2.5**

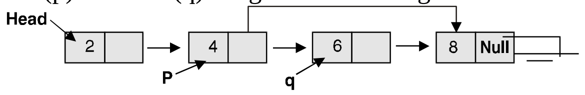
Let's take the same example of algorithm 2.3 and d = 6 i.e. node with INFO 6 is to be deleted.

**Step 1 .** The first thing to be checked is that whether the first node is to be deleted. If first node is to be deleted, than START is required to be moved to the LINK node. After moving the START delete the first node and the algorithm terminates there itself.

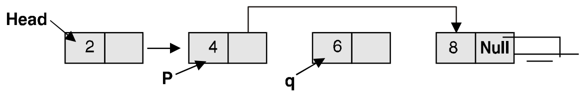
**Step 2.** If the node to be deleted is not the first one, then what is required to be done is that we should keep track of the previous node, which has been denoted by variable p. As p is equated with q, q moves to the LINK node by statement ; q = LINK (q) ; Now, if we have the same link list used in algorithm 2.3 then after the step 2 our link list looks like : (d = 6 )



**Step 3.** The LINK step involves creating of link between the previous node (p) and the node LINK to q. And after the instruction LINK (p) = LINK (q) we get the following list.



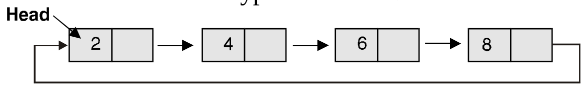
**Step 4.** After the statement delete (q). we have deleted the node pointed by q as shown below.



**2.5 CIRCULAR LINK LIST**

In singly and doubly link list the traversing comes to a halt at **NULL** i.e. at the end or beginning of the link list. There can be another variation in link lists, where a list does not end i.e. circular link list.

A circular link list is one in which last node has the address of the first node. And while traversing one doesn't reach the end of the link list as there is no. **NULL** character in this type of link list.



**Advantage of Circular Link List**

Some times the operations are such that one has to move immediately to the first node from the last node. In that case circular link lists provide us the continuity from last to first node. Whereas in case of doubly or singly link lists one has to refer to START to get to the first node.

**Disadvantage of Circular Link List**

The only disadvantage is that sometimes one may loose track and may not be able to find out where the list begins and where its ends.

**1.Algorithm To Traverse the given Circular Link List**

```
Algorithm Trav(START) /START points to the first node.
        {
              p= START
              do
              {
                    write (INFO(p));
                    p=LINK(p);
              } while(p!=START);
        }
```

**2. Algorithm for insertion of a node at beginning of Circular Link List.**

```
Algorithm insert_at_Begin (d.START)
        {     q=START;
              p=newnode();
              INFO(p)=d ;
              while(LINK(q)!=START)
                q=LINK(q);
              LINK(q)=p;
              LINK(p) = START ;
              START = p ;
        }
```

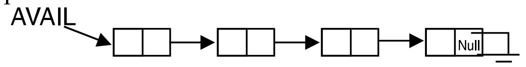
**3. Algorithm for insertion of a node at end of Circular Link List.**

```
Algorithm insert_at_Begin (d.START)
        {     q=START;
              p=newnode();
              INFO(p)=d ;
              while(LINK(q)!=START)
                q=LINK(q);
              LINK(q)=p;
              LINK(p) = START ;

        }
```

**2.6 NEWNODE AND DELETENODE OPERATIONS**
**p=newnode();**
**delete(q);**

To understand newnode and deletenode properly we should understand the pool of available nodes. Think that a huge pool of INFO is available to us. It is represented as a second linked list. Let's call that list by a pointer AVAIL.



Whenever we need a INFO space we can have it from the AVAIL. To get a newnode in a link list, first a node from the AVAIL is deleted and algorithm will look follows:
**p=newnode();**

If AVAIL not equal to zero newnode = AVAIL
AVAIL = LINK (AVAIL)

Algorithm for p=newnode() operation**;**

```
Algorithm newnode()
{
if (AVAIL!=0)
{
 p = AVAIL;
AVAIL = LINK (AVAIL);
return p:
}
else
write("memory not available");
}
```

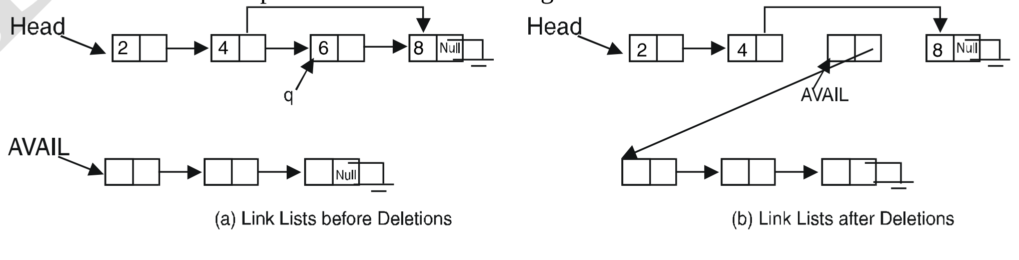
**delete(q);**

In case of deletenode the node that is to be deleted is just added in the list AVAIL.
LINK(oldnode) = AVAIL
AVAIL = oldnode

Algorithm for delete(q) operation**;**

```
    Algorithm delete()
    {
    LINK(q) = AVAIL;
    AVAIL = q;
}
```

Lets take the example of algorithm where q is to be deleted. The q is just passed to AVAIL. Both the link lists i.e. INFO and available link list before and after operations are shown in diagram.


Head
2
4
6
8
Null
q
AVAIL
Null
(a) Link Lists before Deletions
Head
2
4
8
Null
AVAIL
(b) Link Lists after Deletions

As seen from the figure (b) the deleted node from INFO is again added to the available nodes list. In this way, whenever node is added to the INFO link list, it's just taken from AVAIL list and whenever a node is deleted, it just gets back into AVAIL

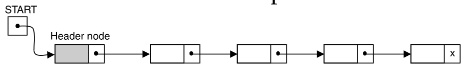
Application of Link List: Link lists can be used in any algorithms where there are collections of INFO. All the problems where array is used can be implemented using link list. But link list is really advantageous to use in Stacks, Queues, Graphs, Trees etc.

## 2.7 HEADER LINKED LISTS

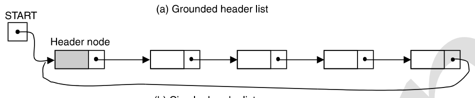
A header linked list is a linked list which always contains a special node, called the header node at the beginning of the list. The following are two kinds of widely used header lists:

(1) A **grounded header** list is a header list where the last node contains the null pointer. (The term "grounded" comes from the fact that many texts use the electrical ground symbol to indicate the null pointer.)

(2) A circular header list is header list where the last node points back to the header node.

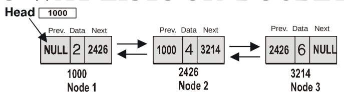


(a) Grounded header list



(b) Circular header list

## 2.8 TWO-WAY LISTS OR DOUBLY LINK LIST



In Doubly Link List each node contains three parts.
1. prev.
2. INFO
3. LINK
INFO part and two pointer part.
**prev**. part contains the address of previous node.
**LINK** part contains the address of LINK node
and **INFO** part contains the information.

for e.g. in above link list Node 1. address is 1000
Node 2 address 2426 and Nodes address is 3214

Node 1 previous part does note point to any node LINK of node 1 point to node 2 (i.e. contains address of node 2 which is 2426 node 2 previous points to node 1 and node 2 LINK points to node 2.

(i.e. node 2 previous points to address of node 1 which is 1000 and LINK of node 2 points to address of node 3 which is 3214.

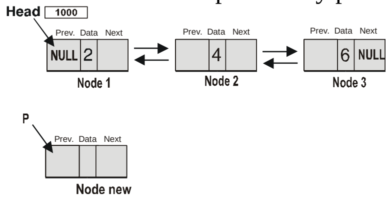
| Algorithm for insertion of a node at beginning at Doubly Link List. |
|---|

```
insert_at_beg( START, d)
   {
      p = newnode ( ) ;
      INFO (p) =d ;
      LINK (p) = NULL ;
      prev.(p) = NULL ;
      LINK (p) = START ;
      prev. (START) = p ;
      START = p ;
   }
```

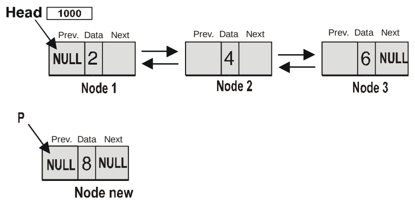
**Explanation of Algorithm** :

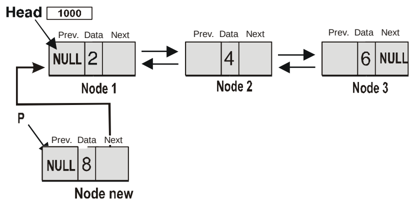
Here d is INFO part of newnode

p = newnode ( ) ;
newnode is created pointed by p

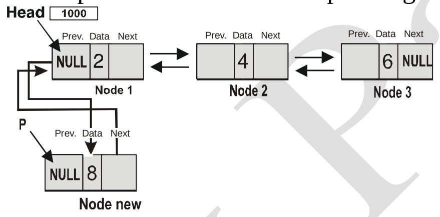


INFO (p) =d ;
LINK (p) = prev.(p) = NULL ;
Value of d is assign to INFO part let value of d is 8 so INFO (p) = 8;





LINK (p) = START
LINK of P will point where START is pointing to node 1 so LINK of P will Point to node 1

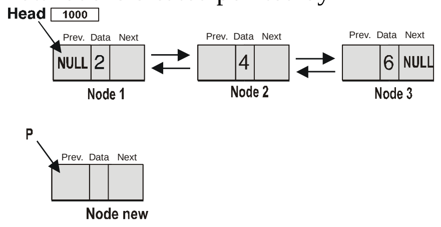


prev. (START) = p
prev. of START will point where P is pointing
START = P
now we know that START will always point to the starting node as node new is starting so by making START point where P is pointing START = P

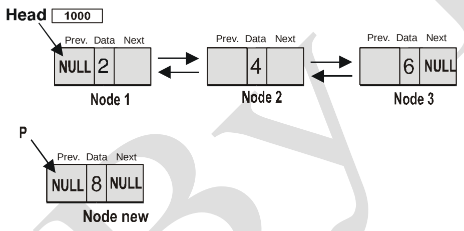
Algorithm for insertion of a node at end of Doubly Link List.

```
insert_at_end ( START, d)
    {
       q = START ;
       p = newnode ( ) ;
       INFO (p) =d ;
       LINK (p) = NULL ;
       prev.(p) = NULL ;
       while (LINK(q) ! = NULL )
             {
              q = LINK (q) ;
             }
       LINK (q) = p ;
       prev.(p) = q ;
    }
```

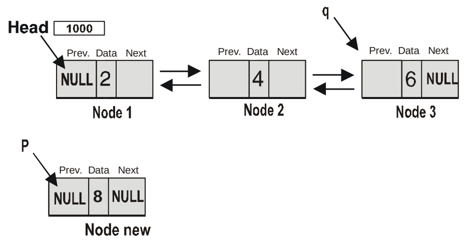
**Explanation of Algorithm** :
Here d is INFO part of newnode
p = newnode ( ) ;
newnode is created pointed by P



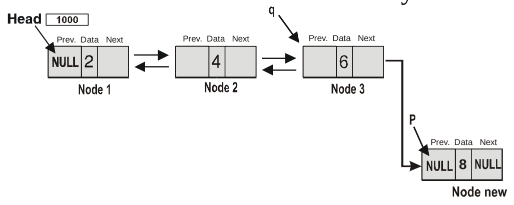
INFO (p) =d ;
LINK (p) = NULL ;
prev.(p) = NULL ;
Value of d is assign to INFO part let value of d is 8 so INFO (p) = 8;
LINK of p and prev. of p becomes NULL



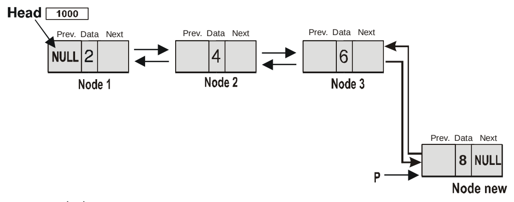
while (LINK(q) ! = NULL)              {
q = LINK (q);
}
by this statement q will point to last of node in Doubly link list.



Insertion of node at the end of Doubly Link List.



LINK (q) = P

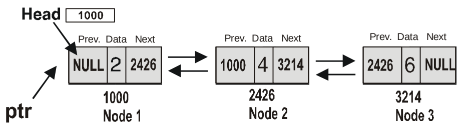


prev.(p) = q

Algorithm for insertion of a node in middle of Doubly Link List.

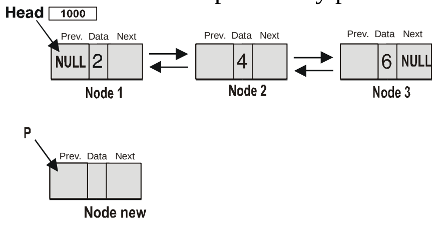
```
insert_in_mid( START , d , n)
    {
       ptr = START ;
       p = newnode ( ) ;
       INFO (p) =d ;
       LINK (p) = NULL ;
       prev.(p) = NULL ;
         while (INFO (ptr) ! = n)
              {
               ptr = LINK (ptr) ;
              }
       prev. (p) = ptr ;
       LINK (p) = LINK (ptr) ;
       prev. (LINK (ptr)) = p ;
       LINK (ptr ) = p ;
    }
```

**Explanation of Algorithm** :
Here d is INFO part of newnode and n is INFO of the node after which newnode is to be inserted
ptr = START
ptr will point where START is pointing.



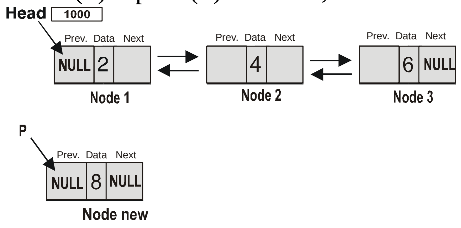
p = newnode ( ) ;

newnode is created pointed by p



INFO (P) =d ;

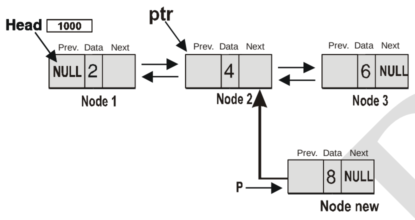
LINK (P) = prev.(P) = NULL;



while (INFO(ptr)!=n)

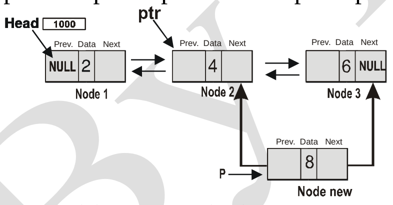
ptr = LINK (ptr) ;

by this statement ptr will point to the node after which new node is to be inserted.

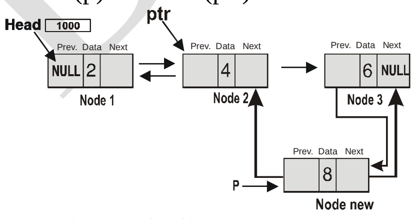


prev.(p) = ptr

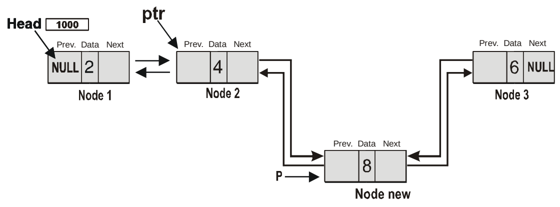
prev. of p will point where ptr is pointing to node 2 So prev. of p will point to node 2.



LINK (p) = LINK (ptr)



prev. (LINK (ptr)) = P



LINK (ptr) = p ;

**2.9 COMPARISION BETWEEN ARRAYS AND LINK LISTS**

| | **Arrays** | **Link Lists** |
|---|---|---|
| 1. | In arrays the memory allocation is contiguous. | Link list uses non-contiguous memory allocation. |
| 2. | The size of the array has to be declared before compilation. | The size can be declared dynamically i.e. during run time. |
| 3. | Direct access of any element of an array is possible by using its subscript. | Direct access of an element is not possible. The only access is while moving sequential. |
| 4. | Arrays may lead to wastage of space if size declared in more than that of what is required. | In this case there is not size declaration and hence the no wastage. |
| 5. | Apart from INFO, no extra space is required for pointers etc. | Extra space is required to maintain the address of LINK node apart form INFO. |
| 6. | Arrays are generally faster to access. | Link list are comparatively slower as compared to arrays. |

**DYNAMIC MEMORY ALLOCATION**

Allocating a Block of Memory: A block of memory may be allocated using the function 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. It takes the following form:

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

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

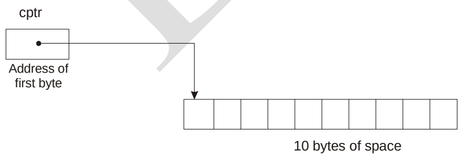
Example : X =(int*)malloc(100*sizeof(int));

On successful execution of this statement, a memory space equivalent to “100 times the size of an **int**” bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer x of type of **int.**

Similarly, the statement

cptr=(char*)malloc(10);

Allocates 10 bytes of space for the pointer **cptr** of type **char.** This is illustrated below:



Note that the storage space allocated dynamically has no name and therefore its contents can be accessed only through a pointer.

We may also use malloc to allocate spece for complex INFO types such as structures. Example:

```
st_var=(struct store*)malloc(sizeof(struct store));
```

where **st_var** is a pointer of type **struct store.**

Link list contain two parts INFO part and address or LINK part, so we use structure to represent link list node.

```
struct NODE
{
      int INFO ;
      struct NODE *LINK ;
} ;
```

here INFO part is of integer type and LINK part contain address of another node so it is a pointer of struct NODE type.

```
struct NODE
{
        int INFO ;
        struct NODE *LINK ;
} ;
typedef struct NODE node ;
node * START ;
START = (node*) malloc(size of (node));
```

using typedef we can represent existing INFO type to define an identifies for example by using

```
typedef    struct NODE       node ;
```

we can use **node** in place of **struct NODE**

node *START ;

now we declare a pointer START of node or (struct NODE) type.

In algorithm we have use

p=newnode( );

to create a new node which is pointed by p;

in C language we use malloc function to create newnode.

So by using statement

p=(node*)malloc(size of (node));

we create new node now in structure the member INFO and LINK can be access by pointer p using -> (arrow) operator. In our algorithm we use INFO(p) to represent INFO part Pointed by p in C

we use p-> INFO to represent INFO (p)

like wise in our algorithm we use LINK(p) to represent LINK part we use

p -> LINK to represent LINK (p)

Now we can convert our algorithm into C program using following notation

| Algorithm | C program |
|---|---|
| p = newnode( ); | p=(node*)malloc(size of (node)); |
| INFO (p) ; | p -> INFO ; |
| LINK (p) ; | p -> LINK ; |

Now we can convert algorithm into c program

**C PROGRAME FOR OPERATION ON SINGLY LINK LIST:**

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
 struct NODE
 {
             int INFO;
             struct NODE * LINK;
 };
 typedef    NODE   node;
 node *START=NULL;
 void main()
 {
```

```
            void create();
            void traverse();
            void reverse();
            void insert_at_beg(int);
            void insert_at_end(int);
            void insert_at_mid(int,int);
            int n=0,d,x;
while(n!=7)
{
            printf("\nEnter 1 for Creation   of Link List\n");
            printf("\nEnter 2 for Traversing of Link List\n");
            printf("\nEnter 3 for Insert node at Beg of Link List\n");
            printf("\nEnter 4 for Insert node at End of Link List\n");
            printf("\nEnter 5 for Insert node at Mid of Link List\n");
            printf("\nEnter 6 for Reverse of Link List\n");
            printf("\nEnter 7 for Exist\n");
            scanf("%d",&n);
switch(n)
            {
             case 1: create();
                         break;

             case 2:  traverse();
                          break;
             case 3:  printf("\nEnter INFO of newnode to insert\n");
                          scanf("%d",&d);
                          insert_at_beg(d); break;
             case 4:  printf("\nEnter INFO of newnode to insert\n");
                          scanf("%d",&d);
                          insert_at_end(d); break;
             case 5:  printf("\nEnter INFO of newnode to insert\n");
                          scanf("%d",&d); break;
                          printf("\nEnter INFO of node after which newnode is to be insert\n");
                          scanf("%d",&x);
                          insert_at_mid(x,d);
                          break;

            case 6: reverse();
                        break;

             }

}
}

void create()
{
 node *temp,*p;
 char choice='y';
 while(choice=='y'||choice=='Y')
           {
           p=(node *)malloc(sizeof(node));
           printf("\nEnter INFO part of new node\n");
           scanf("%d",&p->INFO);
```

```
            p->LINK=NULL;
 if(START==NULL)
              {
              START=temp=p;
              }
 else
            {
              temp->LINK=p;
              temp=temp->LINK;
             }
              printf("\npress y to add more nodes\n");
              fflush(stdin);
              scanf("%c",&choice);
 }
 }
 void traverse()
 {
 node *p;
 p=START;
 while(p!=NULL)
            {
             printf(" %d ",p->INFO);
             p=p->LINK;
             }

 }

void insert_at_beg(int d)
 {
              node *p;
              p=(node *)malloc(sizeof(node));
              p->INFO=d;
              p->LINK=START;
              START=p;
 }
void insert_at_end(int d)
 {
             node *p,*q;
             q=START;
             p=(node *)malloc(sizeof(node));
             p->INFO=d;
             p->LINK=NULL;
             while(q->LINK!=NULL)
                        q=q->LINK;
             q->LINK=p;
  }

void insert_at_mid(int n,int d)
 {
              node *p,*q;
              p=START;
              q=(node *)malloc(sizeof(node));
              q->INFO=d;
              while(p->INFO!=n)
```

```
                    p=p->LINK;
            q->LINK=p->LINK;
            p->LINK=q;
  }
void reverse()
{

          node *prev,*curr,*p;
          curr=p=START;
          prev=NULL;
          while(curr!=NULL)
          {
                    p=p->LINK;
                    curr->LINK=prev;
                    prev=curr;
                    curr=p;

          }
           START=prev;

}
```

**POLYNOMIAL REPRESENTATION**

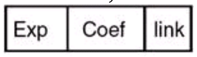
One of the problems that a linked list can deal with is manipulation of symbolic polynomials. By symbolic, we mean that a polynomial is viewed as a list of coefficients and exponents. For example, the polynomial

$3x^2+2x+4$,

can be viewed as list of the following pairs

(3,2),(2,1),(4,0)

Therefore, we can use a linked list in which each node will have three fields, as shown in Figure below



A polynomial $10x^4 + 5x^2 + 1$ can be represented as shown here:

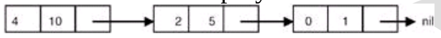


Figure Polynomial representation.

The procedure to add these two polynomials using the linked list is in the following algorithm

Two input link list are pointed by START1 and START2 respectively ,the third link list pointed by START3 will contain addition of these two link list

Assuming insert_at_end(c,e) function create newnode with coef=c and exp=e  and add this node at end of final link list pointed by START3

**<u>Algorithm for addition of two polynomial</u>**

```
Algorithm polyadd()
{
int c,e;
p=START1;
q=START2;
while(p!=NULL and q!=NULL)
    {
            if(exp(p)==exp(q))
            {
            c=coef(p)+coef(q);
            e=exp(p);
            insert_at_end(c,e);
            p=LINK(p);
            q=LINK(q);
            }
            else if(exp(p)<exp(q))
           {
            c= coef(q);
            e=exp(q);
            insert_at_end(c,e);
            q=LINK(q);
            }
            else{
            c=coef(p);
            e=exp(p);
            insert_at_end(c,e);
            p=LINK(p);
            }
}
while(p!=NULL)
{           c=coef(p);
            e=exp(p);
            insert_at_end(c,e);
            p=LINK(p);
}
while(q!=NULL)
  {         c=coef(q);
            e=exp(q);
            insert_at_end(c,e);
            q=LINK(q);
  }

}
```