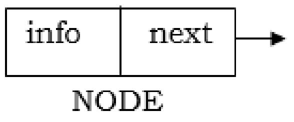
Unit- II Topic –II LINKED LISTS

Introduction:

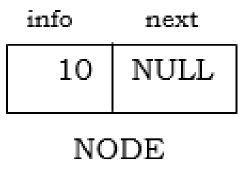
In previous chapter we implement stack and queue by using array (static implementation). One major problem with the array implementation is that, size of an array must be specified at the declaration of array therefore it cannot grow or shrink at run time of program. Also there is problem of memory wastage. (for array, memory is get allocated at compile time) .
To overcome the drawback of static implementation, the concept of linked list (Dynamic data structure) is introduced.

Linked List:

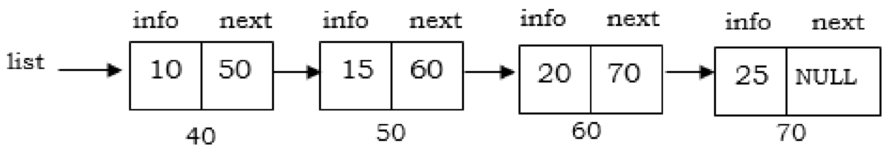
Linked list is linear data structure, which is a collection of nodes and each node contains two parts namely I) info II) next
Here, *info* part is actual data or element
next part is node pointer which holds the address of next node.
Following fig. shows the single node:



e.g.



In above example, the field *next* is node pointer, which stores the address of next node. If there is no next node then it contains the NULL.
Also, note that every linked list has one external node pointer 'list' which always holds the address of first node and due to this 'list' pointer it is possible to read or traverse the entire linked list.
Following fig. shows the linke d list:



In above figure '*list '* is an external pointer which always holds address of first node and is useful to traverse the entire linked list .
The NULL pointer in linked list state that three things as fallow:

1) End of linked list
2) Last node in the linked
3) Empty pointer field

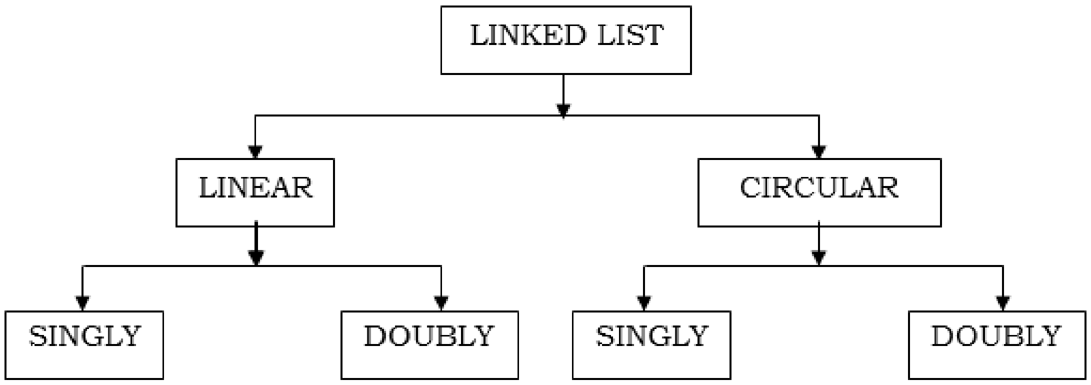
Note: The linked list with no nodes is called "NULL or empty linked list".

Advantages of Linked list: (Difference between Array and linked list)

1) We know that linked list is dynamic data structure. Therefore, first advantage of linkedlist over arrays is that linked list can grow or shrink during execution of program.
2) Element insertion process is easy in linked list as compared to array.
3) There is no memory wastage problem in linked list (because it is dynamic in nature) but array has memory wastage problem.
4) The linked list is more flexible than array, stack and queue i.e. we can insert or remove nodes in linked list at any position.

Types of linked list:

Basically linked list can be categorized into two main categories viz. Linear Linked list and Circular Linked list. Further these two linked list have two sub types and therefore there are total four types of linked list.
Following tree diagram shows the types of linked list:

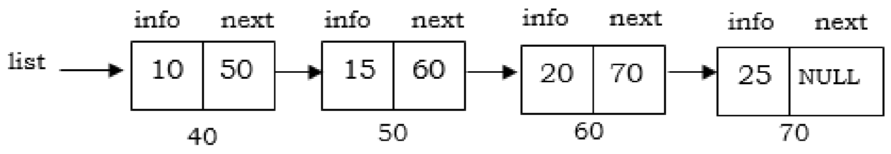


1) Linear Linked List:

The linked list which is collection of nodes and that contains NULL pointer i.e. there is end of linked list such a linked list called "Linear Linked List".
Also Linear linked list have two types as fallow:

I) Singly (Single) Linear Linked list:

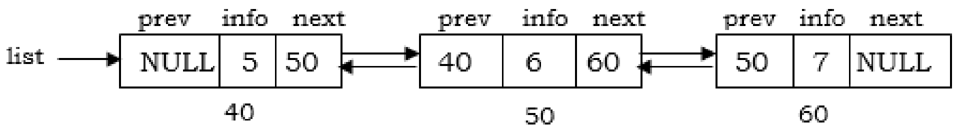
The node in the linked list contains only one (Single) node pointer is called "Singly or Single linked list". Also *info* part of node is compulsory, that is node of singly linked list contains two parts viz *info* and *next.*
Following fig. shows the singly linear linked list:



The above linked list is linear linked list because it contains NULL pointer and it is singly because node in linked list contains single node pointer '*next'* therefore above fig. is the singly linear linked list.

II) Doubly (Double) Linear Linked list :

The node in the linked list contains two (Double) node pointers is called "Doubly or double linked list". Also *info* part of node is compulsory, that is node of doubly linked list contains three parts viz *prev, info* and *next.*
Here, *prev* is node pointer which holds address of previous node.
info is actual element or data
next is node pointer which holds address of next node.
Following fig. shows the doubly linear linked list:



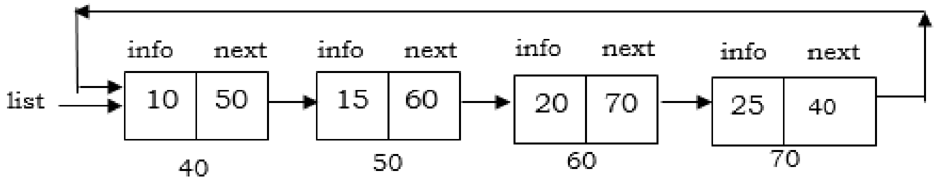
The above linked list is linear linked list because it contains NULL pointer and it is doubly because node in linked list contains double node pointers *prev* and *next* therefore above fig. is the doubly linear linked list.

2) Circular Linked List:

The linked list which is collection of nodes but that **does not** contains NULL pointer and this NULL pointer is replaced by address of first node i.e. there is **not** end of linked list such a linked list called "Circular Linked List".
Also Circular linked list has two types as fallow:

I) Singly (Single) Circular Linked list:

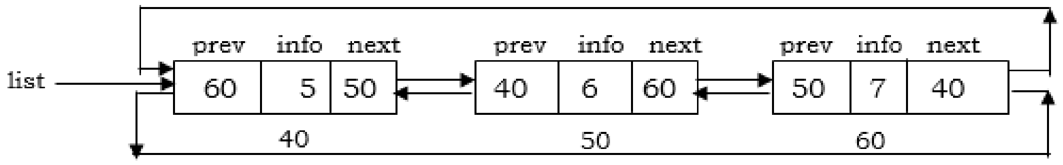
The node in this linked list contains only one (Single) node pointer (next) and such linked list does not contains NULL pointer is called "Singly circular linked list". Also *info* part in node is compulsory that is node of singly circular linked list contains two parts viz *info* and *next.*
Following fig. shows the singly circular linked list:



The above linked list is circular linked list because it does not contains NULL pointer this NULL pointer is replaced by address of first node and it is singly becaus e node in linked list contains single node pointer *next* therefore above fig. is the singly circular linked list.

II) Doubly (Double) Circular Linked list:

The node in this linked list contains two (Double) node pointers (next & prev) and such linked list does not contains NULL pointer is called "Doubly circular linked list". Also *info* part in node is compulsory that is node of doubly circular linked list contains three parts viz *info, next* and *prev.*
Here, *prev* is node pointer which holds address of prev ious node.
info is actual element or data
next is node pointer which holds address of next node.
Following fig. shows the doubly circular linked list:



The above linked list is circular linked list because it **does not** contains NULL pointer a nd it is doubly because node in linked list contains double node pointers *prev* and *next* therefore above fig. is the doubly circular linked list.

Let's see all these linked list types in details;

1) Singly Linear Linked list:

To implement singly linear li nked list we have to define one class which is as fallow:

```
class          node
{
        int       info;
        node    *next;
        public:
        // declaration of operations of SLLL.
} *list;
```

Operations performed on to the singly linear linked list:

There are some following operations that can be operate on to the linked list:

1) Create operation:

This operation creates new node dynami cally and returns node pointer towards called function.

Operation:

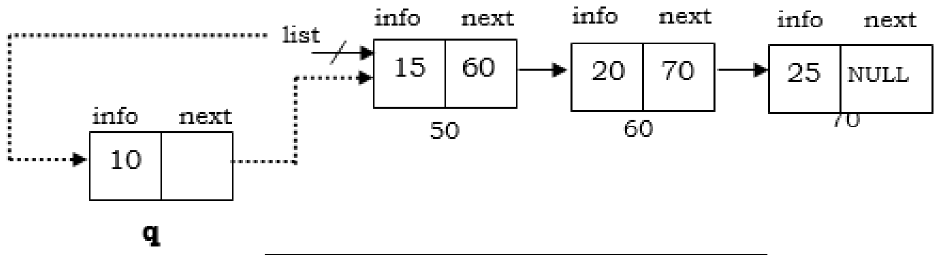
```
node*    create( )
{
        node    *z;
        z = new  node;
        return(z);
}
```

2) Insert Operation:

This operation inserts new node into linked list. Also there are three possible cases to insert new node into linked list:

I) Inserting node at the beginning of linked list:

This operation inserts new node a t the beginning of linked list, which is shown in fallowing figure:



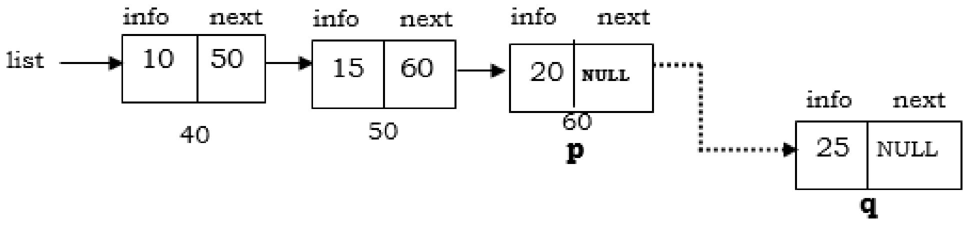
```
q = create ( );
q →info = x ;        // x is integer
q →next = p;
list = q ;
```

Operation:

```
void     ins_beg ( int x )
{
        node    *p, *q ;
        p = list;
        if ( p = = NULL )           // if linked list is empty
        {
                p = create( );        //   newnode  created
                p →info = x;
                p →next = NULL;
                list = p;
        }
        else
        {
                q = create( ) ;     //   newnode created
                q→info = x ;
                q → next = p;
                list = q ;
        }
}
```

II) Inserting node at the end of linked list:

This operation inserts new node at the end of linked list. For that we have to move first node 'p' in forward direction and set 'p' as last node, after setting 'p' node as last node then insert new node 'q' after node 'p' which is last node of linked list, is shown in fallowin g figure:



```
q = create( ) ;
q→info = x ;        // x is integer
q→next = NULL;
p→next = q ;
```

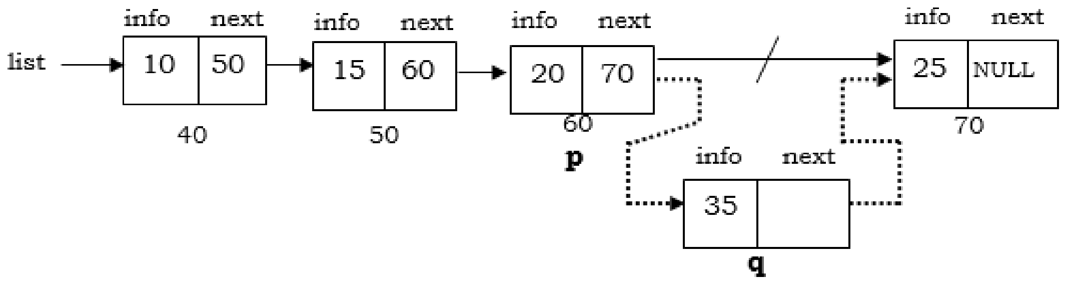
Operation:

```
void    ins_end ( int  x )
{
        node    *p, *q ;
        p = list;
        if ( p = = NULL )          // if linked list is empty
        {
                p = create( );       //  newnode created
                p→ info = x;
                p→ next = NULL;
                list = p;
        }
        else
        {       //  move from first node to last node and set 'p' as last node
                while ( p→next != NULL )
                {
                        p = p→next;
                }
                q = create( ) ;     //  insert new node 'q' after 'p' which is last node.
                q→ info = x ;
                q→next = NULL;
                p→next  = q ;
        }

}
```

III) Inserting node in between two nodes o f linked list: (insert after)

This operation inserts new node which is in between two nodes of linked list.

For that we have to move from first node towards the last node, while doing so, check ' *info* ' part of node 'p' is matches with ' *after* ' or not. If ' *after* ' is matches with ' *info* ' part then insert new node 'q' which is in between two nodes of linked list, is shown in fallowing figure:



```
q = create( ) ;
q→ info = x ;      // x is integer
q→next = p→next ;
p→next = q ;
```

Operation:

```
void    ins_bet ( int    after, int     x )
{
        node    *p, *q ;
        p = list;

        if ( p = = NULL || p → next = = NULL )   /* if linked list is empty or
                                                      Linked list having one node */
        {
                cout<<“ Insert between is not possible ”;
        }

        else
        {       //  move from first node to last node
                while ( p→ next != NULL )
                {
                        if ( after = = p→info)    // info part matches with after then
                        {
                           q = create( ) ;
                           q→ info = x ;
                           q→ next = p → next;
                           p →next = q ;
                        }
                        p = p → next;
                }

        }

}
```

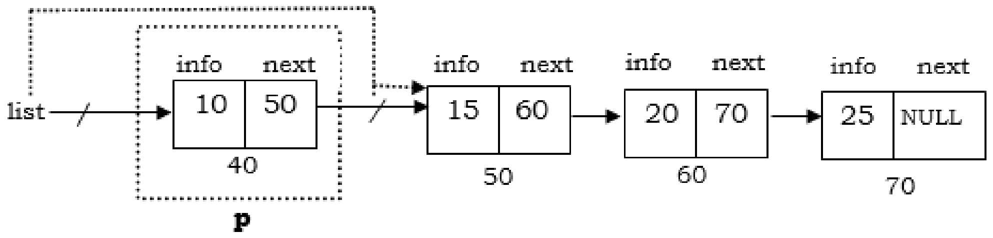
3) Remove Operation:

This operation removes the node from linked list. Also there are three possible cases to remov e the node from linked list:

I) Removing the beginning node of linked list:

This operation removes the beginning node from linked list. Also this function returns one integer value which is ‘*info*’ part of removed node.

The remove begin operation is shown in following fig:



```
n = p →info;
list = p→next;
delete p ;
return ( n) ;
```

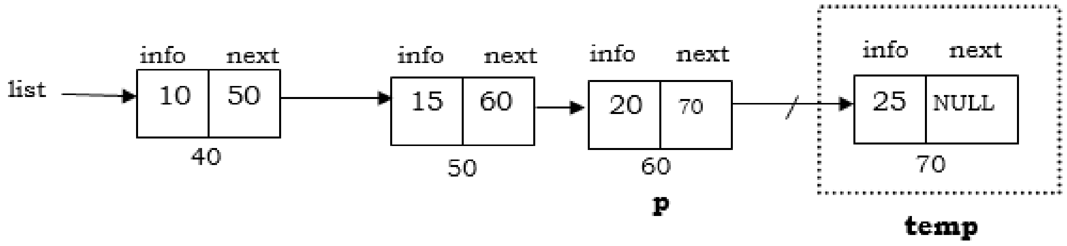
Operation:

```
int    rem_beg( )
{
        int    n;
        node  *p;
        p = list;
        if ( p = = NULL )      // if  linked list is empty
```

```
            {
                cout<<" Linked list is empty .... ";
            }
            else   if( p → next = = NULL )       // Linked list h   aving one node
            {
                n = p → info;
                delete p;
                list = NULL;
                return( n );
            }
            else
            {
               n = p → info;
              list = p→next;
              delete p ;
              return ( n) ;
             }
        }
```

II) Removing th e last (End) node of linked list:

This operation removes the last (End) node from linked list. Also this function returns one integer value which is '*info*' part of removed node.
To remove last node of linked list, we have to move first node ' *p'* at second last position. After setting '*p'* at second last position, make ' *temp'* as next of node '*p'* therefore '*temp'* becomes last node of linked list. And remove node '*temp*'.
The remove end operation is shown in following fig:



```
temp = p → next ;
n = temp→info;
p→next = NULL;
delete temp ;
return ( n) ;
```

Operation:

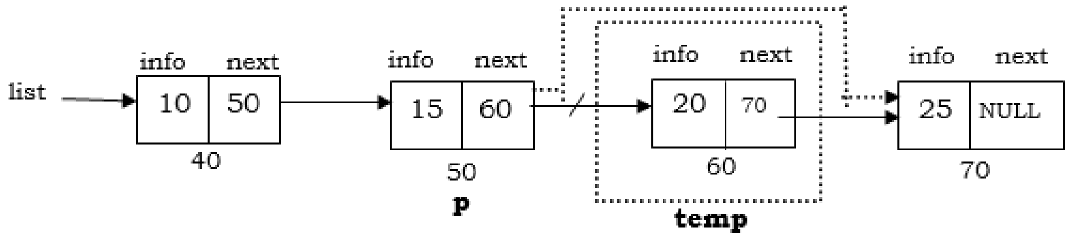
```
   int    rem_end( )
   {
            int    n;
            node  *p, *temp;
            p = list;
            if ( p = = NULL )       // if  linked list is empty
            {
                cout<<" Linked list is empty .... ";
            }
            else   if( p → next = = NULL )       // Linked list having one node
            {
                n = p→ info;
                delete   p;
                list = NULL;
                return( n );
            }
            else
            {
                    // set p as second last position
                while( p→ next→next != NULL)
                {
                        p = p→ next;
                }
                temp = p → next ;        // set temp as last node
                n = temp→ info;
                p→next = NULL;
                delete temp ;
                return ( n) ;
        }
}
```

III) Removing the node which is in between two nodes of linked list:

This operation removes the node which is in between two nodes of linked list. This operation is also called as "Remove after". This function returns one integer value which is ' *info'* part of removed node.

To remove node which is in between two nodes, we have to move from first node towards the last node, while doing so, check ' *info '* part of node 'p' is matches with ' *after '* or not. If 'after ' is matches with ' *info'* part then, make ' *temp'* as next of node '*p'* therefore '*temp'* becomes the node which is in between two nodes of linked list. And remove node ' *temp'*.

The remove in between (Remove after) operati on is shown in following fig:



```
temp = p →   next ;
n = temp→info;
p→next = temp→next
delete temp );
return ( n) ;
```

Operation:

```
   int    rem_bet( int  after)
   {
             int    n;
             node  *p, *temp;
             p = list;
             if ( p = = NULL )       // if  linked list is empty
             {
                 cout<<" Linked list is empty .... ");
             }
                             // if Linked list have    one or two nodes
             else   if( p→ next = = NULL || p →next→next = = NULL)
             {
                cout<<"Remove between is not possible " ;
             }
             else
             {
                            // move from first node to last node
                     while( p→next != NULL)
                     {
                             if ( p→info = = after )
                             {
                                temp = p→next ;
                                n = temp→info;
```

```
                        p→next = temp→ next ;
                        delete temp ;
                        return ( n) ;
                    }
                p = p→next;
            }
    }
```

4) Search Operation:

As the name implies, this operation search whether particular node is present in linked list or not. For that we have to move from first node towards the last node while doing so, check *'info'* part of each node is matches with search value or not.

Operation:

```
        void    search( int  srch )
        {
                int      f = 0;
                node  *p;
                p = list;
                while(p != NULL)
                {
                        if ( p→info = = srch )
                        {
                                f =1;
                                break;
                        }
                        p = p→next;
                }
                if ( f = = 1)
                {
                        cout<<"Node  is  found";
                }
                else
                {
                        cout<<" Node is not found ";
                }
        }
```

5) Count operation:

As the name implies, this operation counts total number of nodes present in linked list. For that we have to move from first node towards the last node while doing so, just increment counter by one.

Operation:

```
        void    count( )
        {
                int   cnt = 0;
                node  *p;
                p = list;
                while(p != NULL)
                {
                        cnt =  cnt + 1;     // increment counter by one
                        p = p→next;
                }
                cout<<"\n Total Number of Nodes in Linked list =" << cnt ;
        }
```

6) <u>Display (Status) operation:</u>

As the name implies, this operation displays the ‘ info’ part (actual element) of each nodes present in linked list. For that we have to move from first node towards the last node while doing so, just display info part of each node.

Operation:

```
void   display( )
{
       node  *p;
       p = list;
       while(p != NULL)
       {
              cout<<“ \t “<< p→info ;        // displays info part of node
              p = p→next;
       }
}
```

7) <u>) Reverse operation:</u>

This operation reverses the entire singly linear linked list.

To reverse the singly linked list, we have to revers e each node i.e. we have to break the ‘next’ part of each node and attach it to its previous node thus every node gets reversed. Lastly attach ‘list’ to last node (Since, after reversal last node becomes first node.)

Operation:

```
void    reverse( )
{
          node *t1,*t2,*t3=NULL;
          t1=list;
          while(t1!=NULL)
          {
            t2 = t1→next;
            t1→next = t3;
            t3 = t1;
            t1 = t2;
          }
          list= t3;
          cout<<“\nLinked list is reversed...";
}
```

2) Doubly Linear Linked list:

To implement doubly linear linked list, we have to define one class which is as fallow:

```
class          node
{
       int        info;
       node    *next, * prev;
       public:
       // declaration of operations of DLLL
} *list;
```

Operations performed on to the doubly linear linked list:

There are some following operations that can be operate on to the linked list:

1) Create operation:

This operation creates new node dynamically and returns node pointer towards called function.

Operation:

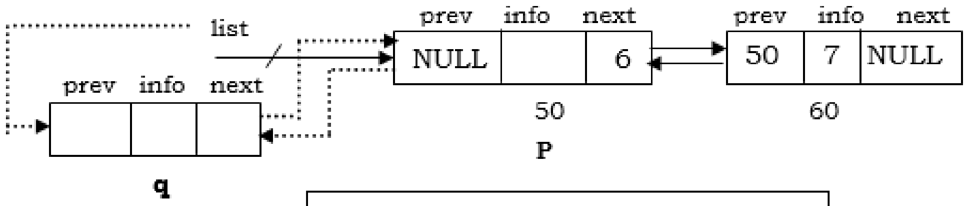
```
node*    create( )
{
         node    *z;
         z = new node;
         return(z);
}
```

2) Insert Operation:

This operation inserts new node into link ed list. Also there are three possible cases to insert new node into linked list:

I) Inserting node at the beginning of linked list:

This operation inserts n ew node at the beginning of linked list, which is shown in fallowing figure:



```
q = create( );
q→info = x;           // x is  int
q→next = p;
q→prev = NULL;
p→prev = q ;
list = q ;
```

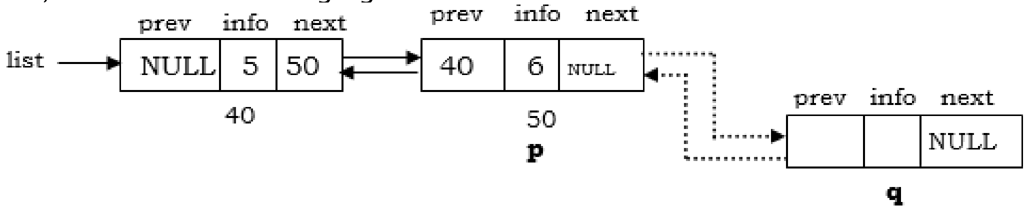
Operation:

```
void     ins_beg ( int x )
{
         node    *p, *q ;
         p = list;
         if ( p = = NULL )           // if linked list is empty
         {
                  p = create( );        //  newnode created
                  p→info = x;
                  p→ next = NULL;
                  p→ prev = NULL;
                  list = p;
         }
         else
         {
                           q = create( );
                           q→ info = x;
                           q→next = p;
                           q→prev = NULL;
                           p→ prev = q ;
                           list = q ;
         }
}
```

II) Inserting node at the end of linked li st:

This operation inserts new node at the end of linked list.

For that we have to move first node ‘p’ in forward direction and set ‘p’ as last node, after setting ‘p’ node as last node then insert new node ‘q’ after node ‘p’ which is last node of linked list, is shown in fallowing figure:



```
q = create( ) ;
q→ info = x ;      // x is integer
q→next = NULL;
q→prev = p ;
p→next = q;
```

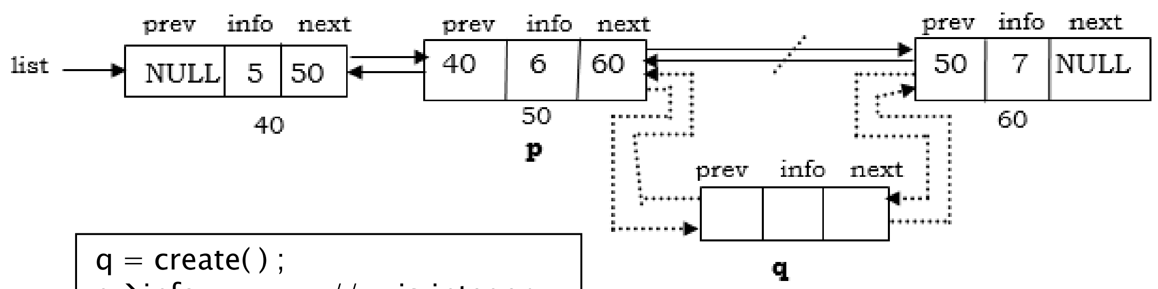
Operation:

```
void    ins_end ( int  x )
{
        node    *p, *q ;
        p = list;
        if ( p = = NULL )           // if linked list is empty
        {
                p = create( );              //  newnode created
                p→info = x;
                p→next = NULL;
                p→prev = NULL;
                list = p;

        }
        else
        {               //  move from first node to last node and set ‘p’ as last node
                while ( p→next != NULL )
                {
                        p = p→next;
                }
                q = create( ) ;      //  insert new node   ‘q’ after ‘p’ which is last node.
                q→info = x ;
                q→ next = NULL;
                q→prev = p ;
                p→next  = q ;
        }

}
```

III) Inserting node in between two nodes of linked list: (insert after)

This operation inserts new node which is in between two nodes o f linked list.

For that we have to move from first node towards the last node, while doing so, check ‘ *info* ’ part of node ‘p’ is matches with ‘ *after* ’ or not. If ‘ *after* ’ is matches with ‘ *info* ’ part then insert new node ‘q’ which is in between two n odes of linked list, is shown in fallowing figure:



```
q = create( ) ;
q→info = x ;      // x is integer
q→next = p→next ;
q→prev = p ;
p→next→prev = q ;
p→next = q ;
```

Operation:

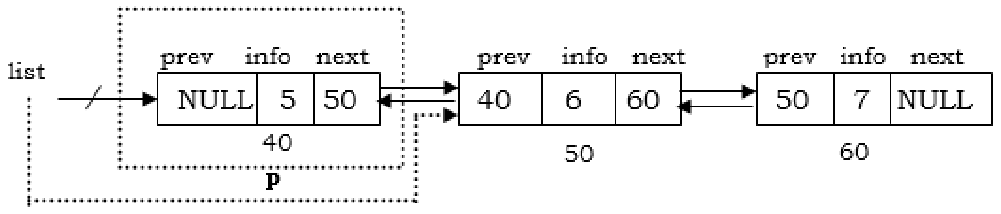
```
void    ins_bet ( int  after, int  x )
{
        node    *p, *q ;
        p = list;

        if ( p = = NULL || (p → next = = NULL & & p →prev = = NULL )
                                        /* if  linked list is empty or  linked list having one node */
        {
                cout<<" Insert between is not possible ";
        }

        else
        {                                // move from first node to last node
                while ( p→ next != NULL )
                {
                        if ( after = = p→info)           // info part matches with after then
                        {
                                q = create( ) ;
                                q→info = x ;      // x is integer
                                q→next = p→next ;
                                q→prev = p ;
                                p→next→prev = q ;
                                p→next = q ;

                        }
                        p = p→next;
                }

        }

}
```

3) Remove Operation:

This operation removes the node from linked list. Also there are three possible cases to remove the node from linked list:

I) Removing the beginning node of linked list:

This operation removes the beginning node from linked list. Also this function returns one integer value which is '*info*' part of removed node.
The remove begin operation is shown in following fig:



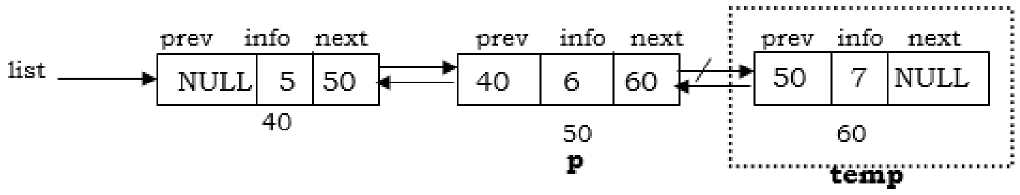
```
n = p→info ;
p→next→prev= NULL;
list = p→next;
delete p;
return(n);
```

Operation:

```
int    rem_beg( )
{
            int    n;
            node  *p;
            p = list;
            if ( p = = NULL )        // if  linked list is empty
            {
                cout<<" Linked list is empty .... ");
            }
            else   if( (p→next = = NULL) &&( p →prev==NULL )        // Linked list   having one node
            {
                n = p→info;
                delete p);
                list = NULL;
                return( n );
            }
            else
            {

                     n = p→info ;
                     p→next→prev= NULL;
                     list = p → next;
                     delete p;
                     return(n);

            }
    }
```

<u>II)</u> <u>Removing the last (End) node of linked list:</u>

This operation removes the last (End) node from linked list. Also this function returns o ne integer value which is '*info'* part of removed node.
To remove last node of linked list, we have to move first node ' *p'* at second last position. After setting '*p'* at second last position, make ' *temp'* as next of node '*p'* therefore '*temp'* becomes last node of linked list. And remove node '*temp*'.
The remove end operation is shown in following fig:



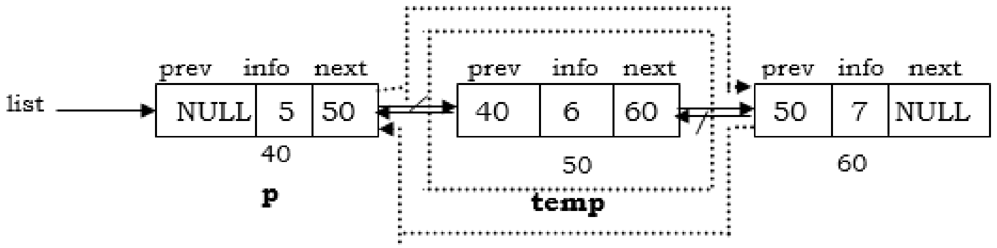
```
n = temp→info ;
p→next= NULL;
delete temp;
return(n);
```

Operation:

```
int        rem_end( )
{
        int   n;
        node  *p, *temp;
        p = list;
        if ( p = = NULL )      // if  linked list is empty
        {
            cout<<" Linked list is empty .... ";
        }
        else   if( ( p→next = = NULL)&&(p →prev = =NULL) )
        {
            n = p→info;
            delete p);
            list = NULL;
            return( n );
        }
        else
        {
                        // set p as second last positio  n
                while( p→next→next != NULL)
                {
                        p = p→next;
                }
                temp = p→next ;        // set temp as last node
                n = temp→info;
                p→next = NULL;
                delete temp;
                return (n) ;
        }
}
```

<u>III) Removing the node which is in between two nodes of lin   ked list:</u>

This operation removes the node which is in between two nodes of linked list. This operation is also called as "Remove after". This function returns one integer value which is '*info'* part of removed node.
To remove node which is in between two no des, we have to move from first node towards the last node, while doing so, check ' *info '* part of node 'p' is matches with ' *after '* or not. If 'after *'* is matches with 'info*'* part then, make ' *temp'* as next of node '*p'* therefore '*temp'* becomes the node which is in between two nodes of linked list. And remove node ' *temp'*.
The remove in between (Remove after) operation is shown in following fig:



```
temp = p→next;
n = temp→info ;
temp→next→prev = p;
p→next = temp→next;
delete   temp;
return(n);
```

Operation:

```
  int    rem_bet( int  after)
  {
                int    n;
                node  *p, *temp;
                p = list;
                if ( p = = NULL )        // if  linked list is empty
                {
                     cout<<" Linked list is empty .... ");
                }
                                                       // if Linked list have one or two nodes
                else   if( (p→next = = NULL)&&(p →prev = = NULL) ||
                              ( p→next→next= =NULL)&& (p→prev = = NULL))
                {
                    cout<<"Remove between is not possible " ;
                }
                else
                {
                              // move from first node to last node
                         while( p→next != NULL)
                         {
                                  if ( p→info = = after )
                                  {
                                           temp = p→next;
                                           n = temp→info ;
                                           temp→next→prev = p;
                                           p→next = temp→next;
                                           delete   temp;
                                           return(n);
                                  }
                            p = p→next;
                       }
   }
```

NOTE:

The **search, count and display** operations of doubly linear linked list is same as singly linear linked and they are as fallows.

4) Search Operation:

As the name implies, this operation search whether particular node is present in linked list or not. For that we have to move from first node towards the last node while doing so, check *'info'* part of each node is matches with search value or not.

Operation:

```
void   search( int  srch )
{
       int     f = 0;
       node  *p;
       p = list;
       while(p != NULL)
       {
              if ( p→info = = srch )
              {
                     f =1;
                     break;
              }
              p = p→next;
       }
       if ( f = = 1)
       {
              cout<<"Node  is  found";
       }
       else
       {
              cout<<" Node is not found ";
       }
}
```

5) Count operation:

As the name implies, this operation counts total number of nodes present in linked list. For that we have to move from first node towards the last node while doing so, just increment counter by one.

Operation:

```
void   count( )
{
       int   cnt = 0;
       node  *p;
       p = list;
       while(p != NULL)
       {
              cnt =  cnt + 1;     // increment counter by one
              p = p→next;
       }
       cout<<"\n Total Number of Nodes in Linked list ="<< cnt ;
}
```

6) Display (Status) operation:

As the name implies, this operation displays the ' info' part (actual element) of each nodes present in linked list. For that we have to move from first node towards the last node while doing so, just display info part of each node.

Operation:

```
void   display( )
{
       node  *p;
```

```
            p = list;
            while(p != NULL)
            {
                    cout<<" \t"<< p→info ;        // displays info part   of node
                    p = p→next;
            }
    }
```

7)) Reverse operation:

This operation reverses the entire doubly linear linked list.

To reverse the doubly linked list, we have to reverse each node i.e. we have to break the 'next' part of each node and attach it to its previ ous node and break the 'prev' part & attach it to its next node, thus every node gets reversed. Lastly attach 'list' to last node (Since, after reversal last node becomes first node.)

Operation:

```
void    reverse( )
{
            node *t1,*t2,*t3=NULL;
            t1=list;
            while(t1!=NULL)
            {
               t2 = t1→next;
               t1→next = t3;
               t1 →prev = t2;
               t3 = t1;
                t1 = t2;
            }
            list=t3;
            cout<<"\nLinked list is reversed...";
}
```

3) Singly circular Linked list:

Singly circular linked list is collection of nodes and each node contains two parts viz 'info' and 'next' also such linked list does not have NULL pointer which is replaced by address of first node.

To implement singly circular linked list we have to define one class which is as fallow:

```
class           node
{
        int        info;
        node    *next;
        public:
        // declaration of operations of SCLL
} *list;
```

Operations performed on to the singly circular linked list:

There are following operations that can be operate on to the singly circular linked list:

1) Create operation:

This operation creates new node dynamically and returns node pointer towards called function.

Operation:

```
node*    create( )
{
        node    *z;
        z = new node;
        return(z);
}
```

2) Insert Operation:

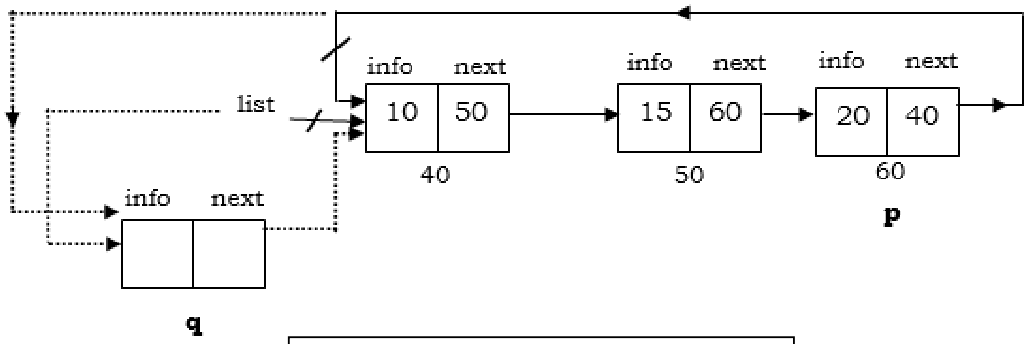
This operation inserts new node into singly circular linked list. Also there are three possible cases to insert new node into circular linked list:

1) Insert begin 2) Insert end 3) Insert Between

I) Inserting n ode at the beginning (First) of linked list:

To insert new node at the begin (first), we have to move node 'p' from first node to the last node and set 'p' as last node. After setting 'p' as last node , then insert new <u>node 'q' **after** node 'p'</u> which is first node of singly circular linked list.

Following fig. shows inserting new no de at the beginning of singly circular linked list:



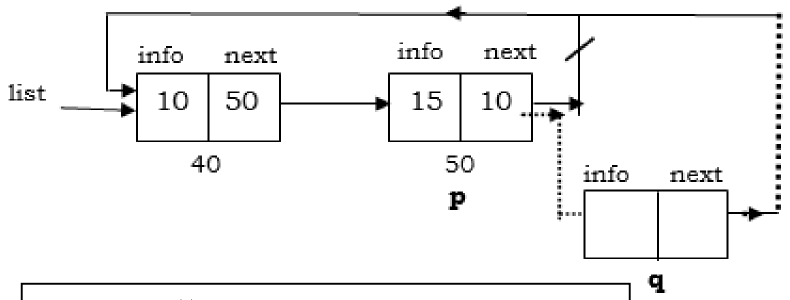
```
q = create( ) ;
q→info = x ;      // x is integer
q→next = p→next ;
p→next = q ;
list = q ;
```

Operation:

```
void   ins_beg(int   x)
{
        node    *p,*q;
        p = list;        // 'p' becomes first node
        if ( p = = NULL )        // L.L is empty
        {
                p = create( );
                p→info = x;
                p→next = p ;
                list = p ;
        }
        else
        {
                while ( p →next != list )
                {
                        p = p→next ;     // set 'p' at  last position
                }
                q = create( );
                q→info = x;
                q → next = p→next;
                p→ next = q;
                list = q;
        }
}
```

II) Inserting no de at the end (Last) of linked list:

To insert new node at the end, we have to move 'p' from first node to the last node and set 'p' as last node. After setting 'p' as last node, then insert new node **near** to node 'p' which is last node of singly circular l inked list.
Following fig. shows inserting new node at the end of singly circular linked list:



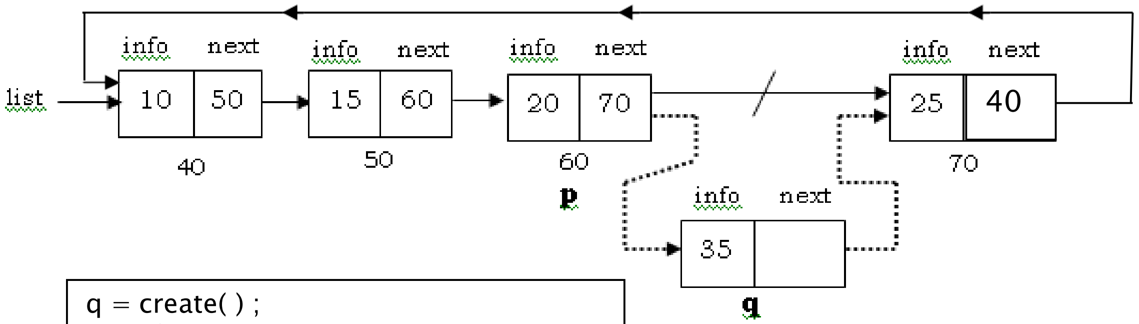
```
q = create( ) ;
q →info = x ;            // x is integer
q→next = p → next ;
p →next = q ;
```

Operation:

```
void   ins_end(int   x)
{
       node    *p,*q;
       p = list;        // 'p' becomes first node
       if ( p = = NULL )        // L.L is empty
       {
              p = create( );
              p→ info = x;
              p→next = p ;
              list = p ;
       }
       else
       {
              while ( p→next != list )
              {
                     p = p→ next ;        // set 'p' at last  position
              }
              q = create( );
              q→ info = x;
              q→ next = p→next;
              p→ next = q;
       }
}
```

III) Inserting new node in between two nodes:

This operation inserts new node in between two nodes of singly circular linked list. For that we have to move from first node towards the last node, while doing so, check 'info' part of every node is matches with 'after' or not . If 'after' part is matches with 'info' part then insert new node 'q' which is in between two nodes of linked list.
This is shown in following figure:



```
q = create( ) ;
q→info = x ;      // x is integer
q→next = p→next ;
p→next = q ;
```

Operation:

```
void   ins_bet(int    after, int x)
{
        node    *p,*q;
        p = list;   // 'p' becomes first node
        if ( p = = NULL | | p  →next = = p)        // L.L is empty OR having one node
        {
                cout<<" Insert Between not possible");
        }
        else
        {
                while( p→next != list )
                {
                        if ( p→info = = after )          // check 'info' part with 'after'
                        {
                                q = create( );
                                q→info = x;
                                q→next = p→next;
                                p→next = q;
                        }
                        p = p →next;
                }
        }
}
```

3) Remove Operation:

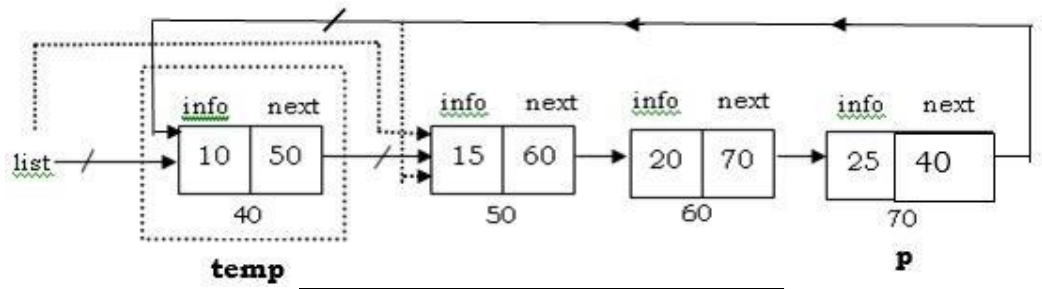
This operation removes the node from linked list. Also there are three possible cases to remove the node from singly circular linked list:

I) Removing the beginning (First) node of linked list:

This operation removes the beginning (first) node from singly circular linked list. Also this function returns one integer value which is ' *info'* part of removed node.

To remove first node of singly circular list, we have to move 'p' from first node to the last node and set 'p' as last node. After setting 'p' as last node then make 'temp' node as **next** of 'p' node therefore 'temp' node becomes first of linked li st. Then remove node 'temp'

Following fig. shows removing first node of singly circular linked list:



```
temp = p → next ;
n = temp → info ;
p → next = temp → next ;
list = temp→next;
delete temp ;
return ( n ) ;
```

Operation:

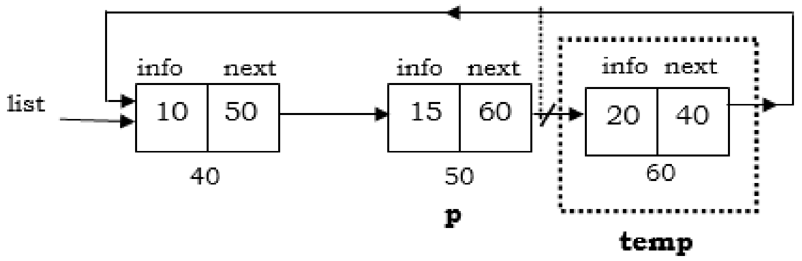
```
int      rem_beg( )
{
         node    *p,*temp;
         int     n;
         p = list;         // 'p' becomes first node
        if ( p == NULL )
       {
                 cout<<"\n    Linked list is empty...";
        }
        else if ( p→next  = = p )                // L.L. having one node
         {
                 n = p→info ;
                 delete   p;
                 list = NULL;
                 return ( n );
         }
         else
         {
                 while( p→next != list )
                 {
                         p = p→ next ;            // set 'p' as  last node
                 }
                 temp = p → next ;              // set 'temp' as next of 'p'
                 n = temp → info ;
                 p → next = temp → next ;
                 list = temp  →  next;
                 delete temp ;
                 return ( n ) ;
         }
}
```

<u>II) Remo ving the last (End) node of linked list:</u>

This operation removes the last node from singly circular linked list. Also this function returns one integer value which is ' *info'* part of removed node.

To remove last node of singly circular list, we have to move 'p' from <u>first node to the second last node</u> and set 'p' as second last node. After setting 'p' as second last node then <u>make 'temp' node as **next** of 'p' node</u> therefore 'temp' node becomes <u>last</u> of linked list. Then remove node 'temp'

Following fig. shows removing last node of singly circular linked list:



```
temp = p → next ;
n = temp → info ;
p → next = temp → next ;
delete temp;
return ( n ) ;
```

Operation:

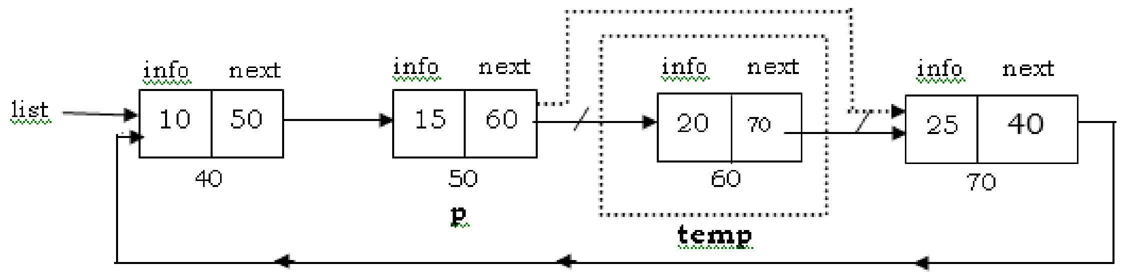
```
int     rem_end( )
{
        node    *p,*temp;
        int     n;
        p = list;        // 'p' becomes first node
       if ( p == NULL )
      {
              cout<<"\n    Linked list is empty...";
       }
       else if ( p → next  = = p )                  // L.L. having one node
        {
              n = p → info ;
              delete p;
              list = NULL;
              return ( n );
        }
        else
        {
              while( p → next→ next != list )
              {
                      p = p → next ;          // set 'p' as  second  last node
              }
              temp = p → next ;       // set 'temp' as next of 'p'
              n = temp → info ;
              p → next = temp → next ;
              delete temp ;
              return ( n ) ;
        }
}
```

<u>III)</u> <u>Removing the node which is in between two nodes of linked l   ist:</u>

This operation removes the node which is in between two nodes of linked list. This operation is also called as "Remove after". This function returns one integer value which is ' *info'* part of removed node.

To remove node which is in between two nodes, we have to move from first node towards the last node, while doing so, check ' *info '* part of node 'p' is matches with ' *after '* or not. If 'after ' is matches with ' *info'* part then, make ' *temp'* as next of node '*p'* therefore '*temp'* becomes the node which is in between two nodes of linked list. And remove node ' *temp'*.

The remove in between (Remove after) operation is shown in following fig:



```
temp = p → next ;
n = temp → info ;
p → next = temp →next ;
delete temp ;
return ( n ) ;
```

Operation:

```
        int      rem_bet(int  after )
        {
                node    *p,*temp;
                int     n;
                p = list;          // 'p' becomes  first node
            if ( p = = NULL )
            {
                        cout<<"\n    Linked list is empty…");
            }
          else if ( p → next  = = p || p →next→next= =p  )    // L.L. having one node OR two nodes
           {
                        cout<<"Remove between not possible";
            }
          else
           {
                        while( p → next != list )
                        {
                                if ( p → info = = after )  // 'after' matches with 'info'
                                {
                                        temp = p→ next ;
                                        n = temp→info ;
                                        p → next = temp → next ;
                                        delete temp ;
                                        return ( n ) ;
                                }
                                p = p →next ;
                        }

                }
        }
```

4) Search Operation:

As the name implies, this operation search whether particular node is present in linked list or not. For that we have to move from first node towards the last node while do ing so, check *'info'* part of each node is matches with search value or not.

Operation:

```
        void    search( int  srch )
        {
                int      f = 0;
                node  *p;
```

```
            p  = list;        // set 'p' as first node
                    do
                    {
                            if ( p →info = = srch )
                            {
                                    f =1;
                                    break;
                            }
                            p = p→next;
                    } while(p != list);

                    if ( f = = 1)
                    {
                         cout<<"Node is found";
                    }
                    else
                    {
                        cout<<" Node is not found ";
                    }
     }
```

5) Count operation:

As the name implies, this operation counts total number of nodes pre sent in linked list. For that we have to move from first node towards the last node while doing so, just increment counter by one.

Operation:

```
    void    count( )
    {
            int   cnt = 0;
            node *p=list;
            if(list= = NULL)
            {
                cnt= 0;
            }
            else
            {
                    do
                    {
                            cnt =   cnt + 1;      // increment counter by one
                            p = p→next;
                    } while(p != list);
            }
            cout<<"\n Total Number of Nodes in Linked list ="<< cnt;
    }
```

6) Display (Status) operation:

As the name implies, this operation displays the ' info' part (actual element) of each nodes present in linked list. For that we have to move from first node towards the last node while doing so, just display info part of each node.

Operation:

```
    void    display( )
    {
            node  *p;
            p = list;
            do
            {
                cout<<" \t"<<p→info ;        // displays info part of node
                p = p→next;
            } while(p != list);
        }
```

7)) Reverse operation:

This operation reverses the entire singly circular linked list.

To reverse the singly linked list, we have to reverse each node i.e. we have to break the 'n ext' part of each node and attach it to its previous node thus every node gets reversed. Lastly attach 'list' to last node (Since, after reversal last node becomes first node.)

Operation:

```
void    reverse( )
{
    node *t1,*t2,*t3=list;
    t1=list;
    do
    {
       t2 = t1→next;
       t1→next = t3;
       t3 = t1;
       t1 = t2;
    }while(t1!=list);
    list= t3;
    t1→next=t3;
    cout<<"\nLinked list is reversed...";
}
```

4) Doubly circular Linked list: (Home work)

***NOTE**:*

- Write all operations of doubly circular linked list with the help of *rough work* and *singly circular linked list notes .*
- The *search, count and display* operations of Doubly circular linked list are same as Singly circular linked list.

Reverse operation:

This operation reverses the entire doubly circular linked list.

To reverse the doubly linked list, we have to reverse each node i.e. we have to break the 'next' part of each node and attach it to its previous node and break the 'prev' part & attach it to its next node, thus every node gets reversed. Lastly attach 'list' to last node (Since, after reversal last node becomes first node.)

Operation :

```
void    reverse( )
{
    node *t1,*t2,*t3=list;
    t1=list;
    do
    {
       t2 = t1→next;
       t1→next = t3;
       t1 → prev = t2;
       t3 = t1;
       t1 = t2;
    }while(t1!=list);
    list= t3;
    t1→next=t3;
    cout<<"\nLinked list is reversed...";
}
```

- **<u>Dynamic stack (Stack using linked list)</u>**

  NOTE:
  - Use notes of this point discussed in the class.
- **<u>Dynamic Queue (Queue using linked list)</u>**

  NOTE:
  - Use notes of this point discussed in the class.

Theory Assignment No: 4

1) What is linked list? Why it is called as 'Dynamic data structure'?
2) What is header node? Write its advantage.
3) Why linked list is called as 'Linear Data structure'?
4) Write the difference between Array and linked list.
5) What is linked list? Explain types of linked list.
6) How Stack is implemented dynamically?
7) How Queue is implemented dynamically?
8) What is advantage of singly linked list over doubly linked list?
9) What is advantage of doubly linked list over singly linked list?
10) What is Singly linear linked list? Explain its following operations:

1) insert begin	2) insert end	3) insert between	
4) remove begin	5) remove end	6) remove between	
7) display	8) Search	9) count	10) reverse

11) What is Doubly linear linked list? Explain its following operations:

1) insert begin	2) insert end	3) insert between	
4) remove begin	5) remove end	6) remove between	
7) display	8) Search	9) count	10) reverse

12) What is Singly circular linked list? Explain its following operations:

1) insert begin	2) insert end	3) insert between	
4) remove begin	5) remove end	6) remove between	
7) display	8) Search	9) count	10) reverse

13) What is Doubly circular linked list? Explain its following operations:

1) insert begin	2) insert end	3) insert between	
4) remove begin	5) remove end	6) remove between	
7) display	8) Search	9)count	10) reverse

Practical Assignment No. 4

Note: Student must submit this assignment in <u>practical journal</u> within seven days from assignment issue date.

1) Write a menu driven program to implement singly linear linked list with its different operations.

2) Write a menu program to implement stack by using linked list. (Dynamic implementation of stack)

3) Write a menu program to implement queue by using linked list. (Dynamic implementation of queue)

4) Write a menu program to implement doubly linear linked list with its different operations.

5) Write a menu program to implement singly circular linked list with its different operations.

6) Write a menu driven program to implement doubly circular linked list with its different operations.