**LINKED LIST**

Array representation had the property that successive information was stored at a fixed distance apart.

1. If the element $a_{ij}$ of an array was stored at location $L_{ij}$, then $a_{i,j+1}$
   was at the location $L_{ij}+1$.

```
 Ex: int a[2][2];       a[0,0]  a[0,1]
                        a[1,0]  a[1,1]
          printf("Base address of 0th 1-d array %x\n", a[0]);
          printf("Base address of 2d-array %x\n",a);
  printf("Address of 0th element in 2-d array %x\n", &a[0][0]);
```

2. If the ith element in a queue is at location $L_i$, then the (i+1)th element will be at location $(L_i + 1)$ % n for the circular representation.
3. If the topmost node of a stack was at location $L_T$, then the node
   beneath it was at location $L_T - 1$ and so on.

Sequential storage schemes proved efficient for the tasks that were performed. (like accessing an arbitrary value in a table, insertion or deletion of stack and queue elements)

However, when a sequential mapping is used for ordered lists, operations such as insertion and deletion of arbitrary elements become expensive.  Consider the following list of 3 letter english words ending in AT.

(BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT, SAT, VAT, WAT)

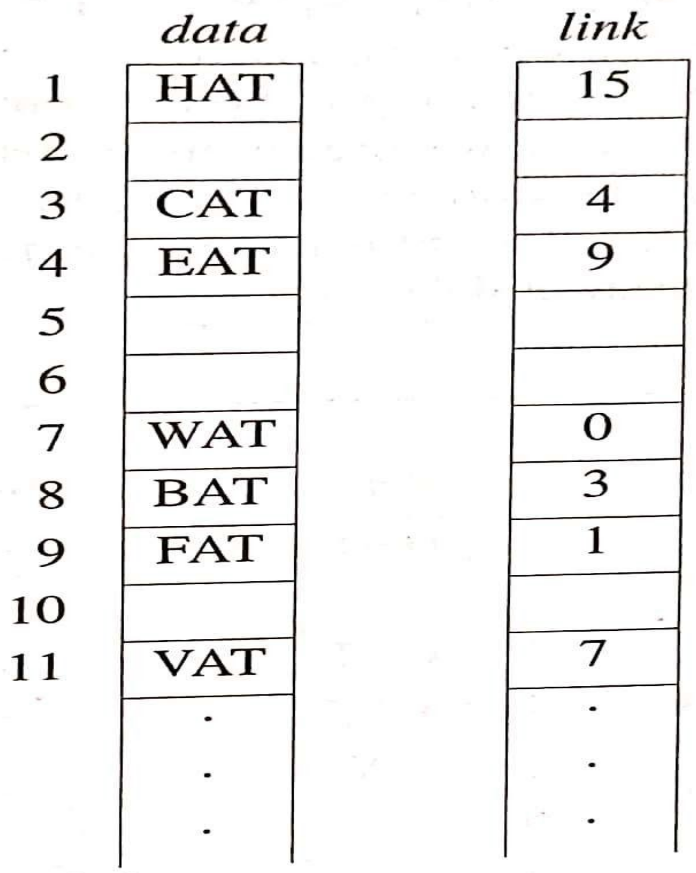
If GAT is to be added (meaning of GAT gun or revolver).
If an array is used to maintain this list, then the insertion of GAT will require already existing elements to be moved one location higher or lower.

If many such insertions are to be done at the middle of the array, neither alternatives (either moving up or down the existing values) will be feasible, because of the amount of data movement.

Excessive data movement also is required for deletions.

Solution to this problem of data movement in ***sequential representation*** is achieved by using ***linked representation***.

***Unlike a sequential representation, in which successive items of a list are located a fixed distance apart, in a linked representation these items may be placed anywhere in memory.***



|  | data | link |
|---|---|---|
| 1 | HAT | 15 |
| 2 |  |  |
| 3 | CAT | 4 |
| 4 | EAT | 9 |
| 5 |  |  |
| 6 |  |  |
| 7 | WAT | 0 |
| 8 | BAT | 3 |
| 9 | FAT | 1 |
| 10 |  |  |
| 11 | VAT | 7 |
|  | . | . |
|  | . | . |
|  | . | . |

Non Sequential list-representation

To access list elements in the correct order, with each element an address or location of the next element in the list will be stored.

Thus associated with each data item in a linked representation is a pointer or link to the next item.

Above diagram shows how some of the elements in the array may be represented in memory by using pointers/indexes.

Elements of the list are stored in a 1-D array called *data*, and the elements are no longer stored in sequential order.

To maintain the real order, a second array *link* is maintained. Values in this array are pointers to elements in the *data* array.

For any i, data[i] and link[i] together comprise a node.
List starts at data[8]=BAT, consider a variable *first=8*,
*link[8]* has the value 3, which means it points to *data[3],* which contains CAT, since *link[3]=4*, the next element, EAT, in the list is in *data[4].* The element after EAT is in *data[link[4]].*

By continuing in this way, all the words can be listed in proper order.

End of the ordered list will be encountered when the link field is having a 0 value.

*In general a linked list consists of nodes: each node has zero or more data fields and one or more link or pointer fields.*

It is customary to draw a linked list as an ordered sequence of nodes with links being represented by arrows.

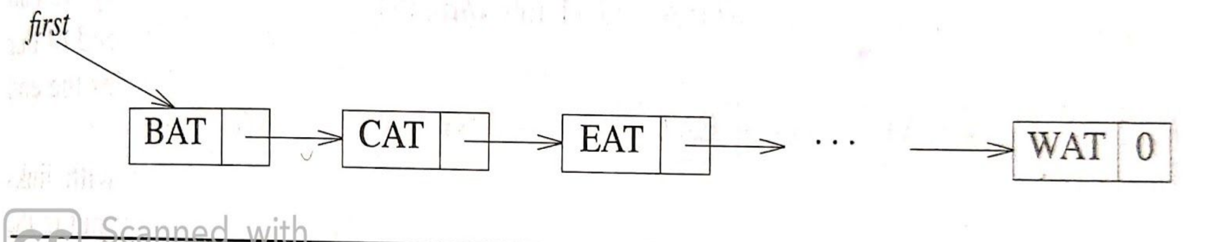


Figure 4.2: Usual way to draw a linked list

Notice that no explicit values have been added to pointers but simply arrows are drawn to indicate that they are there.

**Arrows reinforce the fact that**
**1. The nodes do not actually reside in sequential locations**
**2. The actual locations of nodes are immaterial.**

Therefore, when a code is written to realize a linked list, the address will not have any significance for the nodes in the list, but the link part will be tested for 0. (which signifies the end of the list)

The linked structures of the above 2 diagrams are called ***singly linked list or chains.***

In a singly linked list, each node has exactly one pointer field.

A chain/SLL is a singly linked list that consists of zero or more nodes.

When the number of nodes is zero the chain/SLL is empty.

Nodes of a non-empty chain are in such a way that the first node links to the second node, the second to the third and so on.

The last node of a chain has a 0 link.

This arrangement of data reduces the time complexity that is required in either inserting a value or removing it.

To insert the data item GAT between FAT and HAT, following steps are adequate.

1. Get a node ***a*** that is currently unused

2. Set the data field of ***a*** to GAT

3. Set the link field of ***a*** to point to the node after FAT, which contains HAT.

4. Set the *link* field of the node containing FAT to ***a***.

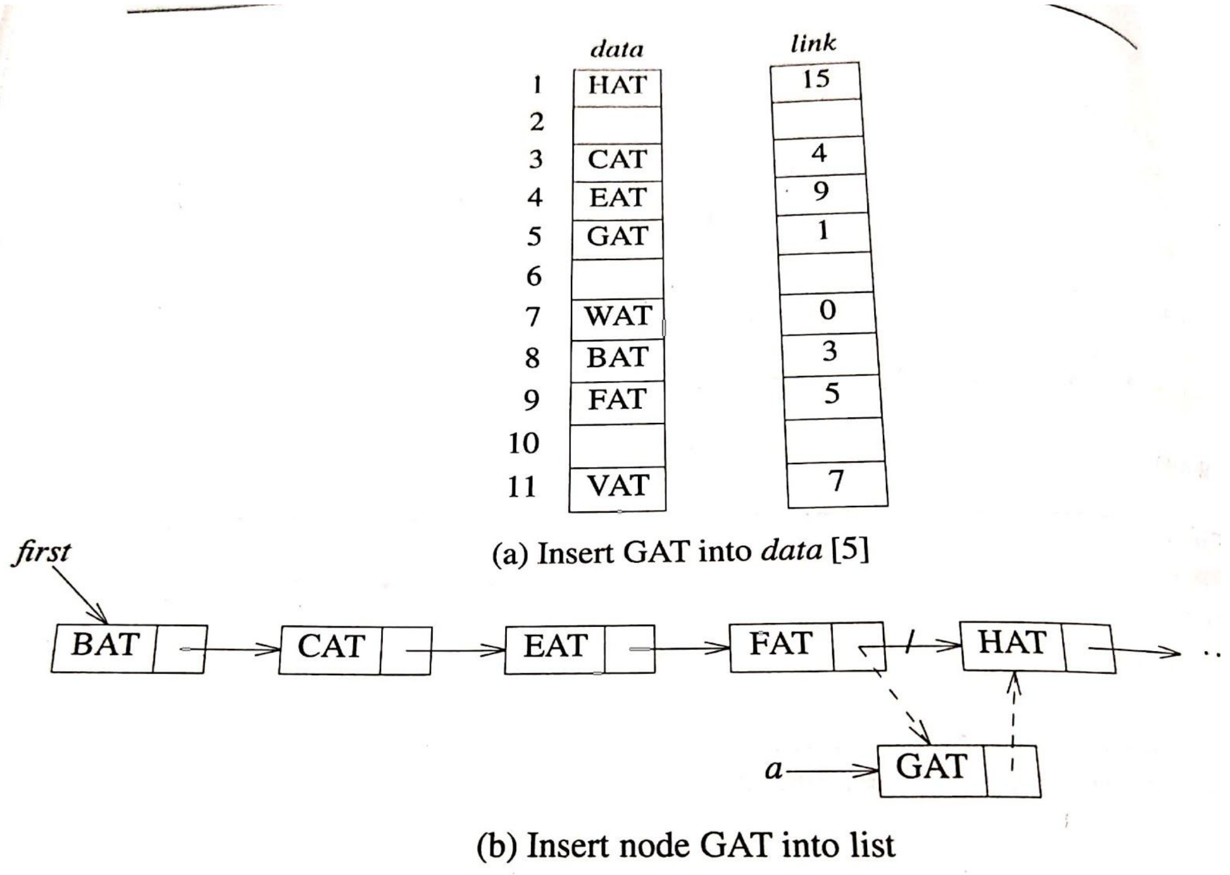


Figure 4.3: Inserting into a linked list

(a) part in the diagram shows the insertion of GAT to arrays data and link. (b) part shows the insertion using arrow notation.

Dashed arrows are the new ones.

**Important thing to notice is that when GAT is inserted, no elements have to be moved in the list.**

***In general a linked list consists of nodes: each node has zero or more data fields and one or more link or pointer fields.***

Following capabilities are required to make linked representation possible.

1. A mechanism for defining a node’s structure,

Node is made up of a data field and link/pointer field.

```
struct node
{
        int data;
        struct node*     link;  //self-referential pointers
};
//typedef struct node nd;
    nd p;
    printf("%x\n", &p);
    printf("%x\n", &(p.data) );
```

Self-referential pointers are the one which are capable of holding the addresses of the memory which holds the information of the type of the pointer.

Ex: int a=10; int *p=&a;

‘p’ is a pointer of similar type as variable a.

A structure which has a member as a pointer and the pointer
type being the same as structure type are termed as ***self-referential structures.***

2. A way to create new nodes when required.

Using Dynamic memory allocation schemes and not in automatic
Manner. !!!

3. A way to remove nodes that are no longer needed.
   *free( )* function handles the operation.

**TO CREATE A LINKED LIST OF WORDS.**
First a structure of type node will be created for the list.
Data part of the node will be a character array of size 4.
pointer /link to the next node.

```
struct  node
{
   char data[4];
   struct node * link;
};
typdedef   struct  node nd;
```

After creating the node's structure, a new empty list is created, which is accomplished by the statement.

```
nd*  first=NULL;
```

This statement indicates that a new list is created called *first*.

*first* contains the address of the start of the list or the first node in the list.

Since the new list is initially empty, its starting address is zero. *NULL* is a reserved word in C, which provides a 0 value.

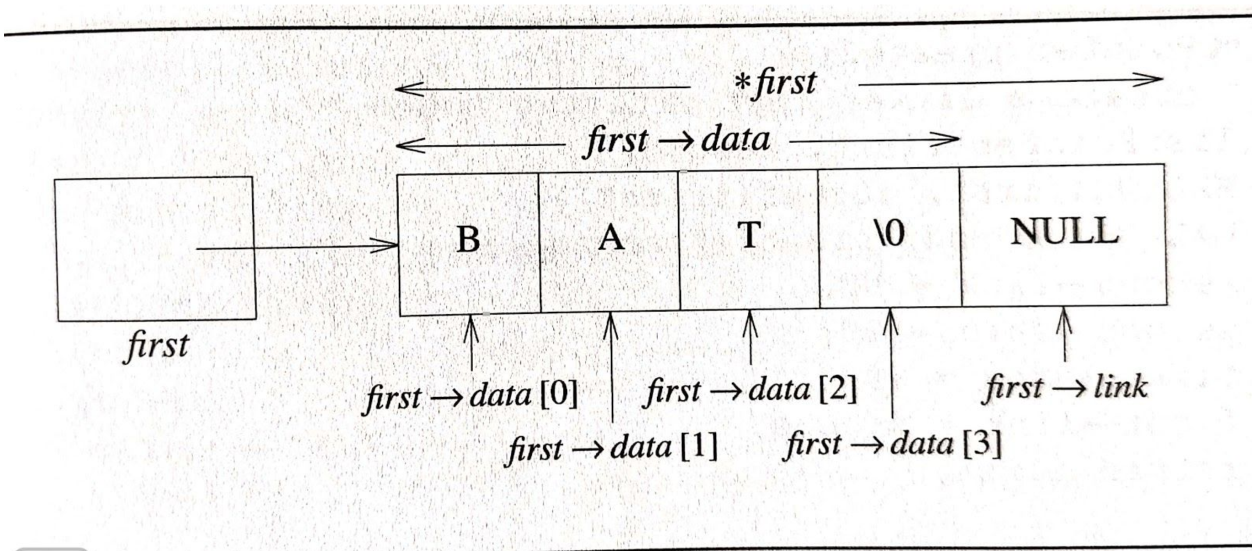
To create a new node for the list malloc will be used.

```
first = (nd *) malloc(sizeof(nd));
```

```
 printf("Contents of first %x\n",first);
 printf("Address of first %x\n", &first);
************************************************
struct student {
   char nm[20],usn[20];
   int marks[3];
};
struct student *  accept()  {
struct student  *t = (struct student *) malloc(sizeof(struct student));
   strcpy(t->nm,"akram");
   strcpy(t->usn,"077");
   t->marks[0]=90;
   t->marks[1]=91;
   t->marks[2]=92;
   return t;
}
void   display( const struct student * f, struct student val)
{
  printf("%s %s %d %d %d\n", f->nm,f->usn,
                          f->marks[0],f->marks[1],f->marks[2]);

 printf("%s %s %d %d %d\n",  val.nm,val.usn, val.marks[0],
val.marks[1], val.marks[2]);
}

int    main( ) {
   struct student  *first;
   first=accept();
   display(first,  *first );
   free(first);
}
```
**************************************************

Values can be assigned to the fields of the node

*strcpy(first->data, "BAT");*
*first->link = NULL;*



re 4.5: Referencing the fields of a node

Printing first node's information

*printf("%s\n", first->data);*

TO CREATE 2 NODES IN THE LIST

Already a node is created and it is pointed by the first pointer.

The link part of the first node is NULL because there is no second node in the list.

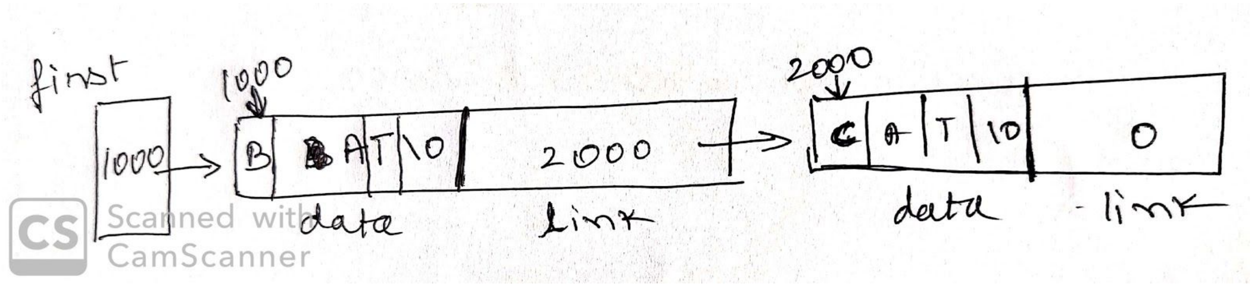
Second node will be created, memory will be acquired using DMA technique and suitable values will be assigned.

A temporary pointer is used for creating the second node.

```
nd * t;
t = (nd *) malloc(sizeof(nd));
strcpy(t->data,"CAT");
t->link = NULL;
```

This second node pointed by t must be attached to the link part of the first node.

```
first->link = t;
```



To print information in Linked List

```
nd *t=first;
printf("%s\n",t->data);
t=t->link;
printf("%s\n",t->data);
```

*first* is a significant pointer in LL, which holds on to the address of the first node in LL.

```
t=first->link;
free(first); free(t);
```

***Content of the first is not supposed to change unknowingly.***

If first contents are changed unknowingly, then access to the LL will be lost.

## INSERTING A NODE TO LL

*********************************

### 1. INSERT NODE to the FRONT END of the LIST

Function *insert_front* will be coded to insert nodes to the front end of the list.

*first* is the pointer declared in main scope, but insertion task is done in the function scope i.e *insert_front*.

Hence, content of *first* has to be shared with the function *insert_front*. (either using CBR or CBV technique)

*Ex:*

```
int main( ) {
    nd * first = NULL;
}
___ insert_front(_____)
{
  cannot access first in this scope until it is shared by main
scope.
}
```

Modified function header

*___ insert_front(nd * f)  // CBV for pointers*

**If 'f' is modified in the scope of *insert_front,* then first in main scope will not be modified. (CBV for pointers: one parameter 'first' shares it's contents with argument 'f')**

Ex:

```
 void   test( int * p)
 {
   printf("%d\n", *p);
   p = (int *) malloc(4);
   *p=1000;
 }
 int  main( ) {
    int  *q=(int *) malloc(4);
    *q= 2000;
    test(q);
    printf("%d\n",*q);
}
```

If a node is added to the front end of the list then *first* content has to be updated if CBV technique is used, return value of *insert_front* function will be *nd **.

Ex:

```
int main( ) {
    nd * first = NULL;
}
___ insert_front(nd * f)
{
```

If a node is considered as the first node or inserted to the front end
of the list then first nodes address has to be returned
*}*

Modified function header insert_front

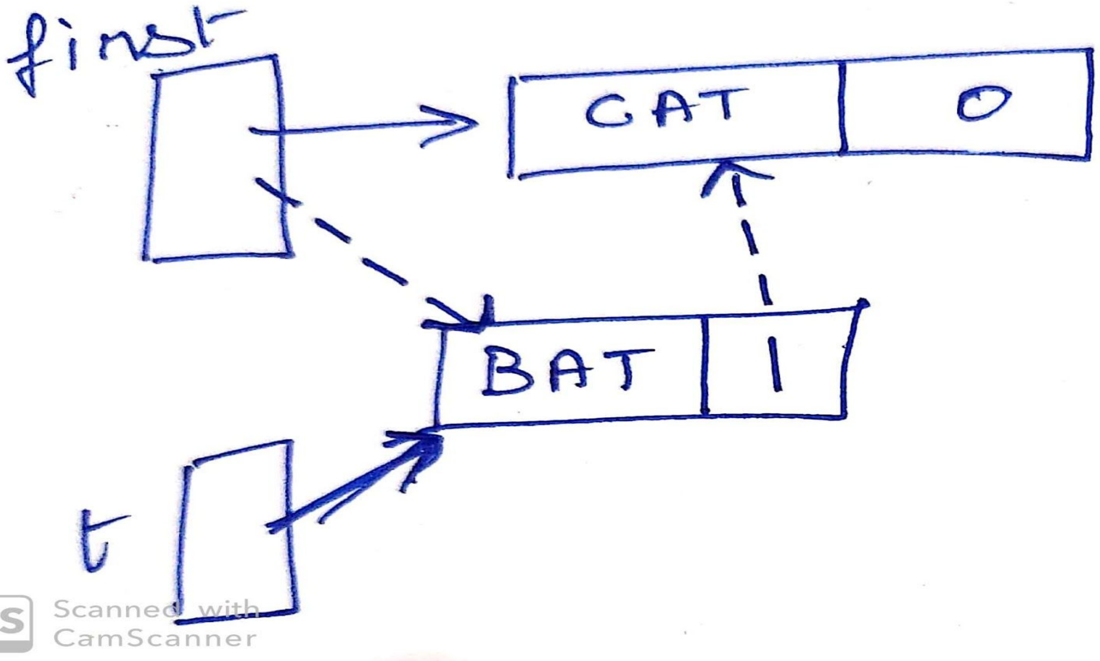
```
nd *    insert_front( nd * f)
{


}
```

STEP 1: If first is NULL or LL is empty, the newly created node will be the first and the last node in the list.

```
struct  node {
       int data;
       struct node * link;  };
typedef struct node nd;

nd *  insert_front( nd * );
int main( ) {
     nd * first = NULL;
    first  = insert_front(first);
   printf("%d\n", first->data);
}
```

```
nd *    insert_front( nd * f)
{
    nd  *t;                                  // 1
    t = (nd *) malloc(sizeof(nd));   // 2
   scanf("%d", &(t->data));          // 3
   t->link=0;                                // 4
/*STEPS 1 to 4 creates a node dynamically and fills data and
link field with suitable values. */

   if    (f  == NULL)
      return t;  // STEP 1
}
```

If LL is not empty, then the newly created node will be appended to the front end of the list.



‘t’  is a temporary pointer used to create a new node
After new node is created,

    first content will be transferred to t->link ( t->link = frist)

    And t’s content will be transferred to the first pointer.

(return t)

```
struct  node {
        int data;
        struct node * link;  };
typedef struct node nd;

nd *  insert_front( nd * );
int main( ) {
     nd * first = NULL;   nd *t;
    first  = insert_front(first);
   printf("%d\n", first->data);

    first  = insert_front(first);
    printf("%d\n", first->data);
   t=first;  t=t->link;
   printf("%d\n", t->data);       }

nd *    insert_front( nd * f)
{
    nd  *t;
    t = (nd *) malloc(sizeof(nd));
   scanf("%d", &(t->data));
   t->link=0;
```

```
  if   (f  != NULL)
       t->link = f;
 return t;      }
```

**Function to display the contents of the list**

*first* pointer is local to main and the function *display* will receive the argument i.e the contents of *first*.

Since the function *display* just reads the contents of the list, the return type of display will be void.

```
void display( nd * f)
{     }
```

LL has to be checked for empty, hence the first statement in display will be to check whether *f==NULL* or not.

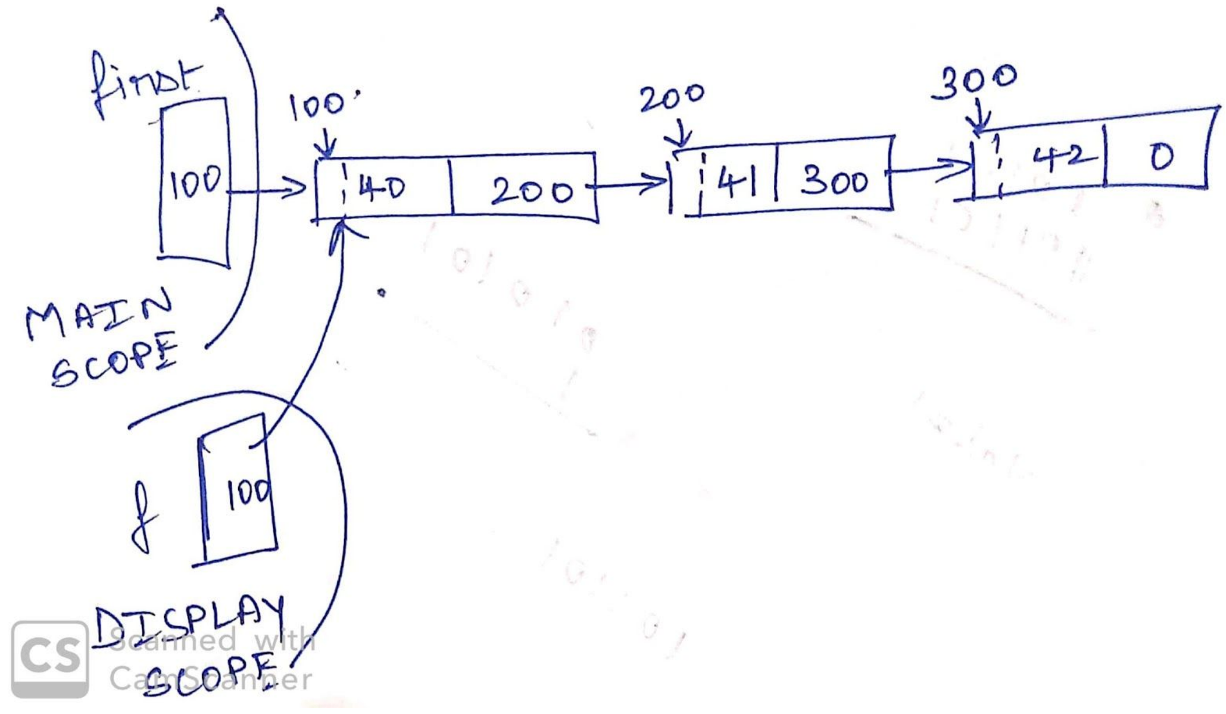
```
void display( nd * f)
{
  if  ( f  ==  NULL)
 {
    printf("LL is empty\n");
   return;
}
 }
```

If LL is not empty, then each node in the LL has to be visited and its information part has to be displayed, **termed as traversing the LL.**

In order to visit each and every node in the LL, traversing has to start always from the first node.

Using the link part of the first node, the next node has to be visited.
Upon visiting each node, the content of each node has to be printed.
Traversing stops when a node with its link value 0 is encountered.

```
void display( nd * f)
{
          if  ( f  ==  NULL)
         {
            printf("LL is empty\n");
           return;
         }
         for( ;  f != NULL ;f = (*f).link )
                printf("%d\n", (*f).data);
}
```



1st iteration of for loop in display
f content is 100
2nd iteration of for loop in display
f content is 200
3rd iteration of for loop in display
f content is 300
**4th iteration of for loop in display**
**f content is 0**

FULL PROGRAM USING *display* FUNCTION

```
struct  node {
        int data;
        struct node * link;  };
typedef struct node nd;

nd *  insert_front( nd * );
```

```
void  display( nd * );

int main( ) {
    nd * first = NULL;   int ch;
```

```
for(; ;)   {
        printf("1. Insert front\n2. Display\n3. Exit\n");
        printf("Enter choice: "); scanf("%d",&ch);
        switch(ch)   {
            case 1: first = insert_front(first); break;
           case 2: display(first); break;
          case 3: return 0;
        }
    }  }

nd *    insert_front( nd * f)  {
    nd  *t;
    t = (nd *) malloc(sizeof(nd));
    printf("Enter the information for data field\n");
   scanf("%d", &(t->data));
   t->link=0;

   if    (f  != NULL)
        t->link = f;
 return t;      }

void display( nd * f)  {
                if   ( f  ==  NULL)    {
                  printf("LL is empty\n");
                 return;
               }
               for( ;  f != NULL; )   {
                       printf("%d\n", f->data);
                       f = f -> link;
        }
}
```

**2. Inserting a node to the end of the list**

Situations of LL can be

1. LL is empty  first=NULL
2. LL is pointing to at least one node

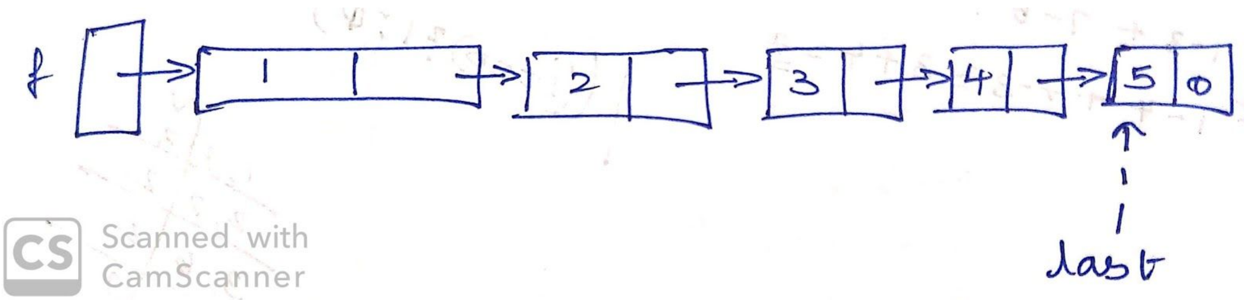
Statements in the function *insert_end* must handle both the situations.

If LL is empty then the significant pointer *first* is in main function, which will not be updated, hence *insert_end* function return type must be *nd **.

```
nd *   insert_end (  nd * f)
{
      nd * t = (nd *) malloc(sizeof(nd));
      scanf("%d", &(t->data));  t->link = NULL;

      if ( f == NULL)
        return t;
}
```

If the ‘if’ condition in *insert_end* is false then there may be 1 node or more than one node in the list.

Assuming that there are 5 nodes in the list, newly created node must be attached to the end of the LL.



A *last* pointer is required which points to the last node in the LL, **USAGE**: by using the last pointer, a new node that is created in *insert_end* function can be attached to the end of the list.

```
nd *   insert_end (  nd * f)
{
       nd * last;      nd * t = (nd *) malloc(sizeof(nd));
       scanf("%d", &(t->data));  t->link = NULL;

       if ( f == NULL)
         return t;

      for( last = f;  last->link != NULL; last = last->link);

      last->link = t;
     return f;     //not at all modified
}
```

FULL PROGRAM USING *insert_end* FUNCTION

```
struct  node {
       int data;
       struct node * link;  };
typedef struct node nd;

nd *  insert_front( nd * );
void  display( nd * );
nd * insert_end( nd *);

int main( ) {
     nd * first = NULL;   int ch;
   for(; ;)   {
       printf("1. Insert front\n2.Display\n3.Insert end\n");
      printf("4. Exit\n");
       printf("Enter choice: "); scanf("%d",&ch);
       switch(ch)   {
           case 1: first = insert_front(first); break;
          case 2: display(first); break;
         case 3: first = insert_end(first); break;
         case 4: return 0;
       }
  }  }

nd *    insert_front( nd * f)  {
    nd  *t;
    t = (nd *) malloc(sizeof(nd));
   scanf("%d", &(t->data));
   t->link=0;
   if    (f  != NULL)
        t->link = f;
 return t;      }

void display( nd * f)  {
                if   ( f  ==  NULL)    {
                  printf("LL is empty\n");
                 return;
              }
              for( ;  f != NULL; )   {
                     printf("%d\n", f->data);
                     f = f -> link;
              }
}

nd *    insert_end (  nd * f)
{
        nd * last;        nd * t = (nd *) malloc(sizeof(nd));
        scanf("%d", &(t->data));  t->link = NULL;

        if (  f == NULL)
          return t;

       for( last = f;  last->link != NULL; last = last->link);

       last->link = t;
     return f;     //not at all modified
}
```

**3. Inserting a node in the given position**

In order to insert a node at an accepted position value, first the number of nodes in the list has to be counted.

In function *insert_pos* first step will be to count the number of nodes in the list if LL is not empty.

Counting of the nodes will start from 1, and if LL is empty and if *pos* is 1 (which is a valid value for pos), then node will be inserted at the front end of the list and newly created node's address will be returned back from the function.

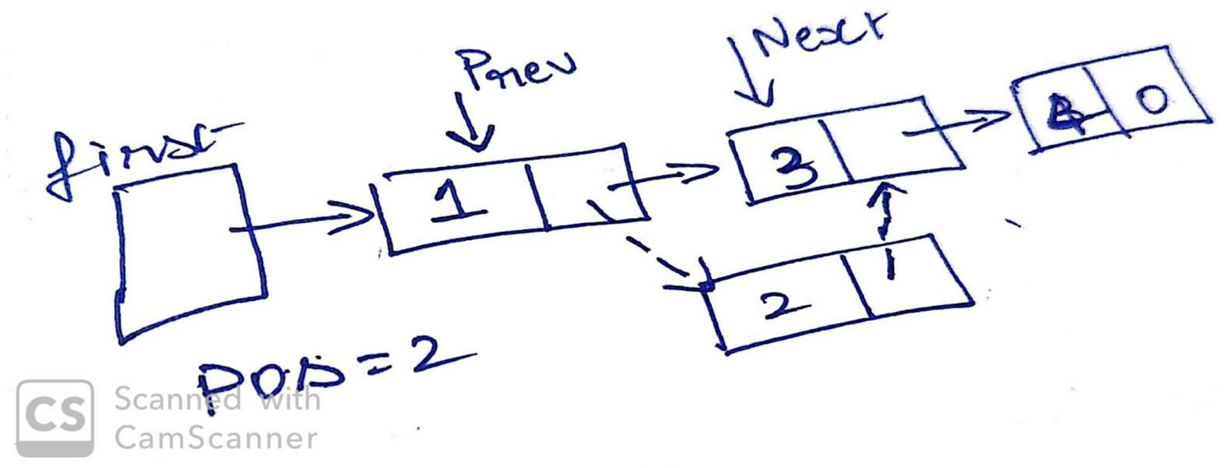
Template will be

```
nd *   insert_pos( nd * f)
{ .... }
```

If LL is having more than one node and *pos* value is 1 then the node has to be inserted at the front end of the list, even in this situation a newly created node's address must be returned back to the main function. (for updation of *first* pointer which is residing in the main scope)

If 3 nodes exist in LL and the *pos* value entered is 4 then the new node will be inserted at the end of the LL.

If 3 nodes exist in LL and the *pos* value entered is 2 then the new node created will be inserted right after the first node.



Two pointers *prev* and *next* are required to point to the previous and to the next node. In between these two pointers a new node will be inserted.
Dotted lines indicated the task that is required.

*first* is a pointer in main scope and has to be shared with *insert_pos( ),* and since node can be inserted as the first node, *first* pointer in main has to be updated and hence return type of *insert_pos( )* function must be *nd* *.

```
nd *  insert_pos  ( nd * f )
{
   int  cnt=0, pos;
   nd *t, *p, *n ;
   printf("Enter position value\n");  scanf("%d", &pos);

   t = (nd *) malloc(sizeof(nd));  scanf("%d", &(t->data));
   t->link = NULL;
```

```
//LL is empty AND pos = 1
if ( f  == NULL  && pos == 1)
     return t;

 //  LL is not empty AND pos = 1
  if ( f != NULL && pos == 1)
   {
       t->link = f;
      return t;
   }

    // to count no of nodes in LL
   for(p=f;  p != 0; p = p->link, cnt++);


   // LL is not empty AND pos = cnt+1
  if (  f != NULL && pos == cnt+1)
// inserting a node to the end of the list
  {
     for(p=f; p->link != 0; p=p->link);

     p->link = t;
    return f;
  }

 for( p=0, n=f;  pos>0; pos--, p=n, n=n->link);

    p->link = t;
    t->link = n;
   return f;
}
```

FULL PROGRAM USING *insert_pos* FUNCTION

```
struct  node {
        int data=90;
        struct node * link;  };
typedef struct node nd;

nd *  insert_front( nd * );
void  display( nd * );
nd * insert_end( nd *);
nd * insert_pos( nd *);

int main( ) {
     nd * first = NULL;   int ch;
   for(; ;)   {
        printf("1. Insert front\n2.Display\n3.Insert end\n");
       printf("4. Insert at position\n5. Exit");
        printf("Enter choice: "); scanf("%d",&ch);
        switch(ch)   {
            case 1: first = insert_front(first); break;
           case 2: display(first); break;
          case 3: first = insert_end(first); break;
          case 4: first = insert_pos(first); break;
          case 5: return 0;
        }
```

```
  }  }

nd *    insert_front( nd * f)  {
    nd  *t;
    t = (nd *) malloc(sizeof(nd));
   scanf("%d", &(t->data));
   t->link=0;

   if    (f  != NULL)
        t->link = f;
 return t;      }

void display( nd * f)  {
                if   ( f  ==  NULL)    {
                  printf("LL is empty\n");
                  return;
               }
               for( ;  f != NULL; )   {
                      printf("%d\n", f->data);
                      f = f -> link;
               }
}

nd *    insert_end (  nd * f) {
        nd * last;       nd * t = (nd *) malloc(sizeof(nd));
        scanf("%d", &(t->data));  t->link = NULL;

        if (  f == NULL)
          return t;
```

```
      for( last = f;  last->link != NULL; last = last->link);

      last->link = t;
     return f;     //not at all modified
}
```

***nd *  insert_pos ( nd * f )***

```
{
   int  cnt=0, pos;
   nd *t, *p, *n ;
   printf("Enter position value\n");  scanf("%d", &pos);

   t = (nd *) malloc(sizeof(nd));  scanf("%d", &(t->data));
   t->link = NULL;

   if (  f  == NULL  && pos == 1)
     return t;

    if ( f != NULL && pos == 1)
   {
      t->link = f;
      f=t;
      return f;
   }
```

```
 for(p=f;  p != 0; p = p->link, cnt++);

  if ( f != NULL && pos == cnt+1)
 {
   for(p=f; p->link != NULL; p=p->link);
   p->link = t;
   return f;
 }

if ( pos > cnt+1)
 {
   printf("Insertion not possible\n");
   free(t);
   return f;
 }

 pos--;

for( p=0, n=f;  pos>0; pos--, p=n, n=n->link);

   p->link = t;
   t->link = n;
  return f;
}
```

**4. Delete a node from the front end of the list**

Situations to handle

1. LL is empty and no nodes to delete
2. LL contains **only one** node.
3. LL contains **at least two** nodes.

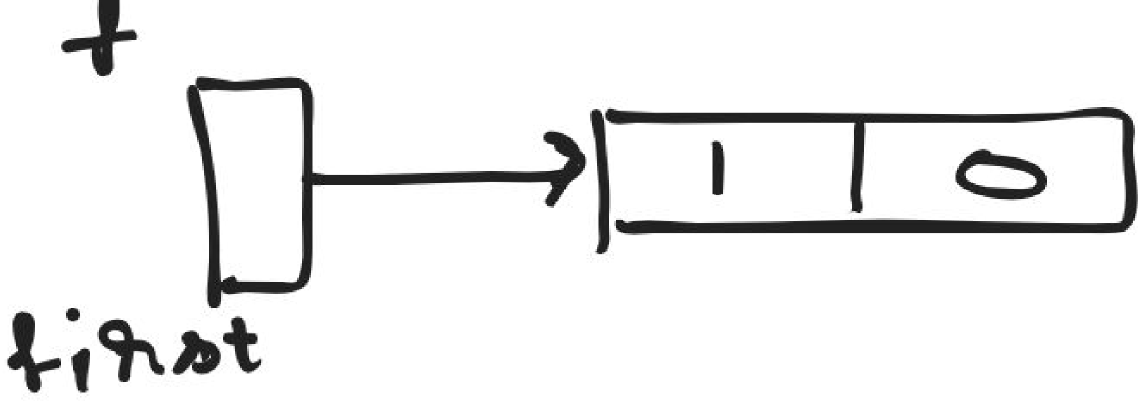
After deleting the first node in the list, by the function *delete_front( )*, the *first* which is present in the main scope must be updated with the first node's address.
Hence, return type of *delete_front( )* function is *nd **.

```
nd *   delete_front ( nd * f )
{ ..... }
```

If LL is empty, print a message and control returns.

```
nd *   delete_front ( nd * f )
{
  if   (  f == NULL)
  {
   printf("LL is empty\n");
   return   f;
 }


}
```

If LL is pointing to only one node.

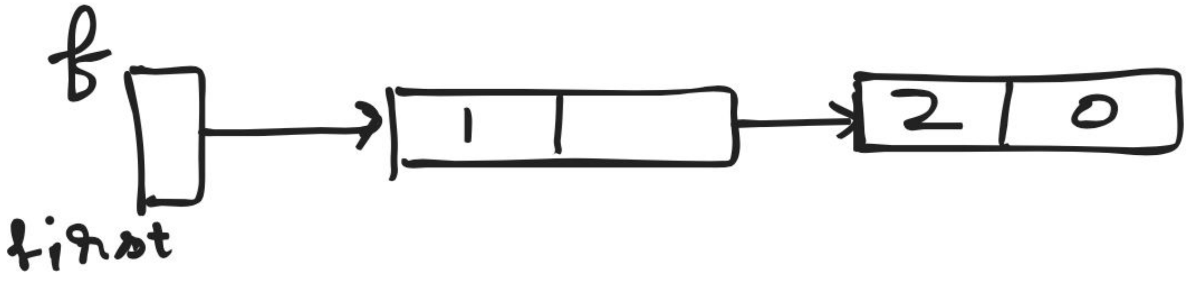


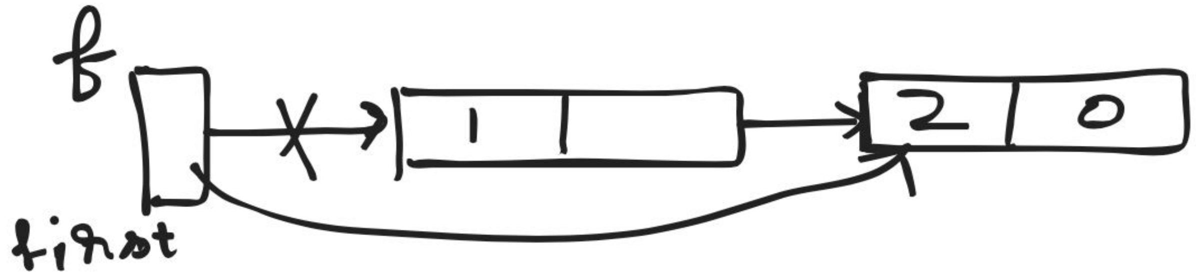
Then the data part of the node pointed by *f* must be printed and the **node's memory must be freed**.

```
nd *   delete_front ( nd * f )
{
   if   (  f == NULL)
  {
    printf("LL is empty\n");
    return   f;
 }

  if  ( f->link == NULL)
 {
    printf("Element deleted is %d\n", f->data);
   free( f );
   return 0;
}
}
```

If LL contains more than 1 one node then,

Node's data pointed by *f* must be displayed.
*first* pointer must point to the second node.
Then returns the address of the second node from the function *delete_front( )*.





```
nd *   delete_front ( nd * f )
{
    nd * t;
   if    (  f == NULL)
  {
    printf("LL is empty\n");
    return   f;
 }
 if  ( f->link == NULL)
 {
    printf("Element deleted is %d\n", f->data);    // redundant line
   free( f );
   return 0;
}
 printf("Element deleted is %d\n", f->data);    // redundant line
t=f;  f=f->link;
free(t);
return f;
}
```

OPTIMIZED *delete_front( )*

```
nd *   delete_front ( nd * f )
{
    nd * t;
   if    (   f == NULL)
  {
    printf("LL is empty\n");
    return   f;
 }

   printf("Element deleted is %d\n", f->data);
  if  ( f->link == NULL)
 {
    free( f );
   return 0;
}
 t=f;     f=f->link;
free(t);
return f;     }
```

**FULL PROGRAM USING *delete_front( )* FUNCTION**

```
struct  node {
        int data;
        struct node * link;  };
typedef struct node nd;

nd *  insert_front( nd * );
void  display( nd * );
nd * insert_end( nd *);
nd * insert_pos( nd *);
nd * delete_front( nd * );

int main( ) {
     nd * first = NULL;   int ch;
   for(; ;)   {
        printf("1. Insert front\n2. Display\n3. Insert end\n");
       printf("4. Insert at position\n5. Delete Front\n6. Exit\n");
        printf("Enter choice: "); scanf("%d",&ch);
        switch(ch)   {
            case 1: first = insert_front(first); break;
           case 2: display(first); break;
          case 3: first = insert_end(first); break;
          case 4: first = insert_pos(first); break;
          case 5: first = delete_front(first); break;
          case 6: return 0;
        }
  }  }

nd *   insert_front( nd * f)  {
    nd  *t;
    t = (nd *) malloc(sizeof(nd));
printf("Enter data\n");
scanf("%d", &(t->data));
   t->link=0;

   if    (f  != NULL)
        t->link = f;
 return t;      }

void display( nd * f)  {
                if   ( f  ==  NULL)    {
                  printf("LL is empty\n");
                 return;
              }
              printf("Contents of list are \n");
              for( ;  f != NULL; )   {
                     printf("%d\n", f->data);
                     f = f -> link;
              }
}

nd *    insert_end (  nd * f) {
       nd * last;       nd * t = (nd *) malloc(sizeof(nd));
       t->link = NULL;
      printf("Enter data\n");
      scanf("%d", &(t->data));

       if ( f == NULL)
```

```
        return t;

      for( last = f;  last->link != NULL; last = last->link);

      last->link = t;
     return f;     //not at all modified
}
```

***nd *  insert_pos ( nd * f )***

```
{
   int  cnt=0, pos;
   nd *t, *p, *n ;
   printf("Enter position value\n");  scanf("%d", &pos);

   t = (nd *) malloc(sizeof(nd));     t->link = NULL;
   printf("Enter data\n");     scanf("%d", &(t->data));

   if ( f  == NULL  && pos == 1)
     return t;

    if ( f != NULL && pos == 1)
   {
      t->link = f;
      f=t;
      return f;
   }
```

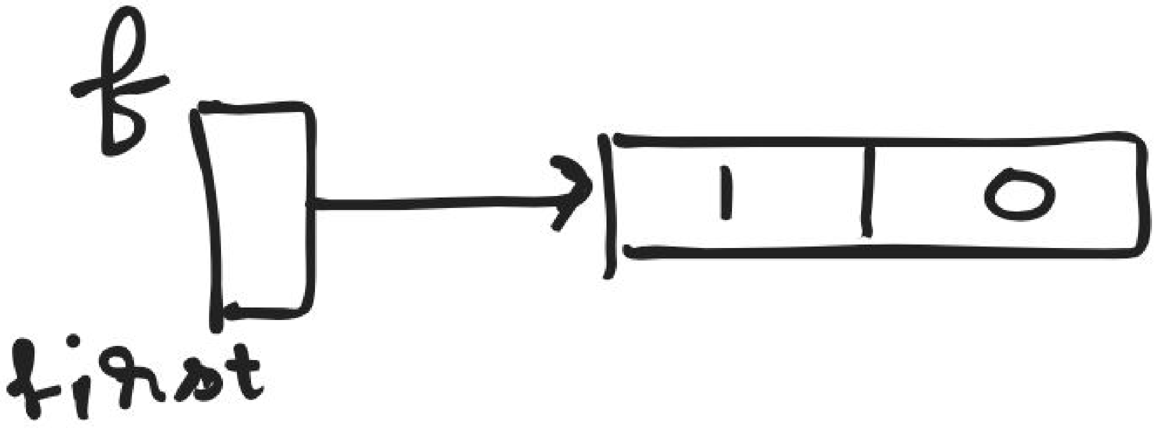
```
 for(p=f;  p != 0; p = p->link, cnt++);

  if (  f != NULL && pos == cnt+1)
  {
    for(p=f; p->link != NULL; p=p->link);
    p->link = t;
    return f;
  }
 if ( pos > cnt+1)
  {
    printf("Insertion not possible\n");
    free(t);
    return f;
  }
  pos--;

 for( p=0, n=f;  pos>0; pos--, p=n, n=n->link);

    p->link = t;
    t->link = n;
   return f;
}

nd *   delete_front ( nd * f )
{
    nd * t;
   if    (   f == NULL)
  {
    printf("LL is empty\n");
   return   f;
 }

   printf("Element deleted is %d\n", f->data);
  if  ( f->link == NULL)
 {
    free( f );
   return 0;
}
 t=f;     f=f->link;
free(t);
return f;     }
```

**5. Delete a node at the end of the list**

Situations to handle

1. LL is empty and no nodes to delete
2. LL contains **only one** node.
3. LL contains **two or more** nodes.

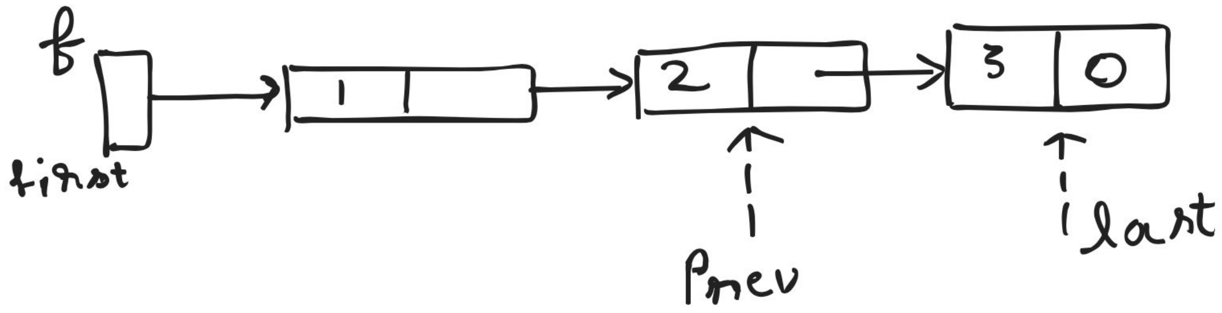
Consider the 2nd situation when only one node exists in LL.



Data part of the node to be deleted must be displayed and the node's memory must be freed for further usage.

```
nd *  delete_end( nd * f )
{
          if  ( f == NULL) // situation 1
         {
           printf("LL is empty\n");
          return f;  // return 0;
         }

        if  ( f->link == NULL) // situation 2
       {
           printf("Information deleted is %d\n", f->data);
          free(f);
         return 0;
      }
}
```

Consider, the 3rd situation where 2 or more nodes exist in LL



In order to delete the last node two pointers are required *prev* and *last*.
Data part of the node pointed by *last* has to be displayed

Link part of the node pointed by *prev* has to be set to NULL.

```
nd *  delete_end( nd * f )
{
            nd  *prev, *last;
           if  ( f == NULL) // situation 1
          {
            printf("LL is empty\n");
           return f;
          }

         if  ( f->link == NULL) // situation 2
        {
            printf("Element deleted is %d\n", f->data);
           free(f);
          return 0;
       }
       //situation 3
for( prev=0, last=f;   last->link != 0; prev=last, last=last->link);
       printf("Element deleted is %d\n", last->data);
       free(last);   prev->link = 0;
      return f;
}
```

**FULL PROGRAM USING *delete_end( )* FUNCTION**

```
struct  node {
       int data;
       struct node * link;  };
typedef struct node nd;
```

```
nd *  insert_front( nd * );
void  display( nd * );
nd * insert_end( nd *);
nd * insert_pos( nd *);
nd * delete_front( nd * );
nd * delete_end(nd *);
int main( ) {
    nd * first = NULL;   int ch;
   for(; ;)   {
       printf("1. Insert front\n2. Display\n3. Insert end\n");
      printf("4. Insert at position\n5. Delete Front\n");
     printf("6. Delete End\n7. Exit\n");
       printf("Enter choice: "); scanf("%d",&ch);
       switch(ch)   {
           case 1: first = insert_front(first); break;
          case 2: display(first); break;
         case 3: first = insert_end(first); break;
         case 4: first = insert_pos(first); break;
         case 5: first = delete_front(first); break;
         case 6: first = delete_end(first); break;
         case 7: return 0;
       }
  }  }

nd *   insert_front( nd * f)  {
   nd  *t;
   t = (nd *) malloc(sizeof(nd));
printf("Enter data\n");
```

```
scanf("%d", &(t->data));
   t->link=0;

   if    (f  != NULL)
        t->link = f;
 return t;      }

void display( nd * f)  {
               if   ( f  ==  NULL)    {
                 printf("LL is empty\n");
                 return;
             }
             printf("Contents of list are \n");
             for( ;  f != NULL; )    {
                     printf("%d\n", f->data);
                     f = f -> link;
             }
}

nd *    insert_end (  nd * f) {
        nd * last;       nd * t = (nd *) malloc(sizeof(nd));
        t->link = NULL;
       printf("Enter data\n");
       scanf("%d", &(t->data));

        if (  f == NULL)
          return t;

       for( last = f;  last->link != NULL; last = last->link);
```

```
      last->link = t;
     return f;     //not at all modified
}

nd *  insert_pos ( nd * f )
{
   int  cnt=0, pos;
   nd *t, *p, *n ;
   printf("Enter position value\n");  scanf("%d", &pos);

   t = (nd *) malloc(sizeof(nd));     t->link = NULL;
   printf("Enter data\n");     scanf("%d", &(t->data));

   if (  f  == NULL  && pos == 1)
     return t;

    if ( f != NULL && pos == 1)
   {
      t->link = f;
      f=t;
      return f;
   }
   if ( pos > cnt+1)
  {
    printf("Insertion not possible\n");
    free(t);
    return f;
  }
```

```
 pos--;
  for(p=f;  p != 0; p = p->link, cnt++);

 if ( f != NULL && pos == cnt+1)
 {
   for(p=f; p->link != NULL; p=p->link);
   p->link = t;
   return f;
 }

for( p=0, n=f;  pos>0; pos--, p=n, n=n->link);

   p->link = t;
   t->link = n;
  return f;
}

nd *   delete_front ( nd * f )
{
   nd * t;
  if    (   f == NULL)
 {
   printf("LL is empty\n");
   return   f;
 }

  printf("Element deleted is %d\n", f->data);
 if  (  f->link == NULL)
 {
   free( f );
   return 0;
}
 t=f;     f=f->link;
free(t);
return f;    }
```

```
nd *  delete_end( nd * f )
{
            nd  *prev, *last;
           if  ( f == NULL) // situation 1
          {
            printf("LL is empty\n");
           return f;
          }

         if  ( f->link == NULL) // situation 2
        {
            printf("Element deleted is %d\n", f->data);
           free(f);
          return 0;
        }
        //situation 3
for( prev=0, last=f;   last->link != 0; prev=last, last=last->link);
       printf("Element deleted is %d\n", last->data);
      free(last);   prev->link = 0;
     return f;
}
```

**6. Delete a node at given position**

Situations to handle

1. LL is empty
2. LL is having at least one node and position value will be in the range pos=1 and pos<=cnt.

Template of *delete_pos( )*

```
nd *  delete_pos(  nd * f)
{
 ......
}
```

1. LL is empty

```
nd *  delete_pos(  nd * f)
{
    if  ( f == NULL)
   { printf("LL is empty\n");  return f; }
}
```

2. LL is having at least one node and position value will be in the range pos=1 and pos<=cnt.

```
nd *  delete_pos(  nd * f)
{
    nd *t, *n, *p;     int cnt,pos;
    if  ( f == NULL)
   { printf("LL is empty\n");  return f; }
```

```
  // counting no of ondes in LL
 for(cnt=0,t=f;  t!=0;   t=t->link, cnt++);

 printf("Enter pos value\n"); scanf("%d",&pos);
 if (  pos > cnt)
 {  printf("Wrong pos value\n"); return f; }

pos--;
for(p=0,n=f;   pos>0;  pos--, p=n, n=n->link);

if ( p == 0 )  // first node deletion
{
     printf("Element to be deleted is %d\n",n->data);
    if ( n->link == 0)  // only one node in LL
    {free(f);  return 0;}

    // if LL contains more than one node
    f = n->link;  free(n);
   return  f;
}

printf("Element to be deleted is %d\n",n->data);
if ( n->link == 0)  // if n is pointing to the last node
{  p->link = 0 ;  free(n); return f;  }

p->link = n->link;  free(n);
return f;    }
```

**FULL program using *delete_pos( )* function**

```
struct  node {
        int data;
        struct node * link;  };
typedef struct node nd;

nd *  insert_front( nd * );
void  display( nd * );
nd * insert_end( nd *);
nd * insert_pos( nd *);
nd * delete_front( nd * );
nd * delete_end(nd *);
nd * delete_pos(nd *);
int main( ) {
     nd * first = NULL;   int ch;
   for(; ;)   {
        printf("1. Insert front\n2. Display\n3. Insert end\n");
       printf("4. Insert at position\n5. Delete Front\n");
      printf("6. Delete End\n7. Delete at position\n8. Exit\n");
        printf("Enter choice: "); scanf("%d",&ch);
        switch(ch)   {
            case 1: first = insert_front(first); break;
           case 2: display(first); break;
          case 3: first = insert_end(first); break;
          case 4: first = insert_pos(first); break;
          case 5: first = delete_front(first); break;
          case 6: first = delete_end(first); break;
          case 7: first = delete_pos(first); break;
          Case 8: return 0;
       }
  }  }
nd *  delete_pos(  nd * f)
{
    nd *t, *n, *p;     int cnt,pos;
    if  ( f == NULL)
   { printf("LL is empty\n");  return f; }

  // counting no of ondes in LL
 for(cnt=0,t=f;  t!=0;   t=t->link, cnt++);

 printf("Enter pos value\n"); scanf("%d",&pos);
 if (  pos > cnt)
 {  printf("Wrong pos value\n"); return f; }

pos--;
for(p=0,n=f;   pos>0;  pos--, p=n, n=n->link);

if ( p == 0 )  // first node deletion
{
     printf("Element to be deleted is %d\n",n->data);
    if ( n->link == 0)  // only one node in LL
    {free(f);  return 0;}

    // if LL contains more than one node
    f = n->link;  free(n);
   return  f;
}
```

```
 printf("Element to be deleted is %d\n",n->data);
if ( n->link == 0)  // if n is pointing to the last node
{  p->link = 0 ;  free(n); return f;  }

p->link = n->link;  free(n);
return f;    }



nd *    insert_front( nd * f)  {
    nd  *t;
    t = (nd *) malloc(sizeof(nd));
printf("Enter data\n");
scanf("%d", &(t->data));
   t->link=0;

   if    (f  != NULL)
        t->link = f;
 return t;      }

void display( nd * f)  {
                 if   ( f  ==  NULL)    {
                  printf("LL is empty\n");
                  return;
                }
                printf("Contents of list are \n");
                for( ;  f != NULL; )   {
                        printf("%d\n", f->data);
                        f = f -> link;
                }
```

```
}

nd *   insert_end (  nd * f) {
       nd * last;       nd * t = (nd *) malloc(sizeof(nd));
       t->link = NULL;
      printf("Enter data\n");
      scanf("%d", &(t->data));

       if (  f == NULL)
         return t;

      for( last = f;  last->link != NULL; last = last->link);

      last->link = t;
     return f;     //not at all modified
}

nd *  insert_pos ( nd * f )
{
   int  cnt=0, pos;
   nd *t, *p, *n ;
   printf("Enter position value\n");  scanf("%d", &pos);

   t = (nd *) malloc(sizeof(nd));     t->link = NULL;
   printf("Enter data\n");     scanf("%d", &(t->data));

   if (  f  == NULL  && pos == 1)
     return t;
    if ( f != NULL && pos == 1)
   {
      t->link = f;
      f=t;
      return f;
   }
   if ( pos > cnt+1)
  {
    printf("Insertion not possible\n");
   free(t);
   return f;
  }
  pos--;
   for(p=f;  p != 0; p = p->link, cnt++);

   if (  f != NULL && pos == cnt+1)
   {
     for(p=f; p->link != NULL; p=p->link);
   p->link = t;
   return f;
   }

for( p=0, n=f;  pos>0; pos--, p=n, n=n->link);

   p->link = t;
   t->link = n;
   return f;
}
```

```
nd *   delete_front ( nd * f )
{
    nd * t;
   if    (  f == NULL)
  {
    printf("LL is empty\n");
    return   f;
 }

   printf("Element deleted is %d\n", f->data);
  if  ( f->link == NULL)
 {
    free( f );
   return 0;
}
 t=f;      f=f->link;
free(t);
return f;     }


nd *  delete_end( nd * f )
{
             nd  *prev, *last;
            if  ( f == NULL) // situation 1
           {
             printf("LL is empty\n");
            return f;
          }
```

```
        if  ( f->link == NULL) // situation 2
        {
            printf("Element deleted is %d\n", f->data);
           free(f);
          return 0;
        }
        //situation 3
for( prev=0, last=f;   last->link != 0; prev=last, last=last->link);
        printf("Element deleted is %d\n", last->data);
       free(last);   prev->link = 0;
      return f;
}
```

**LAb 7**

**Design, Develop and Implement a menu driven Program in C for the following operations on Singly Linked List (SLL) of Student Data with the fields: USN, Name, Programme, Sem, PhNo.**

**a. Create a SLL of N Students Data by using front insertion.**
**b. Display the status of SLL and count the number of nodes in it**
**c. Perform Insertion / Deletion at End of SLL**
**d. Perform Insertion / Deletion at Front of SLL(Demonstration of stack)**
**e. Exit**

**1. Insert front**
**2. Insert end**
**3. Delete front**
**4. Delete rear**
**5. Display and count**
**6. exit**

Linked stack : rename first pointer in main as top,
Functions needed are insert_front( ),
delete_front( ) and display().

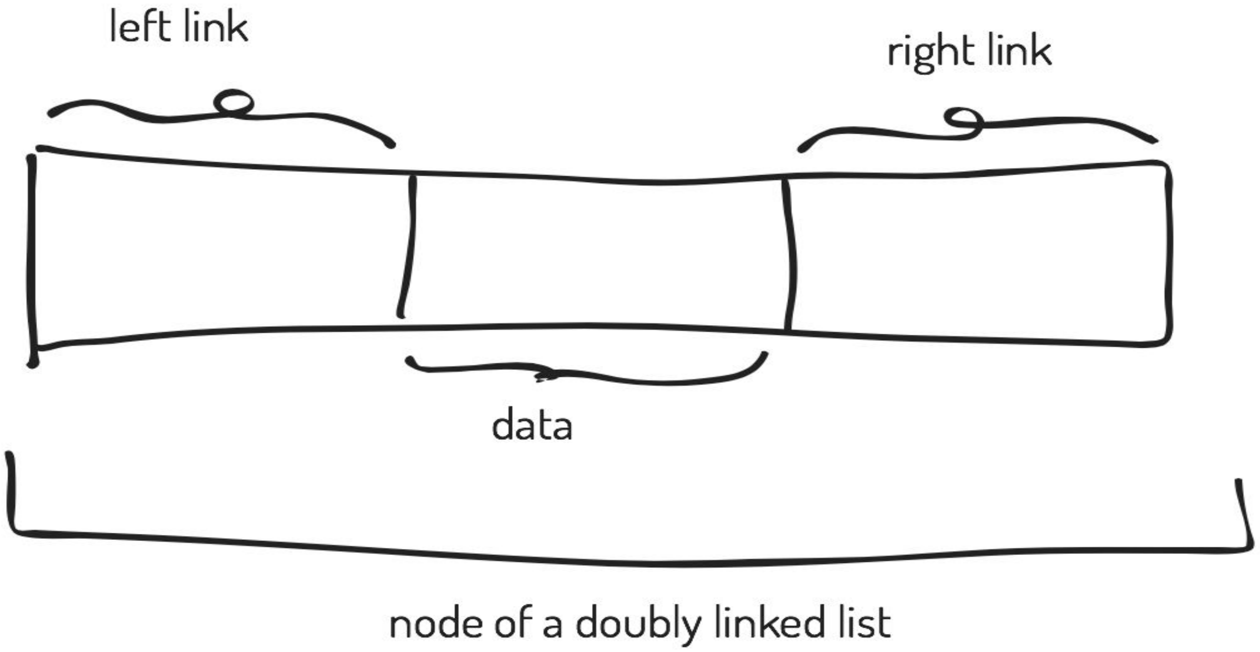
Linked queue: two pointers required front and rear.
Functions needed are insert_rear( ), delete_front( )

```
struct node {
   int data;  struct node *link; };
typedef  struct node nd;

nd *  insert_rear(nd *);
nd * delete_front(nd *);

int  main( ){
  nd *r=0, *f=0; int ch=0;
    for(;;)
    {
  printf("1. Insert rear\n2. Delete front\n3. Display\n4. Exit\n");
    printf("choice : "); scanf("%d",&ch);
    switch(ch) {
       case 1:  r = insert_rear(r); if (f==0) f=r; break;
       case 2:  f = delete_front(f); break;
       case 3: display(f); break;
       case 4: exit(0);
     }
     }
}
nd *  insert_rear(nd * r)
{
   nd *t;
   // creating a new node
   if (r == 0)
      return t;
   r->link = t;
   return t; }

nd *  delete_front( nd *f)
{
  nd* t;
  if ( f == NULL)
   { printf("queue is empty\n"); return 0; }

 printf("Element delete is %d\n", f->data);
 t=f->link;   free(f);
 return t;
}
```

**ASSIGNMENT**
**Write a program to maintain employee information which is made up of e-no, e-name, salary and designation in a linked list data structure.**
**Support the program with a function search, to search the employee based on name.**
**Print search successful else search unsuccessful.**

## DOUBLY LINKED LIST

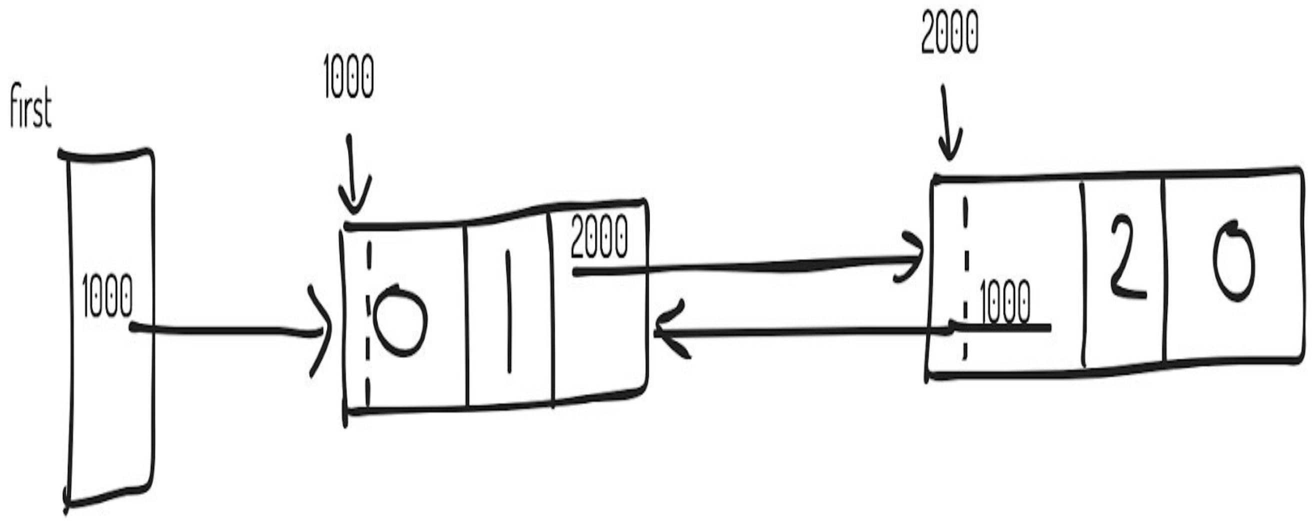
Doubly linked list is a structure made up of two links in a node.



```
struct node {
   struct node * llink;
  int data;
 struct node * rlink;
};
typedef struct node nd;
printf("%d", sizeof(nd));     //12
printf("%d", sizeof(nd*) );  // 4/8
```

*leftlink* is used to point to the previous node (wrt the current node) if any or it will be set to NULL.

*rightlink* is used to point to the next node (wrt the current node) if any or it will be set to NULL.



Creation of nodes will be similar to SLL, only extra part to be dealt with, in coding is leftlink.

*first* pointer in DLL holds the address of the first node.

**1. Insert front function for DLL**

Situation to be dealt with

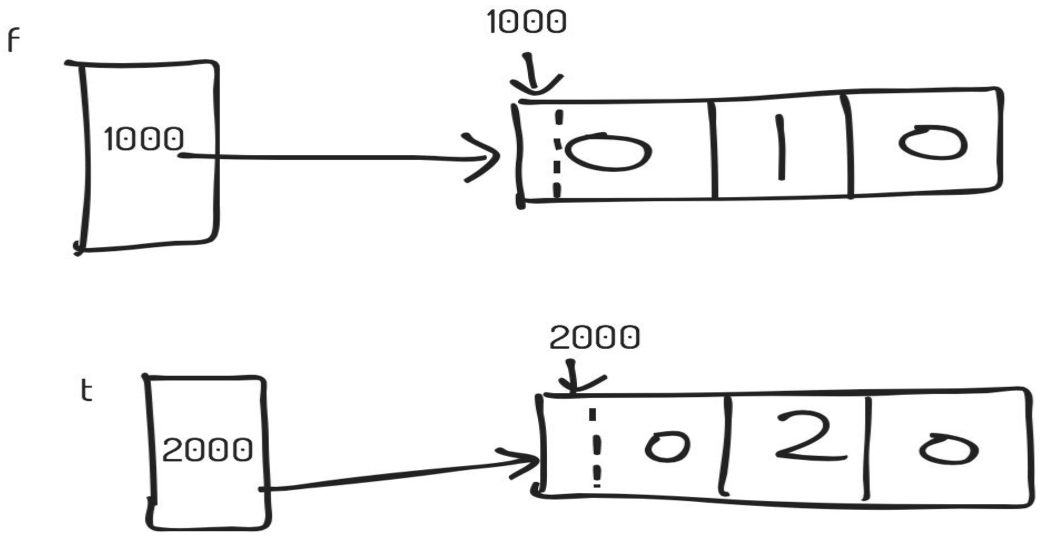
1. If DLL is empty (if first == 0)
2. If DLL is pointing to at least one node. (if first != NULL)

```
int   main( ) {
   nd * first=NULL;
}

nd* insert_front(nd * f)
{
  nd *t= (nd *) malloc(sizeof(nd));
  scanf("%d",&(t->data));
```

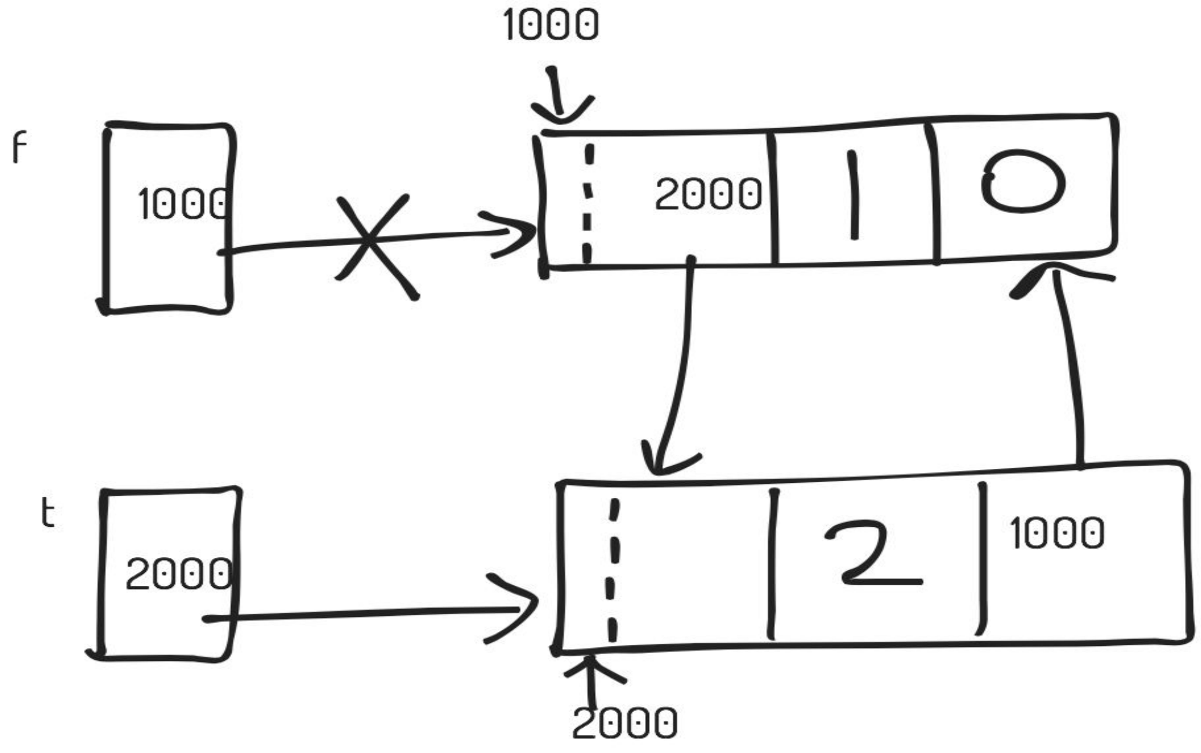
```
 t->llink = t->rlink = 0;

if  ( f == NULL) //situation 1.
     return t;    }
```

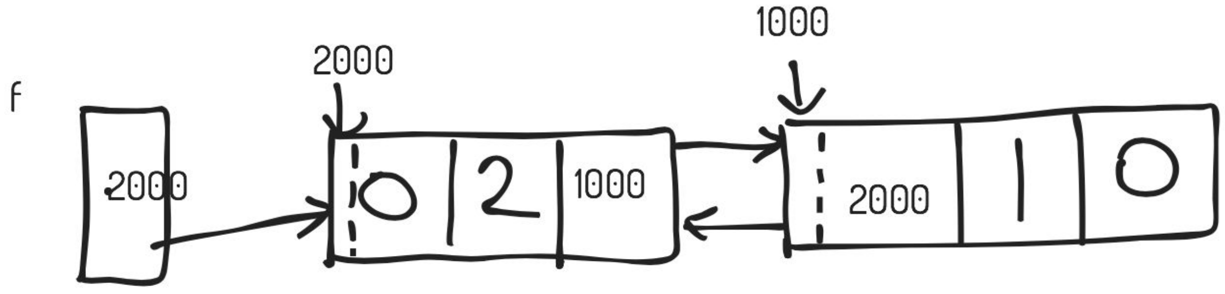
**Situation 2**

Before inserting a node



After inserting a node at front end





**Code part**

```
nd* insert_front(nd * f)
{
   nd *t= (nd *) malloc(sizeof(nd));
   scanf("%d", &(t->data));
   t->llink = t->rlink = 0;

  if   ( f == NULL) //situation 1.
       return t;
  t->rlink=f;  f->llink=t;
 return t;
}

int  main() {
  nd *first=0;
  first=insert_front(first);     first=insert_front(first);
  printf("%d\n",first->data);  first = first->rlink;
    printf("%d\n",first->data);
}
```

**2. To display data part of the nodes in DLL**

In *display( )* nodes information will be displayed from first-to-last and from last-to-first.

i.e traversing the nodes from first-to-last and from last-to-first.

```
void display( nd * f)
{
          if ( f == NULL)
          {  printf("DLL is empty\n"); return; }

         printf("Contents of DLL from FIRST to LAST");
        for( ;  f->rlink != NULL; f=f->rlink)
          printf("%d\n",f->data);

        printf("%d\n",f->data);
        // to print the last node's information
        printf("Contents of DLL from  LAST to FIRST");
       for( ;  f->llink != NULL; f=f->llink)
          printf("%d\n",f->data);

       printf("%d\n",f->data);
}
```

**FULL PROGRAM using insert_front() and display()**

```
# include <stdio.h>
#include <stdlib.h>
struct node {
    struct node * llink;
   int data;
  struct node * rlink;
};
typedef struct node nd;


nd * insert_front(nd *);
void display(nd *);

int main( ) {
   nd * first = NULL;   int ch;
  for(;;) {
                printf("1. Insert front\n2. Display\n3. Exit\n");
               printf("Choice "); scanf("%d", &ch);
              switch(ch) {
                case 1:   first = insert_front(first); break;
               case 2:  display(first); break;
              case 3:  return(0);
    }
  }   }

nd* insert_front(nd * f)   {
   nd *t= (nd *) malloc(sizeof(nd));
   scanf("%d",&(t->data));
   t->llink = t->rlink = 0;

  if   ( f == NULL) //situation 1.
       return t;

  t->rlink=f;  f->llink=t;
 return t;
}

void display( nd * f)    {
          if ( f == NULL)
          {  printf("DLL is empty\n"); return; }

         printf("Contents of DLL from FIRST to LAST\n");
        for( ;  f->rlink != NULL; f=f->rlink)
          printf("%d\n",f->data);

        // to print the last nodes information
        printf("%d\n",f->data);
         printf("Contents of DLL from  LAST to FIRST\n");
        for( ;  f->llink != NULL; f=f->llink)
          printf("%d\n",f->data);

        printf("%d\n",f->data);
}
```

**3. Insert a node at the end of DLL**

Situations to handle

1. DLL is empty
2. DLL is pointing to at least one node

Template of *insert_end( )*

```
nd *  insert_end( nd *f)
{
            nd *last;
            nd *t= (nd *) malloc(sizeof(nd));
           scanf("%d",&(t->data));  t->llink = t->rlink = 0;

          if   ( f == NULL) //situation 1.
               return t;

        //situation 2
       for(last=f; last->rlink != NULL; last=last->rlink);

       last->rlink = t;
       t->llink = last;
       return f;
}
```

**4. Delete a node from the front end of DLL**

Situations to handle

1. If DLL is empty
2. If DLL is pointing to at least one node.
3. If DLL is pointing to more than one node.

```
nd *   delete_front( nd * f )
{
          nd * t;
         if ( f == NULL) // situation 1
         {  printf("DLL is empty\n"); return 0; }

         printf("Element deleted is %d\n",f->data);
        if ( f->rlink == NULL) //situation 2
       {
        free(f);  return 0;
      }

      //situation 3
      t=f;   f=f->rlink;
      free(t);     f->llink=0;
}
```

**5. Delete a node from the end of DLL**

**Situations to handle**

**1. If DLL is empty**

**2. If DLL is pointing to at least one node.**

**3. If DLL is pointing to more than one node.**

```
nd *  delete_end( nd * f) {
    nd * last,*prev;
   if (f == NULL)   // situation 1
   {
        printf("DLL is empty\n"); return 0; }

        if (f->rlink == NULL) // situation 2
       {
              printf("Element deleted is %d\n", f->data);
              free(f); return 0;
       }



       //situation 3
       for(last=f;  last->rlink!=0 ; last=last->rlink);

      printf("Element deleted is %d\n", last->data);
      prev = last->llink;
     free(last); prev->rlink = 0;
    return f;  }
```

*Insert at pos*
*Delete at pos*      ⇒ *student's work*