Linked List UNIT-III DS/4.03/30

Disadvantages of an Array

- Arrays have a fixed dimension. Once the size of an array is decided, it cannot be increased or decreased during execution.
- Array elements are always stored in contigous memory locations. At times it might so happen that enough contigous locations might not be available for the array that we are trying to create. Even though the total space requirement of the array can be met through a combination of non-contigous blocks of memory, we would still not be allowed to create the array.
- Operations like insertion of a new element in an array or deletion of an existing element from the array are pretty tedious. This is because during insertion or deletion each element after the specified position has to be shifted one position to the right (in case of insertion) or one position to the left (in case of deletion).

Such disadvantages can be easily overcomed through Linked List.

- A linked list can grow & shrink in size during its lifetime.
- There is no maximum size of a linked list.

- The nodes (elements) are stored are stored at different memory locations it hardly happens that we fall short of memory when required.

- Unlike arrays, while inserting or deleting the nodes of the linked list, shifting of nodes is not required.

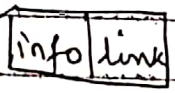
Linked List

A linked list is a data structure which is a collection of zero or more nodes where each node is connected to one or more nodes.

Singly Linked List

A linked list which contains zero or more nodes where each node has 2 fields

Node

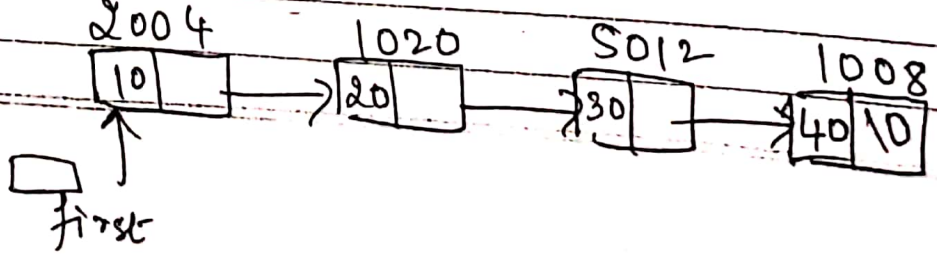


A Node in a list has 2 fields info, link

Info → This field is used to store the data

link → This field contains address of the next node.

eg:- For Singly linked list.



- The variable first contains the address of the first node
- Each node has 2 fields one is data i.e 10, 20, 30, 40 & the other one is link which contains the address of the next node.
- The last node's link field is NULL [indicating that it is last node]
- Observing the addresses of nodes i.e 2004, 1020, 5012, 1008, they are physically far apart, but using link field we can obtain each node in the list in the order specified & hence we say they are logically adjacent.

Empty list

If the variable first contains NULL then it is said as an empty list.

An empty linked list is a pointer variable which contains NULL.

Representing chains in C

The nodes in a linked list are self referential structures.

A self-referential structure is a structure which has at least one field which is pointer to same structure.

eg:-
```
Struct node
{
  int info;
  Struct node *link;
};
```

info - Integer variable which contains the information.

link - It is a pointer field & hence it should contain the address. It normally contains the address of the next node in the list.

```
Struct node
{
  int info;
  Struct node *link;
};

typedef Struct node *NODE;
```

A pointer variable first can be declared as

Method 1 : NODE first;

Method 2: struct node *first;

Now lets create a linked list, Before creating a linked list we create empty list - as -

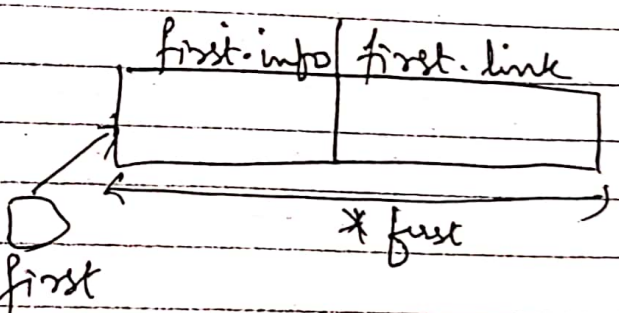
first = NULL

NULL
first

Step 1 - Create a node

A node which is identified by variable first with two fields : info & link. fields can be created using MALLOC()

MALLOC(first, 1, struct node);

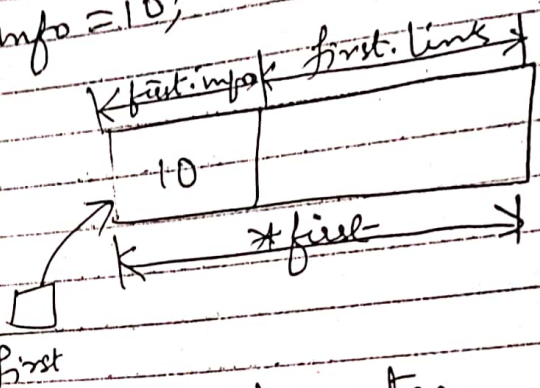


- Using variable first we can access the address of the node.
- Using *first we can access entire contents of node.
- Using (*first).info or (*first)->info we can access the data
- Using (*first).link or (*first)->link we can access the link field

Step 2: Store the data

The data 10 can be stored in the info field using -

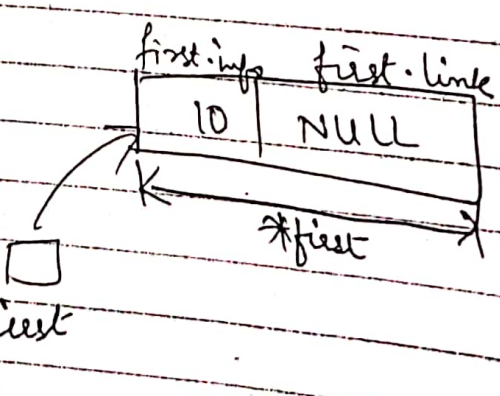
```
first -> info = 10;
      or
(*first).info = 10;
```



Step 3: Store NULL character.

After creating the above node, if we do not want link field to contain address of any other node, we can store NULL in link field

```
first -> link = NULL;
      or
first.link = NULL;
```

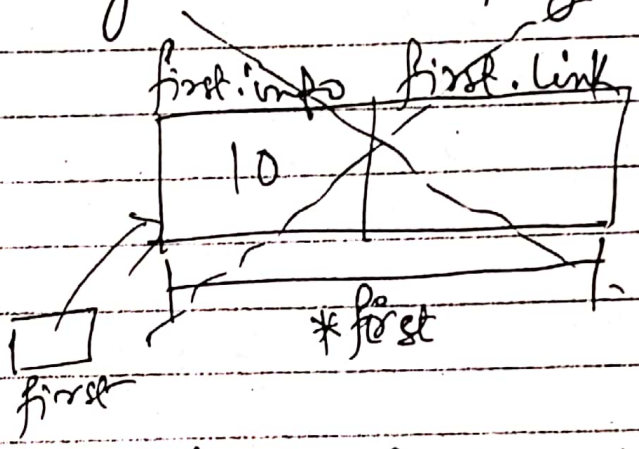


Step 4: Delete a node

A node which is no longer required can be deleted using free()

```
free(first);
```

when above statement is executed, the memory allocated for a node using malloc() with the help of macro MALLOC() will be de-allocated & returned to operating system so that it can be used by some other program.



Now the pointer variable does not contain valid address, hence first is now a dangling pointer.

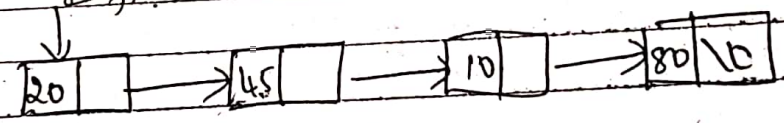
Operations on Singly linked lists

The basic operations of singly linked list are -

- Inserting a node into the list
- Deleting a node from the list
- Search in a list
- Display the contents of list

Inserting a node at the front end

Design - To design the function, let us consider a list with 4 nodes. Here, Pointer first contains address of the first node of the list as shown -

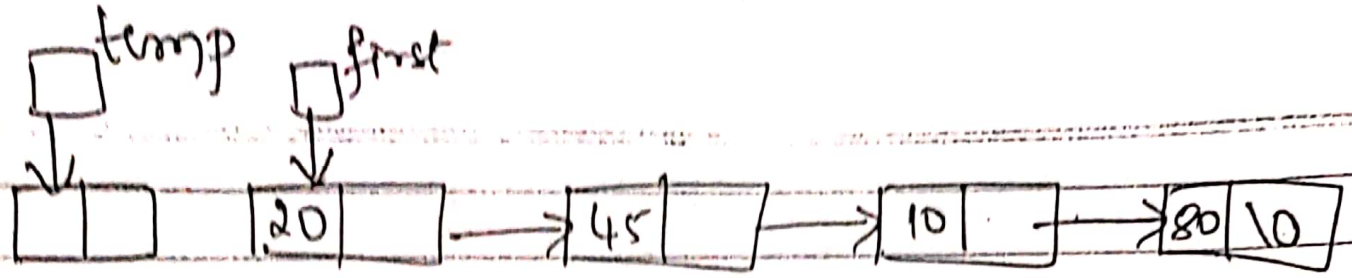


Let's insert 50 at the front end of the list. The sequence of steps to be followed -

Step 1: Allocate memory for a new node using malloc() function with the help of macro MALLOC() & use the variable temp to store the address of the node using -

MALLOC(temp, 1, struct node);

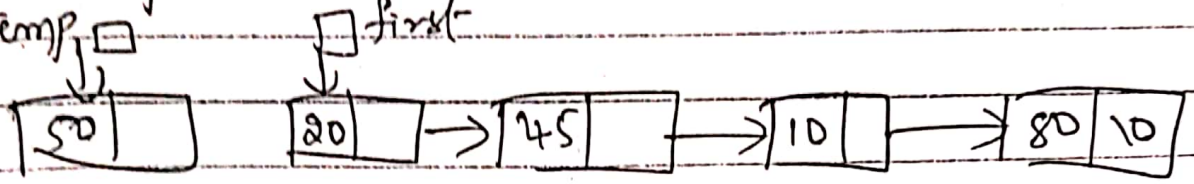
The pictorial representation after executing the above statement



Step 2: Copy the item 50 into info field of temp using the following statement -

```
temp -> info = item;
```

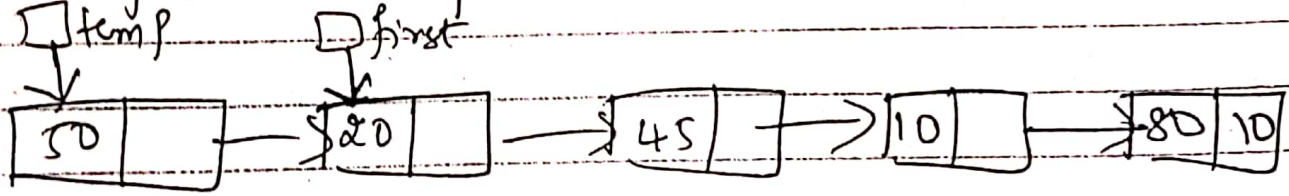
The pictorial statement is as shown below



Step 3: Copy the address of the first node of the list in pointer variable first into link field of temp.

```
temp -> link = first;
```

The pictorial representation is as shown -



Step 4: Now, a node temp has been inserted & observe from the fig. that temp is the first node. Let us always return address of the first node using the statement.

```
return temp;
```

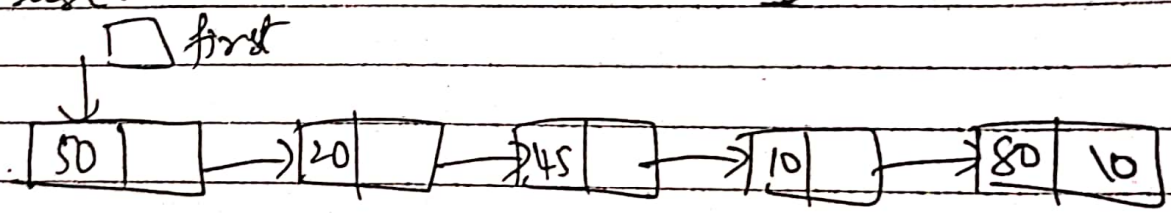
C - function to insert an item at the front end of the list

```
NODE insert_front (int item, NODE first)
{
    NODE temp;
    MALLOC (temp, 1, struct node);
    temp -> info = item;
    temp -> link = first;
    return temp;
}
```

The above function should be called in the calling function -

```
first = insert_front (item, first);
```

After executing the above statement, the variable first in the calling function contains address of the first node of the list.



The algo insert-front() has been designed by assuming that the list already exists.

Now lets assume that list is empty i.e., the pointer variable first contains NULL. Now try to insert an item into the empty list & see that function insert front works

for this case.

Creating a linked list -

first = insert_front(item, first);

If first is NULL & the above statement is executed, a linked list with only one node is created.

If it is executed for the second time, a new node is inserted at the front end & there by number of nodes in the list is 2 & so on.

Thus repeated inserting an element at the front end of the list we can create a linked list.

Display singly linked list

The contents of the list can be displayed using various cases -

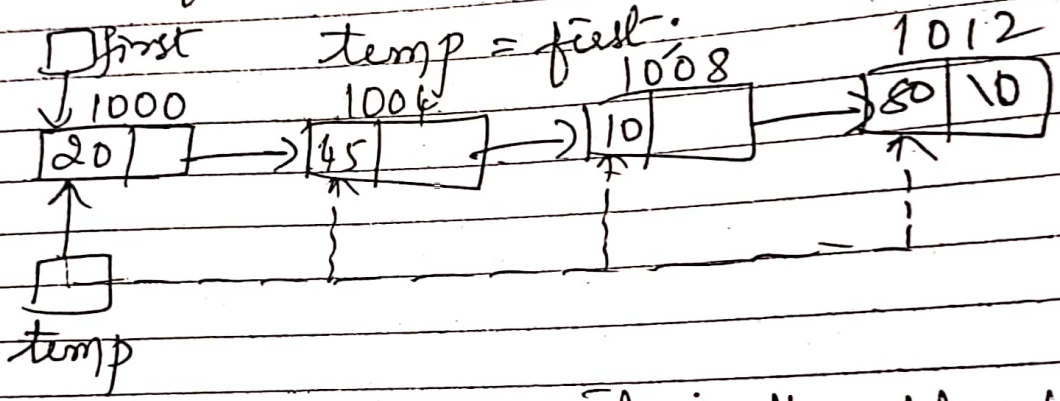
Case 1: List is empty

If the list is empty then the proper message has to be displayed

```
If (first == Null)
{
    printf("List is empty");
    return;
}
```

Case 2: list is existing

Consider the list with four nodes. The variable first always contains address of the first node of the list. So for displaying the contents of list instead of using variable first, let us use another variable temp. In the begining the variable temp also should contains the address of the first node of the list.



temp -> link contains the address 1004. If we copy this value to temp, now temp has the address 1004 which is the address of the second node.

```
temp = temp -> link;
```

We can point temp to next node, Now the items 20, 45, 10 & 80 has to be displayed. This is achieved by repeatedly printing info field of temp & updating temp to point to next node as shown—

```
printf("%d", temp -> info);
temp = temp -> link;
```

After displaying so, note that temp points to NULL indicating that all the nodes in the list are over.

As long as temp is not NULL, repeat the steps.

```
temp = first;
while (temp != NULL)
{
printf("%d", temp->info);
temp = temp->link;
}
```

Other operations on linked list

Delete a node from the front end

A node from the front end can be deleted using various cases-

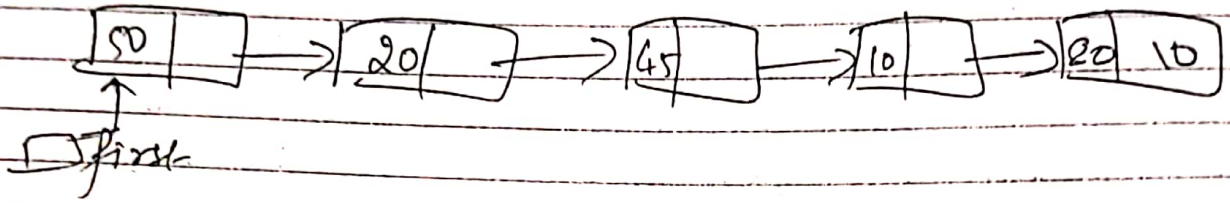
Case 1 : List is Empty

If the list is empty, it is not possible to delete the node, but appropriate message will be displayed as -

```
if (first == NULL) {
printf("List is empty");
return;
}
```

Case 2: List is existing

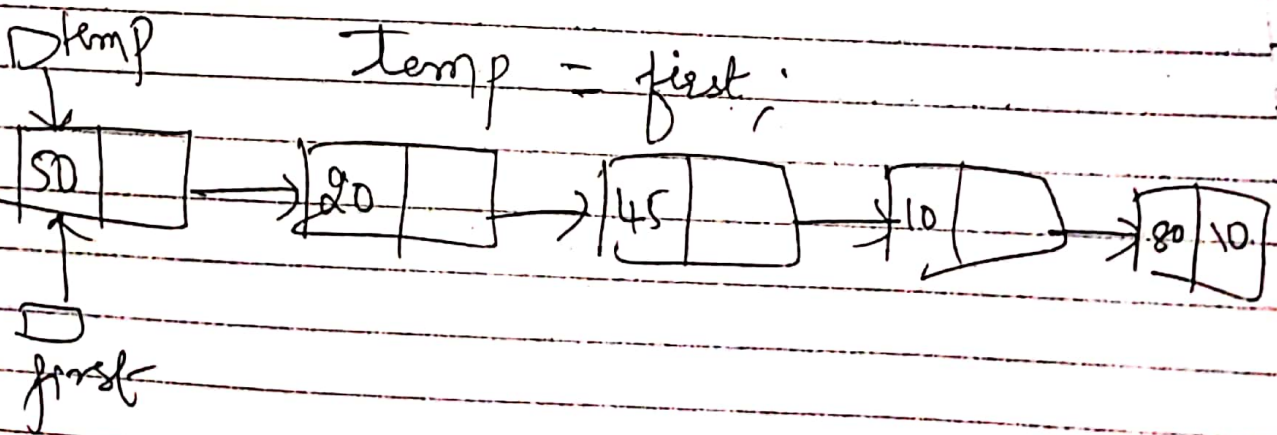
Consider the list with five nodes where the variable first contains address of the first node of the list -



Given the address of the first node of the list, we need to know the address of the second node of the list. This is because, after deleting the first node, the second node will be the first node of the list.

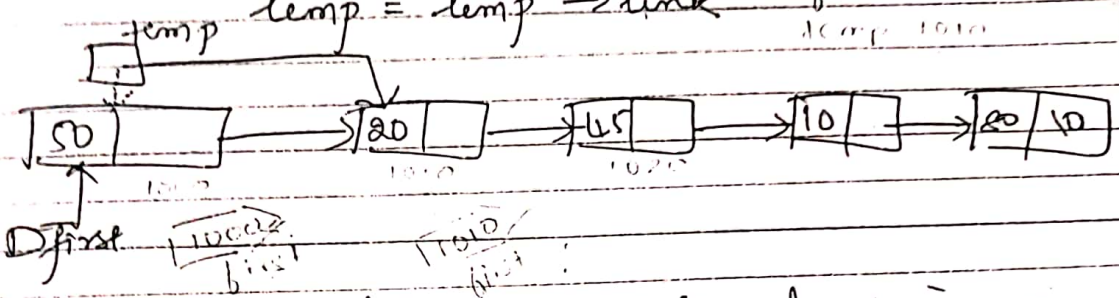
The sequence of steps to be followed while deleting an item -

Step 1 - Use a pointer variable temp & store the address of the first node of the list using the statement



Step 2: Update the pointer temp so that the variable temp contains address of the second node using the statement -

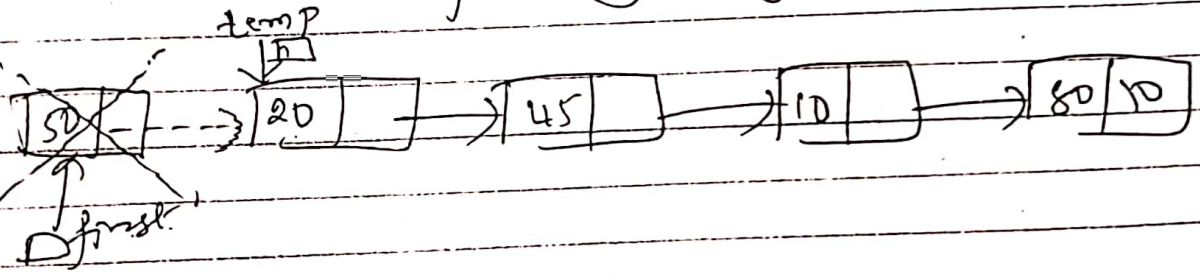
temp = temp → link



Step 3 :- Now delete the first node using -

free(first);

The node pointed to by first is deleted & is returned to operating system.



Step 4: Once the node first is deleted, the node temp will become the first node, so return temp as the first node to the calling function using the statement

return temp;

Now the final code is

```
temp = first;
temp = temp -> link
free(first);
return temp;
```

C - function to delete an item from the front end

```
NODE deletefront(NODE first)
{
  NODE temp;
  if (first == NULL)
  {
    printf("List is empty");
    return first;
  }
  temp = first;
  temp = temp -> link
  printf("Item deleted = %d", first->info);
  free(first);
  return temp;
}
```

Insert a node at the rear end

Implementation of queues using linked list

Insertion is done from one end & deletion from other end, so

C - program to implement queues using linked list

```
#include<stdio.h>
#include<stdlib.h>
#define MALLOC(p,n,type)
```

```
struct node
{
  int info;
  struct node *link;
};
typedef struct node *NODE;
#define MALLOC(P,n,type) \
P=(type *)malloc(n*sizeof(type)); \
if(P==NULL) \
{ \
  printf("Insufficient memory"); \
  exit(0); \
}
void display(NODE first)
{
  NODE temp;
  if(first==NULL)
  {
    printf("List is empty");
    return;
  }
  temp=first;
  while(temp!=NULL)
  {
    printf("%d", temp->info);
    temp=temp->link;
  }
```

```
NODE delete_front(NODE first)
{
 NODE temp;
 if(first==NULL)
  {
  printf("List is empty");
  return list;
  }
 temp= first
 temp = temp -> link
 printf("Item deleted = %d", first->info);
  free(first);
  return temp;
}
NODE insert_rear(int item, NODE first)
{
  NODE temp;
  NODE cur;
  MALLOC(temp, 1, struct node)
  temp->info = item;
  temp -> link = NULL;

  if(first == NULL) return temp;

  cur = first;

  while (cur -> link != NULL)
    {
      cur = cur -> link;
    }
    cur -> link = temp;
  return first;
}
```

```
Void main()
{
  NODE first = NULL;
  int choice, item;
  for(;;)
  {
    printf("1: Insert Rear  2: Delete front");
    printf("3: Display  4: Exit");
    printf("Enter the choice\n");
    scanf("%d", &choice);
    switch(choice)
    {
      case 1:
        printf("Enter the item to be inserted");
        scanf("%d", &item);
        first = insert_rear(item, first);
        break;
      case 2:
        first = delete_front(first);
        break;
      case 3:
        display(first);
        break;
      default:
        exit(0);
    }
  }
}
```

Implementation of Stacks using linked list

Insertion & deletion is done from same end so we can utilize the functions

insert_rear()
delete_rear()
display()

OR

insert_front()
~~insert~~ delete_front()
display();

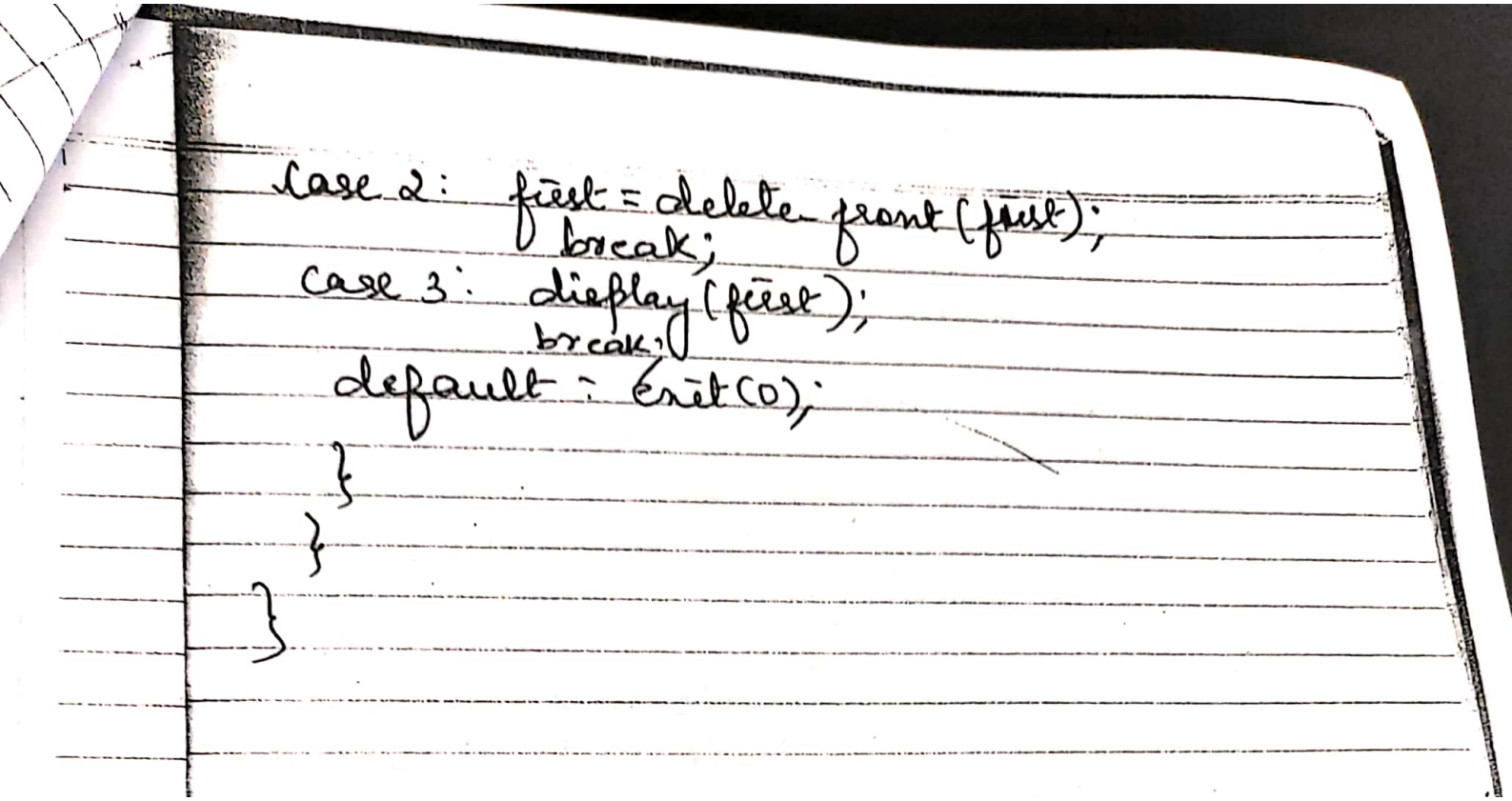
C - program

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

#define MALLOC(P, n, type)                    \
P = (type *) malloc(n * sizeof(type)); (
if(P == NULL)                                 (
{                                             \
printf("Insufficient memory\n");              \
exit(0);                                      \
}                                             \

NODE insert_front(int item, NODE first)
{
    NODE temp;
    MALLOC(temp, 1, struct node)
    temp -> info = item;
    temp -> link = first;
    return temp;
}
```

Implementation of Stacks using linked lists

```
void display (NODE first)
{
    NODE temp;
    if (first == NULL)
    {
        printf("List is empty");
        return;
    }
    temp = first;
    while (temp != NULL)
    {
        printf("%d", temp->info);
        temp = temp->link
    }
}

NODE delete_from (NODE first)
{
    NODE temp;
    if (first == NULL)
    {
        printf("List is empty cannot delete");
        return list;
    }
    temp = first
    temp = temp->link
```

```
    printf("Item deleted = %d", first->info);
    free(first);
    return temp;
}
Struct node
{
    int info;
    struct node *link;
};
typedef struct node *NODE;
Void main()
{
    int choice, item;
    NODE first = NULL;
    for(;;)
    {
        printf("1. Push 2. Pop 3. Display
                4. exit");
        printf("Enter the choice");
        Scanf("%d", &choice);
        Switch(choice)
        {
            Case 1: printf("enter the item to
                    be inserted");
                Scanf("%d", &item);
            first = insert_front(item, first);
            break;
```



```
case 2: first = delete_front(first);
        break;
case 3: display(first);
        break;
default : exit(0);
    }
  }
}
```

Doubly linked list

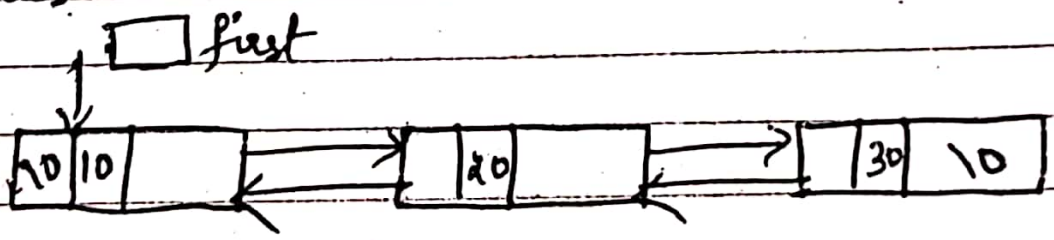
A doubly linked list is a linear collection of nodes where each node is divided into three parts -

Info - This is a field where the information is stored.

llink - This is a field (pointer field) which contains address of the left node or previous node in the list.

rlink - This is a pointer field which contains the address of the right node or next node in the list.

Using such lists it is possible to traverse the list in forward & backward directions. It is also called two way list.



The llink field of the leftmost node & rlink field of rightmost node points to NULL.

Additional list operations

The various additional operations are -

- → Find the length of the list.
- → Search for an item in a list.
- → Delete a node whose information field is specified.
- → Concatenate two lists.
- → Reverse (Invert) a list without creating new nodes.

Concatenate Two lists

Definition - Concatenation of two lists is nothing but joining the second list at the end of the first list.

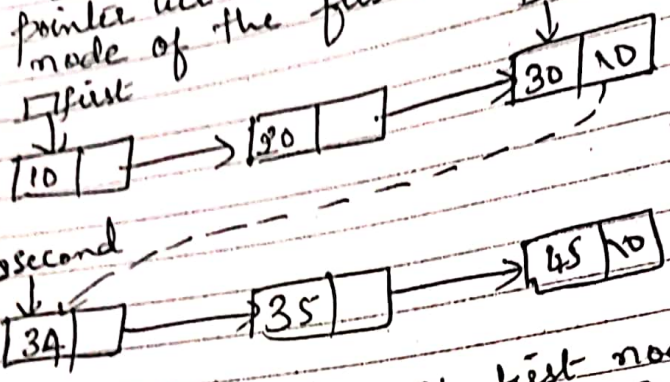
Concatenation of two lists is possible if two lists exists. If any one of the list is empty, return the address of the first node of the non empty list as shown -

```
if (first == NULL) return second;
if (first != second == NULL) return first;
```

If both list exists, obtain the address of the last node of the first list. The code to obtain address of the last node of the first list -

```
cur = first;
while(cur -> link != NULL)
cur = cur -> link;
```

Once control comes out of the loop, pointer cur contains address of the last node of the first list



Now, attach address of first node of the second list identified by second to cur node as shown is dotted line.

```
cur -> link = second
```

Now, first contains the address of the first node of the concatenated list & return this node to the calling function using -

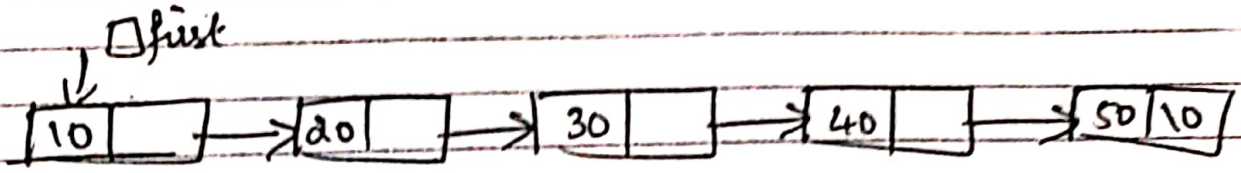
```
return first;
```

C - function

```
NODE Concat (NODE first, NODE sec)
{
  NODE cur;
  if (first == NULL) return second;
  if (second == Null) return first;
  cur = first;
  while (cur -> link != NULL)
    cur = cur -> link;
  cur -> link = sec;
  return first;
}
```

Reverse a list without creating a new nodes

Consider the list shown below -



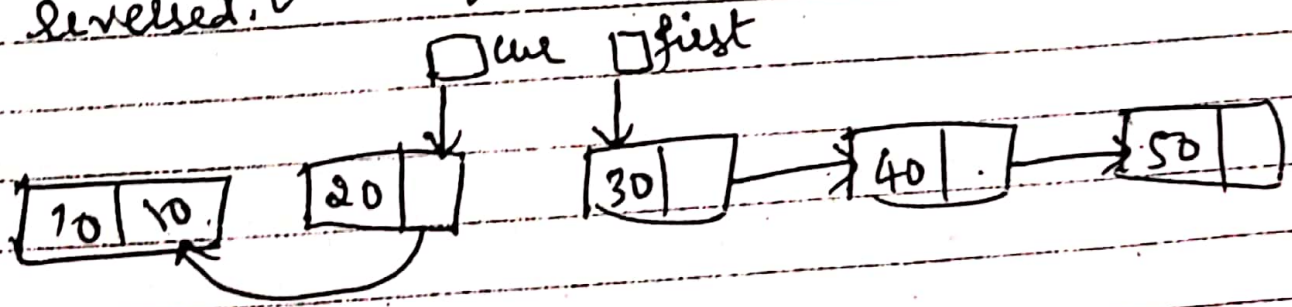
After reversing, the o/p should be

50 40 30 20 10

Design: Assume that given list is divided into 2 sub lists such that first list with 2 nodes is reversed & the 2nd list with 3 nodes is yet to be reversed

Now let us assume 2 things:

- The pointer variable cur always contains the address of the first node of the partially reversed list.
- The pointer variable first always contains address of the first node of the list to be reversed.

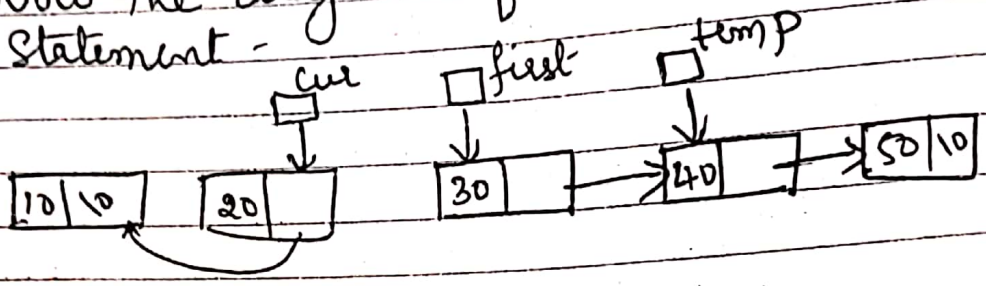


Now, to reverse the list the following steps are used -

Step 1:- obtain the address of the 2nd node of the list to be reversed. This can be done using:

```
temp = first -> link;
```

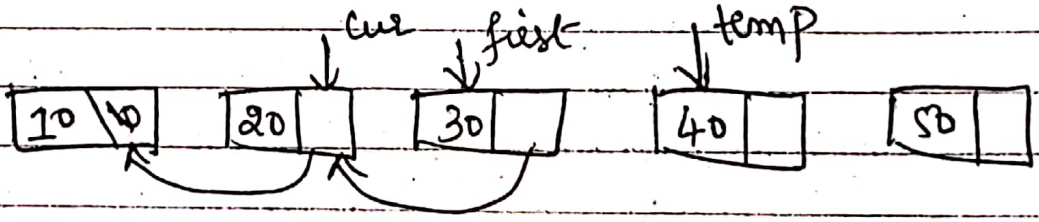
Now the diagram after executing the above statement -



Step 2: Attach the first node of the list to be reversed, to the front end of the partially reversed list. This can be achieved using

```
first -> link = cur;
```

After executing the above statement, the list can be written as -

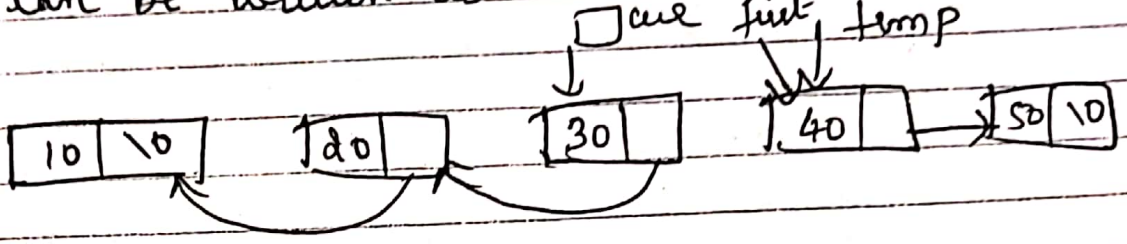


Step 3: The pointer variable cur always should contain address of the first node of the reversed list & first should contain address of the first node of the list to be reversed.

This can be done using

```
cur = first;
first = temp;
```

After executing the above statement, the list can be written as -



The steps 1 through 3 can be repeatedly performed as long as list to be reversed is existing. That is, as long as the pointer variable first is not NULL keep executing statements from step 1 to step 3, The code can be written as -

```
while (first != NULL)
{
    temp = first -> link;
    first -> link = cur;
    cur = first;
    first = temp;
}
```

C - function

```
NODE reverse (NODE first)
{
    NODE cur, temp;
    cur = NULL;          // initial reversed list
    while (first != NULL)
    {
```

```
    temp = first -> link;
    first -> link = cur;
    cur = first;
    first = temp;
  }
  return cur;
}
```

Circular Singly linked List

The various disadvantages of singly linked lists are -

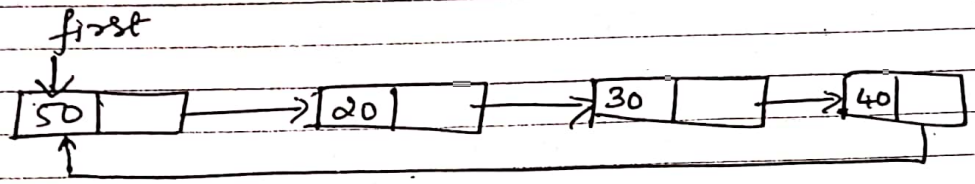
→ In a singly linked list, there is only one link field & hence traversing is done only in one direction. So, given the address of a node x, only those nodes which follow x are reachable but the nodes that precede x are not reachable.

→ To delete a designated node cur, address of the first node of the list should be provided. This is necessary because, to delete a node cur, the predecessor of this node has to be found. For this, search has to begin from the first node of the list.

These disadvantages can be overcome using special type of list called circular list.

Definition - Instead of storing 10 in the link field of a last node, store the starting address of the first node. such a list is called circular list.

In general the link field of last node contains the address of the first node.

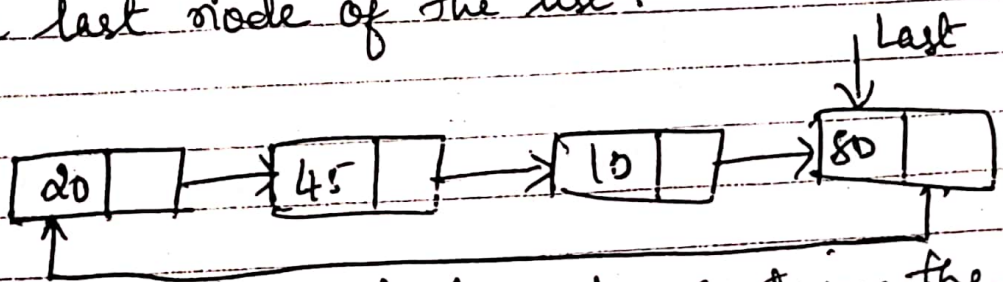


A Circular list can be used as a stack or a queue. To implement these data structures we require the following functions -

→ insert_front, → insert_rear, → delete_front
→ delete_rear, → display

Insert a node at the front end

Design - Let us consider a list with 4 nodes. Here, pointer last contains address of the last node of the list.



The link field of last node contains the address of the first node.

The sequence of steps to be followed are:

Step 1: To insert an item 50 at the front of the list, obtain a free node using malloc() function with the help of macro MALLOC() & insert the item as shown below.

```
MALLOC (temp, 1, struct node);
temp -> info = item;
```
(1)

Step 2: Copy the address of the first node (i.e last -> link) into link field of newly obtained node temp & the statement is

```
temp -> link = last -> link;
```
(2)

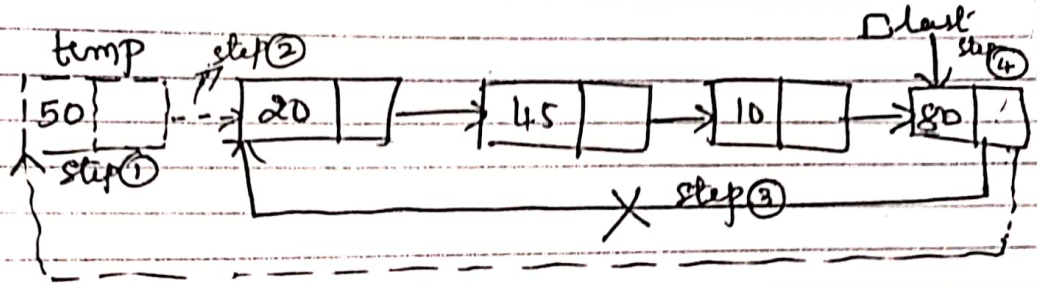
Step 3: Establish a link between the node temp & the last node. This can be done by copying the address of the node temp into link field of node last.

```
last -> link = temp;
```
(3)

Step 4: Finally, we return address of the last node.

```
return last;
```

pictorial representation is as shown -



The modifications from the previous diagram is shown in dotted lines.

C - function to insert an item at the front end of the list

```
NODE insert_front (int item, NODE last)
{
    NODE temp;
    MALLOC (temp, 1, struct node);
    temp -> info = item;
    if (last == NULL)
        last = temp;
    else
        temp -> link = last -> link
        last -> link = temp;
    return last;
}
```

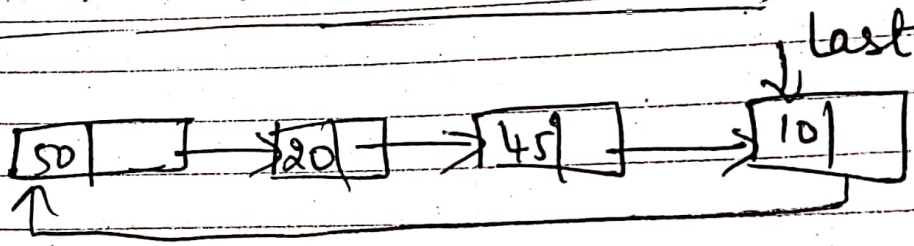
Insert a node at the rear end

Let us consider a list of 4 nodes, here pointer last contains address of the last node of the list.

Let us insert 80 at the rear end-

Step 1: Obtain a node using the function malloc() with the help of macro MALLOC() & store the item in 'info' field as -

```
MALLOC(temp, 1, struct node);
temp -> info = item;
```
(1)



Step 2: Copy the address of first node (i.e last -> link) into link field of newly obtained node temp.

```
temp -> link = last -> link
```
(2)

Step 3: Establish a link between the newly created node temp & the node last. This is achieved by copying the address of the node temp into link field of node last.

last -> link = temp; (3)

Step4: The new node is made as the last node

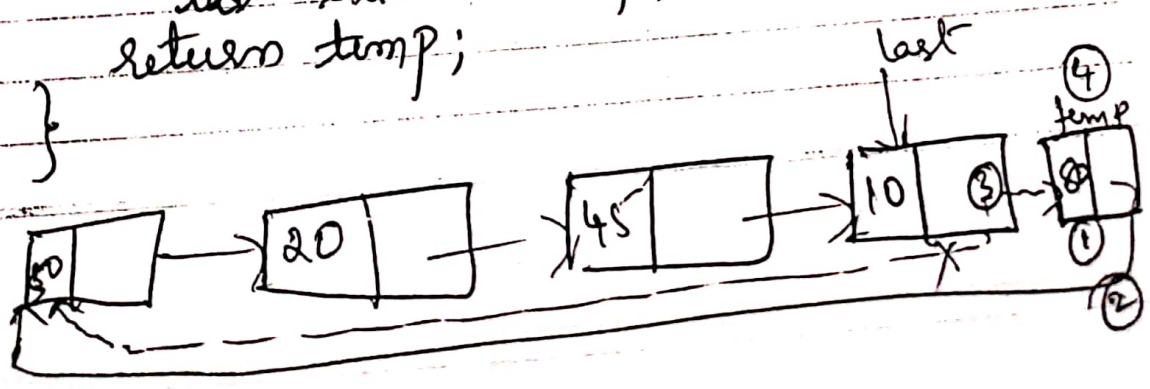
return temp; (4)

The above steps have been designed by assuming the list is already existing.

If the list is empty make temp itself as the first node as well as the last node.

C-function to insert at the rear end

```
NODE insert_rear(int item, NODE last)
{
    NODE temp;
    MALLOC(temp, 1, struct node)
    temp -> info = item;

    if (last == NULL)
        last = temp;
    else
        temp -> link = last -> link;
    last -> link = temp;
    return temp;
}
```



Delete a node from the front end

Consider a list with 5 nodes, here pointer last contains address of the last node of the list.

Step 1: Obtain the address of the first node

```
first = last -> link
```
(1)

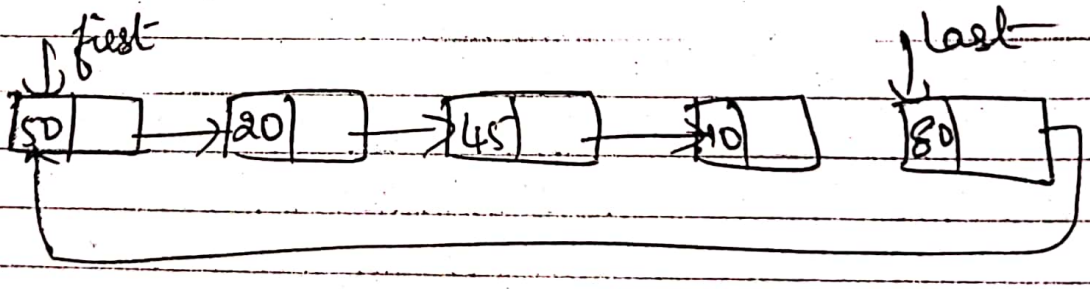
Step 2: Link the last node & new first node.

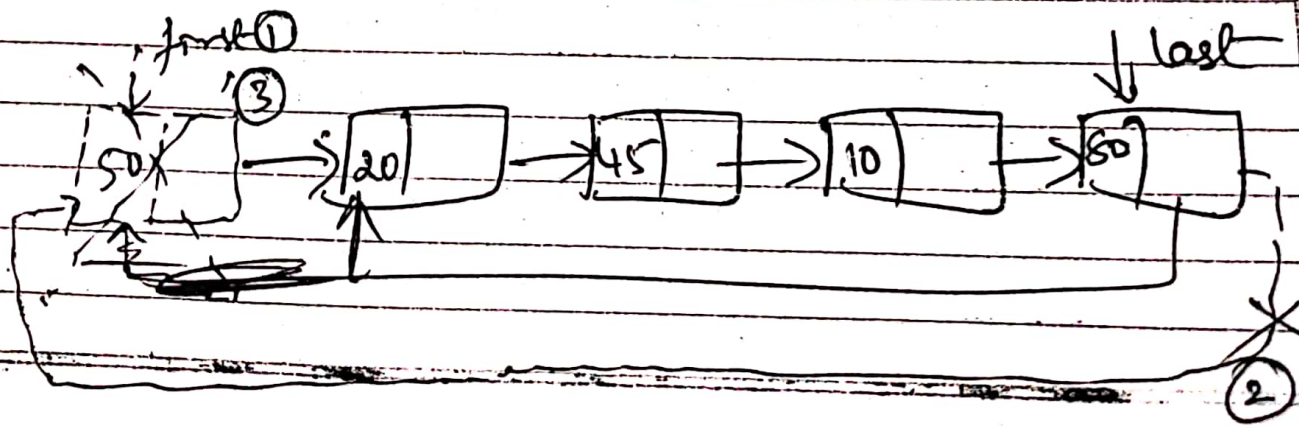
```
last -> link = first -> link
```
(2)

Step 3:- print the deleted item, Delete the old first node

(3)

```
printf("The item deleted = %d", first->info);
free(first);
```





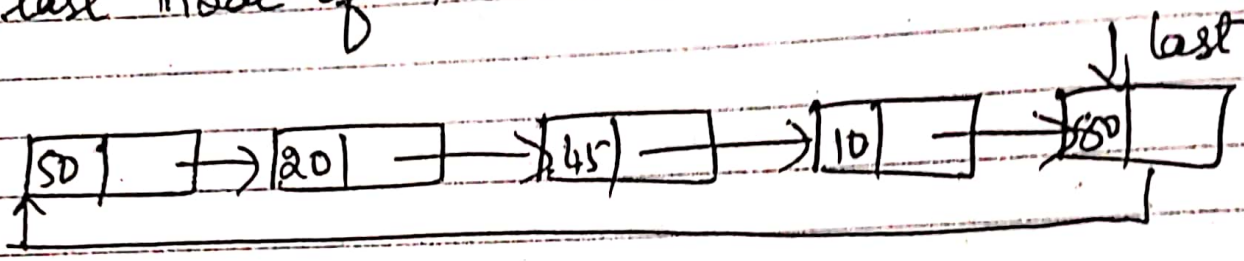
All these steps are designed assuming list is ~~Empty~~ not empty. If list is empty display appropriate message.

C - function to delete an item from the front end

```
NODE delete_front(NODE last)
{
NODE temp, first;
if(last == NULL)
  {
   printf("List is empty");
   return NULL;
  }
if(last -> link == last)        // Delete if only
                                // one node
{
printf("Item deleted = %d", last->info);
free(last);
return NULL;
}
first = last -> link;
last->link = first->link;
printf("Item deleted = %d", first->info);
free(first);
return NULL;
}
```

Delete a node from the rear end

Let us consider a list with 5 nodes. Here pointer last contains address of the last node of the list.



Step 1:- Obtain the address of the predecessor of the node to be deleted. This can be done by traversing from the first node till the link field of a node contains address of the last node.

```
prev = last -> link;

while (prev -> link != last)
  {
    prev = prev -> link;
  }
```

Step 2: The first node & the last but one node (i.e prev) are linked.

```
prev -> link = last -> link;
```

Step 3: The last node can be deleted using the statement

```
free(last);
```

Step 4: Return prev itself as the last node of the result

```
return(prev);
```

If there is only one node delete that node & assign NULL to the pointer variable last indicating that the list is empty.

```
if (last -> link == last)
{
printf("Item deleted = %d", last->info);
free(last);
return NULL;
}
```

If list is empty display appropriate message.

Display the contents of the circular linked list

```
void display(NODE last)
{
NODE temp;
if(last == NULL)
{
printf("List is empty");
return;
}
temp = last -> link;          // Display till we reach last node
while(temp != last)
{
printf("%d", temp -> info);
temp = temp -> link;
}
printf("%d", temp -> info);   // Display last node
}
```