Module -3

Linked List

Introduction

→ In previous sections/modules we have seen various data structures, such as stacks, queues and their representation

→ Let us see advantages of using arrays & implementing it in various data structures:

1. Data accessing is faster.
2. Simple

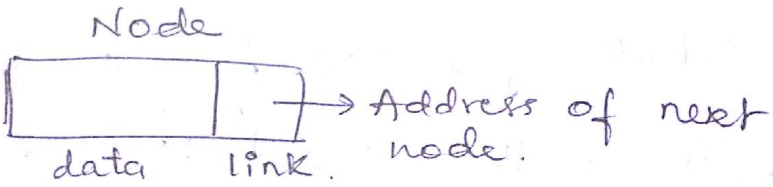
→ The disadvantages of using arrays:

1. Size of the array is fixed.
2. Array items are stored continously.
3. Insertion & deletion operations involving arrays is tedious job.

→ The above disadvantages can be overcome using Linked List.

(1) Linked List

Definition: A linked list or one-way list is the data structure which is the collection of zero or more nodes, where each node has some information. Each node is divided into two parts: data part and link part.



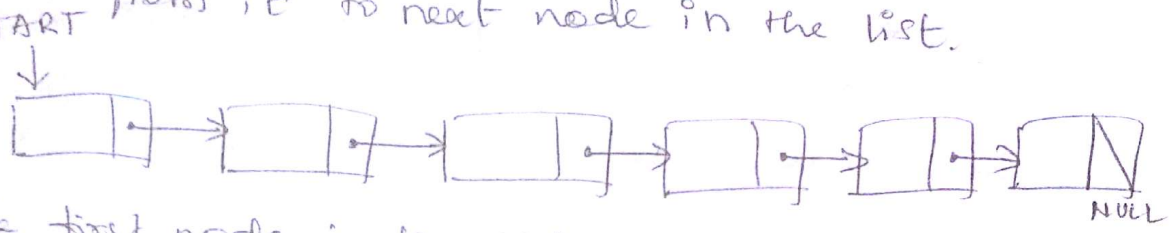
→ The above node consists of 2 parts

1. data: used to store the data or information to be manipulated.
2. link: it contains address of the next node.

→ Example : Consider a linked list with 6 nodes. Each node is pictured into two parts

→ The left part is data part. where data can be stored. (i.e. NAME, ADDRESS etc).

→ The right part is link part, a pointer to store address of next node. and there is a arrow drawn from it to next node in the list.



→ The first node in the list is pointed as START node.

→ The pointer of the last node is indicated or stored with special value called NULL. It indicates end of the list.

→ If START is stored with NULL value, that indicates a empty list. there are no nodes present in the list.

Types of Linked Lists

Linked Lists →
- Singly Linked List
- Doubly Linked List
- Circular Singly Linked List
- Circular Doubly Linked List.

2 Representation of Linked List in Memory

→ The nodes in the linked list are declared as structure.

→ The link part of the node is declared as self referential to structure.

→ A structure is a collection of one/more fields which are of same/different types.

→ A self referential structure is a structure which has atleast one field - a pointer to same structure

→ Consider the following:

```
Struct node
{
    int data;
    struct node *link;
};
```

from above structure difinition, that a node has two fields:

① data - integer field which contains information
② link - pointer field & hence it contains address.
. It contains address of next node in the list.

→ To create the variables of a node, we follow the declaration:

→ Along with structure declaration of node, create a typedef name for node.

i.e.:

```
typedef struct node *NODE;
```

→ A pointer variable for NODE can be declared as shown below:

```
NODE start;
```

→ If the linked list is empty, we store a special value NULL in the start. i.e:

```
start = NULL;
```
// empty linked list.

⇒ Linked List Operations (One-way list)

The following are the set of operations, that can be performed on the linked list/one way list:

1. Creation of Node in the list
2. Inserting a Node into the list.
3. Deleting a Node from the list.
4. Searching / Traversing a list

(1) Creation of Node in the list

The following are the steps used to create the node in the linked list & access the members of the node.

Step1: Create a node: we use malloc() function.

NODE *start;

start = (NODE *) malloc (sizeof (struct node *));

Note: start ⇒ access address of node.

*start ⇒ access entire contents of node.

(*start).data (or) start -> data ⇒ access data,

(*start).link (or) start -> link ⇒ access link.

step 2: store data

start -> data = 10; (or) (*start).data = 10;

step 3: Store NULL character.

start -> link = NULL; (or) (*start).link = NULL;

To Delete the node from the list.

free(first);

(2) Inserting node into the Singly Linked List

positions : (1) Inserting at front
(2) Inserting at end
(3) Inserting in the middle / specified position.
(4) Inserting in sorted list.

(1) Algorithm : Insert front (list, start, temp)

Step 1 : Create a new node called temp.

step 2 : store data in temp → data.

step 3 : store address of first node in temp → link.
i.e link the temp node to start of list.

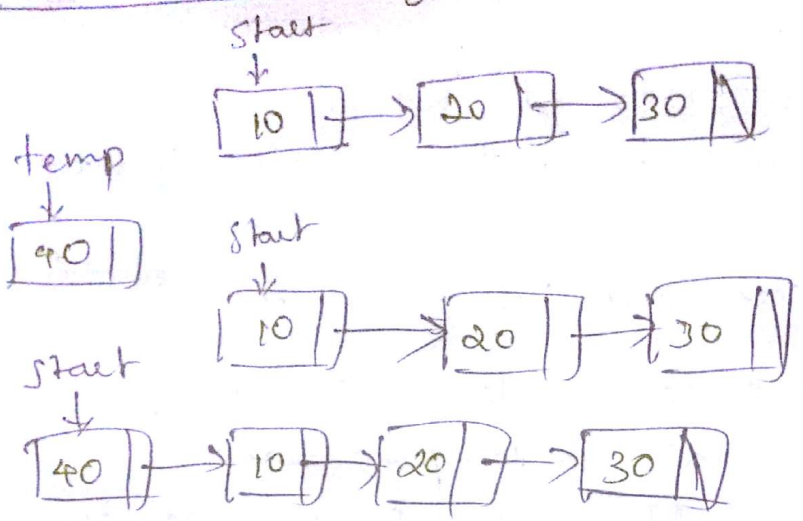
step 4 : Shift start to temp node, which becomes the new start node.
start = temp;

step 5 : Increment count of list.

→ C-function :

```
void insert_front()
{
    NODE temp;
    temp = read();
    if(start == NULL)
    {
        temp->next = NULL;
        start = temp;
    }
    else
    {
        temp->next = start;
        start = temp;
    }
}
```

Ex: Consider existing list



---

② Inserting at End of list.

Algorithm : insertend (list, start, temp)

Step 1 : Create a new node.

temp = (struct node *) malloc (sizeof (struct node *));

Step 2 : Store data in the node.

~~temp = 50~~; temp -> data = 50;

Step 3 : Store NULL in the link part of the temp node.

temp -> link = NULL;

Step 4 : Traverse to the last node in the existing list. t = start;

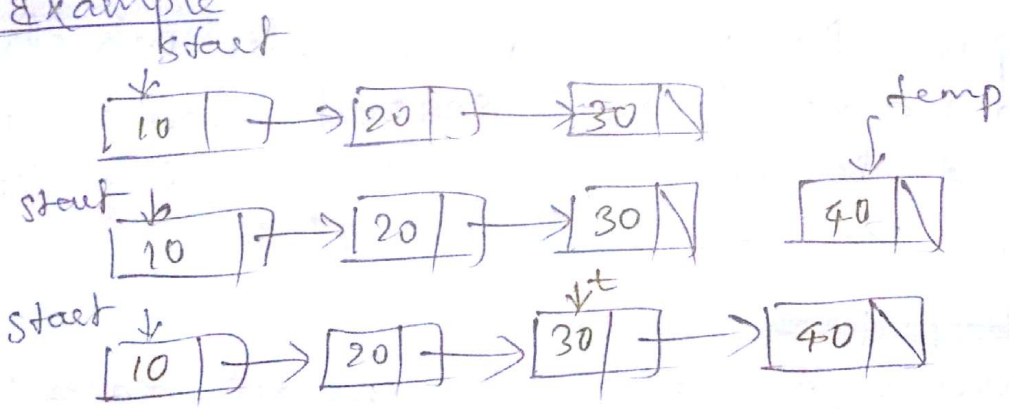
```
while (t -> link != NULL)
{
    t = t -> link;
}
t -> link = temp;
```

Step 5 : Increment the count of the list.

→ C function

```
void insert_end()
{
    int num;
    NODE q, temp;
    temp = read();
    temp -> next = NULL;
    if(start == NULL)
    {
        start = temp;
    }
    else
    {
        q = start;
        while(q -> next != NULL)
            q = q -> next;
        q -> next = temp;
    }
}
```

→ Example



③ Inserting at specified location

→ New node can be inserted at the specified location, by taking any of the two informations from the user.

a) Taking an input from the user about node number.

b) Taking an input about data of the node after which new node to be inserted.

→ We will consider the following algorithm to insert a node after the data of a specified node.

Algorithm : Insert_pos (list, start, temp)

step1 : Create a new node (temp node)

temp = (struct node *) malloc (sizeof(struct node*));

step2 : Store the data into the temp node.

temp → data = data ;

step3 : Input the node's data after which new node to be inserted. Node's data should be existing data from the list.

Read key;

Step4 : ~~while (t ...)~~

Traverse till the key, (node's data) in the list.

```
t = start ;
while ( t → data != key)
{
    t = t → link ;
}
```

Step5 : Mark next node.

next = t → link

Step6 : Connect the new node (temp) to the list.

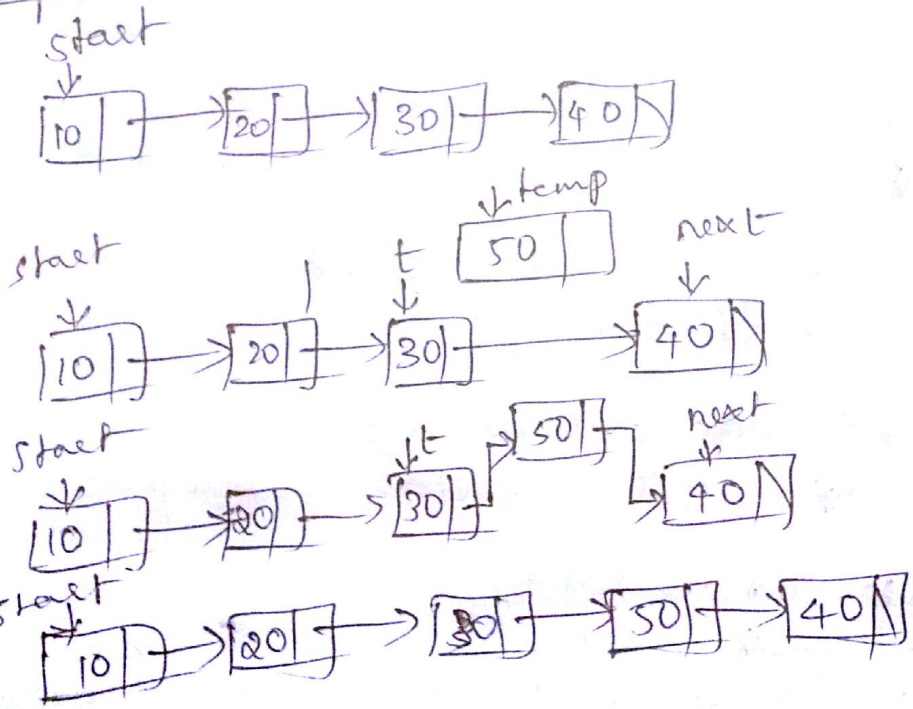
```
t -> link = temp;
temp -> link = next;
```

step 6 : Increment the count of the list.

C-function

```
void insert_pos()
{
    NODE temp, t, next;
    int key, data;
    temp = (struct node *) malloc (sizeof (struct node *));
    temp -> data = data;
    printf(" Enter key data \n");
    scanf ("%d", &key);
    t = start;
    while ( t -> data != key)
    {
        t = t -> link;
    }
    next = t -> link;
    t -> link = temp;
    temp -> link = next;
}
```

Examp 6



(4) Inserting into sorted Linked List

→ Inserting a new node into a sorted list, requires the list to be already sorted.
→ We assume that the existing linked list should be already sorted.
→ Then we start from the first node & compare the value/data of first node with the new node's data.
→ We traverse till the node, whose data is lesser than new node's data.
→ Once, we reach the current data node, we insert the node & connect the node.
→ The algorithm as follows:

Algorithm: insert_sorted()

step1: Create the new node (temp node)

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

step2: Store data into the node.

```
temp -> data = data;
```

step3: Traverse till the appropriate node in the sorted list.

[for 1st node]

```
cur = start;
if (cur -> data < temp -> data)
{
    temp -> link = start;
    start = temp;
}
```

[for 2nd node onwards]

```
while (cur -> link -> data < temp -> data)
{
    cur = cur -> link;
} next = cur -> link;
```

step4: Connect the temp node to the sorted list

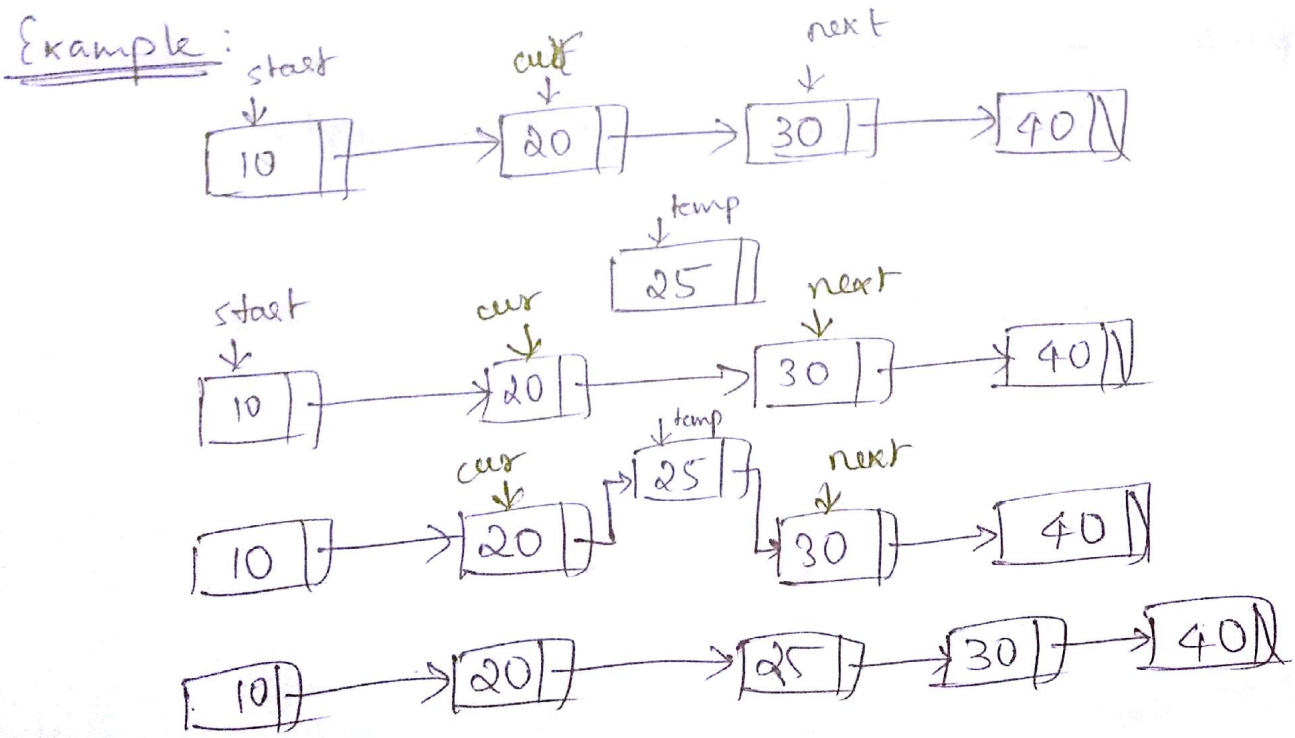
```
cur -> link = temp;
temp -> link = next;
```

Steps: Increment the count of the Sorted list.

C-function

```
void insert_sorted()
{
    NODE temp, cur, next;
    temp=(struct node *)malloc(sizeof(struct node));
    temp->data = data;
    t=start;
    if (cur->data > temp->data)
    {
        temp->link = start;
        start = temp;
    }
    while (cur->link->data < temp->data)
    {
        cur = cur->link;
    }
    next = cur->link;
    cur->link = temp;
    temp->link = next;
}
```

Example:



⇒ (3) Deleting a node from the list

To perform delete operation on linked list/one way list, the nodes can be deleted from one of the following positions.

(a) Delete from front of list.
(b) Delete from end of list.
(c) Delete from specified position.

(a) Delete from front of the list

The algorithm as follows:

Step 1: Mark first node as temp node.

temp = start;

Step 2: Shift start node to next node in the list

start = start -> link;

Step 3: Delink the temp node from next node

~~temp -> next~~

temp -> link = NULL;

Step 4: ~~Free the node~~

Display the temp node data & free the node.

Print temp -> data.

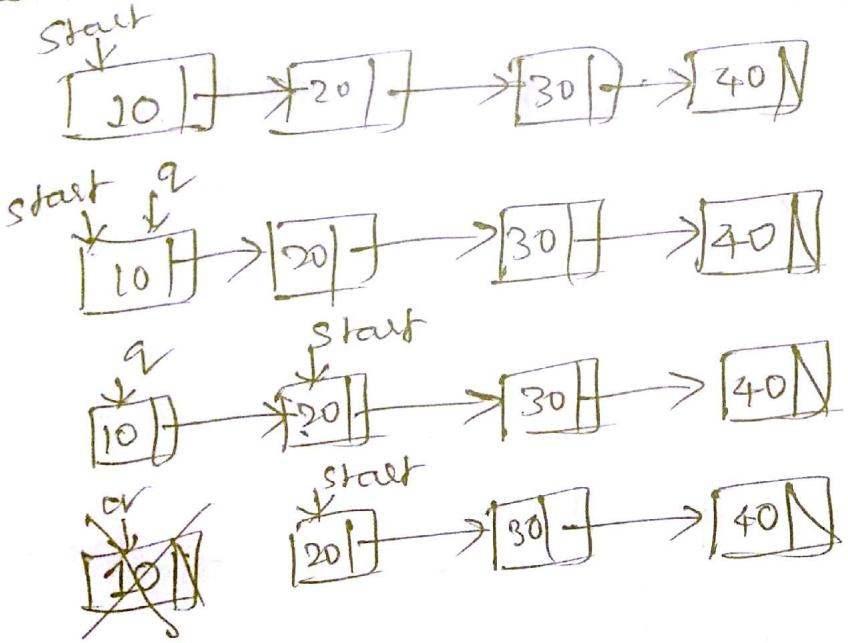
free(temp)

C function

```
void delete_front()
{
    NODE q;
    if(start == NULL)
    {
        printf("List is Empty\n");
    }
    else
    {
        q = start;
        start = start -> next;
        printf("Deleted node is %d\n", q -> data);
        free(q);
    }
}
```

Example

Consider the linked list with 4 nodes



(b) Delete from End of the list

→ The deleting of the last node in the list can be peafrom by traversing till the last but one node & masking the last node.

→ Once we reach last but one node, we delink the last node by storing NULL in the link field of last but one node.

→ Then we display the contents & free the memory from compiter.

Algorithm : Delete_End()

step 1 : Traverse till end of list and locate last but one node in the list

```
temp = start;
if (start -> link == NULL)   [only one node]
{
        print  start ->data
        free (start)
}
[for more than one node]
while (temp -> link -> link != NULL)
{
        temp = temp -> link;
}
```

Step 2 : Mask the last node

```
last = temp -> link
```

step 3 : Mask/Delink the last node from its previous node.

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

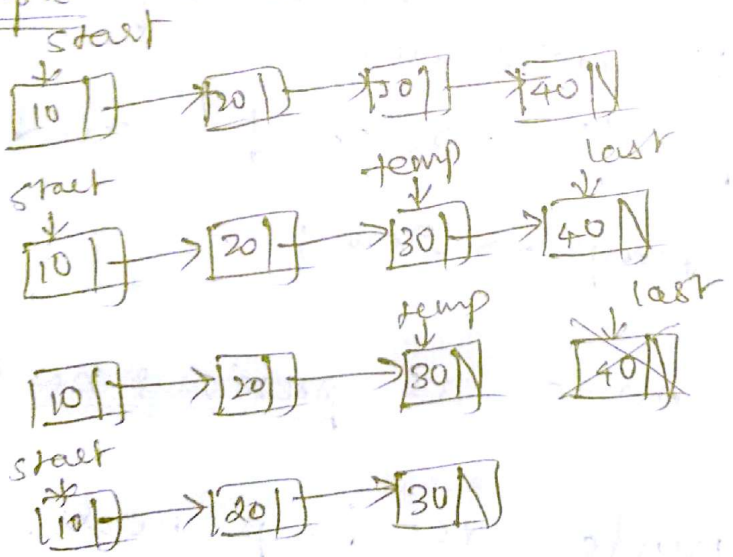
step 4 : Print data of last node & free the last node from memory.

```
print last -> data;
free(last);
```

Function: Delete from end of list

```
void delete_end()
{   NODE q, t;
    if(start == NULL)
        printf("List is Empty \n");
    else if(start -> next == NULL)
    {   printf("Deleted node is %d\n", start -> data);
        start = NULL;
        free(start);
    }
    else
    {   q = start;
        while(q -> next -> next != NULL)
            q = q -> next;
        t = q -> next;
        q -> next = NULL;
        printf("Deleted data is %d\n", t -> data);
        free(t);
    }
}
```

Example: Consider a list with 4 nodes:



ⓒ Delete from specified position

→ To perform deleting of a node from specified position, it can be done in two ways:

ⓐ Deleting based on node number

ⓑ Deleting based on data present in the node

→ Note: To perform deleting from a data of the node, user must enter existing data from the linked list.

→ The following is the algorithm to perform the deleting of a node from specified data of node

Algorithm: delete_pos()

Step 1: Read the key data of the node to be deleted.

Read key data.

Step 2: Traverse to the one node before the specified key data node.

```
while(temp->link->data != key data)
{
    temp = temp->link;
}
```

Step 3: Mark temp node as previous node.

```
prev = temp;
```

Step 4: Mark next node as temp node.

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

Step 5: Mark the temp's next one node as next node in the list.

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

Step 6: Delink the temp node and connect prev node to next node in the list.

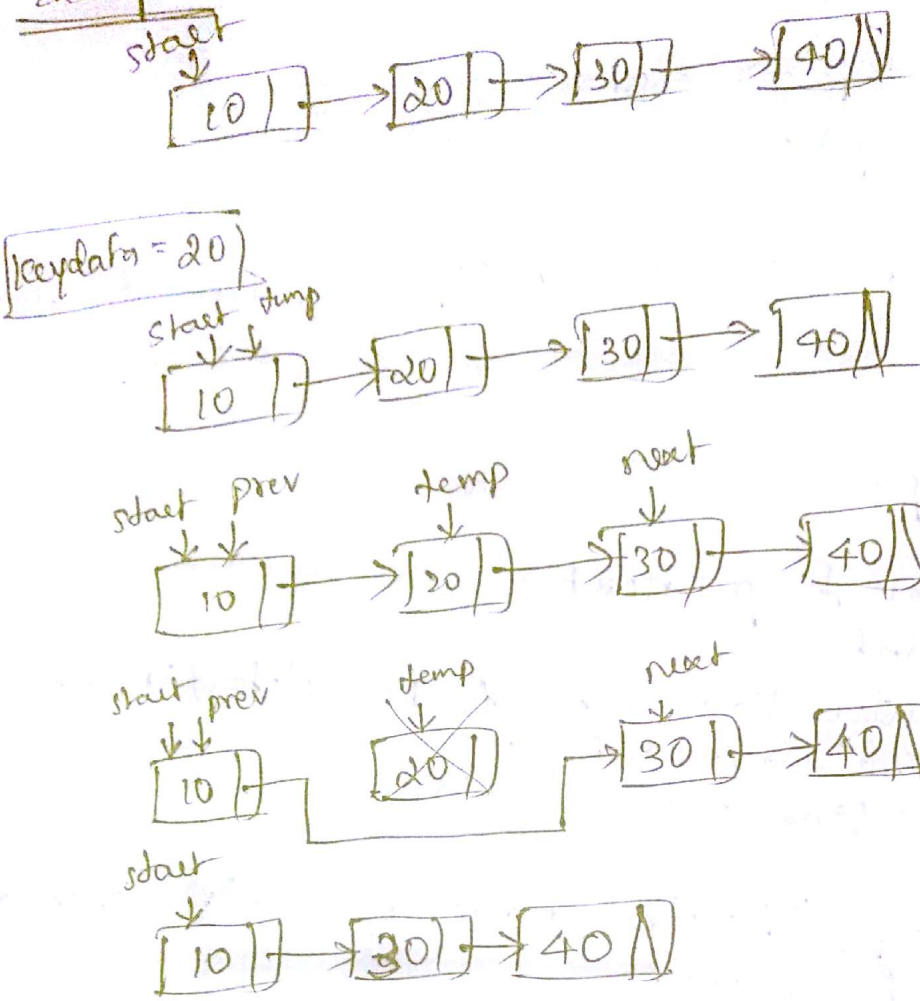
```
prev -> link = next;
temp -> link = NULL;
print temp -> data;
free (temp);
```

step 7 : Decrement the count of the linked list.

C-function

```
void delete_pos()
{
    NODE prev, temp, next;
    int key data;
    printf("Enter Key data to be deleted \n");
    scanf("%d", &keydata);
    if(start -> link == NULL)
    {
        printf("Deleted node is. %d \n", start->data);
        start = NULL;
    }
    while (temp -> link -> data != keydata)
    {
        temp = temp -> link;
    }
    prev = temp;
    temp = temp -> link;
    next = temp -> link;
    prev -> link = next;
    temp -> link = NULL;
    printf(" Deleted node data is %d \n", temp -> data);
    free(temp);
}
```

Example



[4] Traversing the linked List

→ Traversing of a linked list can be performed for different operations / purposes.

(1) To search a specific node in the list.

(2) To display the complete linked list.

(1) To Search the one way List (sorted/unsorted List)

→ To search a specific node in the list, we need to ask the user to enter the node's data to be searched.

→ If the data to be searched is found in the list, then search successful.

→ If we reach end of the list & data not found, search is unsuccessful.

Algorithm : Search a node in existing list

Step 1 : Input the keydata to be searched from the existing linked list.
Read keydata.

step 2 : If list is Empty, no node in the list, Print 'Empty List'

step 3 : Visit each node & compare the keydata with node's data.
while (temp → link != NULL)
{
a) if (temp → data == keydata)
{
printf("Search Successfull \n");
printf("found at position %d \n", count);
break; [stop the search]
}
else
b) Shift the temp to next node in the list.
temp = temp → link;
count the visited node.
count ++;
c) If temp reaches last node & data not found, then search unsuccessful.
Print 'Unsuccessful Search'.

Step 4 : Stop.

C-function

```
void seach()
{
    count = 1;
    int Keydata;
    if(staut == NULL)
        printf("List Empty \n");
    temp = staut;
    while (temp -> link != NULL)
    {
        if (temp -> data == Keydata)
        {
            printf("Successful seach \n");
            printf("found at %d node position \n", count);
            break;
        }
        else
        {
            temp = temp -> link;
            count++;
        }
        if(temp == NULL)
        {
            printf("Seach Unsuccessful \n");
        }
    }
}
```

(b) Traversing the list to display the data

→ To display the linked list, we visit each node of the list & display the data present in the list.
→ Until the last node of the list we visit each node.

Algorithm : display ()
step 1 : If list is empty.
Print 'Empty list'
step 2 : Visit each node & display the data of the node
while (temp -> link != NULL)
{
printf(" %d \t", temp -> data);
temp = temp -> link
count = count + 1;
}
step 3 : ~~Print~~ Display the count of the nodes
Print 'count'
step 4 : End.

Function : display ()

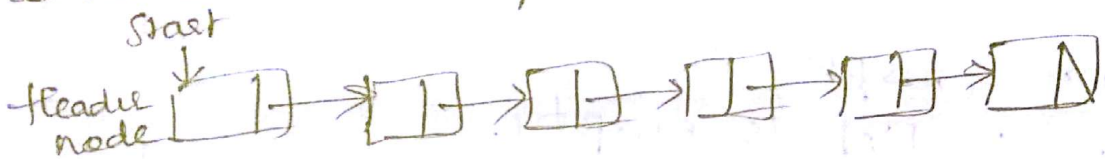
```
void display()
{
  int count = 1;
  if (start == NULL)
      printf("List Empty\n");
  printf("The nodes in the list are\n");
  while (temp -> link != NULL)
  {
      printf("%d \t", temp -> data);
      temp = temp -> link;
      count++;
  }
  printf("\n");
  printf("Number of nodes %d \n", count);
}
```

⇒ Header Linked Lists

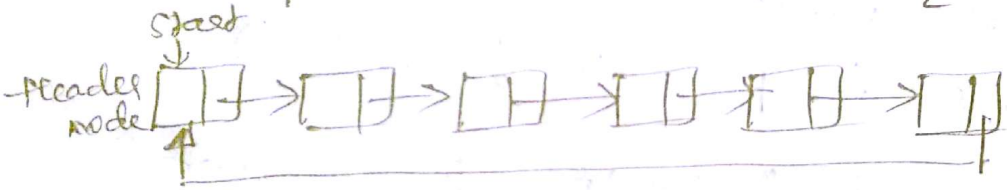
→ A Header linked list is a linked list which always contains a special node, called header node, at the beginning of the list.

→ The following two types of header lists:

① The grounded header list: where the last node contains the NULL pointer.



② The Circular header list: where the last node of the list points back to the header node.



Observe from above header lists, in ①st one, `temp -> link = NULL` while traversing the list acts as a end of the list.

In ②nd list, the header node address acts as a end of the list: `temp -> link = header`.

→ Traversing a Circular Header List

Algorithm: Traverse()

```
Step 1 : If the list is Empty,
           Print 'Empty List'
Step 2 : Visit each node & display data till link is
         equal header node's address.
         while (temp -> link != header)
         {  Print ' temp -> data.
            temp = temp -> link.
            count = count + 1.
         }
Step 3 : Display total count of the nodes.
```

Doubly Linked List / Two Way List

→ One of the most power variations of linked list is Doubly linked list.

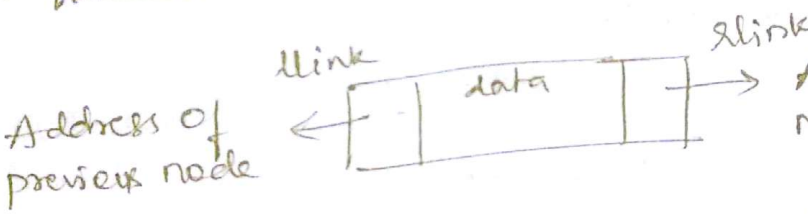
→ Def^n: A doubly linked list/Two way list is a linear collection of nodes where each node is divided into 3-parts.

ⓐ info - field where data/information to be stored

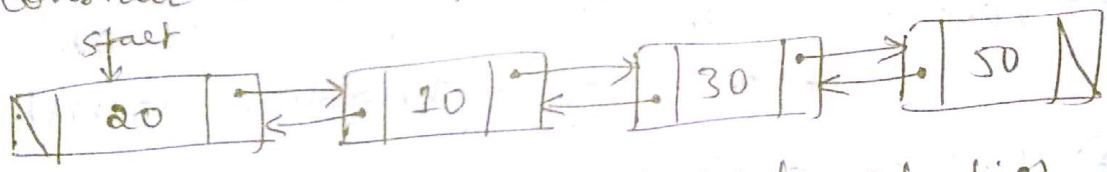
ⓑ llink - pointer which contains address of left node/previous node in the list.

ⓒ rlink - pointer which contains address of right node/next node in the list.

→ Using such lists, it is possible to traverse the list in forward & backward directions.



→ Consider an example of 4 nodes in DLL.



* Start points to the starting node of list.
* The llink part of first node is ~~end~~ stored NULL. that indicates, it is first node in the list.
* The rlink part of last node 50, is stored with NULL, that indicates it is last node in the list.

→ Doubly linked list Operations (Two way list Operations)

The following are the operations that can be performed on Doubly Linked List.

1. Creation of Doubly Linked List
2. Inserting into the DLL
3. Deleting from the "
4. Traversing the DLL.

① Creation of Doubly Linked List

To create the nodes in doubly linked list, the following are the steps followed:

Step 1: Create of new node.

→ Representation of Doubly Linked List

To create a new node, we use structure as follows:

```
struct node
{
    int data;
    struct node *rlink;
    struct node *llink;
};
```

Above structure has 3 fields:

① node → to store data/information

② rlink → a link to a structure to store address of right side node/next node

③ llink → a link to a structure to store address of left side node/previous node.

→ Typedef node can be provided to the node.

```
typedef struct node *NODE;
```

```
NODE start;
```

```
start = NULL; // start pointing to NULL, empty list.
```

① Creation of Doubly Linked list node.

The following is algorithm to create a DLL node & linking the 2 link parts – rlink & llink to the appropriate nodes in the list.

Step 1 : Create a node, using malloc() function.

```
NODE start;
start = (NODE *) malloc(sizeof(struct node));
```

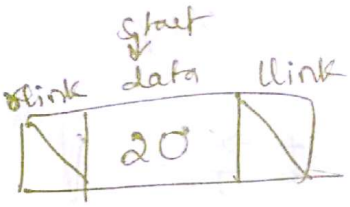
Step 2 : Store data into node.

```
start → data = value;
```

Step 3 : Store NULL values at both links to indicate ~~starting~~ one node in list.

```
start → rlink = NULL;
start → llink = NULL;
```

Eg:



② Inseeting into Doubly Linked list

Inseeting can be done at different positions

(a) Inseet at front of DLL
(b) Inseet at End of DLL
(c) Inseet at specified position in DLL.

(a) Inseet at front of DLL

Algorithm : Inseet_front_dll()

Step 1 : Create a new node.
```
temp = (struct node *) malloc(sizeof(struct node));
```
Step 2 : Store data in the temp node
```
temp → data = value.
```
Step 3 : Store the address of first node in temp node's rlink. & Store NULL in temp node's llink.
```
temp → rlink = start.
temp → llink = NULL.
```

start -> llink = temp.

step 4 : Shift the start to new node (temp).
Now temp becomes the start node.
start = temp.

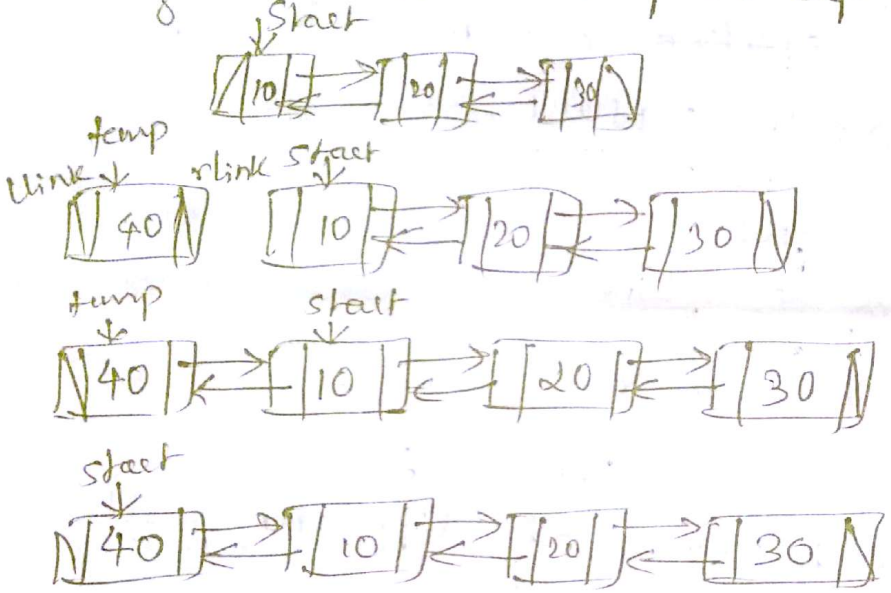
step 5 : Increment count of the list.

step 6 : End

C-function: Refer lab program 08. insert_front().
[Note: links are denoted as next for rlink & prev for llink]

Example:

Consider existing list of 3 nodes, we perform inserting a new node at front of Doubly Linked list.



(b) Insert at end of Two way list/DLL.

The following is the algorithm to perform inserting a new node to the end of DLL.

step 1 : Create a new node
temp = (struct node *) malloc(sizeof(struct node))

step 2 : Store data in the temp node
temp -> data = value;
temp -> rlink = NULL;
temp -> llink = NULL;

step 3 : If DLL is empty, Make temp node as start node.
start = temp.

step4 : Traverse till the end of the DLL.

```
cur = start;
while(cur -> rlink != NULL)
{
    cur = cur -> rlink.
}
```

step 5 : Link the new (temp node) node to the last node of the DLL.

```
cur -> rlink = temp.
temp -> llink = cur.
```

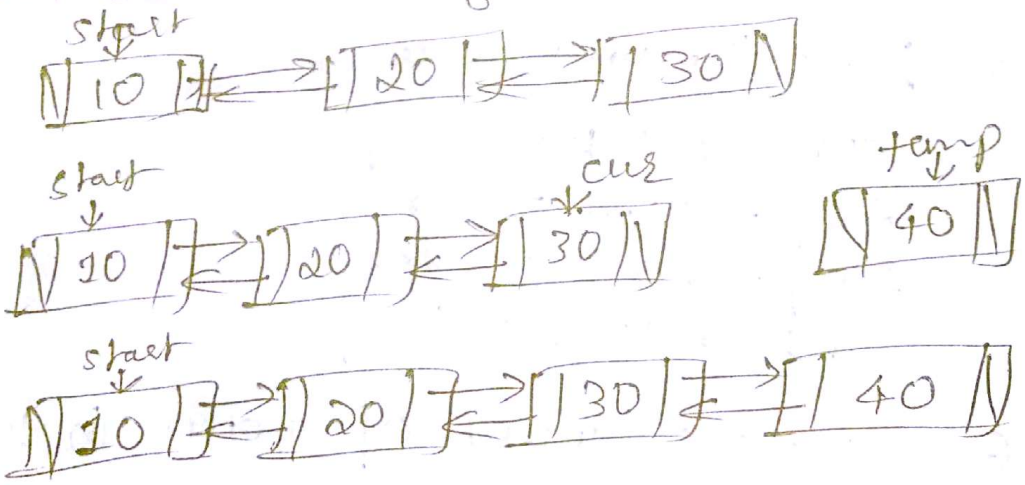
step 6 : Count Increment for number of nodes.

step 7 : End.

C-function : Refer insert_end() from Lab Pgm: 08

Example : Consider DLL with 3 nodes.

Perform inserting new node at end of DLL



© Inserting at specified position

-Algorithm : insert_pos()

step 1 : Create a new node

step 2 : Store data in new node

step 3 : Read the data of existing DLL till which, next node to be inserted.
Read 'value'

step 4 : Traverse till the data of node, after which new node to be inserted.

~~cur = start~~
~~while(cur -> rlink != NULL && &~~

```
while(cur -> data != value)
{
    cur = cur -> rlink;
}
next = cur -> rlink. [mark next node]
```

step 5 : Link the new node

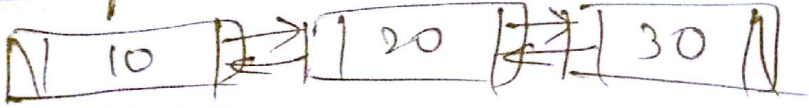
```
cur -> rlink = temp.
temp -> llink = cur.
temp -> rlink = next.
next -> llink = temp.
```

step 6 : Increase count of nodes in the list.

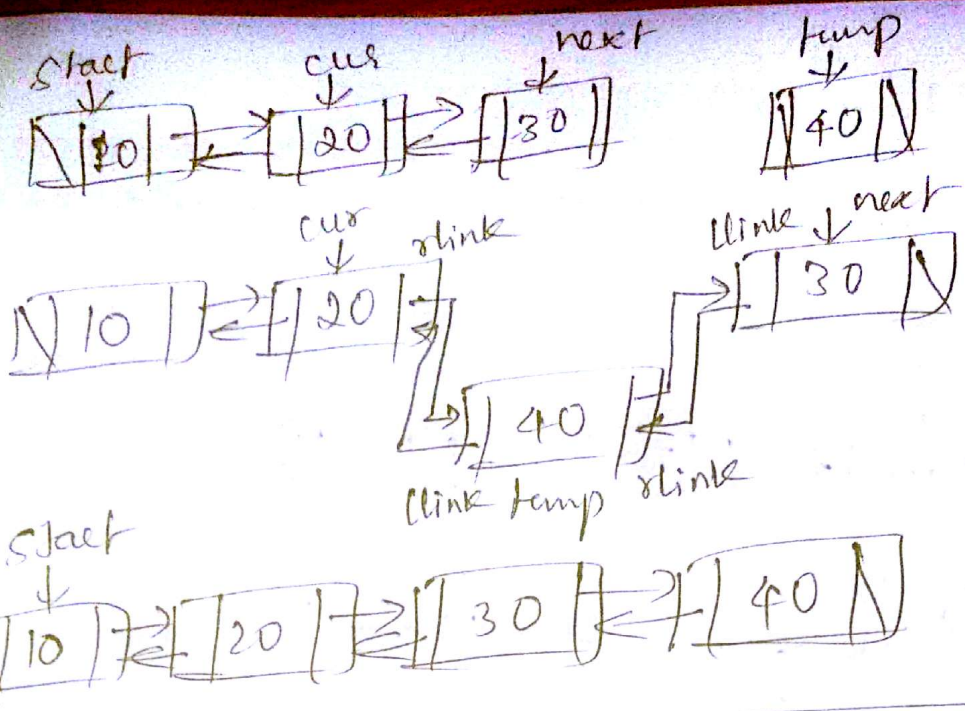
step 7 : End.

~~C function : Refer Data Prog~~

Example :



value = 20 after 20, new node to be inserted.



---

(2) Deleting from DLL

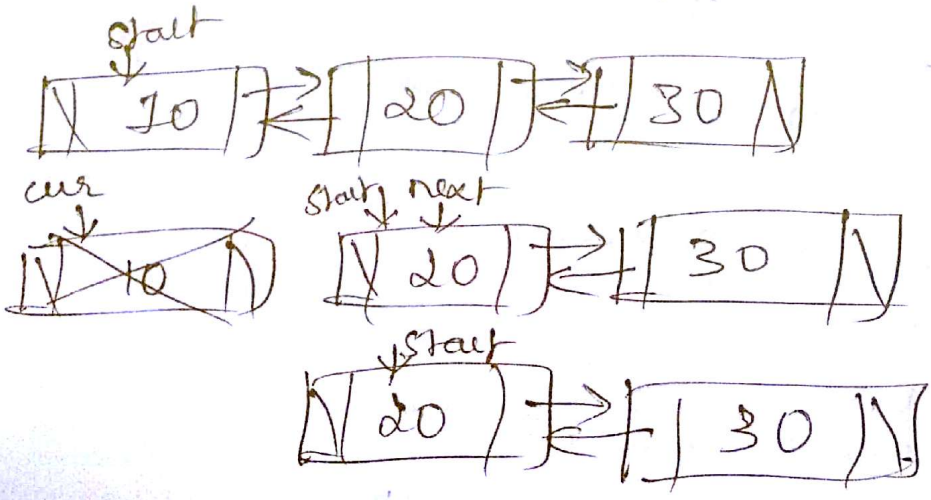
To perform deleting of a node, it can be done from different positions of the DLL.

(a) Delete from front of DLL
(b) Delete from end of DLL
(c) Delete from specified position.

(a) Delete from front of DLL

Refer Lab Expt 08 : delete_front() function.

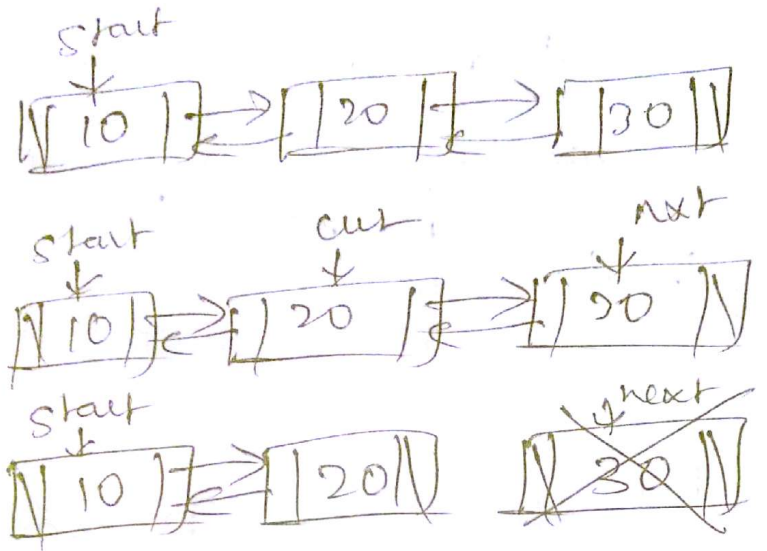
Ex:



(b) Delete from end of DLL

(function: Lab Expt 08 : delete_end())

Ex :



(c) Delete from specified position

Algorithm

step 1 : Read the data of the node which node has to be deleted.
Read 'value'

step 2 : Traverse till one node before the deleting node.

```
cur = start.
while (cur -> link -> data != value)
{
    cur = cur -> link.
}
```

step 3 : Mark previous & next nodes in the list.

```
prev = cur.
cur = cur -> link
next = cur -> link.
```

step4 : Now cur is pointing to node to be deleted.
Delink the cur node & link the prev & next node of the existing DLL.

prev → rlink = next.

next → llink = prev.

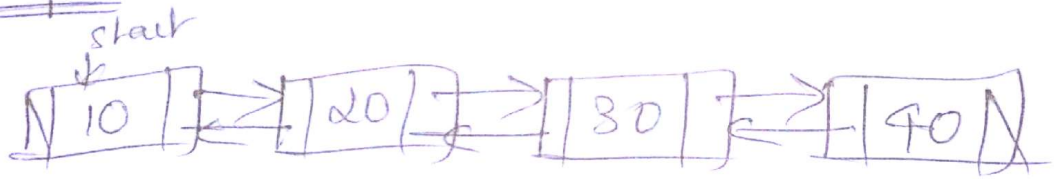
cur → rlink = cur → llink = NULL.

Print 'cur → data'

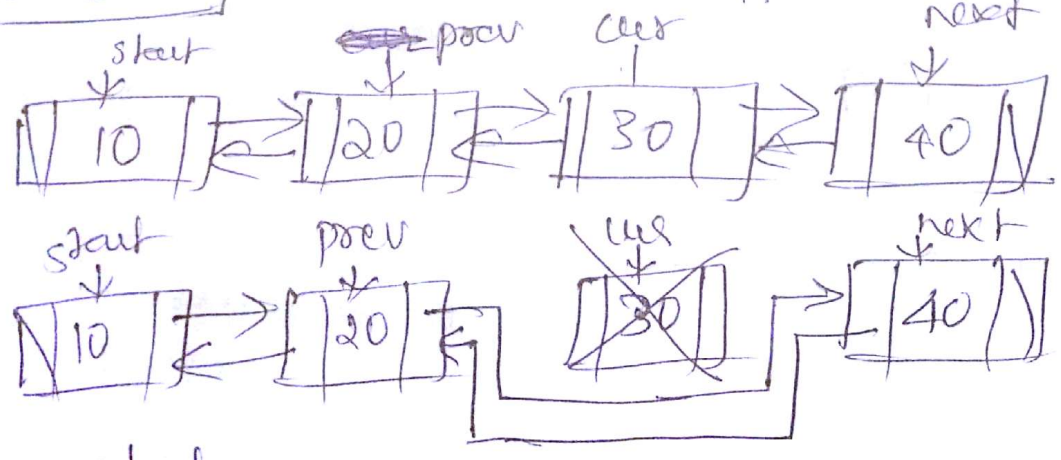
step5 : Decrement Count of DLL.

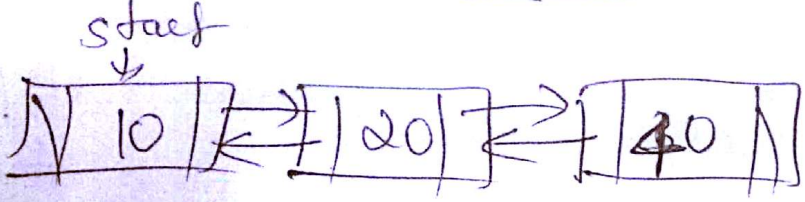
step6 : End.

Example :



value = 30 node to be deleted.





[3] Traversing the DLL

We perform traversing the DLL/Two way list for the following purposes.

(a) Displaying all the nodes of DLL
(b) Searching a node in the DLL.

(a) Displaying the nodes of DLL

To display the data of every node of the DLL, we visit each node & display the data.

→ The main advantage of DLL is, since it is having 2-link fields, the DLL can be displayed in 2-directions,
  * forward direction.
  * Reverse direction

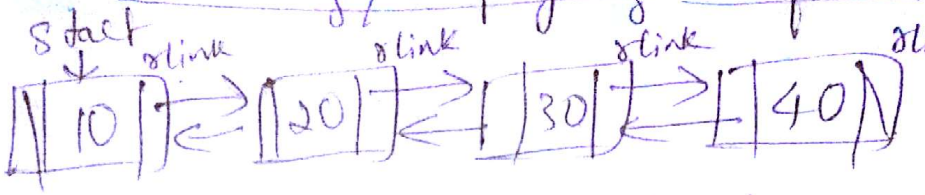
→ In forward direction, the data of the DLL is displayed from 1st node till the last node

* If reverse direction, data is displayed from last node to first node.

→ forward direction can be traversed by shifting to next node using rlink of node.
→ Reverse direction can be traversed, first by reaching last node & then, displaying data from last node, then to move to previous node, we use llink field of each node. Display data till first node of the list.

Example

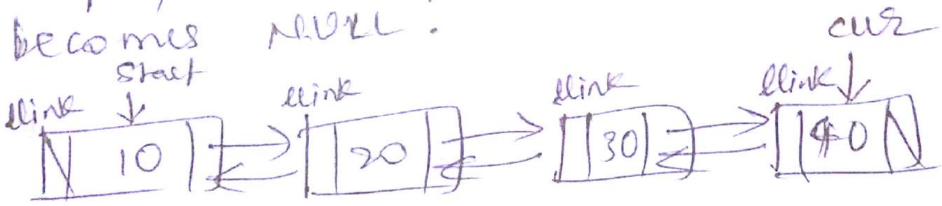
(a) Traversing / Displaying in forward direction.



→ We start displaying node from start node.
→ Using rlink field of every node we visit each & display the data of the node.
→ Display all nodes untill rlink becomes NULL.
Output will be a forward list.
10 → 20 → 30 → 40.

(*) Displaying in Reverse direction

→ To display the DLL in reverse direction, first reach the last node of the list.
→ Once we are at last node, using llink from last node, we can visit the previous node of list.
→ Display data of each node untill llink becomes NULL.



→ Output displayed will be a reverse list
40 → 30 → 20 → 10.

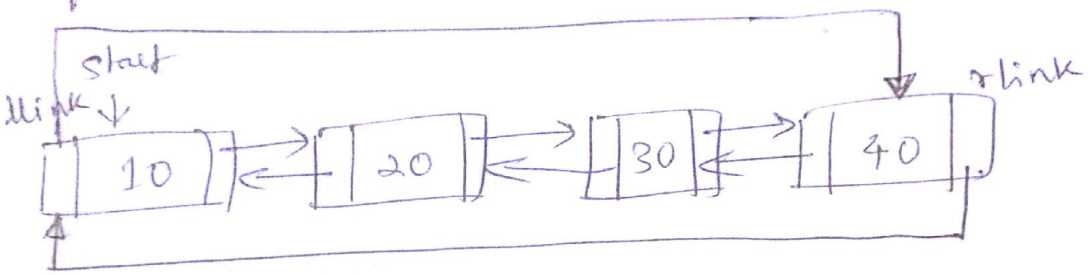
(b) Searching a DLL

→ Searching operation can be performed same as one way list, instead from start node, we check for rlink to visit next node in the list.
→ If data of node is present, Search successful
If data is not present till last node, search unsuccessful.

Circular DLlist / Two Way Circular List

(or) Circular Doubly Linked list

→ A circular DLL / Two Way Circular list is variation of DLL, in which
* rlink (right link) of last node contains address of first node.
* llink (left link) of first node contains address of last node.

→ 

→ The first of the DLL is start node, whose llink part stores address of last node. & last node's rlink part stores address of first node.

→ Thus making the DLL circular.

→ Advantage: If we want to traverse the DLL in reverse order, from llink of start node, we can directly reach the last node of DLL

→ The problem of traversing till last node & visiting each node in reverse direction in normal DLL is eliminated.

① Insert at front of Circular DLL

→ To perform insertion at front of the circular DLL,

a) Create a new node,

b) Store data in the new node.

c) Link the new node to start node of existing list.

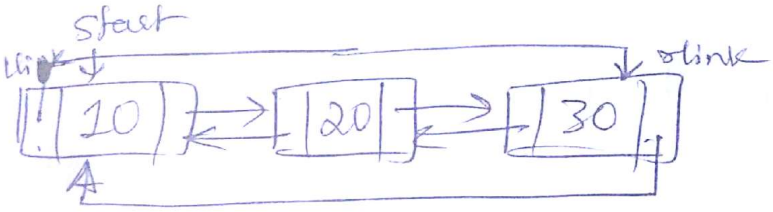
store start address in rlink of new node

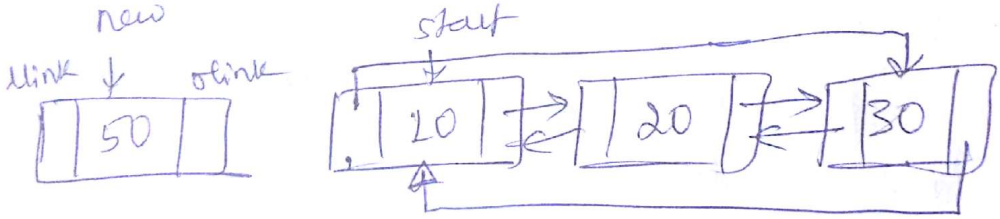
store last node address in llink of new node

shift the start to new node.

d) Increment the count of list.

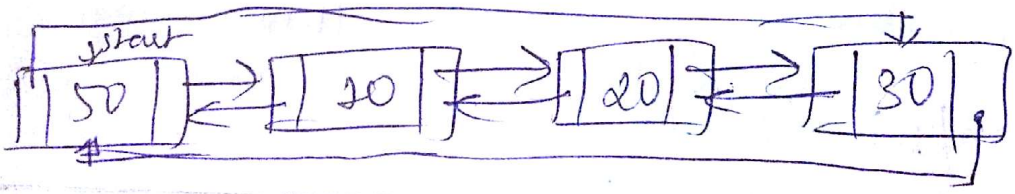
Example: Consider a circular DLL of 3 nodes:





Linking

```
new → llink = start → llink;
last = start → llink;
last → rlink = new;
new → rlink = start;
start → llink = new;
start = new.
```



② Insert at end of Circular DLL

→ To perform insertion at the end of circular DLL, first we need to traverse till the last node of list.

→ Once we reach last node, perform linking of new node.

a) Create a new node

b) Store data in the node

c) Traverse till last node in the list

```
temp = start
while (temp -> rlink != start)
{
    temp = temp -> rlink;
}
```

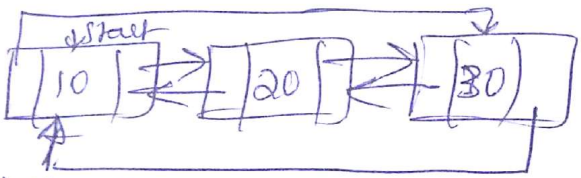
d) Link the new to last node.

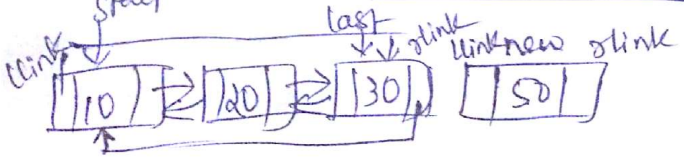
→ Store rlink address of last node in rlink of new node.

→ make first node & store new node address in llink of first node.

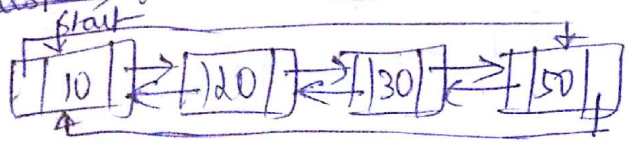
→ Link the last node to new node.

Example:





~~last -> rlink = new~~




```
new -> rlink = start -> rlink;
start -> llink = new;
last -> rlink = new;
new -> llink = last;
```

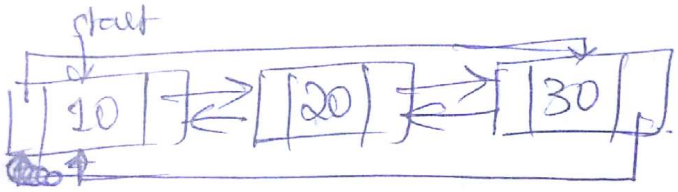
Delete from circular DLL

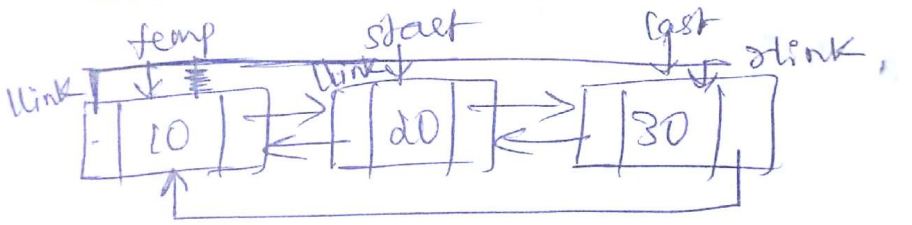
(a) Delete from front of circular DLL.

→ To peeform deletion from front of the list, mark the first node of list & next node as start node of the list.

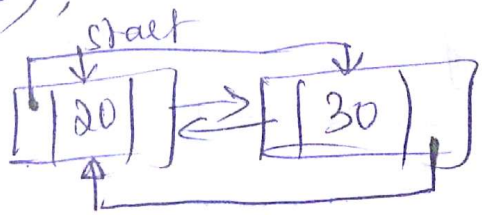
→ Once nodes are marked, remove the links of first node & connect the second node to the last node of the list.

Example: Consider a circular DLL of 3 nodes.






```
Last = temp → llink;
Last → rlink = start;
start → llink = Last;
temp → llink = NULL;
temp → rlink = NULL;
Print 'temp → data;
free (temp);
```



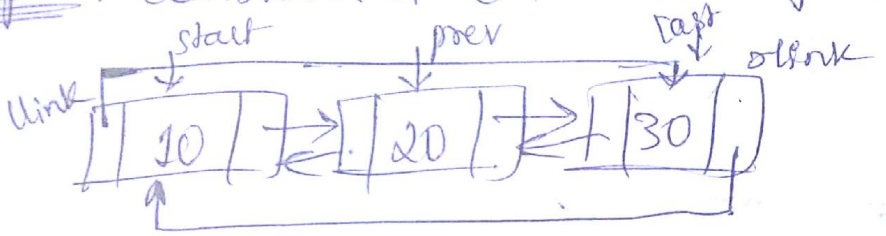
(5) Delete from End of Circular Doubly Linked List

→ To perform deletion of last node in the Circular doubly linked list, first we need to move to last node of the list.

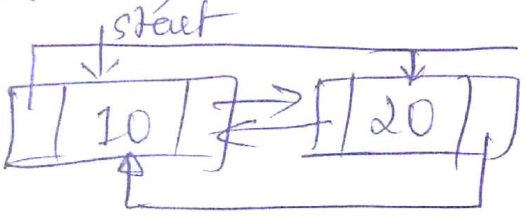
→ By visiting the llink field of start node, we can reach the last node of the list.

→ Then mark its previous node to perform delinking & connecting the previous node start node of the list.

Example: Consider a Circular DLL of 3 nodes




```
last = start -> llink;
prev = last -> llink;
prev -> rlink = last -> rlink;
start -> llink = prev;
last -> llink = last -> rlink = NULL;
Print 'last -> data'
free(last)
```



→ Linked Stacks & Queues

① Implementation of a Stack using singly linked list.

→ We have discussed the concept of linked list in previous modules, It is LIFO data structure.
→ Here insertion is done from one end & deletion is performed from same end.
→ Therefore, we can utilize the singly linked list functions to perform this operation:

a) insert_front() ⇒ works as a push() fn.
b) delete_front() ⇒ works as a pop() fn.
c) display() ⇒ displays status of stack.

→ Write a C program to implement a stack using singly linked list

```
#include <stdio.h>
#include <stdlib.h>
struct node
{   int data;
    struct node *link;
};
typedef struct node *NODE;
NODE stack = NULL;
void insert_front();
void delete_front();
void display();
int main()
{ int choice, n, i;
  do
  { printf(" Stack Operation using SLL \n");
```

```
printf("1:Push \n 2:Pop \n 3:Display \n");
printf("Enter your choice \n");
scanf("%d", &choice);
switch (choice)
{
    case 1 : [redacted]
             insert_front();
             break;
    case 2 : if (start == NULL)
             printf("Stack Underflow \n");
             else delete_front();
             break;
    case 3 : if (start == NULL)
             printf("Stack Empty \n");
             else display();
             break;
    case 4 : exit(0);
    default : printf("Invalid Choice \n");
              break;
}
} while (choice != 4);
}
void insert_front()
{
    NODE temp; int data;
    printf("Enter data for push operation \n");
    scanf("%d", &data);
    temp -> data = data;
    temp -> link = NULL;
    if (start == NULL)
    {
        temp -> link = NULL;
        start = temp;
    }
```

```
        else
        {
            temp -> link = start;
            start = temp;
        }
}

void delete_front()
{
    NODE temp;
    if(start == NULL)
    {
        printf("stack underflow\n");
    }
    else
    {
        temp = start;
        start = start -> next;
        printf("Popped item is %d\n", temp -> data);
        free(temp);
    }
}

void display()
{
    int count = 0;
    NODE temp;
    temp = start;
    printf("stack status is\n");
    while(temp != NULL)
    {
        printf("%d\n", temp -> data);
        temp = temp -> link;
        count++;
    }
    printf("Number of elements in Stack are %d\n", count);
}
```

Output :

```
Stack Opeations using SLL.
1: Push
2: Pop
3: Display
4: Exit
Enter your choice
1
Enter item to be pushed
10
1:
2:
3:
4:
Enter your choice
2
Popped item is 10
1:
2:
3:
4:
Enter your choice
3
stack is Empty.
1:
2:
3:
4:
Enter your choice
4
```

(2) Implementation of Queue using Linked List

→ The concept of linked list can be used to implement as a Queue.

→ A queue, works on a FIFO data structure. i.e. First In First Out, the item inserted first should be removed first.

→ We insert from rear end & delete from front end.

→ This operation of Queue can be performed by using the following functions:

a) insert_rear() ⇒ works as insert into Queue.
b) delete_front() ⇒ works as delete from Queue.
c) display() ⇒ displays status of Queue.

→ C program to implement Queue as SLL.

```
#include<stdio.h>
#include<stdlib.h>
struct node
{  int data;
   struct node *link;
};
typedef struct node *NODE;
NODE start=NULL;
void insert_rear();
void delete_front();
void display();
int main()
{  int choice;
```

```
do
{
    printf("Queue Opeations using SLL \n");
    printf("1:Insert \n2:Delete \n3:Display \n4:Exit\n");
    printf("Enter Your choice \n");
    scanf("%d", &choice);
    switch(choice)
    {
        case 1 : insert_rear();
                 break;
        case 2 : delete_front();
                 break;
        case 3 : display();
                 break;
        case 4 : exit(0);
        default: printf("Invalid choice\n");
    }
while(choice!=4);
return 0;
}

void insert_rear()
{
    int data;
    NODE temp, q;
    printf("Enter data to insert \n");
    temp = (struct node*) malloc(sizeof(struct node));
    scanf("%d", &data);
    temp -> data = data;
    temp -> link = NULL;
    if(start == NULL)
        start = temp;
    else
    {   q = start;
        while(q -> next != NULL)
            q = q -> next;
        q -> next = temp;
    }
}
```

```
void delete_front()
{
    NODE temp;
    if(start == NULL)
        printf("Queue Underflow \n");
    else
    {
        temp = start;
        start = start -> next;
        printf("Deleted item is %d \n", temp->data);
        free(temp);
    }
}

void display()
{
    int count = 0;
    NODE temp;
    temp = start;
    printf("Queue Status is \n");
    while(temp != NULL)
    {
        printf("%d \n", temp->data);
        temp = temp->link;
        count++;
    }
    printf("Number of Items in Queue are %d \n", count);
}
```

Output

```
Queue Operations using SLL
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice
1
Enter data to insert
5
```

```
1:
2:
3:
4:
Enter your choice: 1
Enter data to insert: 10
1
2
3
4
Enter your choice: 3
The Queue status is
5 10
1
2
3
4
Enter your choice: 2
Deleted item is 5
1
2
3
4
Enter your choice: 3
10
1
2
3
4
Enter your choice: 4.
```

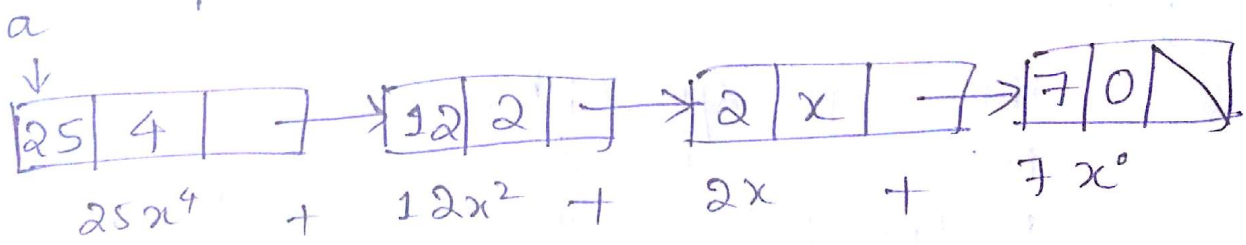
Polynomial Representation

→ The polynomial can be represented in linked list as a node that contains coefficient & exponent stored in a node, along with a pointer to the next term.

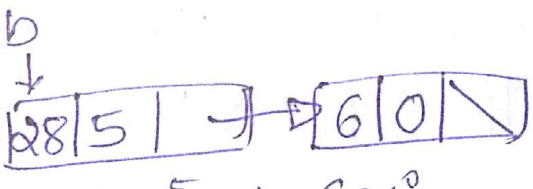
→ A node of polynomial can be declared as shown below:

```
struct node
{
    int coeff;      // coefficient of term.
    int exp;        // exponent of term
    struct node *link;   // points to next term.
};
typedef struct node *NODE;
NODE a, b;      // a points to 1st polynomial..
                // b points to 2nd ——"——.
```

→ Consider a polynomial $a = 25x^4 + 12x^2 + 2x + 7$. can be represented as linked list as shown:



→ $b = 28x^5 + 6$ can be represented as follows:



→ Polynomial Operations

1. Creation of Polynomial
2. Representing a Polynomial / Displaying Polynomial
3. Adding Two Polynomials.

1. Creation of Polynomial

→ Creation of polynomial is same as that of creating the new node in one way list.

→ First, dynamically allocate the memory for new term of the polynomial.

→ Store the coefficient & exponent value, that is power of x in the term.

→ Link the term to the previous terms of the polynomial list.

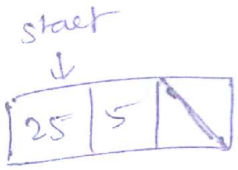
→ Read the node of the polynmial as a term of polynomial.

```
NODE read()
{
    int coeff, xpower, n;
    NODE start = NULL, t;
    printf("Enter number of terms \n");
    scanf("%d", &n);
    for(i=0; i<n; i++)
    {
      printf("Enter a term (coeff, xpower) : \n");
      scanf("%d%d", &coeff, &xpower);
      t.coeff = coeff;
      t.xpower = xpower;
      t.link = NULL;
    start = attach(t, start); } return start;
```

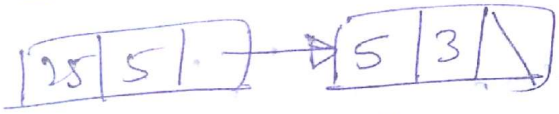
→ Once we read the components of a term, attach the term to existing polynomial.

```
NODE attach(t, start)
{
    NODE temp, cur
    temp = (struct node *) malloc (struct node));
    temp -> coeff = t.coeff;
    temp -> exp = t.exp;
    temp -> link = NULL;
    if (start == NULL)
        return temp;
    cur = start;
    while (cur -> link != NULL)
    {
        cur = cur -> link;
    }
    cur -> link = temp;
    return start;
}
```

Example:

→ 

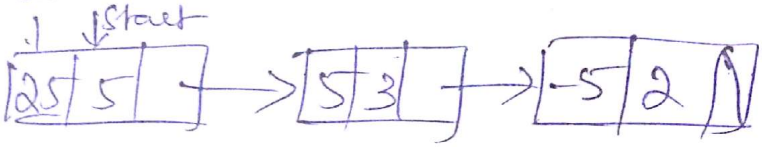
One term $25x^5$

→ 

Attaching 2nd term.

$\therefore 25x^5 + 5x^3$.

→ Attach another 3rd term



$25x^5 + 5x^3 - 5x^2$.

→ The read function & attach function performs reading of a term & attaching the terms to existing polynomial respectively.

(2) Representing a Polynomial / Display a Polynomial

→ The polynomial is represented in the following form, by the Computer.

→ Consider $q = 25x^5 + 5x^3$, a polynomial, which can be represented as follows in computer.

25x^5 +5x^3

→ When we display a polynomial stored in computer memory, it can be displayed as represented above polynomial terms.

→ The following is the function, that displays nodes of polynomial one after other.

```
void display (NODE start)
{
    NODE temp;
    if(start == NULL)
        printf("Polynomial not created\n");
    temp = start
    while(temp != NULL)  // displays each node.
    {
        printf(" + %d x ^ %d ", temp -> coeff, temp ->
                xpower);
        temp = temp -> link;
    }
}
```

(3) Addition of Two Polynomials

→ Two polynomials a & b can be added and the resultant polynomial can be obtained based on the following rules of addition.

→ Rules to add the polynomials a & b.

(1) If the power of x is same for a term in polynomial a & in polynomial b, then the coefficients can be added. & term is attached.

(2) If the power of x is diffuent in polynomial a, attach the term to resultant polynomial.

(3) If the power of x is diffeent in polynomial b, attach the term to resultant polynomial.

→ Consider the following polynomials a & b, & an sum polynomial can be obtained by following above rules.

$a = 25x^4 + 12x^2 + 2x + 7$

$b = 28x^5 + 6$

$sum = 28x^5 + 25x^4 + 12x^2 + 2x + 13x^0$

→ In above example, for x^0 the coeffients are added. i.e. 6 & 7 ⟹ 6+7 = 13.

→ Where as for other terms, the power of x is diffent, therefore one after the other, terms are attached to the sum polynomial.

→ Once we display, the resultant polynomial will be shown as: 28x^5+25x^4+12x^2+2x^1+13x

c-function :

```
NODE addpoly(NODE a, NODE b)
{
    NODE sum=NULL;
    int coeff;
    while(a!=NULL && b!=NULL)
    {
        if(a->xpower == b->xpower)
        {
            coeff = a->coeff + b->coeff;
            if(coeff != 0)
                attach(coeff, a->xpower, sum);
            a=a->link;
            b=b->link;
        }
        else if(a->xpower > b->xpower)
        {
            sum = attach(a->coeff, a->xpower, sum);
            a=a->link;
        }
        else
        {
            sum = attach(b->coeff, p->xpower, sum)
            b = b->link;
        }
    //Attach remaining nodes of a & b.
    while(a!=NULL)
    {
        sum = attach(a->coeff, a->xpower, sum);
        a=a->link;
    }
    while(b!=NULL)
    {
        sum= attach(b->coeff, b->xpower, sum);
        b=b->link;
    }
    return sum;
}
```

⇒ Polynomial using Circular lists

→ A polynomial, whose last term's link part stores the address of starting node of the polynomial, forms a circular polynomial.

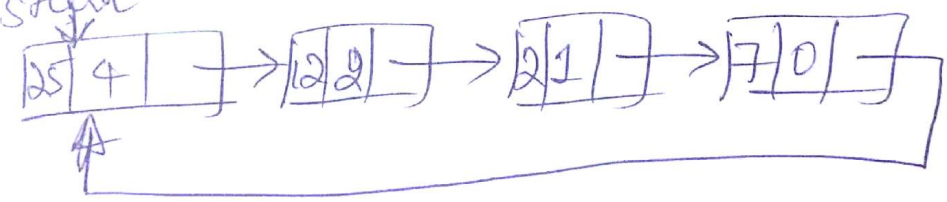
→ A declaration of a node as similar to creation of a node for polynomials. w/o circular lists.

```
struct node
{
    int coeff;
    int xpower;
    struct node *link;
};
typedef struct node *NODE;
NODE start = NULL;
```

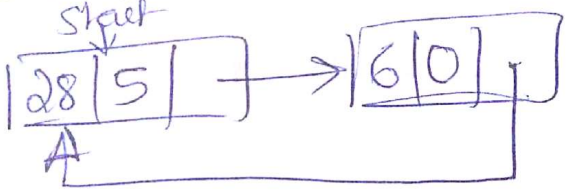
→ A polynomial in circular linked list can be shown as below:

Suppose $a = 25x^4 + 12x^2 + 2x + 7$.

a is polynomial stored in nodes as:



$b = 28x^5 + 6$



→ The advantage of circular polynomial is, we can visit the first node of polynomial, once we have visited the last node of the polynomial list.

→ Traversing a Circular Polynomial

→ To traverse & display the polynomial terms, we visit each term till the link part has value of header/start node's address in it.

→ A node that has start node's address, as the link part, is the last node in the polynomial.

```
void display(NODE start)
{
    if(start->link == start)
    {
        printf("Polynomial does not exist\n");
        return;
    }
    temp = start;
    while(temp != start)
    {
        printf("+ %d x^%d ", temp->coeff, temp->xpower);
        temp = temp->link;
    }
}
```

→ Creating a Polynomial

→ first we read the data i.e. coefficient & xpower in the node.

→ Once node is created, store start address in the link part of polynomial.

→ If other terms already exists, attach the new term to polynomial & update last term's link part to new node/term address.