Circular Linked list (CLL):-

* A circular list is a list in which the link field of the last node is made to point to the start / first node of the list.

* A circular linked list has no begining and end.

* In circular linked list No NULL pointers are used

* circular linked list can be implemented both.

- circular singly linked list
- circular Doubly linked list.

1) Circular singly linked list:-

* The last node next pointer field in a circularly singly linked list will contain the address of the first node of the list.

* The operations are:-

- creation
- Insertion
- deletion.

Routine for circular singly linked list:

```
struct node
{
int data;
struct node * next;
};
```

Creation of circular singly linked list.

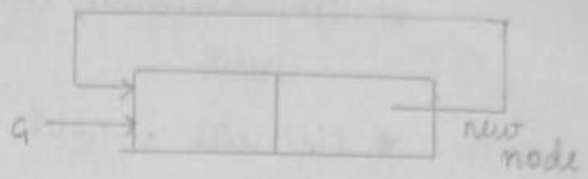
* This is done as follows

L = (NODE *) malloc (size of (NODE));

* elements are added to the list following
* similarly, an empty circular list is defined

C1 = (NODE *) malloc - (size of (NODE));

next (C1) = C1;



Insertion of circular linked list:

* There are three possible cases
  * First insertion
  * Middle insertion
  * Last insertion

First Insertion:

* If we want to insert a node at first we will search last node in the list.
* Then that node is mentioned as temp.

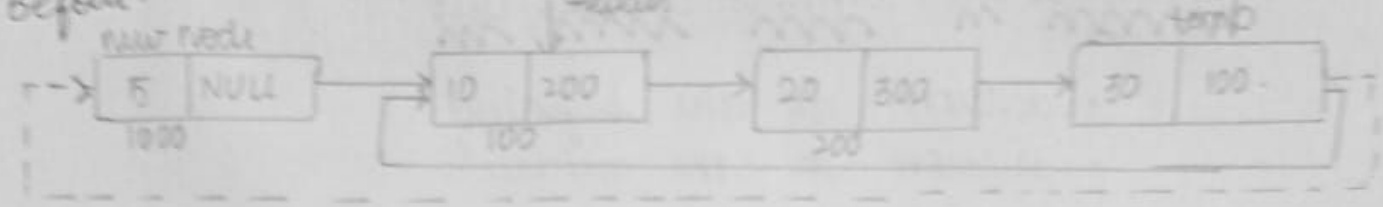
Routine:

newnode → next = header

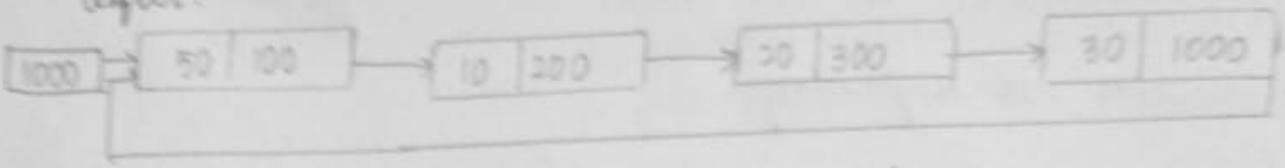
header = newnode

temp → next = header.

Before:

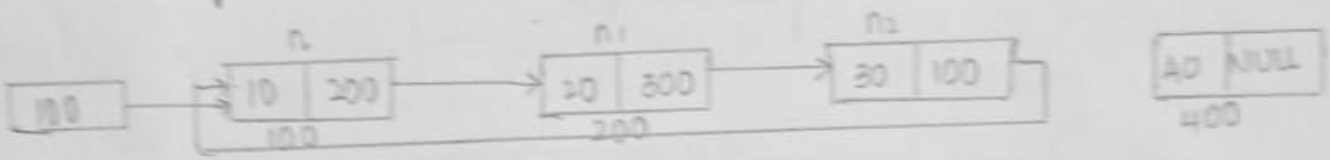


after:



. Insert a node at the last position:

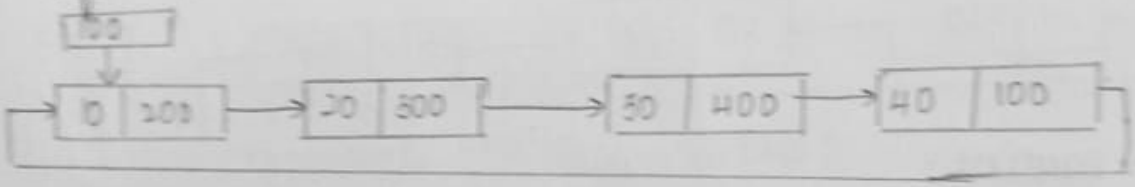
* create a newnode
* set the newnode next to itself.
* If the list is empty return newnode.
* So our new nodes must be the front.
* set tails next to newnode
* If we want to insert a newnode to other.



Routine: $n_2 \rightarrow next = newnode$

$newnode \rightarrow next = head$;

after insert we have:



. Insertion a node at middle:

If we want to insert a newnode as in between n_1 & n_2 other.

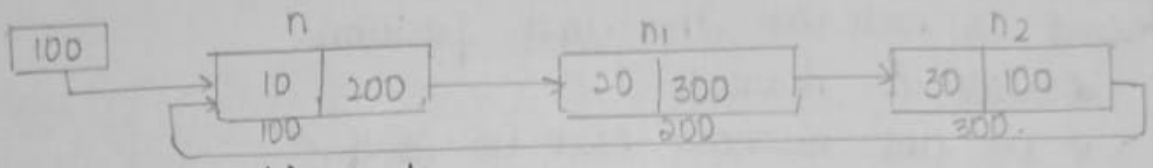
Routine: $n_1 \rightarrow next = newnode$;

$newnode \rightarrow next = n_2$;

Deletion in Circular Linked list:

- deletion at the begining
- deletion at the middle
- deletion at the end

1) deletion at beginning:-



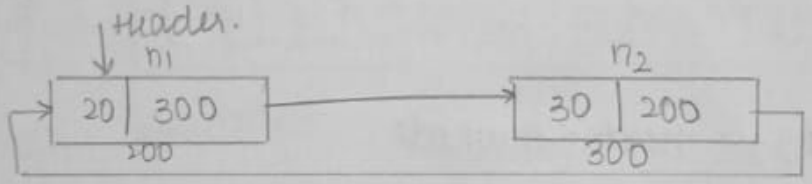
delete this node.

Routine: temp = head

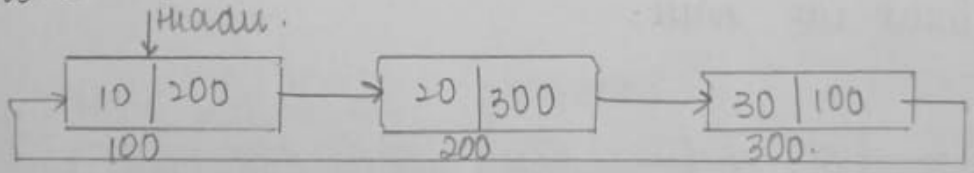
head = head → next

n_2 → next = head

free (temp);



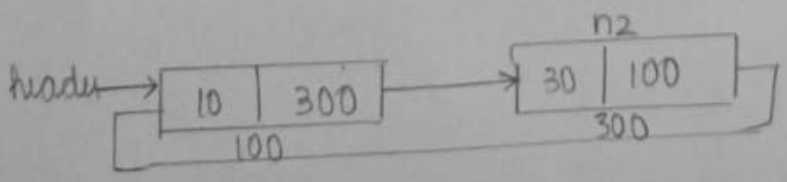
2) delete at middle:



Routine:

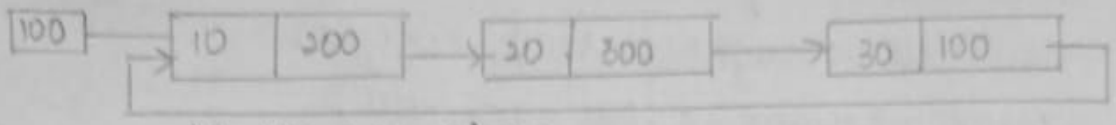
head → next = n_2;

free (n_1);



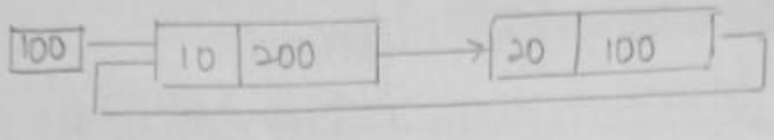
3) delet last node:

If we want to delete last node n_2 then.

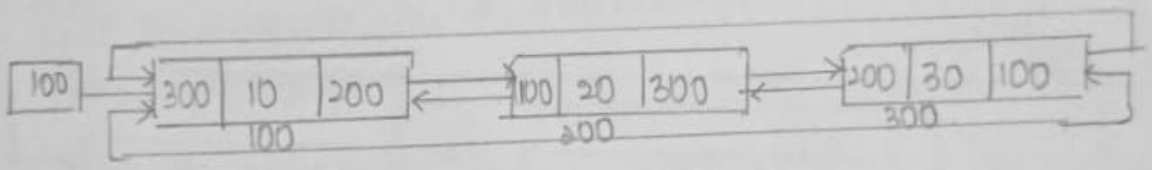


if we routine: $n_1 \rightarrow next = head$

free (n_2);



→ Circular Doubly Linked list:



basic operations: creation
deletion
Insertion

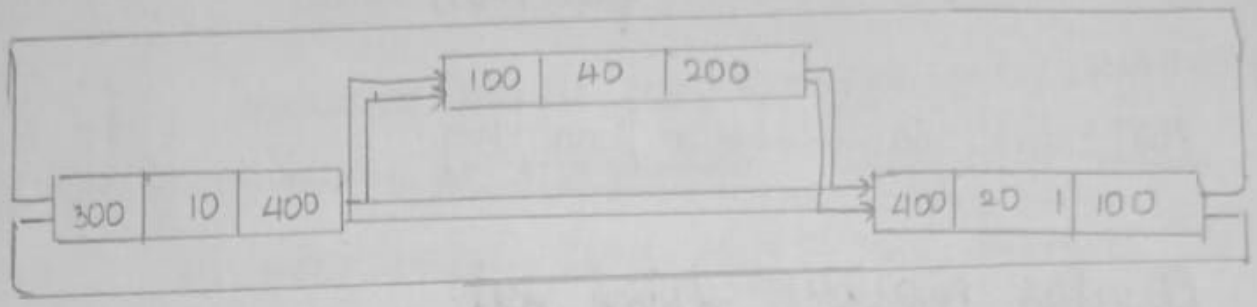
1. creation:

```
newnode = get node();
start = newnode;
newnode → left = start;
newnode → right = start;
newnode → left = start → left
newnode → right start;
start → left → right = newnode;
start → left = new node;
```

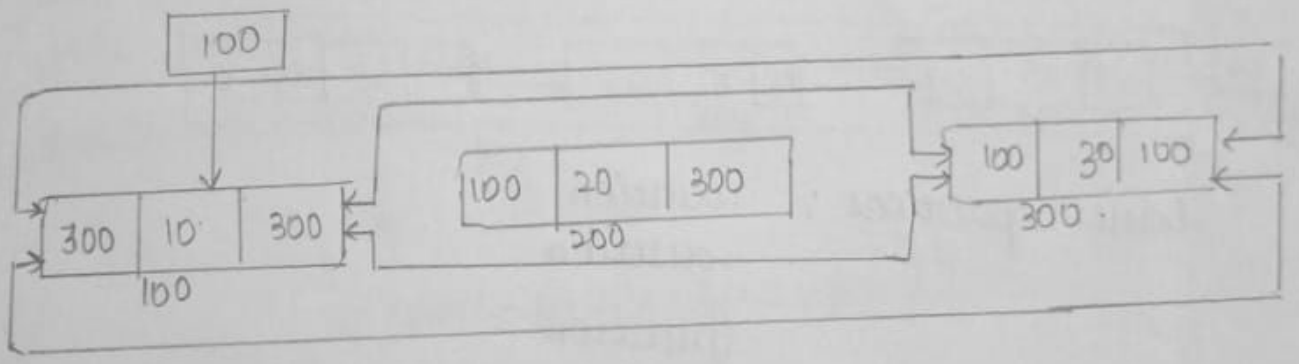
2) Insertion:

Routine:

```
newnode → left = temp;
newnode → right = itemp → right;
itemp → right → left = newnode;
itemp → right = newnode;
```



3) Deletion:



Routine:

```
itemp = istart;
print itemp → data;
itemp = itemp → right;
while (itemp != start)
{
print itemp → data;
itemp = itemp → right;
}
```

Subtraction of two polynomial:

```
void sub()  poly * ptr1, * ptr2 , * newnode ;
            ptr1 = list 1;
            ptr2 = list 2;
            while ( ptr1 != NULL && ptr2 != NULL)
            {
          newnode = (struct poly *) malloc ( size of (struct poly));
if (ptr1 -> power == ptr2 -> power)
        {
          newnode -> coeff = (ptr1 - coeff ) - (ptr2 -> coeff );
          newnode -> power = ptr1 -> power;
newnode -> next = NULL;
list 3 = create (list 3 , newnode);
ptr1 = ptr1 -> next;
ptr2 = ptr2 -> next;
                }
                else
                {
                if (ptr1 - power > ptr2 - power)
              {
              newnode -> coeff = ptr1 -> coeff;
              newnode -> power = ptr1 -> power ;
              newnode -> next = NULL ;
              list 3 = create (list 3, newnode);
              ptr1 = ptr1 -> next ;
            }
        else
        {
          newnode -> coeff ( ptr2 -> coeff );
          newnode -> power = ptr2 -> power;
```

Queue ADT:

The collection of elements in which the elements can be inserted by one end called rear of elements can be deleted by other end called front.

operation: * Insert
* Delete

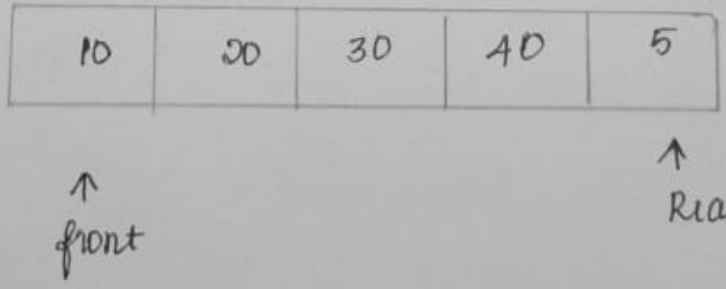
* Queue is nothing but the collection of items both the ends of the queue can having their own functional

* The Queue is also called as FIFO.
(First in first out).

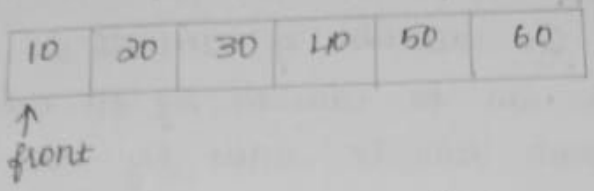
C representation of queue:

Routine:
```
struct queue
{
int queue [size];
int front ;
int rear;
};
```

Insertion: The insertion of any element in the queue always take place from the rear end.

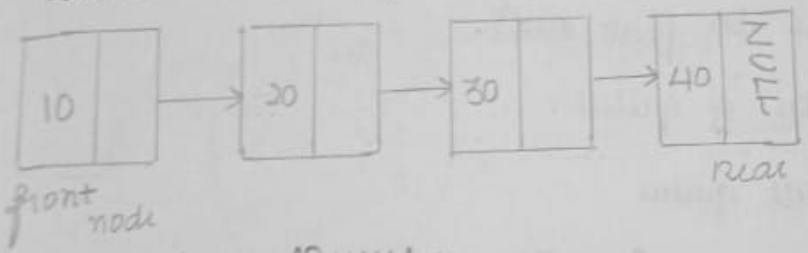


Representation of queue using array:



Implementation of queue ADT

The main advantage of using linked representation of queue is there is no limit on the size of the queue. We can insert many elements as we want in the queue by creating elements as required no of nodes.

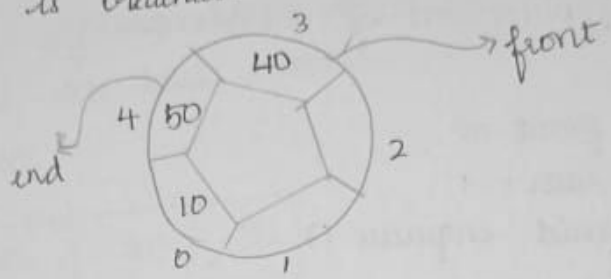


Circular Queue:

* Circular queue is another form of linear queue in which the last position is connected to the first position of the list. Initially, the front and rear ends are at the same time.

* when we insert an element the rear pointer moves one by one. until the front end is reached

* If the next position of rear is next, the queue is fully occupied by this we can't add data when we delete the elements in the front pointer moves one by one until the rear pointer is reached.



* There is a formula which has to be applied for setting the front and rear position

rear = (rear + 1)% size

front = (front + 1)% size

rear = (rear + 1)% size = (4 + 1) % 5

rear = 0.

So we can store the element 10 at 0th location similarly, while deleting the element

front = (front + 1) % 5

front = 1

So, delete the element at 1st location i.e, element 50.

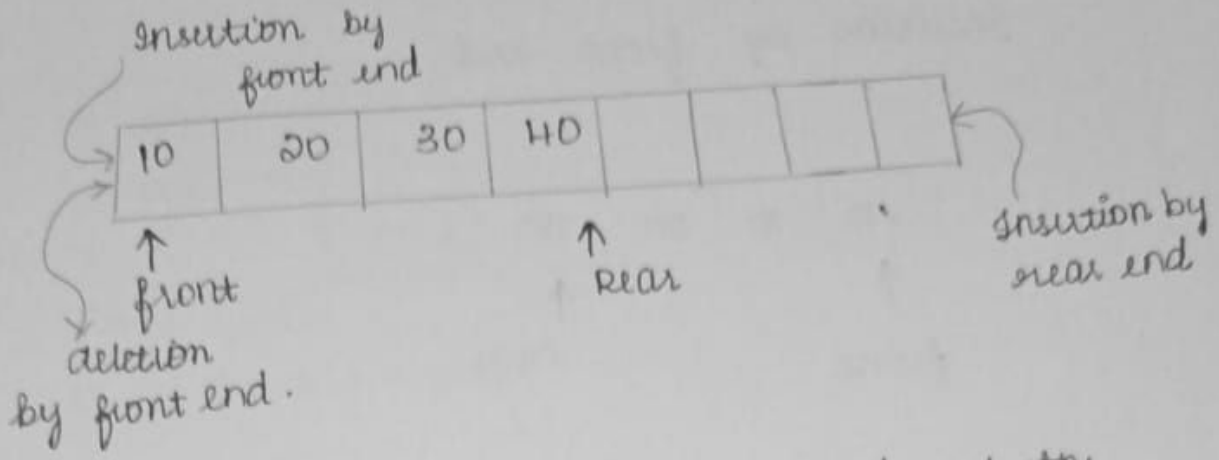
Array Implimentation of circular queue

Routine:

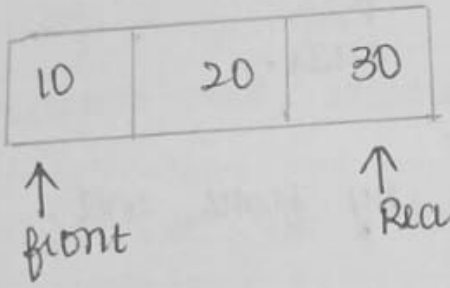
```
front = 0
rear = -1
void enqueue ()
{
if (rear = maxsize)
printf ("queue is full");
else;
queue [rear] = value;
}
```

Dequeue:

Doubly ended queue is a queue in which insertion and deletion both can be done by both the ends.



* As we know normally we insert the elements by rear and delete the elements from front end. We have inserted the element 10, 20, 30, by rear end



* If we wish to insert any element front end then first we have to shift all the elements to the right.

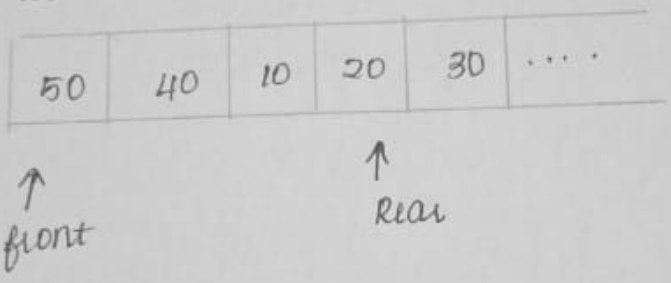
For example:

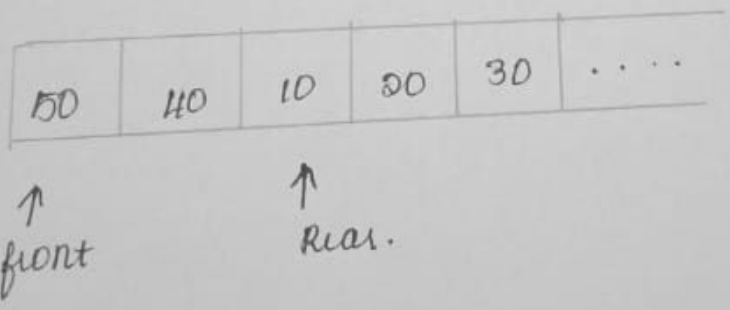
* If we want to insert 40 by front end then the dequeue will be.

```
{
*front = 0;
*rear = 0;
}
else {
if (*front == size)
*front = 12
else
*front = *front + 1;
}
}
```

deletion by rear end:

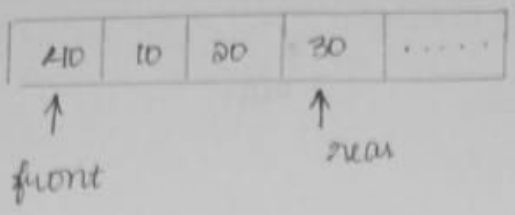
We can delete the element from Rear



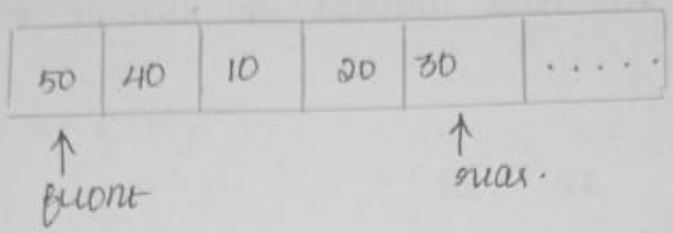


* We can place -1 for the element which has to be deleted.

Insution by front end



Insution 40 in front end.



Insutid 50 in the queue by front end.

Routine:

```
void dequeue()
{
int data;
if (rear == -1)
printf ("Queue is empty")
else
{
data = queue (* front);
if (* front == * rear);
```