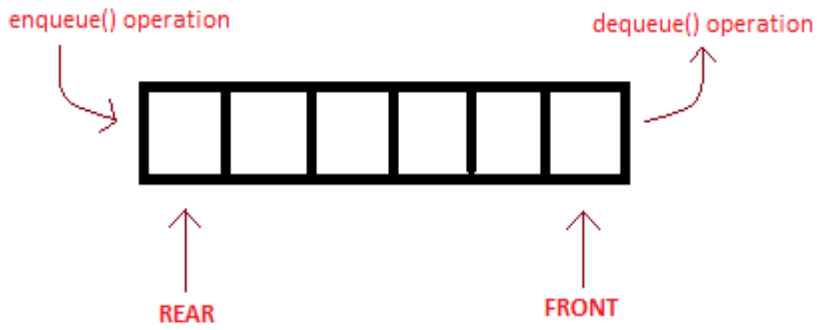
QUEUES

Queue is a linear Data structure.

Definition: Queue is a collection of homogeneous data elements, where insertion operation is performed at rear end and deletion operation is performed at front end.

- The insertion operation in Queue is termed as ENQUEUE.
- The deletion operation in Queue is termed as DEQUEUE.
- In the Queue the ENQUEUE (insertion) operation is performed at REAR end and DEQUEUE (deletion) operation is performed at FRONT end.
- Queue follows FIFO principle i.e. First In First Out principle i.e. an element First inserted into Queue, that element only First deleted from Queue.

Diagrammatic Representation of Queue



Representation of Queue

A Queue can be represented in two ways

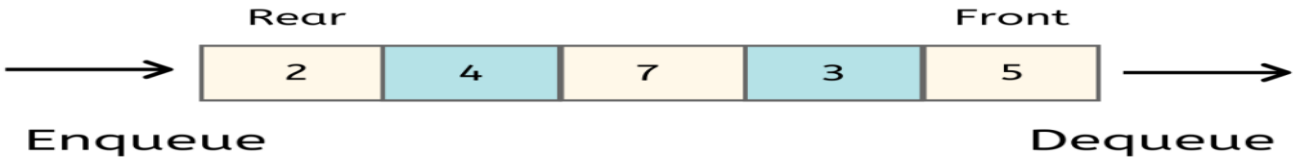
1. Using arrays
2. Using Linked List

1. Representation of Queue using arrays

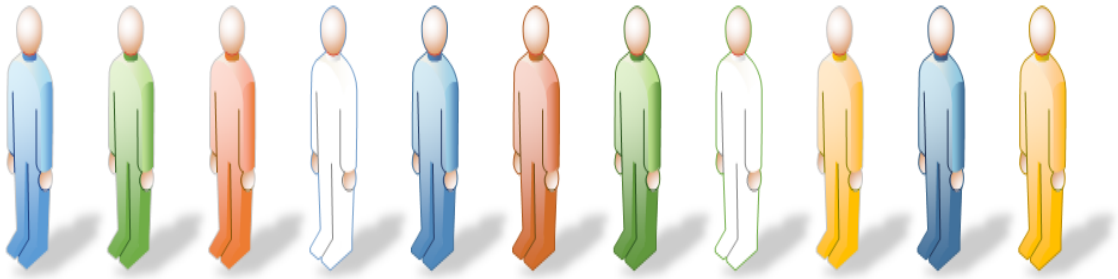
A one dimensional array Q[0-SIZE-1] can be used to represent a queue.

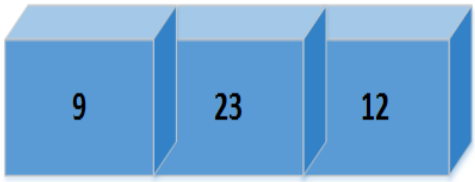
Queue

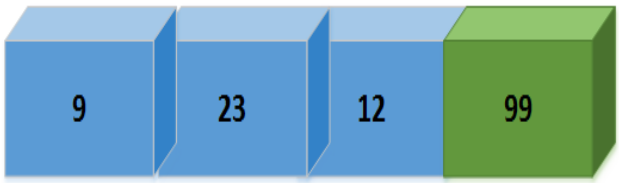
insertion and deletion happens at different ends



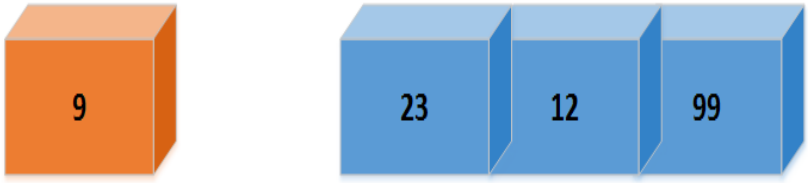
First in First out
(FIFO)



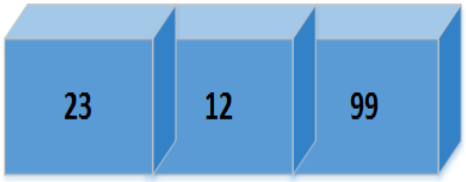




Enqueue 99 from Rear



Dequeue 9 from Front



Final

Array representation of Queue

In array representation of Queue, two variables are used to indicate two ends of Queue i.e. front and rear end.

Initial condition: rear=-1 & front=0

Queue overflow: Trying to perform ENQUEUE (insertion) operation in full Queue is known as Queue overflow.

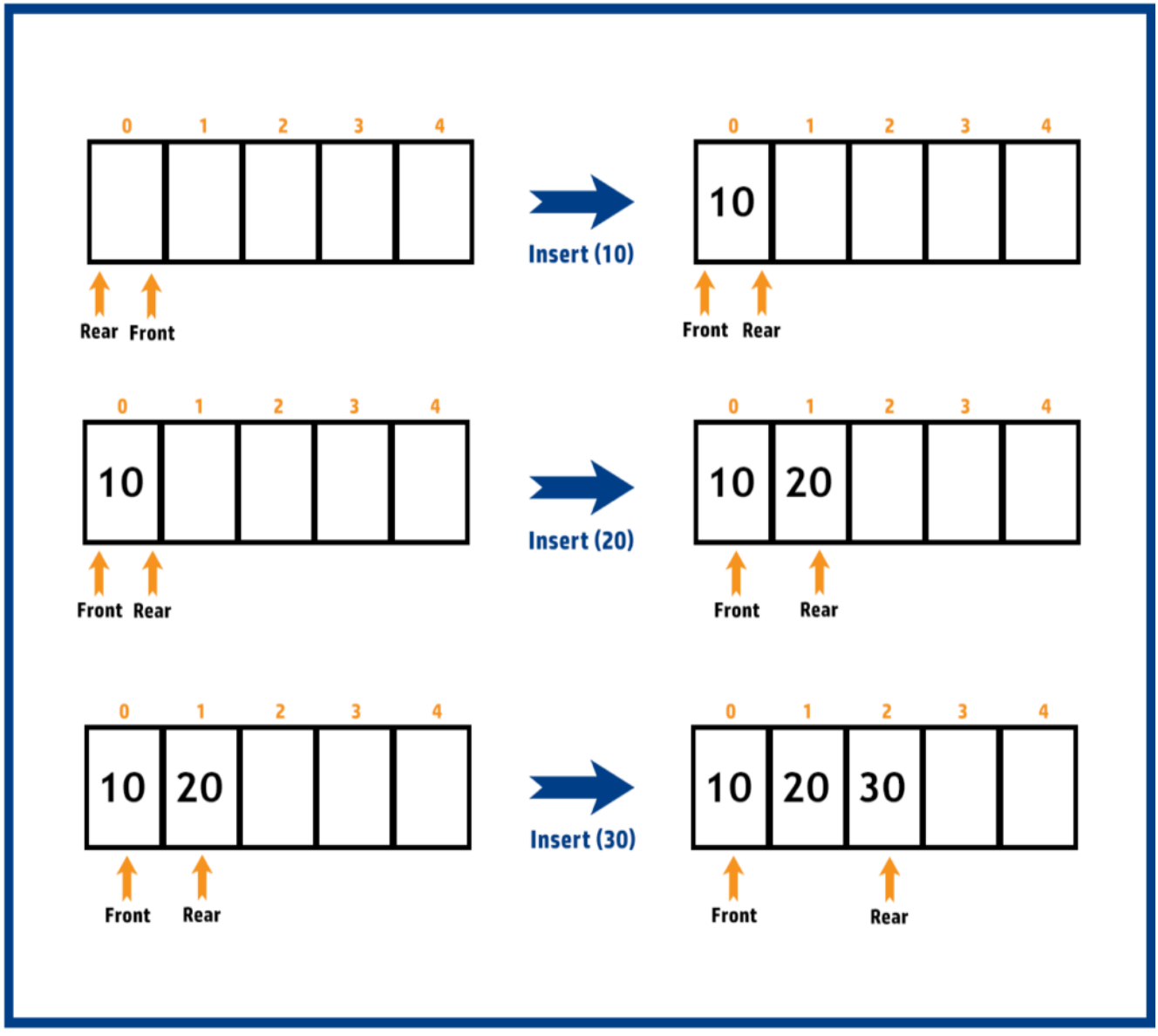
Queue overflow condition is rear = = ARRAYSIZE-1

Queue Underflow: Trying to perform DEQUEUE (deletion) operation on empty Queue is known as Queue Underflow.

Queue Underflow condition is rear<front

Operation on Queue

1. ENQUEUE : To insert element in to Queue
2. DEQUEUE : To delete element from Queue



Algorithm Enqueue(item)

Input:item is new item insert in to queue at rear end.

Output: Insertion of new item at rear end if queue is not full.

1.if(rear= = ARRAYSIZE-1)

a) print(queue is full, not possible for enqueue operation)

2.else

i) read element
ii)rear++
iii)queue[rear]=ele

End Enqueue

While performing ENQUEUE operation two situations are occur.

1. if queue is empty, then newly inserting element becomes first element and last element in the queue. So Front and Rear points to first element in the list.

2. If Queue is not empty, then newly inserting element is inserted at Rear end.

Algorithm Dequeue( )

Input: Queue with some elements.

Output: Element is deleted from queue at front end if queue is not empty.

1.if(rear<front)

a) print(Queue is empty, not possible for dequeue operation)

2.else

i)ele=queue[front]
ii)print(Deleted element from queue is ele)
iii)front++;

End Dequeue

While performing DEQUEUE operation two situations are occur.

1. if queue has only one element, then after deletion of that element Queue becomes empty. So if front>rear then queue is empty.

2. If Queue has more than one element, then first element is deleted at Front end.

C program to implement Queues using arrays:-

```
#include<stdio.h>
#include<conio.h>
#define ARRAYSIZE 5
void enqueue(int ele );
void dequeue( );
void display( );
int queue[SIZE],front=0,rear=-1,ele;
```

```
void main( )
{
	int ch;
	while(1)
	{
		clrscr();
		printf("\t \t Queue ADT using Arrays\n");
		printf("\n1.enqueue\n2.dequeue\n3.display");
		printf("\nenter your choice");
		scanf("%d",&ch);
		switch(ch)
		{
			case 1:printf("\nenter an element to be inserted into the queue");
				scanf("%d",&ele);
				 enqueue(ele );
				 break;
			case 2:dequeue( );
				break;
			case 3:display( );
				break;
			default:exit( );
		}
	}
}

void enqueue(int ele )
{
	if(rear==ARRAYSIZE-1)
		printf("\nQueue is overflow");
	else
	{
		rear++;
		queue[rear]=ele;

	}
}

void dequeue( )
{
	if(rear<front)
		printf("\nQueue is empty");
	else
	{
		ele=queue[front];
		printf("\nDeleted element from the queue is %d",ele);
		front++;
	}
}

void display( )
{
	int i;
	if(rear<front)
```

```
            printf("\nQueue is underflow");
        else
        {
            printf("\nThe elements of queue are\n");
            for(i=front;i<=rear;i++)
                printf("%d\t",queue[i]);
        }
}
```

2. Representation of Queue using Linked List

- Array representation of Queue has static memory allocation only.
- To overcome the static memory allocation problem, Queue can be represented using Linked List.
- In Linked List Representation of Queue, **Front** always points to **First** node in the Linked List and **Rear** always points to **Last** node in the Linked List.

The Linked List representation of Queue stated as follows.

1. Empty Queue condition is

Front = NULL and Rear = NULL

2. Queue full condition is not available in Linked List representation of Queue, because in Linked List representation memory is allocated dynamically.

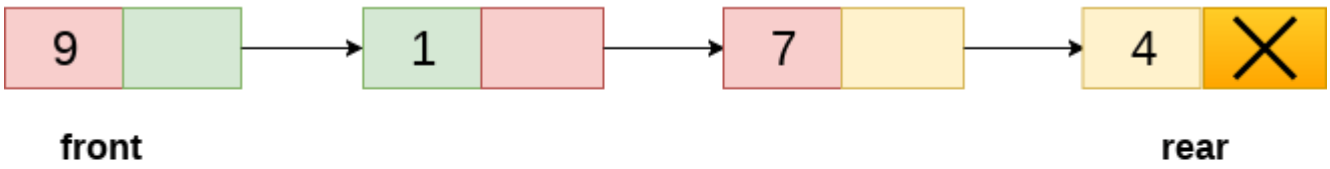
3. Queue has only one element

Front = = Rear

Operation on Linked List Representation of Queue

1. ENQUEUE : To insert an element at end
2. DEQUEUE : To delete an element from begining

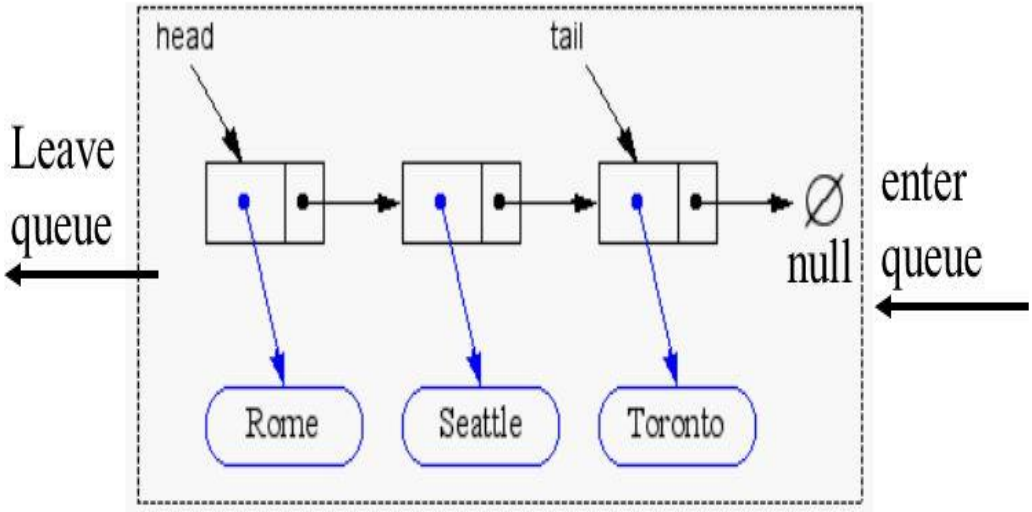
Linked list Representation of Queues:-



Linked Queue

The point of all this is…..

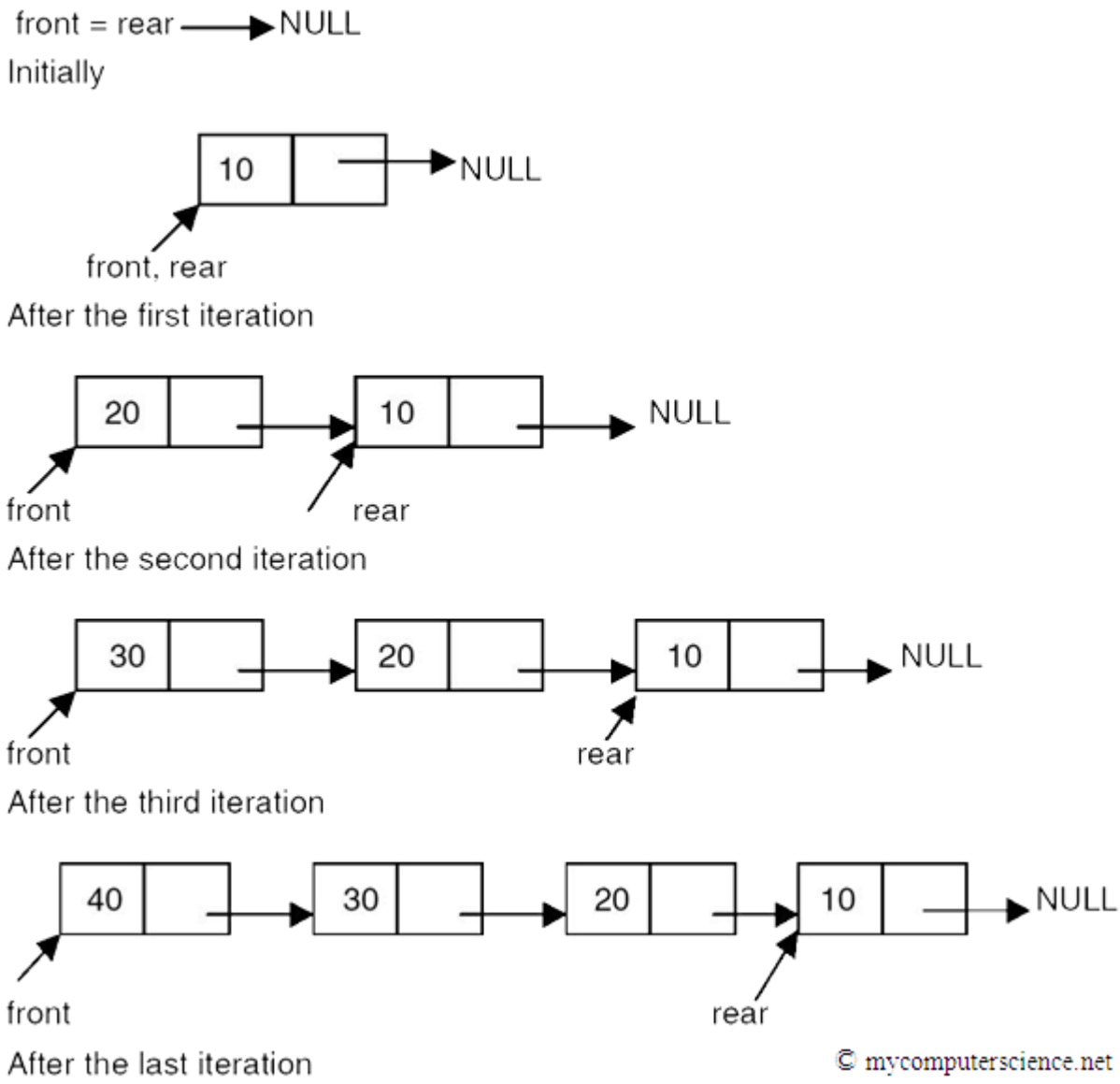
Using linked lists to implement Queue



The **head** of the list is the **front** of the queue, and the **tail** of the list is the **rear** of the queue.

Why not the opposite?

LinkedListQueue is an alternative implementation of Queue to ArrayQueue.
Unlimited capacity -- no wasted memory, no more QueueFullException



C program **to implement queues using linked list:-**

/* QUEUE ADT USING LINKED LIST */

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *next;
};
struct node *rear,*front;
void insend(int ele); /* void enqueue(int ele)*/
void delbeg();        /* void dequeue()   */
void display();
void main()
{
        int ele,choice;
```

```
	rear=front=NULL;
	while(1)
	{
		clrscr();
		printf("\t\t queue list operations\n");
		printf("1.enqueue\n");
		printf("2.dequeue\n");
		printf("3.displau\n");
		printf("4.exit\n");
		printf("enter choice\n");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1:printf("\n enetr ele ");
				scanf("%d",&ele);
				insend(ele);
				break;
			case 2:delbeg();
				break;
			case 3:display();
				break;
			case 4:exit(0);
		}
	}
}
void insend(int ele)
{
	struct node *temp,*p;
	temp=(struct node*)malloc(sizeof(struct node));
	temp->info=ele;
	temp->next=NULL;
	if(rear==NULL)
	{
		rear=temp;
		rear=front=temp;
	}
	else if(rear->next==NULL)   /* or else */
	{
		rear->next=temp;
		rear=temp;
	}
}
void delbeg()
{
	struct node *temp;
```

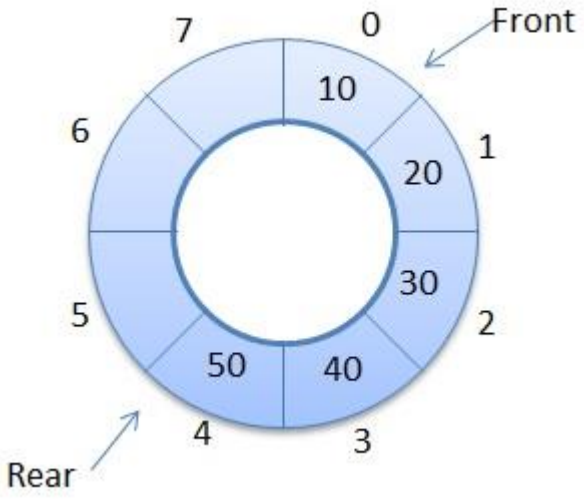
```
        if(front==NULL)
        printf("\n que is empty");
        else if(front->next==NULL)
        {
                temp=front;
                front=NULL;
                rear=NULL;
                free(temp);
        }
        else
        {
                temp=front;
                front=front->next;
                free(temp);
        }
}
void display()
{
        struct node *p;
        if(front==NULL)
        printf("\n list is empty");
        else
        {
                p=front;
                while(p!=NULL)
                {
                        printf("%d->",p->info);
                        p=p->next;
                }
        }
        getch();
}
```

Types of Queues Or Various Queue Structures

1. **Linear Queue**
2. **Circular Queues**
3. **DEQue**
4. **Priority Queue**

1. **Circular Queues:-**

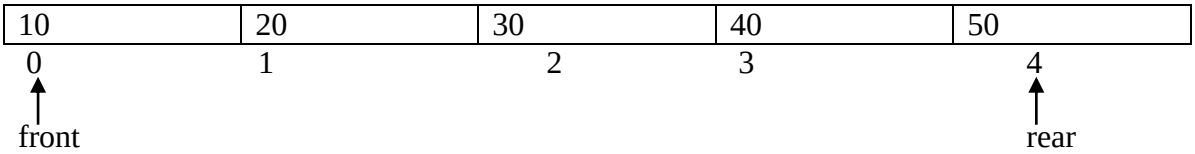
   A circular Queue is a linear data structure in which all the locations are treated as circular such that the first location cqueue[0] follows the last location cqueue[SIZE-1].



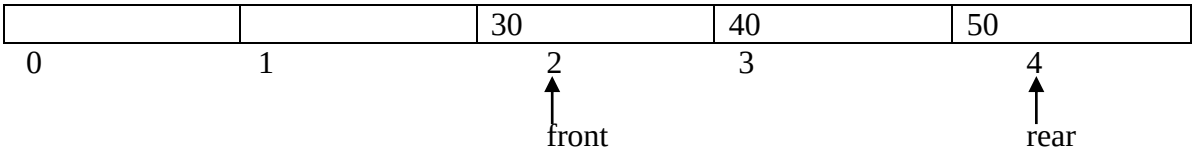
Circular queues are mainly useful to overcome the drawbacks of linear queue.

Drawbacks of linear Queue:-

Let queue contains 5 elements.



After performing 2 Dequeue operations, the queue look likes the below



If rear reaches to the end of queue, then it's not possible to insert an element into the queue even though there is a space at the beginning of the queue.

To solve the above drawback, circular queues are used.

Initial Condtion for Circular Queue is: **rear=front=-1**

Full Condition for Circular Queue: if(front==((rear+1)%arraysize))

Empty Condition for Circular Queue: rear=front=-1

Algorithm Enqueue()

Step 1. **if(front==((rear+1)%arraysize))**
print circular Queue is Full

Step 2. else if(rear==-1 && front==-1) **//inserting first element**
rear++, front++
cq[rear]=ele;

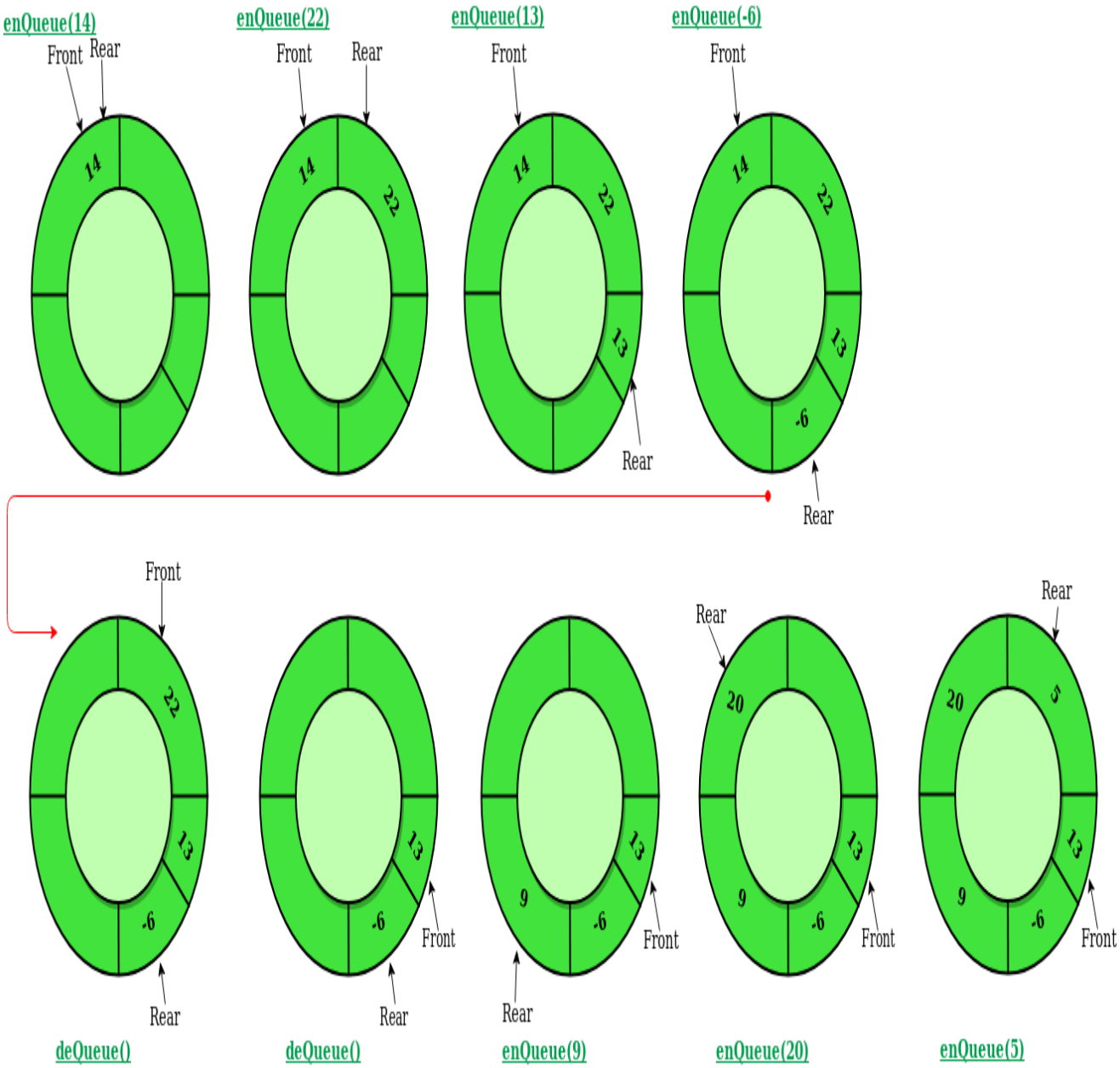
Step 3. else
rear=(rear+1)%size;
cq[rear]=ele;

Algorithm dequeue()

Step 1. **if(rear==-1 && front==-1)**
print Circular Queue is underflow

Step 2. else if(front==rear) // deleting first element
print cq[front]
rear= front=-1

Step 3. else
front=(front+1)%size;



Circular Queue ADT using Arrays

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define arraysize 5
int a[arraysize],rear=-1,front=-1;
void enque(int ele);
void deque();
void display();
void main()
{
        int ele,choice;
        clrscr();
        while(1)
```

```
	{
		clrscr();
		printf("\t\t circular que operations \n");
		printf("1.enque \n2.deque \n3.display \n4.exit\n");
		printf("enter choice\n");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1:printf("enter ele \n");
			       scanf("%d",&ele);
			       enque(ele);
			       break;
			case 2:deque();
			       break;
			case 3:display();
			       break;
			case 4:exit(0);
			         break;
		 }
	}
}
void enque(int ele)
{
	if(front==(rear+1)%arraysize)
	{
		printf("\n Circular que is full");
		getch();
	}
	else if(front==-1 && rear==-1)
	{
		rear++;
		front++;
		a[rear]=ele;
	}
	else
	{
		rear=(rear+1)%arraysize;
		a[rear]=ele;
	}
}
void deque()
{


	if(front==-1 && rear==-1)
```

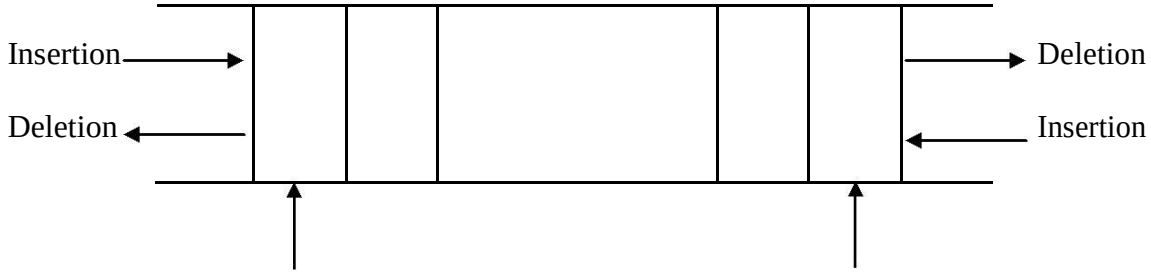
```
        printf("\n circular que is empty ");
        else if(front==rear)
          {
            printf("\n the delted eleis %dfront=%d",a[front],front);
            front=rear=-1;
            printf("front=%d",front);
          }
          else
          {
                printf("\n the delted eleis %d",a[front]);
                front=(front+1)%arraysize;
                printf("front=%d",front);
            }
            getch();
}
void display()
{
        int i;
        if(rear==-1 && front==-1)
        {
                printf("\n cir que is empty");
        }
        i=front;
        while(i!=rear)
        {
                printf("%d ",a[i]);
                i=(i+1)%arraysize;
        }
        printf("%d ",a[i]);
        getch();
}
```

2. DEQueue(Double Ended Queue):-

Another variation of queue is known as DEQue.

In DEQueue, ENQUEUE (insertion) and DEQUEUE (deletion) operations can be made at both front and rear ends.

DEQue is organized from Double Ended Queue.



Front Rear

A DEQue structure

Here DEQue structure is general representation of stack and Queue. In other words, a DEQue can be used as stack and Queue.

DeQue can be represented in two ways.

1. Using Double Linked List
2. Using a Circular Queue

Here Circular array is popular representation of DEQue.

On DEQue, the following four operations can be performed.
1.insertion at rear end.
2.deletion at front end
3.insertion at front end
4.deletion at rear end

C program to implement double ended queue(Dequeue):-

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define SIZE 5
int dequeue[SIZE];
int front=-1,rear=-1;
void display();
void insertatfront();
void insertatrear();
void deleteatfront();
void deleteatrear();
int ch,value;
void main()
{
        clrscr();
        while(1)
        {
                printf("Menu");
                printf("\n 1.Insert at Front:");
                printf("\n 2.Insert at Rear:");
                printf("\n 3.Delete at Front:");
                printf("\n 4.Delete at Rear:");
                printf("\n 5.Display");
                printf("\n 6.Exit");
                printf("\nEnter your choice");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1:insertatfront();
                               break;

                        case 2:insertatrear();
                               break;

                        case 3:deleteatfront();
                               break;

                        case 4:deleteatrear();
                               break;

                        case 5:display();
                               break;
```

```
                        case 6:exit(1);
                }
            getch();
        }
}
void insertatfront()
{
        if(front==0)
        {
                 printf("\n Queue is FULL");
        }
        else
        {
                printf("\nenter an element to be inserted into the queue");
                scanf("%d",&value);
                if(front==-1 && rear==-1)
                {
                        front++;
                        rear++;
                }
                else
                {
                        front--;
                }
                dequeue[front]=value;
        }
 }
void insertatrear()
{
        if(rear==SIZE-1)
        {
                printf("\n Queue is FULL");
        }
        else
        {
                rear=rear+1;
                printf("\n Enter a value:");
                scanf("%d",&value);
                dequeue[rear]=value;
                if(front==-1)
                        front=0;
        }
}
void deleteatfront()
{
        if(front==-1 || front>rear)
        {
                printf("\n Queue is EMPTY");
        }
        else
        {
                value=dequeue[front];
                printf("\n Deleted element from the queue is %d",value);
                front++;
        }
}
void deleteatrear()
{
        if(rear==-1)
        {
                printf("\n Queue is EMPTY");
        }
        else
        {
                value=dequeue[rear];
                printf("\n deleted element from the queue is %d",value);
```

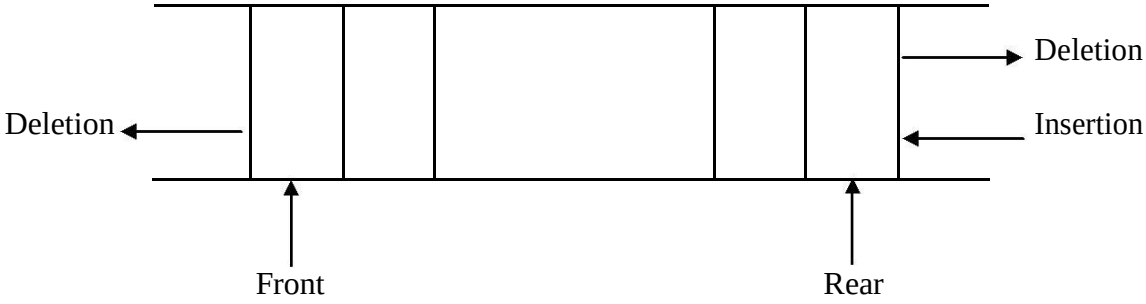
```
                if(front==rear)
                        front=rear=-1;
                else
                        rear--;
        }
}

void display()
{
        int i;
        if(front==-1)
                printf("\nQueue is empty");
        else
        {
                printf("\n The Queue is::");
                for(i=front;i<=rear;i++)
                {
                        printf("%d\t ", dequeue[i]);
                }
        }
}
```

There are two variations of DEQue known as

1. Input restricted DEQue
2. Output restricted DEQue

1. Input restricted DEQue

Here DEQue allows insertion at one end (say REAR end) only, but allows deletion at both ends.



Input restricted DEQue

2. Output restricted DEQue

Here DEQue allows deletion at one end (say FRONT end) only, but allows insertion at both ends.
DEQue is organized from Double Ended Queue.

Front Rear Output restricted DEQue