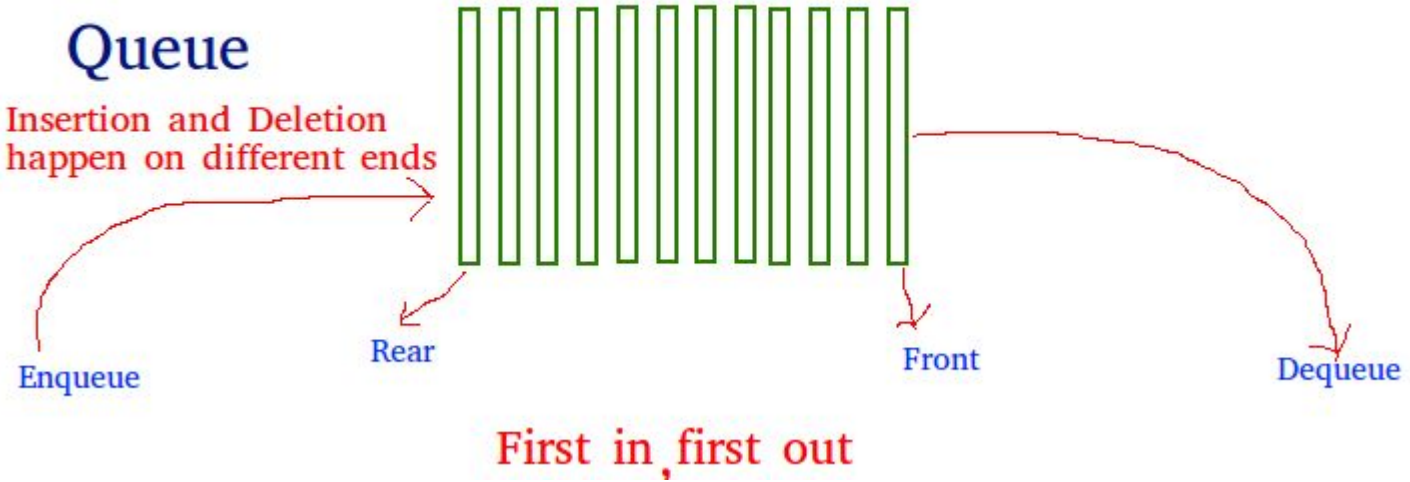
DSA Queue

Chapter 3

Queue

- It is data structure that follows operations on FIFO (First In First Out) order.
- One of the real life example of Queue as it name implies is Queue in Cinema Hall or ticket Counter.



Basic Operation on Queue

- Enqueue: Insertion of element in a queue

  It is similar with the push operation in stack. Enqueue operation is performed from one end i.e. Rear

- Dequeue: Deletion of element from a queue

  It is similar with the pop operation in stack. Dequeue operation is performed from another end i.e. front

Operations on Queue

- Enqueue(n)
- Dequeue()
- front()
- isFull()
- isEmpty()

Queue as an ADT

```
Class Queue{
    Int queue[];
    Int front=-1;
    Int rear=-1;
    enqueue();
    dequeue();
    display();
}
```

Algorithm for enqueue

- **Step 1:** IF REAR = MAX - 1
  Write Queue is full
  Go to step
  [END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]
- **Step 3:** Set QUEUE[REAR] = NUM
- **Step 4:** EXIT

Algorithm for dequeue

- **Step 1:** IF FRONT = -1 or FRONT > REAR
  Write Queue is empty
  ELSE IF : FRONT==REAR, FRONT=REAR=-1

  SET VAL = QUEUE[FRONT]
  SET FRONT = FRONT + 1
  [END OF IF]
- **Step 2:** EXIT

Advantages of Queue

- A large amount of data can be managed efficiently with ease.
- Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.
- Queues are useful when a particular service is used by multiple consumers.
- Queues are fast in speed for data inter-process communication.
- Queues can be used in the implementation of other data structures.

Disadvantages of Queue

- The operations such as insertion and deletion of elements from the middle are time consuming.
- Limited Space.
- In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
- Searching an element takes O(N) time.
- Maximum size of a queue must be defined prior.

Application of Queue

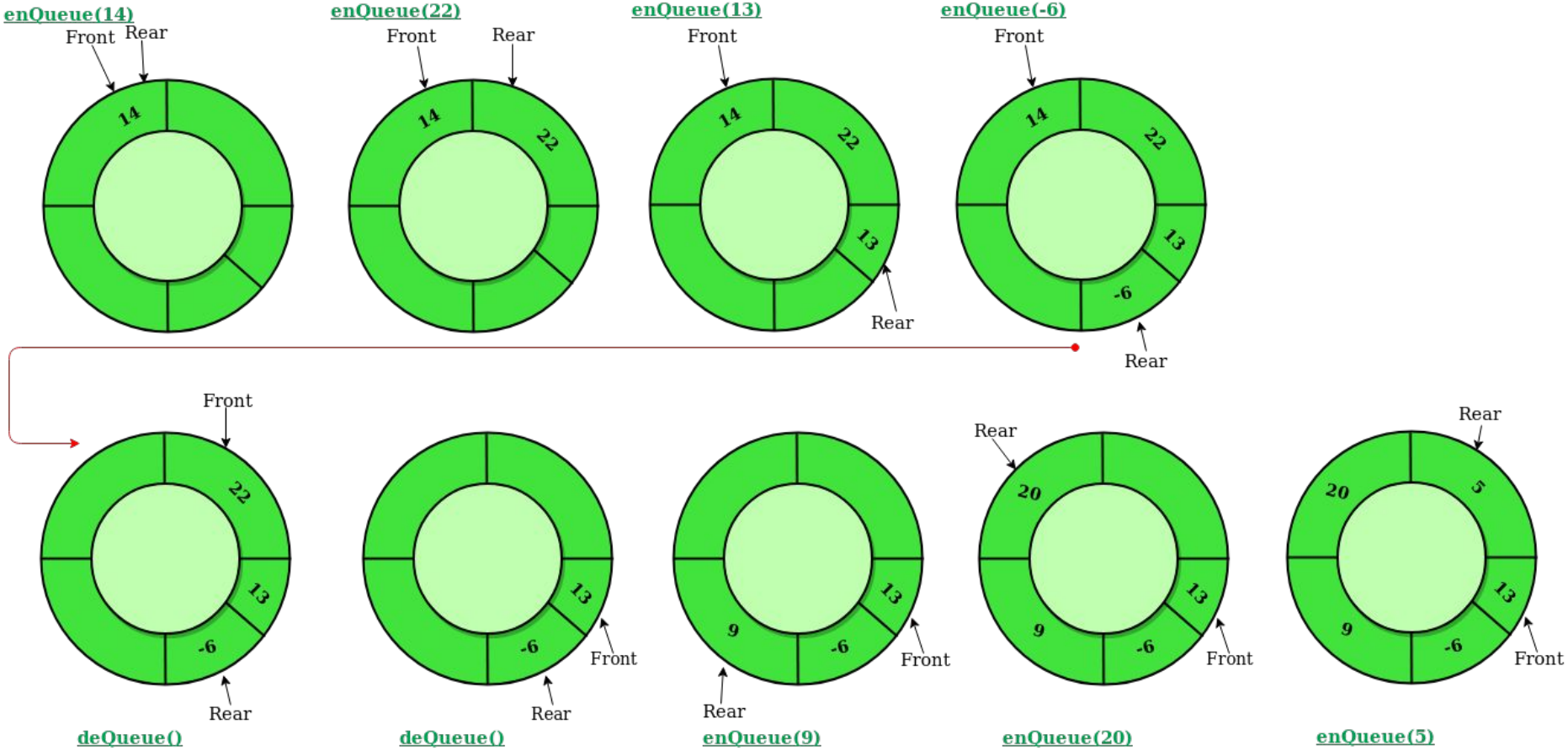
- **Multi programming:** Multi programming means when multiple programs are running in the main memory. It is essential to organize these multiple programs and these multiple programs are organized as queues.
- **Network:** In a network, a queue is used in devices such as a router or a switch. another application of a queue is a mail queue which is a directory that stores data and controls files for mail messages.
- **Job Scheduling:** The computer has a task to execute a particular number of jobs that are scheduled to be executed one after another. These jobs are assigned to the processor one by one which is organized using a queue.
- **Shared resources:** Queues are used as waiting lists for a single shared resource.

Real-time application of Queue:

- ATM Booth Line
- Ticket Counter Line
- Key press sequence on the keyboard
- CPU task scheduling
- Waiting time of each customer at call centers.

Circular Queue

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.


enQueue(14)
Front
Rear
14
enQueue(22)
Front
Rear
14
22
enQueue(13)
Front
14
22
13
Rear
enQueue(-6)
Front
14
22
13
-6
Rear
Front
22
13
-6
Rear
deQueue()
13
-6
Front
Rear
deQueue()
9
13
-6
Front
Rear
enQueue(9)
Rear
20
9
13
-6
Front
enQueue(20)
Rear
20
5
9
13
-6
Front
enQueue(5)

Circular Enqueue

1. Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
2. If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.

Circular Enqueue

```
Void Enqueue(int x)
{       if(front==-1 && rear==-1)
                {       front=rear=0;
                        queue[rear]=X;
                }
        elseif((rear+1)%N==front)
                {       printf(“Circular Queue is full”);
                }
        else{
                rear=(rear+1)%N;
                queue[rear]=X;
        }
```

Circular Dequeue

1. Check whether queue is Empty means check (front==-1).
2. If it is empty then display Queue is empty. If queue is not empty then step 3
3. Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

Circular Dequeue

```
Void dequeue(){
        if(front==-1 && rear ==-1)
                printf(“Circular queue is empty”);
        elseif(front==rear)
                front=rear=-1;
        else
                front=(front+1)%N;
}
```

Applications of Circular Queue:

- In a page replacement algorithm, a circular list of pages is maintained and when a page needs to be replaced, the page in the front of the queue will be chosen.
- Computer systems supply a holding area for maintaining communication between two processes or two programs. This memory area is also known as a ring buffer.
- CPU Scheduling: In the Round-Robin scheduling algorithm, a circular queue is utilized to maintain processes that are in a ready state.

Real-time Applications of Circular Queue:

- Months in a year: Jan – Feb – March – and so on upto Dec- Jan – . . .
- Eating: Breakfast – lunch – snacks – dinner – breakfast – . . .
- Traffic Light is also a real-time application of circular queue.

Advantages of Circular Queue:

- It provides a quick way to store FIFO data with a maximum size.
- Efficient utilization of the memory.
- Doesn’t use dynamic memory.
- Simple implementation.
- All operations occur in O(1) constant time.

Disadvantages of Circular Queue:

- In a circular queue, the number of elements you can store is only as much as the queue length, you have to know the maximum size beforehand.

Priority Queue

Priority Queue is an extension of the Queue data structure where each element has a particular priority associated with it.

It is based on the priority value, the elements from the queue are deleted.

For our program, we are applying a procedure that the higher the value of the number, higher the priority

Priority Enqueue

```
void insert_by_priority(int data)
{    if (rear >= MAX - 1)    {
        printf("\nQueue overflow no more elements can be inserted");
        return;    }
    if ((front == -1) && (rear == -1))    {
        front++;
        rear++;
        pri_que[rear] = data;
        return;    }
    else
        check(data);
    rear++;
}
```

Priority Dequeue

```
void delete_by_priority(int data){
    int i;  // Code to check whether queue is empty or not.
    for (i = 0; i <= rear; i++)    {
        if (data == pri_que[i])       {
            for (; i < rear; i++)           {
                pri_que[i] = pri_que[i + 1];           }
        pri_que[i] = -99;
        rear--;
        if (rear == -1)            front = -1;
        return;       }
    }
    printf("\n%d not found in queue to delete", data); }
```

Double Ended Queue

The deque stands for Double Ended Queue.

Deque is a linear data structure where the insertion and deletion operations are performed from both ends.

We can say that deque is a generalized version of the queue.

Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows –

Representation of deque

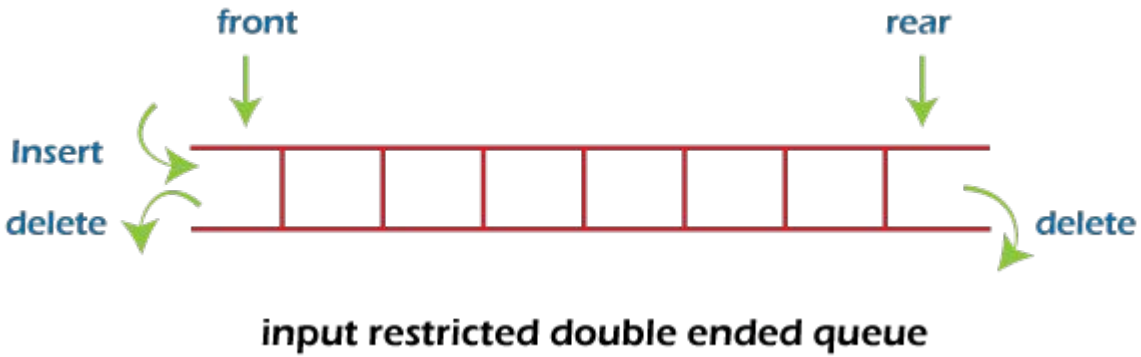
Types of Double ended Queue

1. Input Restricted

2. Output Restricted

Input Restricted

In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

Output Restricted Deque

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.

Output restricted double ended queue

Operations on Deque

1. Insert at front
2. Delete at front
3. Insert at rear
4. Delete at rear

Pseudo Code for insert at front

```
void insert_front(int x)    {
    if((f==0 && r==size-1) || (f==r+1)) {
        printf("Overflow");
    }
    else if((f==-1) && (r==-1))     {
        f=r=0;
        deque[f]=x;
    }
else if(f==0)    {
        f=size-1;
        deque[f]=x;
    }
    else    {
        f=f-1;
        deque[f]=x;
    }   }
```

Pseudo Code for insert at rear

```
void insert_rear(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("Overflow");
    }
    else if((f==-1) && (r==-1))
    {
        r=0;
        deque[r]=x;
    }
    else if(r==size-1)
    {
        r=0;
        deque[r]=x;
    }
    else
    {
        r++;
        deque[r]=x;
    }
}
```

Pseudo Code for insert at rear

```
void delete_front()    {
    if((f==-1) && (r==-1))    {
        Display "Deque is empty" ;
    }
    else if(f==r)
    {
        Display "\nThe deleted element is
deque[f];
        f=-1;
        r=-1;
    }
    else if(f==(size-1))
    {
        Display “The deleted element is
deque[f]”;
        f=0;
    }
    else
    {
        Display “The deleted element is
deque[f]”;
        f=f+1;
    }
}
```

Pseudo Code for delete at front

```
void delete_front()    {
    if((f==-1) && (r==-1))    {
        Display "Deque is empty" ;
    }
    else if(f==r)
    {
        Display "\nThe deleted element is
deque[f];
        f=-1;
        r=-1;
    }
 else if(f==(size-1))
    {
        Display “The deleted element is
deque[f]”;
        f=0;
    }
    else
    {
         Display “The deleted element is
deque[f]”;
        f=f+1;
    }
}
```

Pseudo Code for delete at rear

```
void delete_rear()    {
    if((f==-1) && (r==-1))    {
        printf("Deque is empty");
    }
    else if(f==r)    {
        printf("\nThe deleted element is %d", deque[r]);
        f=-1;
        r=-1;
    }
else if(r==0)
    {
        printf("\nThe deleted element is %d", deque[r]);
        r=size-1;
    }
    else
    {
         printf("\nThe deleted element is %d", deque[r]);
        r=r-1;
    }
}
```