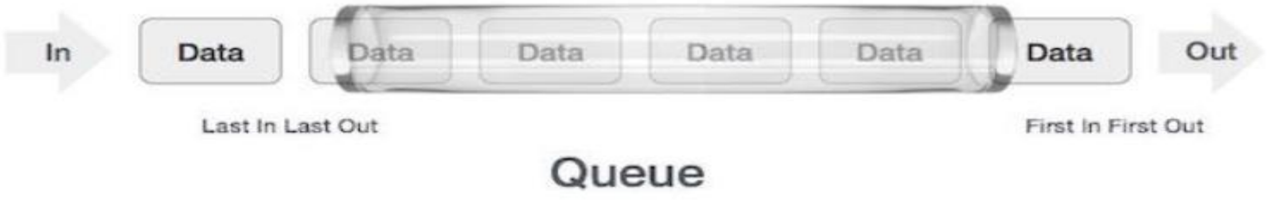
Chapter 5

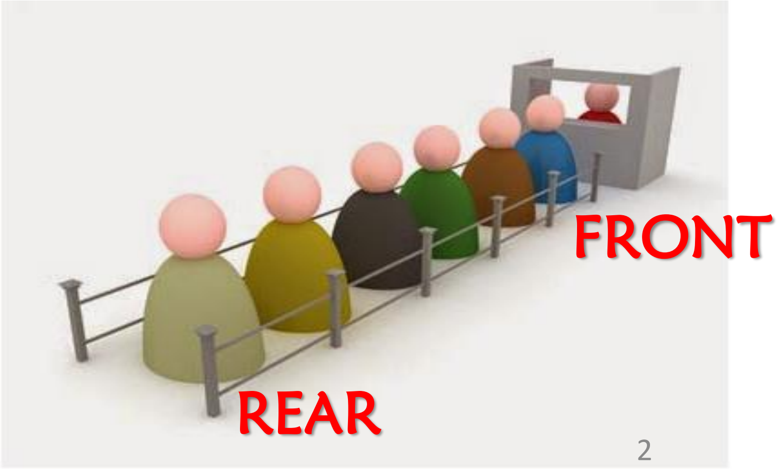
Queue and its Application

Introduction

- Queue is a linear data structure which enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.
- Queue follows the First In First Out (FIFO) rule - i.e., the data item stored first will be accessed first.

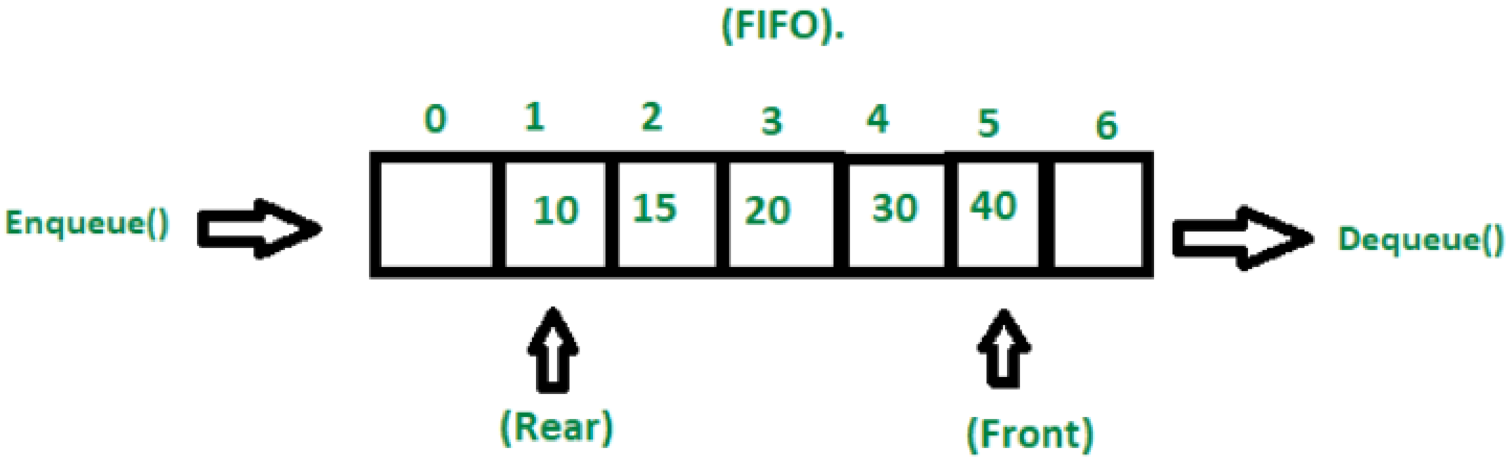


- For example:
    - people waiting in line for a rail ticket form a queue.



Introduction

- FIFO Principle of Queue:
  - A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e., First come first serve).
  - Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue (sometimes, head of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the rear (or the tail) of the queue.



Basic Operations of Queue

- **Enqueue():** Add an element to the end of the queue.
- **Dequeue():** Remove an element from the front of the queue.
- **IsEmpty():** Check if the queue is empty.
- **IsFull():** Check if the queue is full.
- **Peek():** Get the value of the front of the queue without removing it.

Basic Operations of Queue: peek() & isfull()

peek(): Algorithm:

```
begin procedure peek
      return queue[front]
end procedure
```

Implementation:

```
int peek()
{
return queue[front];
}
```

isfull(): Algorithm:

```
begin procedure isfull
      if rear equals to MAXSIZE
            return true
      else
            return false
      endif
end procedure
```

Implementation:

```
bool isfull()
{
      if(rear == MAXSIZE - 1)
            return true;
      else
            return false;
}
```

Basic Operations of Queue: isempty()

Algorithm:

```
begin procedure isempty
        if front is less than MIN OR front is greater than rear
                return true
        else
                return false
        endif
end procedure
```

Implementation:

```
bool isempty()
{
        if(front < 0 || front > rear)
                return true;
        else
                return false;
 }
```

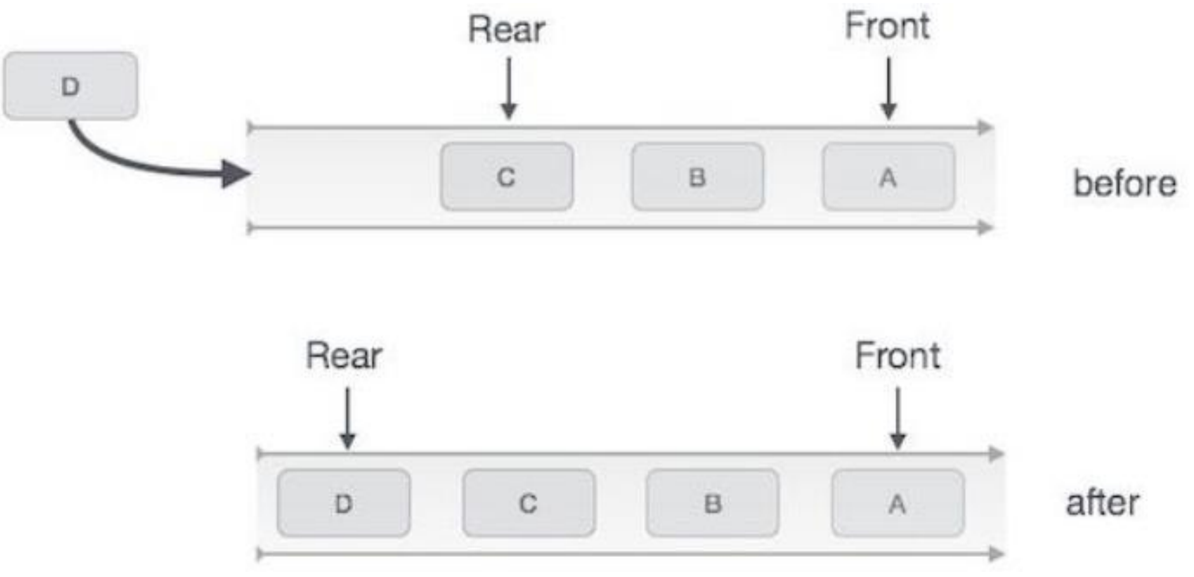
If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Working of Queue

- Queue operations work as follows:
  - Two pointers, FRONT and REAR, are required
  - FRONT track the first element of the queue
  - REAR track the last element of the queue
  - Initially, set value of FRONT and REAR to -1

Basic Operations of Queue: enqueue()

- The following steps should be taken to enqueue (insert) data into a queue:
  1. Check if the queue is full or not
  2. If the queue is full, produce overflow error and exit.
  3. If the queue is not full, increment rear pointer to point the next empty space.
  4. Add data element to the queue location, where the rear is pointing.
  5. return success.



Basic Operations of Queue: enqueue()

Algorithm

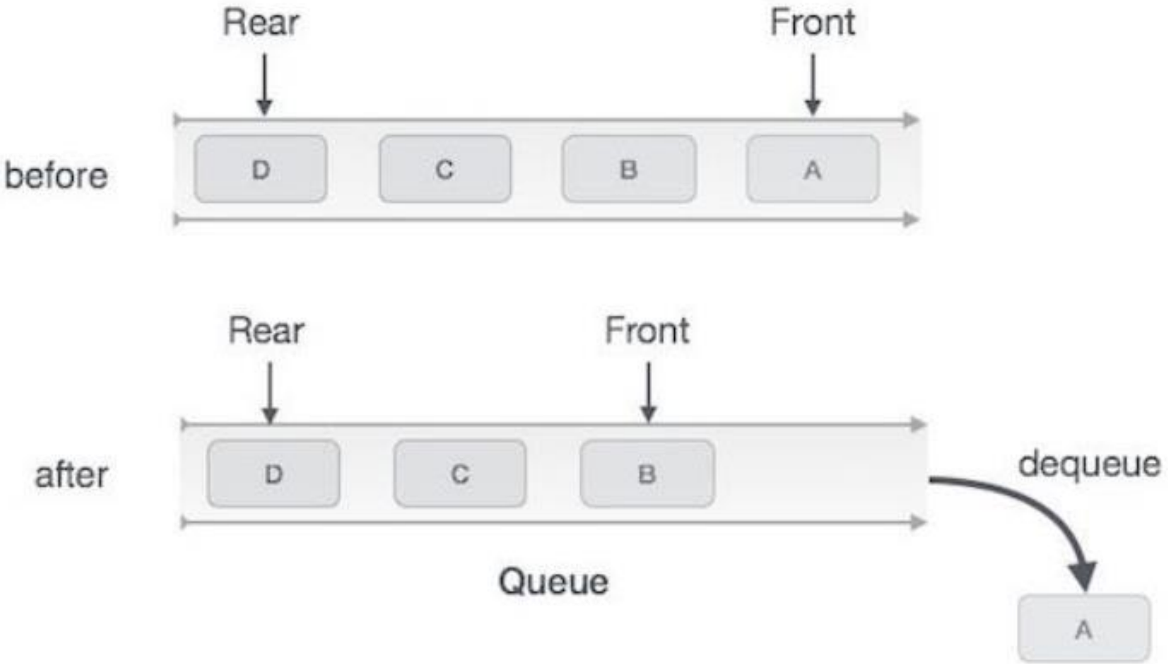
```
begin procedure enqueue(data)
    if queue is full
        return overflow
    endif
    rear ← rear + 1
    queue[rear] ← data
    return true
end procedure
```

Implementation

```
int enqueue(int data)
{
    if(isfull())
        return 0;
    rear = rear + 1;
    queue[rear] = data;
    return 1;
}
```

Basic Operations of Queue: dequeue()

- The following steps are taken to perform dequeue operation:
  1. Check if the queue is empty or not.
  2. If the queue is empty, produce underflow error and exit.
  3. If the queue is not empty, access the data where front is pointing.
  4. Increment front pointer to point to the next available data element.
  5. Return success.



Basic Operations of Queue: dequeue()

Algorithm

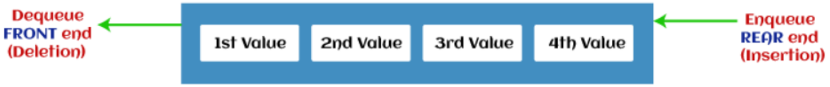
```
begin procedure dequeue
    if queue is empty
        return underflow
    endif
    data = queue[front]
    front ← front + 1
    return true
end procedure
```

Implementation

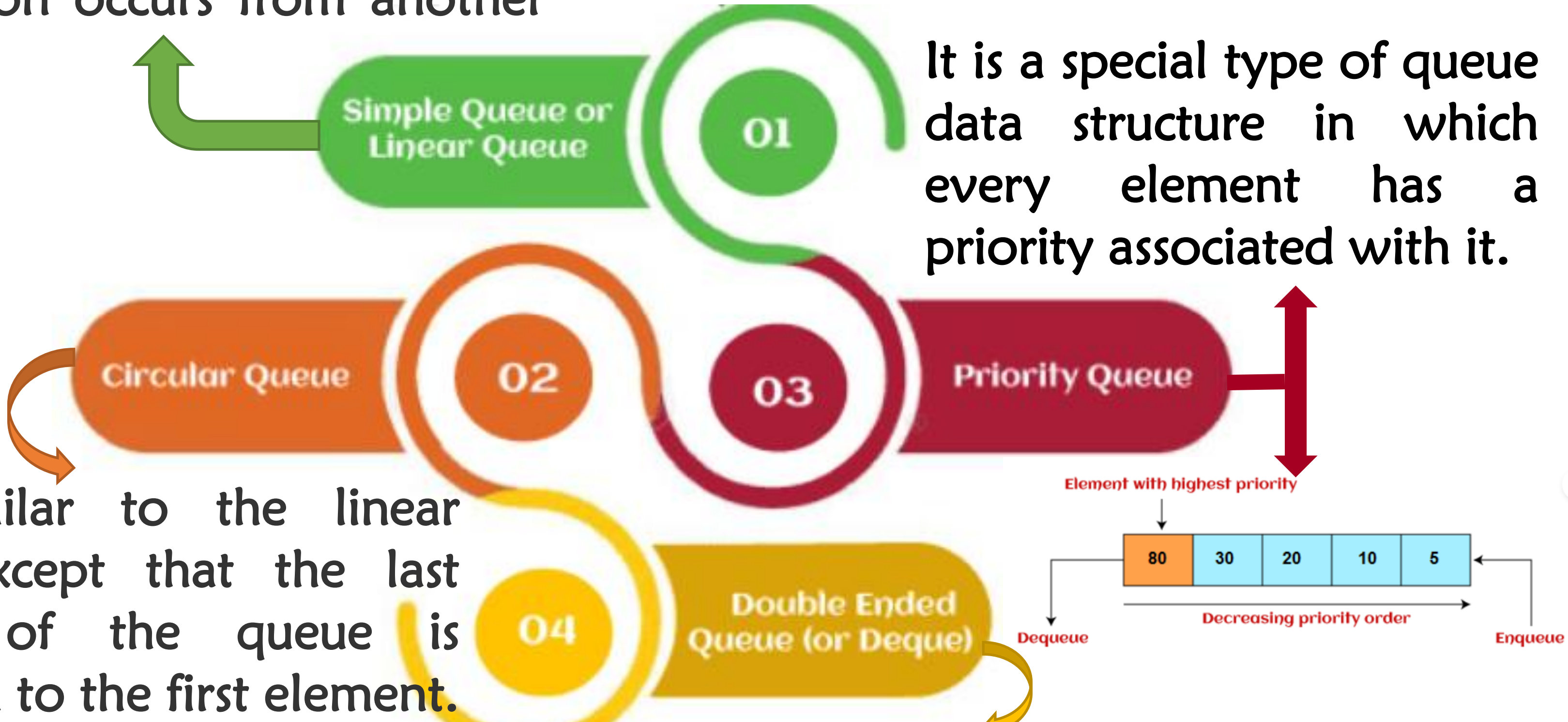
```
int dequeue()
{
    if(isempty())
        return 0;
    int data = queue[front];
    front = front + 1;
    return data;
}
```

Types of Queue

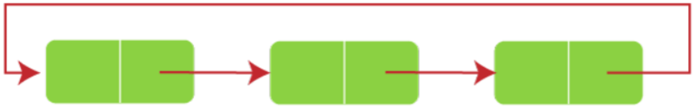
In Linear Queue, an insertion takes place from one end while the deletion occurs from another end.



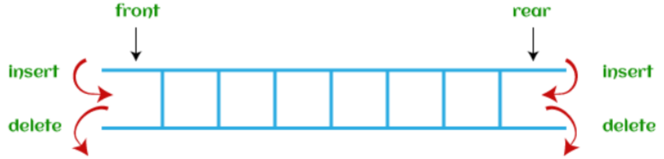
It is a special type of queue data structure in which every element has a priority associated with it.


Simple Queue or Linear Queue
01
Circular Queue
02
03
Priority Queue
04
Double Ended Queue (or Deque)
Element with highest priority
80
30
20
10
5
Decreasing priority order
Dequeue
Enqueue

It is similar to the linear Queue except that the last element of the queue is connected to the first element.



In Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear.



Ways to implement the queue

- There are two ways of implementing the Queue: **Array** and **Linked list**.

Using Array

```
void enqueue(int queue[], int item)
{
    if (isfull())
    {
       cout<<"overflow";
    }
    else
    {
        rear = rear + 1;
        queue[rear]=item;
    }
}
```

```
int  dequeue (int queue[], int item)
{
     if (isempty())
       {
         cout<<"underflow";
       }
    else
    {
         item  = queue[front];
         front = front + 1;
       return item;
    }
}
```

Ways to implement the queue: using Linked list

```
void  enqueue(struct node *ptr, int item) {
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL) {
        cout<<"\nOVERFLOW\n";
        return;
    }
    else {
        ptr -> data = item;
        if(front == NULL) {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else  {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}
```

```
void  dequeue (struct node *ptr)
{
    if(front == NULL)
    {
        cout<<"\nUNDERFLOW\n";
        return;
    }
    else
    {
        ptr = front;
        front = front -> next;
        delete ptr;
    }
}
```

Applications of Queue

- In CPU scheduling and Disk Scheduling.
- In asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
- In operating systems for handling interrupts.
- As buffers in most of the applications like MP3 media player, CD player, etc.
- To maintain the play list in media players in order to add and remove the songs from the play-list.
- Center phone systems use Queues to hold people calling them in order.

Thank You

Question?