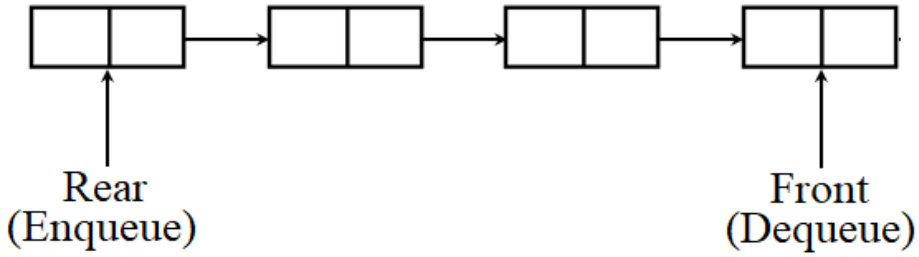
Data Structure & Algorithms

Queues

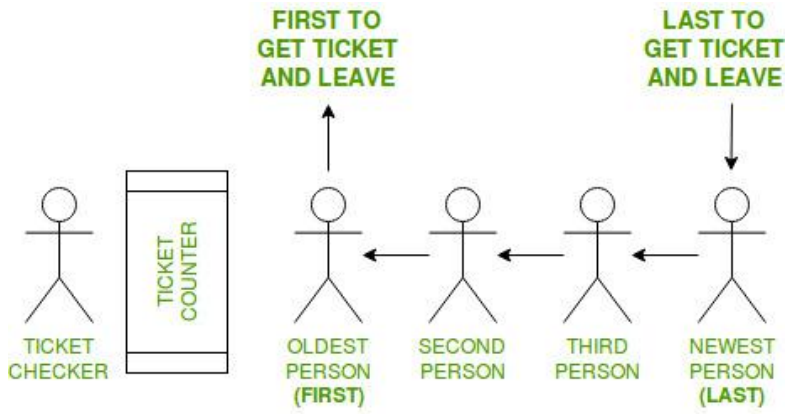
What is Queue?

- A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).
- A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.



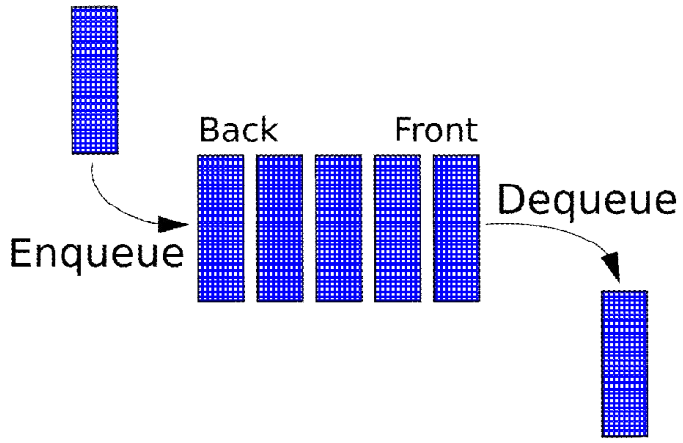
FIFO

- In computing and in systems theory, FIFO (an acronym for first in, first out) is a method for organizing the manipulation of a data structure (often, specifically a data buffer) where the oldest (first) entry, or "head" of the queue, is processed first.
- Take a look at this example:



Operations on queue

- Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues.



Rear and Front

Assumption: queue can only be used for one time, it cannot be refilled.

Empty queue conditions:

Front == 1 , Rear == 0, Rear + 1 == Front

Queue with 1 element conditions:

Front == 1 , Rear == 1

Full queue conditions:

Front == 1 , Rear == maxsize

Enqueue Operation

We call the ADD operation on a queue ENQUEUE. When an element is enqueued, it takes its place at the tail of the queue, just as a newly arriving customer takes a place at the end of the line. Take a look at the steps:

- Step 1 − Check if the queue is full.
- Step 2 − If the queue is full, produce overflow error and exit.
- Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
- Step 4 − Add data element to the queue location, where the rear is pointing.
- Step 5 − return success

Enqueue Operation - Algorithm

```
int items[maxsize];
int front, rear;
int Enqueue (int item){
  if(rear == maxsize){
    print("Queue is full!");
    return -1;   //Error!
  }

  items[++rear] = item;
  return 0;    //Success!
}
```

Dequeue Operation

We call the DELETE operation on a queue DEQUEUE. Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation:

- Step 1 − Check if the queue is empty.
- Step 2 − If the queue is empty, produce underflow error and exit.
- Step 3 − If the queue is not empty, access the data where front is pointing.
- Step 4 − Increment front pointer to point to the next available data element.
- Step 5 − Return success.

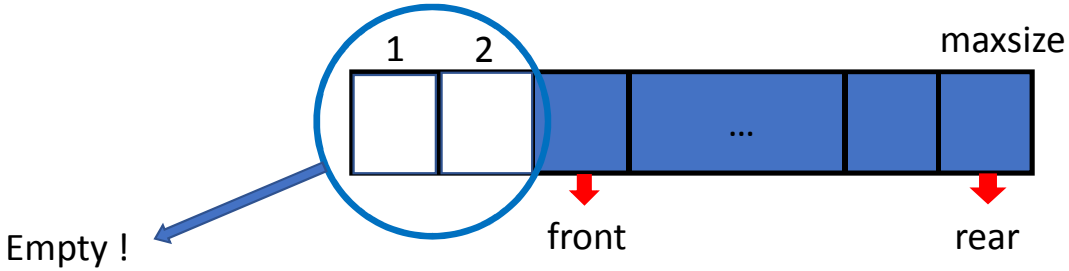
Dequeue Operation - Algorithm

```
int items[maxsize];
int front, rear;
int Dequeue (){
  if(rear+1 == front){
    print("Queue is empty!");
    return -1;    //Error!
  }

  return items[front++]; //Success!
}
```

Where is the problem?

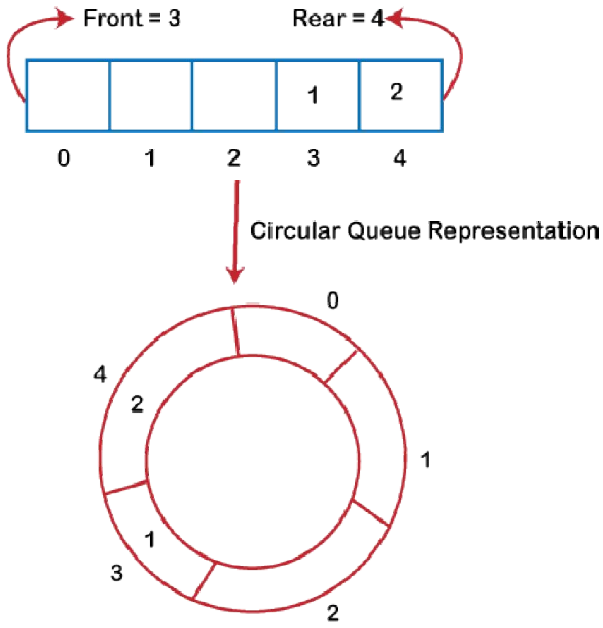
- There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced. Take a look at this situation:



What is the solution? Circular Queue!

Circular Queue

- A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle



Circular Queue

Assumptions: Front = 1, Rear = 0

Empty circular queue condition:

Rear mod maxsize + 1 == Front

Full queue condition:

(Rear + 1) mod maxsize + 1 == Front

Enqueue Operation - Algorithm

```
int items[maxsize];
int front, rear;
int Enqueue (int item){
  if( (rear + 1) mod maxsize + 1 == front){
    print("Queue is full!");
    return -1;   //Error!
  }

  rear = rear mod maxsize + 1
  items[rear] = item;
  return 0;    //Success!
}
```

Dequeue Operation - Algorithm

```
int items[maxsize];
int front, rear;
int Dequeue (){
  int x ;
  if( rear mod maxsize + 1 == front){
    print("Queue is empty!");
    return -1;   //Error!
  }

  x = items[front];
  front = front mod maxsize + 1;
  return x;
}
```