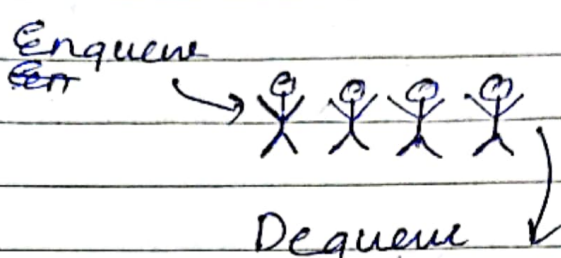
Queue

Queue is a FIFO data structure



For queues we need to pointers, one pointing at the insertion end & other at deletion end

Methods:

1) Enqueue (): to insert an element.
2) Dequeue (): to remove an element.
3) firstval(): to return the value at first pos^n.
4) lastval(): to return the value at last pos^n
5) peek(position): to return the value at some specific position.
6) isEmpty()/isFull(): to check whether queue is empty or not.

Implementation: 1) Arrays
2) Stacks 3) Linked Lists

Queue Front: Examines the element at the front of queue

Queue Rear: Examines element at the rear of the queue.

If there is no data in the queue, then the queue is in underflow state.

Approach 1

Enqueue: TC → O(1)

→ Check if queue is full
→ Increment rear by 1
→ Insert the element

Other style
front = rear = -1

Dequeue:

→ Check if queue is empty
→ Remove element at 0th index
→ Shift all other elements to their immediate left
→ Decrement rear by 1
→ Here our first element is at index 0 & rearmost element at index rear

Our Initializations: (My Initialization)
front = 0 rear = -1

Condition for Queue full = rear == n - 1

Queue empty = rear < front

Approach

Enqueue → same as 1st approach

Dequeue → TC(O(1))

→ Increment front by 1

→ Always check if queue is empty

Our 1st element is at index front+1 & rearmost at backend

Harry style:

```
#include <stdio.h>
#  "  <stdlib.h>

struct queue {
    int size;
    int f;
    int r;
    int *arr;
};

int isEmpty(struct queue *q){
    if(q->f == q->r){
        return 1;
    }
    return 0;
}
```

→ q->f == -1

```
int isFull(struct queue *q){
    if(q->r == q->size-1){
        return 1;
    }
    return 0;
}

void enqueue(struct queue *q, int val){
    if(isFull(q)){
        pf("The queue is full\n");
    }
    else{
        if(q->f == -1)
            q->f = 0;
        q->r++;
        q->arr[q->r] = val;
        pf("Enqueued element: %d\n", val);
    }
}

int dequeue(struct queue *q){
    int a = -1;
    if(isEmpty(q)){
        pf("The queue is empty\n");
    }
    else{
        a = q->arr[q->f];
        q->f++;
        if(q->f > q->r)
            q->f = q->r = -1;
    }
    return a;
}
```

```
int main(){
    struct queue q;
    q.size = 4;
    q.f = q.r = -1;

    q.arr = (int*)malloc(q.size*sizeof(int));

    .
    .
    .

}
```

Tests:
```
if (isEmpty(&q)){
    pf("Queue is empty\n");
}
```

o/p: Queue is empty

Tests:
```
enqueue(&q, 12);
--"--(--"--, 15);
```

o/p:
```
Enqued element : 12
--"--  --"--  : 15
```

Tests:
```
pf("Dequeing element %d\n", dequeue(&q));
```

o/p:
```
Dequeing element 12
--"--  --"--  15
```

My method:

```
class queue{
    int size;
    int rear, front;
    int [] q;

    queue(int n){
      front=0;
      rear=-1;
      this.size=n;
      q= new int[n];
    }

boolean isEmpty(){
    return front>rear;
}

boolean isFull(){
    return rear=size-1;
}

void enqueue(int data){
    if(isFull()){
      sout("Queue is full");
    }
    else{
    queue[++rear]=data;
    }
}
```

```
void dequeue() {
    if(isEmpty()) {
        sout("Queue is empty");
    }
    else {
        front++;
        sout("Dequeued:" + q[front]);
        front++;
    }
}
```

Circular Queue

Initialization:	Full: (rear+1)%size=front
front=0	Empty: front==rear
rear=0	

Functions:

```
boolean isEmpty() {
    return front==rear;
}

boolean isFull() {
    return (rear+1)%size==front;
}
```

```
void Dequeue (){

    if (q.Empty ()) {
    cout ("Underflow");
                }

    else{
    cout ("Dequeued:" + arr[(front +1) % size];

    front = (front +1) % size;

        }

}

void Enqueue (int element){

    if(q.FULL()) {
    cout ("Overflow");
            }

    else {
    rear = (rear + 1) % size;
    arr[rear] = element;
    cout ("Enqueued " + element);

        }
}
```