QUEUES

→ a kind of list where insertion is done on one side called the rear & deletion is done at the Front

REAR ⟶ INSERTION
FRONT ⟶ DELETION

→ think of the line at the cashier or any other service where its "first come, first serve"

→ First in First out (FIFO)

→ insertLast() & deleteFirst()

QUEUE IMPLEMENTATIONS
① POINTER IMPLEMENTATION
② ARRAY IMPLEMENTATION
③ CURSOR BASED

QUEUE OPERATIONS

1) ENQUEUE (x, Q) — inserts element at the rear of Q (insertLast())

2) DEQUEUE (Q) — deletes the element at the Front of Q (deleteFirst())

3) FRONT (Q) — returns the element at the front of Q

4) EMPTY (S) — returns non-zero value if empty

5) INITIALIZE (S) — initialize the queue to be NULL to be used for the first time.

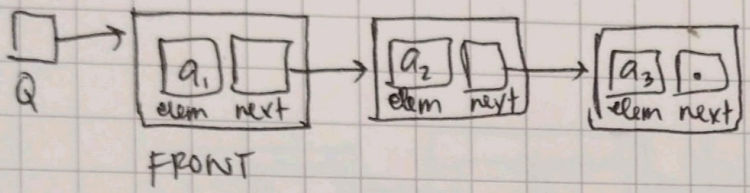
QUEUE - POINTER IMPLEMENTATION

VERSION 1 - QUEUE is a pointer to a self referencing structure

Two views:

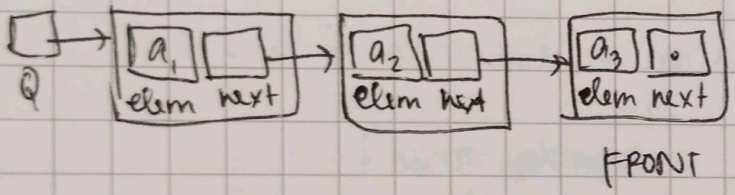
ENQUEUE = insertLast() DEQUEUE = deleteFirst()

① The front element is stored at a cell pointed to be Q



$O(n)$ $O(1)$

② Front element is stored in a cell whose field is NULL



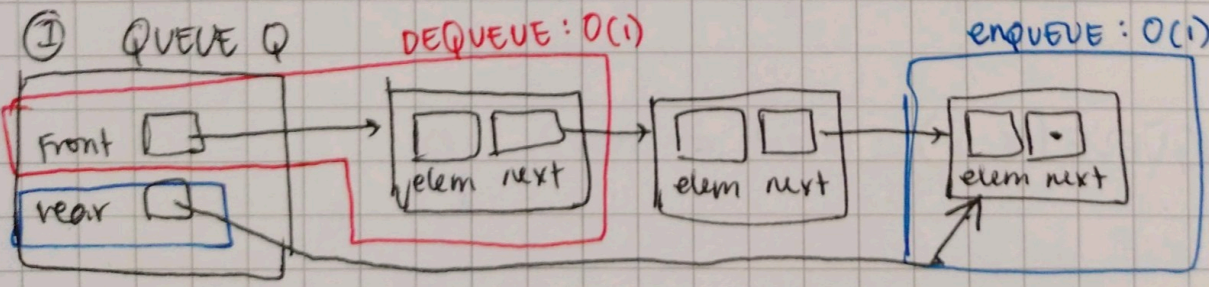
$O(1)$ $O(n)$

Both versions have the same running time.

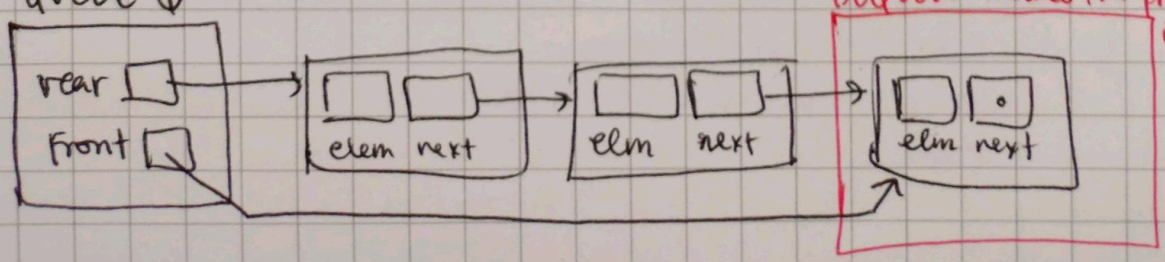
VERSION 2 - QUEUE is a structure containing both front & rear pointers.

Two views:

① QUEUE Q



② QUEUE Q



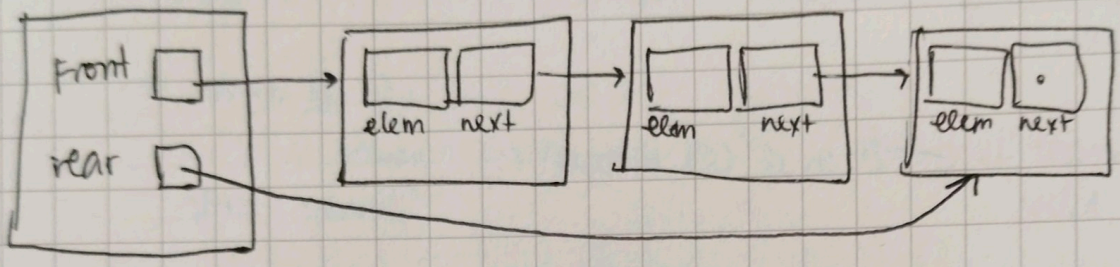
view 1: only need to insertlast () at rear and deleteFirst() at front. ✓

view 2: ~~The node pointed to be front will be deleted, needs to traverse in order to reach front, which will be deleted~~ - making the previous node inaccessible

X

PRACTICE EXERCISE

QUEUE Q



Q. Write the appropriate declaration & definition of QUEUE Q.

ANS.

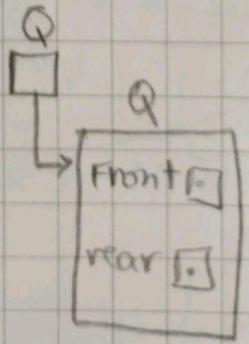
```
typedef struct node {
    char elem;
    struct node * next;
} node;

typedef struct {
    node * Front, * rear;
} QUEUE;

QUEUE Q;
```

Q. WRITE the code of the queue operations.

ANS.

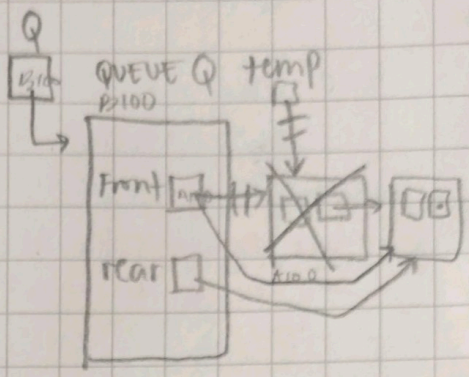
A] initQueue ()

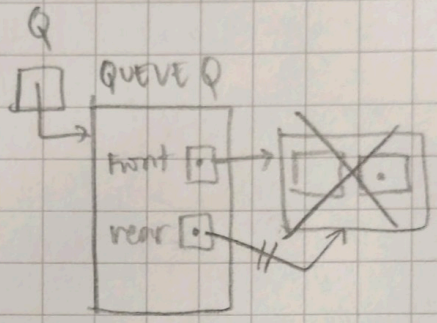
```
void initQueue (QUEUE *Q) {
    Q → Front = NULL;
    Q → rear = NULL;
} // code so that the queue
    can be used for the first
    time
```



C] DEQUEUE(Q) ≈ delete from the front

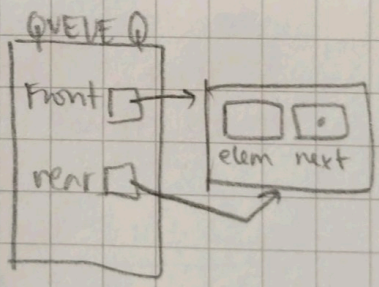
```
void dequeue (QUEUE *Q){
    node temp;
    if(Q→front != NULL){
        temp = Q→front;
        Q→front = temp→next;
        free(temp);
    }
    if(Q→front == NULL){
        Q→rear = NULL;
    }
}
```





D] FRONT (Q) return the element at the front of queue

```
char front (QUEUE Q){
    return (Q.front != NULL)?
            (Q.front)→elem : '\0';
}
```

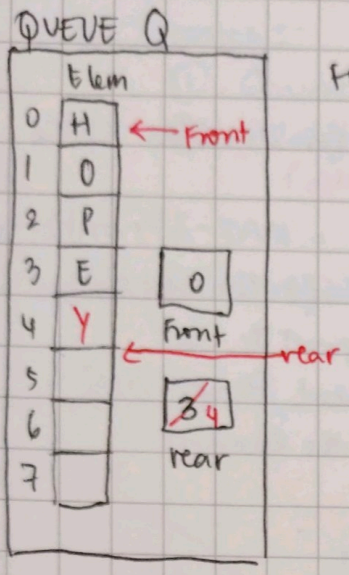


E] EMPTY(Q) return nonzero value if empty

```
int isEmpty (QUEUE Q){
    return (Q.front == NULL && Q.rear == NULL)? 1 : 0;
```

QUEUE - ARRAY IMPLEMENTATION

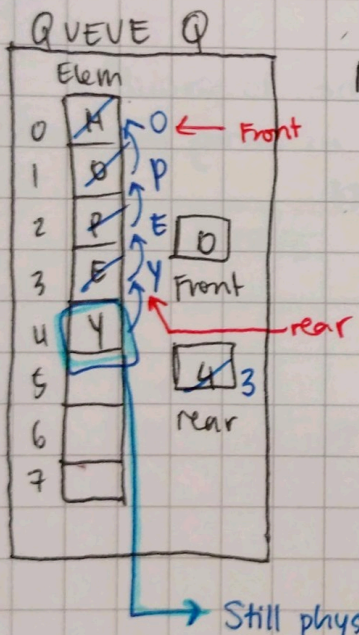
VARIATION 1



For instance,

Enqueue 'y':

① check for available space
② increment rear
③ insert the element



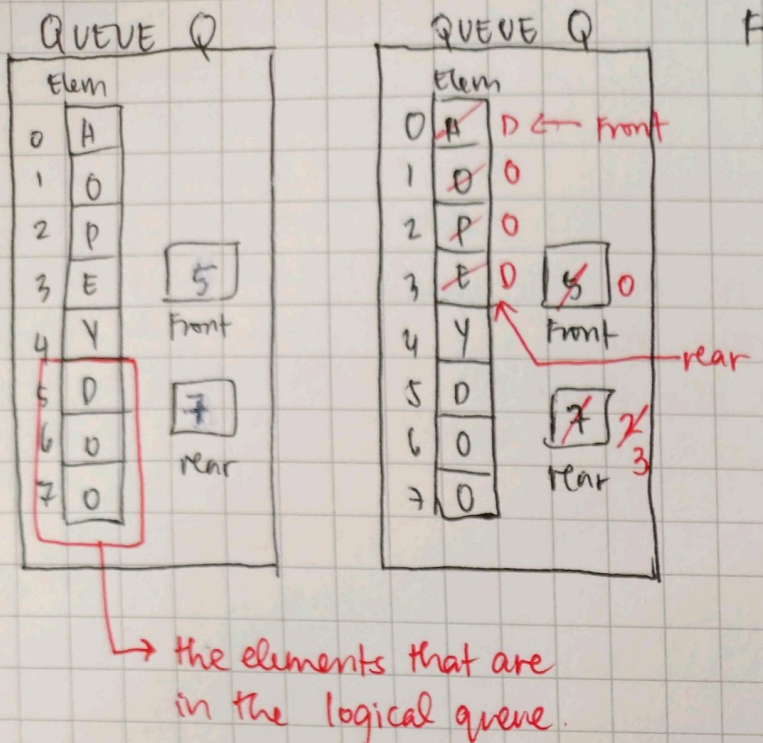
Now,

Dequeue (Q):

① check if the queue is not empty
② shift elements
③ decrement rear

Still physically at index 4 but it is no longer part of the logical queue

VARIATION 2



For instance,

Enqueue 'D':

① check for available space (no space)

② if rear = MAX-1 && front != 0 then shift the elements

③ Adjust front & rear

④ increment rear

⑤ insert the element

CIRCULAR ARRAY IMPLEMENTATION

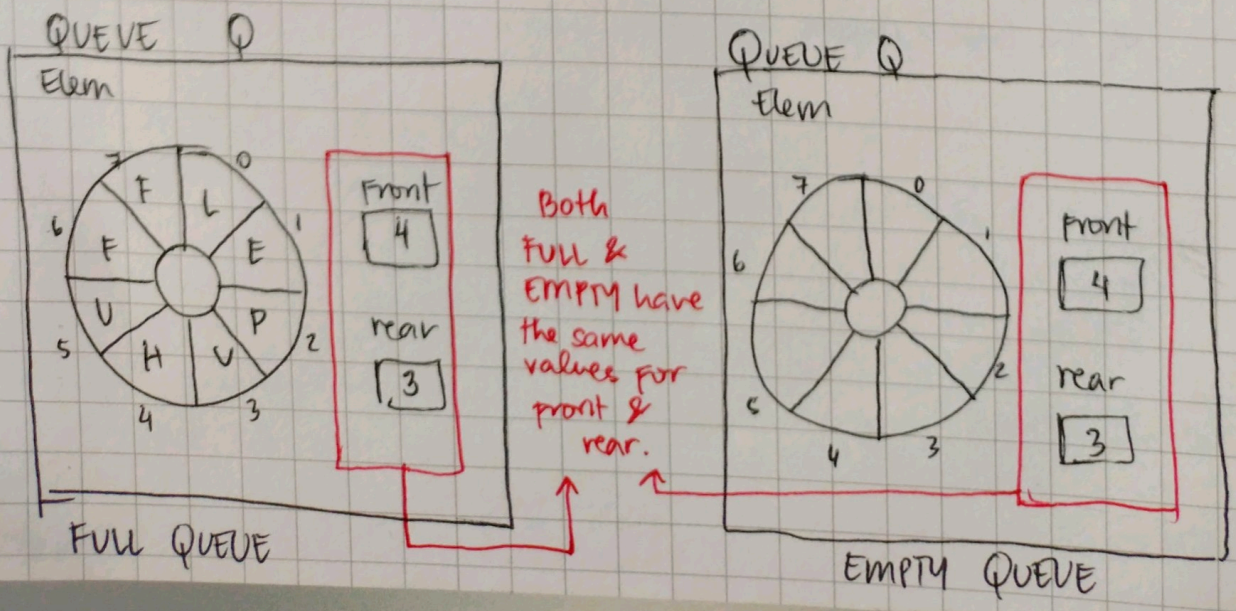
- a solution to avoid shifting in array implementation of queues.

★ declared like any other array in C

★ "circular orientation"
↳ referring to the way we manipulate the array

★ "circular" → the array has no definite beginning nor end.
The first element can be stored anywhere.

★ COUNTER CLOCKWISE or CLOCKWISE direction of insertion are both possible.

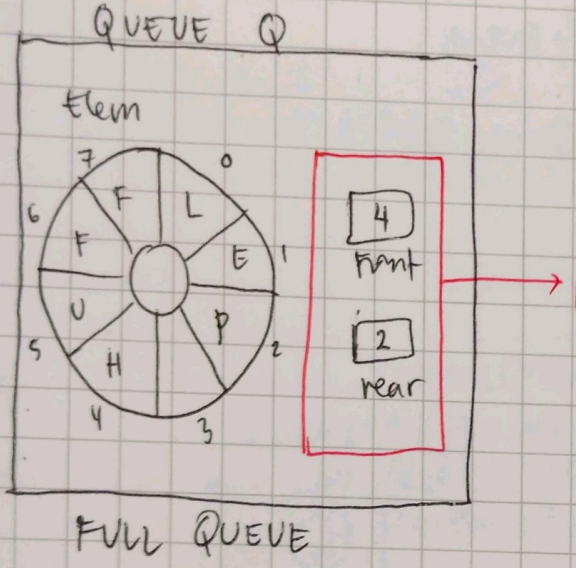


Front & rear cant be the same in both situations.
How can we differentiate between FULL & EMPTY?

2 SOLUTIONS:

① Counter variable
② sacrifice 1 cell (FULL = MAX-1 elements instead of FULL = MAX elements)

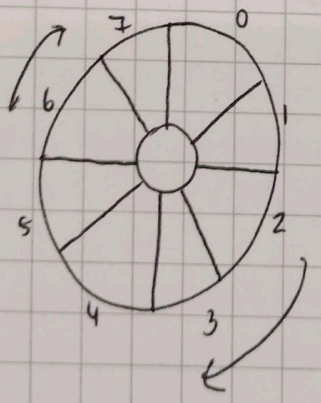
So, a full queue will look like this,



EMPTY

REAR	FRONT
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	0

MAX = 8



FULL

REAR	FRONT
0	2
1	3
2	4
3	5
4	6
5	7
6	0
7	1

(rear+1) % MAX == Front

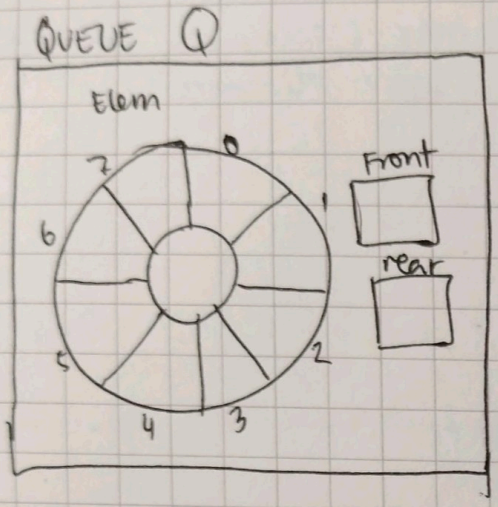
TO MOVE:
Front = (front+1) % MAX
rear = (rear+1) % MAX

(rear+2) % MAX == Front

PRACTICE PROBLEM

Q. 1) Write the appropriate declaration of the circular array
2) Write the code of the queue operations.

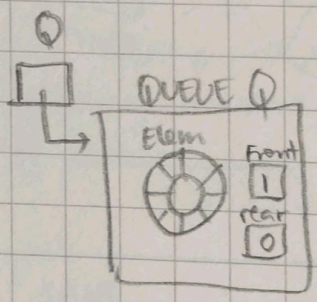
ANS.




```
#define MAX 8

typedef struct {
    char Elem[MAX];
    int front, rear;
} QUEUE;

QUEUE Q;
```

A] initQueue()

```
void initQueue (QUEUE *Q) {
    Q → front = 1;
    Q → rear = 0;
}
```



* going CLOCKWISE
* EMPTY QUEUE = Front is ahead of rear by 1.

isEmpty & isFull() are both needed to check the state of the queue

B] isEmpty()

```
int isEmpty (QUEUE Q) {
    return ((Q.rear+1) % MAX
        == Q.front) ?
        1 : 0;
}
```

EMPTY
↳ front is ahead of rear by one cell

isFull()

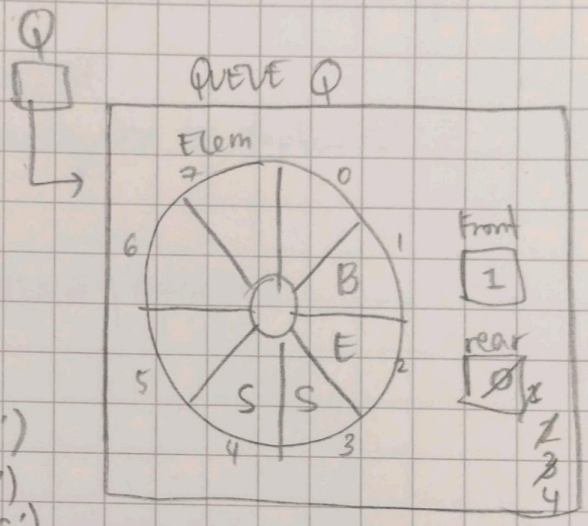
```
int isFull (QUEUE Q) {
    return ((Q.rear+2) % MAX
        == Q.front) ? 1 : 0;
}
```

FULL
↳ Front is ahead of rear by two cells

C] ENQUEUE (Q, x)

```
void enqueue (QUEUE *Q, char elem)
    if (isFull (*Q) != 1) {
        Q→rear = (Q→rear + 1) %. MAX;
        Q→Elem [Q→rear] = elem;
}   }
```

SUBTASKS
1. check if Queue is full
2. increment rear
3. insert the element




```
enqueue(&Q, 'B')
enqueue(&Q, 'E')
enqueue(&Q, 'S')
enqueue(&Q, 'S')
```

D] DEQUEUE (Q)

```
void dequeue (QUEUE *Q) {
    if (isEmpty (*Q) != 1) {
        Q→Front = (Q→Front + 1) %. MAX;  // incrementing front removes the 1st index of the queue from the logical list.
}   }
```

E] FRONT (Q)

```
void Front (QUEUE Q) {
    return (isEmpty (Q) != 1) ? Q.Elem[Q.Front] : '\0';
```

F] display Queue (Q)

```
void displayQueue (QUEUE Q) {
    int ndx;
    if (isEmpty(Q) != 1) {
        for (ndx = Q.front; ndx != (Q.rear + 1) % MAX;
                            ndx = (ndx + 1) % MAX) {
            printf("%c ", Q.Elem[ndx]);
        }
    }
```