# Array Queue

Insertion is done on one side, deletion is done on the other (FRONT & REAR).

- FIRST IN FIRST OUT
- LAST IN LAST OUT

Can be either **CLOCKWISE** or **COUNTERCLOCKWISE**.

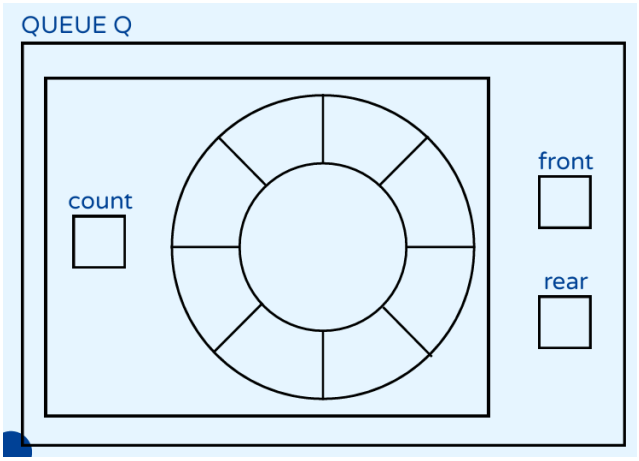
- Clockwise - front/rear starts from **first position** and moves to the **right**.
- Counterclockwise - front/rear starts from **last position** and moves to the **left**.

**Important:**

You are not allowed to directly access the content of the Queue from outside the operations. You must use the operations for all actions such as traversal and accessing.

**Variation 1**

Queue is a static array with **count**, **front**, and **rear**.




```
#define MAX 10
typedef struct {
   int items[MAX];
   int count;
} List;
typedef struct {
   List list;
   int front;
   int rear;
} Queue;
Queue Q;
```

| Operations | Checklist | Example |
|---|---|---|
| `Queue* initialize();` | ☐ Allocate memory for a Queue structure<br>☐ Initialize the queue's list count to 0<br>☐ Initialize the front and rear pointers to -1 to indicate an empty queue<br>☐ Return the pointer to the queue | `Queue* Q = initialize();` |

| | | |
|---|---|---|
| `bool isFull(Queue* q);` | ☐ count == MAX | |
| `bool isEmpty(Queue* q);` | ☐ count == 0 | |
| `void enqueue(Queue* q, int value);` | ☐ Check if the queue is full<br>☐ If the queue is empty, set both front and rear to 0<br>☐ Otherwise, update the rear pointer circularly (rear = (rear + 1) % MAX)<br>☐ Add the new value to the list at the rear position<br>☐ Increment the count | `Before:`<br>`  items: [ 1, 3, 2, 5, ... ]`<br>`  count: 4`<br>`  front: 0`<br>`  rear: 3`<br><br>`enqueue(Q, 4);`<br><br>`After:`<br>`  items: [ 1, 3, 2, 5, 4, ... ]`<br>`  count: 5`<br>`  front: 0`<br>`  rear: 4` |
| `int dequeue(Queue* q);` | ☐ Check if the queue is empty<br>☐ Store the value found at the front of the queue before it is removed<br>☐ If this is the last element in the queue, reset the queue to its initial empty state<br>☐ If not, update the front pointer circularly (front = (front + 1) % MAX)<br>☐ Decrement the count<br>☐ Return the removed value | `Before:`<br>`  items: [ 1, 3, 2, 5, ... ]`<br>`  count: 4`<br>`  front: 0`<br>`  rear: 3`<br><br>`int value = dequeue(Q);`<br><br>`After:`<br>`  items: [ 1, 3, 2, 5, ... ]`<br>`  count: 3`<br>`  front: 1`<br>`  rear: 3` |
| `int front(Queue* q);` | ☐ Check if the queue is empty<br>☐ Return the value at the front of the queue | |
| `void display(Queue* q);` | ☐ Check if the queue is empty<br>☐ Loop through the queue from front to rear and print each element | |

**Note:**

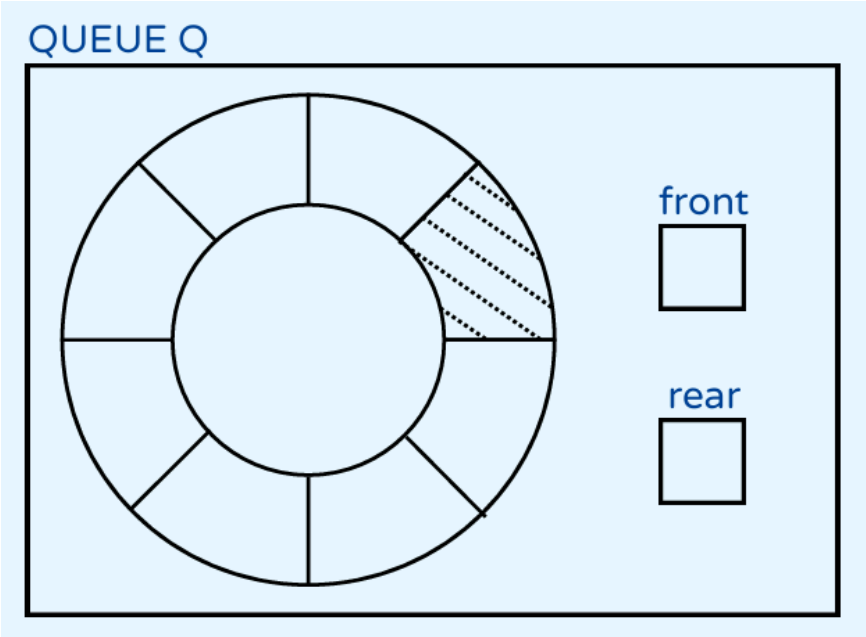
For most operations, it is also common to return a **boolean value** representing whether the operation is successful or not.

---

**Variation 2**

Queue is a static array with **front**, **rear**, and a **sacrificial space**.

Sacrificial space will always be the space before front. [ (front - 1 + MAX) % MAX ]




```
#define MAX 10
typedef struct {
   int items[MAX];
   int front;
   int rear;
} Queue;
Queue Q;
```

| Operations | Checklist | Example |
| --- | --- | --- |
| `Queue* initialize();` | ☐ Allocate memory for the Queue structure<br>☐ Initialize front to 1 and rear to 0 | `Queue* Q = initialize();` |
| `bool isEmpty(Queue* q);` | ☐ front == (rear + 1) % MAX | |
| `bool isFull(Queue* q);` | ☐ front == (rear + 2) % MAX | |
| `void enqueue(Queue* q, int value);` | ☐ Check if the queue is full<br>☐ Increment the rear pointer circularly (rear = (rear + 1) % MAX)<br>☐ Insert the new element at the new rear position | |
| `int dequeue(Queue* q);` | ☐ Check if the queue is empty<br>☐ Get the element at the front of the queue<br>☐ Increment the front pointer circularly (front = (front + 1) % MAX)<br>☐ Return the dequeued element | |
| `int front(Queue* q);` | ☐ Check if the queue is empty<br>☐ Return the element at the current front position | |
| `void display(Queue* q);` | ☐ Check if the queue is empty<br>☐ Loop through the queue from front to rear and print each element | |