LeetCode 622 — Design Circular Queue

1. Problem Title & Link

- **Title:** LeetCode 622 — Design Circular Queue
- **Link:** https://leetcode.com/problems/design-circular-queue/

2. Problem Statement (Short Summary)

Implement a **circular queue** with fixed capacity k.
Support the operations:

- enQueue(value) → insert element at rear
- deQueue() → delete front element
- Front() → get front
- Rear() → get rear
- isEmpty() → check if empty
- isFull() → check if full

Queue must behave **circularly**, i.e., when reaching the end, it wraps around to the beginning.

3. Examples (Input → Output)

```
MyCircularQueue q = new MyCircularQueue(3);
q.enQueue(1); // true
q.enQueue(2); // true
q.enQueue(3); // true
q.enQueue(4); // false (queue full)
q.Rear();     // returns 3
q.isFull();   // true
q.deQueue();  // true
q.enQueue(4); // true
q.Rear();     // returns 4
```

4. Constraints

- 1 <= k <= 1000
- All operations must be **O(1)** time.
- Queue must NOT grow beyond capacity.

5. Core Concept (Pattern / Topic)

Circular Buffer Using Array

Use:

- A fixed-size array arr
- Two pointers:
  - front → index of current front
  - rear → index where next element will be inserted
- A variable count or check full using pointers

Use **modular arithmetic**:

rear = (rear + 1) % size

front = (front + 1) % size

6. Thought Process (Step-by-Step Explanation)

We maintain:

- size → capacity of queue
- count → current number of items
- front pointer → index of first element
- rear pointer → index of next insertion position

Algorithm:

enQueue(value):

- If queue is full → return false
- Insert at arr[rear]
- Move rear circularly: rear = (rear + 1) % size
- Increase count

deQueue():

- If queue is empty → return false
- Move front circularly: front = (front + 1) % size
- Decrease count
- (We don't overwrite values, count manages visibility)

Front():

- If empty return -1
- Else return arr[front]

Rear():

- If empty return -1
- Rear element is at index:
  (rear - 1 + size) % size
-

isEmpty():

- count == 0

isFull():

- count == size

7. Visual / Intuition Diagram

Let size = 3:

Initially:

front = 0

rear = 0

count = 0

arr = [_, _, _]

After enQueue 1:

arr = [1, _, _]

front = 0

rear = 1

count = 1

After enQueue 2:

arr = [1, 2, _]

front = 0

rear = 2

count = 2

After enQueue 3:

arr = [1, 2, 3]

front = 0

rear = 0 (wrapped around)

count = 3 (FULL)

After deQueue:

front = 1

count = 2

Now enQueue(4):

arr = [4, 2, 3]

rear = 1

8. Pseudocode

```
initialize arr of size k
front = 0
rear = 0
count = 0

function enQueue(value):
```

```
    if count == k: return false
    arr[rear] = value
    rear = (rear + 1) % k
    count += 1
    return true

function deQueue():
    if count == 0: return false
    front = (front + 1) % k
    count -= 1
    return true

function Front():
    if count == 0: return -1
    return arr[front]

function Rear():
    if count == 0: return -1
    return arr[(rear - 1 + k) % k]

function isEmpty():
    return count == 0

function isFull():
    return count == k
```

9. Code Implementation

✅ Python

```
class MyCircularQueue:

    def __init__(self, k: int):
        self.size = k
        self.arr = [0] * k
        self.front = 0
        self.rear = 0
        self.count = 0

    def enQueue(self, value: int) -> bool:
        if self.count == self.size:
            return False
        self.arr[self.rear] = value
        self.rear = (self.rear + 1) % self.size
```

```
        self.count += 1
        return True

    def deQueue(self) -> bool:
        if self.count == 0:
            return False
        self.front = (self.front + 1) % self.size
        self.count -= 1
        return True

    def Front(self) -> int:
        if self.count == 0:
            return -1
        return self.arr[self.front]

    def Rear(self) -> int:
        if self.count == 0:
            return -1
        return self.arr[(self.rear - 1 + self.size) % self.size]

    def isEmpty(self) -> bool:
        return self.count == 0

    def isFull(self) -> bool:
        return self.count == self.size
```

✅ **Java**

```
class MyCircularQueue {
    private int[] arr;
    private int front;
    private int rear;
    private int count;
    private int size;

    public MyCircularQueue(int k) {
        this.size = k;
        arr = new int[k];
        front = 0;
        rear = 0;
        count = 0;
    }
```

```
    public boolean enQueue(int value) {
        if (isFull()) return false;
        arr[rear] = value;
        rear = (rear + 1) % size;
        count++;
        return true;
    }

    public boolean deQueue() {
        if (isEmpty()) return false;
        front = (front + 1) % size;
        count--;
        return true;
    }

    public int Front() {
        if (isEmpty()) return -1;
        return arr[front];
    }

    public int Rear() {
        if (isEmpty()) return -1;
        return arr[(rear - 1 + size) % size];
    }

    public boolean isEmpty() {
        return count == 0;
    }

    public boolean isFull() {
        return count == size;
    }
}
```

10. Time & Space Complexity

Operation	Time
enQueue	O(1)
deQueue	O(1)
Front	O(1)
Rear	O(1)
isEmpty	O(1)
isFull	O(1)

Space: O(k)

11. Common Mistakes / Edge Cases

❌ **Mistakes:**

- Miscomputing rear index using (rear + 1) without modulo
- Forgetting to check both empty/full conditions
- Confusing rear index value vs last element index
- Overwriting elements without adjusting front pointer

Edge cases:

- size = 1
- enQueue → full → deQueue → enQueue again (wrap-around)
- consecutive deQueue until empty

12. Detailed Dry Run (Step-by-Step)

Let queue size = 3

Operations:

enQueue(1)

enQueue(2)

enQueue(3)

enQueue(4) -> false

Rear() => 3

isFull() => true

deQueue()

enQueue(4)

Rear() => 4

Step-by-step:

Step	arr	front	rear	count	Output
enQueue(1)	[1,,]	0	1	1	TRUE
enQueue(2)	[1,2,_]	0	2	2	TRUE
enQueue(3)	[1,2,3]	0	0	3	TRUE
enQueue(4)	full	—	—	3	FALSE
Rear()	—	—	—	—	3
isFull()	—	—	—	—	TRUE
deQueue()	[1,2,3]	1	0	2	TRUE
enQueue(4)	[4,2,3]	1	1	3	TRUE
Rear()	—	—	—	—	4

Perfect behavior.

13. Common Use Cases

- Circular buffers
- Real-time data streams
- Task schedulers
- Queue in embedded systems
- Network packet processing
- Fixed memory queues (OS, networking)

14. Common Traps

- Not handling wrap-around correctly
- Returning wrong index for Rear
- Misusing modulo arithmetic
- Forgetting to track count → causes full/empty ambiguity

15. Builds To (Related Problems)

- **LC 641** — Design circular deque
- **LC 933** — Recent counter (queue-based)
- **LC 346** — Moving average (queue window)
- Producer–consumer problems

16. Alternate Approaches + Comparison

Approach	Time	Space	Notes
Circular array	O(1)	O(k)	✔ Best
Linked list	O(1)	O(k)	Harder to manage wrap-around
Dynamic array	Not allowed	—	Changes capacity

17. Why This Solution Works (Short Intuition)

Circular queues reuse space efficiently by wrapping around the ends using modular arithmetic. Maintaining front, rear, and count ensures constant-time operations with predictable behavior for fixed-size queues.

18. Variations / Follow-Up Questions

- Implement **circular deque** (front + rear insertions)
- Make it thread-safe (locks or semaphores)
- What if you cannot use modulo? (Use if/else branching)
- Implement resizing circular queue (dynamic buffer)