Problem 622: Design Circular Queue

Problem Information

Difficulty: Medium
Acceptance Rate: 53.35%
Paid Only: No
Tags: Array, Linked List, Design, Queue

Problem Description

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle, and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Implement the `MyCircularQueue` class:

* `MyCircularQueue(k)` Initializes the object with the size of the queue to be `k`. * `int Front()` Gets the front item from the queue. If the queue is empty, return `-1`. * `int Rear()` Gets the last item from the queue. If the queue is empty, return `-1`. * `boolean enQueue(int value)` Inserts an element into the circular queue. Return `true` if the operation is successful. * `boolean deQueue()` Deletes an element from the circular queue. Return `true` if the operation is successful. * `boolean isEmpty()` Checks whether the circular queue is empty or not. * `boolean isFull()` Checks whether the circular queue is full or not.

You must solve the problem without using the built-in queue data structure in your programming language.

Example 1:

Input ["MyCircularQueue", "enQueue", "enQueue", "enQueue", "enQueue", "Rear", "isFull", "deQueue", "enQueue", "Rear"] [[3], [1], [2], [3], [4], [], [], [], [4], []] **Output** [null,

true, true, true, false, 3, true, true, true, 4] **Explanation** MyCircularQueue myCircularQueue = new MyCircularQueue(3); myCircularQueue.enQueue(1); // return True myCircularQueue.enQueue(2); // return True myCircularQueue.enQueue(3); // return True myCircularQueue.enQueue(4); // return False myCircularQueue.Rear(); // return 3 myCircularQueue.isFull(); // return True myCircularQueue.deQueue(); // return True myCircularQueue.enQueue(4); // return True myCircularQueue.Rear(); // return 4

Constraints:

* `1 <= k <= 1000` * `0 <= value <= 1000` * At most `3000` calls will be made to `enQueue`, `deQueue`, `Front`, `Rear`, `isEmpty`, and `isFull`.

Code Snippets

C++:

```
class MyCircularQueue {
public:
MyCircularQueue(int k) {

}

bool enQueue(int value) {

}

bool deQueue() {

}

int Front() {

}

int Rear() {

}

bool isEmpty() {

}
```

```
bool isFull() {

}
};

/**
* Your MyCircularQueue object will be instantiated and called as such:
* MyCircularQueue* obj = new MyCircularQueue(k);
* bool param_1 = obj->enQueue(value);
* bool param_2 = obj->deQueue();
* int param_3 = obj->Front();
* int param_4 = obj->Rear();
* bool param_5 = obj->isEmpty();
* bool param_6 = obj->isFull();
*/
```

Java:

```
class MyCircularQueue {

public MyCircularQueue(int k) {

}

public boolean enQueue(int value) {

}

public boolean deQueue() {

}

public int Front() {

}

public int Rear() {

}

public boolean isEmpty() {
```

```
}

public boolean isFull() {

}
}

/**
* Your MyCircularQueue object will be instantiated and called as such:
* MyCircularQueue obj = new MyCircularQueue(k);
* boolean param_1 = obj.enQueue(value);
* boolean param_2 = obj.deQueue();
* int param_3 = obj.Front();
* int param_4 = obj.Rear();
* boolean param_5 = obj.isEmpty();
* boolean param_6 = obj.isFull();
*/
```

Python3:

```
class MyCircularQueue:

def __init__(self, k: int):


def enQueue(self, value: int) -> bool:


def deQueue(self) -> bool:


def Front(self) -> int:


def Rear(self) -> int:


def isEmpty(self) -> bool:


def isFull(self) -> bool:
```

```
# Your MyCircularQueue object will be instantiated and called as such:
# obj = MyCircularQueue(k)
# param_1 = obj.enQueue(value)
# param_2 = obj.deQueue()
# param_3 = obj.Front()
# param_4 = obj.Rear()
# param_5 = obj.isEmpty()
# param_6 = obj.isFull()
```