# 题目

设计你的循环队列实现。循环队列是一种线性数据结构，其操作表现基于FIFO（先进先出）原则并且队尾被连接在队首之后以形成一个循环。它也被称为“环形缓冲器”。

循环队列的一个好处是我们可以利用这个队列之前用过的空间。在一个普通队列里，一旦一个队列满了，我们就不能插入下一个元素，即使在队列前面仍有空间。但是使用循环队列，我们能使用这些空间去存储新的值。

你的实现应该支持如下操作：

MyCircularQueue(k): 构造器，设置队列长度为 k 。

Front: 从队首获取元素。如果队列为空，返回 -1 。

Rear: 获取队尾元素。如果队列为空，返回 -1 。

enQueue(value): 向循环队列插入一个元素。如果成功插入则返回真。

deQueue(): 从循环队列中删除一个元素。如果成功删除则返回真。

isEmpty(): 检查循环队列是否为空。

isFull(): 检查循环队列是否已满。

# 分析

## 方法一：模拟

**思路：**

**代码：**

class MyCircularQueue {

public:

/\*\* Initialize your data structure here. Set the size of the queue to be k. \*/

MyCircularQueue(int k) {

m\_head = 0;

m\_tail = 0;

m\_capacity = k;

m\_size = 0;

m\_arr = new int[k];

}

/\*Insert an element into the circular queue. Return true if the operation is successful. \*/

bool enQueue(int value) {

if (isFull())

return false;

m\_arr[m\_tail] = value;

++m\_tail;

if (m\_tail == m\_capacity)

{

m\_tail = 0;

}

++m\_size;

return true;

}

/\*Delete an element from the circular queue. Return true if the operation is successful. \*/

bool deQueue() {

if (isEmpty()) **//出队判空，入堆判满**

return false;

++m\_head; //出队要先++m\_head后判断是否越界

if (m\_head == m\_capacity)

{

m\_head = 0;

}

--m\_size;

return true;

}

/\*\* Get the front item from the queue. \*/

int Front() {

if (isEmpty())

return -1;

return m\_arr[m\_head];//front直接用m\_head就可以获取

}

/\*\* Get the last item from the queue. \*/

int Rear() {

if (isEmpty())

return -1;

//注意：最后元素的索引是m\_tail的上一个元素

int index = m\_tail - 1;

if (index < 0)

{

index = m\_capacity - 1;

//对于循环队列rear无法直接使用m\_tail获取

}

return m\_arr[index];

}

/\*\* Checks whether the circular queue is empty or not. \*/

bool isEmpty() {

return m\_size == 0;

}

/\*\* Checks whether the circular queue is full or not. \*/

bool isFull() {

return m\_size == m\_capacity;

}

int size() {

return m\_size;

}

private:

int\* m\_arr;

int m\_head; //待出队位置

int m\_tail; //待入队位置

int m\_size; //元素个数

int m\_capacity; //容量固定长度

};

另外一种写法：

class MyCircularQueue {

private:

int \*m\_data; //数据域

int m\_front; //指针域

int m\_rear;

int m\_size; //用于判空和溢出，这样比较方便

int m\_capacity;

public:

/\*\* Initialize your data structure here. Set the size of the queue to be k. \*/

MyCircularQueue(int k) {

m\_data = new int[k];

m\_front = m\_rear = 0;

m\_size = 0;

m\_capacity = k;

}

/\*\* Insert an element into the circular queue. Return true if the operation is successful. \*/

bool enQueue(int value) {

if(isFull())

return false;

m\_data[m\_rear] = value;

m\_rear++;

if(m\_rear == m\_capacity)

m\_rear = 0;

m\_size++;

return true;

}

/\*\* Delete an element from the circular queue. Return true if the operation is successful. \*/

bool deQueue() {

if(isEmpty())

return false;

m\_front++;

if(m\_front == m\_capacity)

m\_front = 0;

m\_size--;

return true;

}

/\*\* Get the front item from the queue. \*/

int Front() {

if(isEmpty())

return -1;

return m\_data[m\_front];

}

/\*\* Get the last item from the queue. \*/

int Rear() {

if(isEmpty())

return -1;

int index = m\_rear -1;

if(index < 0)

index = m\_capacity - 1;

return m\_data[index];

}

/\*\* Checks whether the circular queue is empty or not. \*/

bool isEmpty() {

return (0 == m\_size);

}

/\*\* Checks whether the circular queue is full or not. \*/

bool isFull() {

return (m\_size == m\_capacity);

}

};

/\*\*

\* 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();

\*/