Title: Understanding the Implementation of MyCircularQueue LEETCODE 622

Problem Statement: A circular queue is a linear data structure that follows the FIFO (First In First Out) principle but connects the last position back to the first position to form a circle. This design allows efficient utilization of memory by using vacant spaces created after dequeuing. The objective is to implement a circular queue with the following functionalities:

1. **MyCircularQueue(k)** - Initializes the queue with size **k**.
2. **Front()** - Returns the front item of the queue. If empty, return **-1**.
3. **Rear()** - Returns the last item of the queue. If empty, return **-1**.
4. **enQueue(int value)** - Inserts an element. Returns **true** if successful.
5. **deQueue()** - Removes an element. Returns **true** if successful.
6. **isEmpty()** - Checks if the queue is empty.
7. **isFull()** - Checks if the queue is full.

** Code Implementation:**

```
class MyCircularQueue {
public:
  int f;
  int b;
  vector<int> arr;
  int c;
  int s;
  MyCircularQueue(int k) {
    f = 0;
    b = 0;
    vector<int> v(k);
    arr = v;
    c = k;
    s = 0;//current size
  }

  bool enQueue(int value) {//to push a value in queue;
  if(s >= c) return false;
arr[b] = value;

b++;

if(b == c) b = 0;

s++;

return true;

};


bool deQueue() {//to pop a value following the fifo principle applied in a queue

   if(s == 0) return false;

   else if(f == c-1) f = 0;

   else f++;

   s--;

   return true;

}


int Front() {

   if(s == 0) return -1;

   else return arr[f];

}


int Rear() {

   if(s == 0) return -1;

   else if(b==0) return arr[c-1];

   else return arr[b-1];

}


bool isEmpty() {

   if(s == 0) return true;

   else return false;

}
```

```
  bool isFull() {
    if(s == c) return true;
    else return false;
  }
};
```

Code Breakdown:

1. **Class Variables:**
   - f (Front index)
   - b (Back index)
   - arr (Vector storing queue elements)
   - c (Capacity of queue)
   - s (Current size of queue)
2. **Constructor:**
   - Initializes f and b to **0**.
   - Creates a vector of size k.
   - Initializes c to **k** and s to **0**.
3. **enQueue(int value)**
   - Checks if queue is full (s >= c).
   - Inserts value at b, then increments b.
   - Wraps b back to 0 if it reaches c (circular behavior).
   - Increments size s.
4. **deQueue()**
   - Checks if queue is empty (s == 0).
   - Increments f and wraps it using f = 0 if needed.
   - Decrements size s.
5. **Front()**
   - Returns the front element (arr[f]) or **-1** if empty.
6. **Rear()**
   - Returns the last element in queue using b-1 handling circular cases.
7. **isEmpty()**** & ****isFull()**

- Simply check if s == 0 or s == c.

Conclusion: This implementation correctly follows the principles of a circular queue using an array. The indices f and b are managed using modular arithmetic to wrap around when reaching the array bounds. The time complexity for all operations is **O(1)**, making it an efficient solution.