Problem 622: Design Circular Queue

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

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()
```

Python:

```
class MyCircularQueue(object):

def __init__(self, k):
"""
:type k: int
"""



def enQueue(self, value):
"""
:type value: int
:rtype: bool
"""



def deQueue(self):
"""
:rtype: bool
"""



def Front(self):
```

```
        """
        :rtype: int
        """


    def Rear(self):
        """
        :rtype: int
        """


    def isEmpty(self):
        """
        :rtype: bool
        """


    def isFull(self):
        """
        :rtype: 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()
```

JavaScript:

```
/**
* @param {number} k
*/
var MyCircularQueue = function(k) {

};
```

```
/**
* @param {number} value
* @return {boolean}
*/
MyCircularQueue.prototype.enQueue = function(value) {

};

/**
* @return {boolean}
*/
MyCircularQueue.prototype.deQueue = function() {

};

/**
* @return {number}
*/
MyCircularQueue.prototype.Front = function() {

};

/**
* @return {number}
*/
MyCircularQueue.prototype.Rear = function() {

};

/**
* @return {boolean}
*/
MyCircularQueue.prototype.isEmpty = function() {

};

/**
* @return {boolean}
*/
MyCircularQueue.prototype.isFull = function() {

};
```

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

TypeScript:

```
class MyCircularQueue {
constructor(k: number) {

}

enQueue(value: number): boolean {

}

deQueue(): boolean {

}

Front(): number {

}

Rear(): number {

}

isEmpty(): boolean {

}

isFull(): boolean {

}
```

```
}

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

C#:

```
public class MyCircularQueue {

public MyCircularQueue(int k) {

}

public bool EnQueue(int value) {

}

public bool DeQueue() {

}

public int Front() {

}

public int Rear() {

}

public bool IsEmpty() {

}

public 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();
*/
```

C:

```
typedef struct {

} MyCircularQueue;


MyCircularQueue* myCircularQueueCreate(int k) {

}

bool myCircularQueueEnQueue(MyCircularQueue* obj, int value) {

}

bool myCircularQueueDeQueue(MyCircularQueue* obj) {

}

int myCircularQueueFront(MyCircularQueue* obj) {

}

int myCircularQueueRear(MyCircularQueue* obj) {
```

```
}

bool myCircularQueueIsEmpty(MyCircularQueue* obj) {

}

bool myCircularQueueIsFull(MyCircularQueue* obj) {

}

void myCircularQueueFree(MyCircularQueue* obj) {

}

/**
* Your MyCircularQueue struct will be instantiated and called as such:
* MyCircularQueue* obj = myCircularQueueCreate(k);
* bool param_1 = myCircularQueueEnQueue(obj, value);

* bool param_2 = myCircularQueueDeQueue(obj);

* int param_3 = myCircularQueueFront(obj);

* int param_4 = myCircularQueueRear(obj);

* bool param_5 = myCircularQueueIsEmpty(obj);

* bool param_6 = myCircularQueueIsFull(obj);

* myCircularQueueFree(obj);
*/
```

Go:

```
type MyCircularQueue struct {

}


func Constructor(k int) MyCircularQueue {
```

```
}


func (this *MyCircularQueue) EnQueue(value int) bool {

}


func (this *MyCircularQueue) DeQueue() bool {

}


func (this *MyCircularQueue) Front() int {

}


func (this *MyCircularQueue) Rear() int {

}


func (this *MyCircularQueue) IsEmpty() bool {

}


func (this *MyCircularQueue) IsFull() bool {

}


/**
* Your MyCircularQueue object will be instantiated and called as such:
* obj := Constructor(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();
```

```
*/
```

Kotlin:

```
class MyCircularQueue(k: Int) {

fun enQueue(value: Int): Boolean {

}

fun deQueue(): Boolean {

}

fun Front(): Int {

}

fun Rear(): Int {

}

fun isEmpty(): Boolean {

}

fun isFull(): Boolean {

}

}

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

Swift:

```
class MyCircularQueue {

init(_ k: Int) {

}

func enQueue(_ value: Int) -> Bool {

}

func deQueue() -> Bool {

}

func Front() -> Int {

}

func Rear() -> Int {

}

func isEmpty() -> Bool {

}

func isFull() -> Bool {

}
}

/**
* Your MyCircularQueue object will be instantiated and called as such:
* let obj = MyCircularQueue(k)
* let ret_1: Bool = obj.enQueue(value)
* let ret_2: Bool = obj.deQueue()
* let ret_3: Int = obj.Front()
* let ret_4: Int = obj.Rear()
* let ret_5: Bool = obj.isEmpty()
```

```
* let ret_6: Bool = obj.isFull()
*/
```

Rust:

```
struct MyCircularQueue {

}


/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl MyCircularQueue {

fn new(k: i32) -> Self {

}

fn en_queue(&self, value: i32) -> bool {

}

fn de_queue(&self) -> bool {

}

fn front(&self) -> i32 {

}

fn rear(&self) -> i32 {

}

fn is_empty(&self) -> bool {

}

fn is_full(&self) -> bool {
```

```
}
}

/**
* Your MyCircularQueue object will be instantiated and called as such:
* let obj = MyCircularQueue::new(k);
* let ret_1: bool = obj.en_queue(value);
* let ret_2: bool = obj.de_queue();
* let ret_3: i32 = obj.front();
* let ret_4: i32 = obj.rear();
* let ret_5: bool = obj.is_empty();
* let ret_6: bool = obj.is_full();
*/
```

Ruby:

```
class MyCircularQueue

=begin
:type k: Integer
=end
def initialize(k)

end


=begin
:type value: Integer
:rtype: Boolean
=end
def en_queue(value)

end


=begin
:rtype: Boolean
=end
def de_queue()

end
```

```
=begin
:rtype: Integer
=end
def front()

end


=begin
:rtype: Integer
=end
def rear()

end


=begin
:rtype: Boolean
=end
def is_empty()

end


=begin
:rtype: Boolean
=end
def is_full()

end


end

# Your MyCircularQueue object will be instantiated and called as such:
# obj = MyCircularQueue.new(k)
# param_1 = obj.en_queue(value)
# param_2 = obj.de_queue()
# param_3 = obj.front()
# param_4 = obj.rear()
# param_5 = obj.is_empty()
```

```
# param_6 = obj.is_full()
```

PHP:

```
class MyCircularQueue {
/**
* @param Integer $k
*/
function __construct($k) {

}

/**
* @param Integer $value
* @return Boolean
*/
function enQueue($value) {

}

/**
* @return Boolean
*/
function deQueue() {

}

/**
* @return Integer
*/
function Front() {

}

/**
* @return Integer
*/
function Rear() {

}

/**
```

```
* @return Boolean
*/
function isEmpty() {

}

/**
* @return Boolean
*/
function isFull() {

}
}

/**
* Your MyCircularQueue object will be instantiated and called as such:
* $obj = MyCircularQueue($k);
* $ret_1 = $obj->enQueue($value);
* $ret_2 = $obj->deQueue();
* $ret_3 = $obj->Front();
* $ret_4 = $obj->Rear();
* $ret_5 = $obj->isEmpty();
* $ret_6 = $obj->isFull();
*/
```

Dart:

```
class MyCircularQueue {

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 = MyCircularQueue(k);
* bool param1 = obj.enQueue(value);
* bool param2 = obj.deQueue();
* int param3 = obj.Front();
* int param4 = obj.Rear();
* bool param5 = obj.isEmpty();
* bool param6 = obj.isFull();
*/
```

Scala:

```
class MyCircularQueue(_k: Int) {

def enQueue(value: Int): Boolean = {

}

def deQueue(): Boolean = {

}

def Front(): Int = {

}
```

```
def Rear(): Int = {

}

def isEmpty(): Boolean = {

}

def isFull(): Boolean = {

}

}

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

Elixir:

```
defmodule MyCircularQueue do
@spec init_(k :: integer) :: any
def init_(k) do

end

@spec en_queue(value :: integer) :: boolean
def en_queue(value) do

end

@spec de_queue() :: boolean
def de_queue() do
```

```
end

@spec front() :: integer
def front() do

end

@spec rear() :: integer
def rear() do

end

@spec is_empty() :: boolean
def is_empty() do

end

@spec is_full() :: boolean
def is_full() do

end
end

# Your functions will be called as such:
# MyCircularQueue.init_(k)
# param_1 = MyCircularQueue.en_queue(value)
# param_2 = MyCircularQueue.de_queue()
# param_3 = MyCircularQueue.front()
# param_4 = MyCircularQueue.rear()
# param_5 = MyCircularQueue.is_empty()
# param_6 = MyCircularQueue.is_full()

# MyCircularQueue.init_ will be called before every test case, in which you
can do some necessary initializations.
```

Erlang:

```
-spec my_circular_queue_init_(K :: integer()) -> any().
my_circular_queue_init_(K) ->
.

-spec my_circular_queue_en_queue(Value :: integer()) -> boolean().
```

```
my_circular_queue_en_queue(Value) ->
.

-spec my_circular_queue_de_queue() -> boolean().
my_circular_queue_de_queue() ->
.

-spec my_circular_queue_front() -> integer().
my_circular_queue_front() ->
.

-spec my_circular_queue_rear() -> integer().
my_circular_queue_rear() ->
.

-spec my_circular_queue_is_empty() -> boolean().
my_circular_queue_is_empty() ->
.

-spec my_circular_queue_is_full() -> boolean().
my_circular_queue_is_full() ->
.


%% Your functions will be called as such:
%% my_circular_queue_init_(K),
%% Param_1 = my_circular_queue_en_queue(Value),
%% Param_2 = my_circular_queue_de_queue(),
%% Param_3 = my_circular_queue_front(),
%% Param_4 = my_circular_queue_rear(),
%% Param_5 = my_circular_queue_is_empty(),
%% Param_6 = my_circular_queue_is_full(),

%% my_circular_queue_init_ will be called before every test case, in which
you can do some necessary initializations.
```

Racket:

```
(define my-circular-queue%
(class object%
(super-new)
```

```
; k : exact-integer?
(init-field
k)

; en-queue : exact-integer? -> boolean?
(define/public (en-queue value)
)
; de-queue : -> boolean?
(define/public (de-queue)
)
; front : -> exact-integer?
(define/public (front)
)
; rear : -> exact-integer?
(define/public (rear)
)
; is-empty : -> boolean?
(define/public (is-empty)
)
; is-full : -> boolean?
(define/public (is-full)
)))

;; Your my-circular-queue% object will be instantiated and called as such:
;; (define obj (new my-circular-queue% [k k]))
;; (define param_1 (send obj en-queue value))
;; (define param_2 (send obj de-queue))
;; (define param_3 (send obj front))
;; (define param_4 (send obj rear))
;; (define param_5 (send obj is-empty))
;; (define param_6 (send obj is-full))
```

Solutions

C++ Solution:

```
/*
* Problem: Design Circular Queue
* Difficulty: Medium
* Tags: array, linked_list, queue
```

```
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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 Solution:

```
/**
* Problem: Design Circular Queue
* Difficulty: Medium
* Tags: array, linked_list, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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 Solution:

```
"""
Problem: Design Circular Queue
Difficulty: Medium
Tags: array, linked_list, queue

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class MyCircularQueue:

def __init__(self, k: int):


def enQueue(self, value: int) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class MyCircularQueue(object):

def __init__(self, k):
"""
:type k: int
"""


def enQueue(self, value):
"""
:type value: int
:rtype: bool
"""


def deQueue(self):
"""
:rtype: bool
"""


def Front(self):
"""
:rtype: int
"""


def Rear(self):
"""
:rtype: int
"""


def isEmpty(self):
"""
:rtype: bool
"""


def isFull(self):
"""
:rtype: 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()
```

JavaScript Solution:

```
/**
* Problem: Design Circular Queue
* Difficulty: Medium
* Tags: array, linked_list, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number} k
*/
var MyCircularQueue = function(k) {

};

/**
* @param {number} value
* @return {boolean}
*/
MyCircularQueue.prototype.enQueue = function(value) {

};

/**
* @return {boolean}
```

```
*/
MyCircularQueue.prototype.deQueue = function() {

};

/**
* @return {number}
*/
MyCircularQueue.prototype.Front = function() {

};

/**
* @return {number}
*/
MyCircularQueue.prototype.Rear = function() {

};

/**
* @return {boolean}
*/
MyCircularQueue.prototype.isEmpty = function() {

};

/**
* @return {boolean}
*/
MyCircularQueue.prototype.isFull = function() {

};

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

```
*/
```

TypeScript Solution:

```
/**
* Problem: Design Circular Queue
* Difficulty: Medium
* Tags: array, linked_list, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class MyCircularQueue {
constructor(k: number) {

}

enQueue(value: number): boolean {

}

deQueue(): boolean {

}

Front(): number {

}

Rear(): number {

}

isEmpty(): boolean {

}

isFull(): boolean {
```

```
}
}

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

C# Solution:

```
/*
* Problem: Design Circular Queue
* Difficulty: Medium
* Tags: array, linked_list, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class MyCircularQueue {

public MyCircularQueue(int k) {

}

public bool EnQueue(int value) {

}

public bool DeQueue() {

}

public int Front() {
```

```
}

public int Rear() {

}

public bool IsEmpty() {

}

public 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();
*/
```

C Solution:

```
/*
* Problem: Design Circular Queue
* Difficulty: Medium
* Tags: array, linked_list, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
typedef struct {

} MyCircularQueue;


MyCircularQueue* myCircularQueueCreate(int k) {

}

bool myCircularQueueEnQueue(MyCircularQueue* obj, int value) {

}

bool myCircularQueueDeQueue(MyCircularQueue* obj) {

}

int myCircularQueueFront(MyCircularQueue* obj) {

}

int myCircularQueueRear(MyCircularQueue* obj) {

}

bool myCircularQueueIsEmpty(MyCircularQueue* obj) {

}

bool myCircularQueueIsFull(MyCircularQueue* obj) {

}

void myCircularQueueFree(MyCircularQueue* obj) {

}

/**
* Your MyCircularQueue struct will be instantiated and called as such:
* MyCircularQueue* obj = myCircularQueueCreate(k);
* bool param_1 = myCircularQueueEnQueue(obj, value);
```

```
* bool param_2 = myCircularQueueDeQueue(obj);

* int param_3 = myCircularQueueFront(obj);

* int param_4 = myCircularQueueRear(obj);

* bool param_5 = myCircularQueueIsEmpty(obj);

* bool param_6 = myCircularQueueIsFull(obj);

* myCircularQueueFree(obj);
*/
```

Go Solution:

```
// Problem: Design Circular Queue
// Difficulty: Medium
// Tags: array, linked_list, queue
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

type MyCircularQueue struct {

}


func Constructor(k int) MyCircularQueue {

}


func (this *MyCircularQueue) EnQueue(value int) bool {

}


func (this *MyCircularQueue) DeQueue() bool {
```

```
}


func (this *MyCircularQueue) Front() int {

}


func (this *MyCircularQueue) Rear() int {

}


func (this *MyCircularQueue) IsEmpty() bool {

}


func (this *MyCircularQueue) IsFull() bool {

}


/**
* Your MyCircularQueue object will be instantiated and called as such:
* obj := Constructor(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();
*/
```

Kotlin Solution:

```
class MyCircularQueue(k: Int) {

fun enQueue(value: Int): Boolean {

}
```

```
fun deQueue(): Boolean {

}

fun Front(): Int {

}

fun Rear(): Int {

}

fun isEmpty(): Boolean {

}

fun isFull(): Boolean {

}

}

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

Swift Solution:

```
class MyCircularQueue {

init(_ k: Int) {
```

```
}

func enQueue(_ value: Int) -> Bool {

}

func deQueue() -> Bool {

}

func Front() -> Int {

}

func Rear() -> Int {

}

func isEmpty() -> Bool {

}

func isFull() -> Bool {

}
}

/**
* Your MyCircularQueue object will be instantiated and called as such:
* let obj = MyCircularQueue(k)
* let ret_1: Bool = obj.enQueue(value)
* let ret_2: Bool = obj.deQueue()
* let ret_3: Int = obj.Front()
* let ret_4: Int = obj.Rear()
* let ret_5: Bool = obj.isEmpty()
* let ret_6: Bool = obj.isFull()
*/
```

Rust Solution:

```rust
// Problem: Design Circular Queue
// Difficulty: Medium
// Tags: array, linked_list, queue
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

struct MyCircularQueue {

}



/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl MyCircularQueue {

fn new(k: i32) -> Self {

}

fn en_queue(&self, value: i32) -> bool {

}

fn de_queue(&self) -> bool {

}

fn front(&self) -> i32 {

}

fn rear(&self) -> i32 {

}

fn is_empty(&self) -> bool {

}
```

```
fn is_full(&self) -> bool {

}
}

/**
* Your MyCircularQueue object will be instantiated and called as such:
* let obj = MyCircularQueue::new(k);
* let ret_1: bool = obj.en_queue(value);
* let ret_2: bool = obj.de_queue();
* let ret_3: i32 = obj.front();
* let ret_4: i32 = obj.rear();
* let ret_5: bool = obj.is_empty();
* let ret_6: bool = obj.is_full();
*/
```

Ruby Solution:

```
class MyCircularQueue

=begin
:type k: Integer
=end
def initialize(k)

end


=begin
:type value: Integer
:rtype: Boolean
=end
def en_queue(value)

end


=begin
:rtype: Boolean
=end
def de_queue()
```

```
end


=begin
:rtype: Integer
=end
def front()

end


=begin
:rtype: Integer
=end
def rear()

end


=begin
:rtype: Boolean
=end
def is_empty()

end


=begin
:rtype: Boolean
=end
def is_full()

end


end

# Your MyCircularQueue object will be instantiated and called as such:
# obj = MyCircularQueue.new(k)
# param_1 = obj.en_queue(value)
# param_2 = obj.de_queue()
```

```
# param_3 = obj.front()
# param_4 = obj.rear()
# param_5 = obj.is_empty()
# param_6 = obj.is_full()
```

PHP Solution:

```
class MyCircularQueue {
/**
* @param Integer $k
*/
function __construct($k) {

}

/**
* @param Integer $value
* @return Boolean
*/
function enQueue($value) {

}

/**
* @return Boolean
*/
function deQueue() {

}

/**
* @return Integer
*/
function Front() {

}

/**
* @return Integer
*/
function Rear() {
```

```
}

/**
* @return Boolean
*/
function isEmpty() {

}

/**
* @return Boolean
*/
function isFull() {

}
}

/**
* Your MyCircularQueue object will be instantiated and called as such:
* $obj = MyCircularQueue($k);
* $ret_1 = $obj->enQueue($value);
* $ret_2 = $obj->deQueue();
* $ret_3 = $obj->Front();
* $ret_4 = $obj->Rear();
* $ret_5 = $obj->isEmpty();
* $ret_6 = $obj->isFull();
*/
```

Dart Solution:

```
class MyCircularQueue {

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 = MyCircularQueue(k);
* bool param1 = obj.enQueue(value);
* bool param2 = obj.deQueue();
* int param3 = obj.Front();
* int param4 = obj.Rear();
* bool param5 = obj.isEmpty();
* bool param6 = obj.isFull();
*/
```

Scala Solution:

```
class MyCircularQueue(_k: Int) {

def enQueue(value: Int): Boolean = {

}

def deQueue(): Boolean = {
```

```
}

def Front(): Int = {

}

def Rear(): Int = {

}

def isEmpty(): Boolean = {

}

def isFull(): Boolean = {

}

}

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

Elixir Solution:

```
defmodule MyCircularQueue do
@spec init_(k :: integer) :: any
def init_(k) do

end

@spec en_queue(value :: integer) :: boolean
```

```
def en_queue(value) do

end

@spec de_queue() :: boolean
def de_queue() do

end

@spec front() :: integer
def front() do

end

@spec rear() :: integer
def rear() do

end

@spec is_empty() :: boolean
def is_empty() do

end

@spec is_full() :: boolean
def is_full() do

end
end

# Your functions will be called as such:
# MyCircularQueue.init_(k)
# param_1 = MyCircularQueue.en_queue(value)
# param_2 = MyCircularQueue.de_queue()
# param_3 = MyCircularQueue.front()
# param_4 = MyCircularQueue.rear()
# param_5 = MyCircularQueue.is_empty()
# param_6 = MyCircularQueue.is_full()

# MyCircularQueue.init_ will be called before every test case, in which you
can do some necessary initializations.
```

Erlang Solution:

```
-spec my_circular_queue_init_(K :: integer()) -> any().
my_circular_queue_init_(K) ->
.

-spec my_circular_queue_en_queue(Value :: integer()) -> boolean().
my_circular_queue_en_queue(Value) ->
.

-spec my_circular_queue_de_queue() -> boolean().
my_circular_queue_de_queue() ->
.

-spec my_circular_queue_front() -> integer().
my_circular_queue_front() ->
.

-spec my_circular_queue_rear() -> integer().
my_circular_queue_rear() ->
.

-spec my_circular_queue_is_empty() -> boolean().
my_circular_queue_is_empty() ->
.

-spec my_circular_queue_is_full() -> boolean().
my_circular_queue_is_full() ->
.


%% Your functions will be called as such:
%% my_circular_queue_init_(K),
%% Param_1 = my_circular_queue_en_queue(Value),
%% Param_2 = my_circular_queue_de_queue(),
%% Param_3 = my_circular_queue_front(),
%% Param_4 = my_circular_queue_rear(),
%% Param_5 = my_circular_queue_is_empty(),
%% Param_6 = my_circular_queue_is_full(),

%% my_circular_queue_init_ will be called before every test case, in which
you can do some necessary initializations.
```

Racket Solution:

```
(define my-circular-queue%
(class object%
(super-new)

; k : exact-integer?
(init-field
k)

; en-queue : exact-integer? -> boolean?
(define/public (en-queue value)
)
; de-queue : -> boolean?
(define/public (de-queue)
)
; front : -> exact-integer?
(define/public (front)
)
; rear : -> exact-integer?
(define/public (rear)
)
; is-empty : -> boolean?
(define/public (is-empty)
)
; is-full : -> boolean?
(define/public (is-full)
)))

;; Your my-circular-queue% object will be instantiated and called as such:
;; (define obj (new my-circular-queue% [k k]))
;; (define param_1 (send obj en-queue value))
;; (define param_2 (send obj de-queue))
;; (define param_3 (send obj front))
;; (define param_4 (send obj rear))
;; (define param_5 (send obj is-empty))
;; (define param_6 (send obj is-full))
```