# Question

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.

Implementation 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.

**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.

**Follow up:** Could you solve the problem without using the built-in queue?

# Solution

#### **Approach 1: Array**

**Intuition**

Based on the description of the problem, an intuitive data structure that meets all the requirements would be a **ring** where the head and the tail are adjacent to each other.

However, there does not exist a ring data structure in any programming language. A similar data structure at our disposal is the one called **Array** which is a collection of elements that reside continuously in one dimensional space.

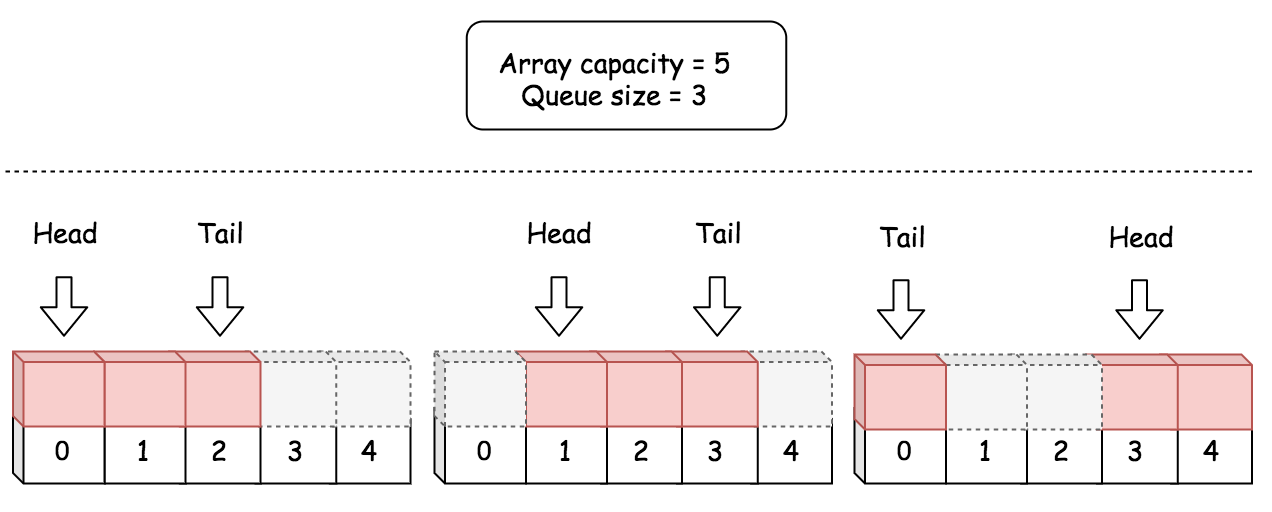
The essence of many design problems, is how one can build more advanced data structures with the basic building blocks such as Array.

In this case, to build a circular queue, we could form a virtual ring structure with the Array, via the manipulation of index.

Given a fixed size array, any of the elements could be considered as a head in a queue. As long as we know the length of the queue, we then can instantly locate its tail, based on the following formula:

tailIndex=(headIndex+count−1)modcapacity

where the variable capacity is the size of the array, the count is the length of the queue and the headIndex and tailIndex are the indice of head and tail elements respectively in the array. Here we showcase a few examples how a circular queue could reside in a fixed size array.



**Algorithm**

The procedure to design a data structure lies essentially on how we design the attributes within the data structure.

One of the traits of a good design is to have as less attributes as possible, which arguably could bring several benefits.

* Less attributes usually implies little or no redundancy among the attributes.
* The less redundant the attributes are, the simpler the manipulation logic, which eventually could be less error-prone.
* Less attributes also requires less space and therefore it could also bring efficiency to the runtime performance.

However, it is not advisable to seek for the minimum set of attributes. Sometimes, a bit of redundancy could help with the time complexity. After all, like many other problems, we are trying to strike a balance between the space and the time.

Following the above principles, here we give a list of attributes and the thoughts behind each attribute.

* queue: a fixed size array to hold the elements for the circular queue.
* headIndex: an integer which indicates the current head element in the circular queue.
* count: the current length of the circular queue, i.e. the number of elements in the circular queue. Together with the headIndex, we could locate the current tail element in the circular queue, based on the formula we gave previously. Therefore, we choose not to add another attribute for the index of tail.
* capacity: the capacity of the circular queue, i.e. the maximum number of elements that can be hold in the queue. One might argument that it is not absolutely necessary to add this attribute, since we could obtain the capacity from the queue attribute. It is true. Yet, since we would frequently use this capacity in our algorithm, we choose to keep it as an attribute, instead of invoking function len(queue) in Python at every occasion. Though in other programming languages such as Java, it might be more efficient to omit this attribute, since it is part of the attributes (queue.length) in Java array. Note: for the sake of consistency, we keep this attribute for all implementations.

|  |
| --- |
| class MyCircularQueue {  private int[] queue;  private int headIndex;  private int count;  private int capacity;  /\*\* Initialize your data structure here. Set the size of the queue to be k. \*/  public MyCircularQueue(int k) {  this.capacity = k;  this.queue = new int[k];  this.headIndex = 0;  this.count = 0;  }  /\*\* Insert an element into the circular queue. Return true if the operation is successful. \*/  public boolean enQueue(int value) {  if (this.count == this.capacity)  return false;  this.queue[(this.headIndex + this.count) % this.capacity] = value;  this.count += 1;  return true;  }  /\*\* Delete an element from the circular queue. Return true if the operation is successful. \*/  public boolean deQueue() {  if (this.count == 0)  return false;  this.headIndex = (this.headIndex + 1) % this.capacity;  this.count -= 1;  return true;  }  /\*\* Get the front item from the queue. \*/  public int Front() {  if (this.count == 0)  return -1;  return this.queue[this.headIndex];  }  /\*\* Get the last item from the queue. \*/  public int Rear() {  if (this.count == 0)  return -1;  int tailIndex = (this.headIndex + this.count - 1) % this.capacity;  return this.queue[tailIndex];  }  /\*\* Checks whether the circular queue is empty or not. \*/  public boolean isEmpty() {  return (this.count == 0);  }  /\*\* Checks whether the circular queue is full or not. \*/  public boolean isFull() {  return (this.count == this.capacity);  }  } |

**Complexity**

* Time complexity: \mathcal{O}(1)O(1). All of the methods in our circular data structure is of constant time complexity.
* Space Complexity: \mathcal{O}(N)O(*N*). The overall space complexity of the data structure is linear, where N*N* is the pre-assigned capacity of the queue. However, it is worth mentioning that the memory consumption of the data structure remains as its pre-assigned capacity during its entire life cycle.

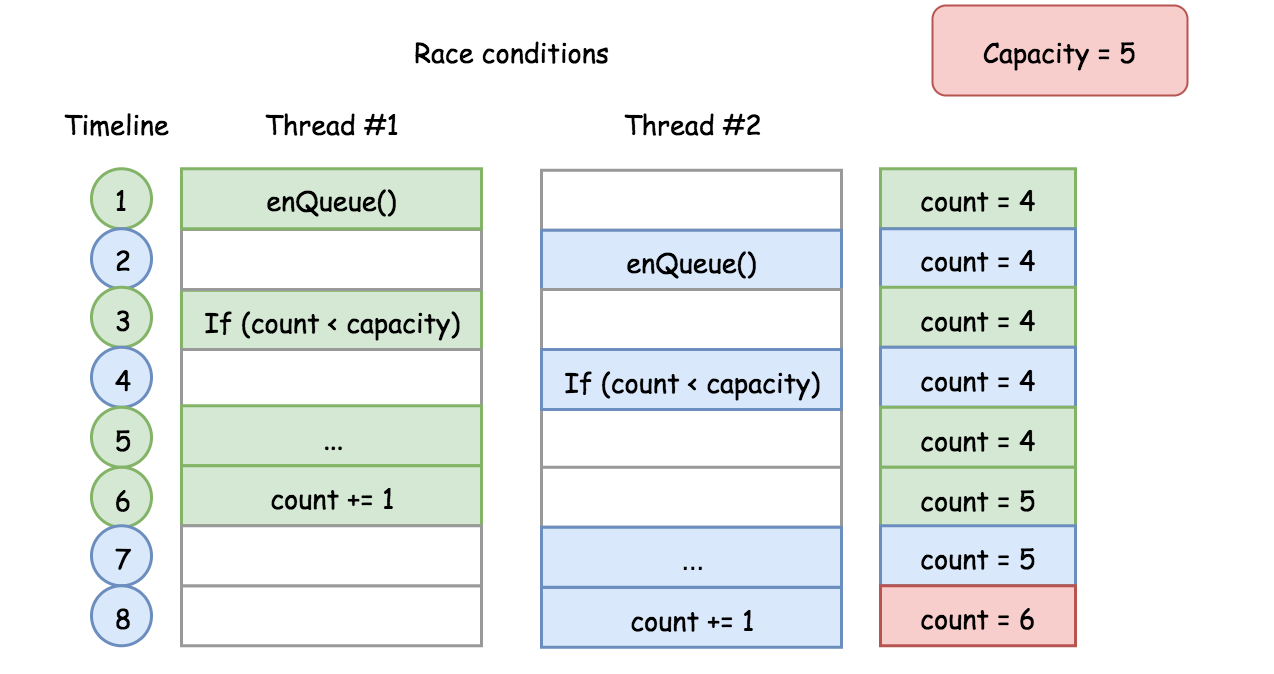
**Improvement: Thread-Safe**

One might be happy with the above implementation, after all it meets all the requirements of the problem.

As a followup question though, an interviewer might ask one to point out a potential defect about the implementation and the solution.

This time, it is not about the space or time complexity, but **concurrency**. Our circular queue is NOT thread-safe. One could end up with corrupting our data structure in a multi-threaded environment.

For example, here is an execution sequence where we exceed the designed capacity of the queue and overwrite the tail element undesirably.



The above scenario is called race conditions as described in the problem of [Print in Order](https://leetcode.com/problems/print-in-order/solution/). One can find more [concurrency problems](https://leetcode.com/problemset/concurrency/) to practice on LeetCode.

There are several ways to mitigate the above concurrency problem. Take the function enQueue(int value) as an example, we show how we can make the function thread-safe in the following implementation:

|  |
| --- |
| class MyCircularQueue {  private Node head, tail;  private int count;  private int capacity;  // Additional variable to secure the access of our queue  private ReentrantLock queueLock = new ReentrantLock();  /\*\* Initialize your data structure here. Set the size of the queue to be k. \*/  public MyCircularQueue(int k) {  this.capacity = k;  }  /\*\* Insert an element into the circular queue. Return true if the operation is successful. \*/  public boolean enQueue(int value) {  // ensure the exclusive access for the following block.  queueLock.lock();  try {  if (this.count == this.capacity)  return false;  Node newNode = new Node(value);  if (this.count == 0) {  head = tail = newNode;  } else {  tail.nextNode = newNode;  tail = newNode;  }  this.count += 1;  } finally {  queueLock.unlock();  }  return true;  }  } |

With the protection of locks, we now feel more confident to apply our circular queue in critical scenarios.

The above improvement does not alter the time and space complexity of the original data structure, though overall the thread-safe measures do incur some extra costs.

#### **Approach 2: Singly-Linked List**

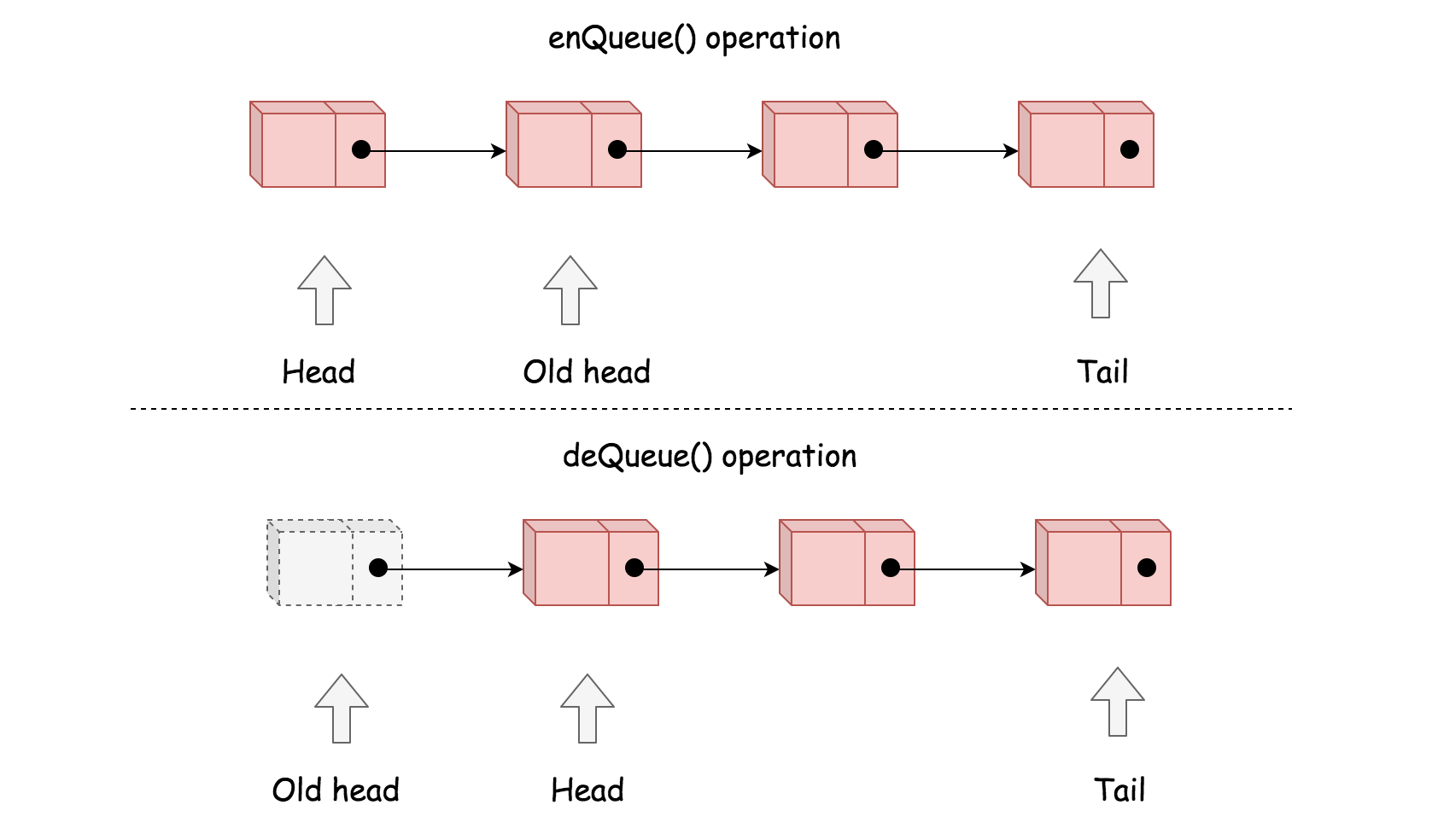
**Intuition**

Similar with Array, the Linked List is another common building block for more advanced data structures.

Different than a fixed size Array, a linked list could be more memory efficient, since it does not pre-allocate memory for unused capacity.

With a singly-linked list, we could design a circular queue with the same time and space complexity as the approach with Array. But we could gain some memory efficiency, since we don't need to pre-allocate the memory upfront.

In the following graph, we show how the operations of enQueue() and deQueue() can be done via singly-linked list.



**Algorithm**

Here we give a list of attributes in our circular queue data structure and the thoughts behind each attribute.

* capacity: the maximum number of elements that the circular queue will hold.
* head: the reference to the head element in the queue.
* count: the current length of the queue. This is a critical attribute that helps us to do the boundary check in each method.
* tail: the reference to the tail element in the queue. Unlike the Array approach, we need to explicitly keep the reference to the tail element. Without this attribute, it would take us \mathcal{O}(N)O(*N*) time complexity to locate the tail element from the head element.

|  |
| --- |
| class Node {  public int value;  public Node nextNode;  public Node(int value) {  this.value = value;  this.nextNode = null;  }  }  class MyCircularQueue {  private Node head, tail;  private int count;  private int capacity;  /\*\* Initialize your data structure here. Set the size of the queue to be k. \*/  public MyCircularQueue(int k) {  this.capacity = k;  }  /\*\* Insert an element into the circular queue. Return true if the operation is successful. \*/  public boolean enQueue(int value) {  if (this.count == this.capacity)  return false;  Node newNode = new Node(value);  if (this.count == 0) {  head = tail = newNode;  } else {  tail.nextNode = newNode;  tail = newNode;  }  this.count += 1;  return true;  }  /\*\* Delete an element from the circular queue. Return true if the operation is successful. \*/  public boolean deQueue() {  if (this.count == 0)  return false;  this.head = this.head.nextNode;  this.count -= 1;  return true;  }  /\*\* Get the front item from the queue. \*/  public int Front() {  if (this.count == 0)  return -1;  else  return this.head.value;  }  /\*\* Get the last item from the queue. \*/  public int Rear() {  if (this.count == 0)  return -1;  else  return this.tail.value;  }  /\*\* Checks whether the circular queue is empty or not. \*/  public boolean isEmpty() {  return (this.count == 0);  }  /\*\* Checks whether the circular queue is full or not. \*/  public boolean isFull() {  return (this.count == this.capacity);  }  } |

**Complexity**

* Time complexity: \mathcal{O}(1)O(1) for each method in our circular queue.
* Space Complexity: The upper bound of the memory consumption for our circular queue would be \mathcal{O}(N)O(*N*), same as the Array approach. However, it should be more memory efficient as we discussed in the intuition section.