讲究先来后到的“队列”

今日目标：

1：能够说出队列的存储结构及操作特点

2：能够说出java中队列相关的API

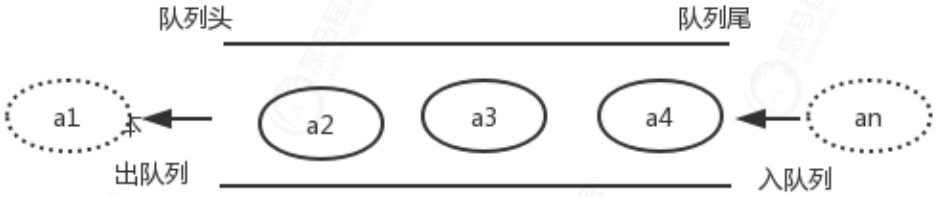
3：完成基于单链表的队列实现

4：完成实战题目

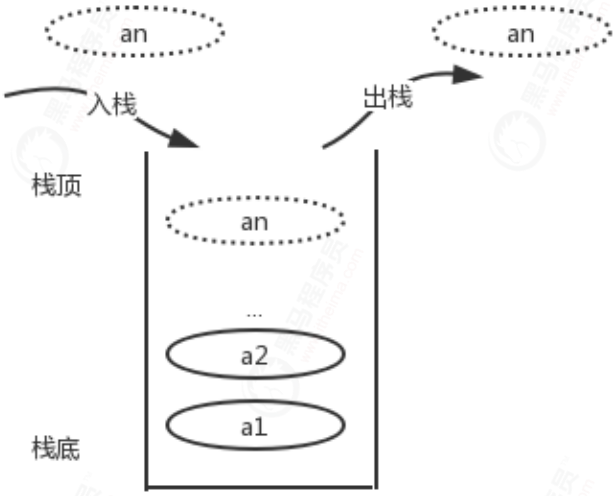
1、存储结构及特点

队列（**Queue**）和栈一样，代表具有某一类操作特征的数据结构，我们拿日常生活中的一个场景来举例说明，我们去车站的窗口买票，那就要排队，那先来的人就先买，后到的人就后买，先来的人排到队头，后来的人排在队尾，不允许插队，**先进先出，这就是典型的队列。** 队列先进先出的特点英文表示为：First In First Out即**FIFO**，为了更好的理解队列这种数据结构，我们以一幅图的形式来表示，并且我们将队列的特点和栈进行比较，如下：

队列：存储结构及操作特点



栈：存储结构及操作特点



队列和栈一样都属于一种操作受限的线性表，栈只允许在一端进行操作，分别是入栈和出栈，而队列跟栈很相似，支持的操作也有限，最基本的两个操作一个叫入队列，将数据插入到队列尾部，另一个叫出队，从队列头部取出一个数据。

注意：入队列和出队列操作的时间复杂度均为O(1)

2、队列的实现

2.1、java API

像队列和栈这种数据结构在高级语言中的实现特别的丰富，也特别的成熟。

Interface Queue：<https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Interface Deque：https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html

在两端支持元素插入和移除的一种线性集合，这个接口定义了访问deque两端元素的方法。

	First Element (Head)		**Last Element (Tail)**	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

Class PriorityQueue：https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html

元素不再遵循先进先出的特性了，出队列的顺序跟入队列的顺序无关，只跟元素的优先级有关系。队列中的每个元素都会指定一个优先级，根据优先级的大小关系出队列。

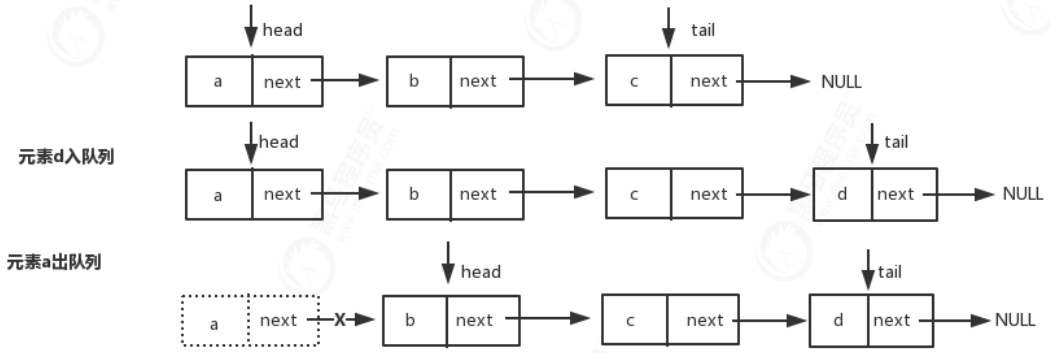
插入操作是O(1)的复杂度，而取出操作是O(log n)的复杂度。

PriorityQueue底层具体实现的数据结构较为多样和复杂度：heap，BST等

2.2、基于链表实现队列

跟栈一样，队列可以用数组来实现，也可以用链表来实现。用数组实现的栈叫作顺序栈，用链表实 现的栈叫作链式栈。同样，**用数组实现的队列叫作顺序队列，用链表实现的队列叫作链式队列。**

这一节我们来看基于单链表实现的队列，我们同样需要两个指针：head 指针和 tail 指针。它们分别指向链表的第一个结点和最后一个结点。如图所示，入队时，tail->next= new_node, tail = tail->next；出队时，head= head->nex，如下图所示：



（1）创建队列接口Queue：`com.itheima.queue.Queue<E>`

```
package com.itheima.queue;

/**
 * Created by 传智播客*黑马程序员.
 */
public interface Queue<E> {

    /**
     * 在不违反容量限制的情况下立即将指定的元素插入此队列，成功时返回true，
     * 如果当前没有可用空间，则抛出IllegalStateException异常
     * @param e
     * @return
     */
    boolean add(E e);

    /**
     * 在不违反容量限制的情况下立即将指定的元素插入到此队列中。成功时返回true，
     * @param e
     * @return
     */
    boolean offer(E e);

    /**
     * 检索并删除此队列的头。如果队列为空抛出NoSuchElementException
     * @return
     */
    E remove();

    /**
     * 检索并删除此队列的头，如果此队列为空，则返回null。
     * @return
     */
    E poll();

    /**
     * 检索但不删除此队列的头。如果队列为空抛出NoSuchElementException
     * 此方法与peek的不同之处在于，如果该队列为空，则会抛出异常。
```

```
     * @return
     */
    E element();

    /**
     * 检索但不删除此队列的头，或如果此队列为空，则返回null。
     * @return
     */
    E peek();
    /**
     * 返回队列中元素个数
     * @return
     */
    int size();

    /**
     * 判断队列是否为空
     * @return
     */
    boolean isEmpty();
}
```

(2) 创建实现类：`LinkedListQueue` 并实现 `Queue<E>` 接口，添加相应方法

(3) 编写构造，创建链表节点对象 `Node`，添加属性 `size`，头尾指针 `head`，`tail`

```
int size;

Node<E> head;

Node<E> tail;

public LinkedListQueue(){}

private static class Node<E>{
    E val;
    Node<E> next;
    public Node(E val,Node<E> next){
        this.val = val;
        this.next = next;
    }
}
```

(4) 完成 `size,isEmpty` 方法

```java
@Override
public int size() {
    return size;
}

@Override
public boolean isEmpty() {
    return size == 0;
}
```

(5) 完成`add,offer`方法

```java
@Override
public boolean add(E e) {
    linkLast(e);
    return true;
}

private void linkLast(E e){
    Node<E> t = tail;
    Node<E> newNode = new Node<>(e,null);
    tail = newNode;
    if (t == null) {
        head = newNode;
    }else {
        t.next = newNode;
    }
    size++;
}

@Override
public boolean offer(E e) {
    linkLast(e);
    return true;
}
```

> add和offer的差异不是任何情况下都有的，如果基于数组实现当容量不够时add可以抛异常，offer可以直接返回false，当然也可以扩容！

(6) 完成`remove,poll`方法

```java
@Override
public E remove() {
    if (size == 0) {
        throw new NoSuchElementException("队列为空!");
    }
    Node<E> h = unlinkHead();
    return h.val;
}

private Node<E> unlinkHead(){
```

```
    Node<E> h = head;
    head = h.next;
    h.next = null;
    size--;
    return h;
}

@Override
public E poll() {
    if (size == 0) {
        return null;
    }
    Node<E> h = unlinkHead();
    return h.val;
}
```

(7) 完成`element,peek`方法

```
@Override
public E element() {
    if (size == 0) {
        throw new NoSuchElementException("队列为空!");
    }
    Node<E> h = head;
    return h.val;
}

@Override
public E peek() {
    if (size == 0) {
        return null;
    }
    Node<E> h = head;
    return h.val;
}
```

(5) 完成`toString`方法

```
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    Node<E> h = head;
    while (h!=null){
        sb.append(h.val).append("->");
        h = h.next;
    }
    return sb.append("null").toString();
}
```

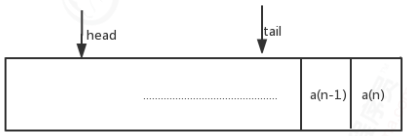
(6) 编写测试类：`com.itheima.queue.LinkedListQueueTest`

```java
public class LinkedListQueueTest {
    public static void main(String[] args) {
        Queue queue = new LinkedListQueue();
        queue.add("黑马程序员");
        queue.offer("博学谷");
        queue.offer("传智汇");
        queue.offer("传智专修学院");
        System.out.println("队列是否为空："+queue.isEmpty()+",队列元素个数为："+queue.size());
        System.out.println(queue);
        System.out.println("队列头元素:"+queue.remove());
        System.out.println(queue);
        System.out.println("队列头元素:"+queue.poll());
        System.out.println(queue);
        System.out.println("队列头元素:"+queue.element());
        System.out.println(queue);
        System.out.println("队列头元素:"+queue.peek());
        System.out.println(queue);
    }
}
```

2.3、小结

1. 队列的实现如果基于数组，其实就是操作下标，我们维护两个下标，head，tail分别代表队列的头，尾指针，如图：

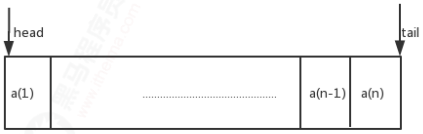
用两个下标head，tail代表队列头，尾指针



需要注意的几个特殊情况如下：

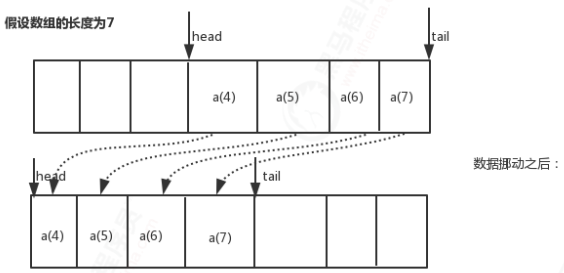
判断 尾指针已到达数组最右端，如图所示：

此时：我们无法再向数组尾部插入元素了，在这样一个情况下又分为两种情况：
1：数组中还有位置可以存储元素
2：数组中已经没有位置可以存储元素了
所以接下来有一个判断:`if(head==0){`



这属于第二种情况：数组中已经没有位置可以存储元素了，那接下来就要对队列进行扩容，执行 `grow(elements.length);` 底层的扩容逻辑和之前的扩容逻辑和ArrayList的扩容逻辑一样这里就不再赘述了

如果判断条件不满足：则属于第一种情况，数组中还有位置可以存储，只不过数组末尾不能在存储数据了而已，那此时我们需要做的事情就是将数组中的数据依次向前挪动，将数组尾部空出来可以继续插入元素，如下图所示：



数据挪动之后：

> **课后作业**：请按照此思路实现一个基于数组的队列

2. 当然，在java中有一个比较常用的实现：`java.util.LinkedList`，我们先来看它的定义

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
    {

    }
```

LinkedList实现了`List`，`Deque`接口，而`Deque`又继承自`Queue`接口

```
public interface Deque<E> extends Queue<E> {}
```

https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html

从`Deque`接口的定义可以看出，它里面不仅包含队列操作的相关api，比如`add,offer,peek,poll`等，还有双端队列操作的api，如`addFirst,offerFirst,peekFirst`等等。除此之外它还包含栈相关的操作api，如`push,pop`。

也就是说**`LinkedList`功能是多样性的，能当作`List`集合用，能当作`Queue`队列用，能当作`Deque`双端队列用，也能当作`Stack`栈来使用。**

> **课后作业**：请分析`LinkedList`底层的源码实现。

3、实战

3.1、622. 设计循环队列

https://leetcode-cn.com/problems/design-circular-queue/

循环队列很重要的一个作用是复用之前使用过的内存空间，适合用数组实现，使用链表的实现，创建结点和删除结点都是动态的，也就不存在需要循环利用的问题了。

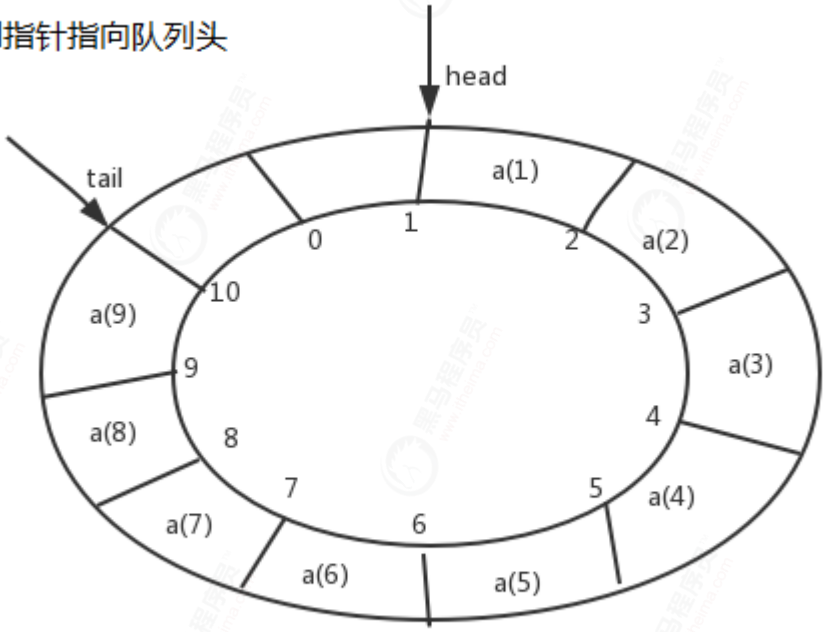
而且用数组实现循环队列也不要求我们对数组进行动态扩容与缩容。

思路分析：

循环队列，顾名思义，它长得像一个环。原本数组是有头有尾的，是一条直线。现在我们把首尾相连，形成了一个环，如下图所示：

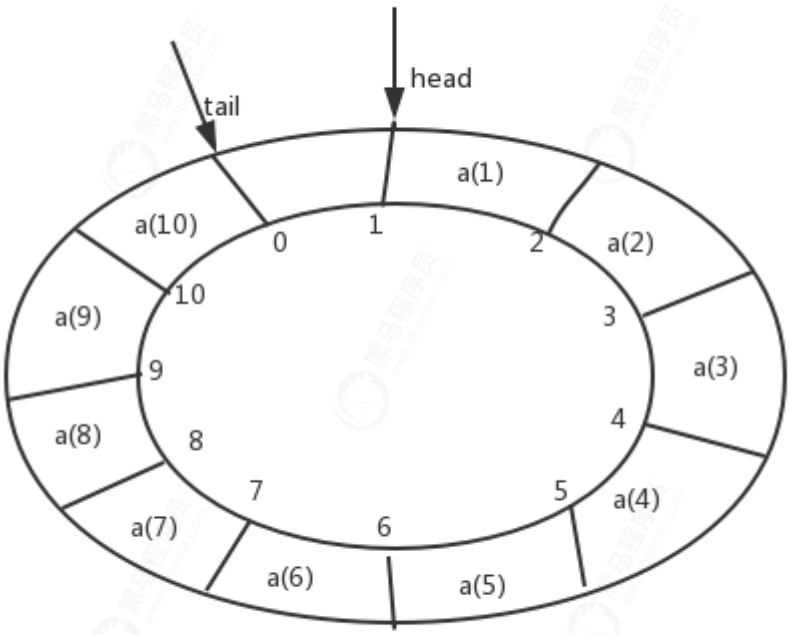
数组size=11

head指针指向队列头



tail指针指向队列尾部（即最后 1 个有效数据）的下一个位置，即下一个从队尾入队元素的位置。

从图中可知队列的大小为11，当前 head=1，tail=10。队列中9个元素，当有一个新的元素 a(10) 入队时，我们放入下标为10的位置。但这个时候，我们并不把 tail 更新为11，而是将其在环中后移一位，到下标为 0 的位置。所以，在 a(10)入队之后，循环队列中的元素就变成了下面的样子：



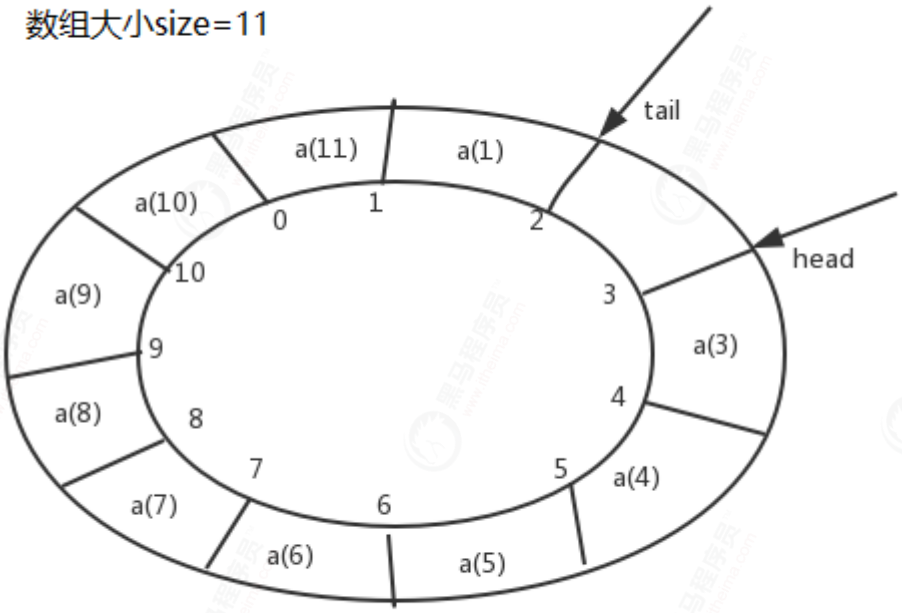
通过这样的方法，我们成功避免了数据搬移操作，也能重复利用已有的内存空间，看起来不难理解，但是循环队列的代码最关键的是如何确定好**队空和队满的判定条件。**

那如何来判定循环队列为空或者已满呢？

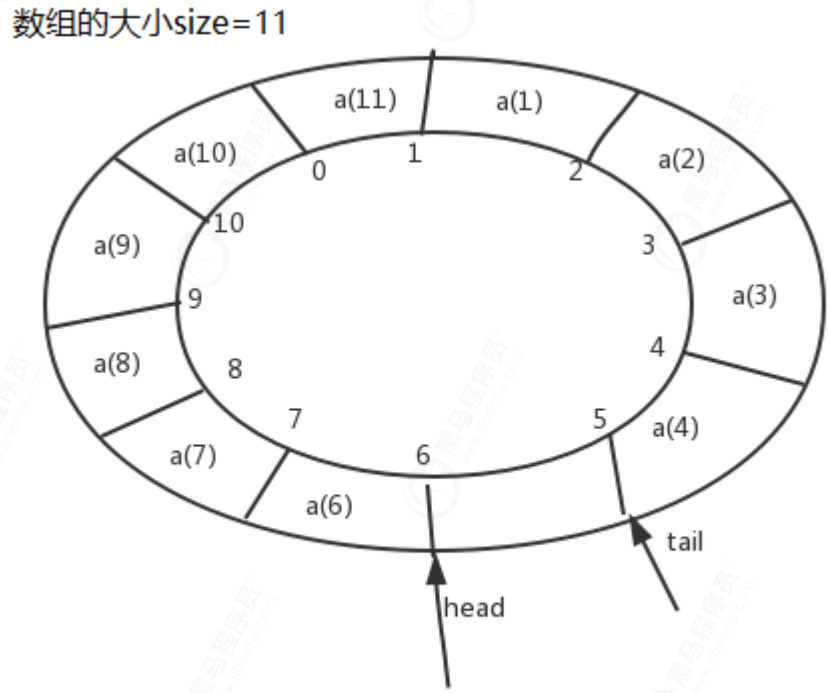
看到上面的图很多人立马想到说如果再向循环队列中存一个元素a(11)，将a(11)存入下标为0的位置，然后尾指针加1变成1，此时head=1,tail=1，所以立马决断出当满足head=tail时循环队列已满，这个结果真的对吗？

那我们在转换分析一下什么情况下队列为空？注意我们现在说的都是基于数组的循环队列，对比我们之前非循环的顺序队列判断为空的条件来看，如果是循环队列为空的条件仍然是head=tail，那此时就有冲突了，当head=tail时到底是队列为空还是队列已满？

因此我们关于队列已满的判断条件并不正确，我们也不认为当把元素a(11)存入之后队列就满了，反而我们认为上面图中所画情况就是队列已满的情况，如果你还是不甚明白，我们在接着画几个队列已满的情况



或者



我们把这种情况下的循环队列称之为队列已满，我们发现当队列满时，图中的 tail 指向的位置实际上是没有存储数据的，所以：

为了避免“队列为空”和“队列为满”的判别条件冲突，我们有意浪费了一个位置。

- 判别队列为空的条件是：head== tail;;
- 判别队列为满的条件是：(tail+ 1) % capacity == head;。可以这样理解，当 tail 循环到数组的前面，要从后面追上 front，还差一格的时候，判定队列为满，其中capacity 为数组的大小

```
class MyCircularQueue {
    int capacity;
    //内容数组
    int[] elementData;
    //头指针
    int front;
    //尾指针
    int rear;


    /** Initialize your data structure here. Set the size of the queue to be k. */
    public MyCircularQueue(int k) {
        this.capacity = k+1;
        elementData = new int[k+1];
        front = rear = 0;
    }



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

```
successful. */
    public boolean enQueue(int value) {
        if (isFull()) {
            return false;
        }
        elementData[rear] = value;
        rear = (rear+1) % capacity;
        return true;
    }

    /** Delete an element from the circular queue. Return true if the operation is
successful. */
    public boolean deQueue() {
        if (isEmpty()) {
            return false;
        }
        front = (front+1) % capacity;
        return true;
    }

    /** Get the front item from the queue. */
    public int Front() {
        if (isEmpty()) {
            return -1;
        }
        return elementData[front];
    }

    /** Get the last item from the queue. */
    public int Rear() {
        if (isEmpty()) {
            return -1;
        }
        return elementData[(rear+capacity-1)%capacity];
    }

    /** Checks whether the circular queue is empty or not. */
    public boolean isEmpty() {
        return front == rear;
    }

    /** Checks whether the circular queue is full or not. */
    public boolean isFull() {
        return front == (rear+1) % capacity;
    }
}
```

3.2、641. 设计循环双端队列

https://leetcode-cn.com/problems/design-circular-deque/

与622是同类题目

```
class MyCircularDeque {
    //定义数组容量
    int capacity;
    //定义数组
    int[] elementData;
    //定义front
    int front;
    //定义rear
    int rear;

    /** Initialize your data structure here. Set the size of the deque to be k. */
    public MyCircularDeque(int k) {
        this.capacity = k+1;
        elementData = new int[capacity];
        front = rear = 0;
    }

    /** Adds an item at the front of Deque. Return true if the operation is successful. */
    public boolean insertFront(int value) {
        if (isFull()) {
            return false;
        }
        front = (front-1+capacity) % capacity;
        elementData[front] = value;
        return true;
    }

    /** Adds an item at the rear of Deque. Return true if the operation is successful. */
    public boolean insertLast(int value) {
        if (isFull()) {
            return false;
        }
        elementData[rear] = value;
        rear = (rear+1) % capacity;
        return true;
    }

    /** Deletes an item from the front of Deque. Return true if the operation is successful.
*/
    public boolean deleteFront() {
        if (isEmpty()) {
            return false;
        }
        front = (front+1) % capacity;
        return true;
    }

    /** Deletes an item from the rear of Deque. Return true if the operation is successful.
*/
    public boolean deleteLast() {
        if (isEmpty()) {
```

```
            return false;
        }
        rear = (rear-1+capacity) % capacity;
        return true;
    }

    /** Get the front item from the deque. */
    public int getFront() {
        if (isEmpty()) {
            return -1;
        }
        return elementData[front];
    }

    /** Get the last item from the deque. */
    public int getRear() {
        if (isEmpty()) {
            return -1;
        }
        return elementData[(rear-1+capacity)%capacity];
    }

    /** Checks whether the circular deque is empty or not. */
    public boolean isEmpty() {
        return front == rear;
    }

    /** Checks whether the circular deque is full or not. */
    public boolean isFull() {
        return front == (rear+1) % capacity;
    }
}
```

3.3、703. 数据流中的第K大元素

https://leetcode-cn.com/problems/kth-largest-element-in-a-stream/

采用java内置的PriorityQueue实现

```
class KthLargest {

    int k;
    //队列头是最小值,出队列的元素是队列中的最小值，也是队列中第k大的元素
    PriorityQueue<Integer> queue; //此时队列元素的优先级由自然数的大小关系定义

    public KthLargest(int k, int[] nums) {
        this.k = k;
        queue = new PriorityQueue(k);
        for (int num: nums) {
            add(num);
        }
```

```
    }

    public int add(int val) {
        if (queue.size() < k) {
            queue.offer(val);
        }else if (val > queue.peek()) {
            queue.poll();
            queue.offer(val);
        }
        return queue.peek();
    }
}
```