门徒计划——队列

1.队列的基础知识

队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。

类似于我们日常生活中排队买票，我们购票的时候是要排在队尾的——入队操作，当到我们买票的时候我们已经在队首了，买完票离开就是一个出队的操作。总结来说，队列就是一个先进先出的线性表。

2.链表的复习

分隔链表

思路：

1.这道题类似于快排，找到一个中间值，比它大的放后面，比它小的放后面，但是不同的是，分割链表的相对位置要保持不变

2.创建两个链表，一个存储比x小的元素，另一个是比x大的元素

3.为两个链表定义两个指针

4.定义原链表的头指针，然后进行比较，连接到对应的链表，然后进行移动

5.将两个链表拼接到了一起

```
/*
 *
 * [86] 分隔链表
 */

// @lc code=start
```

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} x
 * @return {ListNode}
 */
var partition = function (head, x) {
    if (!head) return null;
    let h1 = new ListNode(), h2 = new ListNode(), p1 = h1, p2 = h2;
    for (let p = head, q; p; p = q) {
        q = p.next;
        p.next = null;
        if (p.val < x) {
            [p1.next, p1] = [p, p];
        } else {
            [p2.next, p2] = [p, p];
        }
    }
    p1.next = h2.next;
    return h1.next;
};
```

复制带随机指针的链表

思路：

1.创建两个指针，一个指向头指针

2.遍历整个链表，复制每个节点，并将值复制成一样的，然后拼接到原节点的后面

3.找到一个克隆节点，然后进行修正random,并将克隆节点的random指向克隆节点

4.拆分链表，分成原节点链表和克隆节点链表

5.返回我们的克隆节点链表

```
/*
 *
 * [138] 复制带随机指针的链表
 */

// @lc code=start
/**
 * // Definition for a Node.
 * function Node(val, next, random) {
 *    this.val = val;
 *    this.next = next;
 *    this.random = random;
 * };
 */

/**
 * @param {Node} head
 * @return {Node}
 */
var copyRandomList = function (head) {
    if (!head) return null;
    let p = head, q;
    while (p) {
        q = new ListNode(p.val);
        q.random = p.random;
        q.next = p.next;
        p.next = q;
        p = q.next;
    }
    p = head.next;
    while (p) {
        p.random && (p.random = p.random.next);
        (p = p.next) && (p = p.next);
    }
    p = q = head.next;
    while (q.next) {
        head.next = head.next.next;
        q.next = q.next.next;
        head = head.next;
        q = q.next;
    }
    head.next = null;
    return p;
```

```
};
```

3.队列的封装与使用

设计循环队列

思路：

1.首先创建一个容量为k的数组，用来存储数据

2.定义变量：capacity 表示队列最大容量,front 表示队首的索引,rear 表示队尾的索引,count 表示当前队列长度

3.capacity 初始值为 k,front 初始值为 0,rear 初始值为 0,count 初始值为 0，count == 0表示链表为空

4.添加一个元素时，向queue[rear]位置插入元素,然后队尾索引向后移动rear = (rear + 1) % capacity，并且count++

5.添加一个元素时，rear = (rear + 1) % capacity，并且count++

6.删除一个元素时，front向后移动即可，front = (front + 1) % capacity，并且count--

7.添加一个元素时，rear = (rear + 1) % capacity，并且count++

8.获取最后一个元素 (rear - 1 + capacity) % capacity,注意：为了循环到队尾所以加上 capacity

9.添加一个元素时，rear = (rear + 1) % capacity，并且count++

10.当capacity和count相等时，说明队列已满

```
/*
 *
 * [622] 设计循环队列
 */

// @lc code=start
/**
 * @param {number} k
 */
```

```
var MyCircularQueue = function (k) {
    this.front = 0;
    this.rear = 0;
    this.max = k;
    this.queue = Array(k + 1);
};

/**
 * @param {number} value
 * @return {boolean}
 */
MyCircularQueue.prototype.enQueue = function (value) {
    if (this.isFull()) return false;
    this.queue[this.rear] = value;
    this.rear = (this.rear + 1) % (this.max + 1);
    return true;
};

/**
 * @return {boolean}
 */
MyCircularQueue.prototype.deQueue = function () {
    if (this.isEmpty()) return false;
    this.front = (this.front + 1) % (this.max + 1);
    return true;
};

/**
 * @return {number}
 */
MyCircularQueue.prototype.Front = function () {
    if (this.isEmpty()) return -1;
    return this.queue[this.front];
};

/**
 * @return {number}
 */
MyCircularQueue.prototype.Rear = function () {
    if (this.isEmpty()) return -1;
    return this.queue[(this.rear + this.max) % (this.max + 1)];
};
```

```
/**
 * @return {boolean}
 */
MyCircularQueue.prototype.isEmpty = function () {
    return ((this.rear - this.front + this.max + 1) % (this.max +
1)) == 0;
};

/**
 * @return {boolean}
 */
MyCircularQueue.prototype.isFull = function () {
    return ((this.rear - this.front + this.max + 1) % (this.max +
1)) == this.max;
};
```

设计循环双端队列

思路：

1.首先创建一个容量为k的数组，用来存储数据

2.定义变量：capacity 表示队列最大容量;front 表示队首的索引;rear 表示队尾的索引，它指向下一个从队尾入队元素的位置;count 表示当前队列长度;

3.capacity 初始值为 k;front 初始值为 0;rear 初始值为 0;count 初始值为 0，count == 0表示链表为空;

4.从队尾添加一个元素时，向queue[rear]位置插入元素,然后队尾索引向后移动rear = (rear + 1) % capacity，并且count++

5.从队尾添加一个元素时，先添加元素，然后rear后移，rear = (rear + 1) % capacity，并且count++

6.从队尾添加一个元素时，先添加元素，然后rear后移，rear = (rear + 1) % capacity，并且count++

7.从队尾删除一个元素时，rear向前移动，rear = (rear − 1 + capacity) % capacity，并且count++

8.从队尾添加一个元素时，先添加元素，然后rear后移，rear = (rear + 1) % capacity，并且count++

9.从队首删除一个元素时，front向后移动即可，front = (front + 1) % capacity，并且count--

10.从队首添加一个元素时，front先向前移动一位，front = (front − 1 + capacity) % capacity再添加元素，并且count++

11.从队尾添加一个元素时，先添加元素，然后rear后移，rear = (rear + 1) % capacity，并且count++

12.当capacity和count相等时，说明队列已满

```
/*
 *
 * [641] 设计循环双端队列
 */

// @lc code=start
/**
 * Initialize your data structure here. Set the size of the deque
to be k.
 * @param {number} k
 */
var MyCircularDeque = function (k) {
    this.front = 0;
    this.rear = 0;
    this.max = k;
    this.deque = Array(k + 1);
};

/**
 * Adds an item at the front of Deque. Return true if the operation
is successful.
 * @param {number} value
 * @return {boolean}
 */
MyCircularDeque.prototype.insertFront = function (value) {
    if (this.isFull()) return false;
    this.front = (this.front + this.max) % (this.max + 1);
    this.deque[this.front] = value;
    return true;
};

/**
```

```
 * Adds an item at the rear of Deque. Return true if the operation
is successful.
 * @param {number} value
 * @return {boolean}
 */
MyCircularDeque.prototype.insertLast = function (value) {
    if (this.isFull()) return false;
    this.deque[this.rear] = value;
    this.rear = (this.rear + 1) % (this.max + 1);
    return true;
};

/**
 * Deletes an item from the front of Deque. Return true if the
operation is successful.
 * @return {boolean}
 */
MyCircularDeque.prototype.deleteFront = function () {
    if (this.isEmpty()) return false;
    this.front = (this.front + 1) % (this.max + 1);
    return true;
};

/**
 * Deletes an item from the rear of Deque. Return true if the
operation is successful.
 * @return {boolean}
 */
MyCircularDeque.prototype.deleteLast = function () {
    if (this.isEmpty()) return false;
    this.rear = (this.rear + this.max) % (this.max + 1);
    return true;
};

/**
 * Get the front item from the deque.
 * @return {number}
 */
MyCircularDeque.prototype.getFront = function () {
    if (this.isEmpty()) return -1;
    return this.deque[this.front];
};
```

```
/**
 * Get the last item from the deque.
 * @return {number}
 */
MyCircularDeque.prototype.getRear = function () {
    if (this.isEmpty()) return -1;
    return this.deque[(this.rear + this.max) % (this.max + 1)];
};

/**
 * Checks whether the circular deque is empty or not.
 * @return {boolean}
 */
MyCircularDeque.prototype.isEmpty = function () {
    return ((this.rear - this.front + this.max + 1) % (this.max +
1)) == 0;
};

/**
 * Checks whether the circular deque is full or not.
 * @return {boolean}
 */
MyCircularDeque.prototype.isFull = function () {
    return ((this.rear - this.front + this.max + 1) % (this.max +
1)) == this.max;
};
```

设计前中后队列

思路：

1.这道题的整体思路是，对一个队列/数组，在第一位新增 / 删除一个新的元素,在最后一位新增 / 删除一个元素，在最中间新增 / 删除一个元素

2.首先举例，我们有【1，2，3，4，5，7】这样的一个队列，我们整体的思路是，命名两个队列，进行增添操作后，最中间的位置在右队列里面（这个不强制，随自己的意愿），然后进行删除和新增操作

3.第一个要求，在队列的最前面新增一位，举例，将元素6 加到最前面。用两个队列，将队列分为左右两个队列，左队列 leftArray【1，2，3】用黄括号括起来，右队列 rightArray【4，5，7】用绿括号括起来。左队列使用方法unshift,将元素6添加到队列的第一位，这个时候，左队列的长度 > 右队列的长度，但是我们发现，左队列【6，1，2，3】 右队列【4，

5，7】中间元素3在左队列里面;接下来，左队列使用方法pop 删除最后一位，右队列使用方法unshift 将元素3添加到第一位；就是这样的效果左队列【6，1，2】右队列【3，4，5，7】这就是在队列的最前面添加元素的效果【6，1，2，3，4，5，7】

4.第二个要求，在队列的最中间新增一位，举例将 6 添加到队列的最中间。用两个队列，左队列 leftArray【1，2，3】用黄括号括起来，右队列rightArray【4，5，7】用绿色括号,首先，判断如果左队列的长度 = 右队列的长度,将左队列的最后一位删除，用到了pop;然后，将元素3，添加到右队列的第一位，用到了unshift。然后就是这种效果：左队列【1，2】右队列【3，4，5，7】然后，将元素6 添加到左队列的最后一位，用到了方法push。就是这样的效果左队列【1，2，6】右队列【3，4，5，7】最后，在队列的正中间添加元素的效果就是【1，2，6，3，4，5，7】

5.第三个要求，在队列的最后面新增一位，举例将 6 添加到队列的最后面。用两个队列，左队列 leftArray【1，2，3】用黄括号括起来，右队列rightArray【4，5，7】用绿色括号括起来。首先，我们使用push 在右队列的最后一位新增元素6。这时候，效果是：左队列【1，2，3】右队列【4，5，7，6】我们发现，最中间的数在右队列里面，并且左队列的长度< 右队列的长度，符合。这便是在队列最后面新增一位的效果：【1，2，3，4，5，7，6】

6.第四个要求，将 最前面 的元素从队列中删除并返回值，如果删除之前队列为空，那么返回 -1；举例还是这个【1，2，3，4，5，7】队列，删掉元素1；用两个队列，左队列 leftArray【1，2，3】用黄色括号括起来，右队列 rightArray【4，5，7】用绿括号括起来。首先，判断左队列是否为空，为空返回-1；否则，使用方法shift删除队列最前面的元素。这便是删除后的效果：【2，3，4，5，7】
这里注意一下：如果删除第一位后，左队列的长度 > 右队列的长度，举例：左队列【9，8，2，3】右队列【4，5，7】这时候，左队列使用pop删除最后一位,将删除的元素，新增到右队列的第一位；

7.第五个要求，将 正中间 的元素从队列中删除并返回值，如果删除之前队列为空，那么返回 -1；举例还是这个【1，2，3，4，5，7】队列，删掉元素3；用两个队列，左队列 leftArray【1，2，3】用黄色括号括起来 右队列 rightArray【4，5，7】用绿括号括起来。首先，判断左队列是否为空，为空返回-1；否则，在左队列的最后一位删除，用到了pop，此时队列变成：左队列【1，2】右队列【4，5，7】然后判断，左队列的长度 < 右队列的长度；如果符合就不用继续操作。如果不符合，举例：左队列【8，1，2，9】右队列【4，5，7】左队列的长度> =右队列的长度,这时候我们还是将左队列的最后一位删除，将它加到右队列的第一位。便是这种效果：左队列【8，1，2】右队列【9，4，5，7】

8.第六个要求，将 最后面 的元素从队列中删除并返回值，如果删除之前队列为空，那么返回 -1；举例还是这个【1，2，3，4，5，7】队列，删掉元素7；用左右两个队列，黄色括号是左队列【1，2，3】，绿色括号是右队列【4，5，7】首先，判断右队列是否为空，为空返回-1；否则，我们使用方法pop来删除右队列的最后一位。此时，最后一位被删除，但是需要注意的地方是，左队列的长度 > 右队列的长度。我们要让，左队列的最后一位用方法

pop删除，添加到右队列的第一位。此时，便是删除队列的最后一位的效果【1，2，3，4，5】

自行封装：

```
/*
 *
 * [1670] 设计前中后队列
 */

// @lc code=start
var Node = function (val) {
    this.val = val;
    this.pre = null;
    this.next = null;
}
Node.prototype.insert_pre = function (p) {
    p.next = this;
    p.pre = this.pre;
    this.pre && (this.pre.next = p);
    this.pre = p;
    return;
}
Node.prototype.insert_next = function (p) {
    p.pre = this;
    p.next = this.next;
    this.next && (this.next.pre = p);
    this.next = p;
    return;
}
Node.prototype.erase = function () {
    this.next && (this.next.pre = this.pre);
    this.pre && (this.pre.next = this.next);
    return;
}
var deQueue = function () {
    this.cnt = 0;
    this.head = new Node(-1);
    this.tail = new Node(-1);
    this.head.next = this.tail;
    this.tail.pre = this.head;
    console.log(this.head)
}
```

```
deQueue.prototype.push_back = function (value) {
    this.tail.insert_pre(new Node(value))
    this.cnt += 1;
    return;
}
deQueue.prototype.push_front = function (value) {
    this.head.insert_next(new Node(value));
    this.cnt += 1;
}
deQueue.prototype.pop_back = function () {
    let ret = this.tail.pre.val;
    if (this.cnt) {
        this.tail.pre.erase();
        this.cnt -= 1;
    }
    return ret;
}
deQueue.prototype.pop_front = function () {
    let ret = this.head.next.val;
    if (this.cnt) {
        this.head.next.erase();
        this.cnt -= 1;
    }
    return ret;
}
deQueue.prototype.front = function () {
    return this.head.next.val;
}
deQueue.prototype.back = function () {
    return this.tail.pre.val;
}
deQueue.prototype.size = function () {
    return this.cnt;
}
var FrontMiddleBackQueue = function () {
    this.q1 = new deQueue();
    this.q2 = new deQueue();
};

FrontMiddleBackQueue.prototype.maintain = function () {
    if (this.q2.size() > this.q1.size()) {
        this.q1.push_back(this.q2.pop_front());
    } else if (this.q1.size() == this.q2.size() + 2) {
```

```
        this.q2.push_front(this.q1.pop_back());
    }
    return;
}
/**
 * @param {number} val
 * @return {void}
 */
FrontMiddleBackQueue.prototype.pushFront = function (val) {
    let ret = this.q1.push_front(val);
    this.maintain();
    return;
};

/**
 * @param {number} val
 * @return {void}
 */
FrontMiddleBackQueue.prototype.pushMiddle = function (val) {
    if (this.q1.size() == this.q2.size() + 1) {
        this.q2.push_front(this.q1.pop_back())
    }
    this.q1.push_back(val);
    this.maintain();
    return;
};

/**
 * @param {number} val
 * @return {void}
 */
FrontMiddleBackQueue.prototype.pushBack = function (val) {
    this.q2.push_back(val);
    this.maintain();
    return;
};

/**
 * @return {number}
 */
FrontMiddleBackQueue.prototype.popFront = function () {
    let ret = this.q1.pop_front();
    this.maintain();
```

```
    return ret;
};

/**
 * @return {number}
 */
FrontMiddleBackQueue.prototype.popMiddle = function () {
    let ret = this.q1.pop_back();
    this.maintain();
    return ret;
};

/**
 * @return {number}
 */
FrontMiddleBackQueue.prototype.popBack = function () {
    let ret = this.q2.size() ? this.q2.pop_back() :
this.q1.pop_back();
    this.maintain();
    return ret;
};

/**
 * Your FrontMiddleBackQueue object will be instantiated and called
as such:
 * var obj = new FrontMiddleBackQueue()
 * obj.pushFront(val)
 * obj.pushMiddle(val)
 * obj.pushBack(val)
 * var param_4 = obj.popFront()
 * var param_5 = obj.popMiddle()
 * var param_6 = obj.popBack()
 */
```

数组实现：

```
var FrontMiddleBackQueue = function () {
    this.leftArray = [];
    this.rightArray = [];
};

/**
```

```
 * @param {number} val
 * @return {void}
 */
FrontMiddleBackQueue.prototype.pushFront = function (val) {
    this.leftArray.unshift(val);
    if (this.leftArray.length > this.rightArray.length)
this.rightArray.unshift(this.leftArray.pop());
};

/**
 * @param {number} val
 * @return {void}
 */
FrontMiddleBackQueue.prototype.pushMiddle = function (val) {
    if (this.leftArray.length == this.rightArray.length)
this.rightArray.unshift(val);
    else this.leftArray.push(val);
};

/**
 * @param {number} val
 * @return {void}
 */
FrontMiddleBackQueue.prototype.pushBack = function (val) {
    this.rightArray.push(val);
    if (this.leftArray.length + 1 < this.rightArray.length)
this.leftArray.push(this.rightArray.shift());
};

/**
 * @return {number}
 */
FrontMiddleBackQueue.prototype.popFront = function () {
    if (!this.rightArray.length) return -1;
    if (!this.leftArray.length) return this.rightArray.shift();
    let a = this.leftArray.shift();
    if (this.rightArray.length > this.leftArray.length + 1)
this.leftArray.push(this.rightArray.shift());
    return a;
};

/**
 * @return {number}
```

```
 */
FrontMiddleBackQueue.prototype.popMiddle = function () {
    if (!this.rightArray.length) return -1;
    if (this.rightArray.length == this.leftArray.length) return
this.leftArray.pop();
    return this.rightArray.shift();
};

/**
 * @return {number}
 */
FrontMiddleBackQueue.prototype.popBack = function () {
    if (!this.rightArray.length) return -1;
    let a = this.rightArray.pop();
    if (this.leftArray.length > this.rightArray.length)
this.rightArray.unshift(this.leftArray.pop());
    return a;
};

/**
 * Your FrontMiddleBackQueue object will be instantiated and called
as such:
 * var obj = new FrontMiddleBackQueue()
 * obj.pushFront(val)
 * obj.pushMiddle(val)
 * obj.pushBack(val)
 * var param_4 = obj.popFront()
 * var param_5 = obj.popMiddle()
 * var param_6 = obj.popBack()
 */
```

最近请求次数

4.智力发散题

第K个数

亲密字符串

柠檬水找零

煎饼排序

任务调度器