第3章　栈与队列

- 栈
- **队列**
- 栈的应用
  - 递归到非递归的转换

队列的定义

- **先进先出（First In First Out）**
  - 限制访问点的线性表
    - 按照到达的顺序来释放元素
    - 所有的插入在表的一端进行，所有的删除都在表的另一端进行
- **主要元素**
  - 队头（front）
  - 队尾（rear）

队列的主要操作

- 入队列（enQueue）
- 出队列（deQueue）
- 取队首元素（getFront）
- 判断队列是否为空（isEmpty）

队列的抽象数据类型

```
template <class T> class Queue {
public:                  // 队列的运算集
    void clear();        // 变为空队列
    bool enQueue(const T item);
                         // 将item插入队尾，成功则返回真，否则返回假
    bool deQueue(T & item) ;
                         // 返回队头元素并将其从队列中删除，成功则返回真
    bool getFront(T & item);
                         // 返回队头元素，但不删除，成功则返回真
    bool isEmpty(); // 返回真，若队列已空
    bool isFull();  // 返回真，若队列已满
};
```

队列的实现方式

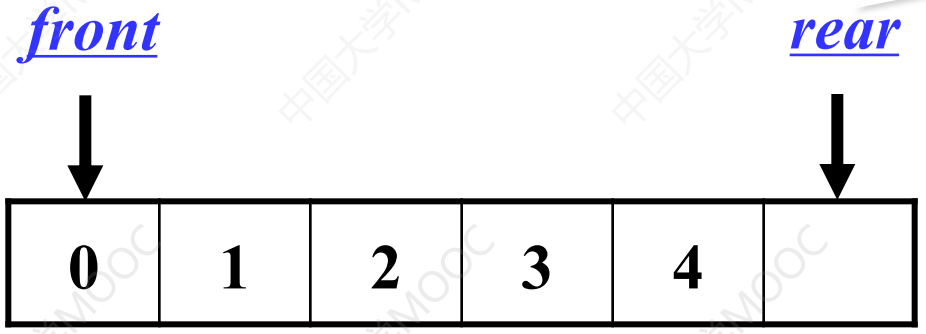
- **顺序队列**
  - **关键**是如何防止假溢出

- **链式队列**
  - 用单链表方式存储，队列中每个元素对于链表中的一个结点

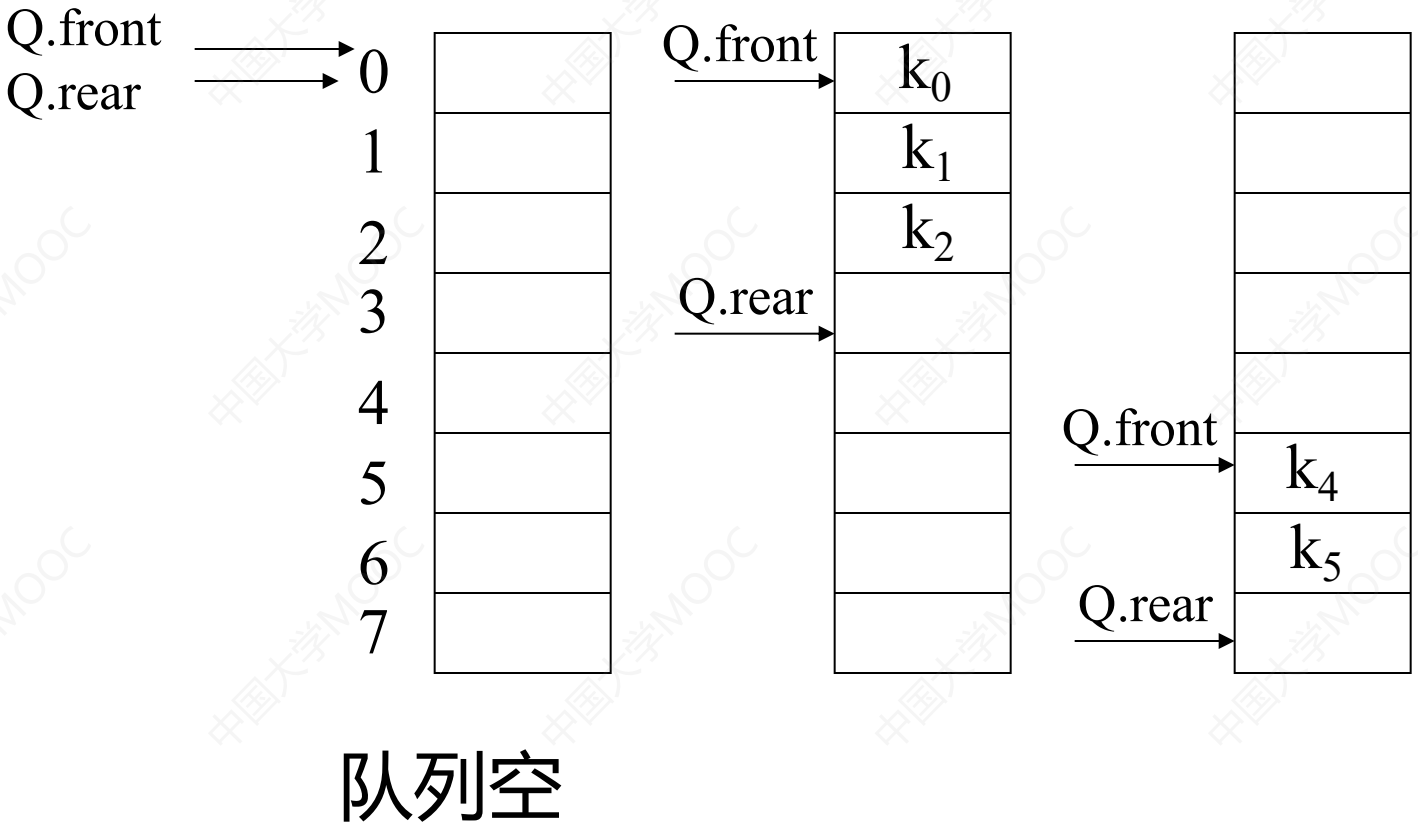
顺序队列

- 用向量存储队列元素，用两个变量分别指向队列的前端(front)和尾端(rear)
  - *front*：指向当前待出队的元素位置（地址）
  - *rear*：指向当前待入队的元素位置（地址）

当前队尾元素的直接后继“空”位置



顺序队列



队列空

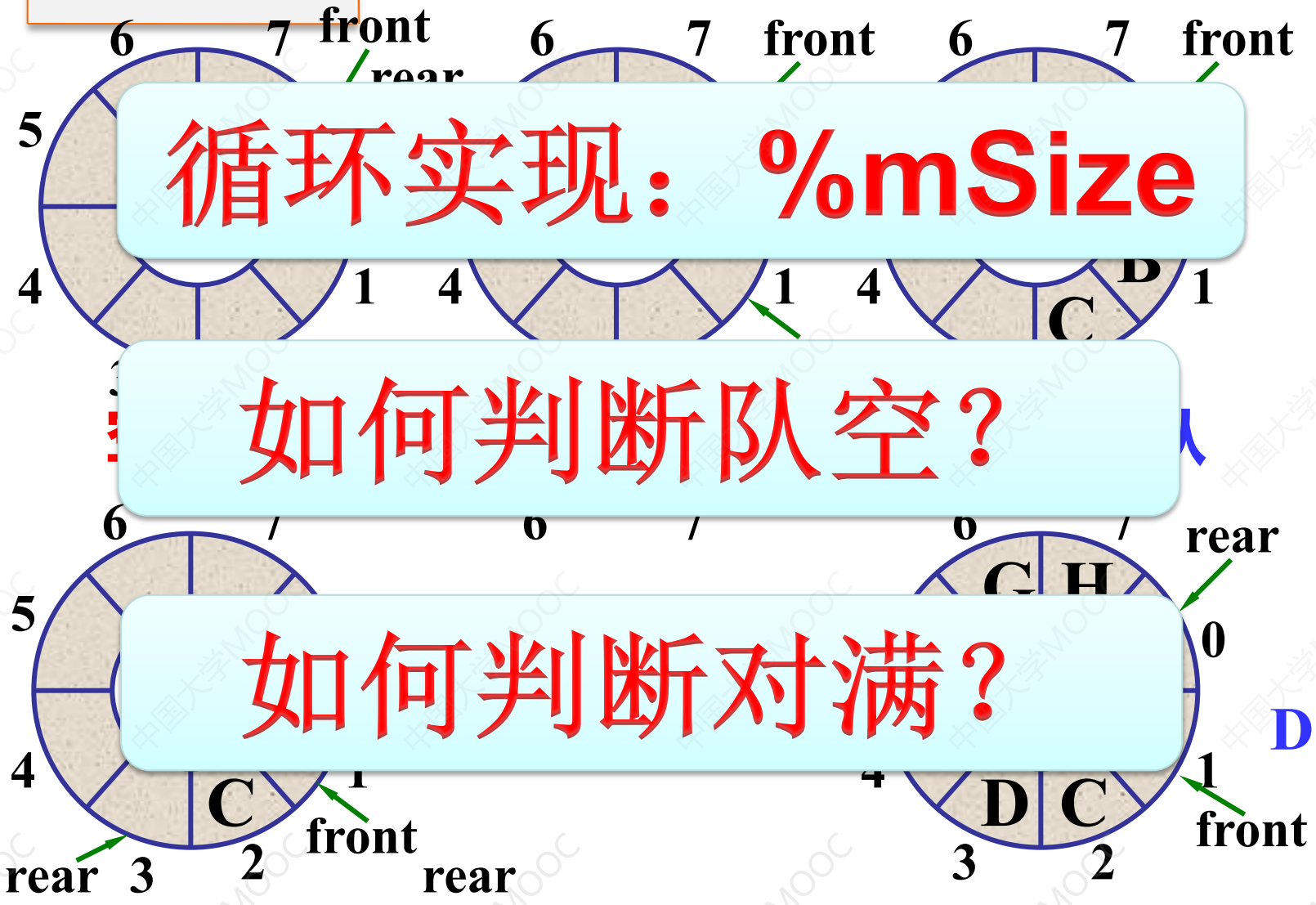
再进队一个元素如何？

顺序队列

队列的溢出

- 上溢
  - 当队列满时，再做进队操作，所出现的现象
- 下溢
  - 当队列空时，再做删除操作，所出现的现象
- 假溢出
  - 当 rear = mSize-1 时，再作插入运算就会产生溢出，如果这时队列的前端还有许多空位置，这种现象称为假溢出

循环队列运行示意图



队列的类定义

```
class arrQueue: public Queue<T> {
    private:
      int       mSize;                    // 存放队列的数组的大小
      int       front;                    // 表示队头所在位置的下标
      int       rear;                     // 表示待入队元素所在位置的下标
      T         *qu;                      // 存放类型为T的队列元素的数组
    public:                                // 队列的运算集
      arrQueue(int size) {          // 创建队列的实例
        mSize = size +1;             // 浪费一个存储空间，以区别队列空和队列满
        qu = new T [mSize];
        front = rear = 0;
      }
      ~arrQueue() {                 // 消除该实例，并释放其空间
        delete [] qu;
      }
}
```

入队列的操作

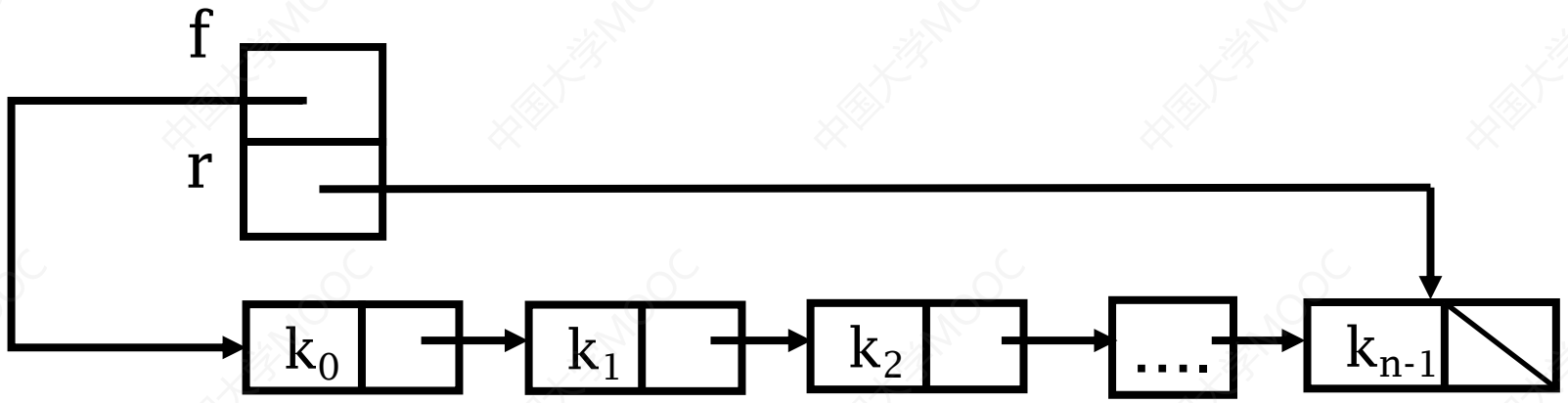
```
bool arrQueue<T> :: enQueue(const T item)  {
                                        //  item入队，插入队尾
    if (((rear + 1 ) % mSize) == front) {
        cout << "队列已满，溢出" << endl;
        return false;
    }
    qu[rear] = item;
    rear = (rear +1) % mSize;           // 循环后继
    return true;
}
```

出队列的操作

```
bool arrQueue<T> :: deQueue(T& item)  {
                                    // 返回队头元素并从队列中删除
    if ( front == rear)  {
          cout << "队列为空" << endl;
          return false;
    }
    item = qu[front];
    front = (front +1) % mSize;
    return true;
}
```

链式队列的表示

- 单链表队列
- 链接指针的方向是从队列的前端向尾端链接



链式队列的类定义

```
template <class T>
class lnkQueue: public Queue<T> {
private:
    int size;                    // 队列中当前元素的个数
    Link<T>* front;              // 表示队头的指针
    Link<T>* rear;               // 表示队尾的指针
public:                          // 队列的运算集
    lnkQueue(int size);          // 创建队列的实例
    ~lnkQueue();                 // 消除该实例，并释放其空间
}
```

链式队列代码实现

```
bool enQueue(const T item) { //  item入队，插入队尾
    if (rear == NULL) {       // 空队列
        front = rear = new Link<T> (item, NULL);
    }
    else  {                   // 添加新的元素
        rear-> next = new Link<T> (item, NULL);
        rear = rear ->next;
    }
    size++;
    return true;
}
```

链式队列代码实现

```
bool deQueue(T* item)  {              // 返回队头元素并从队列中删除
    Link<T> *tmp;
    if (size == 0)  {                 // 队列为空，没有元素可出队
        cout << "队列为空" << endl;
        return false;
    }
    *item = front->data;
    tmp = front;
    front = front -> next;
    delete tmp;
    if (front == NULL)
        rear = NULL;
    size--;
    return true;
}
```

顺序队列与链式队列的比较

- **顺序队列**
  - 固定的存储空间
- **链式队列**
  - 可以满足大小无法估计的情况

都不允许访问队列内部元素