# 概述

所谓的队列是一种先入先出（FIFO，First In First Out）的数学模型。

# 分类

## 静态队列

静态队列对于队列的长度有一个限制，如果内存不能继续分配时，则队尾不能继续向后增加。

静态队列有两种处理方式：

1. 队头出去以后后面的元素依次向前移动一个位置，这样元素移动多处理效率低；
2. 队列元素不移动，只移动队头指针，这样比较浪费空间资源。

注：**静态队列即用数组实现的，链式队列是用链表实现的**。

**静态队列通常都是循环队列（否则空间不足）**。为了弥补普通队列的缺点，引入了环形队列。

环形/循环队列的排队是有顺时针和逆时针之说的。

在元素出队后，队头指针是不断向后移动的，队尾也是可以向后移动的，这样处理速度快，而且也可以充分利用空间。

## 循环队列

解决假溢出的办法就是如果后面满了，就再从头开始，也就是头尾相接的循环。

### 定义

**循环队列它的容量是固定的**，并且它的队头和队尾指针都可以随着元素入出队列而发生改变，这样循环队列逻辑上就好像是一个环形存储空间。

注：在实际内存当中，不可能有真正的环形存储区，我们只是用顺序表模拟出来的逻辑上的循环。

循环队列只需要改变front和rear指针，也就是让front或rear指针不断加1，即使超出了地址范围，也会自动从头开始，我们可以采取取模运算处理：

(rear+1)%QueueSize

(front+1)%QueueSize

取模就是取余数的意思，他取到的值永远不会大于除数。

#define MAXSIIZE 100

typedef struct

{

ElemType \*base; //用于存放内存分配基地址,也可用数组存放

int front;

int rear;

}

### 参数

循环队列需要2个参数front和rear确定，2个参数不同场合有不同意义。

1. 队列初始化

front和rear的值都是零

1. 队列非空

front代表的是队列的第一个元素

rear代表的是队列的最后一个有效元素的下一个元素

1. 队列空

front和rear的值相等，但不一定是零

### 初始化

initQueue(cycleQueue \*q)

{

q->base = (ElemType \*)malloc(MAXSIZE \* sizeof(ElemType));

if( !q->base )

exit(0);

q->front = q->next = 0;

}

### 入队

入队主要两步完成：

1. 将值存入rear代表的位置
2. 增加长度

错误的写法：q->rear = q->rear+1;

正确的写法：q->rear = (q->rear+1) % MAXSIZE;

代码：

InsertQueue(cycleQueue \*q, ElemType e)

{

if( (q->rear+1)%MAXSIZE == q->front ) //队列满

return;

q->base[q->rear] = e;

q->rear = (q->rear+1) % MAXSIZE;

}

### 出队

循环队列出队的步骤：q->front = (q->front+1) %数组长度。

代码：

DeleteQueue(cycleQueue \*q, ElemType \*e)

{

if( q->front == q->rear ) //队列空

return;

\*e = q->base[q->front];

q->front = (q->front+1) % MAXSIZE;

}

### 队列判空

如果front和rear的值相等，则该队列就一定为空。

### 队列已满

判断循环队列是否已满的方法：

1. 增加一个标识符参数（一般不用该方法）；
2. 少用一个元素（通常使用这种方法）

**如果front和rear的值紧挨着，则表示循环队列已满**

C语言伪算法：

if((q->rear+1)%数组长度 == q->front)

{

已满

}

else

{

不满

}

## 链式队列

## 映射队列

参考：

映射队列：<http://mysql.taobao.org/monthly/2020/09/08/>

## 优先队列

优先队列即堆。

## 单调队列

单调队列是一种特殊的队列数据结构，其主要特点是队列中的元素保持特定的单调性（递增或递减）。这种队列通常用于解决一些特定类型的问题，比如在滑动窗口中寻找最大值或最小值等。

单调队列通常支持以下两种操作：

入队操作：将元素加入队尾，但要维持队列的单调性，可能会删除队尾元素以满足单调性要求。

出队操作：从队头移除元素，但是仅当队头元素等于要出队的元素时才执行。

实现单调队列时，需要考虑保持队列的单调性。如果需要一个递增的单调队列，每次入队操作时，将队尾小于当前元素的元素出队，以保持队列递增的性质。相应地，如果需要一个递减的单调队列，每次入队操作时，将队尾大于当前元素的元素出队。

// 递增单调队列的入队操作

void insertIncreasing(deque<int>& dq, int x) {

// 删除队尾小于等于当前元素的元素，保持递增性质

while (!dq.empty() && dq.back() <= x) {

dq.pop\_back();

}

dq.push\_back(x); // 插入当前元素

}

// 单调队列的出队操作（出队首元素）

void dequeueOperation(deque<int>& dq, int x) {

if (!dq.empty() && dq.front() == x) {

dq.pop\_front(); // 如果队头元素等于要出队的元素，则出队

}

}

# 操作

## 出队

## 入队

## 存储

## 顺序存储

### 定义

typedef struct SeqQueue{

int data[MAXSIZE]; //也可以采用int \*data动态数组的形式存储

int front; // front头指针

int rear; // rear尾指针，若队列不为空，指向队列尾元素的下一个位置

}SeqQueue;

这里的定义是假设存储的是int类型数据，最理想的方式使用：

typedef int ElementType;

### 初始化

int SqQueueInit(SqQueue \*q){

q->front=0;

q->rear=0;

return 0;

}

### 销毁

void destroySeqQueue(SeqQueue \*q) {

// 如果data是动态分配的内存空间，释放它

// 假设data是静态数组，不需要释放内存

// 释放动态分配的内存空间

// 如果是静态数组，这一步可以忽略不做

// 如果是动态数组，需要释放内存

// 示例代码：

// free(q->data);

// 将队列指针置为NULL，表示队列已经被销毁

q = NULL;

}

### 队列长度

int SqQueueLength(SqQueue \*q){

return (q->rear-q->front+MAX\_SIZE)%MAX\_SIZE;

}

当计算队列长度时，使用 `(q->rear - q->front + MAX\_SIZE) % MAX\_SIZE` 这个表达式，其含义和计算方法如下：

1、Rear 和 Front 之间的距离：

- (q->rear - q->front): 这部分计算了队列中最后一个元素和第一个元素之间的距离。

2、加上数组长度：

- + MAX\_SIZE：这个步骤是为了确保即使 rear 已经经过循环回到数组的开头，也能得到一个非负的值。

3、取模运算：

- % MAX\_SIZE：这个运算确保了最后的结果在数组大小范围内。它消除了任何可能超出数组大小的情况，将结果限制在 0 到 MAX\_SIZE - 1 之间。

综合来说，这个计算方法保证了即使 rear 已经经过循环回到数组开头，也能正确计算出队列的长度。例如，如果 rear 在数组的末尾，而 front 在数组的开头，则 (rear - front) 会给出一个负数，加上 MAX\_SIZE 后变成一个非负数，最后再进行取模运算，确保了计算出的长度是准确的，并且处于合理的范围内（0 到 MAX\_SIZE - 1）。

注意：

上述方法使用了取模运算 (q->rear - q->front + MAX\_SIZE) % MAX\_SIZE，这是因为队列的循环性质。在循环队列中，当 rear 指针超过数组的末尾时，它会回到数组的开头。所以 (q->rear - q->front + MAX\_SIZE) % MAX\_SIZE 的计算可以确保即使 rear 被循环到数组开头，也能正确计算出队列的长度。

如下的方法：

int SqQueueLength(SqQueue \*q){

return (q->rear-q->front+1);

}

上述方法 return (q->rear - q->front + 1); 并未考虑到队列的循环性质。这种方式只计算了 rear 和 front 之间的距离，但没有处理队列循环的情况，可能导致错误的长度计算。如果 rear 已经循环到数组开头，而 front 还在数组中间，这种简单相减的方法就会得到错误的队列长度。

因此，第一个方法通过取模运算考虑了队列循环的情况，可以更准确地计算出循环队列的长度。

### 获取元素

int SeqQueueGetElement(SeqQueue \*q,int \*e){

\*e=q->data[q->front];

q->front--;

}

### 入队

思路：插入元素需要更新rear变量

代码：

int SqQueueEnQueue(SqQueue \*q, int \*element){

if((q->rear+1)%MAX\_SIZE == q->front){

return -1;

}

q->data[q->rear]=\*element;

q->rear=(q->rear+1)%MAX\_SIZE;

return 0;

}

### 出队

思路：删除元素需要判断是否为空（即front是都等于rear），然后更新front成员变量

代码：

int SqQueueDeQueue(SqQueue \*q, int \*element){

if(q->front == q->rear){

//队列为空（队列为空使用空余一个元素的方式，不使用flag）

return -1;

}

\*element = q->data[q->front];//将队头元素赋值给element

q->front = (q->front+1)%MAX\_SIZE;

//将front指针向后移动一个位置，如到最后则转到数组头部

// 这里没有直接将q->front++也是考虑了循环队列的情况

return 0;

}

## 链式存储

注：队列重点掌握链表存储。

### 定义

typedef struct LinkQueueNode{

int data; //存储节点的数据信息

struct LinkQueueNode \*next;

}LinkQueueNode, \*LinkQueuePtr;

注：上述是定义一个数据节点，下面才是真正定义数据结构特有属性。

typedef struct LinkQueue{

LinkQueuePtr front,rear;

}LinkQueue;

说明：LinkQueuePtr与LinkQueue的定义不同，LinkQueuePtr p表示指针，而LinkQueuePtr \*q才表示指针！

或者：

typedef struct Node {

int data; // 节点中存储的数据

struct Node \*next; // 指向下一个节点的指针

} Node;

typedef struct LinkedQueue {

Node \*front; // 指向队列头部的指针

Node \*rear; // 指向队列尾部的指针

} LinkedQueue;

### 初始化

initQueue(LinkQueue \*q)

{

q->front = q->rear = (LinkQueuePtr)malloc(sizeof(LinkQueueNode));

if( !q->front)

exit(0);

q->front->next = NULL; //头结点指针域设置为NULL

}

### 获取元素

思路：获取元素只能获取队尾和队首的

### 入队

思路：插入新元素，首先申请一个新节点，把新节点（此时还是孤立的节点）的成员变量（即data和next），然后更新原来节点rear与新节点的连接关系以及队列的rear。

代码：

int LinkQueueEnQueue(LinkQueue \*q, int \*element){

LinkQueuePtr p = (LinkQueuePtr)malloc(sizeof(LinkQueueNode));

if(NULL == p) exit(0);

p->data=\*element;

p->next=NULL;//新节点的下一个节点为NULL

//更新新节点的参数，接下来更新整个链表的成员变量，即front和rear

q->rear->next=p;

//操作队尾，原来的rear指向现在的新节点

q->rear=p;

//更新rear节点（前面是更新节点之间的指向关系）

return 0；

}

注：注意定义LinkQueuePtr p而不是LinkQueuePtr \*p，因为LinkQueuePtr本身就是指针类型。

或：

int LinkQueueEnQueue(LinkQueue \*q, int \*element) {

// 创建新节点

Node \*newNode = (Node \*)malloc(sizeof(Node));

if (newNode == NULL) {

// 内存分配失败

return 0; // 返回 0 表示入队失败

}

// 将数据存储到新节点中

newNode->data = \*element;

newNode->next = NULL; // 新节点将成为队列尾部，因此其next指针为NULL

if (q->rear == NULL) {

// 如果队列为空，新节点即是队列头部又是队列尾部

q->front = newNode;

q->rear = newNode;

} else {

// 否则将新节点连接到队列尾部，并更新rear指针

q->rear->next = newNode;

q->rear = newNode;

}

return 1; // 返回 1 表示入队成功

}

### 出队

int LinkQueueDeQueue(LinkQueue \*q, int \*element){

LinkQueuePtr p;//只需要声明变量而不需要实际分配内存空间

if(q->front==q->rear){

return -1;

}

//更新新节点的成员变量

p=q->front->next;//该节点成了最新的头结点

\*element = p->data;//头结点的数据

q->front->next=p->next;

if(q->rear==p){

q->rear=q->front;

}

free(p);

return 0;

}

或：

int LinkQueueDeQueue(LinkQueue \*q, int \*element) {

if (q->front == NULL) {

// 队列为空，无法执行出队操作

return 0; // 返回 0 表示出队失败

}

// 将头部节点数据传出

\*element = q->front->data;

Node \*temp = q->front;

q->front = q->front->next; // 移动front指针到下一个节点（这里不能是temp，后面会释放temp）

// 如果队列只有一个节点，出队后队列为空，需要更新rear指针为NULL

if (q->front == NULL) {

q->rear = NULL;

}

free(temp); // 释放出队节点的内存

return 1; // 返回 1 表示出队成功

}

### 销毁

DestroyQueue(LinkQueue \*q)

{

while(q->front){

q->rear = q->front->next;

free(q->front);

q->front = q->rear;

}

}

或：

void DestroyQueue(LinkQueue \*q) {

Node \*current = q->front;

while (current != NULL) {

Node \*temp = current;

current = current->next;

free(temp); // 释放节点内存

}

q->front = NULL;

q->rear = NULL;

}

# 实现

与栈实现类似，队列的实现方式包括三种：

1. 用简单的循环数组实现
2. 用动态循环数据实现
3. 用链表实现

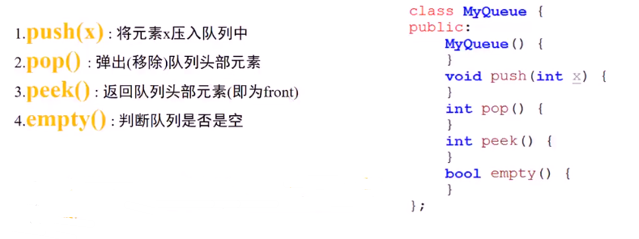
## 简单数组

## 动态数组

## 链表

## 栈

设计一个队列，支持基本的队列操作，这个队列的内部存储数据的结构为栈，栈的方法只能包括push、top、pop、size、empty等标准的栈方法。



注：Leetcode 232

**分析：**

**方法一：**





**方法二：**









**代码：**

**方法一：**



**方法二：**



**测试：**



# 应用

操作系统把优先级相等的任务按照到达的先后顺序安排在队列中

输入输出区接受键盘的输入就是按队列的形式输入和输出的

异步数据传输（文件IO、管道、socket）

用来决定每一位求助于客服中心的用户所需等待的时间

所有和时间有关的操作都有队列的影子

## 最近请求次数

**题目：**写一个 RecentCounter 类来计算最近的请求。

它只有一个方法：ping(int t)，其中 t 代表以毫秒为单位的某个时间。

返回从3000毫秒前到现在的 ping 数。

任何处于 [t-3000, t] 时间范围之内的 ping 都将会被计算在内，包括当前（指 t 时刻）的ping。

保证每次对ping的调用都使用比之前更大的t值。

**分析：**

**代码：**

注：Leetcode 933

## 设计双端循环队列

**题目：**设计实现双端循环队列。

你的实现需要支持以下操作：

MyCircularDeque(k)：构造函数,双端队列的大小为k。

insertFront()：将一个元素添加到双端队列头部。如果操作成功返回true。

insertLast()：将一个元素添加到双端队列尾部。如果操作成功返回true。

deleteFront()：从双端队列头部删除一个元素。如果操作成功返回true。

deleteLast()：从双端队列尾部删除一个元素。如果操作成功返回true。

getFront()：从双端队列头部获得一个元素。如果双端队列为空，返回-1。

getRear()：获得双端队列的最后一个元素。如果双端队列为空，返回-1。

isEmpty()：检查双端队列是否为空。

isFull()：检查双端队列是否满了。

示例：

MyCircularDeque circularDeque = new MycircularDeque(3); // 设置容量大小为3

circularDeque.insertLast(1); // 返回 true

circularDeque.insertLast(2); // 返回 true

circularDeque.insertFront(3); // 返回 true

circularDeque.insertFront(4); // 已经满了，返回 false

circularDeque.getRear(); // 返回 2

circularDeque.isFull(); // 返回 true

circularDeque.deleteLast(); // 返回 true

circularDeque.insertFront(4); // 返回 true

circularDeque.getFront(); // 返回 4

提示：

所有值的范围为[1, 1000]

操作次数的范围为[1, 1000]

请不要使用内置的双端队列库。

**分析：**

head是头部待出队位置（获取头元素，即直接访问arr[head]）

从头插入，head是逆时针旋转（需要先减1）

从头删除，head是顺时针旋转

tail是尾部待入队位置，（获取尾元素，需要先将tail逆时针旋转1）

从尾插入，tail是顺时针旋转

从尾删除，tail是逆时针旋转

**代码：**

class MyCircularDeque {

private:

int \*m\_data;

int m\_head;

int m\_tail;

int m\_size;

int m\_capacity;

public:

/\*\* Initialize your data structure here. Set the size of the deque to be k. \*/

MyCircularDeque(int k) {

m\_data = new int[k];

m\_head = m\_tail = 0;

m\_size = 0;

m\_capacity = k;

}

/\*\* Adds an item at the front of Deque. Return true if the operation is successful. \*/

bool insertFront(int value) {

if(isFull())

return false;

m\_head--;

if(m\_head < 0)

{

m\_head = m\_capacity -1;

}

m\_data[m\_head] = value;

m\_size++;

return true;

}

/\*\* Adds an item at the rear of Deque. Return true if the operation is successful. \*/

bool insertLast(int value) {

if(isFull())

return false;

m\_data[m\_tail] = value; //先赋值，然后m\_tail++

m\_tail++;

if(m\_tail == m\_capacity)

{

m\_tail = 0;

}

m\_size++;

return true;

}

/\*\* Deletes an item from the front of Deque. Return true if the operation is successful. \*/

bool deleteFront() {

if(isEmpty())

return false;

m\_head++;

if(m\_head == m\_capacity)

{

m\_head = 0;

}

m\_size--;

return true;

}

/\*\* Deletes an item from the rear of Deque. Return true if the operation is successful. \*/

bool deleteLast() {

if(isEmpty())

return false;

m\_tail--;

if(m\_tail < 0)

{

m\_tail = m\_capacity-1;

}

m\_size--;

return true;

}

/\*\* Get the front item from the deque. \*/

int getFront() {

if(isEmpty())

return -1;

return m\_data[m\_head];

}

/\*\* Get the last item from the deque. \*/

int getRear() {

if(isEmpty())

return -1;

int index = m\_tail - 1;

if(index < 0)

{

index = m\_capacity - 1;

}

return m\_data[index];

}

/\*\* Checks whether the circular deque is empty or not. \*/

bool isEmpty() {

return (0 == m\_size);

}

/\*\* Checks whether the circular deque is full or not. \*/

bool isFull() {

return (m\_size == m\_capacity);

}

};

/\*\*

\* Your MyCircularDeque object will be instantiated and called as such:

\* MyCircularDeque\* obj = new MyCircularDeque(k);

\* bool param\_1 = obj->insertFront(value);

\* bool param\_2 = obj->insertLast(value);

\* bool param\_3 = obj->deleteFront();

\* bool param\_4 = obj->deleteLast();

\* int param\_5 = obj->getFront();

\* int param\_6 = obj->getRear();

\* bool param\_7 = obj->isEmpty();

\* bool param\_8 = obj->isFull();

\*/

注：Leetcode 641

## 设计循环队列

**题目：**

你的实现应该支持如下操作：

MyCircularQueue(k): 构造器，设置队列长度为 k 。

Front: 从队首获取元素。如果队列为空，返回 -1 。

Rear: 获取队尾元素。如果队列为空，返回 -1 。

enQueue(value): 向循环队列插入一个元素。如果成功插入则返回真。

deQueue(): 从循环队列中删除一个元素。如果成功删除则返回真。

isEmpty(): 检查循环队列是否为空。

isFull(): 检查循环队列是否已满。

**分析：**

**代码：**

class MyCircularQueue {

public:

/\*\* Initialize your data structure here. Set the size of the queue to be k. \*/

MyCircularQueue(int k) {

m\_head = 0;

m\_tail = 0;

m\_capacity = k;

m\_size = 0;

m\_arr = new int[k];

}

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

bool enQueue(int value) {

if (isFull())

return false;

m\_arr[m\_tail] = value;

++m\_tail;

if (m\_tail == m\_capacity)

{

m\_tail = 0;

}

++m\_size;

return true;

}

/\*Delete an element from the circular queue. Return true if the operation is successful. \*/

bool deQueue() {

if (isEmpty()) **//出队判空，入堆判满**

return false;

++m\_head; //出队要先++m\_head后判断是否越界

if (m\_head == m\_capacity)

{

m\_head = 0;

}

--m\_size;

return true;

}

/\*\* Get the front item from the queue. \*/

int Front() {

if (isEmpty())

return -1;

return m\_arr[m\_head];

}

/\*\* Get the last item from the queue. \*/

int Rear() {

if (isEmpty())

return -1;

//注意：最后元素的索引是m\_tail的上一个元素

int index = m\_tail - 1;

if (index < 0)

{

index = m\_capacity - 1;

}

return m\_arr[index];

}

/\*\* Checks whether the circular queue is empty or not. \*/

bool isEmpty() {

return m\_size == 0;

}

/\*\* Checks whether the circular queue is full or not. \*/

bool isFull() {

return m\_size == m\_capacity;

}

int size() {

return m\_size;

}

private:

int\* m\_arr;

int m\_head; //待出队位置

int m\_tail; //待入队位置

int m\_size; //元素个数

int m\_capacity; //容量固定长度

};

另外一种写法：

class MyCircularQueue {

private:

int \*m\_data; //数据域

int m\_front; //指针域

int m\_rear;

int m\_size; //用于判空和溢出，这样比较方便

int m\_capacity;

public:

/\*\* Initialize your data structure here. Set the size of the queue to be k. \*/

MyCircularQueue(int k) {

m\_data = new int[k];

m\_front = m\_rear = 0;

m\_size = 0;

m\_capacity = k;

}

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

bool enQueue(int value) {

if(isFull())

return false;

m\_data[m\_rear] = value;

m\_rear++;

if(m\_rear == m\_capacity)

m\_rear = 0;

m\_size++;

return true;

}

/\*\* Delete an element from the circular queue. Return true if the operation is successful. \*/

bool deQueue() {

if(isEmpty())

return false;

m\_front++;

if(m\_front == m\_capacity)

m\_front = 0;

m\_size--;

return true;

}

/\*\* Get the front item from the queue. \*/

int Front() {

if(isEmpty())

return -1;

return m\_data[m\_front];

}

/\*\* Get the last item from the queue. \*/

int Rear() {

if(isEmpty())

return -1;

int index = m\_rear -1;

if(index < 0)

index = m\_capacity - 1;

return m\_data[index];

}

/\*\* Checks whether the circular queue is empty or not. \*/

bool isEmpty() {

return (0 == m\_size);

}

/\*\* Checks whether the circular queue is full or not. \*/

bool isFull() {

return (m\_size == m\_capacity);

}

};

/\*\*

\* 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();

\*/

注：Leetcode 622

## 用两个栈实现队列

题目要求：用两个栈实现一个队列。队列的声明如下，请实现它的两个函数appendTail和deleteHead，分别完成在队列尾部插入结点和队列头部删除结点的功能。

注：剑指offer P59

## 滑动窗口的最大值

**题目：**给定一个数组和滑动窗口的大小，请找出所有滑动窗口里的最大值。例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小3，那么一共存在6个滑动窗口，它们的最大值分别为{4,4,6,6,6,5}。

**分析：**

**代码：**

注：剑指offer P290

## 第k个数

## 和至少为 K 的最短子数组

注：Leetcode 862