# 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；

1. 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;

}

### 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

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

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;

}

## 链式队列

# 操作

## 出队

## 入队

# 实现

## 顺序存储

### 定义

typedef struct SeqQueue{

int data[MAXSIZE];

int front,rear;

}SeqQueue;

### 初始化

int SqQueueInit(SqQueue \*q){

q->front=0;

q->rear=0;

return 0;

}

### 销毁

### 队列长度

int SqQueueLength(SqQueue \*q){

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

}

### 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

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

return 0;

}

## 链式存储

### 定义

typedef struct LinkQueueNode{

int data;

struct LinkQueueNode \*next;

}LinkQueueNode, \*LinkQueuePtr;

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！

### 初始化

initQueue(LinkQueue \*q)

{

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

if( !q->front)

exit(0);

q->front=>next = NULL;

}

### 销毁

### 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));

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；

}

### 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;

}

# 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

# 应用

直接应用：

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

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

间接应用：