

# 第三章 栈和队列

2022-04-29 0:09

- 栈和队列的定义和特点
- 案例引入
- 栈的表示和操作的实现
- 栈与递归
- 队列的表示和操作的实现
- 案例分析与实现

## 一、栈和队列的定义和特点

### 1、总括

- 栈和队列是两种常用的、重要的数据结构。
- 栈和队列是限定插入和删除只能在表的“端点”进行的线性表。
  - > 栈和队列是线性表的子集（是插入和删除位置受限的线性表）

线性表	栈	队列
Insert(L, i, x) $1 \leq i \leq n+1$	Insert(S, n+1, x)	Insert(Q, n+1, x)
Delete(L, i) $1 \leq i \leq n$	Delete(S, n)	Delete(Q, 1)
	➡	➡

栈的应用：

由于栈的操作具有后进先出的固有特性，使得栈成为程序设计中的有用工具。另外，如果问题求解的过程具有“后进先出”的天然特性的话，则求解的算法中也必然需要利用栈。

数制转换	表达式求值
括号匹配的检验	八皇后问题
行编辑程序	函数调用
迷宫求解	递归调用的实现

队列的应用：

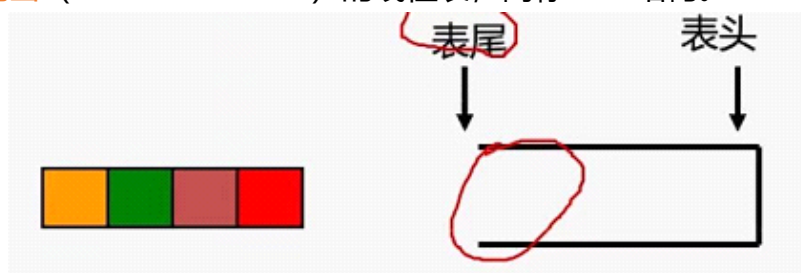
由于队列的操作具有先进先出的特性，使得队列成为程序设计中解决类似排队问题的有用工具。

- 脱机打印输出：按申请的先后顺序依次输出
- 多用户系统中，多个用户排成队，分时地循环使用CPU和主存
- 按用户的优先级排成多个队，每个优先级一个队列
- 时事控制系统中，信号按接收的先后顺序依次处理

网络电文传输，按到达的时间先后顺序依次进行

## 2、栈的定义和特点

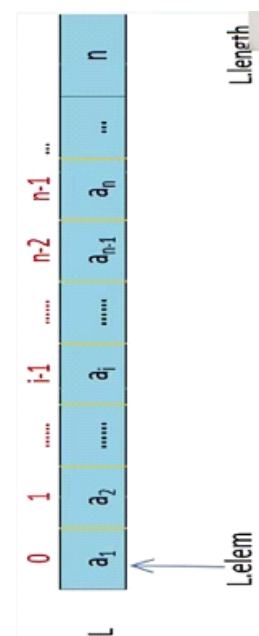
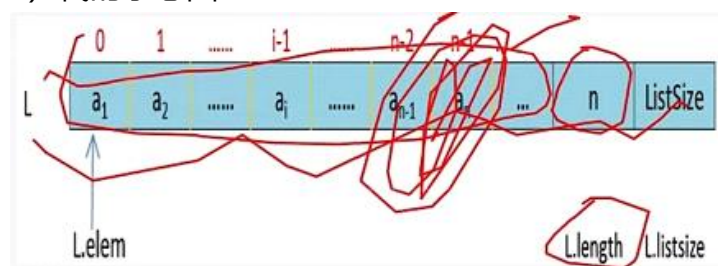
**栈** (stack) 是一个特殊的线性表，是限定仅在一端（通常是表尾）进行插入和删除操作的线性表。又称为**后进先出** (Last In First Out) 的线性表，简称**LIFO**结构。



### 1) 栈的相关概念:

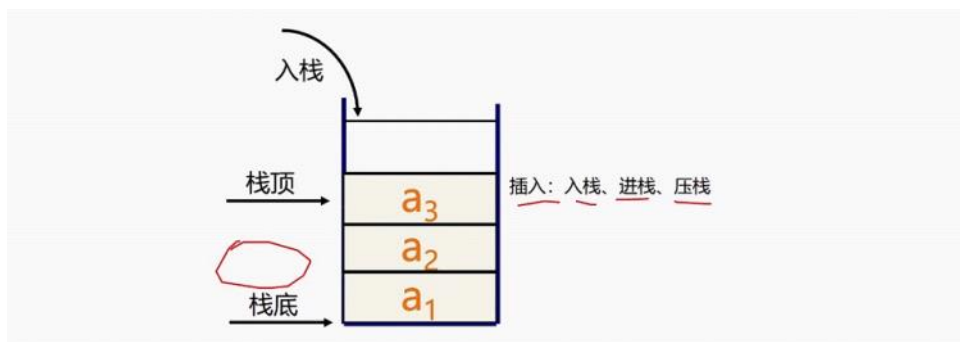
**栈** 是仅在表尾进行插入、删除操作的线性表。  
表尾(即 $a_n$ 端)称为**栈顶** Top; 表头(即 $a_1$ 端)称为**栈底** Base  
例如: 栈  $s = (a_1, a_2, a_3, \dots, a_{n-1}, a_n)$   
 $a_1$ 称为栈底元素  $a_n$ 称为栈顶元素  
插入元素到**栈顶**(即表尾)的操作, 称为**入栈**。  
从**栈顶**(即表尾)删除最后一个元素的操作, 称为**出栈**。  
“入” = 压入 = **PUSH** (x) “出” = 弹出 = **POP** (y)

### 2) 栈的示意图:

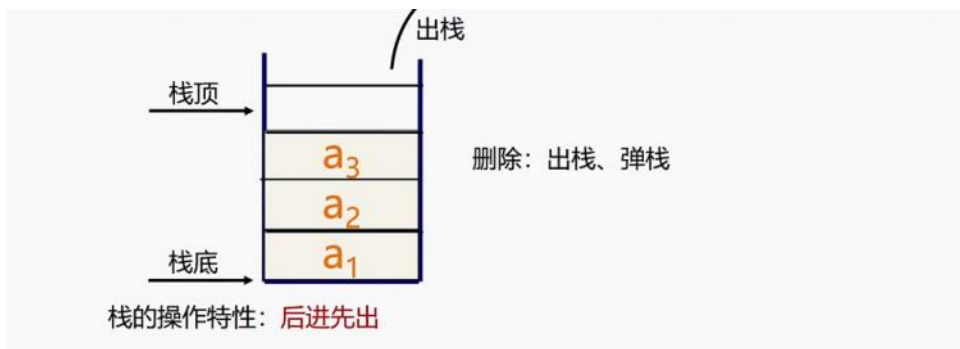


### 3) 入栈与出栈:

入栈的操作示图:

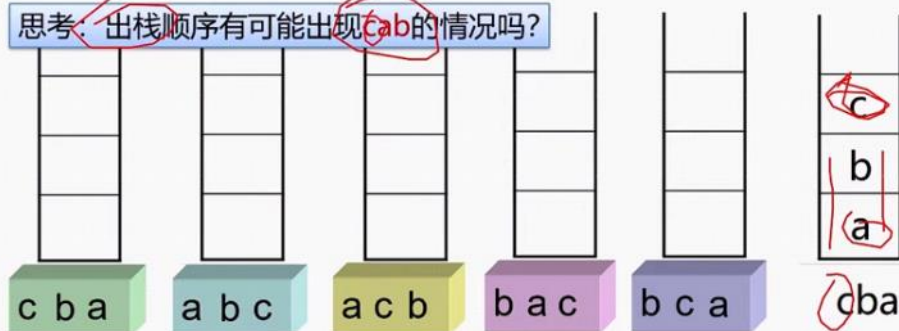


出栈的操作示意图：



【思考】假设有3个元素a, b, c, 入栈顺序是a, b, c, 则它们的出栈顺序有几种可能？

思考：出栈顺序有可能出现cab的情况吗？



#### 4) 栈的相关概念：

1. 定义 限定只能在表的一端进行插入和删除运算的线性表（只能在栈顶操作）
2. 逻辑结构 与同线性表相同，仍为一对一关系。
3. 存储结构 用顺序栈或链栈存储均可，但以顺序栈更常见
4. 运算规则 只能在栈顶运算，且访问结点时依照后进先出（LIFO）的原则。
5. 实现方式 关键是编写入栈和出栈函数，具体实现依顺序栈或链栈的不同而不同。

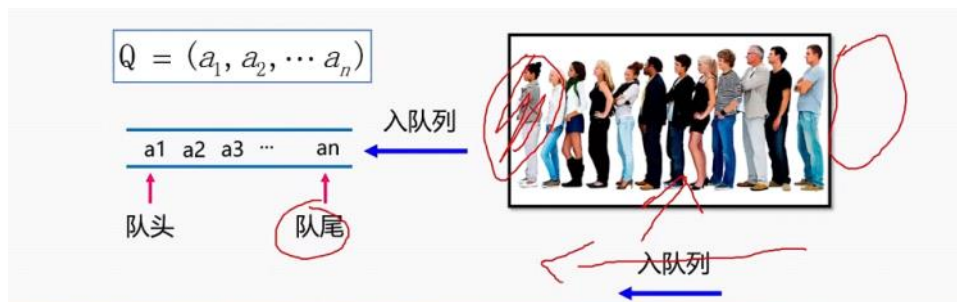
#### 5) 栈与一般线性表有什么不同

栈与一般线性表的区别在于：运算规则不同

<p>一般线性表</p> <p>逻辑结构：一对一</p> <p>存储结构：顺序表、链表</p> <p>运算规则：随机存取</p>	<p>栈</p> <p>逻辑结构：一对一</p> <p>存储结构：顺序栈、链栈</p> <p>运算规则：后进先出(LIFO)</p>
--	--

### 3、队列的定义和特点

队列是一种先进先出（First In First Out---FIFO）的线性表。在表一端插入（表尾），在表的另一端（表头）删除



#### 1) 队列的相关概念

1.定义	只能在表的一端进行插入运算，在表的另一端进行删除运算的线性表（头删尾插）
2.逻辑结构	与同线性表相同，仍为一对一关系。
3.存储结构	顺序队或链队，以循环顺序队列更常见。
4.运算规则	只能在队首和队尾运算，且访问结点时依照先进先出（FIFO）的原则。
5.实现方式	关键是掌握入队和出队操作，具体实现依顺序队或链队的不同而不同。

## 二、案例引入

栈：

- 案例3.1：进制转换
- 案例3.2：括号匹配的检验
- 案例3.3：表达式求值

队列：

- 案例3.4：舞伴问题

### 1、案例3.1—进制转换

>十进制整数N向其它进制数d（二、八、十六）的转换是计算机实现计算的基本问题。





### 3、案例3.3—表达式求值

表达式求值是程序设计语言编程中一个最基本的问题，它的实现也需要运用栈。

这里介绍的算法是由运算符优先级确定运算顺序的对表达式求值算法——**算法优先算法**

#### ▪ 表达式的组成

- **操作数(operand)**：常数、变量。
- **运算符(operator)**：算术运算符、关系运算符和逻辑运算符。
- **界限符(delimiter)**：左右括弧和表达式结束符。

- 任何一个**算术表达式**都由**操作数**（常数、变量）、**算术运算符**（+、-、\*、/）和**界限符**（括号、表达式结束符 '#'、虚设的表达式起始符 '#'）组成。后两者统称为**算符**。

- 例如：# 3 \* ( 7 - 2 ) #

### 4、案例3.4—舞伴问题

- 假设在舞会上，男士和女士各自排成一队。舞会开始时，依次从男队和女队的队头各出一人配成舞伴。如果两队初始人数不相同，则较长的那一队中未配对者等待下一轮舞曲。现要求写一算法模拟上述舞伴配对问题。



- 显然，**先入队的男士或女士先出队配成舞伴**。因此该问题具有典型的先进先出特性，可以用队列作为算法的数据结构。

- 首先构造两个队列
- 依次将**队头**元素出队配成舞伴
- 某队为空，则另外一队等待着则是下一舞曲第一个可获得舞伴的人。

QM

a1 a2 a3 ... am

QF

b1 b2 b3 ... bm bm+1 bn

## 三、栈的表示和操作实现

### 1、栈的抽象数据类型的类型定义

ADT Stack {

**数据对象：**

$D = \{ a_i \mid a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0 \}$

**数据关系：**

$R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \}$

约定an 端为栈顶，a1 端为栈底。

**基本操作：**初始化、进栈、出栈、取栈顶元素等

} ADT Stack

操作总括：

**InitStack(&S)** 初始化操作

操作结果：构造一个空栈  $S$ 。

**DestroyStack(&S)** 销毁栈操作

初始条件：栈  $S$  已存在。

操作结果：栈  $S$  被销毁。

**StackEmpty(S)** 判定  $S$  是否为空栈

初始条件：栈  $S$  已存在。

操作结果：若栈  $S$  为空栈，则返回 **TRUE**，  
否则 **FALSE**。

**StackLength(S)** 求栈的长度

初始条件：栈  $S$  已存在。

操作结果：返回  $S$  的元素个数，即栈的长度。

**GetTop(S, &e)** 取栈顶元素

初始条件：栈  $S$  已存在且非空。

操作结果：用  $e$  返回  $S$  的栈顶元素。

**ClearStack(&S)** 栈置空操作

初始条件：栈  $S$  已存在。

操作结果：将  $S$  清为空栈。

**Push(&S, e)** 入栈操作

初始条件：栈  $S$  已存在。

操作结果：插入元素  $e$  为新的栈顶元素。

**Pop(&S, &e)** 出栈操作

初始条件：栈  $S$  已存在且非空。

操作结果：删除  $S$  的栈顶元素  $a_n$ ，并用  $e$  返回  
其值。

由于栈本身就是线性表，于是栈也有顺序存储和链式存储两种实现方式。

- 栈的顺序存储---顺序栈
- 栈的链式存储---链栈

### 1) 顺序栈

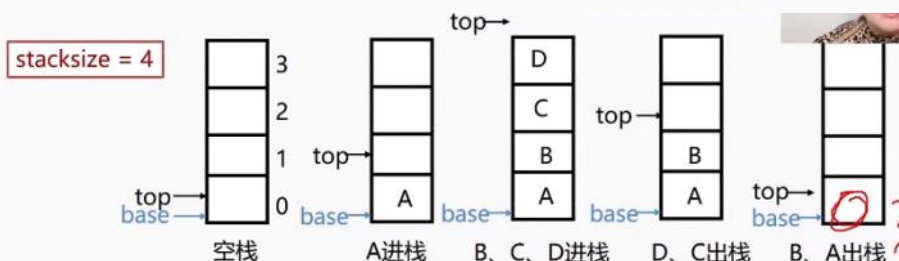
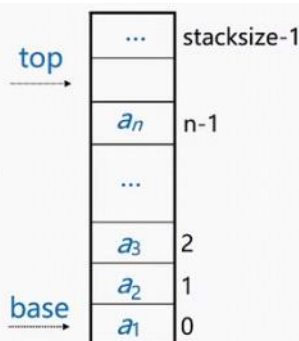
存储方式与一般线性表的顺序存储结构完全相同。

利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素。栈底一般在低地址端。

- 附设 **top** 指针，指示栈顶元素在顺序栈中的位置。
- 另设 **base** 指针，指示栈底元素在顺序栈中的位置。

但是，为了方便操作，通常 **top** 指示真正的  
栈顶元素之上的下标地址

- 另外，用 **stacksize** 表示栈可使用的最大容量



空栈:  $base == top$  是栈空标志

栈满:  $top - base == stacksize$

栈满时的处理方法：

- 1、报错，返回操作系统。
- 2、分配更大的空间，作为栈的存储空间，将原栈的内容移入新栈。

特点：使用数组作为顺序栈存储方式的特点：

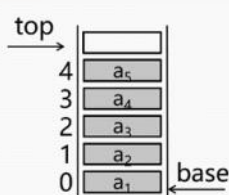
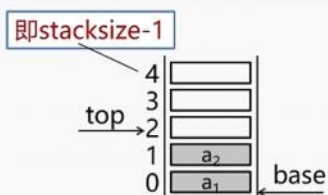
简单、方便、但易产生溢出（数组大小固定）

- 上溢(overflow): 栈已经满，又要压入元素
- 下溢(underflow): 栈已经空，还要弹出元素

注：上溢是一种错误，使问题的处理无法进行；而下溢一般认为是一种结束条件，即问题处理结束。

顺序栈的表示:

```
#define MAXSIZE 100
typedef struct{
    SElemType *base; //栈底指针
    SElemType *top; //栈顶指针
    int stacksize; //栈可用最大容量
}SqStack;
```

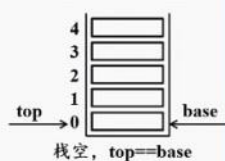


### 【算法一】顺序栈的初始化



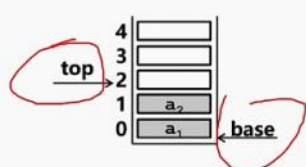
```
Status InitStack(SqStack &S){ // 构造一个空栈
    S.base = new SElemType[MAXSIZE]; //或
    S.base = (SElemType*)malloc(MAXSIZE*sizeof(SElemType));
    if (!S.base) exit (OVERFLOW); // 存储分配失败
    S.top = S.base; //栈顶指针等于栈底指针
    S.stacksize = MAXSIZE;
    return OK;
}
```

### 【算法二】顺序栈判断栈是否为空



```
Status StackEmpty(SqStack S){
    // 若栈为空, 返回TRUE; 否则返回FALSE
    if (S.top == S.base)
        return TRUE;
    else
        return FALSE;
}
```

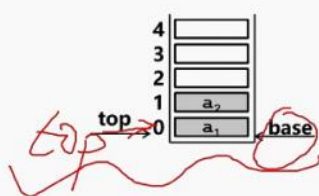
### 【算法三】求顺序栈的长度



```
int StackLength( SqStack S )
{
    return S.top - S.base;
}
```



## 【算法四】清空顺序栈

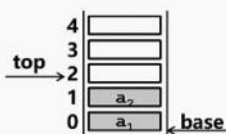


```

Status ClearStack( SqStack S ) {
    if( S.base ) S.top = S.base;
    return OK;
}

```

## 【算法五】销毁顺序栈

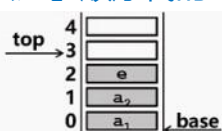


```

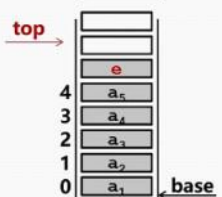
Status DestroyStack( SqStack &S ) {
    if( S.base ) {
        delete S.base ;
        S.stacksize = 0;
        S.base = S.top = NULL;
    }
    return OK;
}

```

## 【算法六】顺序栈的入栈



- (1)判断是否栈满，若满则出错（上溢）
- (2)元素e压入栈顶
- (3)栈顶指针加1



栈满,  $(top - base) == 5$

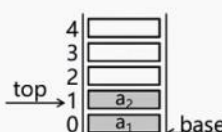
```

Status Push( SqStack &S, SElemType e ) {
    if( S.top - S.base == S.stacksize ) // 栈满
        return ERROR;
    *S.top++ = e;
    return OK;
}

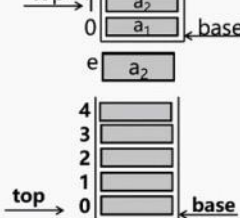
```

\*S.top=e;  
S.top++;

## 【算法七】顺序栈的出栈



- (1)判断是否栈空，若空则出错（下溢）
- (2)获取栈顶元素e
- (3)栈顶指针减1



栈空,  $(top == base)$

```

Status Pop( SqStack &S, SElemType &e ) {
    // 若栈不空, 则删除S的栈顶元素, 用e返回其值, 并返回OK;
    // 否则返回ERROR
    if( S.top == S.base ) // 等价于 if( StackEmpty(S) )
        return ERROR;
    e = *S.top--;
    return OK;
}

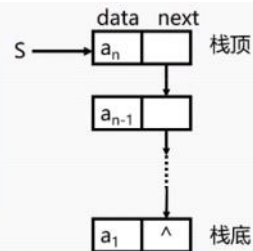
```

--S.top;  
e=\*S.top;

## 2) 链栈的表示

链栈是运算受限的单链表，只能在链表头部进行操作

```
typedef struct StackNode{
    SElemType data;
    struct StackNode *next;
} StackNode, *LinkStack;
LinkStack S;
```



- 链表的头指针就是栈顶
- 不需要头结点
- 基本不存在栈满的情况
- 空栈相当于头指针指向空
- 插入和删除仅在栈顶处执行

注意：链栈中指针的方向

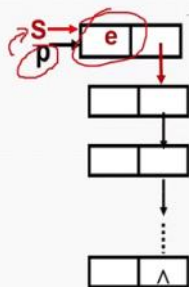
## 【算法八】链栈的初始化

```
void InitStack(LinkStack &S) {
    //构造一个空栈，栈顶指针置为空
    S=NULL;
    return OK;
}
```

## 【算法九】判断链栈是否为空

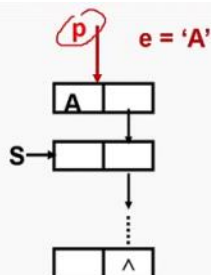
```
Status StackEmpty(LinkStack S)
    if (S==NULL) return TRUE;
    else return FALSE;
}
```

## 【算法十】链栈的入栈



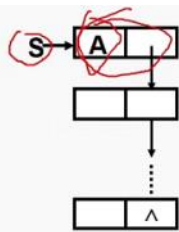
```
Status Push(LinkStack &S, SElemType e){
    p=new StackNode //生成新结点p
    p->data=e; //将新结点数据域置为e
    p->next=S; //将新结点插入栈顶
    S=p; //修改栈顶指针
    return OK;
}
```

## 【算法十一】链栈的出栈



```
Status Pop (LinkStack &S,SElemType &e){
    if (S==NULL) return ERROR;
    e = S-> data;
    p = S;
    S = S-> next;
    delete p;
    return OK;
}
```

## 【算法十二】取栈顶元素



```
SElemType GetTop(LinkStack S) {
    if (S!=NULL)
        return S->data;
}
```

## 四、栈与递归

### >递归的定义

若一个对象部分地包含它自己，或用它自己给自己定义则称这个对象是递归的；

若一个过程直接或间接地调用自己，则称这个过程是递归的过程。

▪ 例如：递归求n的阶乘

```
long Fact ( long n ) {
    if ( n == 0 ) return 1;
    else return n * Fact (n-1);
}
```

常见使用递归的情况：

### 1、递归定义的数学函数

• 阶乘函数:

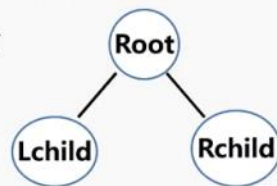
$$Fact(n) = \begin{cases} 1 & \text{若 } n = 0 \\ n \cdot Fact(n-1) & \text{若 } n > 0 \end{cases}$$

• 2阶Fibonacci数列:

$$Fib(n) = \begin{cases} 1 & \text{若 } n = 1 \text{ 或 } 2 \\ Fib(n-1) + Fib(n-2) & \text{其它} \end{cases}$$

### 2、具有递归特性的数据结构

• 二叉树

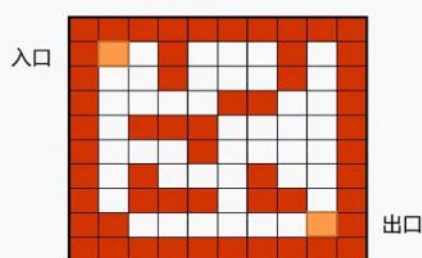


• 广义表

$$A = (a, A)$$

### 3、可递归求解的问题

• 迷宫问题



Hanoi塔问题



## 递归问题——用分治法求解

> **分治法**：对于一个较为复杂的问题，能够分解成几个相对简单的且解法相同或类似的子问题来求解。

必备三个条件：

- ①能够将一个问题转变成一个新问题，而新问题与原问题的解法相同或类同，不同的仅是处理的对象，且这些处理对象是变化有规律的。
- ②可以通过上述转化而使问题简化
- ③必须有一个明确的递归出口，或称递归的边界。

## 分治法求解递归问题算法的一般形式

```
void p (参数表) {  
    if (递归结束条件) 可直接求解步骤; -----基本项  
    else p (较小的参数); -----归纳项  
}
```

例如：

```
long Fact ( long n ) {  
    if ( n == 0) return 1;    //基本项  
    else return n * Fact (n-1); //归纳项  
}
```

函数调用过程：

>函数调用前，系统完成

- (1) 将实参，返回地址等传递给被调用函数
- (2) 为被调用函数的局部变量分配存储区
- (3) 将控制转移到被调用函数的入口

>函数调用后，系统完成

- (1) 保存被调用函数的计算结果
- (2) 释放被调用函数的数据区
- (3) 依照被调用函数保存的返回地址将控制转移到调用函数

当多个函数构成嵌套调用时：



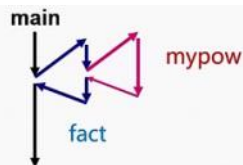
```

int main(void) {
    .....
    y = fact(3);
    .....
}

double fact(int n) {
    .....
    z = mypow(3.5, 2);
    .....
}

double mypow(double x, in n) {
    .....
}

```



遵循后调用的先返回

栈

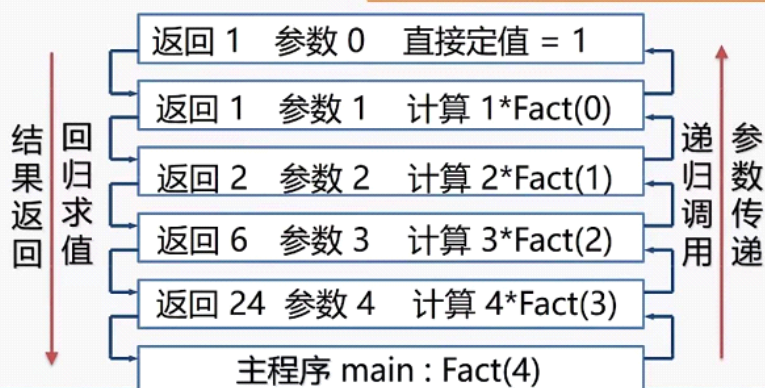
求解阶乘  $n!$  的过程

主程序 main : Fact(4)

```

if ( n == 0 ) return 1;
else return n * Fact (n-1);

```



递归函数调用的实现

"层次"

主函数

0层

第1次调用 1层

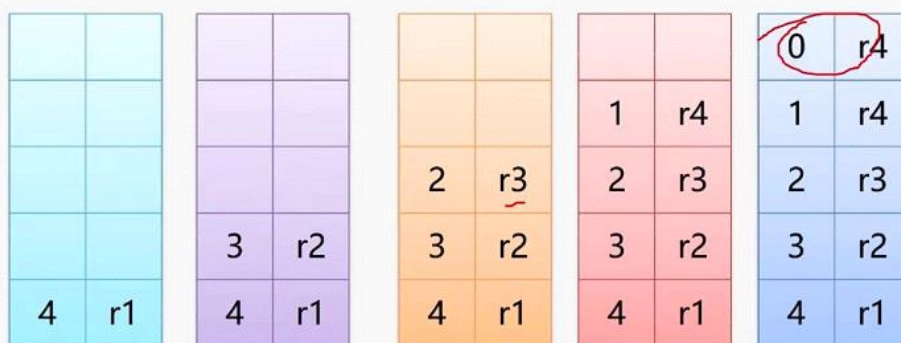
第  $i$  次调用  $i$  层

"递归工作栈" —— 递归程序运行期间使用的数据存储区

"工作记录"

实参, 局部变量, 返回地址

进行 fact(4) 调用的系统栈的变化状态



递归的优缺点

优点: 结构清晰, 程序易读

缺点: 每次调用要生成工作记录, 保存状态信息, 入栈; 返回时要出栈,


恢复状态信息。时间开销大。

解决方法：尾递归、单向递归—>循环结构

自用栈模拟系统的运行时栈

>尾递归—>循环结构

```
long Fact ( long n ) {  
    if ( n == 0 ) return 1;  
    else return n * Fact (n-1); }  
  
long Fact ( long n ) {  
    t=1;  
    for(i=1; i<=n; i++) t=t*i;  
    return t; }
```



>单向递归—>循环结构


虽然有一处以上的递归调用语句，但各次递归调用语句的参数只和主调函数有关，相互之间参数无关，并且这些递归调用语句处于算法的最后。

```
long Fib ( long n ) { // Fibonacci数列  
    if(n==1 || n==2) return 1;  
    else return Fib (n-1)+ Fib (n-2);  
}
```

$$Fib(n) = \begin{cases} 1 & \text{若 } n = 1 \text{ 或 } 2 \\ Fib(n-1) + Fib(n-2) & \text{其它} \end{cases}$$

```
long Fib ( long n ) { // Fibonacci数列  
    if(n==1 || n==2) return 1;  
    else return Fib (n-1)+ Fib (n-2);  
}
```

```
long Fib ( long n ) {  
    if(n==1 || n==2) return 1;  
    else{  
        t1=1; t2=1;  
        for(i=3; i<=n; i++){  
            t3=t1+t2;  
            t1=t2; t2=t3; }  
        return t3; } }
```



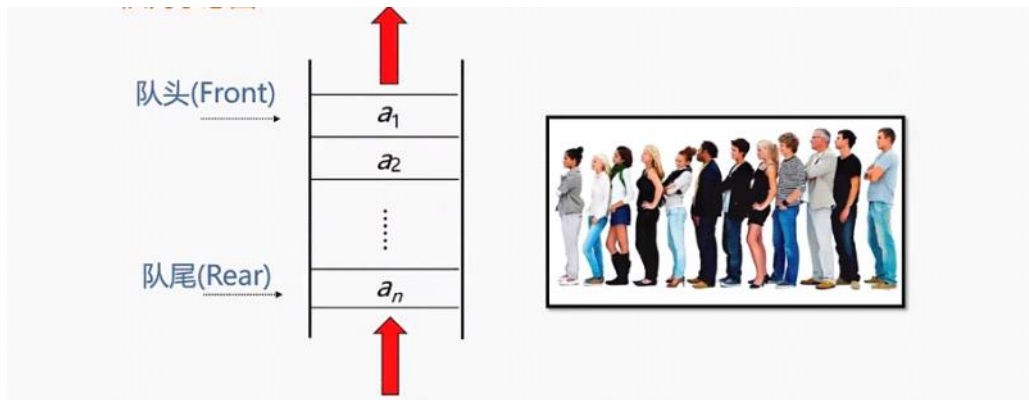
借助栈改写递归方法：

- 递归程序在执行时需要系统提供栈来实现
- 仿照递归算法执行过程中递归工作栈的状态变化可写出相应的非递归程序
- 改写后的非递归算法与原来的递归算法相比，结构不够清晰，可读性较差，有的还需要经过一系列优化

- (1) 设置一个工作栈存放递归工作记录 (包括实参、返回地址及局部变量等)。
- (2) 进入非递归调用入口 (即被调用程序开始处) 将调用程序传来的实在参数和返回地址入栈 (递归程序不可以作为主程序, 因而可认为初始是被某个调用程序调用)。
- (3) 进入递归调用入口: 当不满足递归结束条件时, 逐层递归, 将实参、返回地址及局部变量入栈, 这一过程可用循环语句来实现—模拟递归分解的过程。
- (4) 递归结束条件满足, 将到达递归出口的给定常数作为当前的函数值。
- (5) 返回处理: 在栈不空的情况下, 反复退出栈顶记录, 根据记录中的返回地址进行题意规定的操作, 即逐层计算当前函数值, 直至栈空为止—模拟递归求值过程。

## 五、队列的表示和操作的实现

### 1、队列的示意图:



### 2、相关术语:

- 队列 (Queue) 是仅在表尾进行插入操作, 在表头进行删除操作的线性表。
- 表尾即  $a_n$  端, 称为队尾; 表头即  $a_1$  端, 称为队头。
- 它是一种先进先出 (FIFO) 的线性表。

例如: 队列  $Q = (a_1, a_2, a_3, \dots, a_{n-1}, a_n)$

插入元素称为入队; 删除元素称为出队。  
队列的存储结构为链队或顺序队 (常用循环顺序队) ?

### 3、相关概念

1.定义	只能在表的一端进行插入运算，在表的另一端进行删除运算的线性表（头删尾插）
2.逻辑结构	与同线性表相同，仍为一对一关系。
3.存储结构	顺序队或链队，以循环顺序队列更常见。
4.运算规则	只能在队首和队尾运算，且访问结点时依照先进先出（FIFO）的原则。
5.实现方式	关键是掌握入队和出队操作，具体实现依顺序队或链队的不同而不同。

#### 4、队列的常见应用

- >脱机打印输出：按申请的先后顺序依次输出
- >多用户系统中，多个用户排成队，分时地循环使用CPU和主存
- >按用户的优先级排成多个队，每个优先级一个队列
- >实时控制系统中，信号按接收的先后顺序依次处理
- >网络电文传输，按到达的时间先后顺序依次进行

#### 5、队列的抽象数据类型定义

```

ADT Queue{
    数据对象:  $D=\{a_i | a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0\}$ 
    数据关系:  $R=\{<a_{i-1}, a_i> | a_{i-1}, a_i \in D, i=2,\dots,n\}$  约定其中  $a_1$  端为队列头,  $a_n$  端为队列尾。
    基本操作:
        InitQueue(&Q)      操作结果: 构造空队列Q
        DestroyQueue(&Q)  条件: 队列Q已存在; 操作结果: 队列Q被销毁
        ClearQueue(&Q)    条件: 队列Q已存在; 操作结果: 将Q清空
        QueueLength(Q)    条件: 队列Q已存在 操作结果: 返回Q的元素个数, 即队长
        GetHead(Q, &e)    条件: Q为非空队列 操作结果: 用e返回Q的队头元素
        EnQueue(&Q, e)    条件: 队列Q已存在 操作结果: 插入元素e为Q的队尾元素
        DeQueue(&Q, &e)   条件: Q为非空队列 操作结果: 删除Q的队头元素, 用e返回值
        ..... 还有将队列置空、遍历队列等操作.....
} ADT Queue

```

#### 6、队列的顺序表示及实现

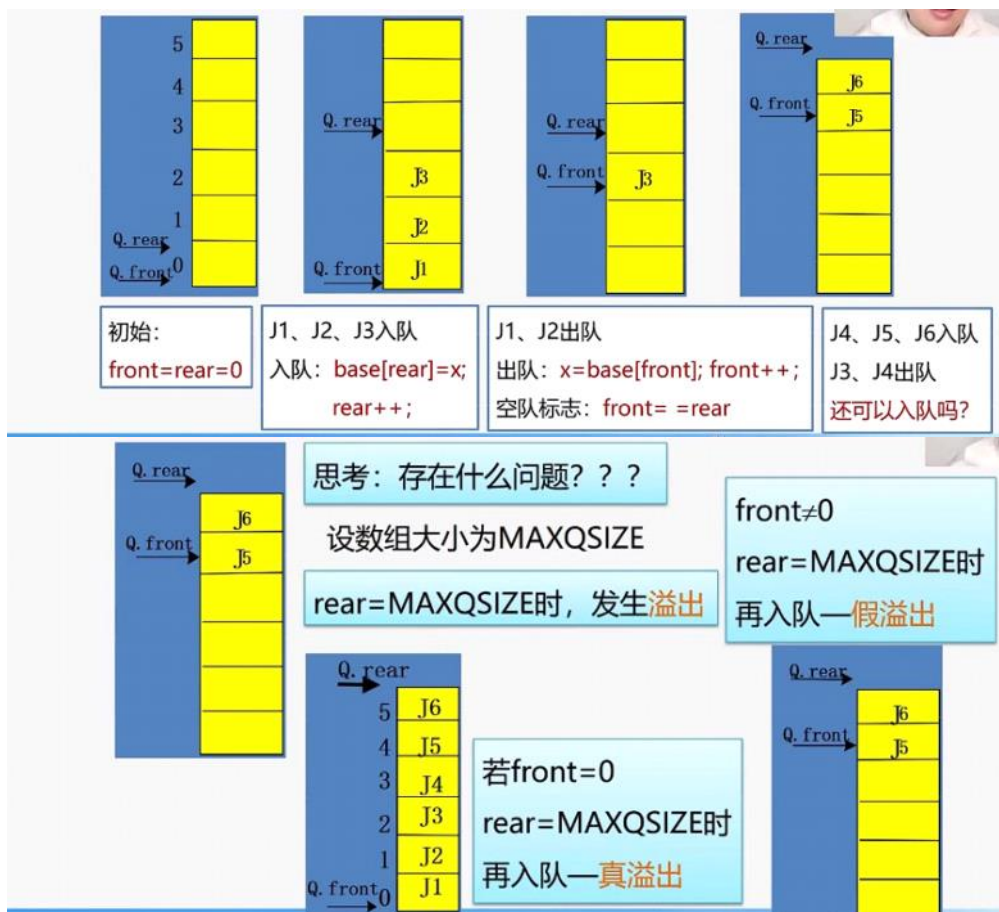
- 队列的顺序表示——用一维数组base[MAXQSIZE]

```

#define MAXQSIZE 100 //最大队列长度
typedef struct {
    QElemType *base; //初始化的动态分配存储空间
    int front;        //头指针
    int rear;         //尾指针
} SqQueue;

```





假上溢：

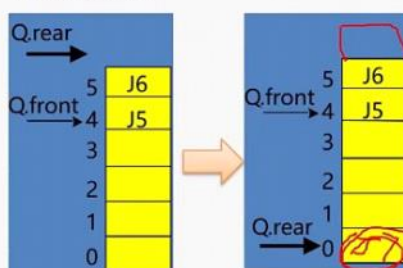
❖ 解决假上溢的方法

1、将队中元素依次向队头方向移动。

缺点：浪费时间。每移动一次，队中元素都要移动。

2、将队空间设想成一个循环的表，即分配给队列的m个存储单元可以循环使用，当rear为maxqsize时，若向量的开始端空着，又可从头使用空着的空间。当front为maxqsize时，也是一样。

MAXQSIZE=6



❖ 解决假上溢的方法——引入循环队列

base[0]接在base[MAXQSIZE -1]之后，若 $rear+1==M$ ，则令 $rear=0$ ；

实现方法：利用模(mod, C语言中：%) 运算。

插入元素：Q.base[Q.rear]=x;

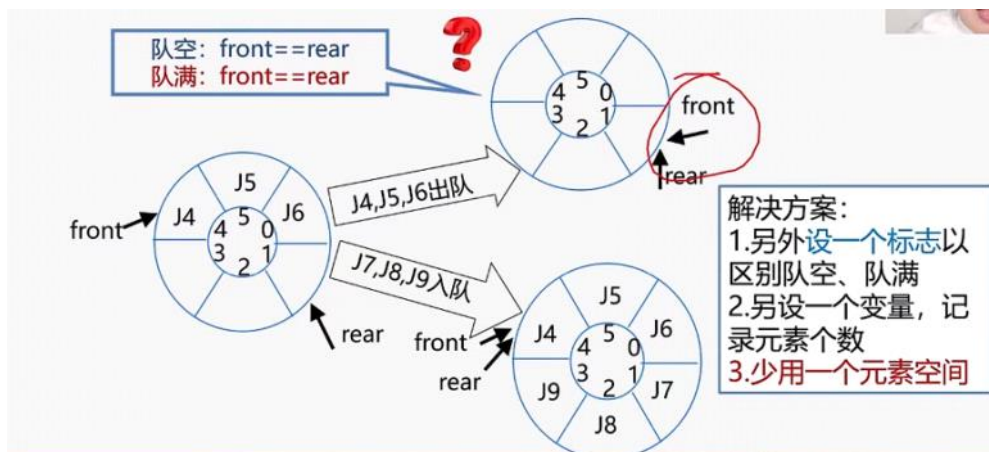
$Q.rear=(Q.rear+1)\% MAXQSIZE$ ;

删除元素：x=Q.base[s.front]

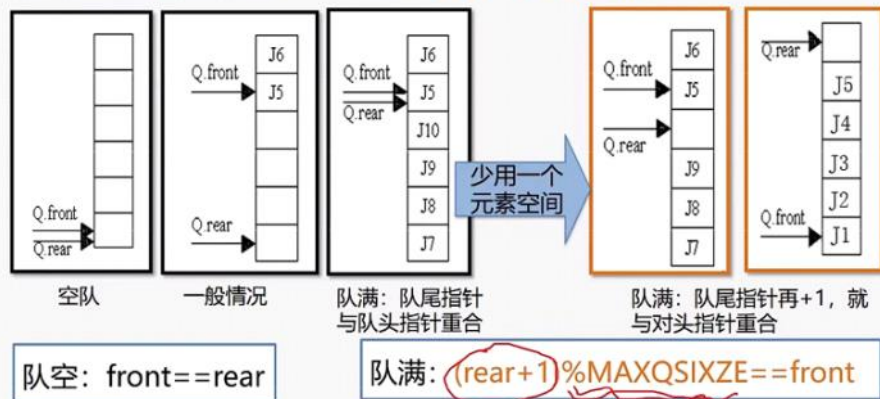
$Q.front=(Q.front+1)\% MAXQSIZE$

循环队列：循环使用为队列分配的存储空间。

循环队列解决队满时判断方法：



循环队列解决队满时判断方法——少用一个元素空间:



## 1) 循环队列

### 类型定义

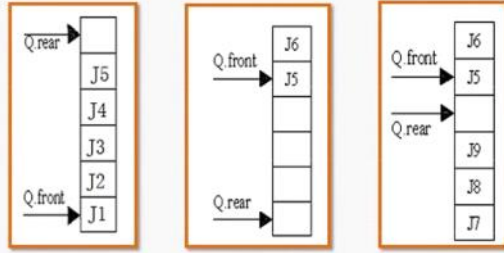
```
#define MAXQSIZE 100 //最大队列长度
typedef struct {
    QElemType *base; // 动态分配存储空间
    int front;        // 头指针, 若队列不空, 指向队列头元素
    int rear;         // 尾指针, 若队列不空, 指向队列尾元素的下一个位置
} SqQueue;
```

### 【算法一】队列的初始化

```
Status InitQueue (SqQueue &Q){
    Q.base = new QElemType[MAXQSIZE] //分配数组空间
    //Q.base = (QElemType*)
    malloc(MAXQSIZE*sizeof(QElemType));
    if(!Q.base) exit(OVERFLOW); //存储分配失败
    Q.front=Q.rear=0; //头指针尾指针置为0, 队列为空
    return OK;
}
```

### 【算法二】求队列的长度

```
int QueueLength (SqQueue Q){
    return ( Q.rear-Q.front+MAXQSIZE)%MAXQSIZE );
}
```



#### 【算法四】循环队列入队

```
Status EnQueue(SqQueue &Q, QElemType e){
    if((Q.rear+1)%MAXQSIZE==Q.front) return ERROR; //队满
    Q.base[Q.rear]=e; //新元素加入队尾
    Q.rear=(Q.rear+1)%MAXQSIZE; //队尾指针+1
    return OK;
}
```

#### 【算法五】循环队列出队

```
Status DeQueue (SqQueue &Q, QElemType &e){
    if(Q.front==Q.rear) return ERROR; //队空
    e=Q.base[Q.front]; //保存队头元素
    Q.front=(Q.front+1)%MAXQSIZE; //队头指针+1
    return OK;
}
```

#### 【算法六】取队头元素

```
SElemType GetHead(SqQueue Q){
    if(Q.front!=Q.rear) //队列不为空
        return Q.base[Q.front]; //返回队头指针元素的值, 队头指针不变
}
```

## 2) 链队

当用户无法估计所用队列的长度，则宜采用链队列

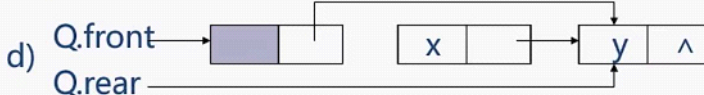
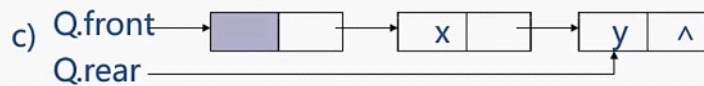
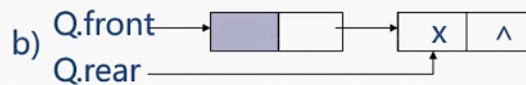


#### ❖ 链队列的类型定义

```
#define MAXQSIZE 100 //最大队列长度
typedef struct Qnode {
    QElemType data;
    struct Qnode *next;
}QNode, *QuenePtr;
```

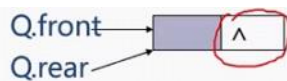
```
typedef struct {
    QuenePtr front; // 队头指针
    QuenePtr rear; // 队尾指针
} LinkQueue; Q
```

#### ❖ 链队列运算指针变化状况



- a) 空队列
- b) 元素x入队列
- c) y入队列
- d) x出队列

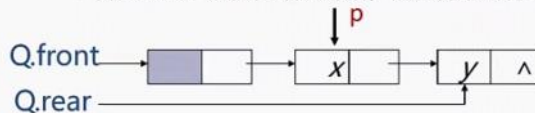
### 【算法一】链队列初始化



```
Status InitQueue (LinkQueue &Q){
    Q.front=Q.rear=(QueuePtr) malloc(sizeof(QNode));
    if(!Q.front) exit(OVERFLOW);
    Q.front->next=NULL;
    return OK;
}
```

### 【算法二】销毁链队列

- 算法思想：从队头结点开始，依次释放所有结点

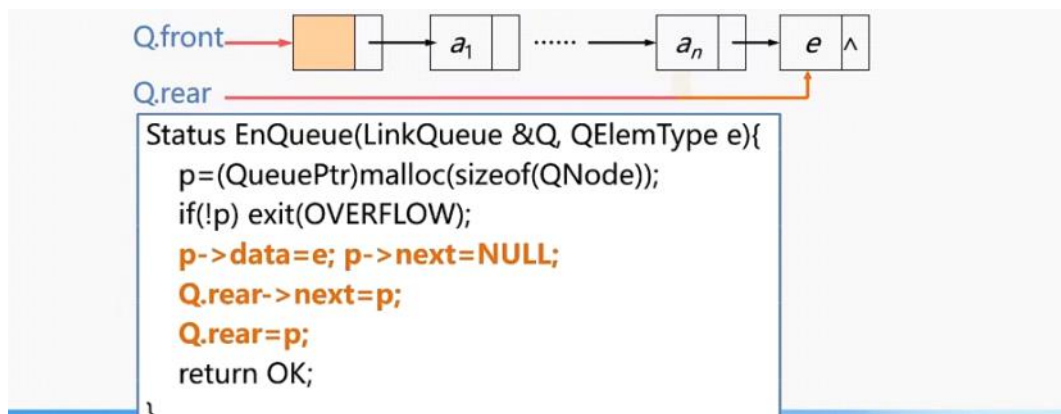


```
Status DestroyQueue (LinkQueue &Q){
    while(Q.front){
        p=Q.front->next; free(Q.front); Q.front=p;
    }
    return OK;
}
```

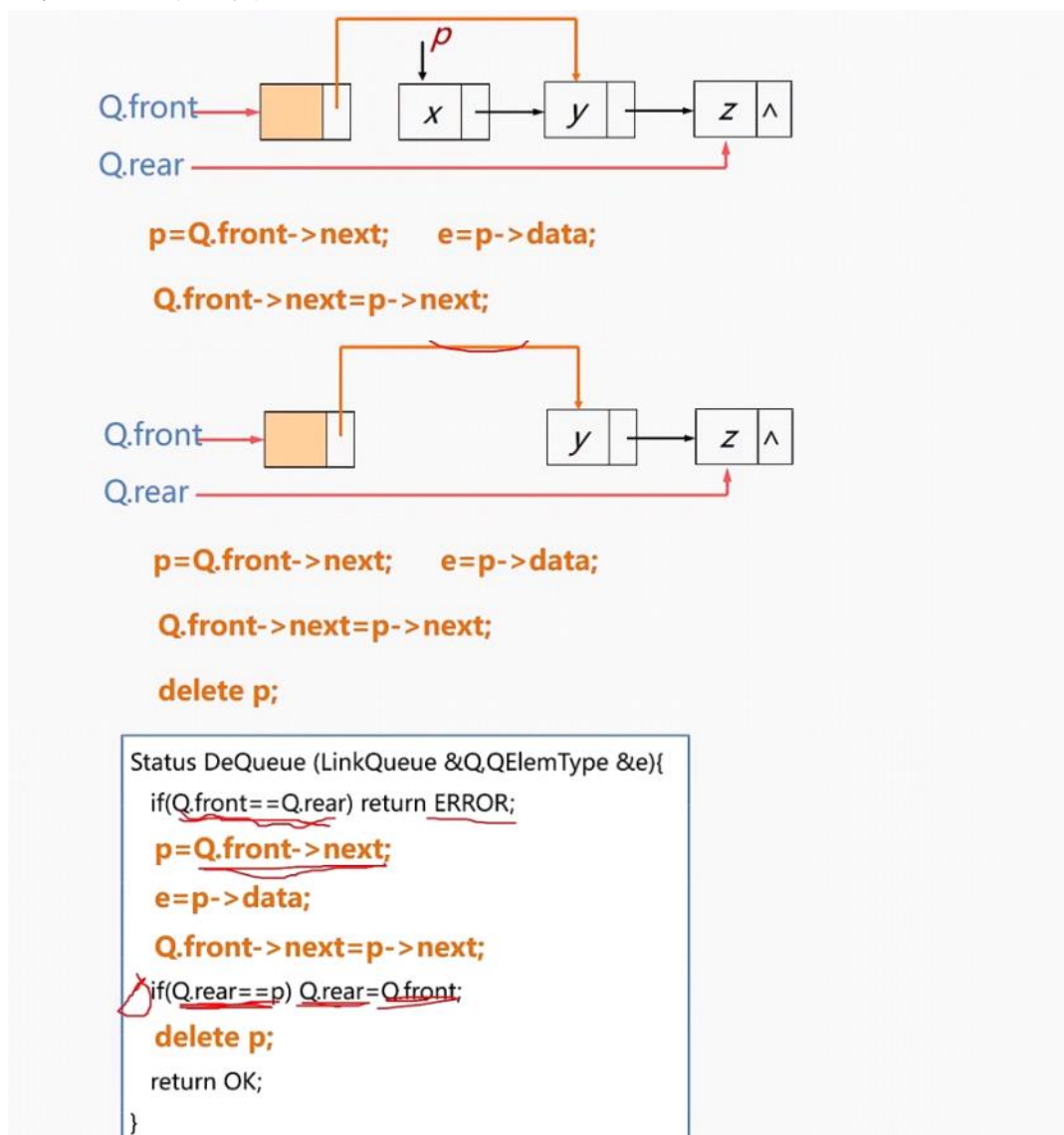
### 【算法三】将元素e入队







#### 【算法四】链队列出队



#### 【算法五】求链队列的队头元素

```
Status GetHead (LinkQueue Q, QElemType &e){  
    if(Q.front==Q.rear) return ERROR;  
    e=Q.front->next->data;
```