数据结构

第三章　栈和队列

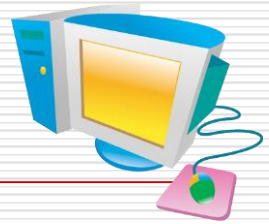
人工智能学院

刘运

主要内容

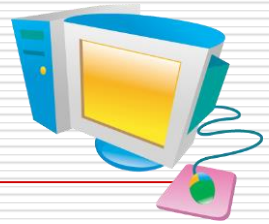
- 3.1 栈和队列的定义和特点
- 3.2 案例引入
- 3.3 栈的表示和操作实现
- 3.4 栈与递归
- 3.5 队列的表示和操作实现
- 3.6 案例分析与实现
- 3.7 小结

学习要点



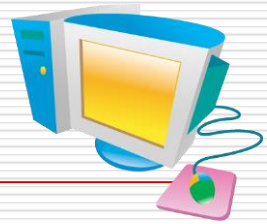
- 栈与队列的特性；
- 栈与队列的抽象数据类型定义（逻辑结构）；
- 栈与队列的存储结构及基本操作实现的算法；
- 栈与队列在程序设计中的应用。

引入： 表达式求值问题



假若我们想在计算机上设计一个小计算器，其功能为：从键盘上输入一个算术表达式（由加、减、乘、除、括号和数字组成）,求要在屏幕上显示输出表达式的求值结果。该如何实现呢？

3.1 栈和队列的定义和特点



3.1.1 栈的定义和特点

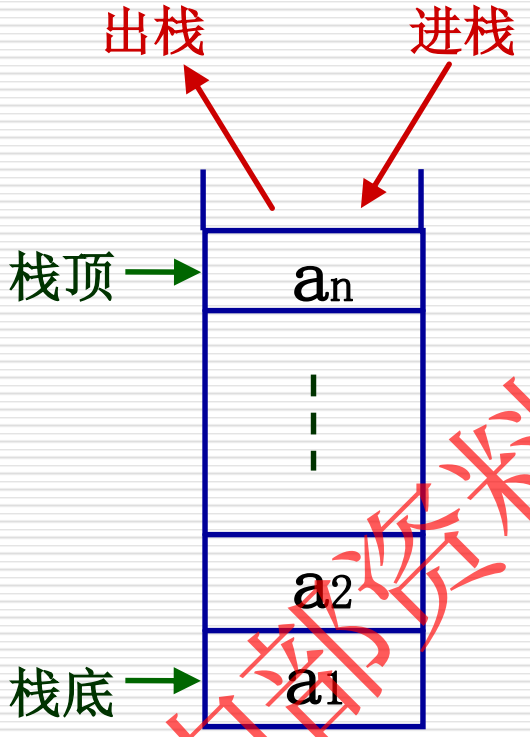
栈（stack）是限定仅在表尾进行插入或删除操作的线性表。

注：

从数据结构角度看，是操作受限的线性表。

从数据类型角度看，是与线性表大不相同的重要的抽象数据类型。

对栈来说，表尾端有其特殊含义，称为栈顶，相应地，表头端称为栈底。



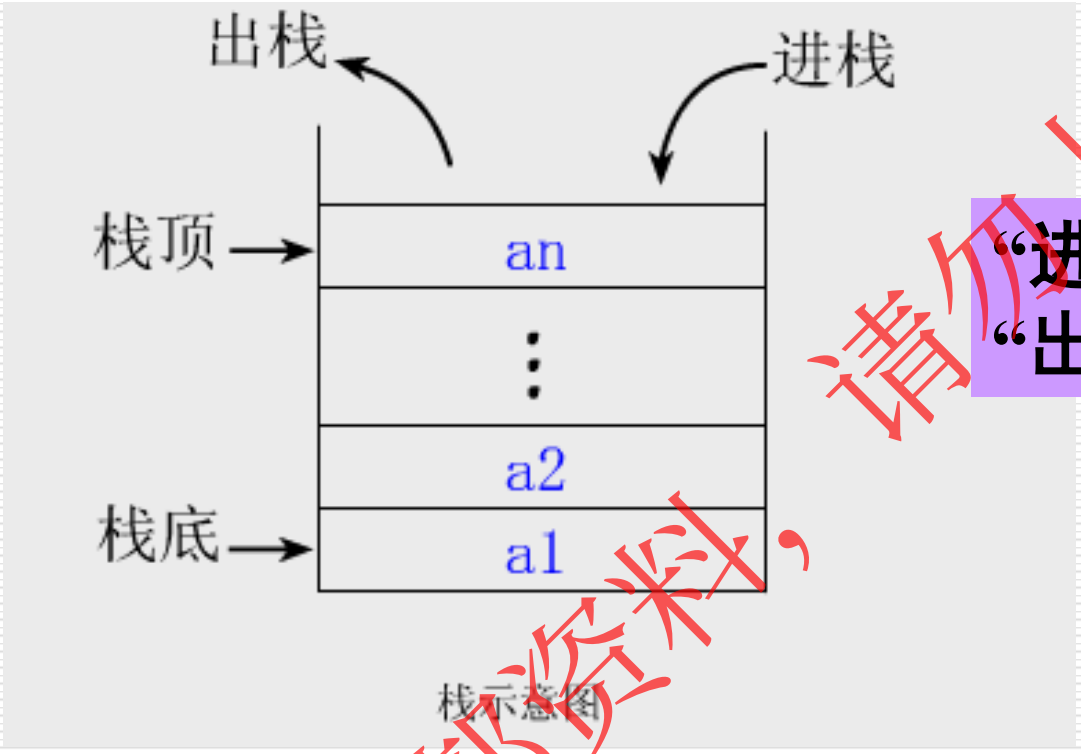
栈的示意图

栈的特点：

后进先出

(last in first out, LIFO)

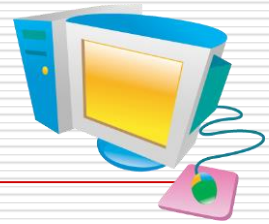
3.3 栈的表示和操作的实现



栈示意图

“进” ＝压入=PUSH()
“出” ＝弹出=POP( )

3.3.1 栈的类型定义



ADT Stack {

数据对象: $D=\{ a_i \mid a_i \in ElemSet, i=1, 2, \ldots, n, n>=0\}$

数据关系: $R1=\{< a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2, \ldots, n\}$

约定其中a_1端为栈底，a_n端为栈顶。

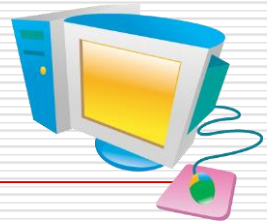
基本操作：

……

}//ADT Stack

建栈、进栈、出栈、取栈顶元素等，教材中列出了9种基本操作。

栈的基本操作



1） 初始化操作InitStack(&S)

操作结果：构造一个空栈S。

2） 销毁栈操作DestroyStack(&S)

初始条件：栈S已存在。

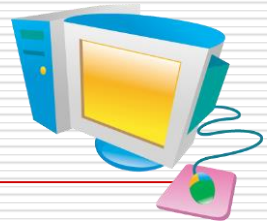
操作结果：栈S被销毁。

3） 清空栈操作ClearStack(&S)

初始条件：栈S已存在。

操作结果：将S清为空栈。

栈的基本操作



4）判栈空操作StackEmpty(S)

初始条件：栈S已存在。

操作结果：若栈S为空栈，则返回TRUE，否则返回FALSE。

5）求栈长操作StackLength(S)

初始条件：栈已经存在。

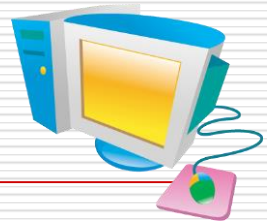
操作结果：返回S的元素个数，即栈的长度。

6） 取栈顶元素操作GetTop(S, &e)

初始条件：栈S已存在并且非空。

操作结果：用e返回S的栈顶元素。

栈的基本操作



7）进栈操作Push(&S, e)

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。

8）出栈操作Pop(&S, &e)

初始条件：栈S已存在并且非空。

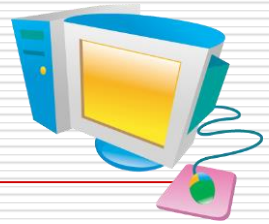
操作结果：删除S的栈顶元素，并用e返回其值。

9）遍历操作StackTraverse(S)

初始条件：栈S已存在并且非空。

操作结果：从栈底到栈顶依次对S的每个数据元素进行访问。

例1



一个栈的输入序列为1，2，3，若在入栈的过程中允许出栈，则可能得到的出栈序列是什么？

答：　可以通过穷举所有可能性来求解：

1）123：　1入1出，2入2出，3入3出；

2）132：　1入1出，2、3入，3、2出；

3）213：　1、2入，2、1出，3入3出；

4）231：　1、2入，2出，3入3出，1出；

5）312：　不可能！

6）321：　1、2、3入，3、2、1出。

例2

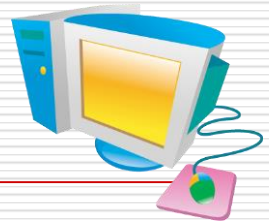
一个栈的输入序列是1，2，……n,若在入栈的过程中允许出栈，则一共有多少种可能的出栈序列？

答：

$$C_n = \frac{(2n)!}{(n+1)!\,n!}$$

神奇的卡特兰数！

例3

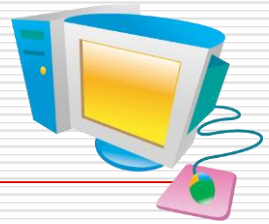


一个栈的输入序列是12345，若在入栈的过程中允许出栈，则栈的输出序列43512可能实现吗？12345的输出呢？

答：43512不可能实现，其中的1不可能先出2出栈；

12345的输出可以实现，每压入一数便立即弹出即可。

例4



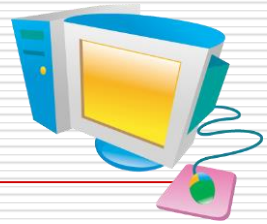
一个栈的输入序列是1，2，……n,若第1个出栈的元素是n，则第i个出栈的元素是————？

答：n-i+1。

一个栈的输入序列是1，2，……n,若第1个出栈的元素是1，则第i个出栈的元素是————？

答：不确定。

2013年数据结构考研真题

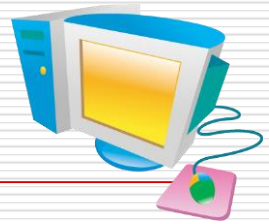


2. 一个栈的入栈序列为1,2,3…,n，其出栈序列是p1,p2,p3,…,pn。若p2=3，则p3可能取值的个数是(C)

A. n-3　　B.n-2　　C.n-1　　D. 无法确定

解析：除了不能取3之外, p3可以取到其它任何值。

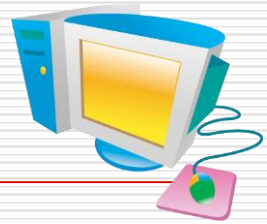
3.3.2 栈的顺序存储和实现



----顺序栈

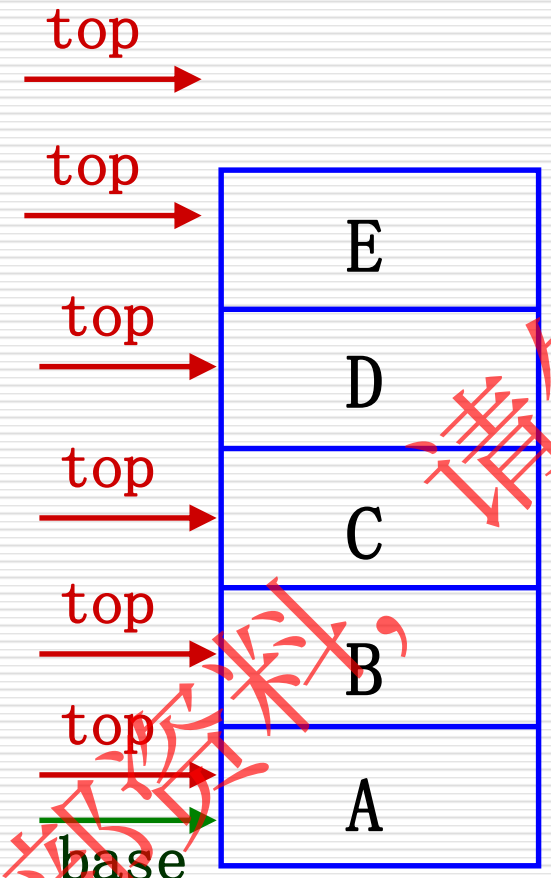
栈的顺序存储结构是利用一组**连续**的内存单元依次存放**自栈底到栈顶**的数据元素，同时附设指针**top**指示栈顶元素在顺序栈中的位置 。

顺序栈的类型定义



```
#define  MAXSIZE 100

typedef struct{

SElemType  *base;    //栈底指针，指向栈空间基址(栈底)

SElemType  *top    //栈顶指针,指向栈顶元素的上一个位置

int stacksize;     //栈可用最大容量

}SqStack;
```

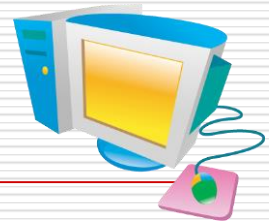
栈操作图示



顺序栈基本操作如
建空栈、进栈、出栈
等操作如何实现？

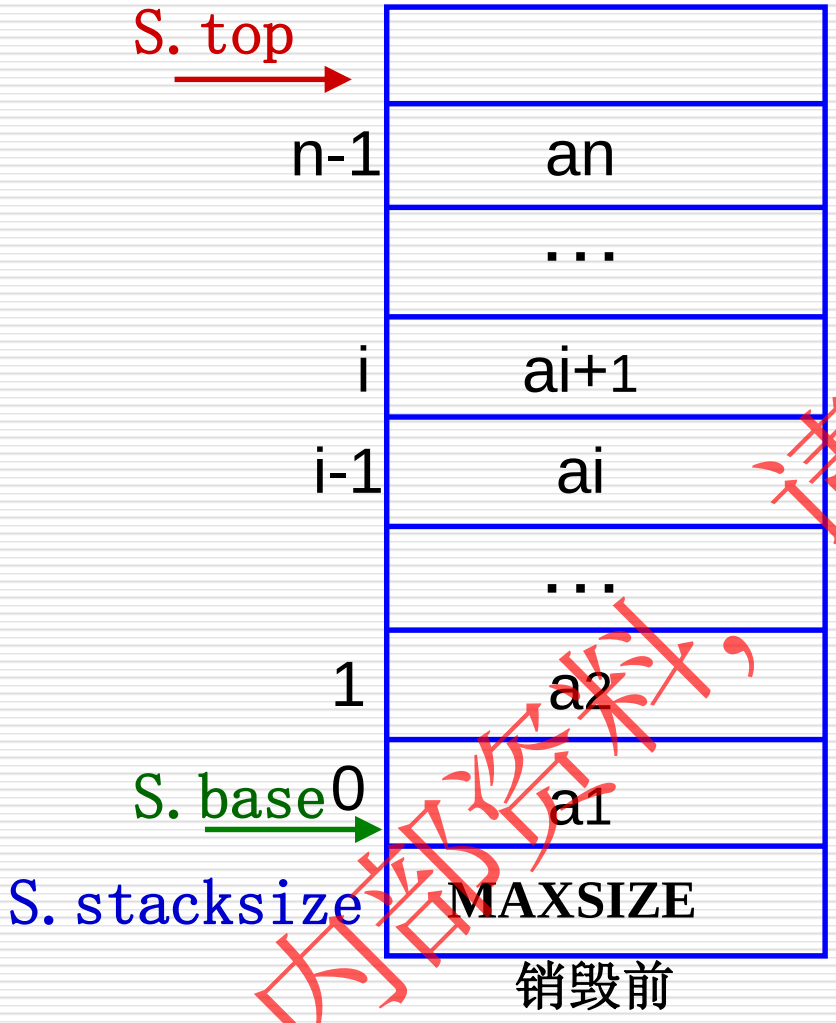
初始化操作图示

1）初始化操作

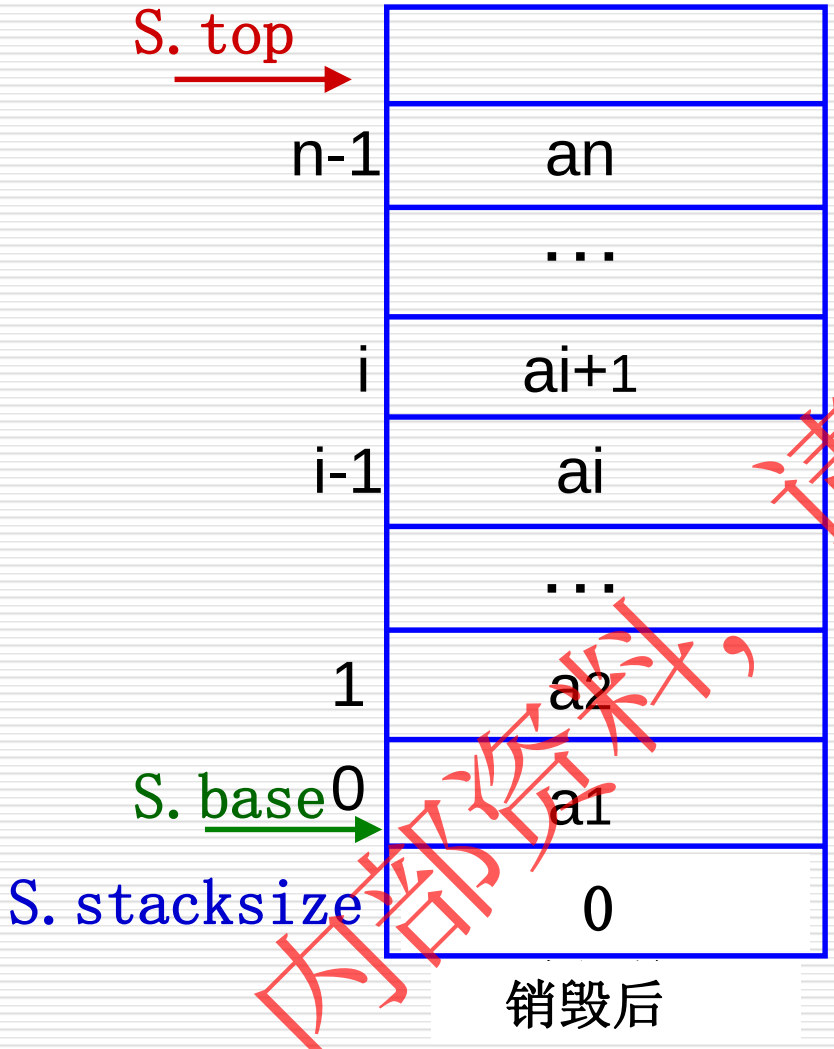


```
Status InitStack_Sq(SqStack &S) {
  //构造一个空栈S
     S.base=new SElemType[MAXSIZE];
     //为顺序栈动态分配存储空间
     if (! S. base) exit(OVERFLOW);    //存储分配失败
     S.top=S.base;
     S.stacksize=MAXSIZE;
     return OK;
}//InitStack_Sq
```

销毁栈操作图示



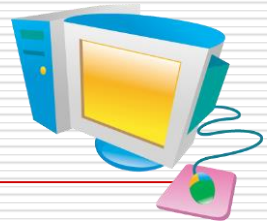
销毁栈操作图示



S. base =null

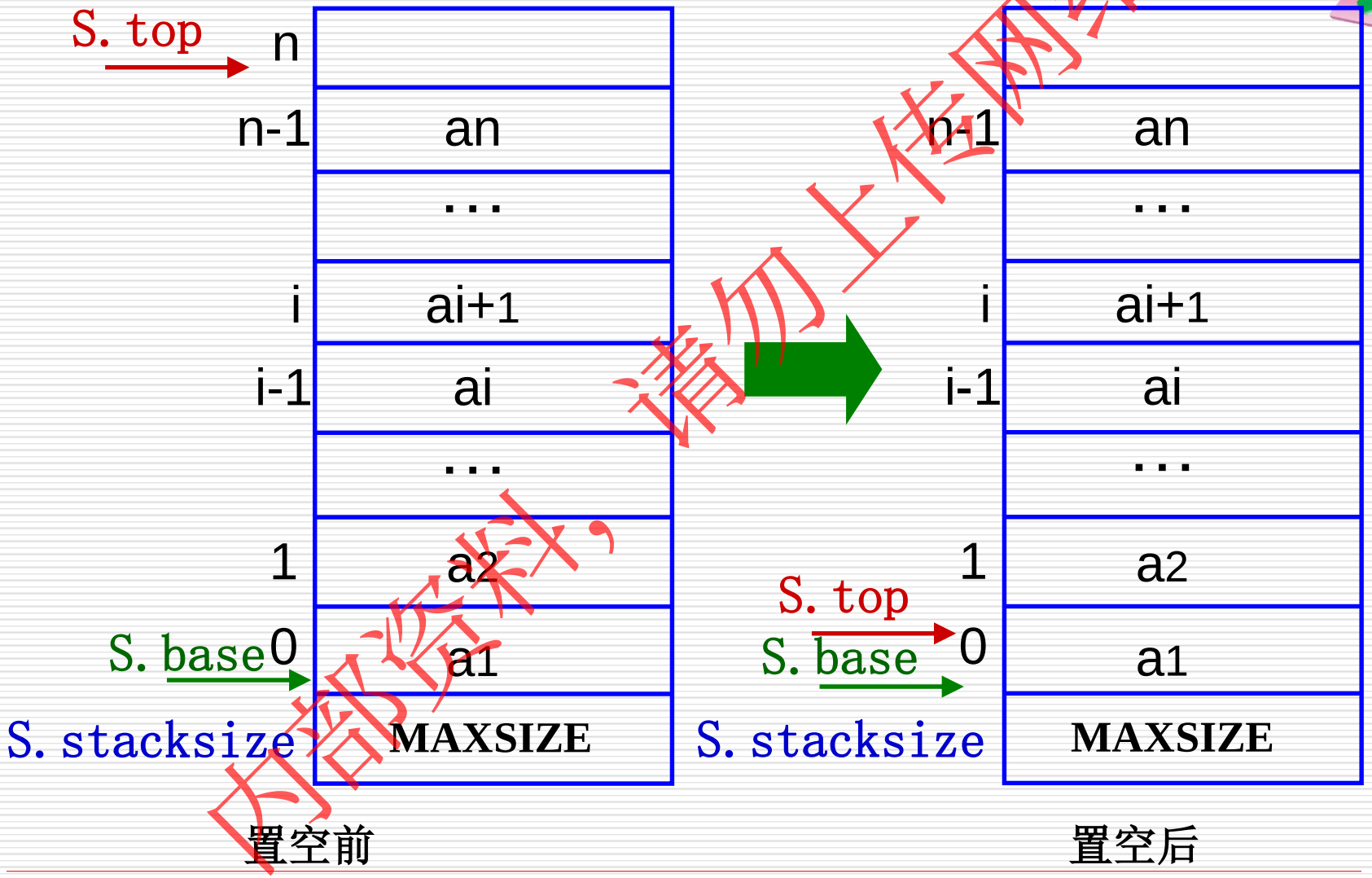
S. top =null

2）销毁栈操作

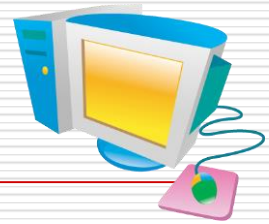


```
Status DetroyStack_Sq ( SqStack &S){
    If (!S.base) return ERROR; // 若栈未建立（尚未分配栈空间）
    free (S.base);                    // 回收栈空间
    S.base = S.top = null;
    S.stacksize = 0;
    return OK;
}// DetroyStack_Sq
```

置空栈操作图示

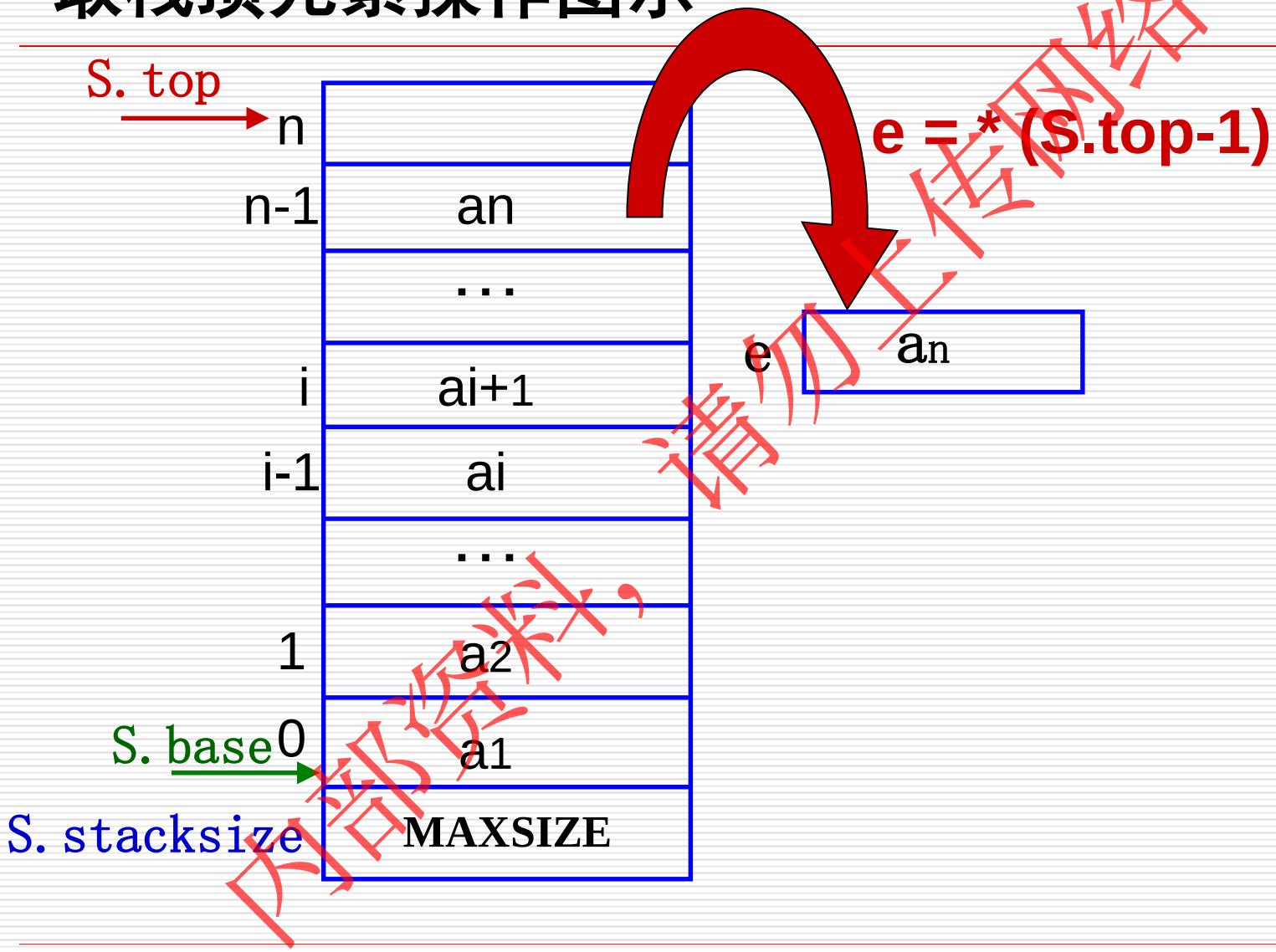


3）置空栈操作

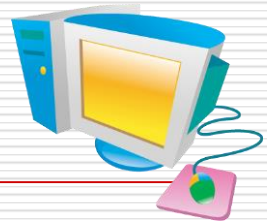


```
Status ClearStack_Sq ( SqStack &S) {
    If (!S.base) return ERROR;
                        // 若栈未建立（尚未分配栈空间）
    S.top = S.base ;
    return OK;
}// ClearStack_Sq
```

取栈顶元素操作图示

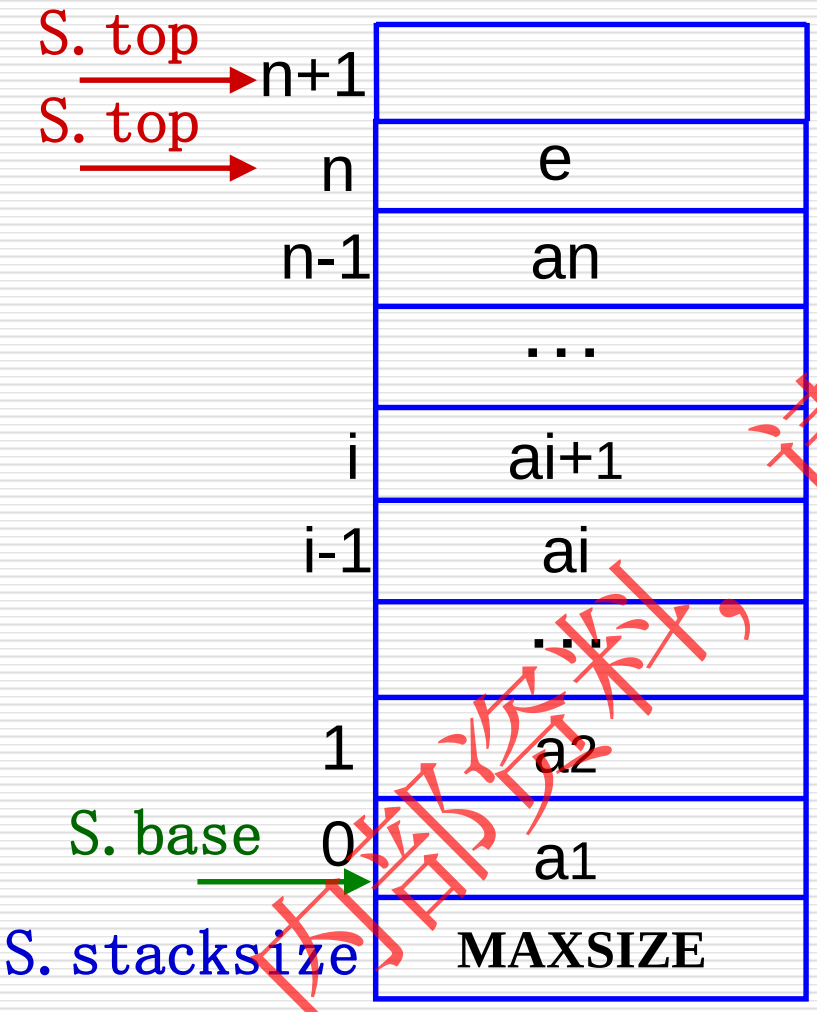


4）取栈顶元素操作



```
Status GetTop_Sq(SqStack S, SelemType &e) {
    // 若栈不空，则用e返回S的栈顶元素，并返回OK；否则返回ERROR
    if (S.top= =S.base)  return ERROR; //若栈为空
    e = * (S.top-1);
    return OK;
}//GetTop_Sq
```

进栈操作图示



进栈操作步骤

1）判断S是否已满

if(S.top-S.base>=S.stacksize)

2）将元素e 写入栈顶

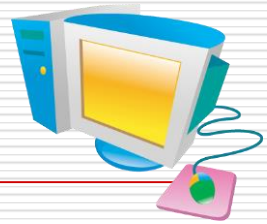
***S.top=e;**

3）修改栈顶指针

S.top++;

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

5）进栈算法描述

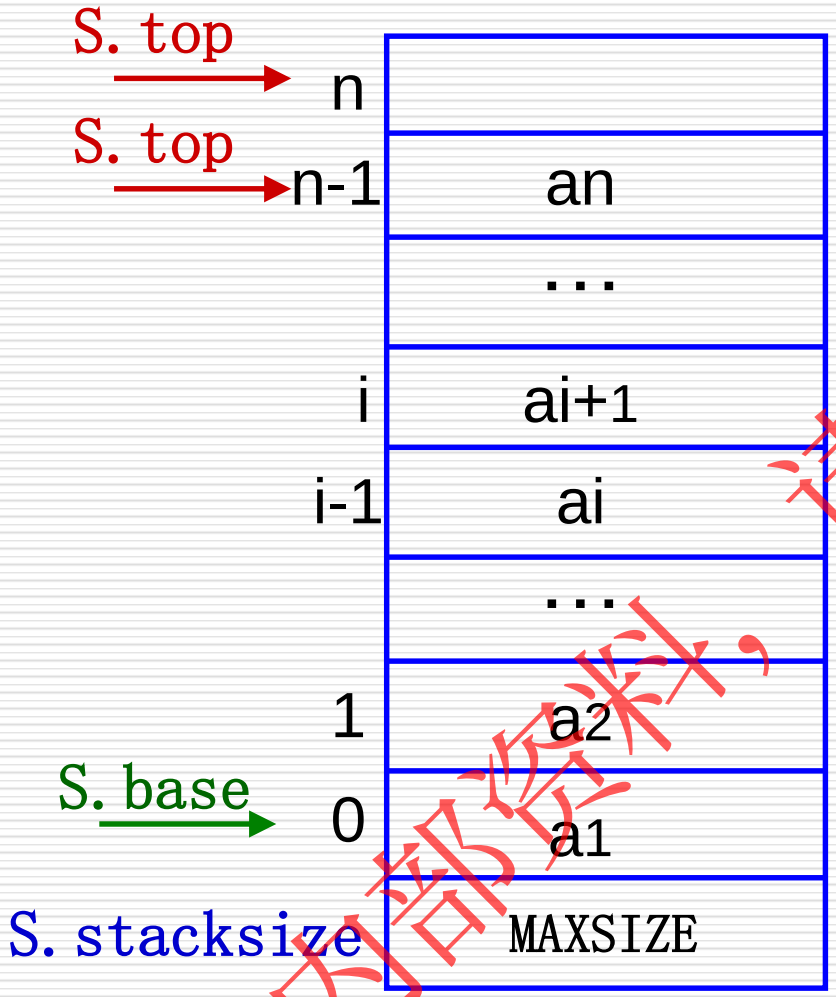


```
Status Push(SqStack &S, SElemType e) {
    //将元素e插入栈成为新的栈顶元素
    if (S.top-S.base==S.stacksize)return ERROR;
     *S.top++=e;     //元素e 插入栈顶，修改栈顶指针
    return OK;
}//Push
```

5）进栈算法描述

```
Status Push(SqStack &S, SElemType e) {
    //将元素e插入栈成为新的栈顶元素
    if (S.top-S.base>=S.stacksize) // 栈满，追加存储空间
     {   S.base= (SElemType * )realloc(S.base,
                (S.stacksize +STACKINCREMENT) * sizeof(ElemType));
       if (! S. base) exit(OVERFLOW);  //存储分配失败
       S.top=S.base+S.stacksize;
       S.stacksize+=STACKINCREMENT;
      }
     *S.top++=e;    //元素e 插入栈顶，修改栈顶指针
    return OK;
}//Push
```

出栈操作图示



出栈操作步骤

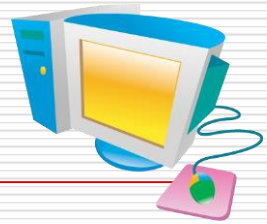
1）判断S是否空

if(S.top= =S.base)

2）取出栈顶元素

e=*--S.top;

6）出栈操作



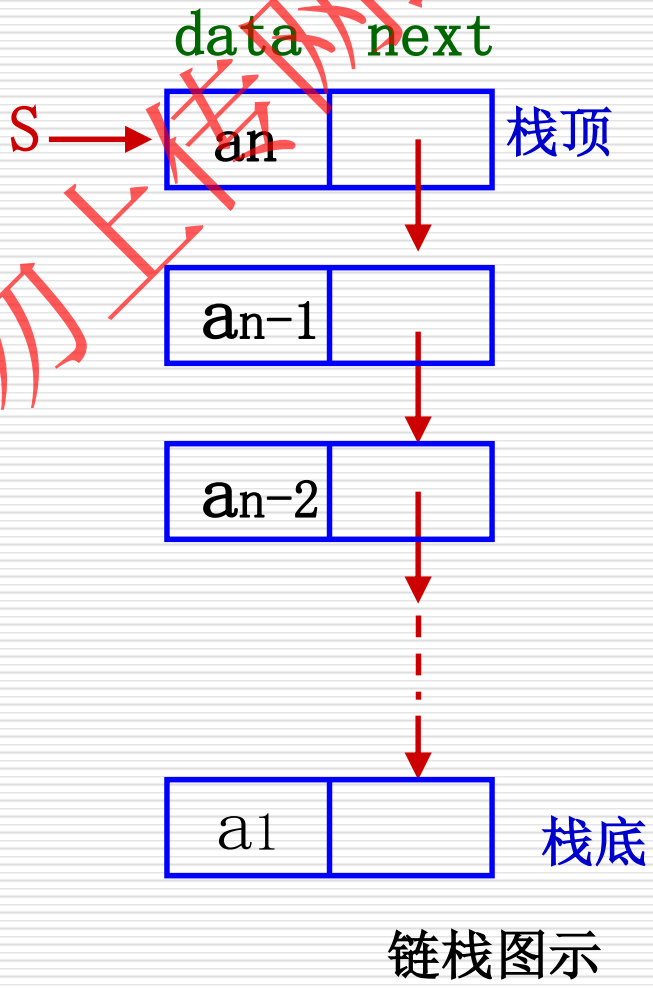
```
Status Pop(SqStack &S, SElemType &e) {
//若栈不空，则删除S的栈顶元素，用e 返回其值并返回OK；
    //否则返回ERROR
    if (S.top= = S.base) return ERROR; //栈空，返回ERROR
    e = * --S.top;        //删除栈顶元素，用e 返回其值，修改栈
顶指针
    return OK;
}//Pop
```

3.3.3 栈的链式存储和实现

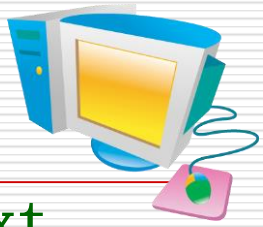
----链栈

链栈通常用单链表表示：

```
typedef  struct SNode{
  Elemtype data;
  struct SNode *next;
}SNode，* LinkStack;
```



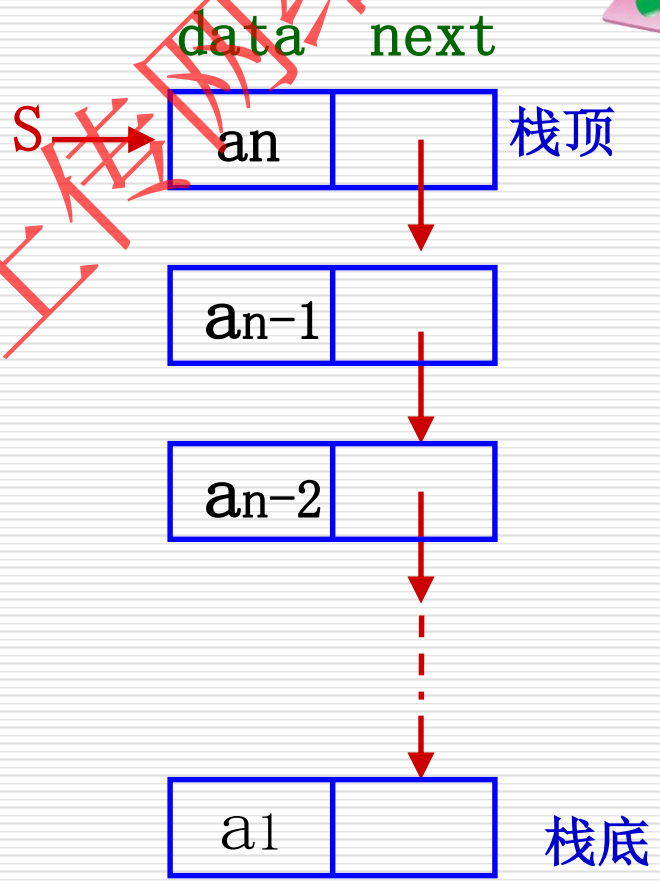
链栈图示



说明：

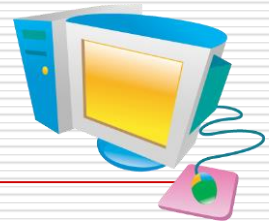
栈顶指针就是链表的头指针；

没有必要再附设一个头结点。



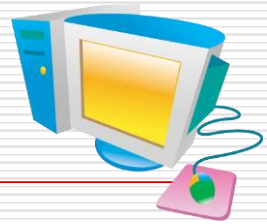
链式栈图示

链栈的初始化



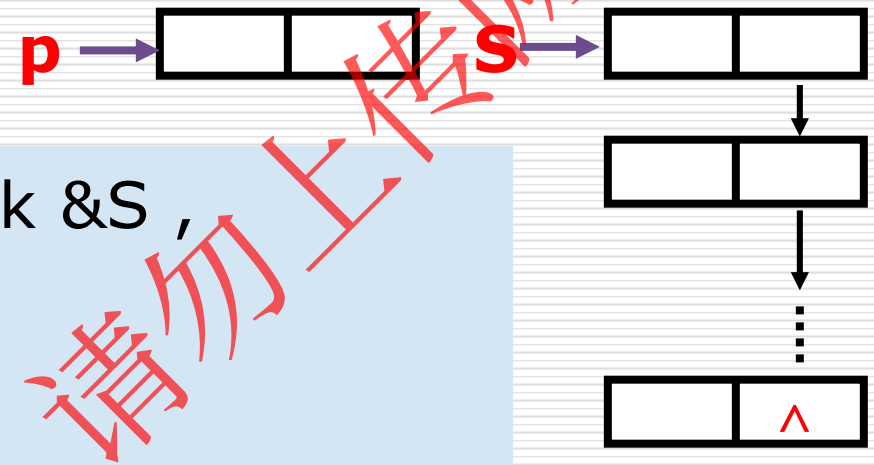

```
void InitStack(LinkStack &S )
{
  S=NULL;
}
```

判断链栈是否为空



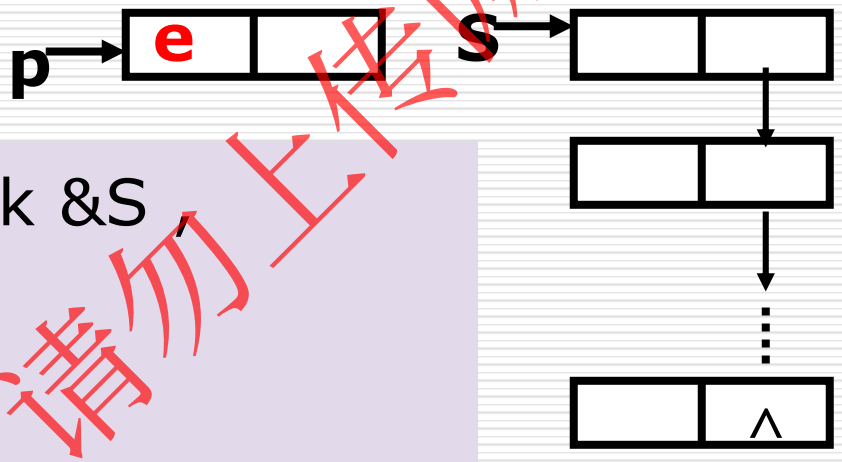
```
Status StackEmpty(LinkStack S)
  {
     if (S==NULL) return TRUE;
     else return FALSE;
  }
```

链栈进栈




```
Status Push(LinkStack &S ,
   SElemType e)
   {
     p=new StackNode;      //生成新
   结点p
     if (!p) exit(OVERFLOW);
     p->data=e; p->next=S; S=p;
      return OK;
}
```

链栈进栈

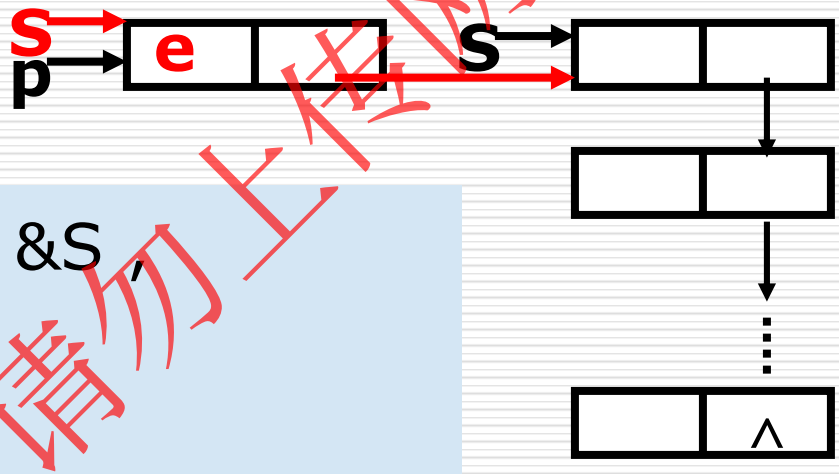



```
Status Push(LinkStack &S ,
   SElemType e)
   {
     p=new StackNode;      //生成新
   结点p
     if (!p) exit(OVERFLOW);
     p->data=e; p->next=S; S=p;
      return OK;
}
```

链栈进栈

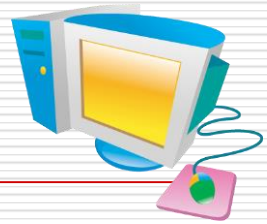

```
Status Push(LinkStack &S ,
    SElemType e)
    {
      p=new StackNode;      //生成新
    结点p
      if (!p) exit(OVERFLOW);
      p->data=e; p->next=S; S=p;
       return OK;
}
```

链栈进栈

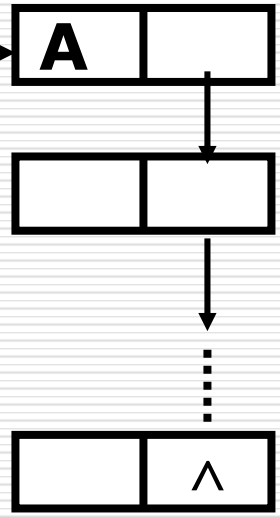



```
Status Push(LinkStack &S ,
    SElemType e)
    {
      p=new StackNode;      //生成新
    结点p
      if (!p) exit(OVERFLOW);
      p->data=e; p->next=S; S=p;
       return OK;
}
```

链栈出栈

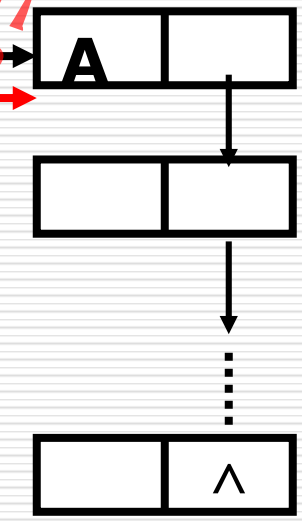


e = 'A'




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

链栈出栈

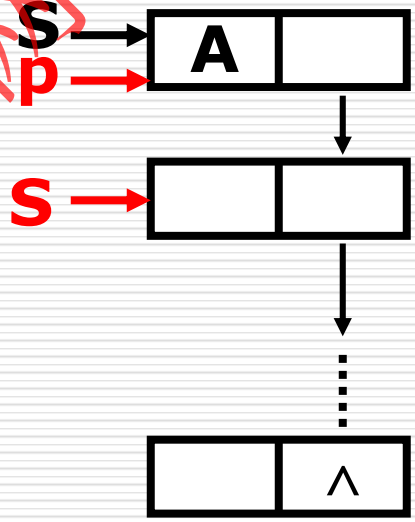


e = 'A'

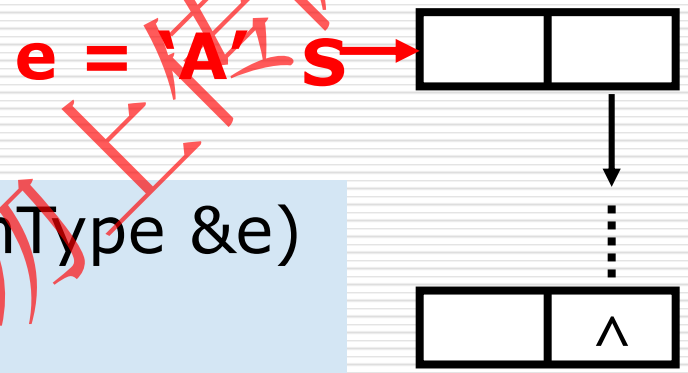
```
Status Pop (LinkStack
&S,SElemType &e)
{
    if (S==NULL) return ERROR;
     e = S-> data;  p = S;   S =  S->
    next;
     delete p;  return OK;
}
```

链栈出栈

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



链栈出栈




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

取链栈栈顶元素

```
SElemType GetTop(LinkStack S)
  {
     if (S==NULL) exit(1);
    else return S->data;
  }
```

数据结构考研真题

1. 若元素a、b、c、d、e、f依次进栈，允许进栈、退栈操作交替进行，但不允许连续三次进行退栈工作，则不可能得到的出栈序列是（ D ）【2010年】

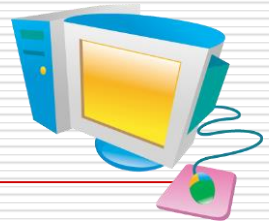
A. dcebfa　B. cbdaef

C. bcaefd　D. afedcb

2.设栈S和队列Q的初始状态均为空，元素abcdefg依次进入栈S。若每个元素出栈后立即进入队列Q，且7个元素出队的顺序是bdcfeag，则栈S的容量至少是（ C ）【2009年】

A. 1　B. 2　C. 3　D. 4

3.4 栈与递归



3.4.1 采用递归算法解决的问题

1.定义是递归的

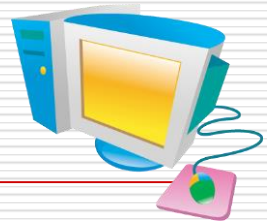
递归定义：一个用自己定义自己的概念,称为递归定义。

例 n!

n!递归定义 n!= 1 当 n=1时

n!= n (n-1)! 当 n>1时

用(n-1)!定义n!



■ 递归函数

一个直接调用自己或通过一系列调用间接调用自己的函数称为递归函数。

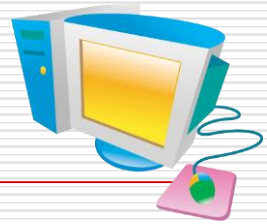
```
A ( ) {
   …
   A( ) ;
   …
}
```

A 直接调用自己

```
B( ) {
   …
   C( );
   …
}
```

```
C( ) {
   …
   B( );
   …
}
```

B间接调用自己



■ 递归算法的编写

1）将问题用递归的方式描述（定义）

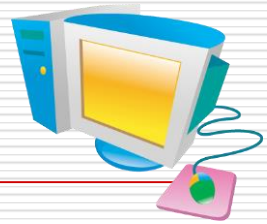
2）根据问题的递归描述（定义）编写递归算法

问题的递归描述（定义）包括两项：

基本项（终止项）：描述递归终止时问题的求解；

递归项：将问题分解为与原问题性质相同，但规模较小的问题；

例1 编写求解 阶乘n! 的递归算法



解： n!的递归定义

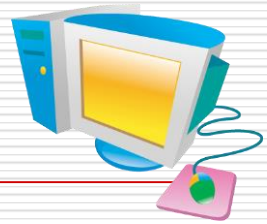
基本项： n!=1 当 n=1

递归项： n!=n (n-1)! 当 n> 1

用(n-1)!定义n!

```
int fact (int n) {
    //算法功能是求解并返回n的阶乘
    if（n==1） return（1）；
    else return（n*fact (n-1)）；
}//fact
```

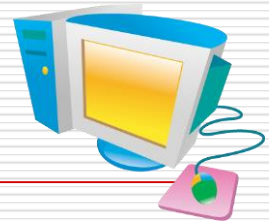
2. 数据结构是递归的



遍历输出链表中各个结点 （算法3.9）

```
void TraverseList(LinkList p)
{
    if(p==NULL) return;
    else
      { cout<<p->data<<endl;
        TraverseList(p->next);
      }
}
```

3. 问题的解法是递归的



例2. 编写求解Hanoi塔问题的递归算法

问题：

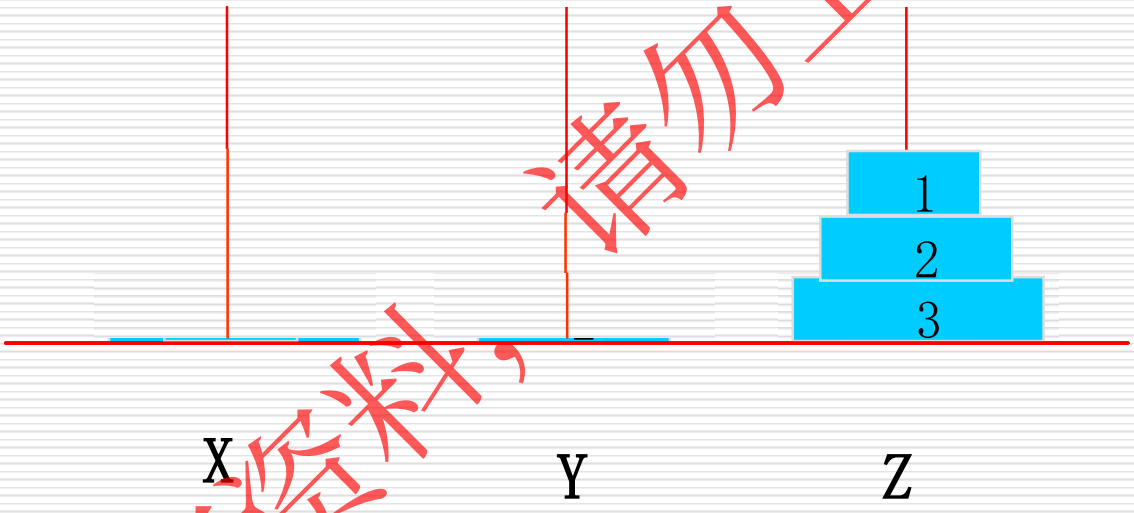
有三个各为X，Y，Z的塔座，在X上有n个大小不同，依小到大编号为1，2…n的圆盘。 现要求将X上的n个圆盘移至Z上，并仍以同样顺序叠放， 圆盘移动时必须遵守下列原则：

1）每次移动一个盘子；

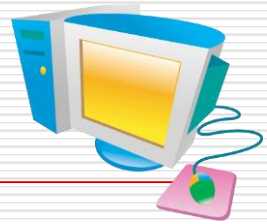
2）圆盘可以放在X，Y，Z中的任一塔座上；

3）任何时刻都不能将较大的圆盘压放在较小圆盘之上；

n=3时圆盘移动的过程如下图所示：



求解Hanoi塔问题的递归描述（定义）



基本项： n=1时，将n号圆盘从X移至Z；

递归项： n>1时，

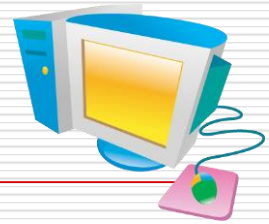
将X上1——n-1号圆盘移至Y；

将X上的n号圆盘从X移至Z；

将Y上1——n-1号圆盘从Y移至Z；

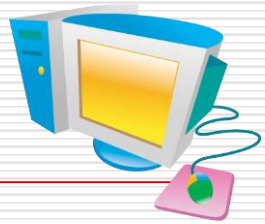
将规模为n的问题的求解归结为规模为n-1的问题的求解

递归算法（算法3. 10）



```
void hanoi (int n, char x, char y, char z)    /*将塔座x上按直径由小到大且自上而下编号为1至n的n个圆盘按规则搬到塔座z上，y可用作辅助塔座。搬动操作move(x, n, z)可定义为(c是初值为0的全局变量，对搬动计数):
printf(“%d Move disk %di from %c to %c\n”, ++c, n, x, z); */
 {
   if (n= =1)
       move(x,1,z);     //将编号为1的圆盘从x移动z
   else {
      hanoi(n-1, x, z, y);   //将x上编号为1至n-1的圆盘移到y, z作辅助塔
      move(x, n, z);         //将编号为 n的圆盘从x移到z
      hanoi(n-1, y, x, z);   //将y上编号为1至n-1的圆盘移到z，x作辅助塔
    }
}
```

函数调用过程



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

调用后，系统完成：

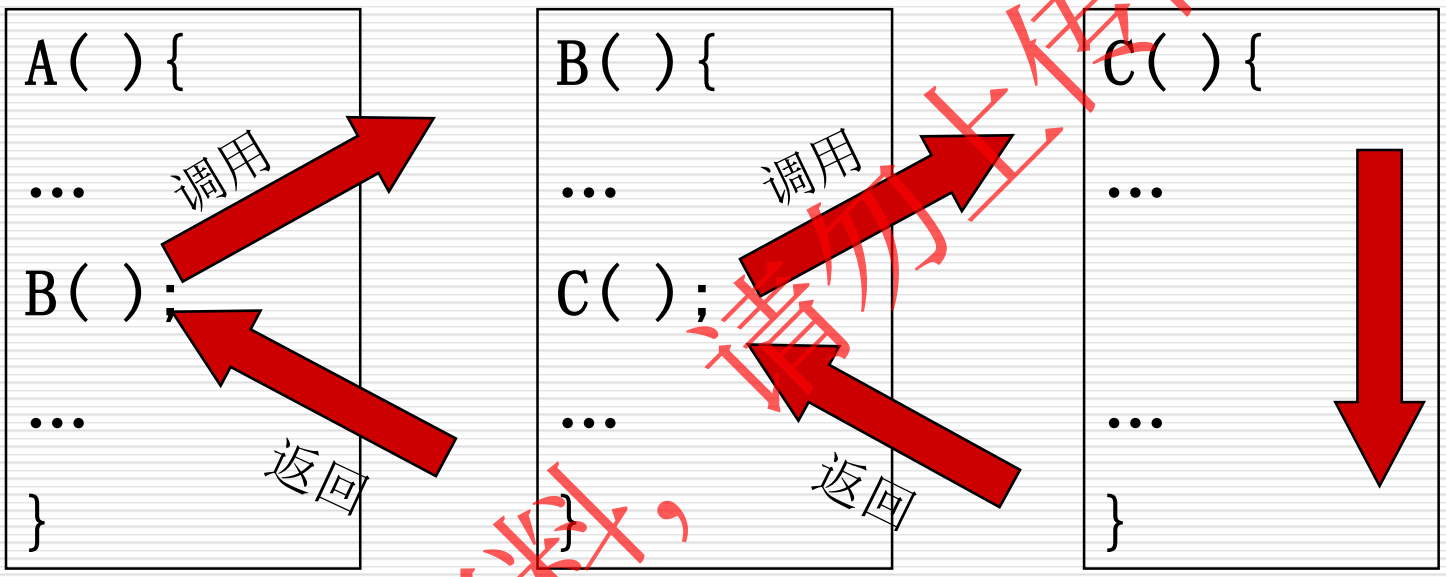
(1)保存被调用函数的计算结果

(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

3 递归函数的实现

一般函数的调用机制的实现：



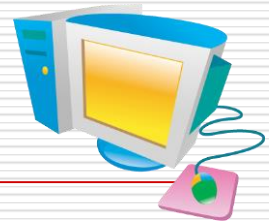
函数调用顺序 A → B → C

函数返回顺序 C → B → A

后调用的函数先返回

函数调用机制可通过栈来实现

n=3 阶乘函数fact(n)的执行过程



```
main( ) {
   int n=3,y;
L  y= fact(n);
}
```

调用 ↓ ↑ 返回 6

```
int fact (int n) {
   if （n= =1） return （1）；
   else
L1   return（n*fact (n-1)）；
}//fact
```

n=3

调用 ↓

```
int fact (int n) {
   if （n= =1） return （1）；
   else
L1   return（n*fact (n-1)）；
}//fact
```

n=2

调用 ↓

```
int fact (int n) {
   if （n= =1） return （1）；
   else
L1   return（n*fact (n-1)）；
}//fact
```

n=1

返回 2 ←

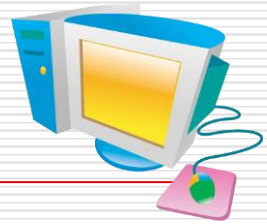
返回 1 ↑

返回地址 实参

L1	1
L1	2
L	3

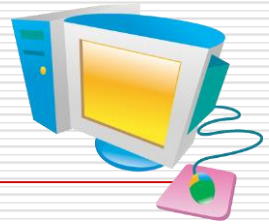
注意递归调用中栈的变化

函数算法的效率分析



设Fact(n)的执行时间是T(n)。此函数中语句if(n==1)return 1;执行时间是O(1),递归调用Fact(n-1)的执行时间是T(n-1)，所以else return（n*fact (n-1)）的执行时间是O(1)+T(n-1)。

函数算法的效率分析



对某常数C、D有如下递归方程：

$$T(n)=\begin{cases} D & n=1 \\ C+T(n-1) & n>=2 \end{cases}$$

设n>2，利用上式对T(n-1)展开，即在上式中用n-1代替n，得

T(n-1)=C+T(n-2)

再代入T(n)=C+T(n-1)中，得

T(n)=2C+T(n-2)

同理，当n>3时有

T(n)=3C+T(n-3)

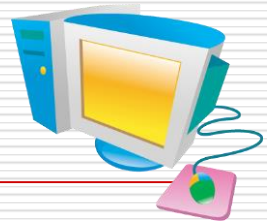
依次类推，当n>i时有

T(n)=iC+T(n-i)

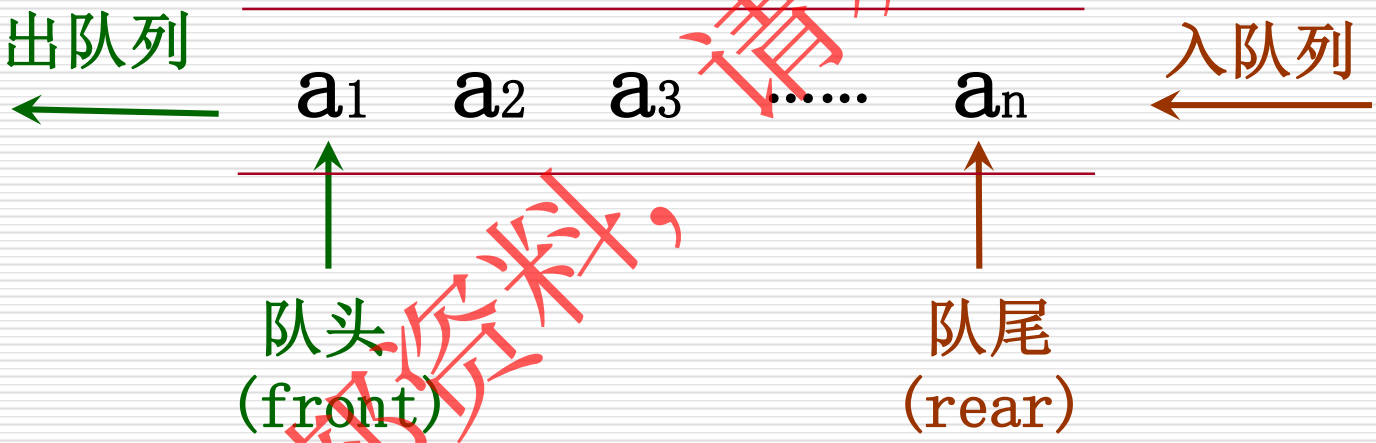
最后，当i=n-1时，T(n)=(n-1)C+T(1)=(n-1)C+D

故求得递归方程的解为：T(n)=O(n)。

3.5 队列的表示和操作实现

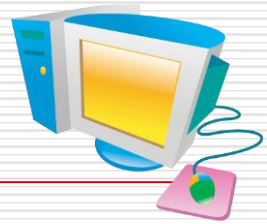


队列（Queue）是一种先进先出（First In First Out，FIFO）的线性表，它只允许在表的一端进行插入，而在另一端删除元素。



队列的示意图

队列的特点



先进先出（First In First Out）

- 例：一个栈的输入序列是1，2，3，可能得到的出栈序列有哪些？

  解：123，132，213，231，321。

- 例：一个队列的输入序列是1，2，3，可能得到的出队序列有哪些？

  解：123。

3.5 队列的表示和操作实现

3.5.1抽象数据类型队列的定义

ADT Queue {

数据对象:

$D=\{ a_i \mid a_i \in ElemSet, i=1, 2, \ldots, n, n>=0\}$

数据关系:

$R1=\{< a_{i-1}, a_i > \mid a_{i-1}, a_i \in D, i=2, \ldots, n\}$

约定其中a_1端为队列头，a_n 端为队列尾。

基本操作：……

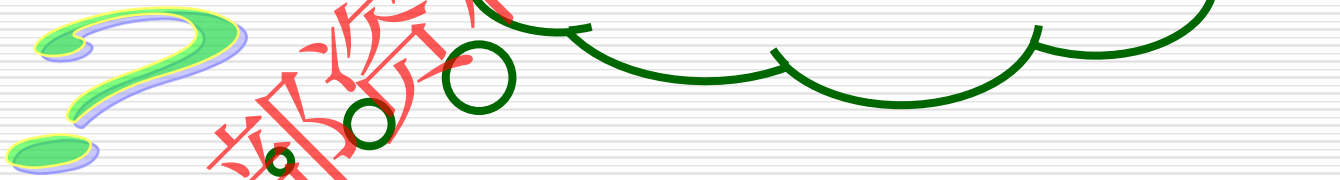
}ADT Queue

队列的基本操作

1	初始化操作	InitQueue(&Q)
2	销毁操作	DestroyQueue(&Q)
3	置空操作	ClearQueue(&Q)
4	判空操作	QueueEmpty(Q)
5	求队长操作	QueueLength(Q)
6	取元素操作	GetQueue(Q, &e)
7	入队操作	EnQueue（&Q,e)
8	出队操作	DeQueue(&Q, &e)
9	遍历操作	ListTraverse (Q,visit( ))

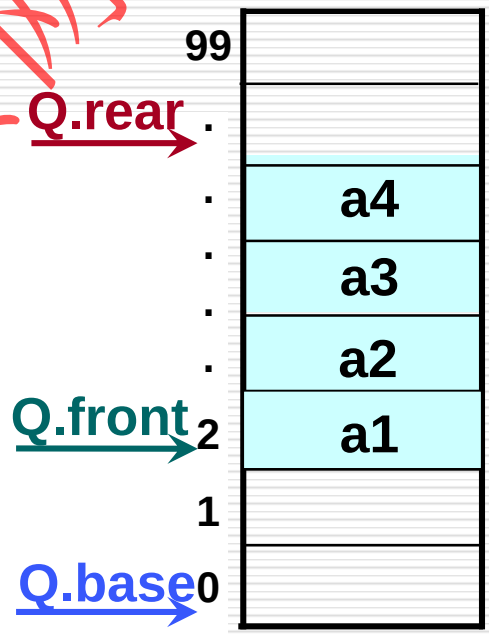
3.5.2 队列的顺序表示和实现

队列的顺序存储结构，就是用一组地址连续的存储单元依次存放从队列头到队列尾的数据元素。

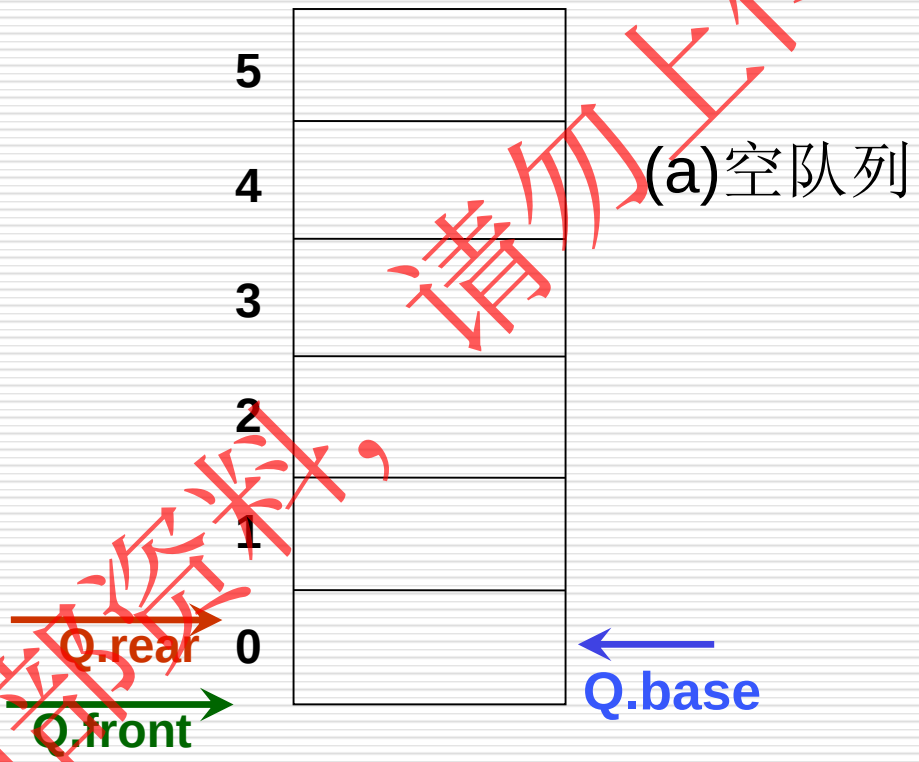


顺序队列的类型定义

```
#define  MAXQSIZE 100
 //  最大队列长度
typedef struct{
    QElemType  *base;
     //指向一维数组的起始地址
    int front;       //指向队头元素
    int rear;      //指向队尾元素的下一个位置
  }SqQueue;
```

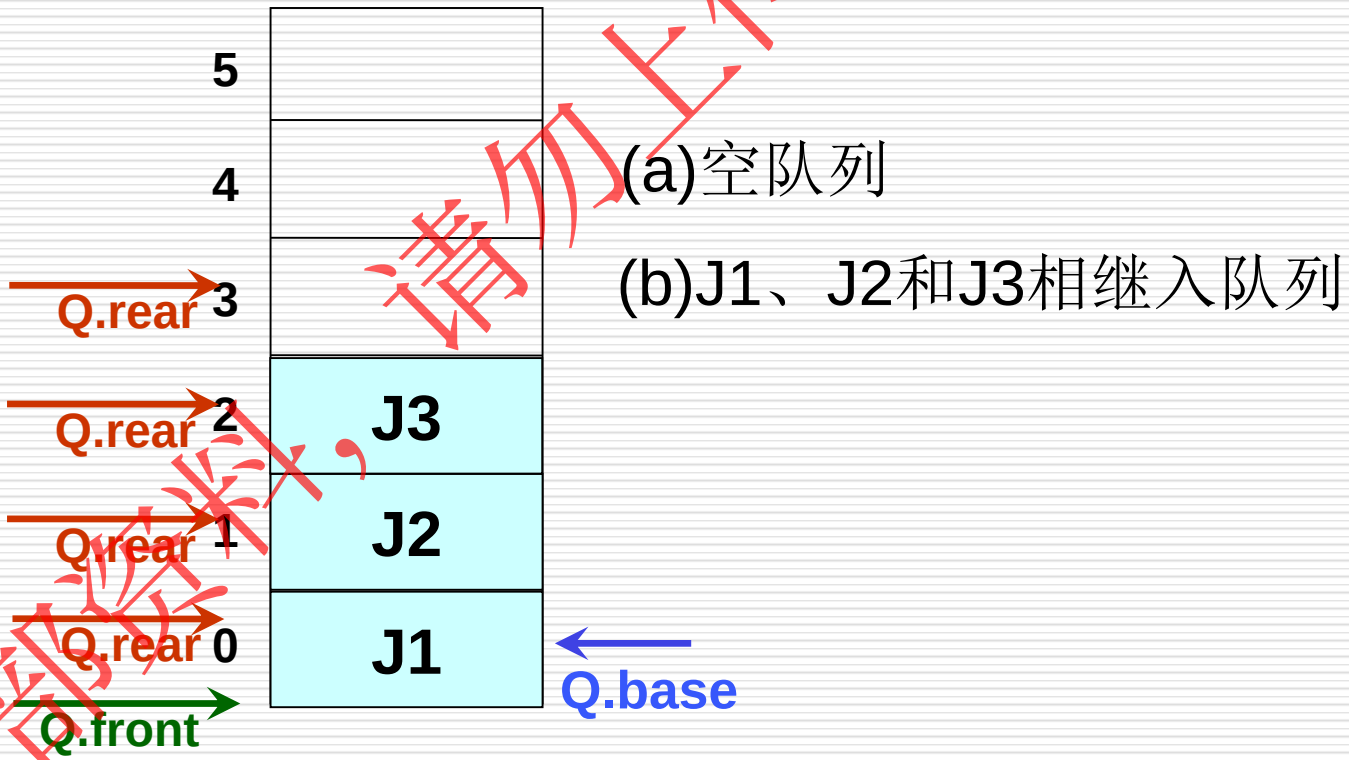


设Q为SqQueue类型的变量，用于存储队列。队列Q的初始分配空间大小为MAXQSIZE=6，约定初始建队时，令Q.front=Q.rear=0;插入新的队列尾元素时，尾指针增1；删除队列头元素时，头指针增1。



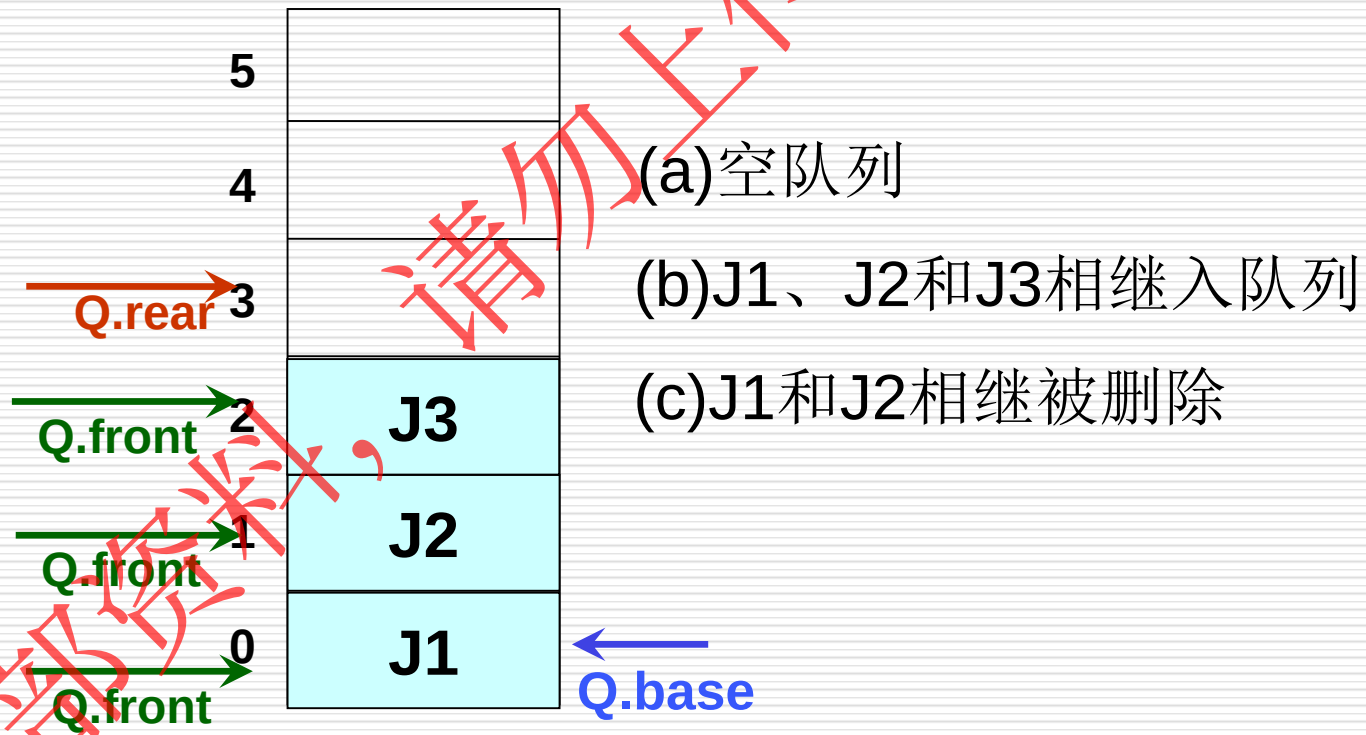
头、尾指针和队列中元素之间的关系图示

设Q为SqQueue类型的变量，用于存储队列。队列Q的初始分配空间大小为MAXQSIZE=6，约定初始建队时，令Q.front=Q.rear=0;插入新的队列尾元素时，尾指针增1；删除队列头元素时，头指针增1。



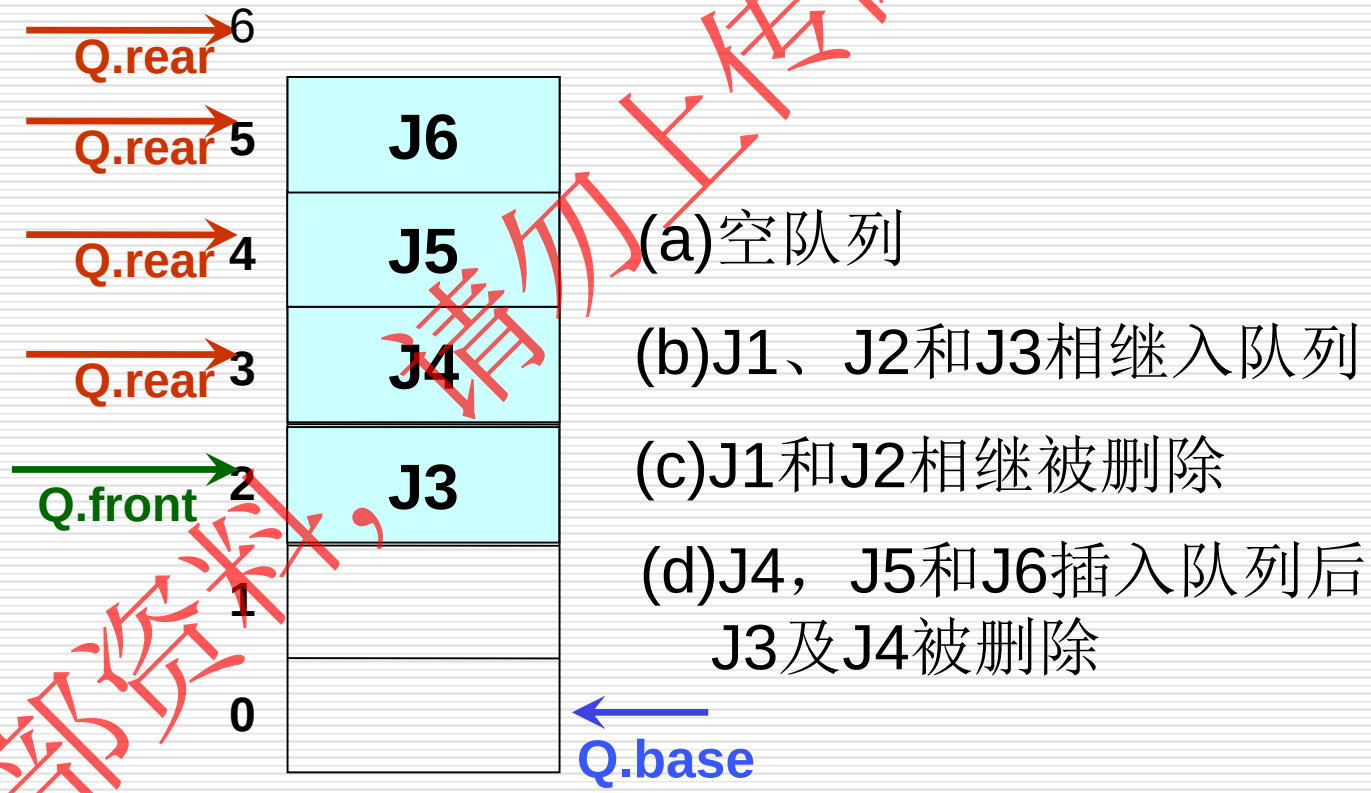
头、尾指针和队列中元素之间的关系图示

设Q为SqQueue类型的变量，用于存储队列。队列Q的初始分配空间大小为MAXQSIZE=6，约定初始建队时，令Q.front=Q.rear=0;插入新的队列尾元素时，尾指针增1；删除队列头元素时，头指针增1。



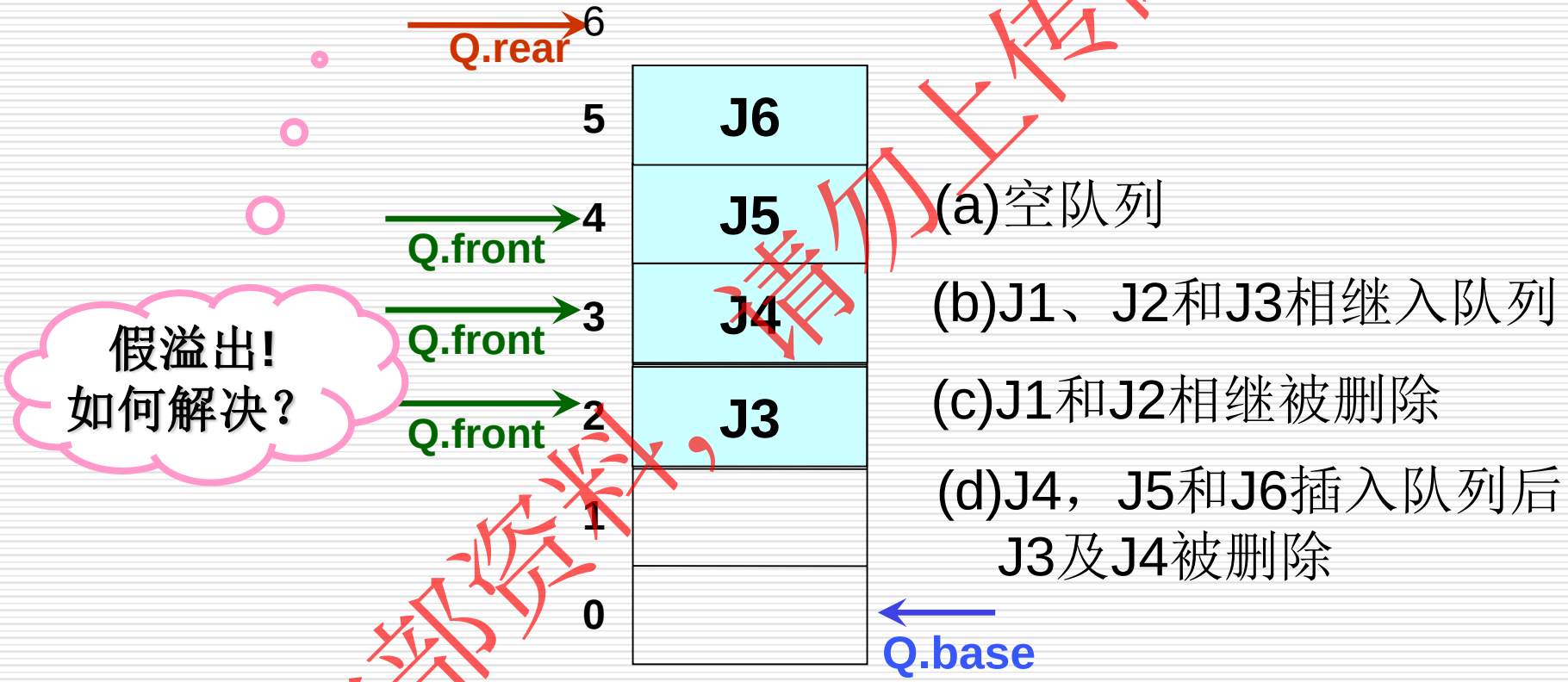
头、尾指针和队列中元素之间的关系图示

设Q为SqQueue类型的变量，用于存储队列。队列Q的初始分配空间大小为MAXQSIZE=6，约定初始建队时，令Q.front=Q.rear=0;插入新的队列尾元素时，尾指针增1；删除队列头元素时，头指针增1。



头、尾指针和队列中元素之间的关系图示

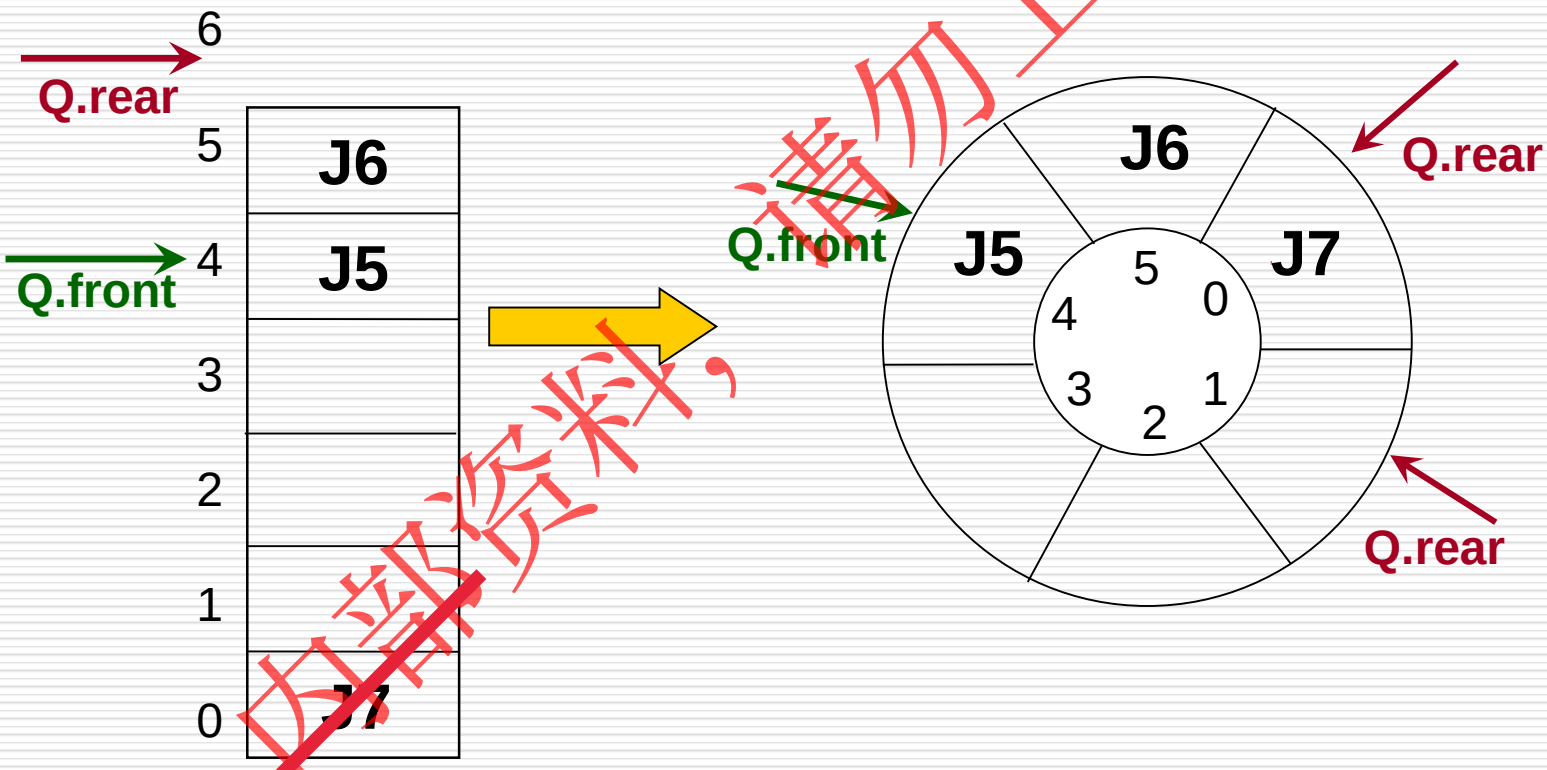
设Q为SqQueue类型的变量，用于存储队列。队列Q的初始分配空间大小为MAXQSIZE=6，约定初始建队时，令Q.front=Q.rear=0;插入新的队列尾元素时，尾指针增1；删除队列头元素时，头指针增1。



头、尾指针和队列中元素之间的关系图示

循环队列

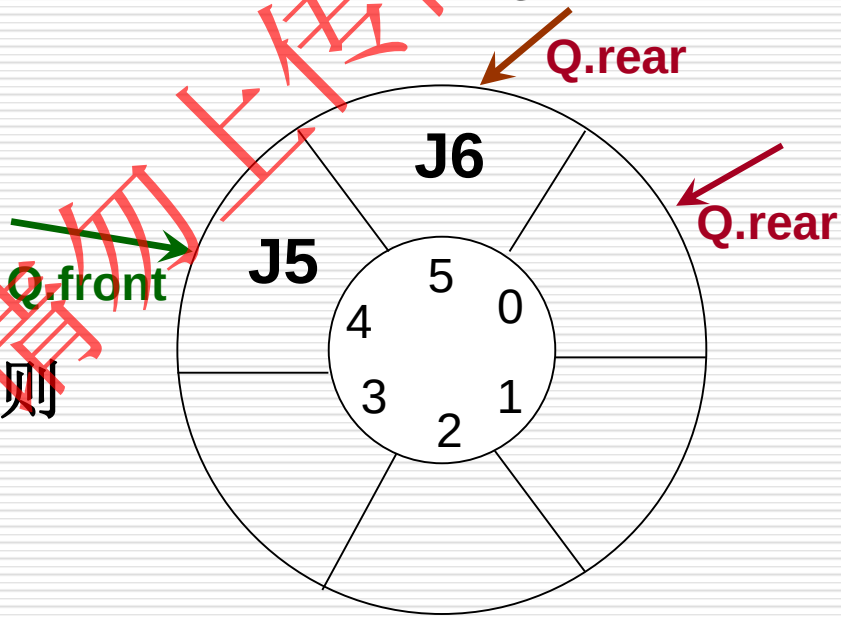
将顺序队列设想为首尾相连的环状空间，如图，当**Q.rear** 值超出队列空间的最大位置时，令**Q.rear= 0** 。使队列空间能“循环”使用。



问题1：如何将队列尾从5变到0？

答：可以利用“取模”运算实现这一跳变。当Q.rear的值为5时，再插入一个元素后，

Q.rear= (5+1)%6 = 0



设队列长度为MAXQSIZE，则

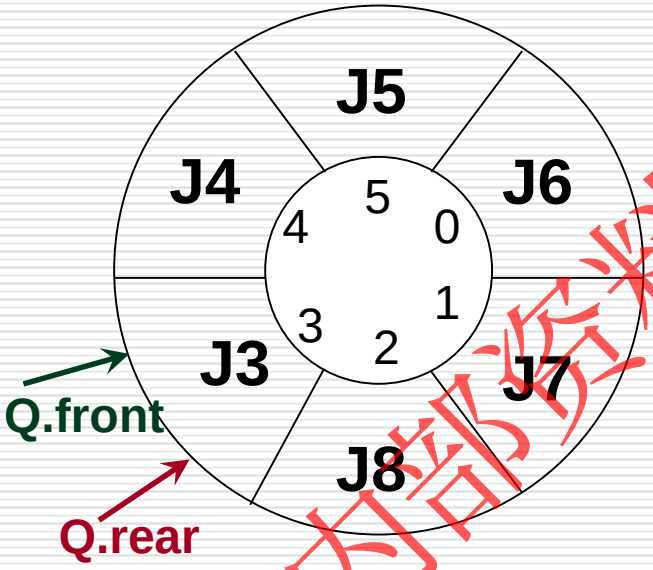
删除元素后：

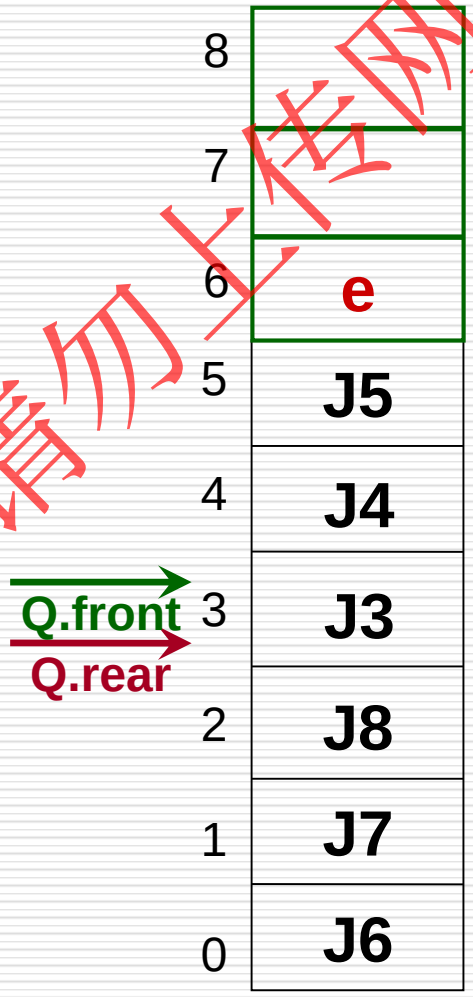
Q.front = (Q.front+1) % MAXQSIZE

问题2：

若队列已满，能否使用realloc函数再分配空间？

答：不能！

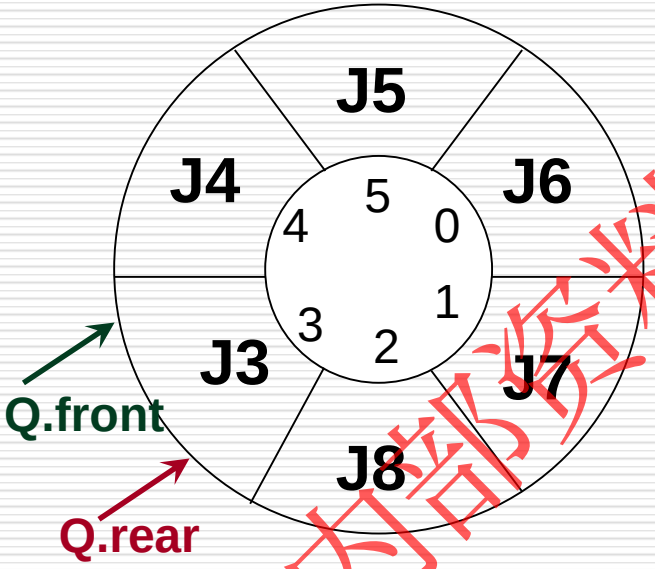


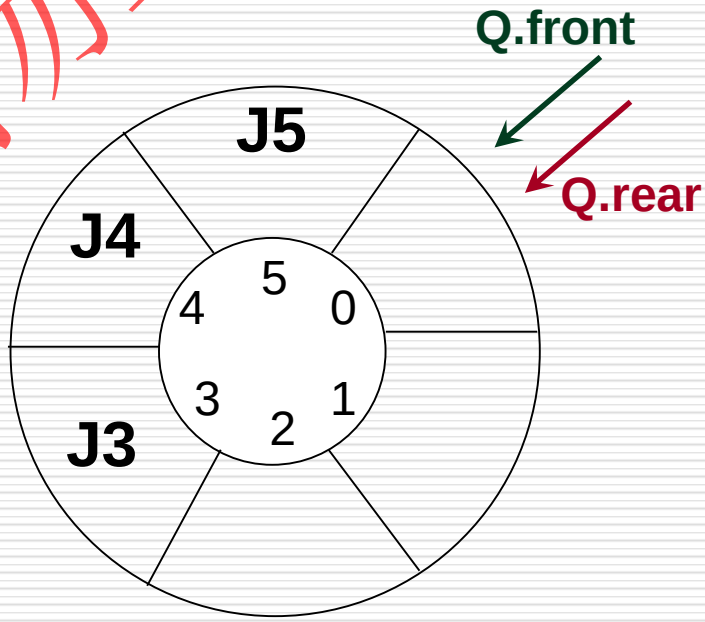


问题3：如何判断循环队列队空、队满？

1）J6、J7、J8依次入队，Q.front==Q.rear;

2）J3、J4、J5依次出队，Q.front==Q.rear;



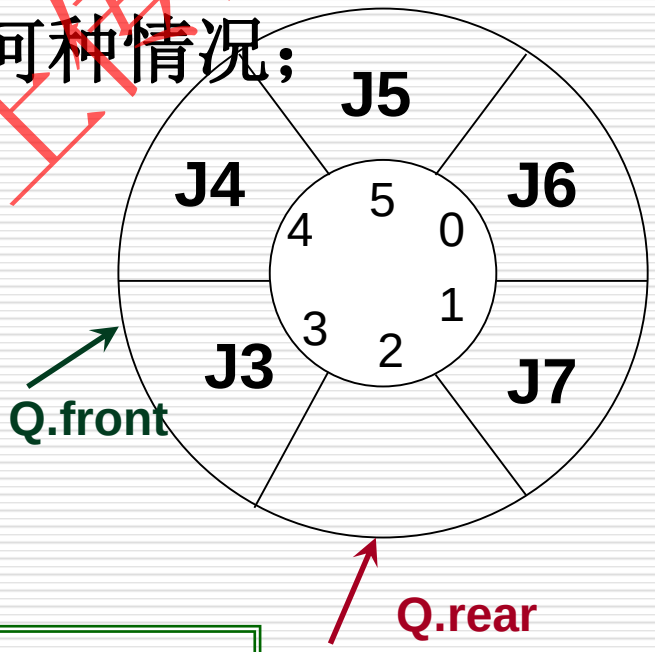


如何区分循环队列队空、队满

① 加设标志位Flag，删除时置0,插入时置1,则可识别当前　Q.front==Q.rear属于何种情况；

②用一个计数器count记录队列中元素个数（队列长度）；

③ 少用一个元素空间。



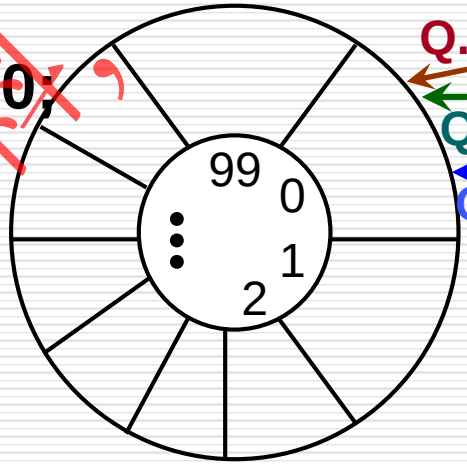
队满特征为：

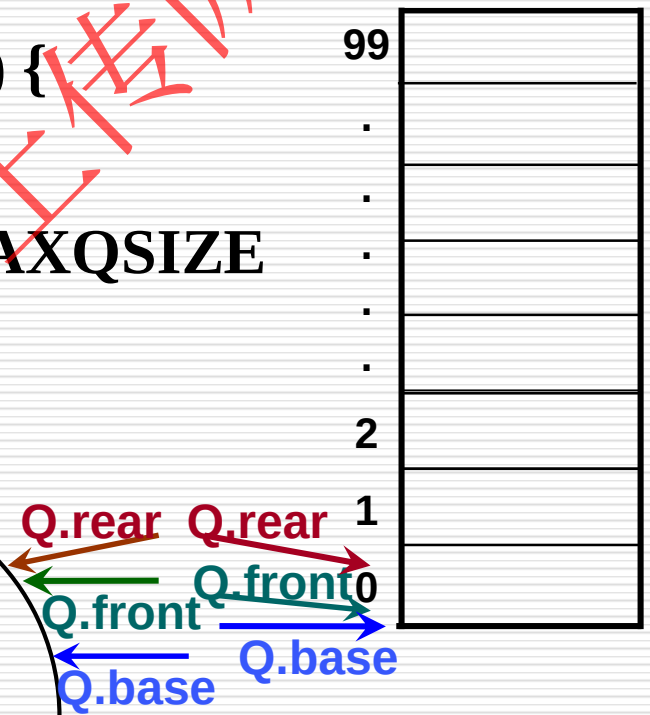
Q.front == (Q.rear+1)%MAXQSIZE

循环队列的基本操作算法

1）初始化操作

```
Status InitQueue_Sq(SqQueue &Q) {
    //构造一个空队列Q
    Q.base=(ElemType * )malloc (MAXQSIZE
*sizeof  (ElemType));
    if (!Q.base) exit (OVERFLOW);
    Q.front = Q.rear=0;
    return OK;
}// InitQueue_Sq
```





2）入队操作

if((Q.rear+1)%MAXQSIZE==Q.front)

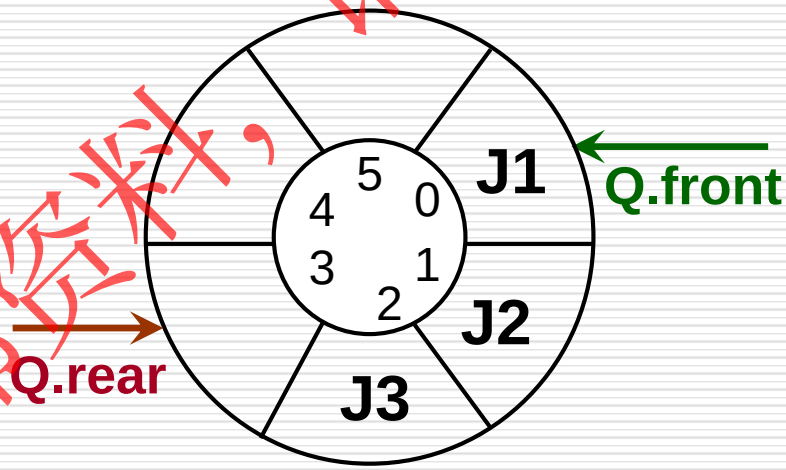
（1）判断Q是否已满， 若满，返回ERROR；否则转（2）；

（2）将元素e 写入队尾；

Q.base[Q.rear] = e

（3）修改队尾指针，使队尾指针指向队尾元素的下一个位置；

Q.rear= (Q.rear+1)%MAXQSIZE



元素e入队前

2）入队操作

if((Q.rear+1)%MAXQSIZE==Q.front)

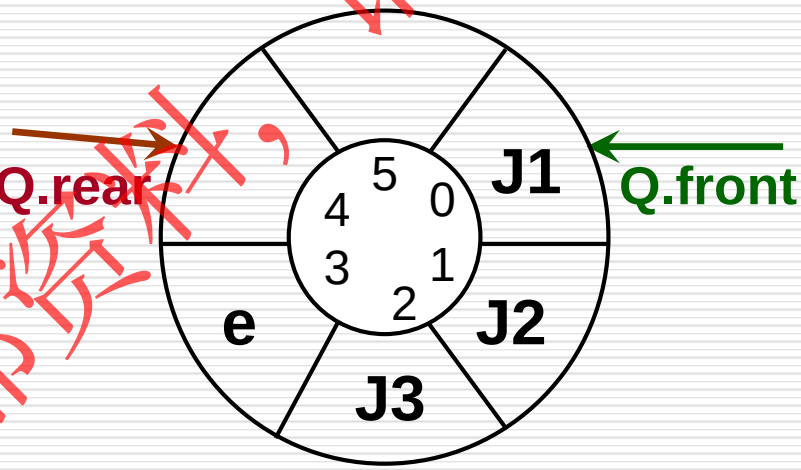
（1）判断Q是否已满， 若满，返回ERROR；否则转（2）；

（2）将元素e 写入队尾；

Q.base[Q.rear] = e

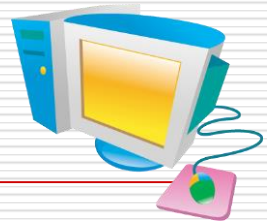
（3）修改队尾指针，使队尾指针指向队尾元素的下一个位置；

Q.rear= (Q.rear+1)%MAXQSIZE



元素e入队后

入队操作算法



```
Status EnQueue_Sq(SqQueue  &Q, QElemType  e ){
  //将元素e插入队尾
     if ((Q.rear+1)%MAXQSIZE= =Q.front) return ERROR ;
     Q.base[Q.rear] = e ;      // 将元素e插入队尾
     Q.rear= (Q.rear+1)%MAXQSIZE;    // 修改队尾指针
     return OK;
}// EnQueue_Sq
```

3）出队操作

if(Q.front= =Q.rear)

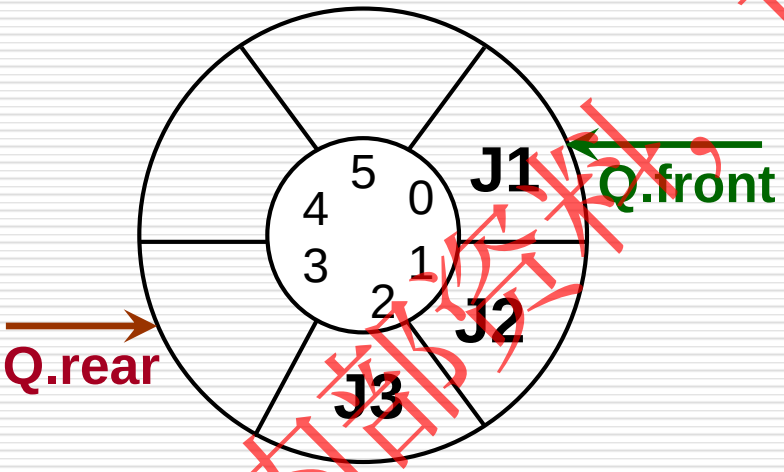
（1）判断Q是否为空，若空，返回ERROR；否则转（2）；

（2）将队头元素赋值给e；

e=Q.base[Q.front]

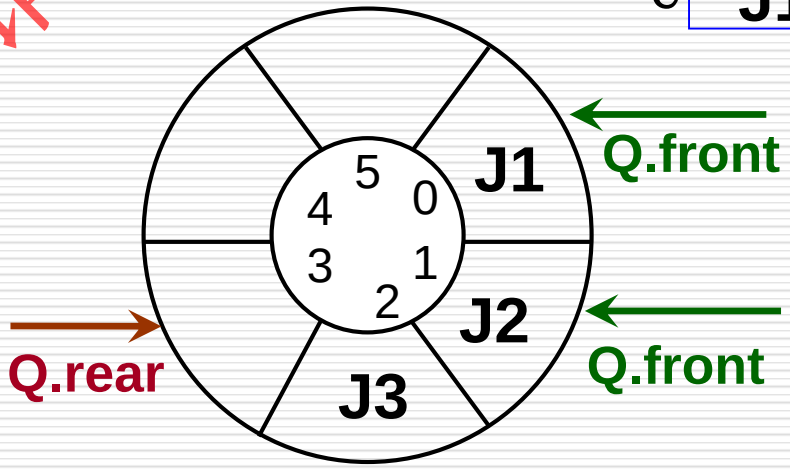
（3）修改队头指针，删除队头元素；

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



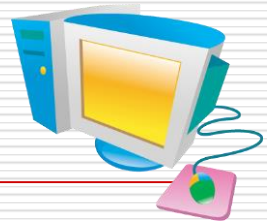
出队操作前

e J1



出队操作后

出队操作算法



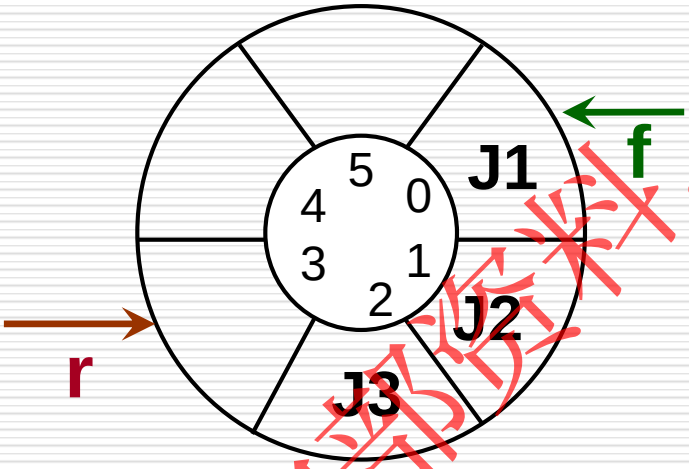
```
Status DeQueue_Sq(SqQueue  &Q, QElemType &e ){
   //若队列不空，删除队头元素，用e返回其值，并返回OK；
   //否则返回ERROR
      if (Q.front= =Q.rear) return ERROR ;
      e =Q.base[Q.front] ;     // 取队头元素的值
      Q.front= (Q.front+1)%MAXQSIZE;    // 修改队头指针
      return OK;
}// DeQueue_Sq
```

例1

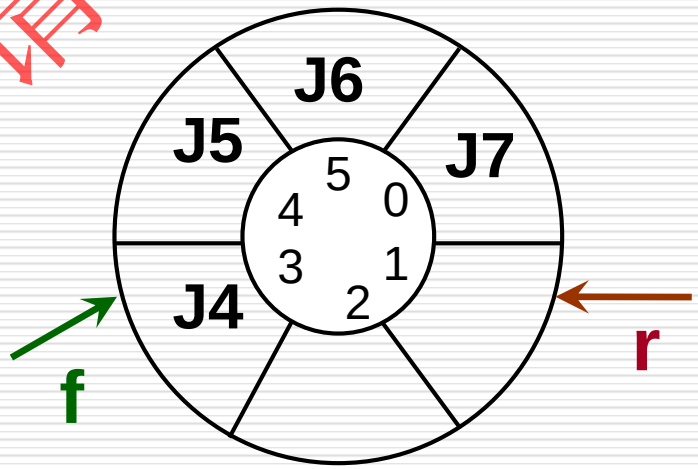
数组Q[n]用来表示一个循环队列，f为当前队列头元素位置，r 为队尾元素的下一位置。假定队列中元素的个数小于n，计算队列中元素的公式为：

（A） r－f　　（B） （f－r+n）% n

（C） r－f+n　　（D） （r－f+n）% n



当 r ≥f 时（A）合理；



当 r < f 时（C）合理。

例1

数组Q[n]用来表示一个循环队列，f为当前队列头元素位置，r 为队尾元素的下一位置。假定队列中元素的个数小于n，计算队列中元素的公式为:

（A） r－f　　（B） （f－r+n）% n

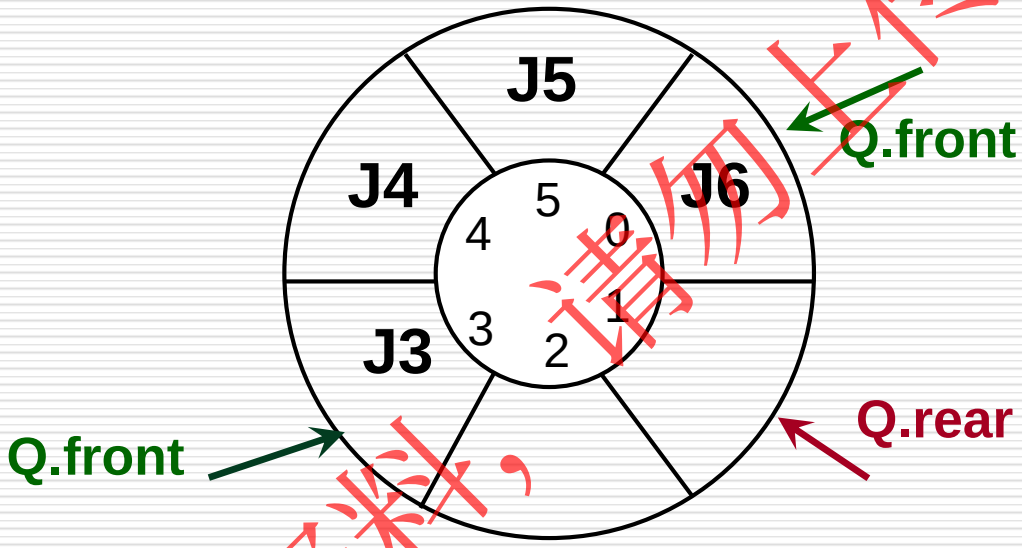
（C）r－f+n　　（D） （r－f+n）% n

循环队列长度（即数据元素个数）：

L=(Q.rear－Q.front＋MAXQSIZE)% MAXQSIZE

例2

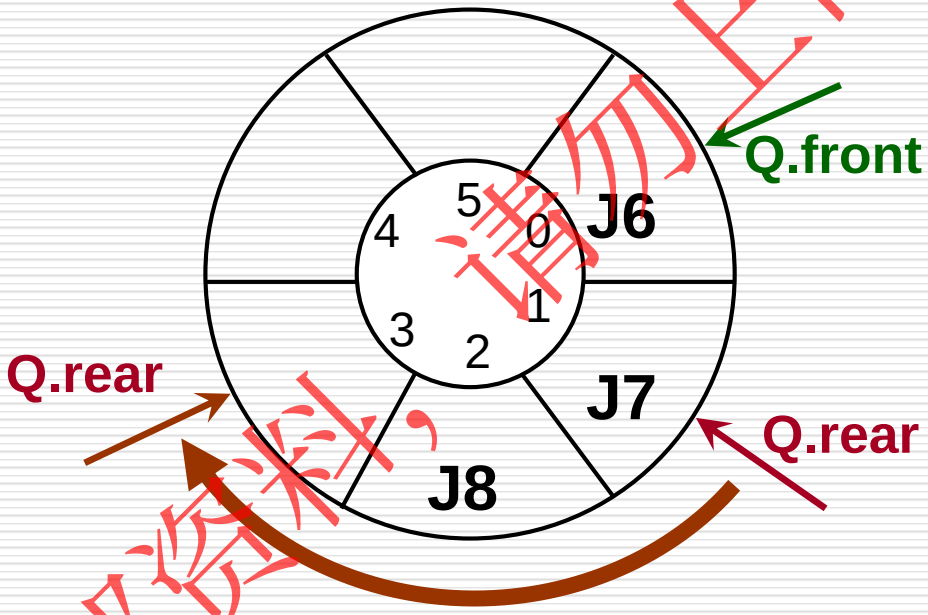
一循环队列如下图所示，若先删除3个元素，再插入2个元素，请问队头和队尾指针分别指向哪个位置？



解：由图可知，队头和队尾指针的初态分别为Q.front=3和Q.rear=1。

删除3个元素后Q.front=(3+3)%6=0；

例2

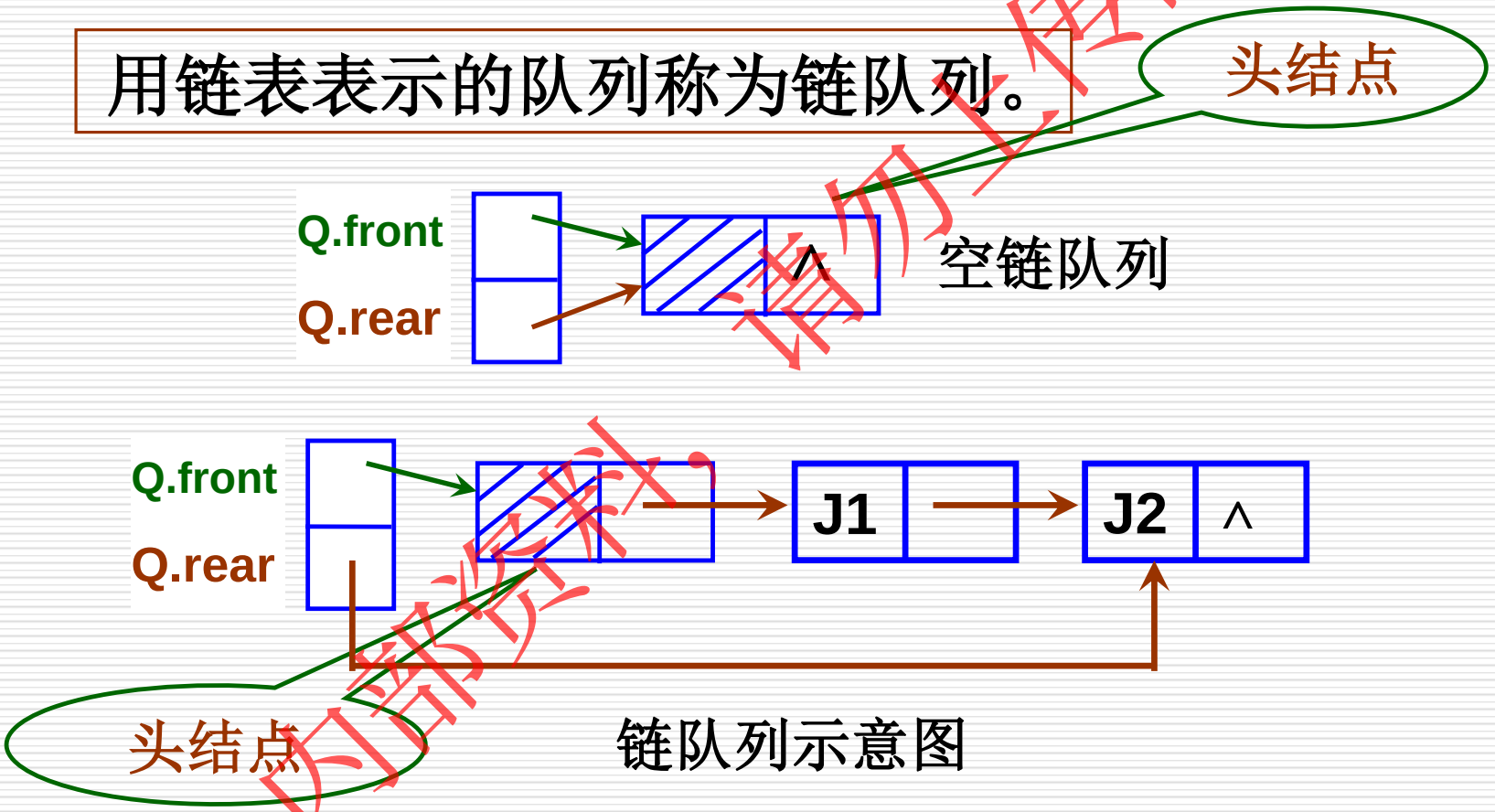


再插入2个元素后，Q.rear=(1+2)%6=3 。

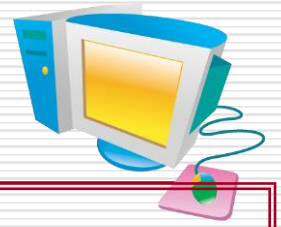
3.5.3 队列的链式存储结构和实现

1. 什么是链队列

用链表表示的队列称为链队列。

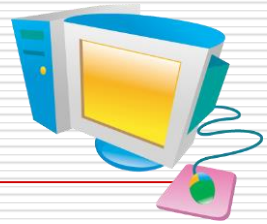


链队列的类型定义



```
typedef struct QNode{
    QElemType  data;      //用于存放队列元素
    struct QNode *next;   //用于存放元素直接后继结点的地址
}QNode,*QueuePtr;  //QNode 类型变量表示链队列的一个结点
                   //QueuePtr：指向QNode的指针类型
typedef struct {
    QueuePtr front;      //队头指针，指向链表的头结点
    QueuePtr rear;       //队尾指针，指向队尾结点
}LinkQueue;              //结构类型, LinkQueue类型的变量用于
                         // 存放队头指针、队尾指针.
```

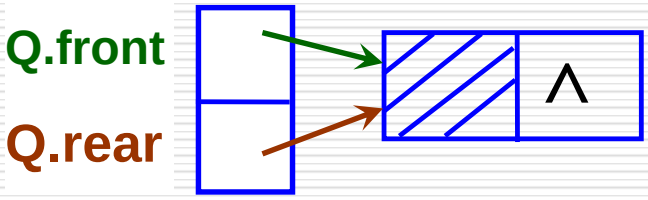
链队列基本操作算法



1）初始化操作

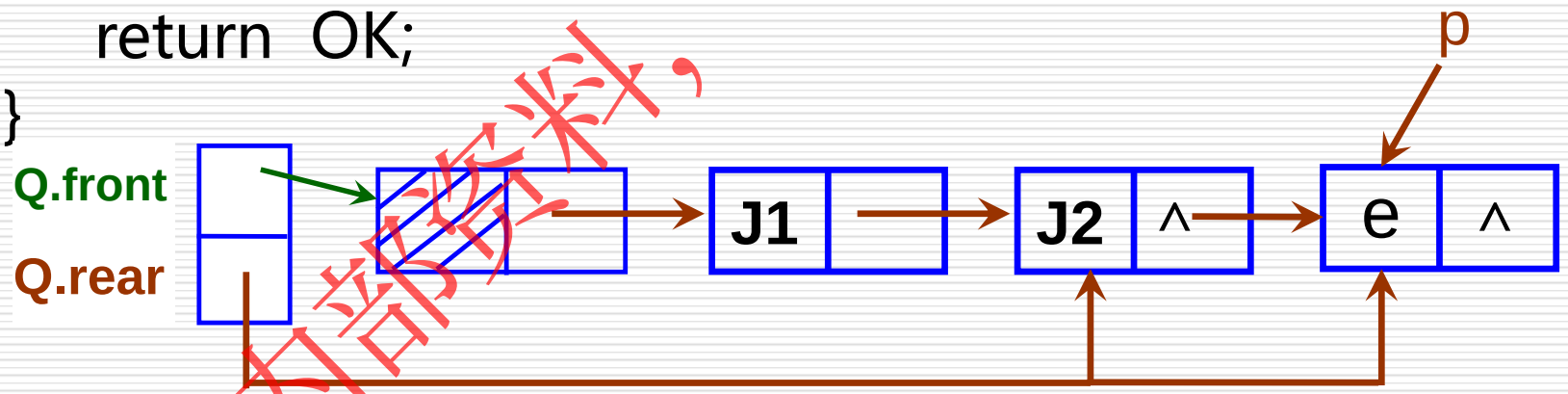
```
Status InitQueue(LinkQueue &Q){
  //建一个空队列Q
  Q.front=Q.rear=new QNode;
  //为链队列的头结点分配空间
  Q.front->next=NULL;
  return OK;
}
```

判空条件：
Q.front= =Q.rear



2）入队操作

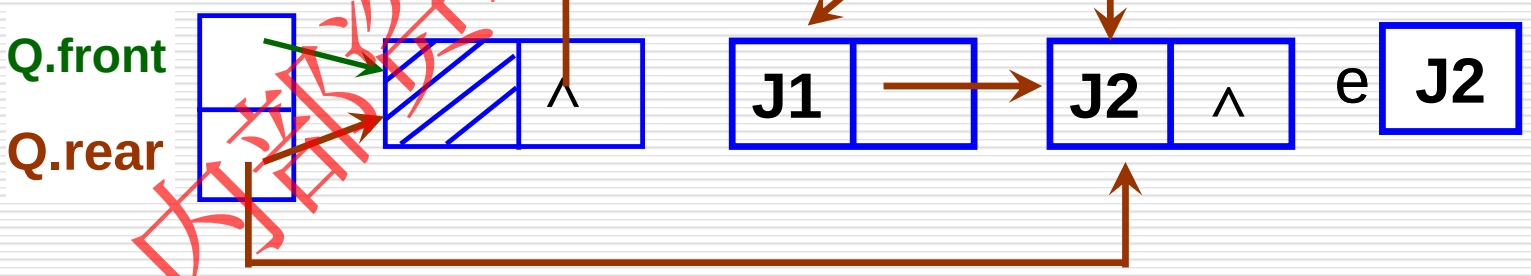
```
Status EnQueue_L(LinkQueue &Q, QElemType e) {
    //插入元素e为新的队尾元素
    p=new QNode;  //为e分配结点
    p->date=e;
    p->next=NULL;
    Q.rear->next=p;          //插入新队尾结点
    Q.rear=p;                //修改队尾指针
    return  OK;
}
```



在链队列Q队尾，插入新的元素e

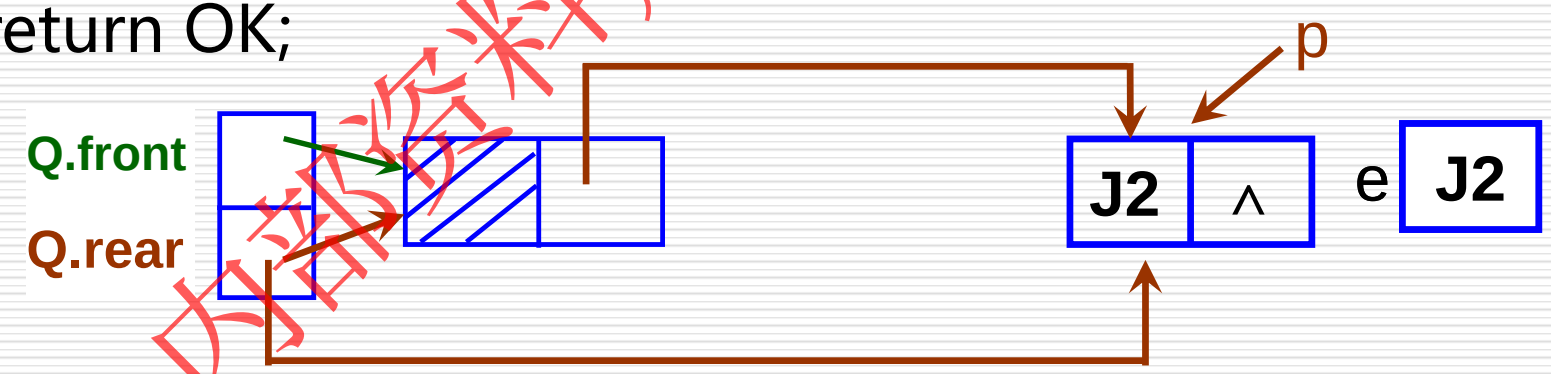
3）出队操作

```
Status DeQueue_L(LinkQueue &Q, QElemType &e) {
  if (Q.front = =Q.rear) return ERROR;  //若队列空返回0
  p=Q.front->next;                      // p指向队头元素
  e=p->data;                            //取队头元素的值
  Q.front->next=p->next;                // 修改链队列头结点指针

  delete p;
  return OK;
}
```

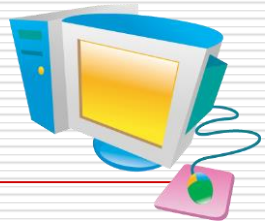


3）出队操作

```
Status DeQueue_L(LinkQueue &Q, QElemType &e) {
  if (Q.front = =Q.rear) return ERROR;  //若队列空返回0
  p=Q.front->next;                      // p指向队头元素
  e=p->data;                            //取队头元素的值
  Q.front->next=p->next;                // 修改链队列头结点指针
  if(Q.rear= =p)Q.rear=Q.front   //只有一个元素结点的情况
                                 //要同时修改队尾指针
  free(p);
  return OK;
}
```



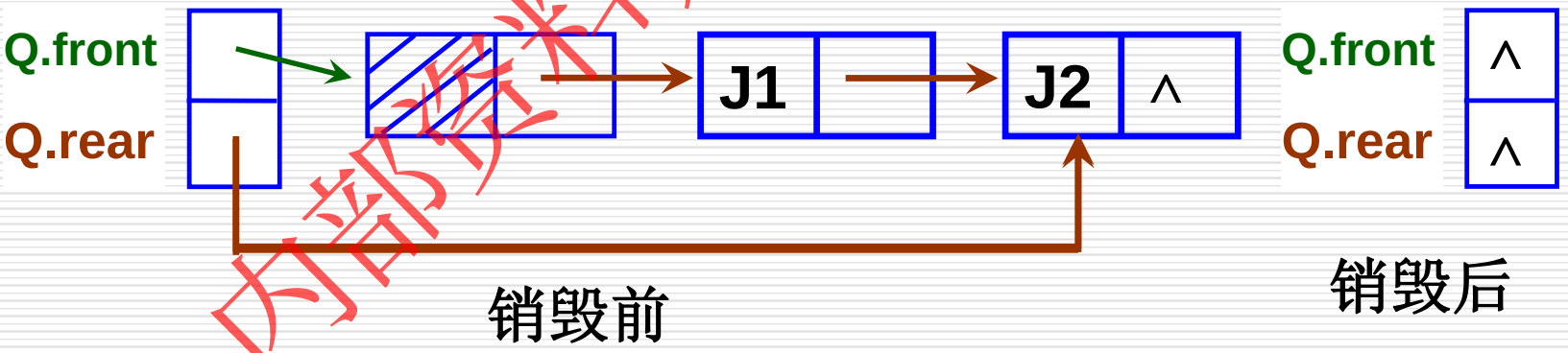
4）取队头元素



```
SElemType GetHead(LinkQueue Q)
{    //返回Q的队头元素，不修改指针
    if(Q.front!=Q.rear)    //队列非空
    return Q.front->next->data;
        //返回队头元素的值，队头指针不变
}//DestroyQueue_L
```

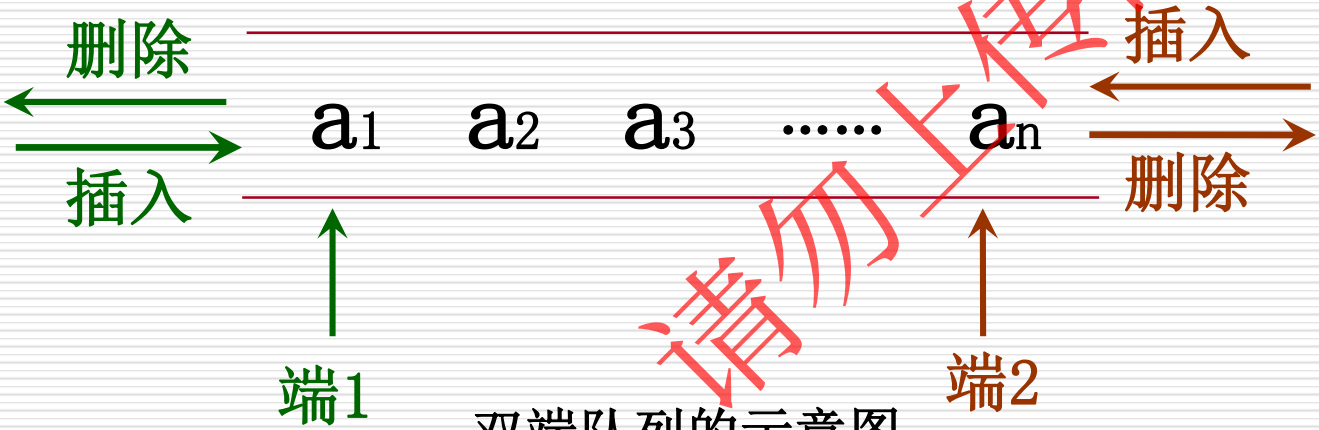
5）销毁操作

```
Status DestroyQueue_L(LinkQueue &Q) {
    //销毁队列Q
    while(Q.front)// 回收链队列的所有元素结点空间
       {   Q.rear=Q.front->next;
           free(Q.front );
           Q.front=Q.rear;}
           return OK;
}//DestroyQueue_L
```



3.5.4 双端队列

双端队列是限定插入和删除操作在两端进行的线性表。



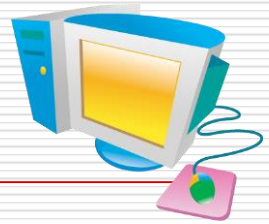
双端队列的示意图

输出受限的双端队列：一个端点允许插入和删除，另一个端点只允许插入；

输入受限的双端队列：一个端点允许插入和删除，另一个端点只允许删除。

如果限定双端队列从某个端点插入的元素只能从该端点删除，则双端队列就蜕变为两个栈底相邻接的栈了。

考研真题



1. 为解决计算机与打印机之间速度不匹配的问题，通常设置一个打印数据缓冲区，主机将要输出的数据依次写入该缓冲区，而打印机则依次从该缓冲区中取出数据。该缓冲区的逻辑结构应该是（ B ）【2009年全国统考】

A. 栈　B. 队列　C. 树　D. 图

2、某队列允许在其两端进行入队操作，但仅允许在一端进行出队操作，若元素a、b、c、d、e、f依次进队列，则不可能得到的顺序是（ C ）【2010年全国统考】

A、bacde　　B、dbace

C、dbcae　　D、ecbad

小结

1　队列是限定仅能在表尾一端进行插入，表头一端删除操作的线性表；

2　队列中的元素具有先进先出的特点；

3　循环队列是本节的重点也是难点；

4　入队操作要修改队尾指针，出队操作要修改队头指针；

3.6 案例分析与实现

案例3.1　数制转换

功能：对于输入的任意一个非负十进制数，显示输出与其等值的八进制数。

求解方法：---除8取余法

以1348为例：

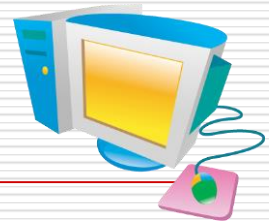
N：十进制数，div：整除运算，mod：求余运算；

N	N div 8	N mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

$$(1348)_{10} = 2\times8^3+5\times8^2+0\times8+4 = (2504)_8$$

可将计算过程中得到的八进制数顺序进栈，出栈序列输出的就是对应的八进制数。

算法3.20



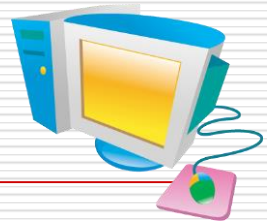
```
void conversion(int N ){
 //对输入的任意非负十进制整数，打印输出其等值的八进制数
      InitStack(S);                    //建空栈
      while(N) {                       // N不等于零循环
            Push(S, N % 8);            // N/8第一个余数进栈
            N=N/8;  }                  //整除运算
      while(! StackEmpty) {            //输出存放在栈中的八制数位。
            Pop(S, e);
            cout<<e;
      }
}//conversion
```

案例3.2： 括号的匹配

算法步骤：

① 初始化一个空栈S。
② 设置一标记性变量flag，用来标记匹配结果以控制循环及返回结果，1表示正确匹配，0表示错误匹配，flag初值为1。

案例3.2：括号的匹配（续）

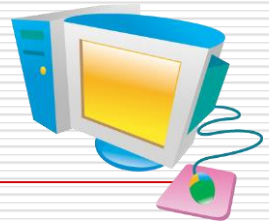


③ 扫描表达式，依次读入字符ch，如果表达式没有扫描完毕或flag非零，则循环执行以下操作：

- 若ch是左括号“[”或“(”，则将其压入栈；
- 若ch是右括号“)”，则根据当前栈顶元素的值分情况考虑：若栈非空且栈顶元素是“(”，则正确匹配，否则错误匹配，flag置为0，退出循环；
- 若ch是右括号“]”，则根据当前栈顶元素的值分情况考虑：若栈非空且栈顶元素是“[”，则正确匹配，否则错误匹配，flag置为0，退出循环。

④ 退出循环后，如果栈空且flag值为1，则匹配成功，返回true，否则返回false。

例2 表达式求值



1）问题的提出

假若我们想在计算机上设计一个小计算器（程序），其功能为：从键盘上输入一个算术表达式（由运算符操作数构成的字符串）,在屏幕上显示输出表达式的求值结果。

2）表达式的构成

操作数、运算符、界符（如括号）

3）表达式的求值：

例　　5+6× (1+2)- 4

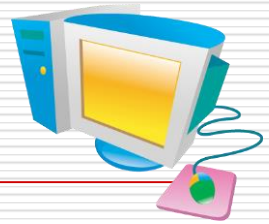
③ ② ① ④

按照四则运算法则，上述表达式的计算过程为：

5+6× (1+2)- 4=5+6×3-4 = 5+18-4= 23-4=19

如何确定运算符的运算顺序？

4）算符优先关系



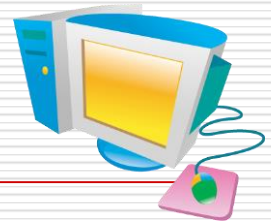
表达式中任何相邻运算符θ_1、θ_2的优先关系有：

$\theta_1<\theta_2$：θ_1的优先级低于θ_2

$\theta_1=\theta_2$：θ_1的优先级等于θ_2

$\theta_1>\theta_2$：θ_1的优先级高于θ_2

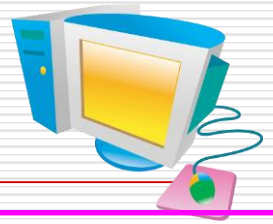
算符间的优先关系表



θ1 \ θ2	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

注：θ1 θ2 是相邻算符，θ1在左，θ2在右

5）算符优先算法



例：5+4× (1+2)- 6

建立两个工作栈：
OPTR栈，保存运算符
OPND栈，保存操作数或运算结果

从左向右扫描表达式：

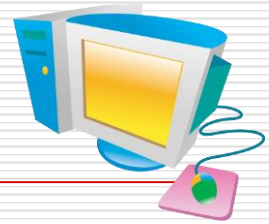
遇操作数——保存；

遇运算符号θ_j——与前面的刚扫描过的运算符θ_i比较

若$\theta_i<\theta_j$ 则保存θ_j，（因此已保存的运算符的优先关系为$\theta_1<\theta_2<\theta_3<\theta_4$。。）

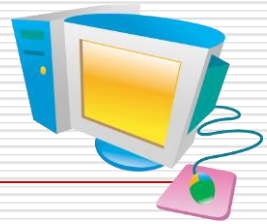
若$\theta_i>\theta_j$ 则说明θ_i是已扫描的运算符中优先级最高者，可进行运算；

若$\theta_i=\theta_j$则θ_i为（，θ_j为），说明括号内的式子已计算完，需要消去括号；



```
operandType EvaluateExpression( ){
  //算术表达式求值的算符优先算法。设OPTR和OPND分别
  //为运算符栈和运算数栈，OP为运算符集合。
  InitStack(OPTR);   Push (OPTR, #);
  InitStack(OPND);   cin>>ch;
  While(ch!=‘ #’   ||  GetTop(OPTR)!=‘#’ ){
       if (! In (ch, OP))
      {Push(OPND, c); cin>>ch}    //不是运算符则进栈
       else                       //In(c, OP)判断c是否是运算符的函数
```

续

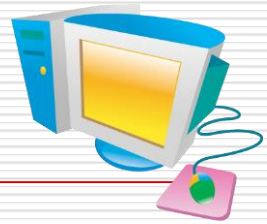


```
switch (Precede(GetTop(OPTR), ch) {
    case '<':     // 新输入的算符c优先级高，c进栈
            Push(OPTR, ch);  cin>>ch;  break;
    case '=':     // 脱括号并接收下一字符
            Pop(OPTR, x);  cin>>ch; break;
  case '>': //新输入的算符c优先级低，即栈顶算符优先权
          // 高，出栈并将运算结果入栈OPND
            Pop(OPTR, theta);
            Pop(OPND, b);  Pop(OPND, a);
            Push(OPND, Operate(a, theta, b));
            break;
```

续

```
        }// switch
    }//while
    return GetTop(OPND);
}//EvaluateExpression
```

后缀表达式



例：用栈实现将中缀表达式8-(3+5)*(5-6/2)转化成后缀表达式。

解：先介绍一种手工转换的方法：

（1）将中缀表达式中所有的子表达式按计算规则用嵌套括号括起来：

(8-((3+5)*(5-(6/2))))

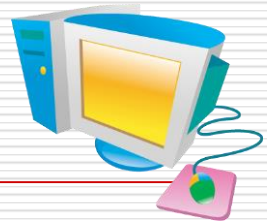
（2）顺序将每对括号中的运算符移到相应括号的后面：

(8((3 5)+(5(6 2)/)-)*)-

（3）删除所有括号：

8 3 5 + 5 6 2 / - * -

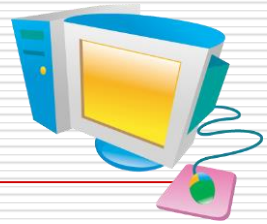
例3. 中缀表达式转换为后缀表达式



中缀表达式str转化为后缀表达式new的算法思想如下：

设操作符栈S，初始为空栈后，压入优先级最低的操作符'#'，对中缀表达式从左向右扫描，遇操作数，直接写入new；若是操作符（记为c），分如下情况处理，直至表达式str扫描完毕。

算法思想



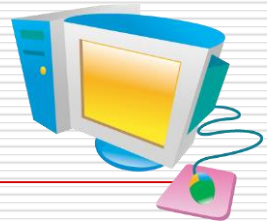
（1）c为一般操作符('+'，'-'，'*'，'/'等)，要与栈顶元素比较优先级，若c优先级高于栈顶操作符，则入栈；否则，栈顶运算符退栈到new，c再与新栈顶操作符作上述比较处理，直至c入栈。

（2）c为左括号'('，c入栈。

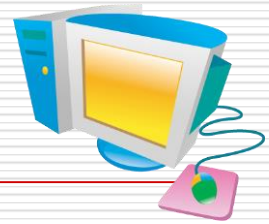
（3）c为右括号')'，操作符退栈并进入new，直到碰到左括号为止，左括号退栈（不能进入new），右括号也去掉，达到new中消除括号的目的。

（4）c为'#'，表示中缀表达式str结束，操作符栈退栈到new，直至碰到'#'，退栈，整个操作结束。

算法描述

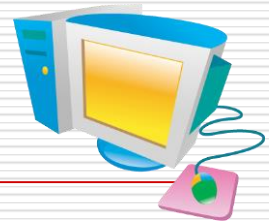


```
void NiBoLan(char *str,char *new)
   //把中缀表达式str转换成逆波兰式new
{  char  *p,*q;
   p=str;q=new; //为方便起见,设str的两端都加上了优先级最低的特殊符号 '#'
   InitStack(S); //S为运算符栈    Push(S,'#');
```



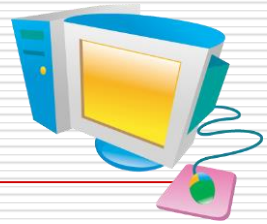
```
while(gettop(S)!=‘#’ || *p!='#')
{   if(*p是字母)) {*q++=*p; p++;}//直接输出
    else {  //若是运算符，则根据优先级决定进栈还是出栈
            c=gettop(S);
            if(*p优先级高于c ) { push(S,*p);  p++; }
            else if(*p优先级等于c) { pop(S,c); p++;}
            else {  while(*p优先级低于c)
                    {  pop(S,c);
*(q++)=c;  c=gettop(S); }                    }//else
     }//else
 }//while
}//NiBoLan
```

例4. 逆波兰式求值



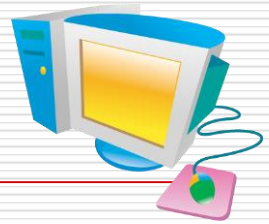
根据上题的假设条件，试写一算法，对以逆波兰式表示的表达式求值。

```
int GetValue_NiBoLan(char *str)//对逆波兰式求值{
  char *p;
  SqStack S;
  p=str;
  InitStack(s); //s为操作数栈
```



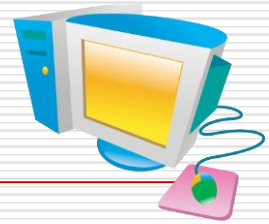
```
while(*p)
  { if(*p是数)    push(s,*p);
    else   {  pop(s,a);  pop(s,b);   r=compute(b,*p,a);
              //假设compute为执行双目运算的过程
              push(s,r);
              }//else
    p++;
  }//while
  pop(s,r);
return r;
}//GetValue_NiBoLan
```

案例3.4： 舞伴问题



【案例分析】

- ✓ 设置两个队列分别存放男士和女士入队者
- ✓ 假设男士和女士的记录存放在一个数组中作为输入，然后依次扫描该数组的各元素，并根据性别来决定是进入男队还是女队。
- ✓ 当这两个队列构造完成之后，依次将两队当前的队头元素出队来配成舞伴，直至某队列变空为止。
- ✓ 此时，若某队仍有等待配对者，则输出此队列中排在队头的等待者的姓名，此人将是下一轮舞曲开始时第一个可获得舞伴的人。



```
//- - - - - 跳舞者个人信息- - - - -
typedef struct {
   char name[20];   //姓名
   char sex;            //性别，'F'表示女性，'M'表示男性
}Person;
//- - - - - 队列的顺序存储结构- - - - -
#define MAXQSIZE 100   //队列可能达到的最大长度
typedef struct {
   Person *base;   //队列中数据元素类型为Person
   int front;                  //头指针
   int rear;                   //尾指针
}SqQueue;
SqQueue Mdancers,Fdancers;   //分别存放男士和女士入队者队列
```

【算法步骤】

① 初始化Mdancers队列和Fdancers队列。

② 反复循环，依次将跳舞者根据其性别插入Mdancers队列或Fdancers队列。

③ 当Mdancers队列和Fdancers队列均为非空时，反复循环，依次输出男女舞伴的姓名。

④ 如果Mdancers队列为空而Fdancers队列非空，则输出Fdancers队列的队头女士的姓名。

⑤ 如果Fdancers队列为空而Mdancers队列非空，则输出Mdancers队列的队头男士的姓名。

【例】汽车加油站

结构：入口和出口为单行道，

加油车道若干条n每辆车加油

都要经过三段路程，三个队列

- 入口处排队等候进入加油车道
- 在加油车道排队等候加油
- 出口处排队等候离开

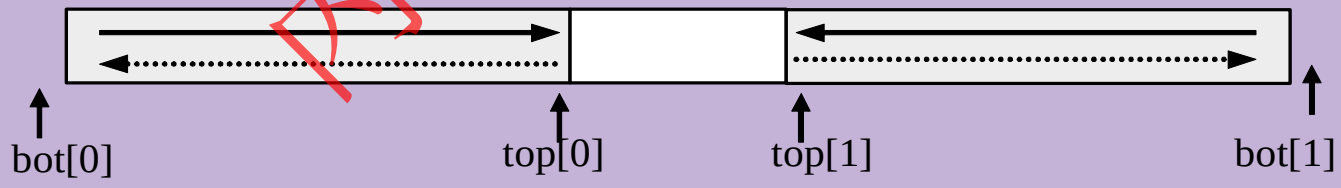
若用算法模拟，需要设置n+2个队列。

【例】模拟打印机缓冲区

- ✓ 在主机将数据输出到打印机时，主机速度与打印机的打印速度不匹配
- ✓ 为打印机设置一个**打印数据缓冲区**，当主机需要打印数据时，先将数据依次写入缓冲区，写满后主机转去做其他的事情
- ✓ 而打印机就从缓冲区中按照**先进先出**的原则依次读取数据并打印

（1）将编号为0和1的两个栈存放于一个数组空间V[m]中，栈底分别处于数组的两端。当第0号栈的栈顶指针top[0]等于-1时该栈为空；当第1号栈的栈顶指针top[1]等于*m*时，该栈为空。两个栈均从两端向中间增长。试编写双栈初始化，判断栈空、栈满、进栈和出栈等算法的函数。双栈数据结构的定义如下：

```
typedef struct{
  int top[2], bot[2];  //栈顶和栈底指针
   SElemType *V;           //栈数组
  int m;                 //栈最大可容纳元素个数
}DblStack;
```



Back

栈、队列与一般线性表的区别

栈、队列是一种特殊（操作受限）的线性表
区别：仅在于运算规则不同

一般线性表

逻辑结构：一对一
存储结构：顺序表、链表
运算规则：随机、顺序存取

栈

逻辑结构：一对一
存储结构：顺序栈、链栈
运算规则：后进先出

队列

逻辑结构：一对一
存储结构：顺序队、链队
运算规则：先进先出

3.7本章小结

1. 掌握栈和队列的特点，并能在相应的应用问题中正确选用；
2. 熟练掌握栈的顺序栈和链栈的进栈出栈算法，特别应注意栈满和栈空的条件；
3. 熟练掌握循环队列和链队列的进队出队算法，特别注意队满和队空的条件；
4. 理解递归算法执行过程中栈的状态变化过程；
5. 掌握表达式求值算法。

第三章　结束

Thank you!