

# 第三章 栈和队列

# 教学目标



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**两种存储结构**的基本操作实现算法，特别注意**栈满和栈空**的条件
3. 熟练掌握**循环队列和链队列**的基本操作实现算法，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**表达式求值方法**

# 教学内容

3.1 栈和队列的定义和特点

3.2 案例引入

3.3 栈的表示和操作的实现

3.4 栈与递归

3.5 队列的表示和操作的实现

3.6 案例分析与实现

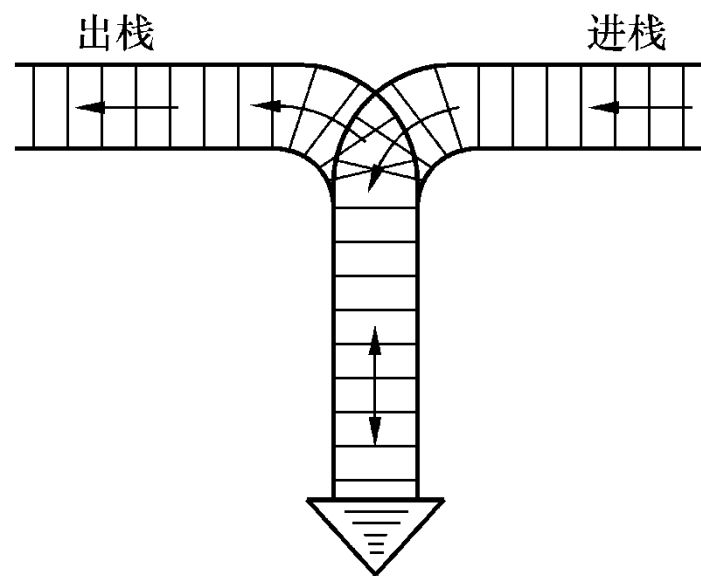
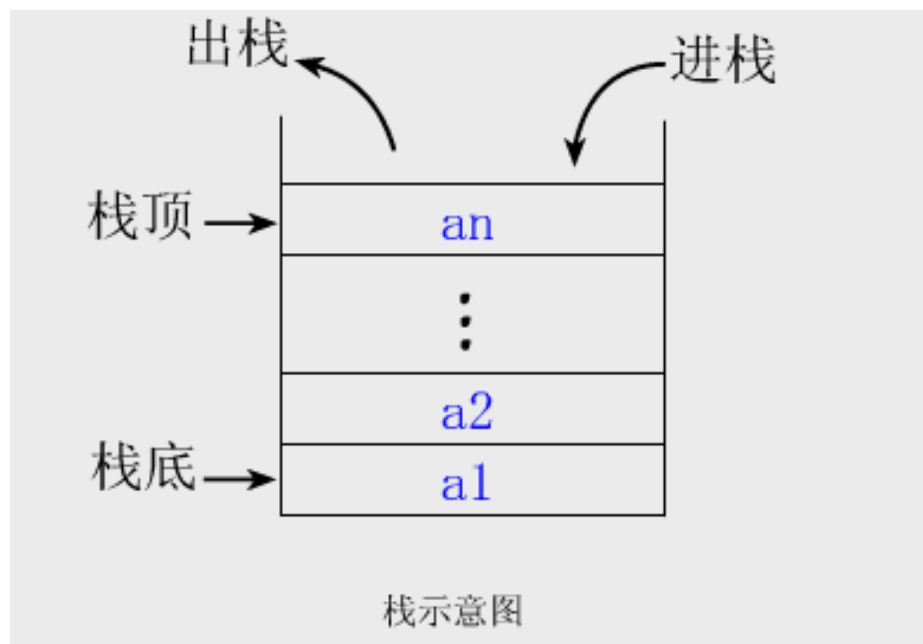
## 栈 (Stack)

1. 定义
2. 逻辑结构
3. 存储结构
4. 运算规则
5. 实现方式

## 队列 (Queue)

1. 定义
2. 逻辑结构
3. 存储结构
4. 运算规则
5. 实现方式

# 栈



用铁路调度站表示栈

# 3.1 栈和队列的定义和特点



## 栈

1. 定义 只能在表的一端（栈顶）进行插入和删除运算的线性表
2. 逻辑结构 与线性表相同，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）或**先进后出**（FILO）的原则

## 5. 实现方式

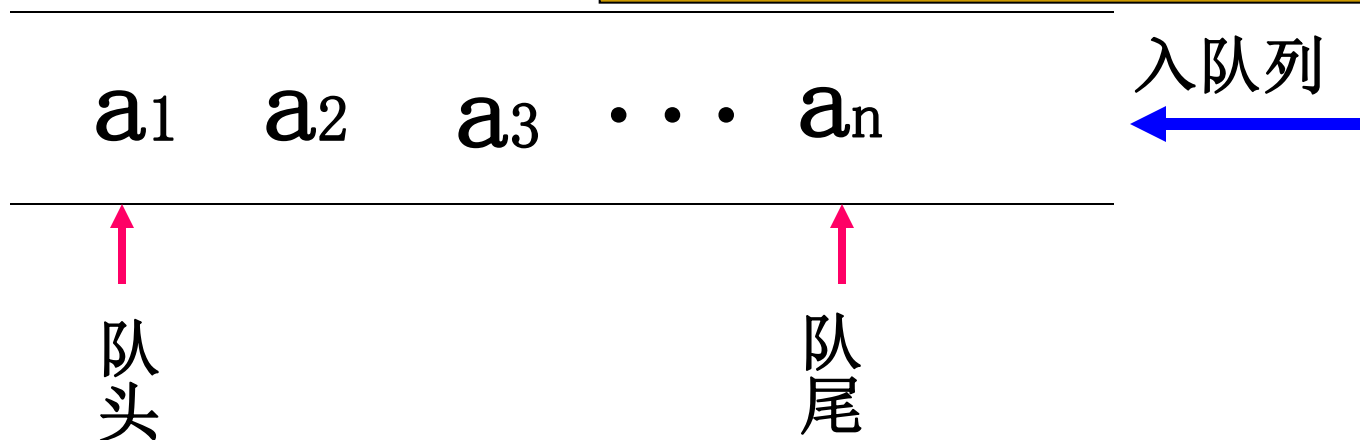
关键是编写**入栈**和**出栈**函数，具体实现依顺序栈或链栈的不同而不同

基本操作有**入栈**、**出栈**、**读栈顶元素值**、**建栈**、**判断栈满**、**栈空**等

队列是一种先进先出(FIFO) 的线性表. 在表一端插入,在另一端删除



$$q = (a_1, a_2, \dots, a_n)$$





$$q = (a_1, a_2, \dots, a_n)$$



出队列  
←

**a<sub>1</sub>**

a<sub>2</sub>

a<sub>3</sub>

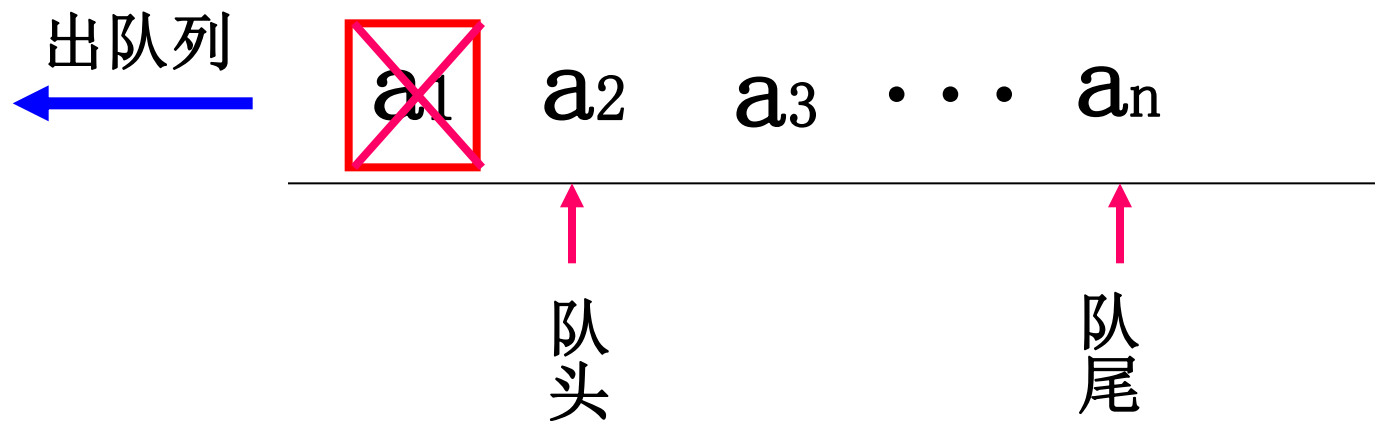
...

a<sub>n</sub>

↑  
队头

↑  
队尾

$$q = (a_1, a_2, \cdots a_n)$$



$$q = (a_1, a_2, \cdots a_n)$$

出队列  
←

~~$a_1$~~

~~$a_2$~~

$a_3$

$\cdots$

$a_n$

↑  
队头

↑  
队尾

# 3.1 栈和队列的定义和特点



## 队列

1. 定义 只能在表的一端（队尾）进行插入，在另一端（队头）进行删除运算的线性表
2. 逻辑结构 与线性表相同，仍为一对一关系
3. 存储结构 用顺序队列或链队存储均可

## 4. 运算规则

先进先出 (FIFO)

## 5. 实现方式

关键是编写入队和出队函数，具体实现依顺序队或链队的不同而不同

# 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**

## 3.2 案例引入



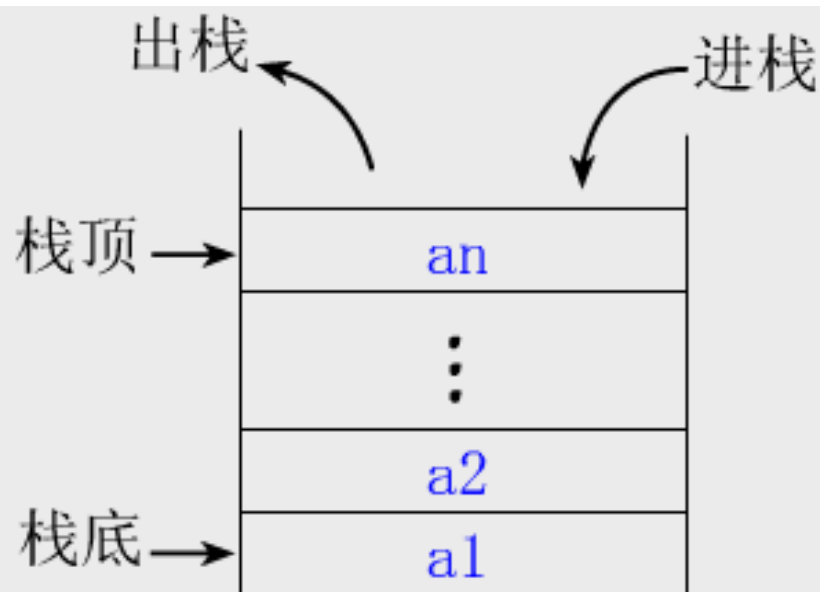
案例3.1：数制的转换

案例3.2：括号匹配的检验

案例3.3：表达式求值

案例3.4：舞伴问题

## 3.3 栈的表示和操作的实现



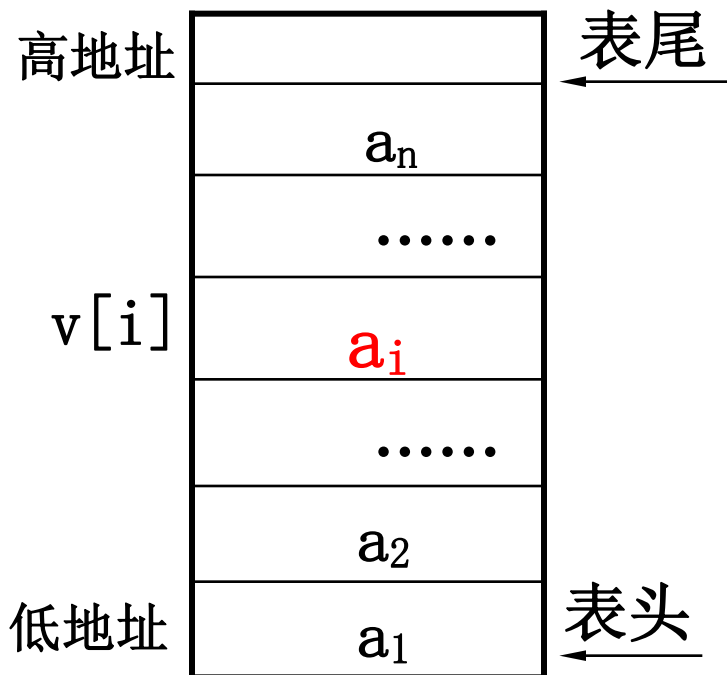
栈示意图

“进” = 压入 = PUSH ( )  
“出” = 弹出 = POP ( )



# 顺序栈与顺序表

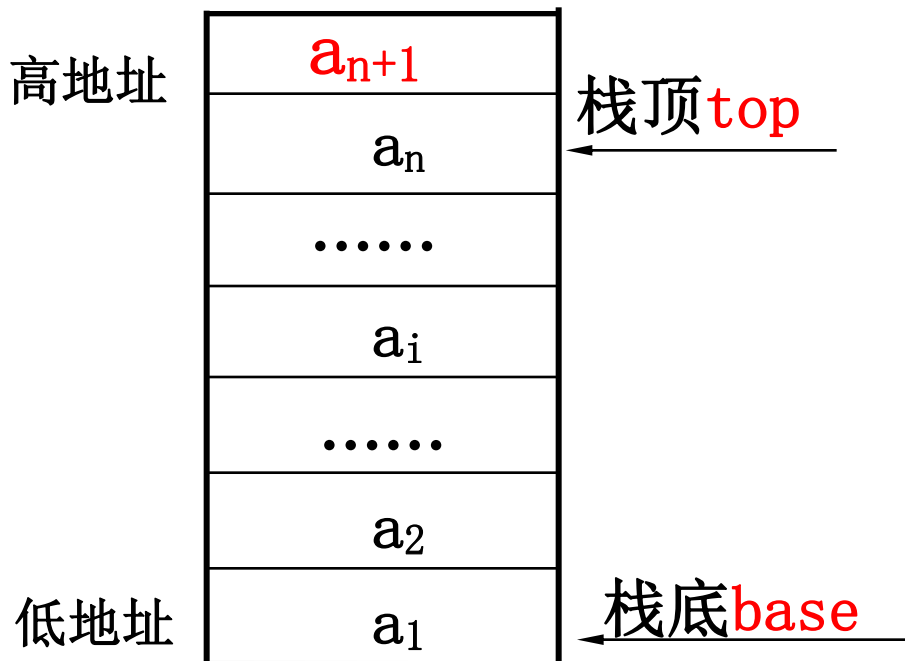
顺序表V[n]



写入:  $v[i] = a_i$

读出:  $x = v[i]$

顺序栈S

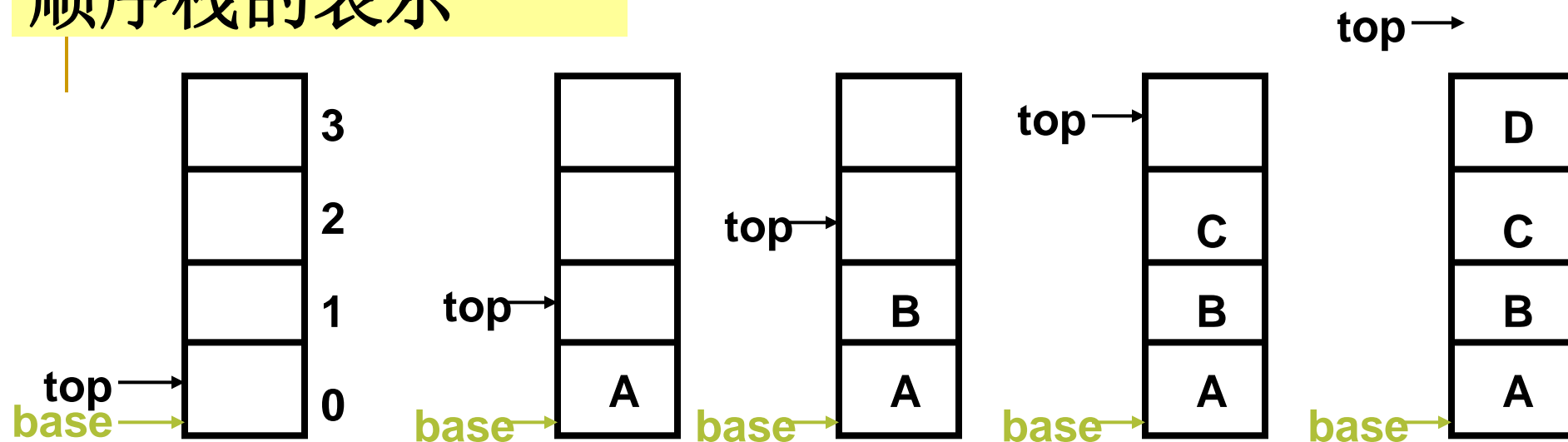


压入: PUSH ( $a_{n+1}$ )

弹出: POP ( $x$ )

前提: 一定要预设栈顶指针top!

# 顺序栈的表示



空栈

$\text{base} == \text{top}$  是  
栈空标志

$\text{stacksize} = 4$

top 指示真正的栈顶元素之上的下标地址  
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。

# 顺序栈的表示

---

```
#define MAXSIZE 100
typedef struct
{
    SElemType *base;
    SElemType *top;
    int stacksize;
}SqStack;
```

---

# 顺序栈初始化

- 构造一个空栈

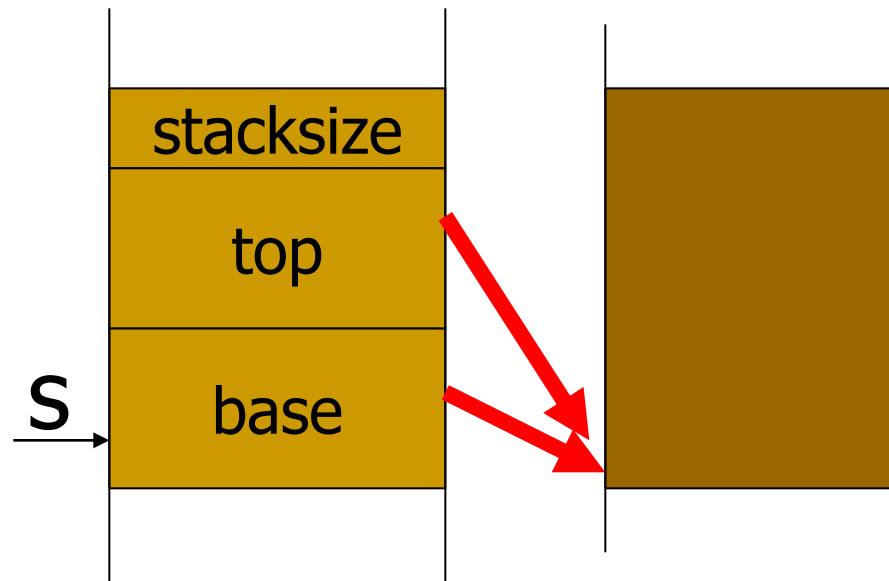
- 步骤:

(1) 分配空间并检查空间是否分配失败，若失败则返回错误

(2) 设置栈底和栈顶指针

**$S.top = S.base;$**

(3) 设置栈大小



# 顺序栈初始化

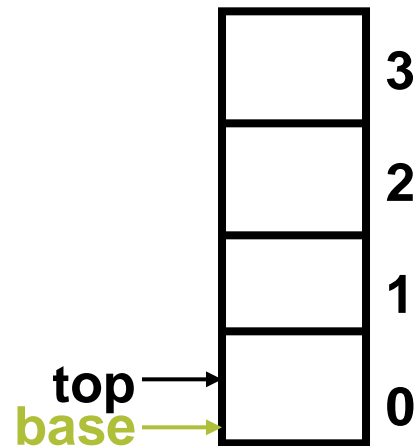
---

```
Status InitStack( SqStack &S )  
{  
    S.base = new SElemType[MAXSIZE];  
    if( !S.base )    return OVERFLOW;  
    S.top = S.base;  
    S.stackSize = MAXSIZE;  
    return OK;  
}
```

---

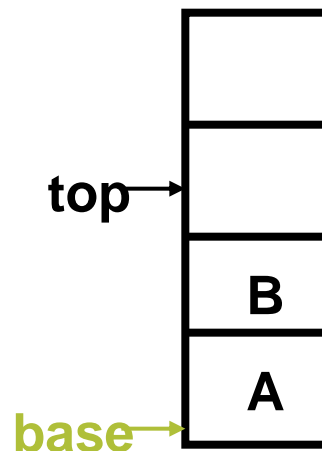
# 判断顺序栈是否为空

```
bool StackEmpty( SqStack S )  
{  
    if(S.top == S.base) return true;  
    else return false;  
}
```



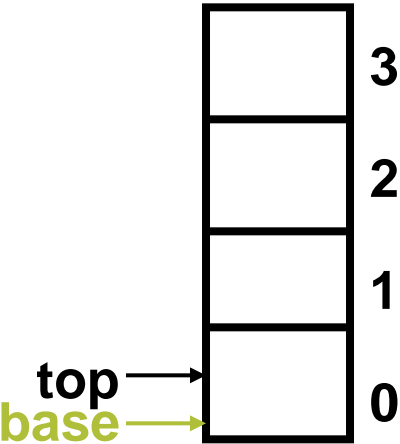
# 求顺序栈的长度

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



# 清空顺序栈

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





# 销毁顺序栈

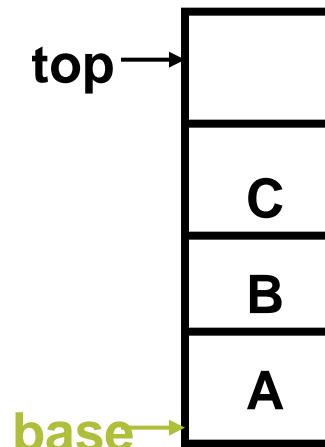
---

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

---

# 顺序栈进栈

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



```
Status Push( SqStack &S, SElemType e)
```

```
{
```

```
    if( S.top - S.base == S.stacksize ) // 栈满
```

```
        return ERROR;
```

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

```
    return OK;
```

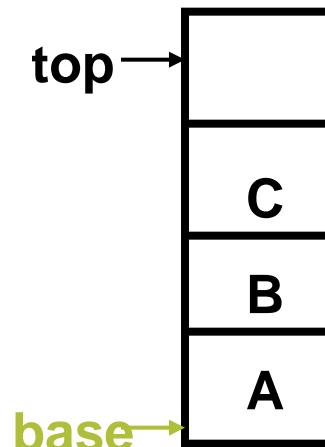
```
}
```

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

**S.top++;**

# 顺序栈出栈

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



```
Status Pop( SqStack &S, SElemType &e)
```

```
{
```

```
    if( S.top == S.base ) // 栈空
```

```
        return ERROR;
```

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

```
    return OK;
```

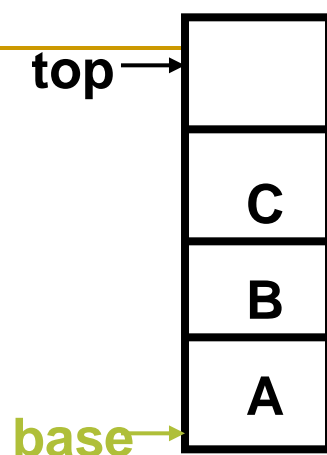
```
}
```

--S.top;

e=\*S.top;

## 取顺序栈栈顶元素

- (1) 判断是否空栈，若空则返回错误
- (2) 否则通过栈顶指针获取栈顶元素



```
Status GetTop( SqStack S, SElemType &e)
```

```
{
```

```
    if( S.top == S.base )    return ERROR; // 栈空
```

```
    e = *( S.top - 1 );
```

```
    return OK;
```

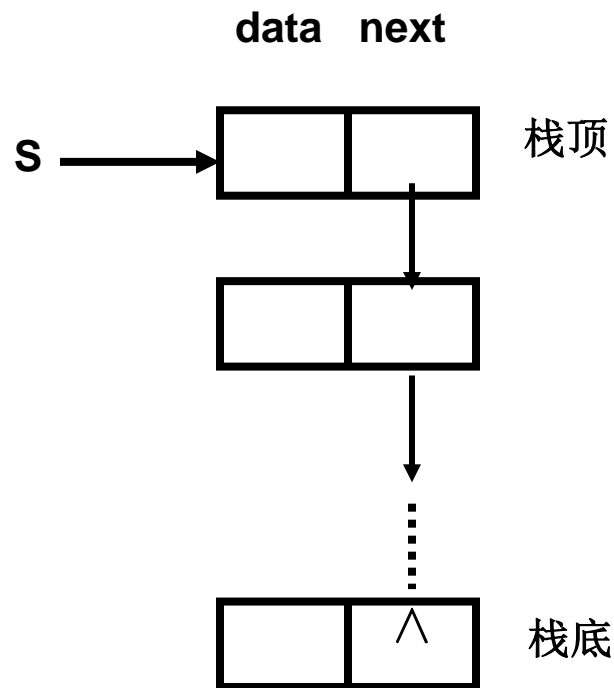
```
}
```

**e = \*( S.top -- );**  
**???**

# 链栈的表示

- ✓ 运算是受限的单链表，只能在链表头部进行操作，故没有必要附加头结点。栈顶指针就是链表的头指针

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



# 链栈的初始化

---

**S**  $\longrightarrow$   $\wedge$

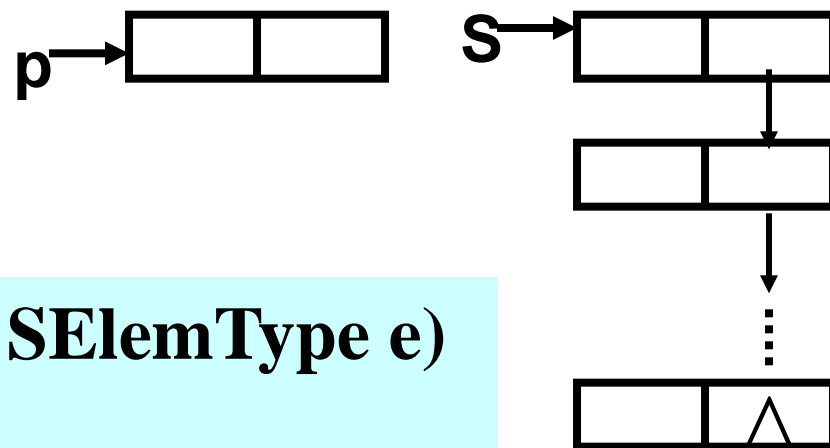
```
void InitStack(LinkStack &S )  
{  
    S=NULL;  
}
```

# 判断链栈是否为空

---

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

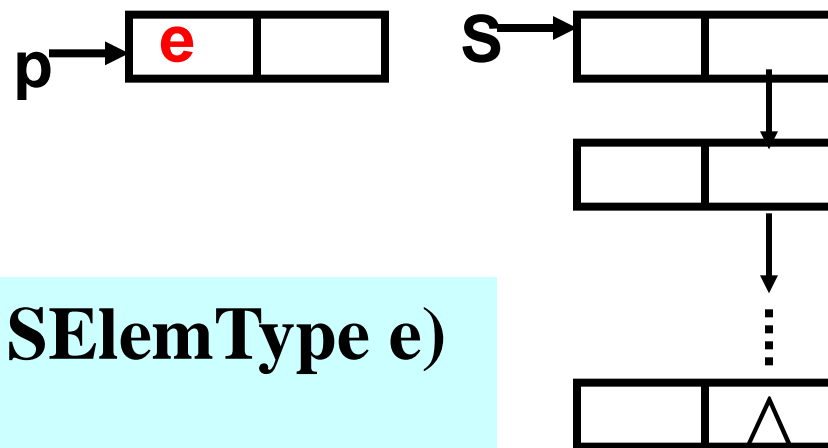
# 链栈进栈



```
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; }
```

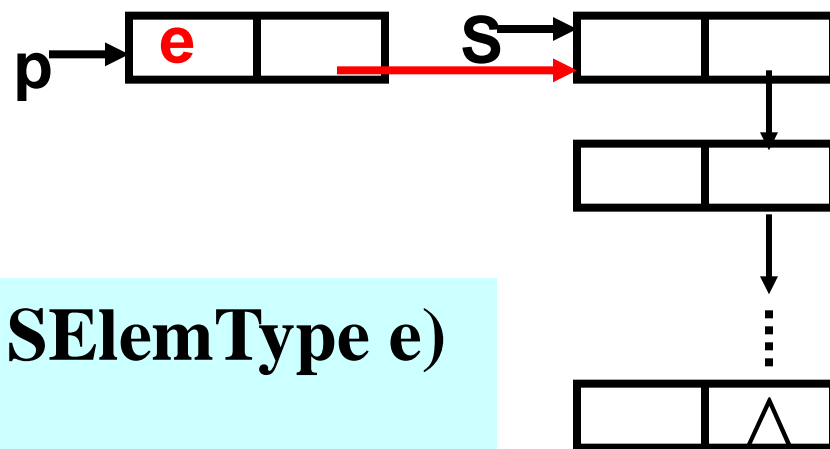


# 链栈进栈



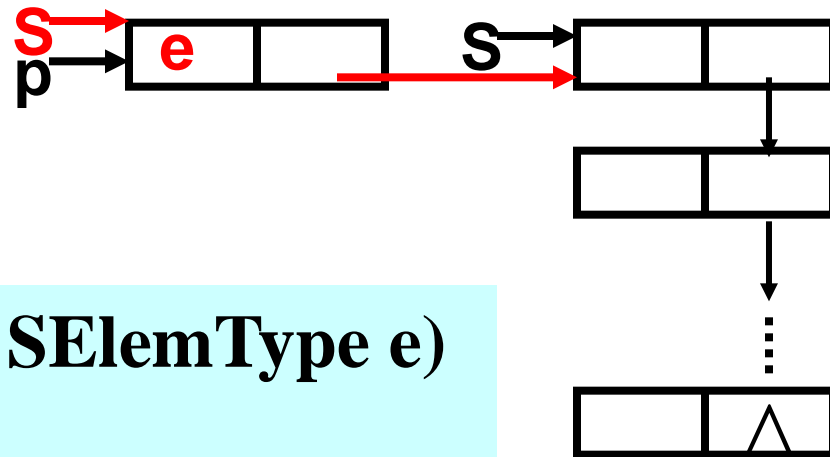
```
Status Push(LinkStack &S , SElemType e)
{
    p=new StackNode;    //生成新结点p
    if (!p) exit(OVERFLOW);
    p->data=e; p->next=S; S=p;
    return OK; }
```

# 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; }
```

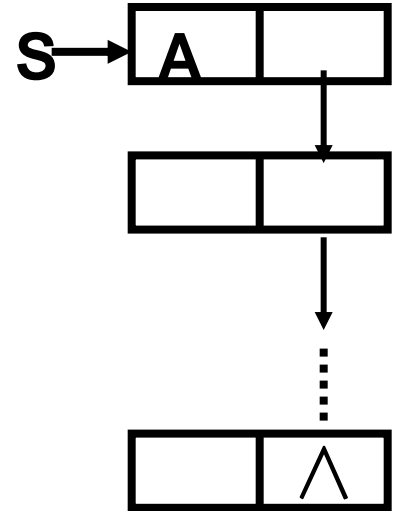
# 链栈进栈



```
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)**

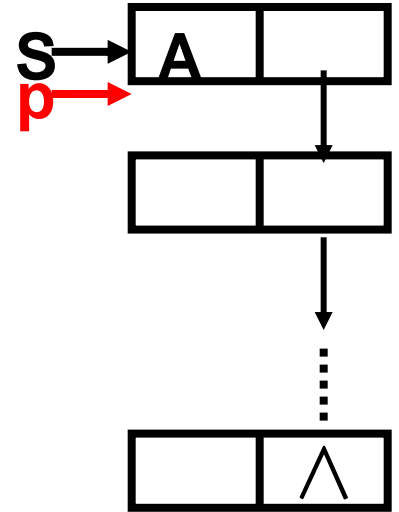
**{if (S==NULL) return ERROR;**

**$e = S \rightarrow data;$   $p = S;$   $S = S \rightarrow next;$**

**delete p; return OK; }**

# 链栈出栈

$e = 'A'$



**Status Pop (LinkStack &S,SElemType &e)**

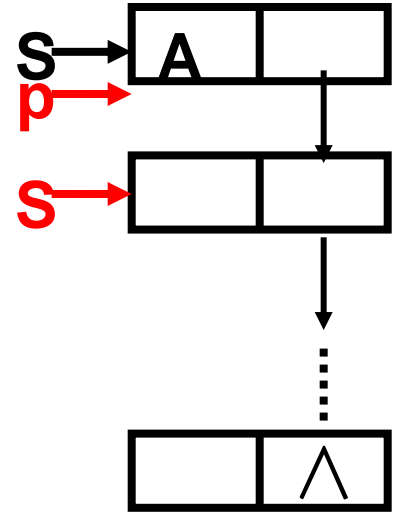
**{if (S==NULL) return ERROR;**

**e = S-> data;  $p = S$ ; S = S-> next;**

**delete p; return OK; }**

# 链栈出栈

$e = 'A'$



**Status Pop (LinkStack &S,SElemType &e)**

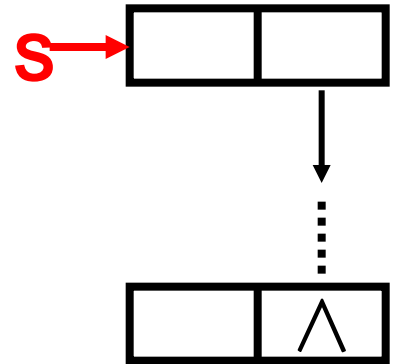
**{if (S==NULL) return ERROR;**

**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; }**

# 取链栈栈顶元素

---

**SElemType GetTop(LinkStack S)**

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

---



## 3.4 栈与递归

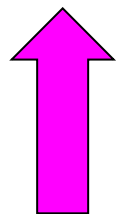


- 递归的定义 若一个对象部分地包含它自己，或用它自己给自己定义，则称这个对象是递归的；若一个过程直接地或间接地调用自己，则称这个过程是递归的过程。

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

当多个函数构成嵌套调用时，遵循

后调用先返回



栈

## ■ 以下三种情况常常用到递归方法

- 递归定义的数学函数
- 具有递归特性的数据结构
- 可递归求解的问题

# 用分治法求解递归问题

**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  
}
```

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

求解阶乘  $n!$  的过程 — `if ( n == 0 ) return 1;`  
`else return n * Fact (n-1);`



# 汉诺塔

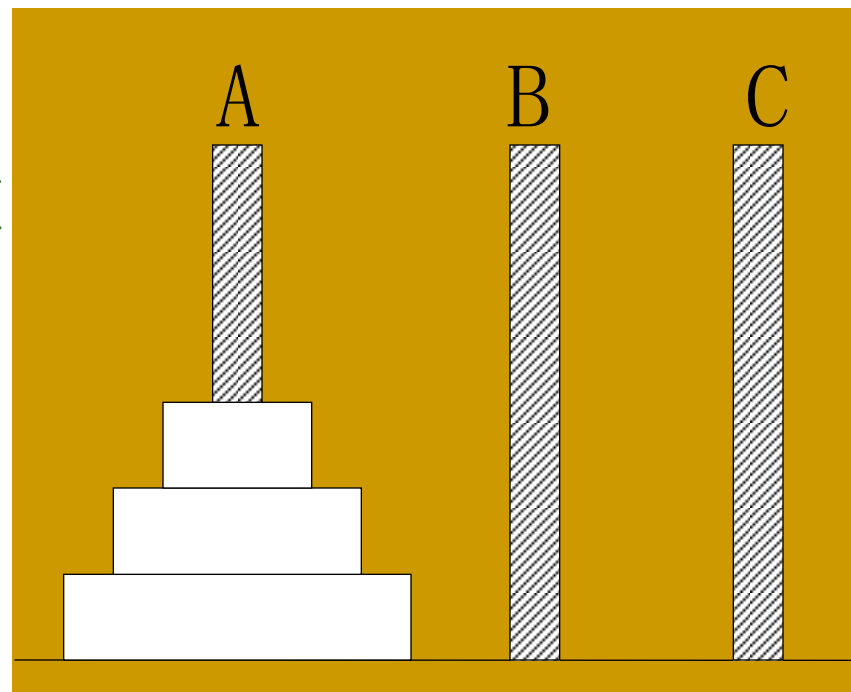


在印度圣庙里，一块黄铜板上插着三根宝石针。  
主神梵天在创造世界时，在其中一根针上穿好了由大到小的64片金片，这就是汉诺塔。  
僧侣不停移动这些金片，一次只移动一片，小片必在大片上面。  
当所有的金片都移到另外一个针上时，世界将会灭亡。

# Hanoi塔问题

规则：

- (1) 每次只能移动一个圆盘
- (2) 圆盘可以插在A, B和C中的任一塔座上
- (3) 任何时刻不可将较大圆盘压在较小圆盘之上





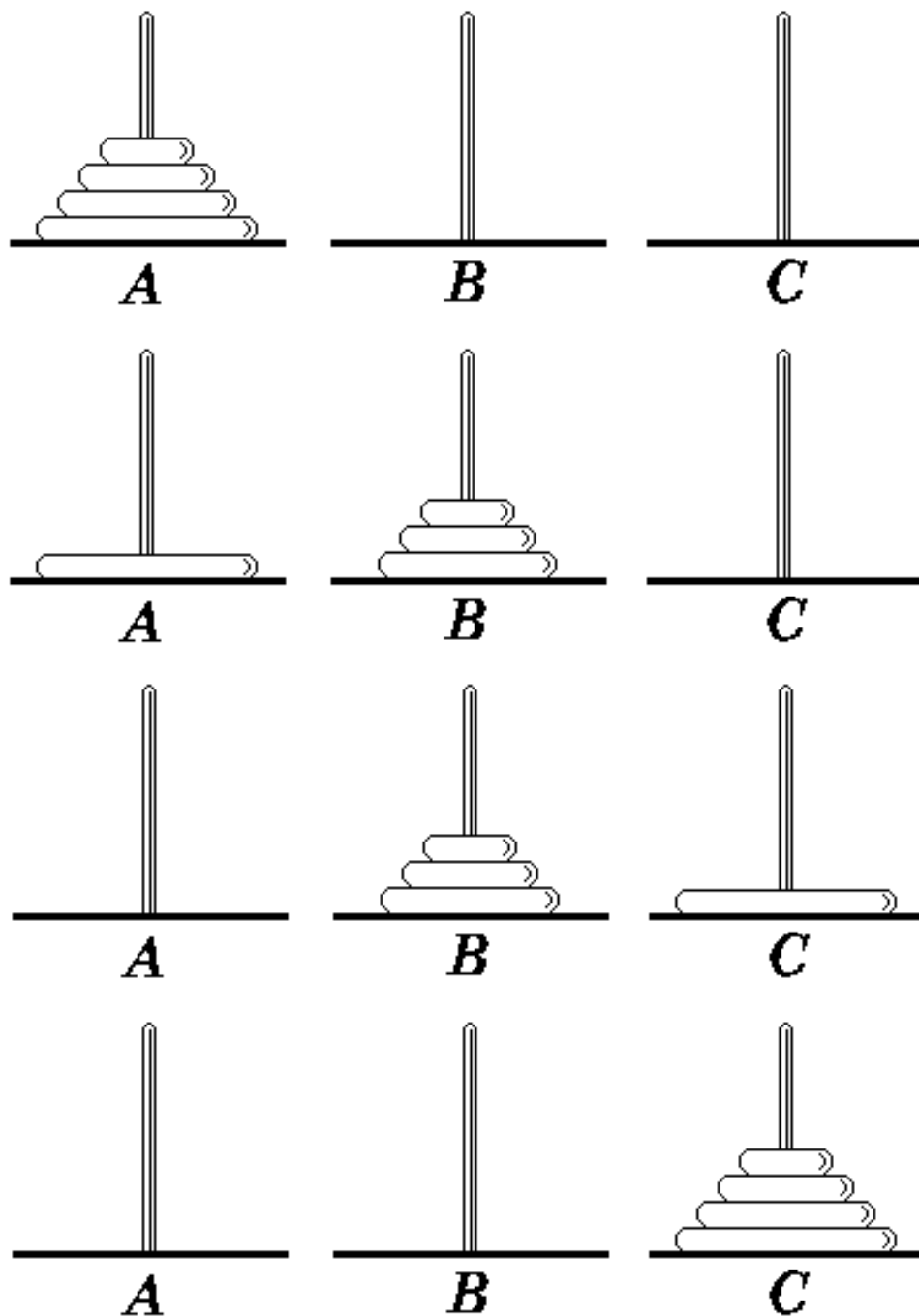
# Hanoi塔问题

$n = 1$ ，则直接从 A 移到 C。否则

(1) 用 C 柱做过渡，将 A 的  $(n-1)$  个移到 B

(2) 将 A 最后一个直接移到 C

(3) 用 A 做过渡，将 B 的  $(n-1)$  个移到 C



跟踪程序，给出下列程序的运行结果，以深刻地理解递归的调用和返回过程

```
#include<iostream.h>

int c=0;

void move(char x,int n,char z)
{cout<<"+c<<","<<n<<","<<x<<","<<z<<endl;}

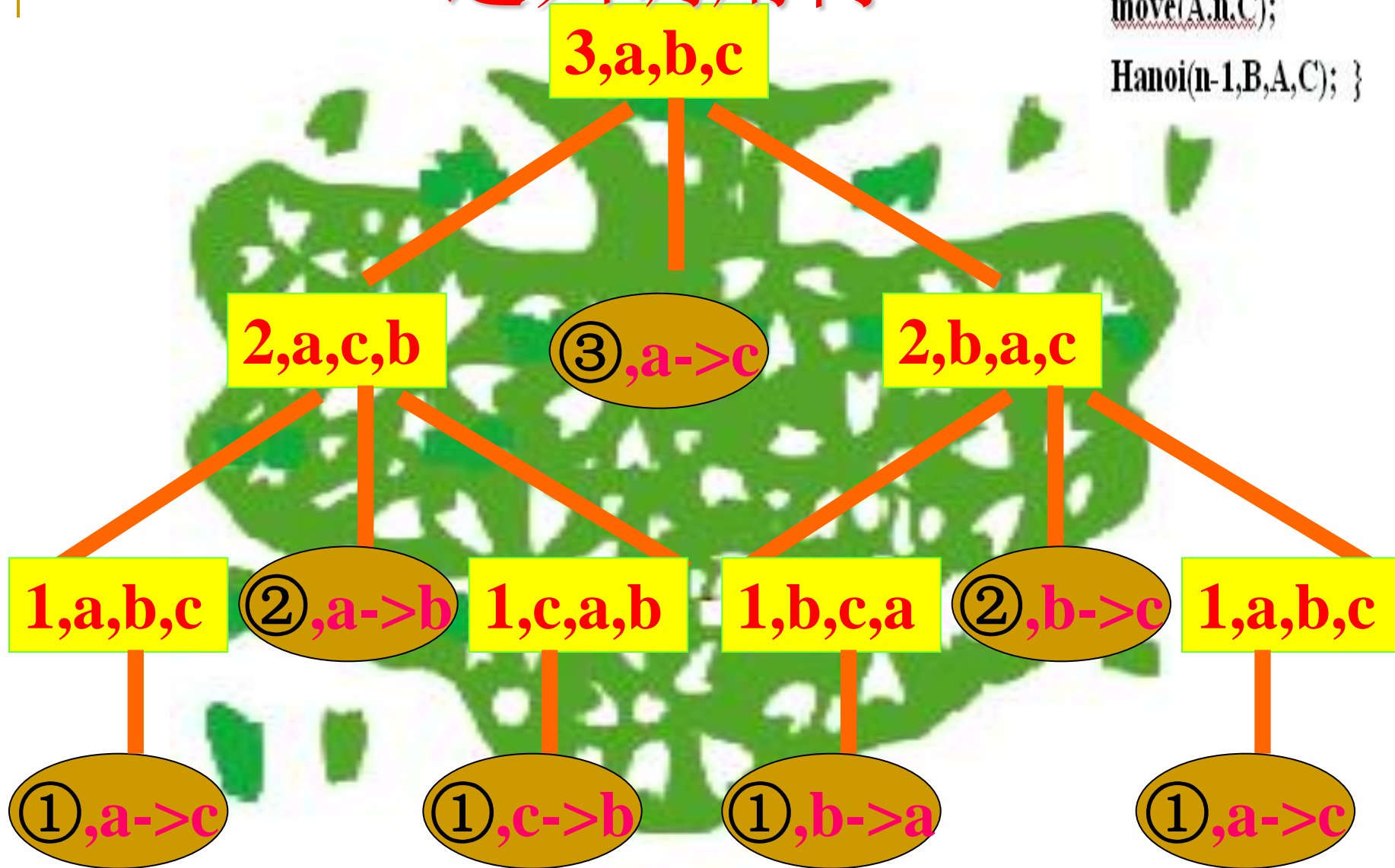
void Hanoi(int n,char A,char B,char C)
{ if(n==1) move(A,1,C);
  else
  {Hanoi(n-1,A,C,B);
   move(A,n,C);
   Hanoi(n-1,B,A,C); }}

void main(){Hanoi(3,'a','b','c');}
```

```
1,1,a,c
2,2,a,b
3,1,c,b
4,3,a,c
5,1,b,a
6,2,b,c
7,1,a,c
```

# 递归调用树

```
{Hanoi(n-1,A,C,B);  
  move(A,n,C);  
  Hanoi(n-1,B,A,C); }
```



# 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

# 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”

“工作记录”  实际参数, 局部变量, 返回地址

# 递归算法的效率分析

空间效率

与递归树的深度成正比

$O(n)$

时间效率

与递归树的结点数成正比



64片金片移动次数： $2^{64}-1=18446744073709551615$

假如每秒钟一次，共需多长时间呢？

一年大约有**31536926**秒，移完这些金片需要**5 8 0 0**多亿年

世界、梵塔、庙宇和众生都已经灰飞烟灭 .....

# 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/空间开销大。

递归→非递归



# 递归→非递归

---

(1) 尾递归、单向递归→循环结构

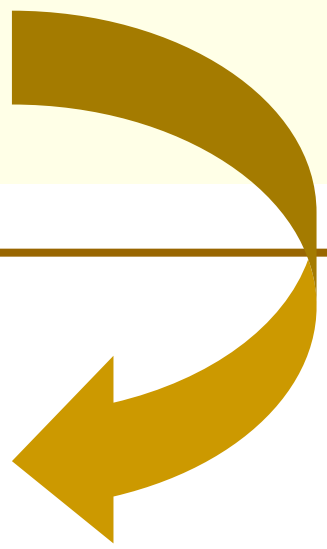
(2) 自用栈模拟系统的运行时栈

---

# 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**。

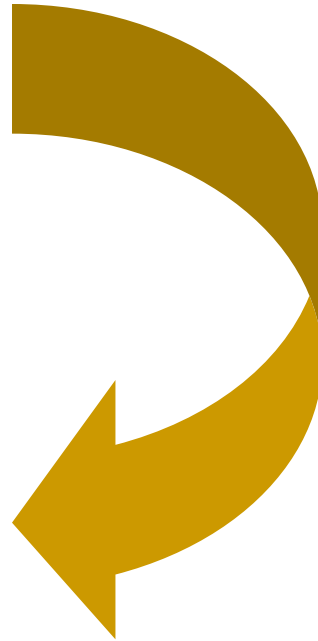
```
long Fib ( long n ) { // Fibonacci数列  
    if(n==1 || n==2) return 1;  
    else return Fib (n-1)+ Fib (n-2);}
```

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

# 尾递归、单向递归→循环结构

```
long Fib ( long n ) {  
    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（了解）

- （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。

## 3.5 队列的表示和操作的实现



### 队列的抽象数据类型

**ADT Queue {**

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

数据关系:  $R_1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i = 1, 2, \dots, n \}$

基本操作: 约定 $a_1$ 端为队列头, $a_n$ 端为队列尾

- (1) **InitQueue (&Q)**      //构造空队列
- (2) **DestroyQueue (&Q)**    //销毁队列
- (3) **ClearQueue (&S)**      //清空队列
- (4) **QueueEmpty(S)**        //判空. 空--TRUE,

# 队列的抽象数据类型

(5) **QueueLength(Q)**           //取队列长度

(6) **GetHead (Q,&e)**           //取队头元素,

(7) **EnQueue (&Q,e)**       //入队列

(8) **DeQueue (&Q,&e)**       //出队列

(9) **QueueTraverse(Q,visit())**   //遍历

**}ADT Queue**

# 队列的顺序表示—— 用一维数组base[M]

```
#define M 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
```

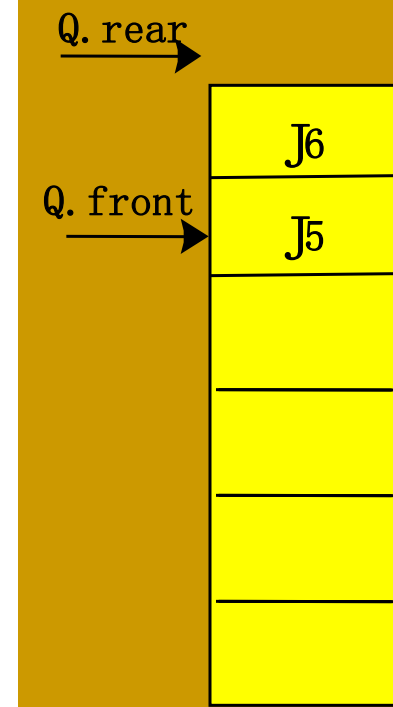
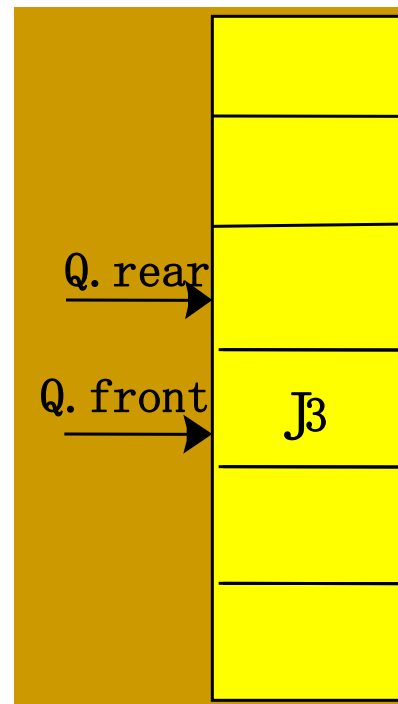
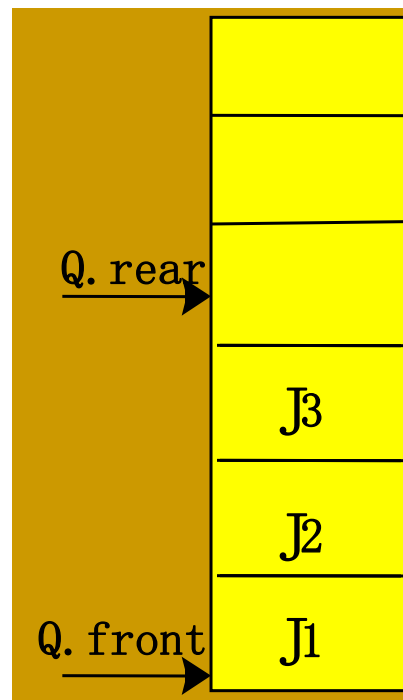
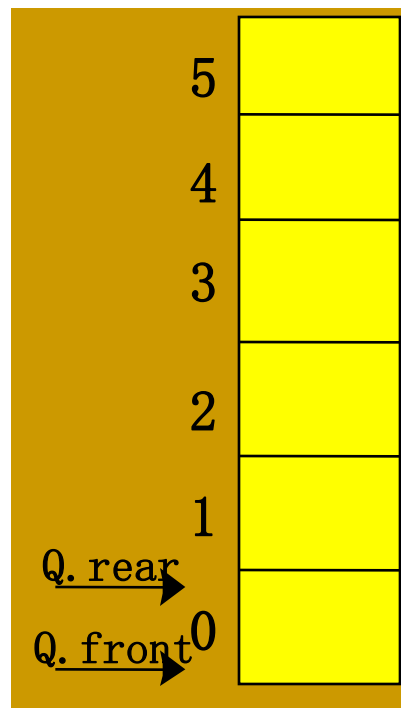
```
    int rear;         //尾指针
```

```
}SqQueue;
```

---



# 队列的顺序表示——用一维数组base[M]



**front=rear=0**

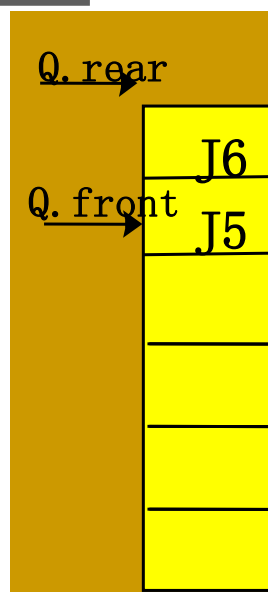
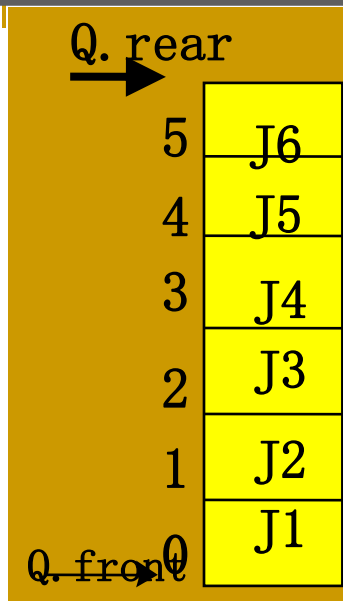
空队标志: **front==rear**

入队: **base[rear++]=x;**

出队: **x=base[front++];**

# 存在的问题

设大小为M



**front=0**

**rear=M时**

再入队—真溢出

**front≠0**

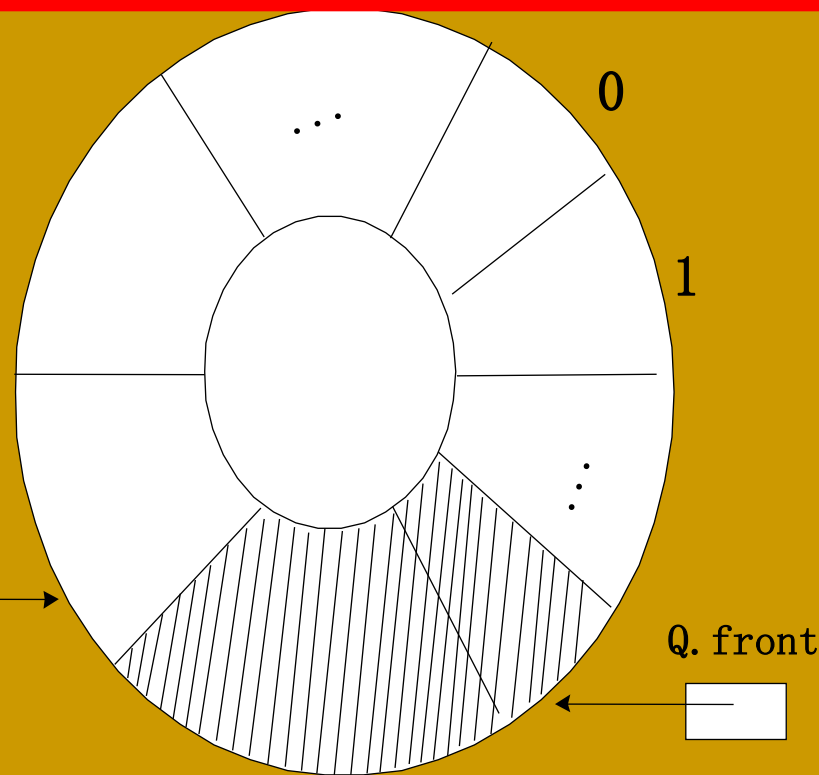
**rear=M时**

再入队—假溢出



# 解决的方法——循环队列

**base[0]接在base[M-1]之后**  
**若 $\text{rear}+1==M$**   
**则令 $\text{rear}=0$ ;**



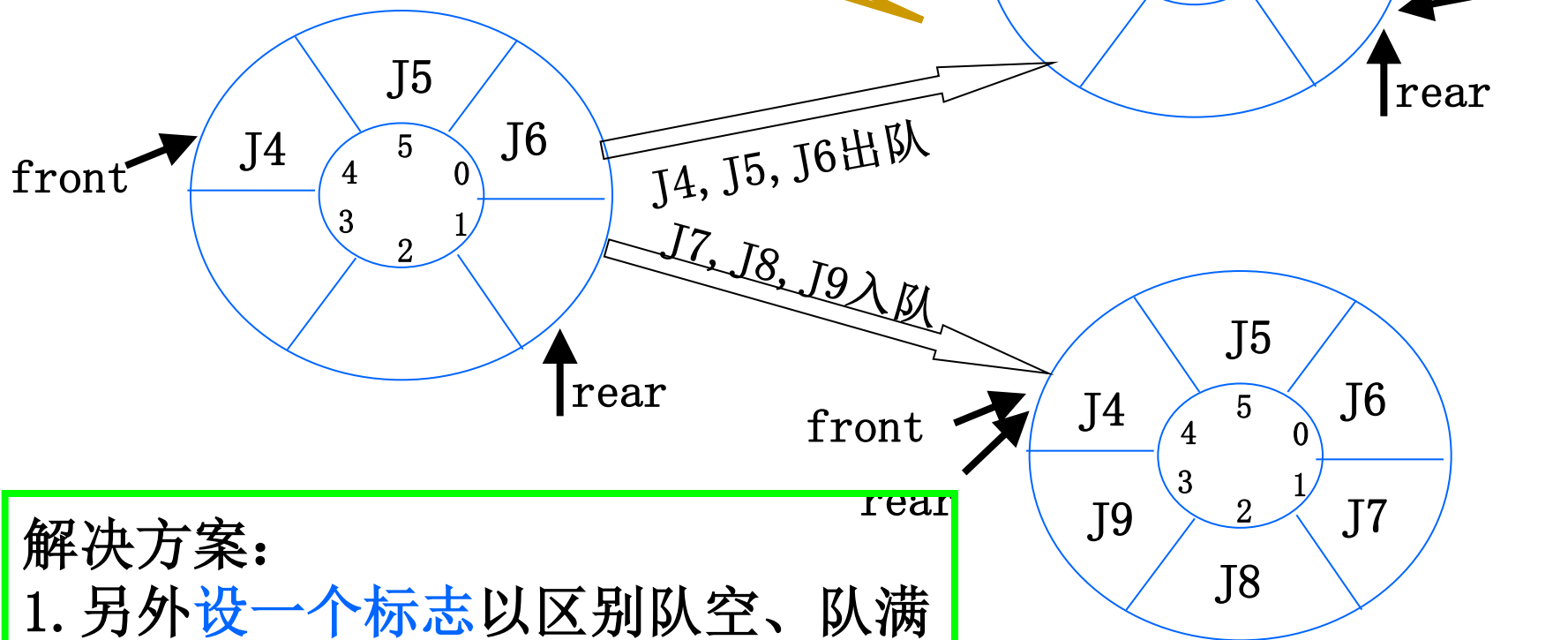
实现：利用“模”运算  
入队：

**$\text{base}[\text{rear}]=x;$**   
 **$\text{rear}=(\text{rear}+1)\%M;$**

出队：

**$x=\text{base}[\text{front}];$**   
 **$\text{front}=(\text{front}+1)\%M;$**

队空:  $\text{front} == \text{rear}$   
队满:  $\text{front} == \text{rear}$



解决方案:

1. 另外设一个标志以区别队空、队满
2. 少用一个元素空间:

队空:  $\text{front} == \text{rear}$

队满:  $(\text{rear} + 1) \% M == \text{front}$

# 循环队列

```
#define MAXQSIZE 100 //最大长度
```

```
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;
```



# 循环队列初始化

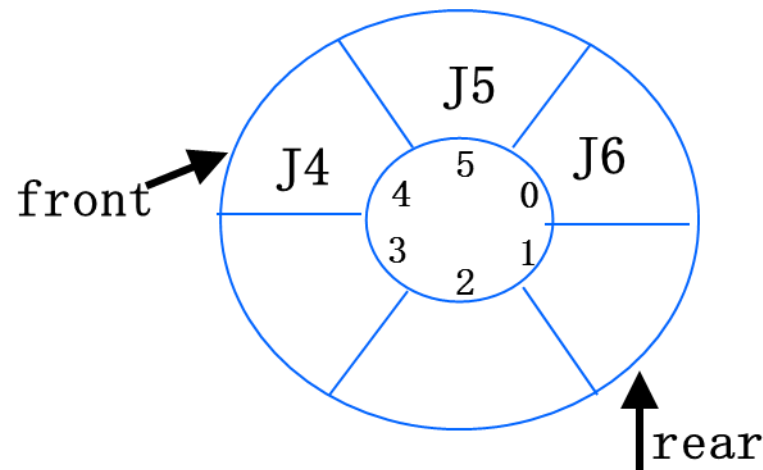
---

```
Status InitQueue (SqQueue &Q){  
    Q.base =new QElemType[MAXQSIZE]  
    if(!Q.base) exit(OVERFLOW);  
    Q.front=Q.rear=0;  
    return OK;  
}
```

---

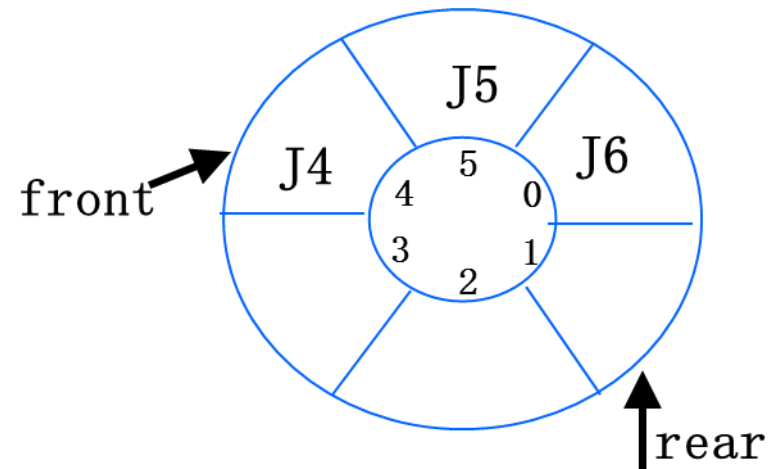
# 求循环队列的长度

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



# 循环队列入队

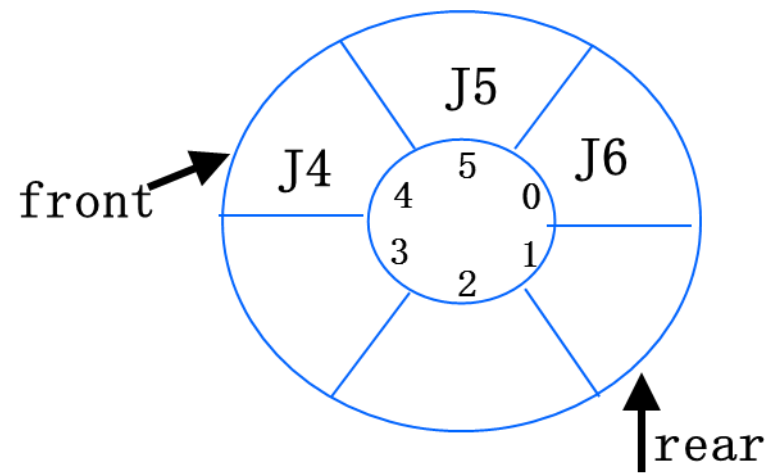
```
Status EnQueue(SqQueue &Q, QElemType e){  
    if((Q.rear+1)%MAXQSIZE==Q.front) return  
    ERROR;  
    Q.base[Q.rear]=e;  
    Q.rear=(Q.rear+1)%MAXQSIZE;  
    return OK;  
}
```



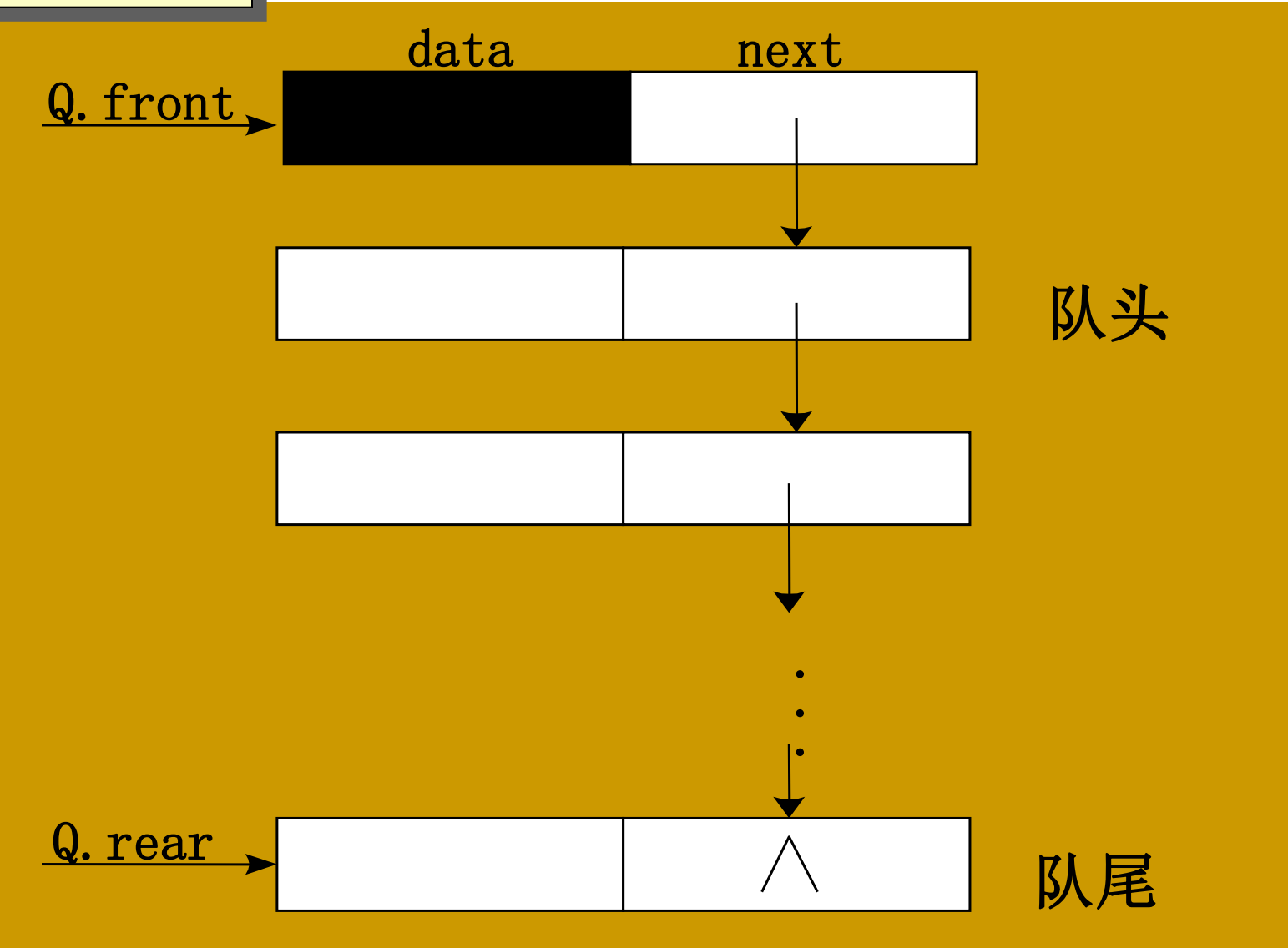


# 循环队列出队

```
Status DeQueue (LinkQueue &Q, QElemType &e){  
    if(Q.front==Q.rear) return ERROR;  
    e=Q.base[Q.front];  
    Q.front=(Q.front+1)%MAXQSIZE;  
    return OK;  
}
```



# 链队列



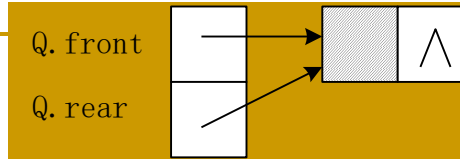
# 链队列

```
typedef struct QNode{
    QElemType  data;
    struct Qnode  *next;
}Qnode, *QueuePtr;

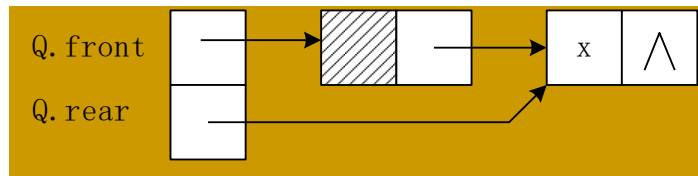
typedef struct {
    QueuePtr front;           //队头指针
    QueuePtr rear;           //队尾指针
}LinkQueue;
```

---

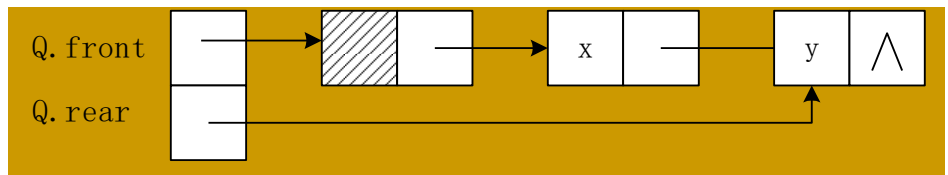
# 链队列



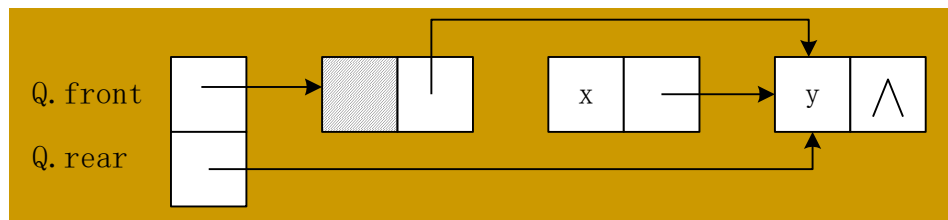
(a) 空队列



(b) 元素x入队列



(c) 元素y入队列



(d) 元素x出队列

# 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;  
}
```

---

# 销毁链队列

---

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

---

# 判断链队列是否为空

---

```
Status QueueEmpty (LinkQueue Q){  
    return (Q.front==Q.rear);  
}
```

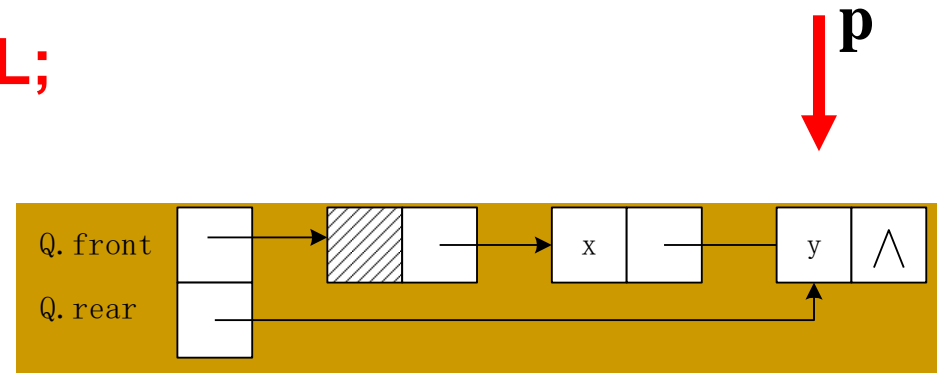
# 求链队列的队头元素

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



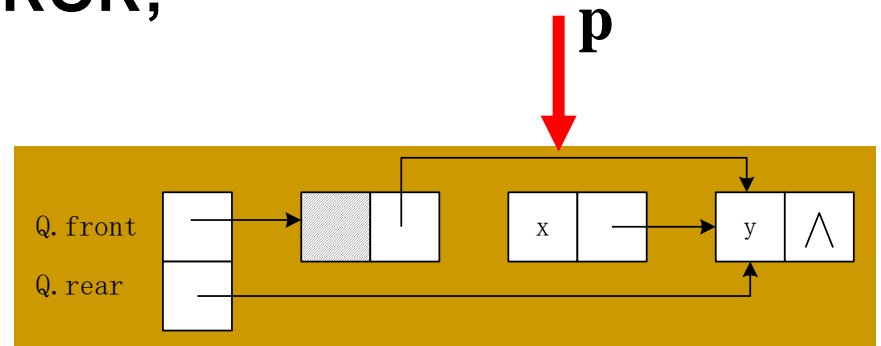
# 链队列入队

```
Status EnQueue(LinkQueue &Q, QElemType e){  
    p=(QueuePtr)malloc(sizeof(QNode));  
    if(!p) exit(OVERFLOW);  
    p->data=e; p->next=NULL;  
    Q.rear->next=p;  
    Q.rear=p;  
    return OK;  
}
```



# 链队列出队

```
Status DeQueue (LinkQueue &Q, QElemType &e){  
    if(Q.front==Q.rear) return ERROR;  
    p=Q.front->next;  
    e=p->data;  
    Q.front->next=p->next;  
    if(Q.rear==p) Q.rear=Q.front;  
    delete p;  
    return OK;  
}
```





## 3.6 案例分析与实现

### 案例3.1：数制的转换

#### 【算法步骤】

- ① 初始化一个空栈S。
- ② 当十进制数N非零时，循环执行以下操作：
  - 把N与8求余得到的八进制数压入栈S；
  - N更新为N与8的商。
- ③ 当栈S非空时，循环执行以下操作：
  - 弹出栈顶元素e；
  - 输出e。

## 案例3.1：数制的转换

### 【算法描述】

```
void conversion(int N)
{ // 对于任意一个非负十进制数，打印输出与其等值的八进制数
    InitStack(S); // 初始化空栈S
    while(N)      // 当N非零时，循环
    {
        Push(S, N%8); // 把N与8求余得到的八进制数压入栈S
        N=N/8;        // N更新为N与8的商
    }
    while(!StackEmpty(S)) // 当栈S非空时，循环
    {
        Pop(S, e);      // 弹出栈顶元素e
        cout<<e;        // 输出e
    }
}
```

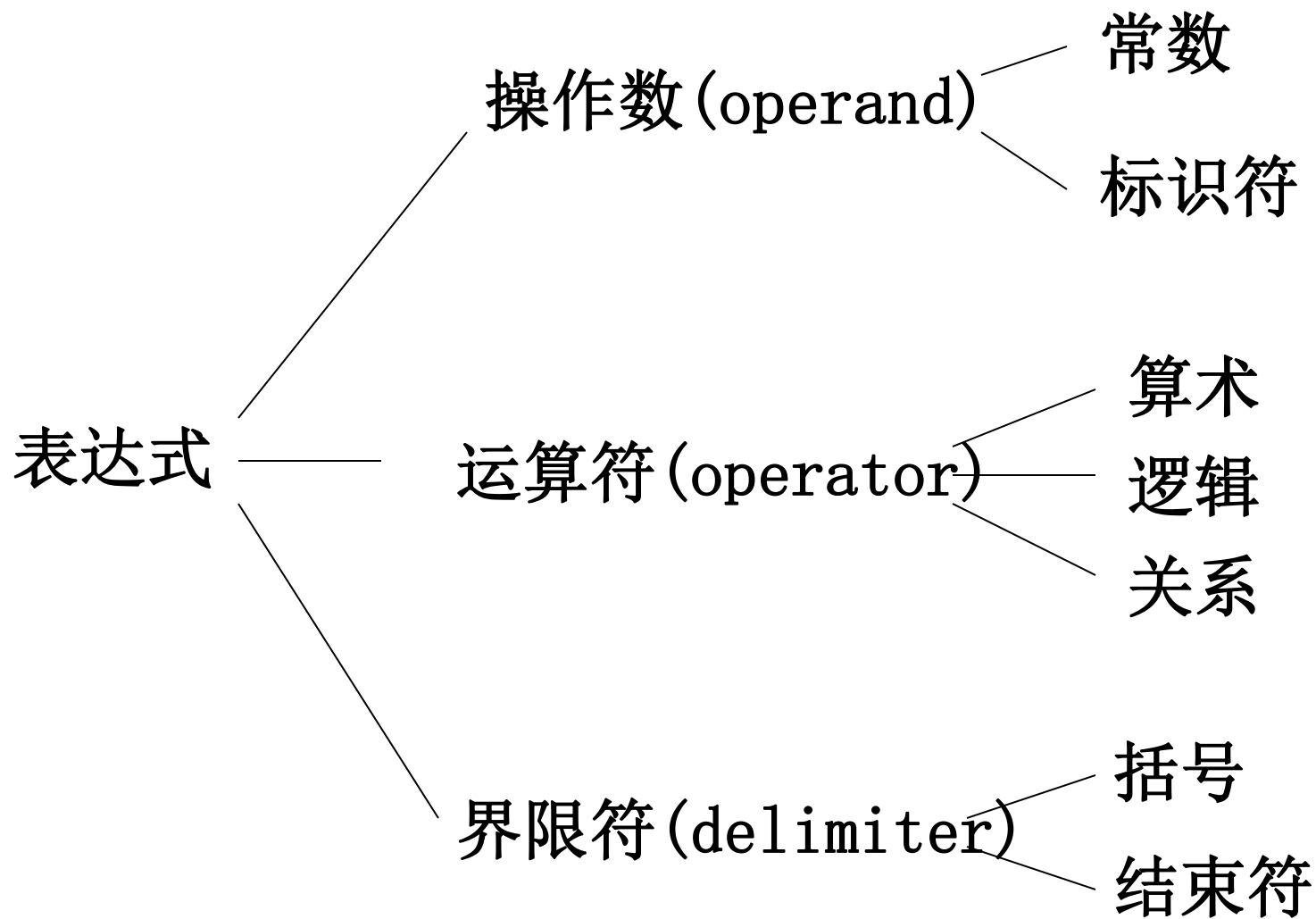
## 案例3.2：括号的匹配

### 【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。
- 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；
  - 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。
- 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**。

### 算术四则运算规则

- (1) 先乘除, 后加减
- (2) 从左算到右
- (3) 先括号内, 后括号外



# 表3.1 算符间的优先关系

$\theta_1 \backslash \theta_2$	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	<b>X</b>
)	>	>	>	>	<b>X</b>	>	>
#	<	<	<	<	<	<b>X</b>	=



# 【算法步骤】

设定两栈：OPND-----操作数或运算结果    OPTR-----运算符

- ① 初始化OPTR栈和OPND栈，将表达式起始符“#”压入OPTR栈。
- ② 扫描表达式，读入第一个字符ch，如果表达式没有扫描完毕至“#”或OPTR的栈顶元素不为“#”时，则循环执行以下操作：
  - 若ch不是运算符，则压入OPND栈，读入下一字符ch；
  - 若ch是运算符，则根据OPTR的栈顶元素和ch的优先级比较结果，做不同的处理：
    - 若是小于，则ch压入OPTR栈，读入下一字符ch；
    - 若是大于，则弹出OPTR栈顶的运算符，从OPND栈弹出两个数，进行相应运算，结果压入OPND栈；
    - 若是等于，则OPTR的栈顶元素是“（”且ch是“）”，这时弹出OPTR栈顶的“（”，相当于括号匹配成功，然后读入下一字符ch。
- ③ OPND栈顶元素即为表达式求值结果，返回此元素。

## OperandType EvaluateExpression( ) {

InitStack (OPTR); Push (OPTR, '#') ;

InitStack (OPND); ch = getchar( );

while (ch!= '#' || GetTop(OPTR)!= '#') {

if (! **In(ch)**){Push(OPND,ch); ch = getchar(); } // ch不是运算符则进栈  
else

switch (**Precede(GetTop(OPTR),ch)**) { //比较优先权

case '<' : //当前字符ch压入OPTR栈，读入下一字符ch

Push(OPTR, ch); ch = getchar(); break;

case '>' : //弹出OPTR栈顶的运算符运算，并将运算结果入栈

Pop(OPTR, theta);

Pop(OPND, b); Pop(OPND, a);

Push(OPND, **Operate(a, theta, b)**); break;

case '=' : //脱括号并接收下一字符

Pop(OPTR,x); ch = getchar(); break;

} // switch

} // while

return GetTop(OPND);} // EvaluateExpression

OPTR	OPND	INPUT	OPERATE
#		3*(7-2)#	Push(opnd,'3')
#	3	*(7-2)#	Push(optr,'*')
#,*	3	(7-2)#	Push(optr,'(')
#,*,(	3	7-2)#	Push(opnd,'7')
#,*,(	3,7	-2)#	Push(optr,'-')
#,*,(,—	3,7	2)#	Push(opnd,'2')
#,*,(,—	3,7,2	)#	Operate(7-2)
#,*,(	3,5	)#	Pop(optr)
#,*	3,5	#	Operate(3*5)
#	15	#	GetTop(opnd)

## 案例3.4：舞伴问题

---

### 【问题描述】

- 假设在周末舞会上，男士们和女士们进入舞厅时，各自排成一队。
  - 跳舞开始时，依次从男队和女队的队头上各出一人配成舞伴。
  - 若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。
-

## 案例3.4：舞伴问题

### 【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。

## 【数据结构】

// - - - - 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。

# 队列的其它应用



## 【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：

- 1. 入口处排队等候进入加油车道
- 2. 在加油车道排队等候加油
- 3. 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

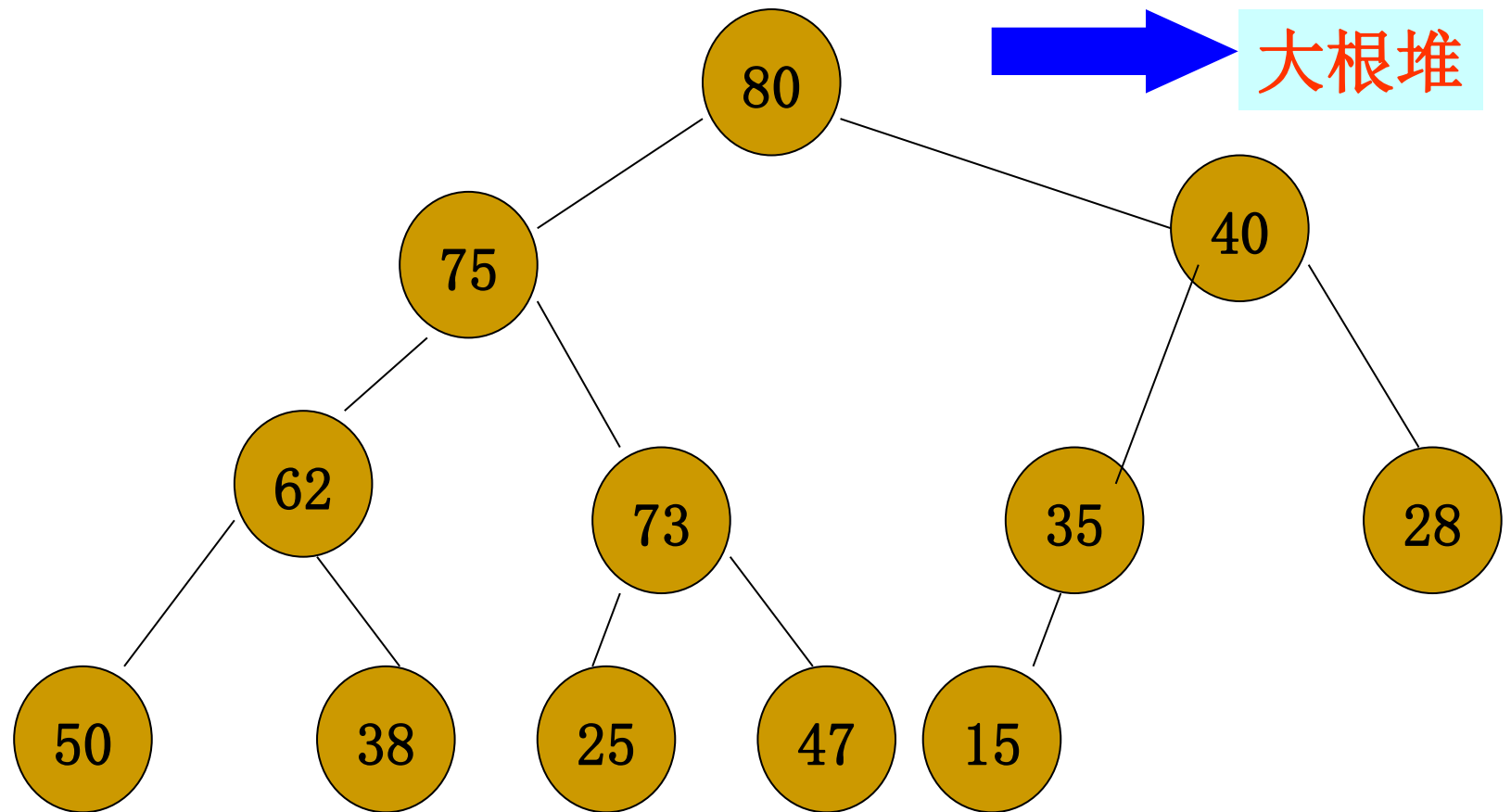


# 优先级队列(priority\_queue)---堆

- 每次从队列中取出的是具有最高优先权的元素
- 任务优先权及执行顺序的关系

任务编号	1	2	3	4	5
优先权	20	0	40	30	10
执行顺序	3	1	5	4	2

数字越小，优先权越高



## 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**表达式求值方法**

# 算法设计题

(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;
```



## 算法设计题

(2) 已知f为单链表的表头指针, 链表中存储的都是整型数据, 试写出实现下列运算的递归算法:

- ① 求链表中的最大整数;
- ② 求链表的结点个数;
- ③ 求所有整数的平均值。

---

# 谢谢大家！

