

练习

2.若已知一个栈的入栈序列是 $1, 2, 3, \dots, n$, 其输出序列为 $p_1, p_2, p_3, \dots, p_n$, 若 $p_1=n$, 则 p_i 为

C

A . i

B . n-i

C . n-i+1

D . 不确定

练习

3. 在一个具有 n 个单元的顺序栈中，假设以地址高端作为栈底，以 top 作为栈顶指针，则当作进栈处理时， top 的变化为

D

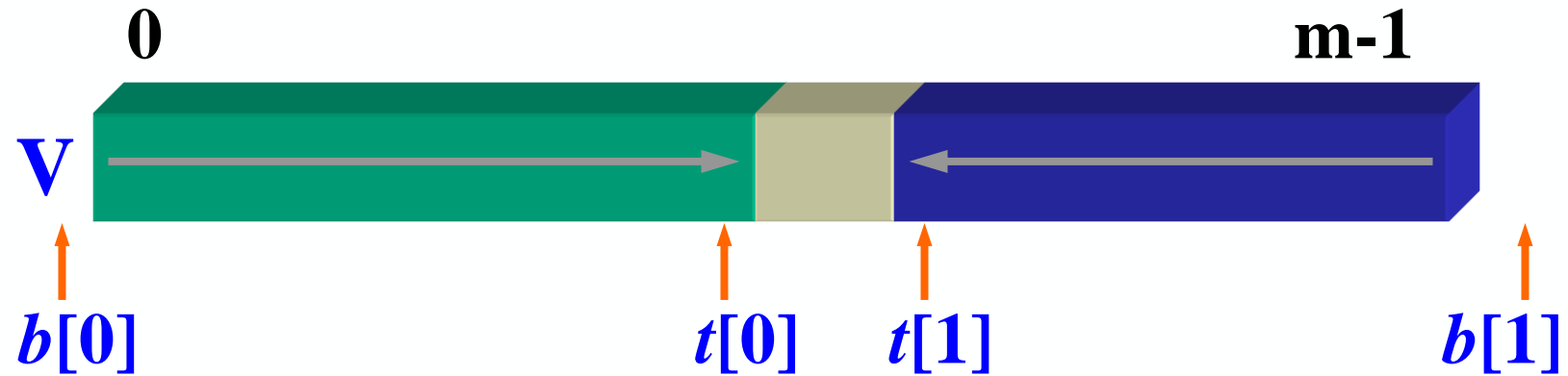
A. top 不变

B. $top=0$

C. $top++$

D. $top--$

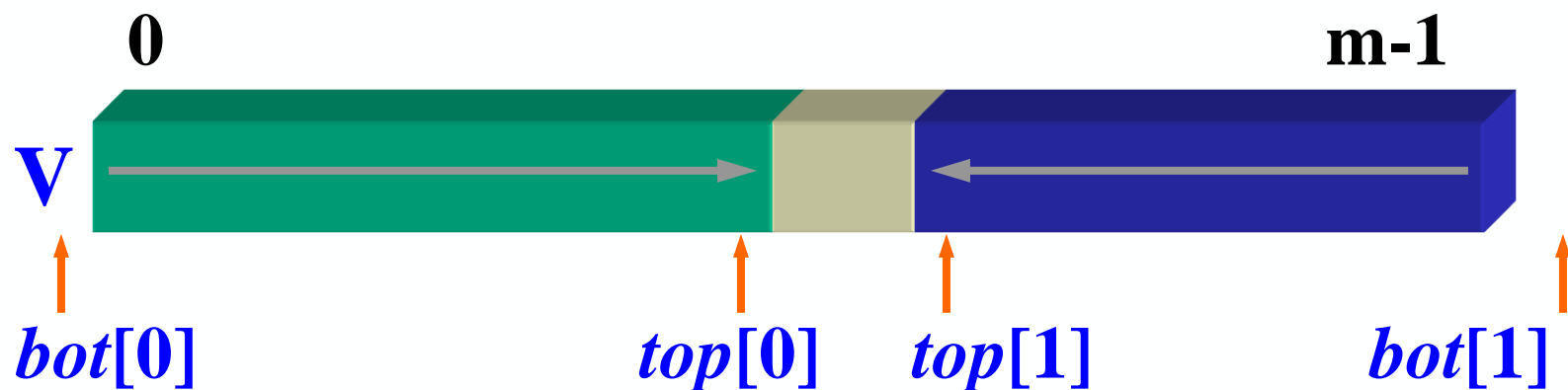
双栈共享一个栈空间



优点：互相调剂，灵活性强，减少溢出机会

考研试题

✓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（如下图所示）。



考研试题

✓数据结构定义如下

```
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;
```

考研试题

✓试编写判断栈空、栈满、进栈和出栈四个算法的函数(函数定义方式如下)

```
void Dbllpush(DblStack &s,SElemType x,int i) ;
```

```
//把x插入到栈i的栈
```

```
int Dbllpop(DblStack &s,int i,SElemType &x) ;
```

```
//退掉位于栈i栈顶的元素
```

```
int IsEmpty(DblStack s,int i) ;
```

```
//判栈i空否, 空返回1, 否则返回0
```

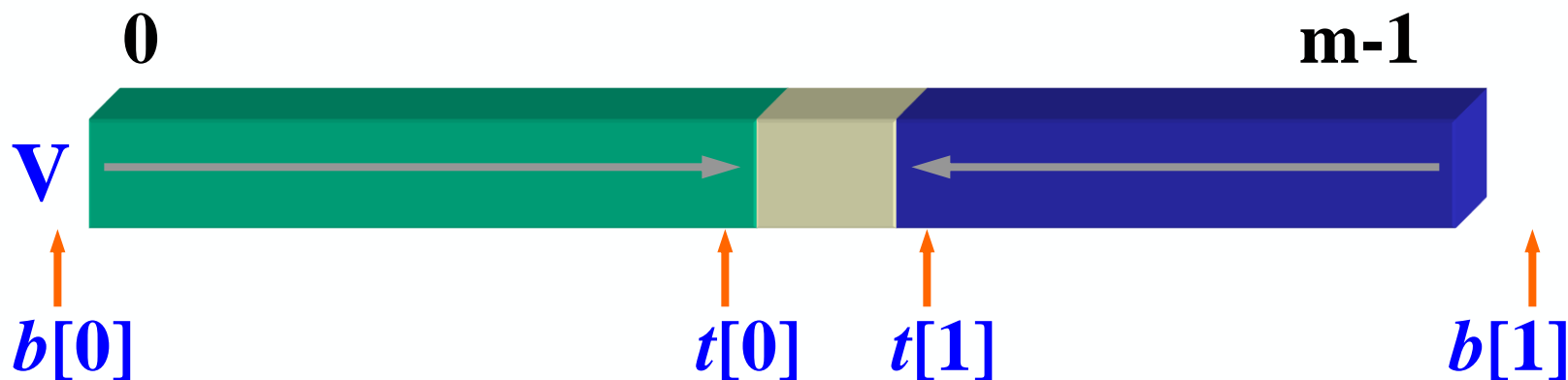
```
int IsFull(DblStack s) ;
```

```
//判栈满否, 满返回1, 否则返回0
```

提示

栈空: $\text{top}[i] == \text{bot}[i]$ i 表示栈的编号

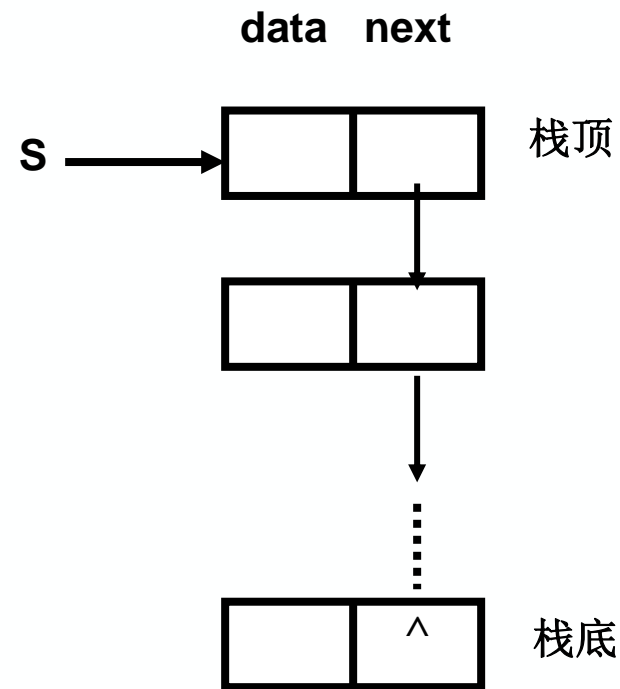
栈满: $\text{top}[0] + 1 == \text{top}[1]$ 或 $\text{top}[1] - 1 == \text{top}[0]$



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

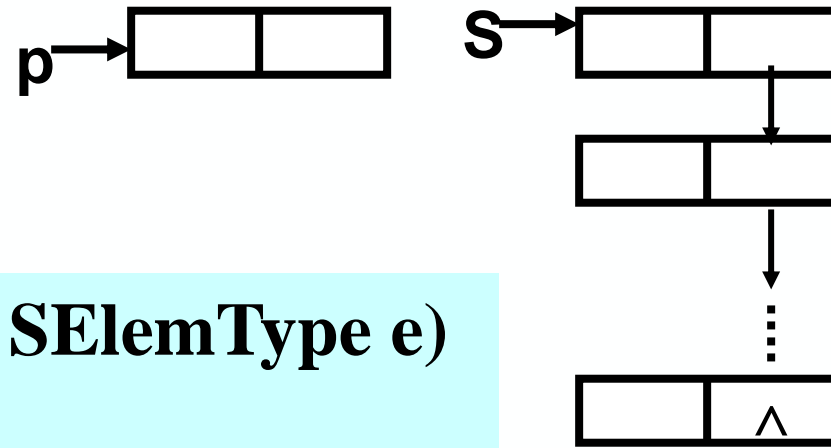
$S \longrightarrow \wedge$

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

判断链栈是否为空

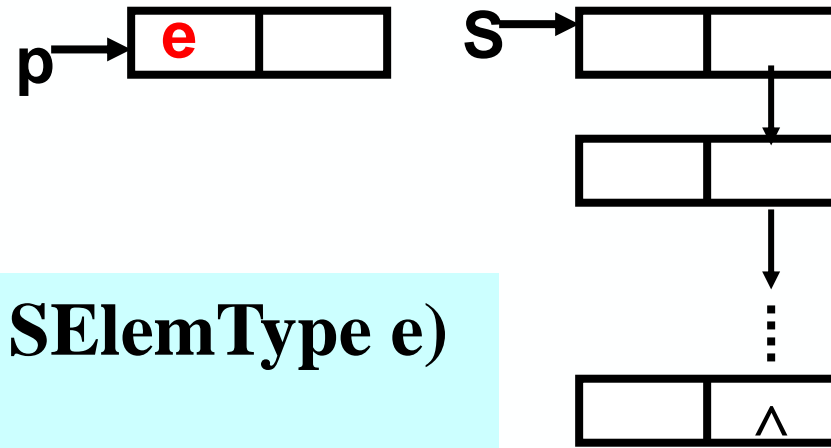
```
Status StackEmpty(LinkStack S)
{
    if (S==NULL) return TRUE;
    else return FALSE;
}
```

链栈进栈



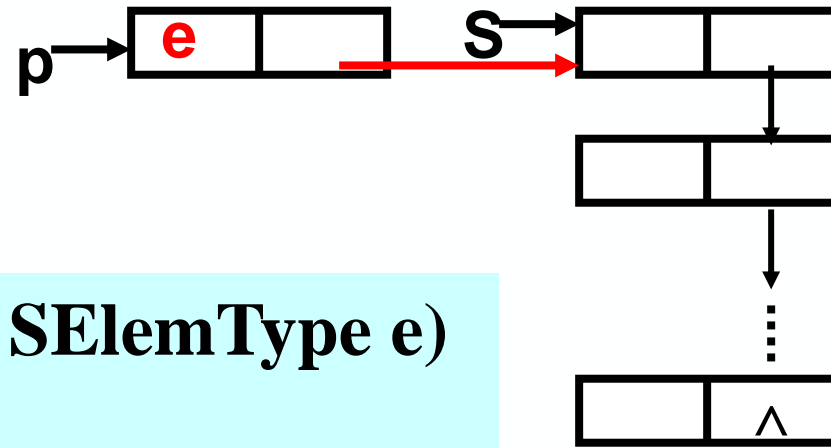
```
Status Push(LinkStack &S , SElemType e)
{
    p=new StackNode;    //生成新结点p
    if (!p) exit(OVERFLOW);
    p->data=e; p->next=S; S=p;
    return OK; }
```

链栈进栈



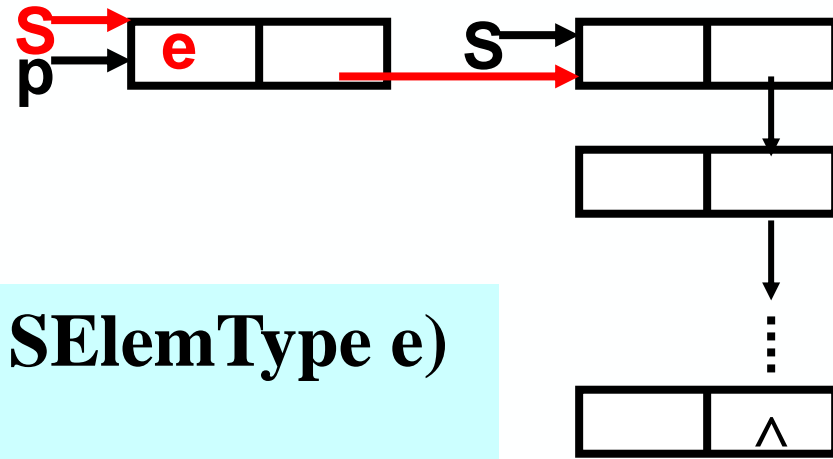
```
Status Push(LinkStack &S , SElemType e)
{
    p=new StackNode;    //生成新结点p
    if (!p) exit(OVERFLOW);
    p->data=e; p->next=S; S=p;
    return OK; }
```

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; }
```

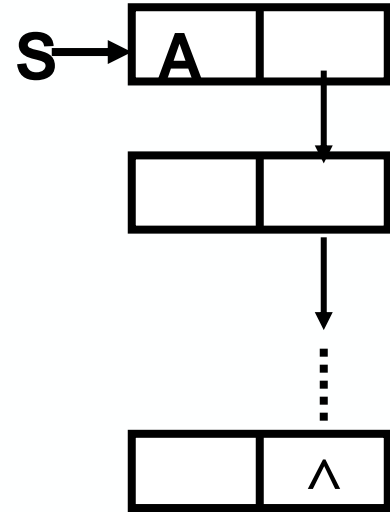
链栈进栈



```
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)

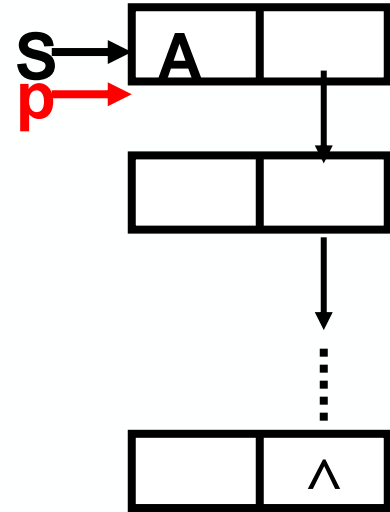
{if (S==NULL) return ERROR;

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

delete p; return OK; }

链栈出栈

$e = 'A'$



Status Pop (LinkStack &S,SElemType &e)

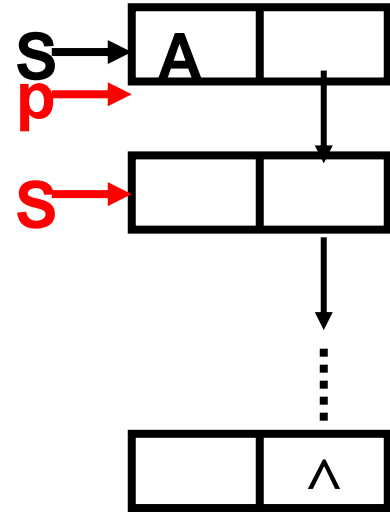
{if (S==NULL) return ERROR;

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

delete p; return OK; }

链栈出栈

$e = 'A'$



Status Pop (LinkStack &S,SElemType &e)

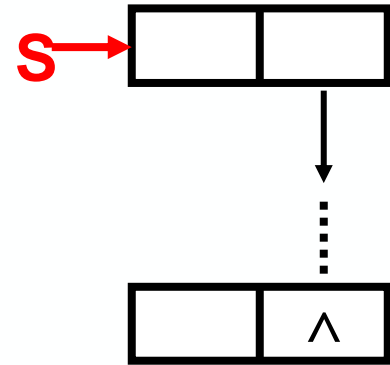
{if (S==NULL) return ERROR;

e = S-> data; p = S; $S = S \rightarrow \text{next};$

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); }
```



- ✓有人送了我**金、银、铜、铁、木**五个宝箱，我想打开金箱子，却没有打开这个箱子的钥匙。
- ✓在**金箱子**上面写着一句话：“打开我的钥匙装在**银箱**子里。”
- ✓于是我来到银箱子前，发现还是没有打开银箱子的钥匙。
- ✓银箱子上也写着一句话：“打开我的钥匙装在**铜箱**子里。”
- ✓于是我再来到铜箱子前，发现还是没有打开铜箱子的钥匙。
- ✓铜箱子上也写着一句话：“打开我的钥匙装在**铁箱**子里。”
- ✓于是我又来到了铁箱子前，发现还是没有打开铁箱子的钥匙。
- ✓铁箱子上也写着一句话：“打开我的钥匙装在**木箱**子里。”



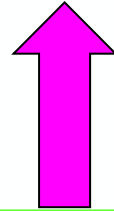
- ✓我来到木箱子前，打开了**木箱**，
- ✓并从木箱里拿出铁箱子的钥匙，打开了**铁箱**，
- ✓从铁箱里拿了出铜箱的钥匙，打开了**铜箱**，
- ✓再从铜箱里拿出银箱的钥匙打开了**银箱**，
- ✓最后从银箱里取出**金箱**的钥匙，打开了我想打开的金箱子。
- ✓晕吧.....很啰嗦地讲了这么长一个故事。



```
void FindKey ( 箱子 ) {  
    if ( 木箱子) return ;  
    else FindKey ( 下面的箱子 ) }
```

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

1. 递归定义的数学函数:

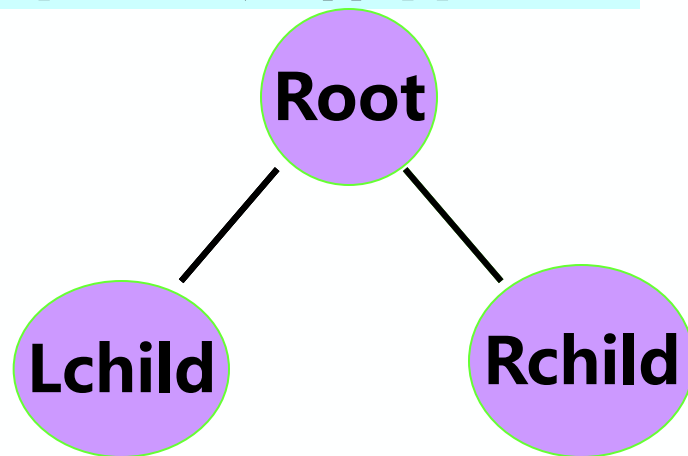
- 阶乘函数:
$$Fact(n) = \begin{cases} 1 & \text{若 } n = 0 \\ n \cdot Fact(n-1) & \text{若 } n > 0 \end{cases}$$

- 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:

- 树



- 广义表

$$A = (a, A)$$

3. 可递归求解的问题:

- 迷宫问题 Hanoi塔问题

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

必备的三个条件

- 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);
```



练习

设有一个递归算法如下:

```
int X(int n)
{ if(n<=3) return 1;
  else return X(n-2)+X(n-4)+1
}
```

则计算 $X(X(8))$ 时需要计算 X 函数 **D** 次.

A. 8

B.9

C.16

D.18

汉诺塔

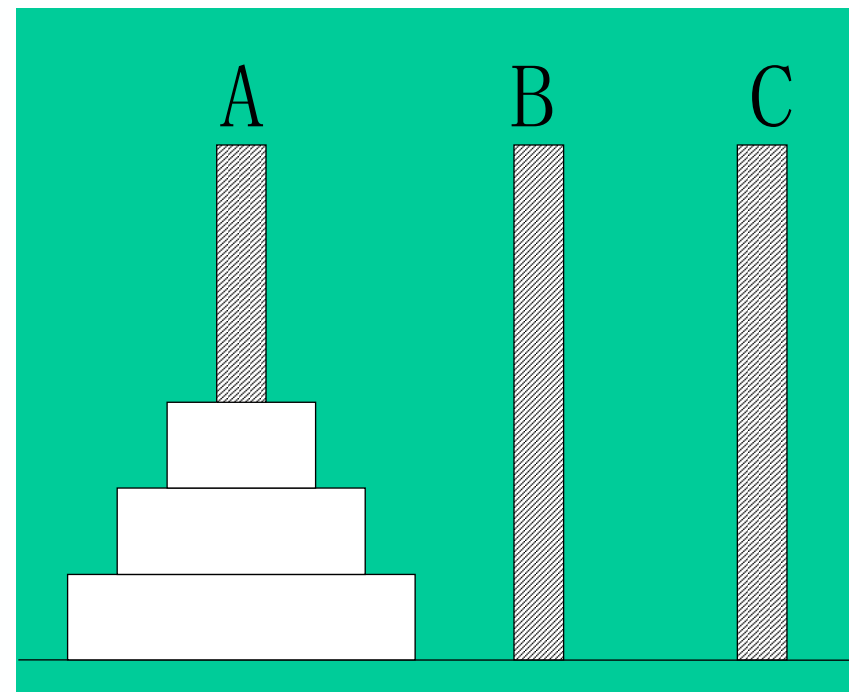


**在印度圣庙里，一块黄铜板上插着三根宝石针。
主神梵天在创造世界时，在其中一根针上穿好了由大到小的64片金片，这就是汉诺塔。
僧侣不停移动这些金片，一次只移动一片，小片必在大片上面。
当所有的金片都移到另外一个针上时，世界将会灭亡。**

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



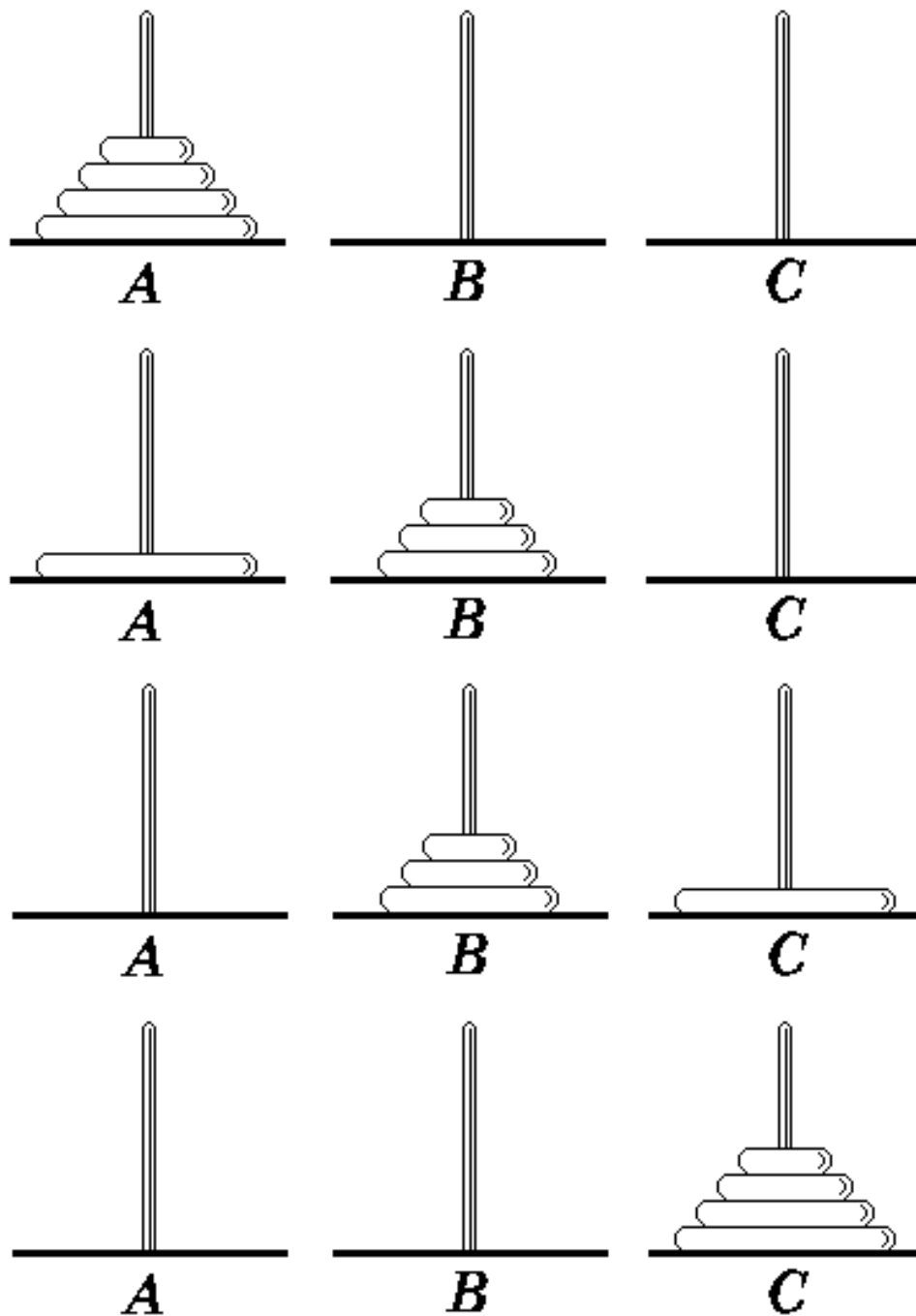
Hanoi塔问题

$n = 1$ ，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

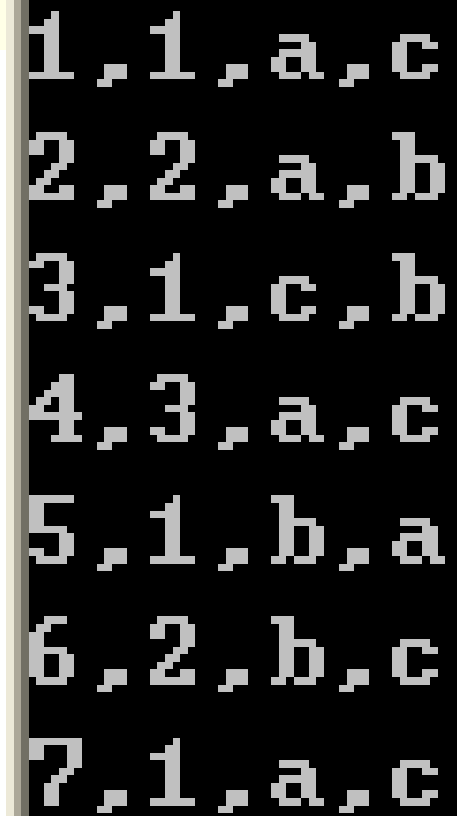
(2)将 A 最后一个直接移到 C

(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); }

