第3章　栈和队列

——操作受限的线性表

- 3.1 栈
  - 3.1.1抽象数据类型栈的定义
  - 3.1.2栈的表示与实现
- 3.2 栈的应用举例
  - 3.2.1 数制转换　3.2.4 迷宫求解　3.2.5 表达式求值
- *3.3 栈和递归的实现
- 3.4 队列
  - 3.4.1抽象数据类型队列的定义
  - 3.4.2 链队列——队列的链式表示与实现
  - 3.4.3 循环队列——队列的顺序表示与实现

一、教学目的与要求

了解栈的定义、学会栈的表示和实现

二、主要教学内容

栈的定义、栈的表示和实现、 栈的应用：数制转换；括号匹配的检验；行编辑程序；迷宫求解；表达式求值

三、教学重点、难点

栈的表示和实现、 栈的应用

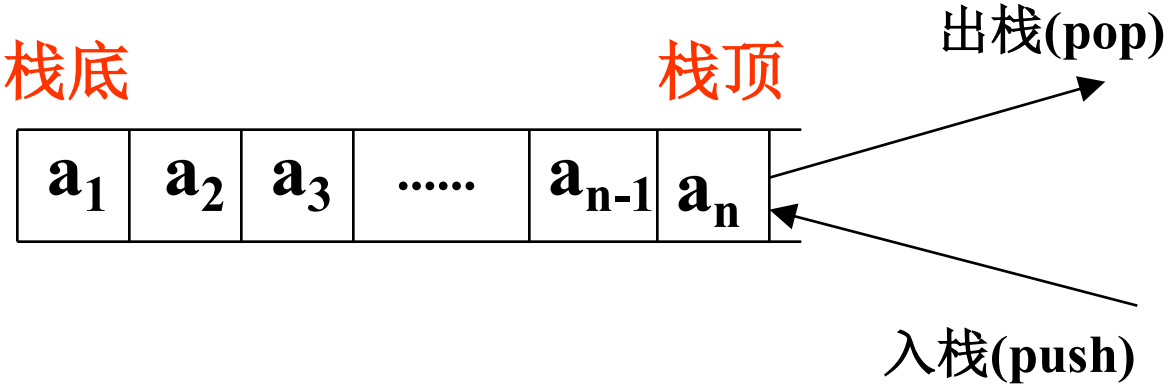
四、授课方法及手段

采用多媒体大屏幕投影授课

五、讲课具体内容（讲稿）

3.1 栈

- 栈（stack）：先进后出(FILO）的线性表。
  - 或后进先出（ LIFO）的线性表。
  - 或仅在表尾进行插入和删除操作的线性表。
- 栈顶（top）：线性表的表尾端，即可操作端。
- 栈底（bottom）：线性表的表头。



栈的抽象数据类型

- **ADT Stack {**
- **数据对象：** $D = \{a_i \mid a_i$属于Elemset, $(i=1, \ldots, n, n \geqslant 0)\}$
- **数据关系：** $R_1 = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i$属于$D, (i=2,3,\ldots,n)\}$
  约定a_n为栈顶, a_1为栈底。
- **基本操作：**
  - InitStack(&S);　DestroyStack(&S);　ClearStack(&S);
  - StackEmpty(S);　StackLength(S) ;　GetTop(S,&e);
  - Push(&S,e);　Pop(&S,&e);　StackTraverse(S,visit ())
- **}ADT Stack**

栈的基本操作(之一)

- InitStack(&S)
  - 操作结果:构造一个空的栈S。
- DestroyStack(&S)
  - 初始条件: 栈S已经存在。
  - 操作结果: 销毁栈S。
- ClearStack(&S)
  - 初始条件: 栈S已经存在。
  - 操作结果: 将栈S重置为空栈。

栈的基本操作(之二)

- **StackEmpty(S)**
  - 初始条件: 栈S已经存在。
  - 操作结果: 若栈S为空栈，则返回TURE;否则返回FALSE。
- **StackLength(S)**
  - 初始条件: 栈S已经存在。
  - 操作结果: 返回栈S中的数据元素个数。
- **GetTop(S,&e)**
  - 初始条件: 栈S已经存在且非空。
  - 操作结果: 用e返回栈S中栈顶元素的值。

栈的基本操作(之三)

- **Push(&S,e)**
  - 初始条件: 栈S已经存在。
  - 操作结果: 插入元素e为新的栈顶元素。
- **Pop(&S,&e)**
  - 初始条件: 栈S已经存在且非空。
  - 操作结果: 删除S的栈顶元素并用e返回其值。
- **StackTraverse(S,visit ())**
  - 初始条件: 栈S已经存在且非空。
  - 操作结果：从栈底到栈顶依次对S的每个元素调用函数visit ()。一旦visit ()失败，则操作失败。

关于栈的题目

- 已知三张火车编号为1, 2, 3，入站的先后次序为1, 2, 3，任何时候火车都可能出站，请写出所有可能的出站顺序。
  - 1, 2, 3
  - 1, 3, 2
  - 2, 1, 3
  - 2, 3, 1
  - 3, 2, 1

关于栈的题目

- 若有n个同学按1,2,3,4,…,n的顺序进入，而出电影院的顺序为S1,S2,S3,…,Sn，当S1=n时则Si为（　　）。
  - A. i　　B. n-i
  - C. n-i+1　　D. 不确定

关于栈的题目

- 若已知一个栈的进栈序列是1，2，3，…，n，其输出序列为p1，p2，p3，…，pn，若pn=n，则pi(1≤i≤n)为（　　）。
  - A. i　　B. n=i
  - C. n-i+1　　D. 不确定

关于栈的题目

- 若已知一个栈的进栈序列是1，2，3，…，n，其输出序列为p1，p2，p3，…，pn，若p1=3，则p2为（　　）。
  - A. 可能是2　　B. 不一定是2
  - C. 可能是1　　D. 一定是1

关于栈的题目

- 若已知一个栈的进栈序列是p1，p2，p3，…，pn，输出序列为1，2，3，…，n，若p3=1，则p1为（　　）。
  - A. 可能是2　　B. 一定是2
  - C. 不可能是2　　D. 不可能是3

关于栈的题目

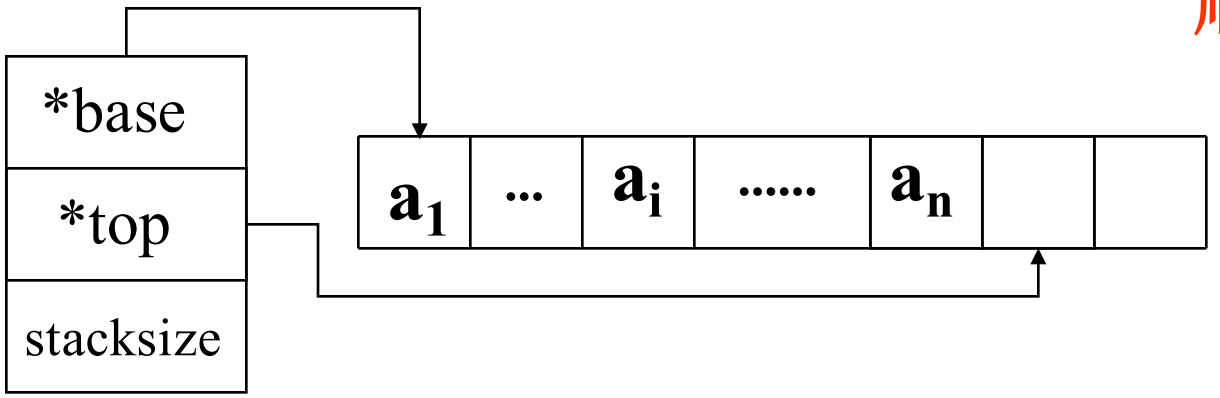
- 若已知一个栈的进栈序列是p1，p2，p3，…，pn，输出序列为1，2，3，…，n，若pn=1，则pi(1≤i<n)为（　　）。
  - A. n-i+1　　B. n-I
  - C. i　　D. 有多种可能

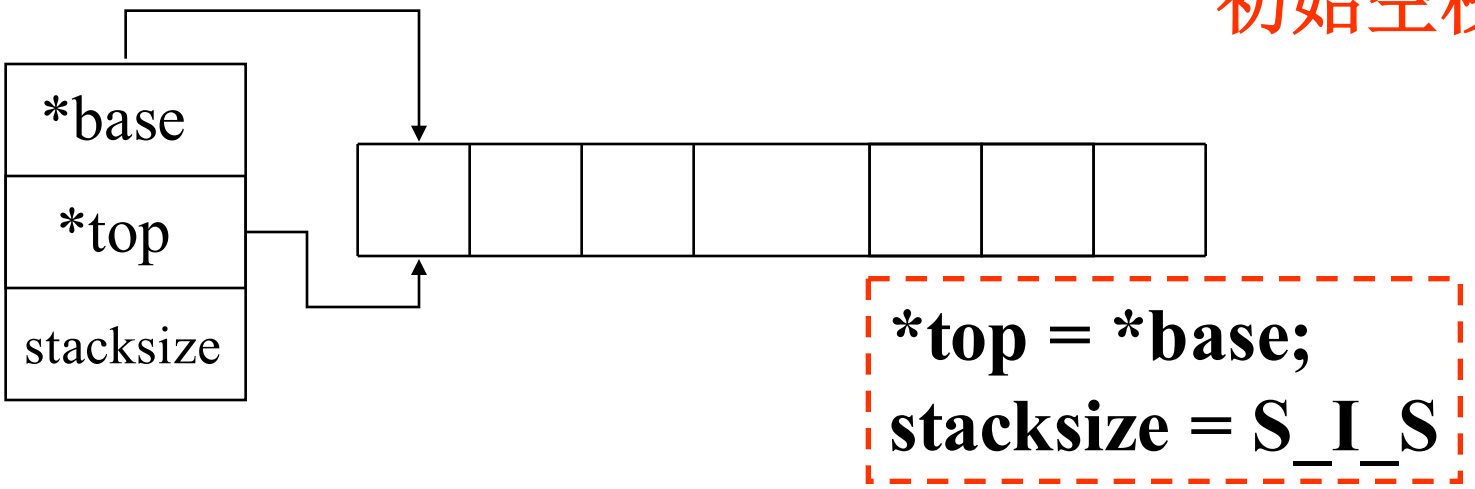
3.1.2 栈的顺序表示与实现(顺序栈)

- typedef struct{
  - SElemType *base;
    //在栈构造前和销毁后，值为NULL
  - SElemType *top; // 栈顶指针
  - int stacksize;
    //当前已分配的存储空间个数
- }SqStack ;

- #define S_I_S 100
- #define SI 10

顺序栈示意图

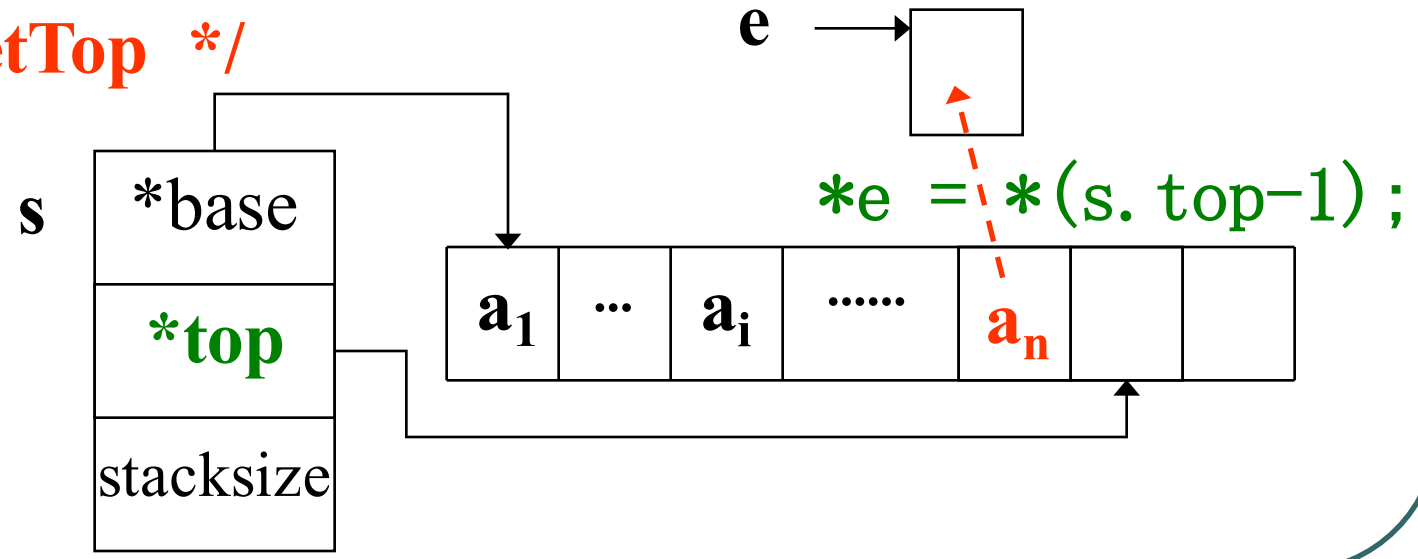

顺序栈
*base
*top
stacksize
a_1
...
a_i
.....
a_n


初始空栈
*base
*top
stacksize
*top = *base;
stacksize = S_I_S

顺序栈的操作实现举例

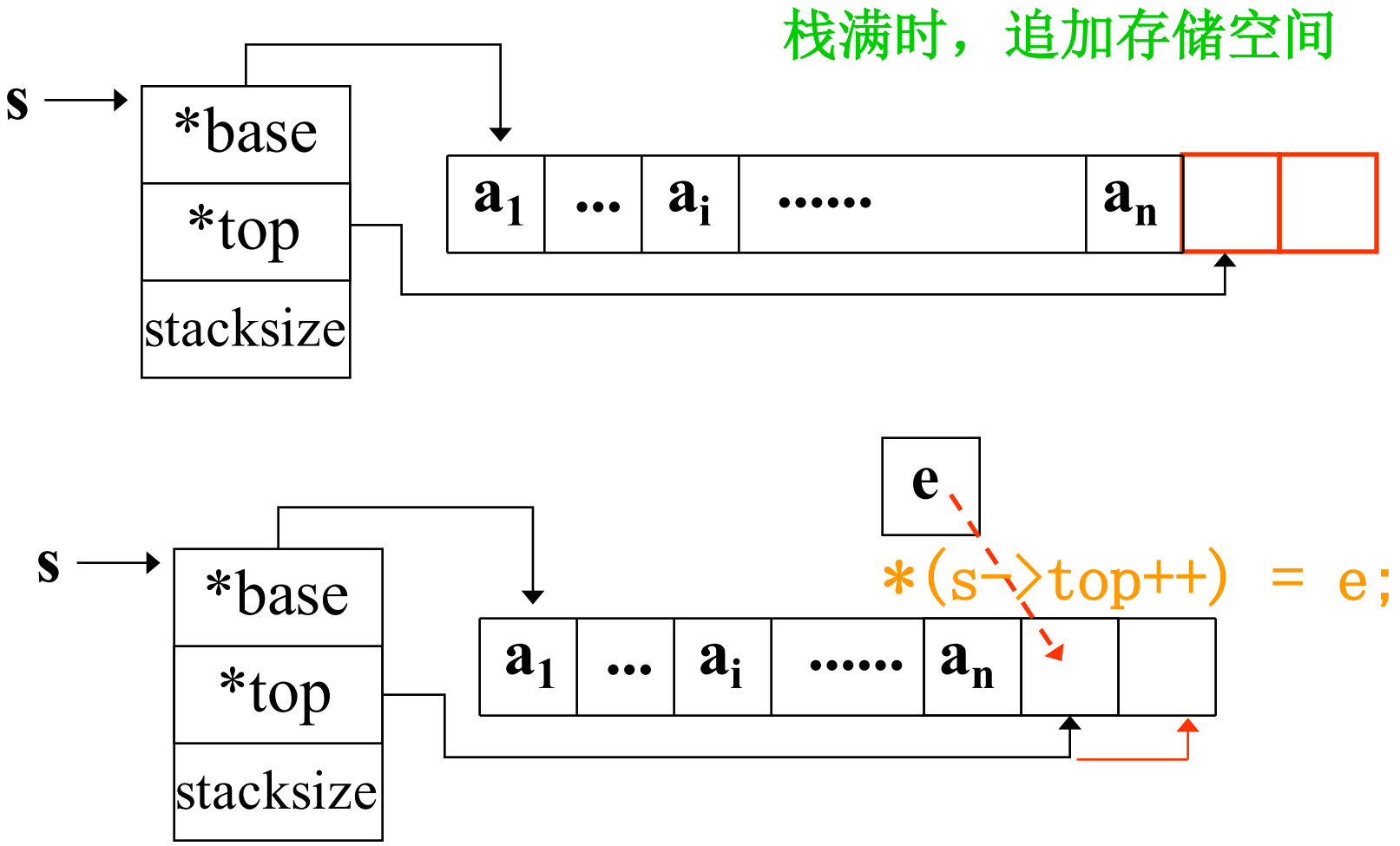
```
Status InitStack(SqStack *s)
{/* 构造一个空栈S  */
   s->base=(SElemType *)malloc
                (S_I_S * sizeof(SElemType));
   if(!s -> base) return(OVERFLOW);
   s -> top = s -> base;
   s -> stacksize = S_I_S;
   return OK;
} /* InitStack */
```

```
Status  GetTop(SqStack s, SElemType *e)
{/*栈S非空，则用e返回栈S中栈顶元素的值，
    并返回OK，则返回ERROR。*/
   if (s.top == s.base) return ERROR;
   *e = *(s.top-1);
   return OK;
} /* GetTop  */
```

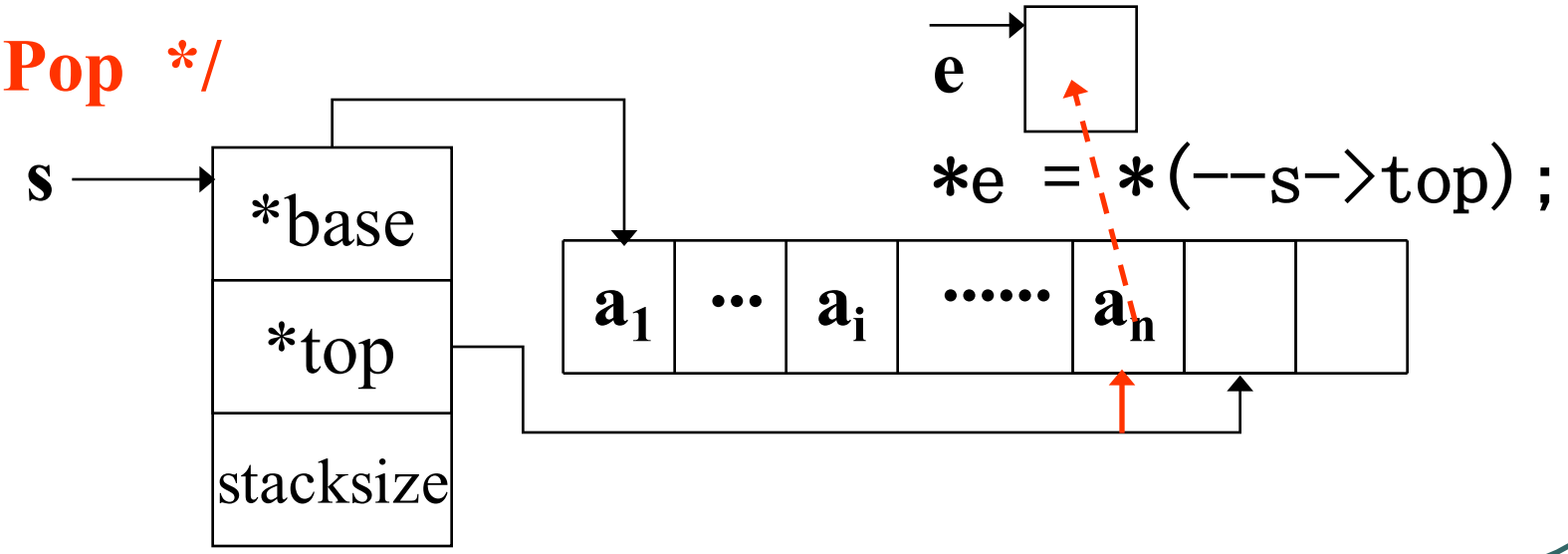


```
Status  Push(SqStack *s, SElemType e)
{/*插入元素e为新的栈顶元素。 */
  if (s->top - s->base>=s->stacksize)
                        /* 栈满，追加存储空间 */
   {l_temp=(SElemType*)realloc(s->base,
        (s->tacksize+SI)*sizeof(SElemType));
    if (!l_temp) return(OVERFLOW);
    s->base = l_temp;
    s->top = s->base + s->stacksize;
    s->stacksize += SI;
   }
  *(s->top++) = e;
  return OK;
} /*  Push */
```

插入新的栈顶元素时，堆栈变化示意

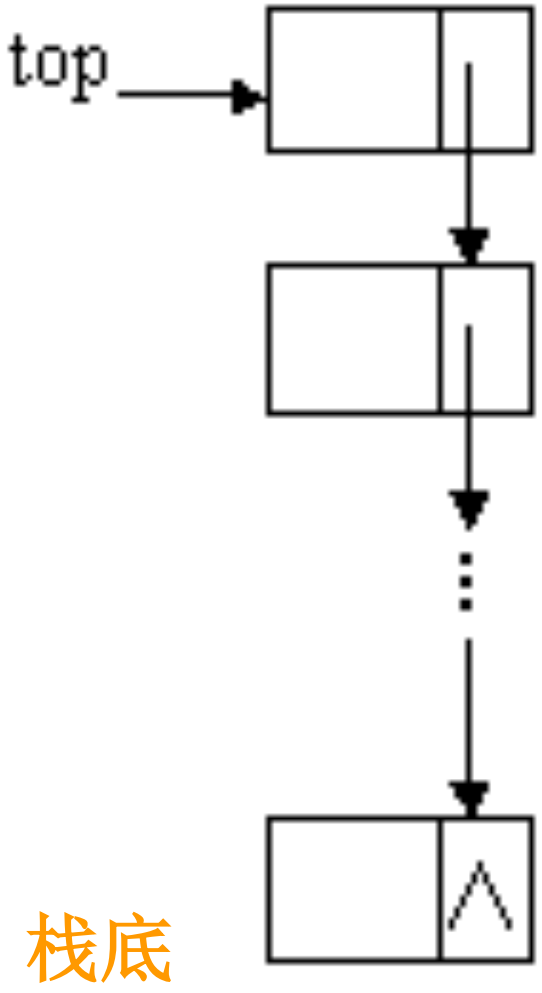


```
Status  Pop(SqStack *s, SElemType *e)
{/* 栈S非空,则删除S的栈顶元素,用e返回栈S中栈
      顶元素的值，并返回OK,否则返回ERROR。*/
   if (s->top == s->base) return ERROR;
   *e = *(--s->top);
   return OK;
} /* Pop  */
```



链　　式　　栈

栈的链式存储称为链式栈。链式栈就是一个特殊的单链表，对于这特殊的单链表，它的插入和删除规定在单链表的同一端进行。链式栈的栈顶指针一般用top表示，链式栈如下图所示。



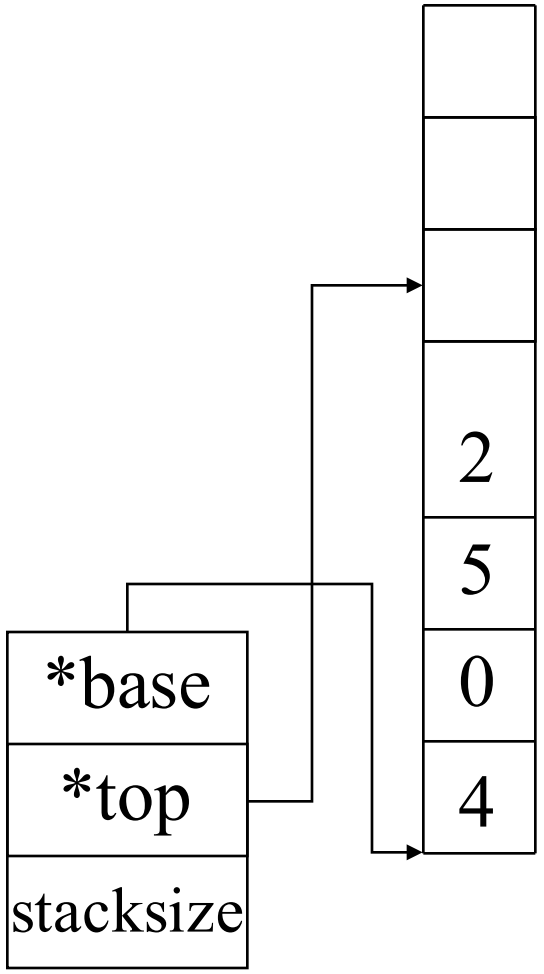
3.2 栈的应用举例

3.2.1 数制转换

- N = (N div d)×d + N mod d;
  1348 = 1348/8 * 8 +1348%8
- 例：（1348）$_{10}$ = （2504）$_{8}$

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

- 先进后出：
  数据生成的顺序：4,0,5,2
  读出的顺序：2,5,0,4



算法3.1：数制转换算法

```
void conversion()
{ /*输入一个非负的十进制数，输出等值的八进制数*/
   InitStack(&s);      scanf("%d",&N);
   while(N) {
      Push(&s, N%8);
      N = N/8;
   }
   while(!StackEmpty(s)) {
      Pop(&s,&e);
      printf("%d",e);
   }
} /* conversion */
```

3.2.2 括号匹配的检验

表达式中可以包含三种括号：小括号、中括号和大括号，各种括号之间允许任意嵌套，如小括号内可以嵌套中括号、大括号，但是不能交叉。举例如下：

正　确：([]{})　　([()])　　{([])}

不正确：{[( ])}　　{(}[])

如何去检验表达式的括号是否匹配呢？

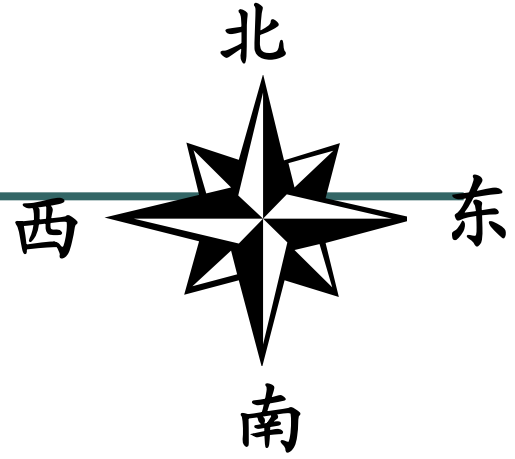
判断表达式括号是否匹配的具体实现:

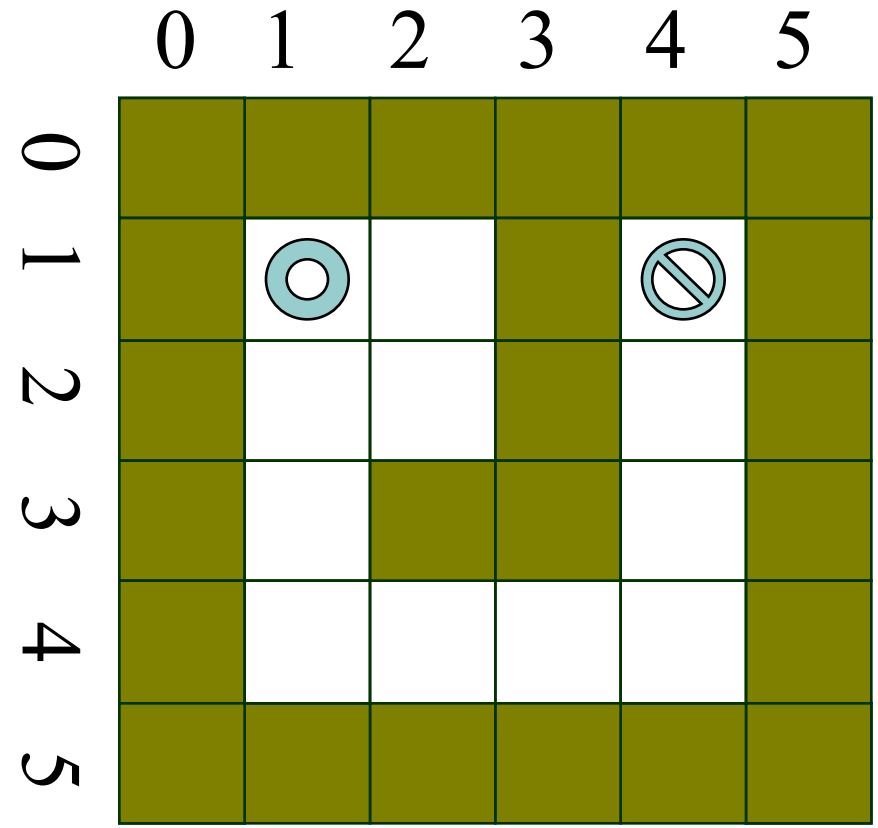
```
int match_kuohao(char c[])
{  int i=0;                    SqStack  s;
   InitStack(&s);
   while (c[i]!='#') {
          switch(c[i]) {
                case '{':
                case '[':
                case '(':  push(&s,c[i]);    break;
```

```
    case '}':  if(!StackEmpty(s) && GetTop(s)=='{' )
                    { pop(&s);      break; }
               else  return 0;
     case ']': if(!StackEmpty(s) && GetTop(s)=='[')
                    { pop(&s); break;}
               else  return 0;
     case ')': if(!StackEmpty (s) && GetTop(s)=='(')
                    { pop(&s);      break; }
               else return 0;
    } //end switch
    i++;
  } //end while
  return (StackEmpty(s));  //返回结果
}
```

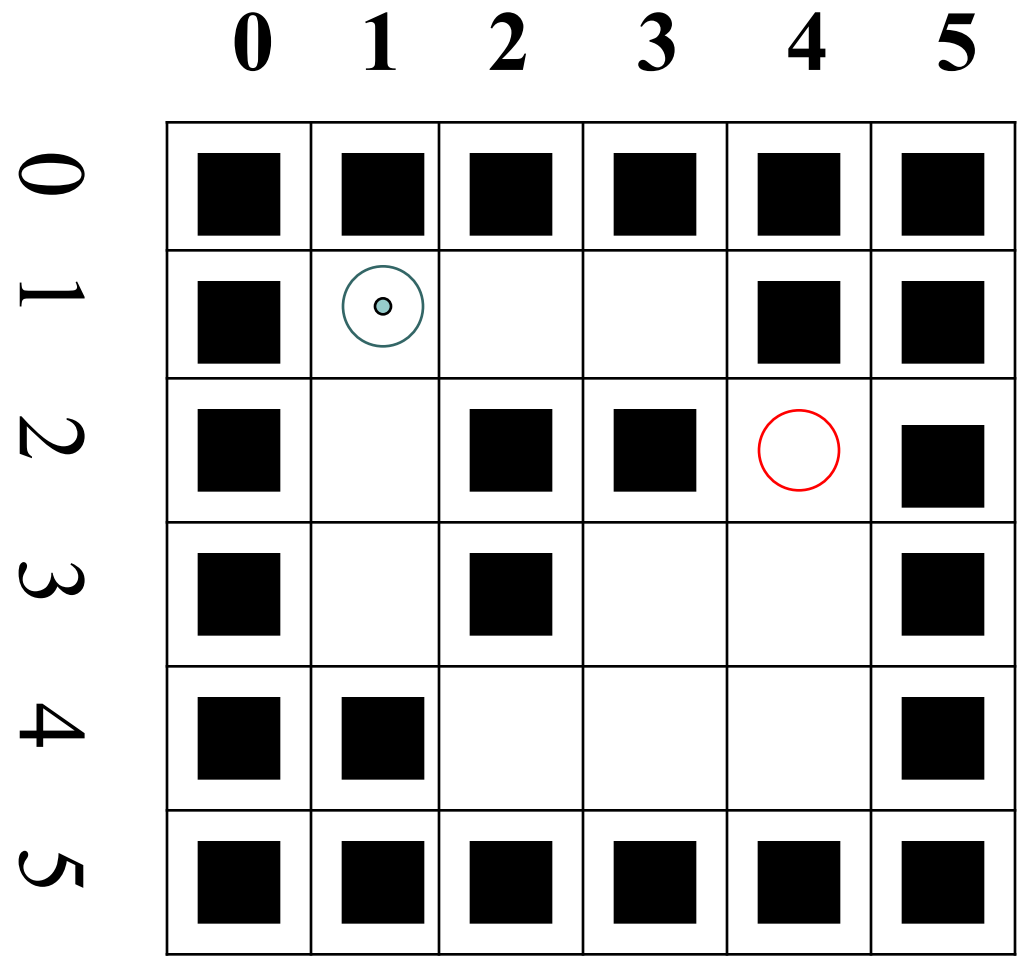
3.2.4 迷宫求解

例： start:(1,1)　　end:(1,4)





例：start:(1, 1)　　end:(2, 4)



设定当前位置的初值为入口位置;
do {若当前位置可通,
则{ 将当前位置插入栈顶;
若该位置是出口位置，则结束;
否则切换当前位置的东邻方块为新的当前位置;
}
否则，若栈不空且栈顶位置尚有其它方向未经探索,
则设定新的当前位置为顺时针方向旋转找到
的栈顶位置的下一相邻块;
若栈不空但栈顶位置的四周均不可通,
则{ 删去栈顶位置;
若栈不空,则重新测试新的栈顶位置,
直至找到一个可通的相邻块或出栈至空栈;
}
} while (栈不空);

```
typedef struct {
    int      ord;        // 通道块在路径上的“序号”
    PosType  seat;       // 通道块在迷宫中的“坐标位置”
    int      di;         // 从此通道块走向下一通道块的“方向”
} SElemType;             // 栈的元素类型

Status MazePath (MazeType maze, PosType start, PosType end)
{ InitStack(s);          curpos = start;          curstep = 1;
  do{  if(Pass(curpos)) {                         // 当前位置可通
         FootPrint(curpos);                       // 留下足迹
         e = (curstep, curpos,1);
         Push(s,e);                               // 加入路径
         if(curpos == end)   return (TRUE);       // 到达终点(出口)
         curpos = NextPos(curpos,1);
                                                  // 下一个位置是当前位置的东邻方块
         curstep ++ ;                             // 探索下一步
       }
```

```
        else{                                   // 当前位置不能通过
            if (!StackEmpty(s)){
                Pop(s,e);
                While(e.di==4 && !StackEmpty(s)) {
                    MarkPrint(e.seat); // 留下不能通过的足迹
                    Pop(s,e);                   // 退回一步
                } //end while
                if (e.di<4) {
                    e.di++;    Push(s,e);     // 换一个方向探索
                    curpos = NextPos(e.seat, e.di);
                     // 设定当前位置是该新方向上的相邻方块
                } //end if
            } //end if
        } //end if
    } while (!StackEmpty(s));
    return (FALSE);
} // MazePath
```

3.2.5 表达式求值

- 算符优先法:

4+2*3-10/5 = 4+6-10/5 = 10-10/5 =10-2 = 8

- 操作数(operand)：进OPND栈
- 操作符(operator)：进OPTR栈
- 界限符(delimiter)：#

算符间的优先关系:

θ_1 \ θ_2	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

Precede: 判定运算符栈的栈顶运算符 θ_1 与读入的运算符 θ_2 之间的优先关系的函数.

Operate: 进行二元运算a θ b的函数.

算法3.4：算术表达式求值

```
OperandType EvaluateExpression()
{ InitStack(OPTR);   Push(OPTR, '#');
  InitStack(OPND);   c = getchar();
  While(c!='#' || GetTop(OPTR)!='#')
     If (!In(c,OP)) { Push(OPND,c); c = getchar();}  //  不是运算符则进栈
     else
          switch (Precede(GetTop(OPTR),c)){
               case '<':  // 栈顶元素优先权低，把读入的操作符入栈
                          Push(OPTR,c);     c = getchar();       break;
               case '＝':  // 脱括号并接受下一个字符
                          Pop(OPTR,x);       c = getchar();      break;
               case '>':  // 退栈并将运算结果入栈
                          Pop(OPTR,theta);   Pop(OPND,b);  Pop(OPND,a);
                          Push(OPND,Operate(a,theta,b));      break;
               default: printf("Expression error!");      return(ERROR);
          } // switch
  } // while
  return GetTop(OPND);
} // EvaluateExpression
```

算法3.4：算术表达式求值

```
OperandType EvaluateExpression()
 { InitStack(OPTR);     Push(OPTR, '#');
   InitStack(OPND);     c = getchar();
    While(c!='#' || GetTop(OPTR)!='#')
        If (!In(c,OP)) { Push(OPND,c); c = getchar();}
                                  //  不是运算符则进栈
        else
            switch (Precede(GetTop(OPTR),c)){
              case '<':  // 栈顶元素优先权低，把读入的操作符入栈
                    Push(OPTR,c);     c = getchar();      break;
              case '=' :  // 脱括号并接受下一个字符
                    Pop(OPTR,x);      c = getchar();      break;
              case '>':  // 退栈并将运算结果入栈
                    Pop(OPTR,theta);  Pop(OPND,b);  Pop(OPND,a);
                    Push(OPND,Operate(a,theta,b));      break;
             default: printf("Expression error!");      return(ERROR);
        } // switch
   } // while
   return GetTop(OPND);
 } // EvaluateExpression
```

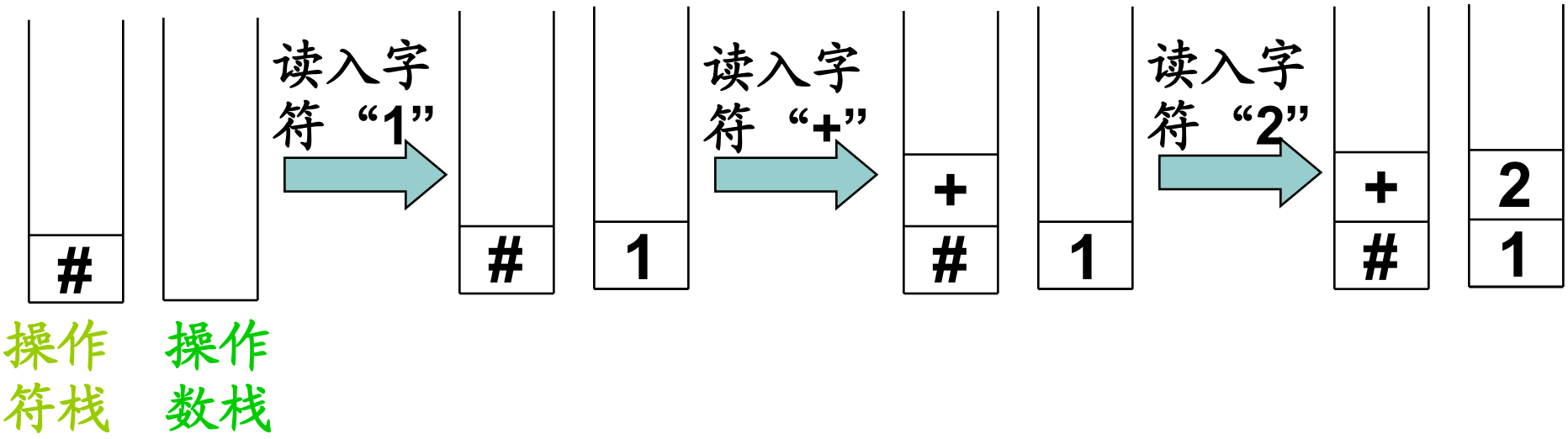
算法3.4：算术表达式求值

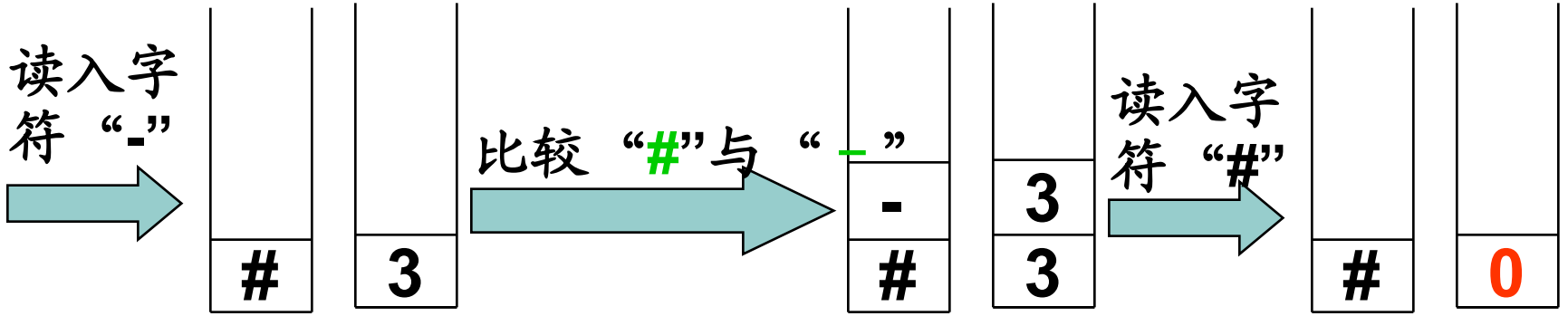
```
OperandType EvaluateExpression()
 { InitStack(OPTR);          Push(OPTR, '#');
   InitStack(OPND);          c = getchar();
     While(c!='#' || GetTop(OPTR)!='#')
        If (!In(c,OP)) { Push(OPND,c); c = getchar();}  // 不是运算符则进栈
        else
             switch (Precede(GetTop(OPTR),c)){
                    case '<':  // 栈顶元素优先权低，操作符入栈
                                Push(OPTR,c);  c = getchar();   break;
                    case '=' :  // 脱括号并接受下一个字符
                                Pop(OPTR,x);    c = getchar();  break;
                    case '>':  // 退栈并将运算结果入栈
                               Pop(OPTR,theta);   Pop(OPND,b);
                                Pop(OPND,a);
                                Push(OPND,Operate(a,theta,b));
                                break;
                   default: printf("Expression error!");
                                return(ERROR);
             } // switch
   } // while
   return GetTop(OPND);
 } // EvaluateExpression
```

算法3.4：算术表达式求值

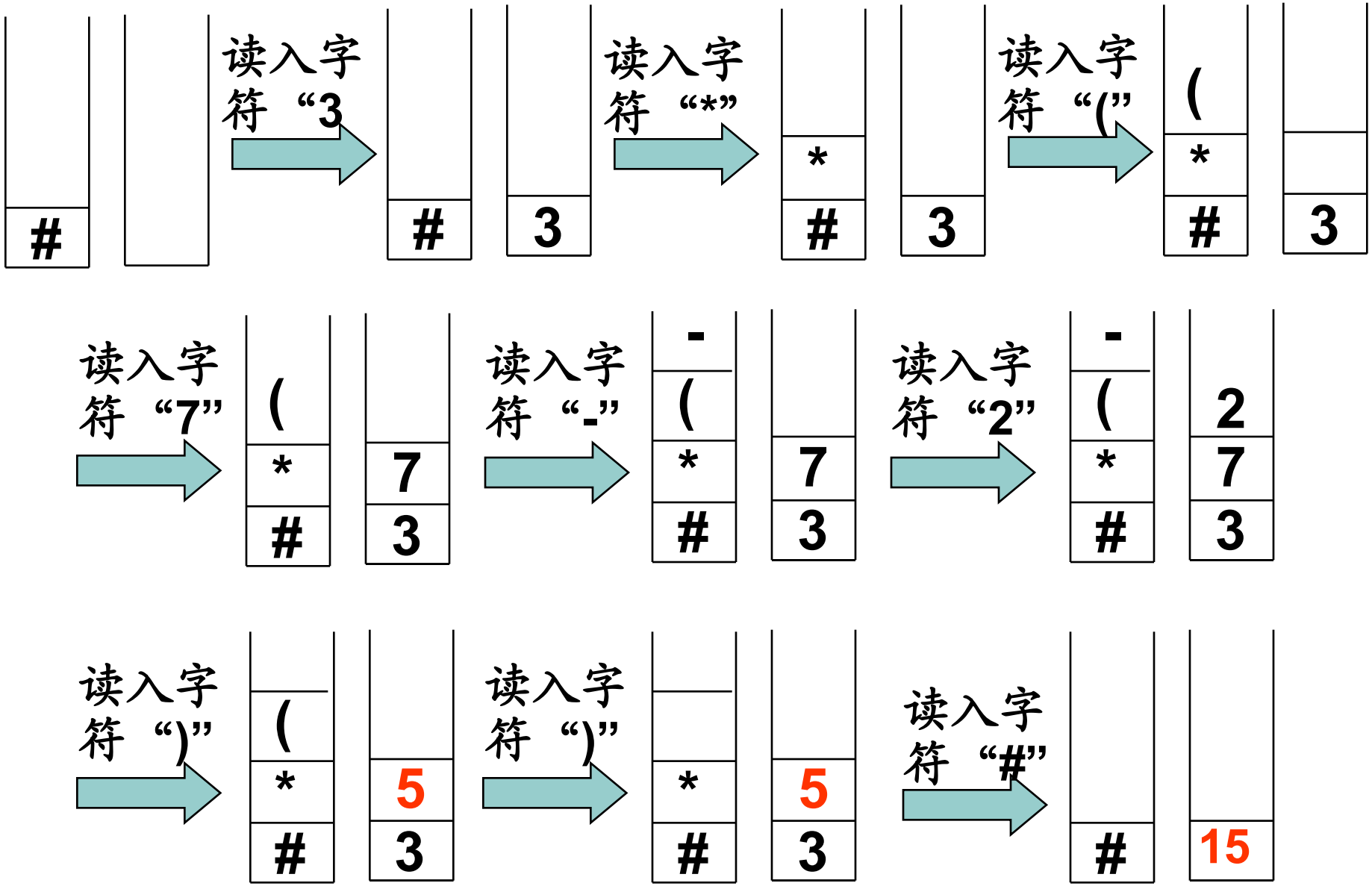
```
OperandType EvaluateExpression()
 { InitStack(OPTR);          Push(OPTR, '#');
   InitStack(OPND);          c = getchar();
     While(c!='#' || GetTop(OPTR)!='#')
        If (!In(c,OP)) { Push(OPND,c); c = getchar();}  // 不是运算符则进栈
        else
             switch (Precede(GetTop(OPTR),c)){
                  case '<':  // 栈顶元素优先权低，操作符入栈
                             Push(OPTR,c);  c = getchar();  break;
                  case '=' :  // 脱括号并接受下一个字符
                             Pop(OPTR,x);   c = getchar();  break;
                  case '>':  // 退栈并将运算结果入栈
                              Pop(OPTR,theta);  Pop(OPND,b);
                             Pop(OPND,a);
                             Push(OPND,Operate(a,theta,b));
                             break;
                   default: printf("Expression error!");
                                 return(ERROR);
                 } // switch
    } // while
    return GetTop(OPND);
 } // EvaluateExpression
```

例：计算表达式 “1+2−3#”





例:对算术表达式3*(7-2)#求值



3.3 栈与递归的实现

- 递归与递归程序设计
  - 递归程序执行过程分析
  - 递归程序到非递归程序的转换
  - 递归程序设计的应用实例

递归与递归程序设计

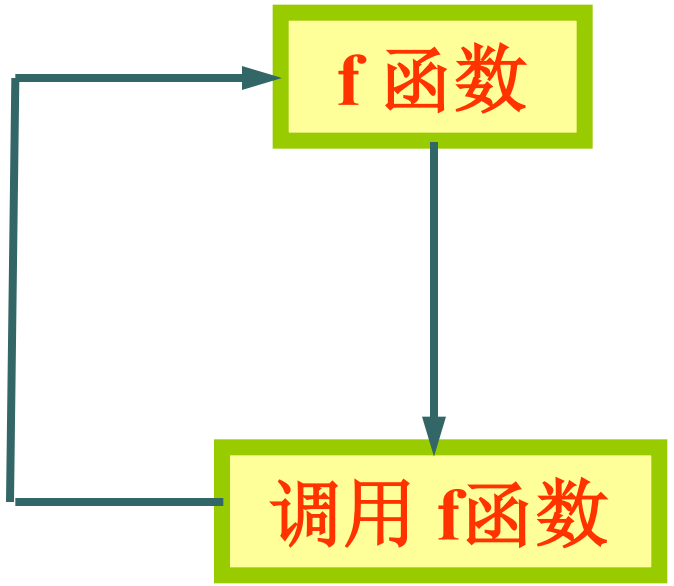
在一个函数的定义中出现了对自己本身的调用，称之为直接递归；

一个函数p的定义中包含了对函数q的调用，而q的实现过程又调用了p，即函数调用形成了一个环状调用链，这种方式称之为间接递归。

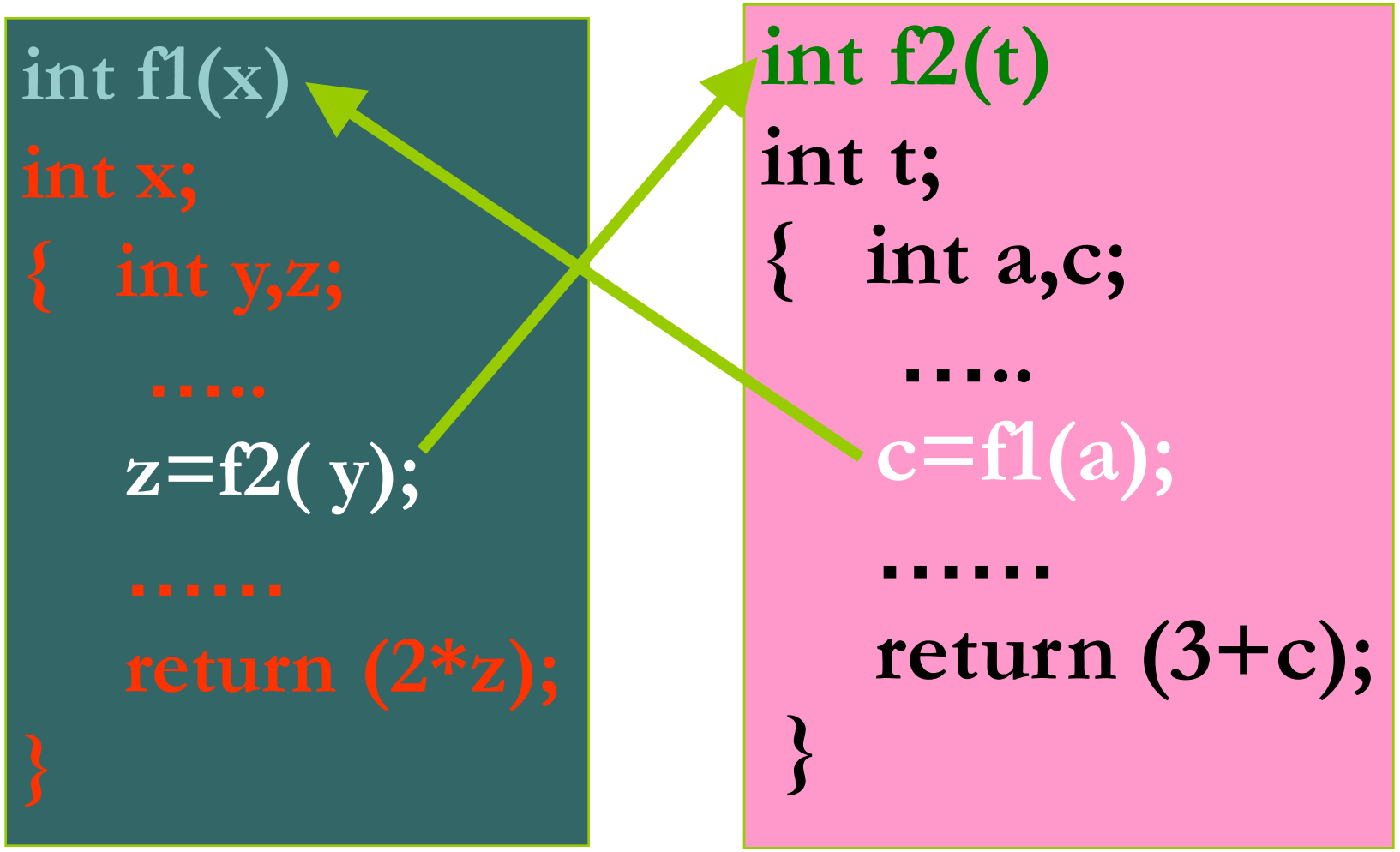
递归技术在算法和程序设计中是一种十分有用的技术，许多高级程序设计语言均提供了支持递归定义的机制和手段。

直接递归

```
int f(int x)
{   int y,z;
     .....
    z=f(x);
    ......
    return (2*z);
 }
```



■间接递归



- 上例是无终止的递归调用，实际应用中，应该给定一个限制递归次数的条件。

例1. 编写递归函数，求正整数n的阶乘值n!

用fact(n)表示n的阶乘值，据阶乘的数学定义可知:

$$fact(n) = \begin{cases} 1 & n=0 \\ n*fact(n-1) & n>0 \end{cases}$$

算法:

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

例2. 试编递归函数，求第n项Fibonacci级数的值。
假设用Fibona(n)表示第n项Fibonacci级数的值，
根据Fibonacci级数的计算公式:

$$\text{Fibona}(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ \text{Fibona}(n-1)+\text{Fibona}(n-2) & n>1\end{cases}$$

算法:

```
int Fibona ( int  n )
{  if  (n= =0)  return (0);
    else if (n= =1) return(1);
         else  return (Fibona(n-1)+ Fibona(n-2));
}
```

递归程序设计具有以下两个特点：

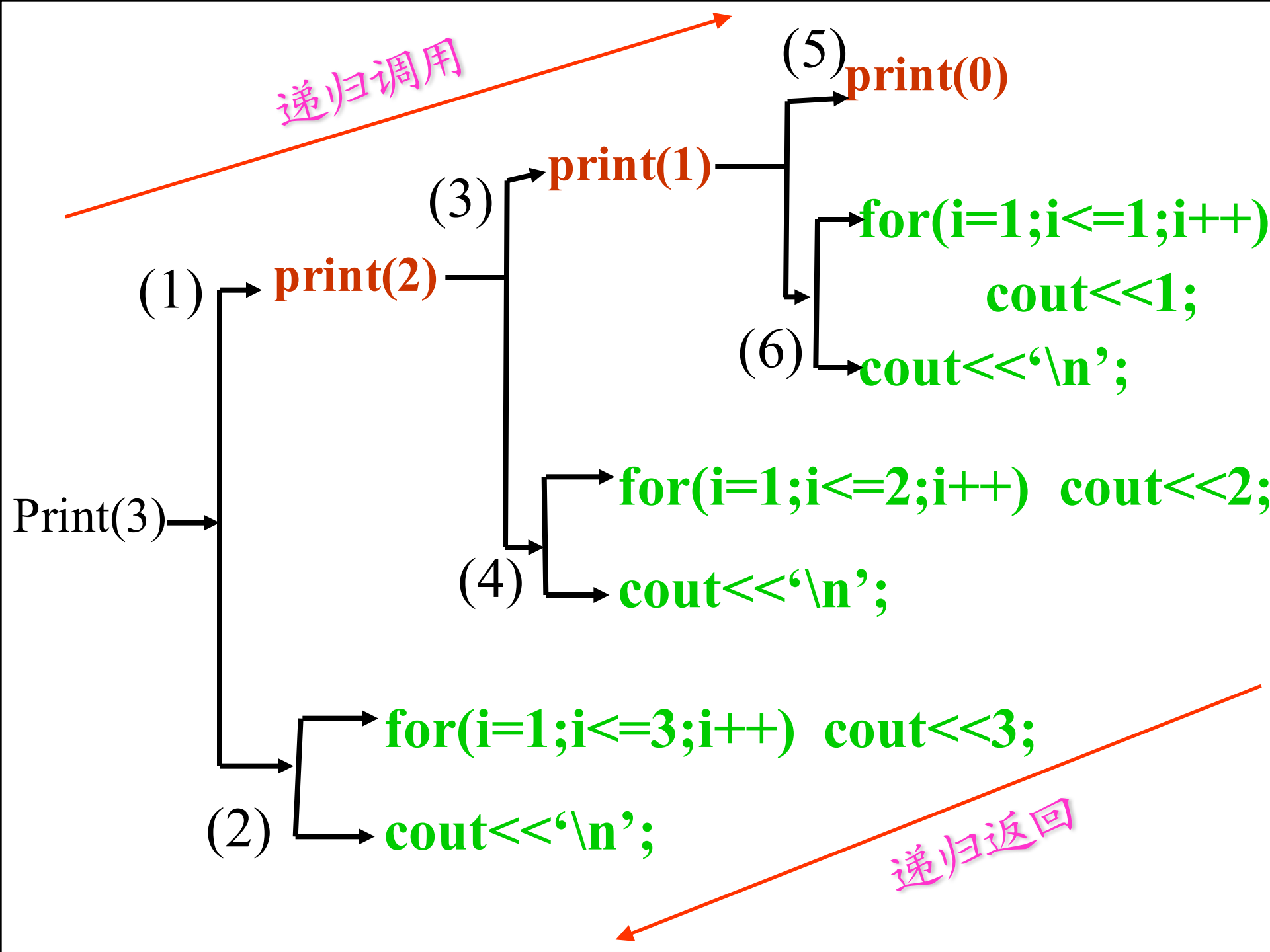
（1）具备递归出口。递归出口定义了递归终止条件，当程序的执行使条件满足时，递归过程便终止。递归出口可能不只一个；

（2）在不满足递归出口的情况下，根据所求解问题的性质，将原问题分解成若干子问题，子问题的结构与原问题相同，但规模较原问题小。

子问题的求解通过以一定的方式修改参数进行函数自身调用加以实现，然后将子问题的解组合成原问题的解。参数的修改最终必须保证递归出口得以满足。

读算法，写结果：

```
print (int n)
{  if (n!=0)
       {  print(n-1);
          for (i=1;i<=n;i++)
                cout<<n;
           cout<<'\n';
       }
}
```


递归调用
(5)
print(0)
print(1)
(3)
for(i=1;i<=1;i++)
cout<<1;
print(2)
(1)
(6)
cout<<'\n';
for(i=1;i<=2;i++) cout<<2;
Print(3)
(4)
cout<<'\n';
for(i=1;i<=3;i++) cout<<3;
(2)
cout<<'\n';
递归返回

功能：在第一行打印输出1个1，第二行打印输出2个2，……第n行打印输出n个n。

例如，当n=5时，调用该函数的输出结果为：

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

递归程序执行过程

在递归程序的运行过程中，系统内部设立了一个栈，用于存放每次函数调用与返回所需的各种数据。

主要包括：函数调用执行完成时的返回地址、函数的返回值、每次函数调用的实际参数和局部变量。

- 在递归程序的执行过程中，必须完成以下任务：

（1）计算当前被调用函数实际参数值；

（2）为当前被调用的函数分配存储空间，用于存放所需的各种数据，并将这片存储空间的首地址压入栈中；

（3）将当前被调用函数的实际参数、将来的返回地址等数据存入上述所分配的存储空间中；

（4）转到被调用函数的函数体，从其第一条可执行语句开始执行。

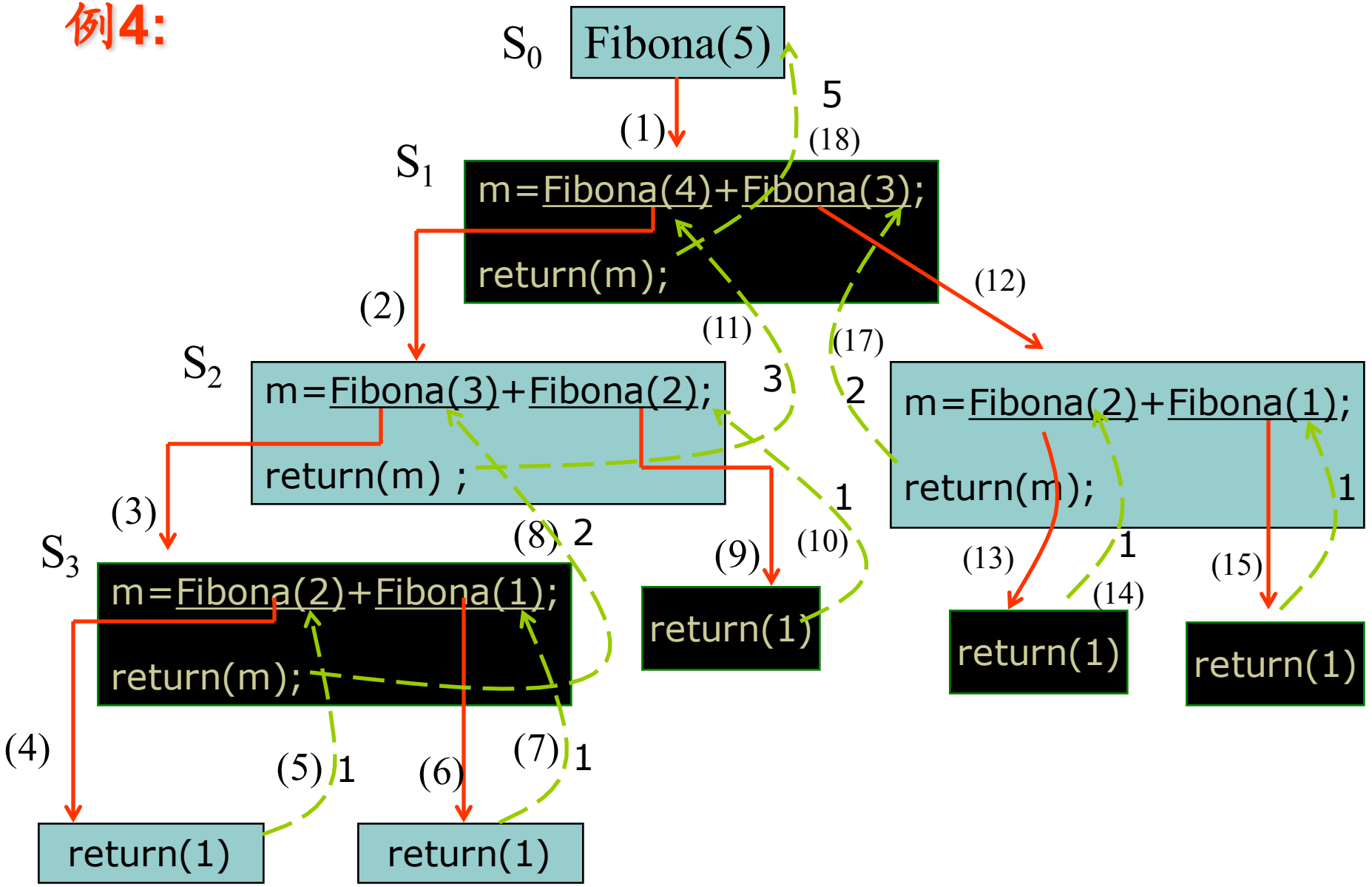
当从被调用函数返回时，必须完成以下任务：

（1）如果被调用函数有返回值，则记下该返回值，同时通过栈顶元素到该被调用函数对应的存储空间中取出其返回地址；

（2）把分配给被调用函数的那片存储空间回收，栈顶元素出栈；

（3）按照被调用函数的返回地址返回到调用点，若有返回值，还必须将返回值传递给调用者，并继续程序的执行。

例4:



Fibona(5)的执行过程

递归程序到非递归程序的转换

采用递归方式实现问题的算法程序具有结构清晰、可读性好、易于理解等优点，但递归程序较之非递归程序无论是空间需求还是时间需求都更高，因此在希望节省存储空间和追求执行效率的情况下，人们更希望使用非递归方式实现问题的算法程序；

另外，有些高级程序设计语言没有提供递归的机制和手段，对于某些具有递归性质的问题（简称递归问题）无法使用递归方式加以解决，必须使用非递归方式实现。因此，本小节主要研究递归程序到非递归程序的转换方法。

一般而言，求解递归问题有两种方式：

（1）在求解过程中直接求值，无需回溯。称这类递归问题为简单递归问题；

（2）另一类递归问题在求解过程中不能直接求值，必须进行试探和回溯，称这类递归问题为复杂递归问题。

两类递归问题在转换成非递归方式实现时所采用的方法是不同的。通常简单递归问题可以采用递推方法直接求解；而复杂递归问题由于要进行回溯，在实现过程中必须借助栈来管理和记忆回溯点。

简单递归程序到非递归程序的转换

采用递归技术求解问题的算法程序是自顶向下产生计算序列，其缺点之一是导致程序执行过程中许多重复的函数调用。

递推技术同样以分划技术为基础，它也要求将需求解的问题分划成若干与原问题结构相同、但规模较小的子问题;

与递归不同，递推方法采用自底向上的方式产生计算序列，其首先计算规模最小的子问题的解，然后在此基础上依次计算规模较大的子问题的解，直到最后产生原问题的解。

由于求解过程中每一步新产生的结果总是直接以已有计算结果为基础，避免了许多重复计算，因而递推算法程序比递归算法具有更高的效率。

例5. 采用非递归实现求n的阶乘值。

仍使用Fact(n)表示n的阶乘值。要求解Fact(n)的值，可以考虑i从0开始，依次取1, 2, ……，一直到n，分别求Fact(i)的值，且保证求解Fact(i)时总是以前面已有的求解结果为基础；当i=n时，Fact(i)的值即为所求的Fact(n)的值。

根据阶乘的递归定义，不失一般性，显然有以下递推关系成立：

$$Fact(i)=\begin{cases}1 & i=0\\ i*Fact(i-1) & i>0\end{cases}$$

阶乘问题的非递归算法的实现如下：

```
int Fact ( int n )
 { int i,fac;
   fac=1;   //将fac初始化为Fact(0)的值
   for (i=1;i<=n; ++i) fac = i*fac;
                //根据递推关系进行递推
   return(fac);
 }
```

复杂递归程序到非递归程序的转换

复杂递归问题在求解的过程中无法保证求解动作一直向前，往往需要设置一些回溯点，当求解无法进行下去或当前处理的工作已经完成时，必须退回到所设置的回溯点，继续问题的求解。

因此，在使用非递归方式实现复杂递归问题的算法时，经常使用栈来记录和管理所设置的回溯点。

例6:

按中点优先的顺序遍历线性表：已知线性表list以顺序存储方式存储，要求按以下顺序输出list中所有结点的值：首先输出线性表list中点位置元素值，然后输出中点左部所有元素的值，再输出中点右部所有元素的值；而无论输出中点左部元素还是输出中点右部元素，均应遵循以上规律。

例如，已知数组list中元素的值为：

18 32 4 9 26 6 10 30 12 8 45

则按中点优先顺序遍历的输出结果为：

6 4 18 32 9 26 12 10 30 8 45

递归算法:

```
typedef int listarr[20];
void listorder(listarr list, int left,
int right)
{  int mid;
   if (left<=right)
     { mid=(left+right)/2;
       printf("%4d",list[mid]);
       listorder(list,left,mid-1);
       listorder(list,mid+1,right);
     }
}
```

非递归实现：在线性表的遍历过程中，输出中点的值后，中点将线性表分成前半部分和后半部分。

进入前半部的遍历之前，应该将后半部保存起来，以便访问完前半部后，再进入后半部的访问，即在此设置一个回溯点并进栈保存，实现时，将后半部起点和终点下标进栈即可，栈中的每个元素均代表一个尚未处理的数组段。

对于每一个当前正在处理的数组段均应采用以上相同的方式进行处理，直到当前正在处理的数组（数组段）为空，则进行回溯，而回溯点恰巧位于栈顶。只要取出栈顶元素，将它所确定的数组段作为下一步即将遍历的对象，继续线性表的遍历，直到当前正在处理的数组段为空且栈亦为空，算法结束。

```
void listorder(listarr list,int left, int
right)
{ typedef struct {
        int l; //存放待处理数组段的起点下标
        int r; //存放待处理数组段的终点下标
  }stacknode; //栈中每个元素的类型
  stacknode stack[20];
  int top,i,j,mid;  //top为栈顶指针

  if (left<=right)  //数组段不为空
     { top= -1;    i=left; j=right;
       while (i<=j || top!=-1)
           {//当前正在处理的数组段非空或栈非空
```

```
    if (i<=j)
        {mid=(i+j)/2;
         printf("%4d",list[mid]);
         ++top;        stack[top].l=mid+1;
         stack[top].r=j;      j=mid-1;
         } //end if
    else //当前正在处理的数组段为空,回溯
        {i=stack[top].l;
         j=stack[top].r;      --top;
         } //end else
  } //end while
 } //end if
} //end
```

hanoi塔问题

例：汉诺塔问题——有A,B,C三个塔座，A上套有n个直径不同的圆盘，按直径从小到大叠放，形如宝塔,编号1,2,3……n。要求将n个圆盘从A移到C，叠放顺序不变。

汉诺塔问题的来源：

- 汉诺塔问题是印度的一个古老的传说。
- 开天辟地的神勃拉玛在一个庙里留下了三根金刚石的棒，第一根上面套着64个圆的金片，最大的一个在底下，其余一个比一个小，依次叠上去。
- 庙里的众僧不倦地把它们一个个地从这根棒搬到另一根棒上，规定可利用中间的一根棒作为帮助，但每次只能搬一个，而且大的不能放在小的上面。
- 经计算，移动圆片的次数为：

$$18,446,744,073,709,551,615$$

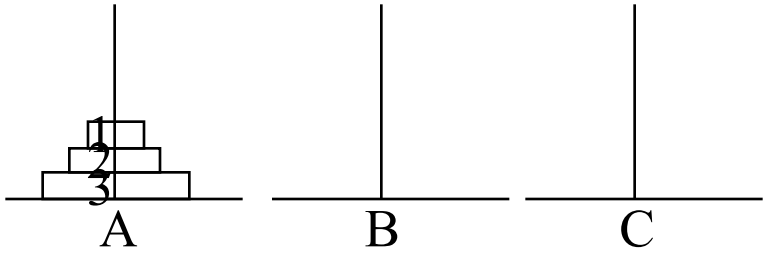
移动原则：

- 每次只能移一个圆盘
- 圆盘可在三个塔座上任意移动
- 任何时刻，每个塔座上不能将大盘压到小盘上

解决方法：

- n=1时，直接把圆盘从A移到C
- n>1时，先把上面n-1个圆盘从A移到B，然后将n号盘从A移到C，再将n-1个盘从B移到C。即把求解n个圆盘的Hanoi问题转化为求解n-1个圆盘的Hanoi问题，依次类推，直至转化成只有一个圆盘的Hanoi问题

```
main()
    { int m;
      printf("Input number of disks");
      scanf("%d",&m);
      printf(" Steps : %3d disks",m);
      hanoi(m,'A','B','C');
(0) }
```

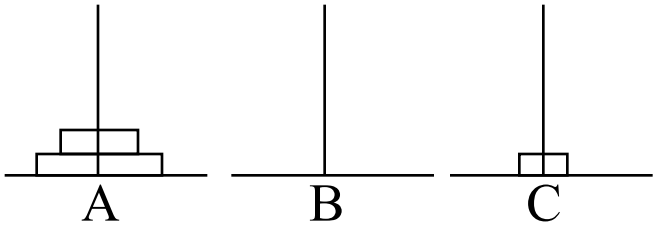



```
void hanoi(int n,char x,char y,char z)
(1)  {
(2)     if(n==1)
(3)        move(1,x,z);
(4)     else{
(5)           hanoi(n-1,x,z,y);
(6)           move(n,x,z);
(7)           hanoi(n-1,y,x,z);
(8)        }
(9)  }
```

3	A	B	C	0

2	A	C	B	6
3	A	B	C	0

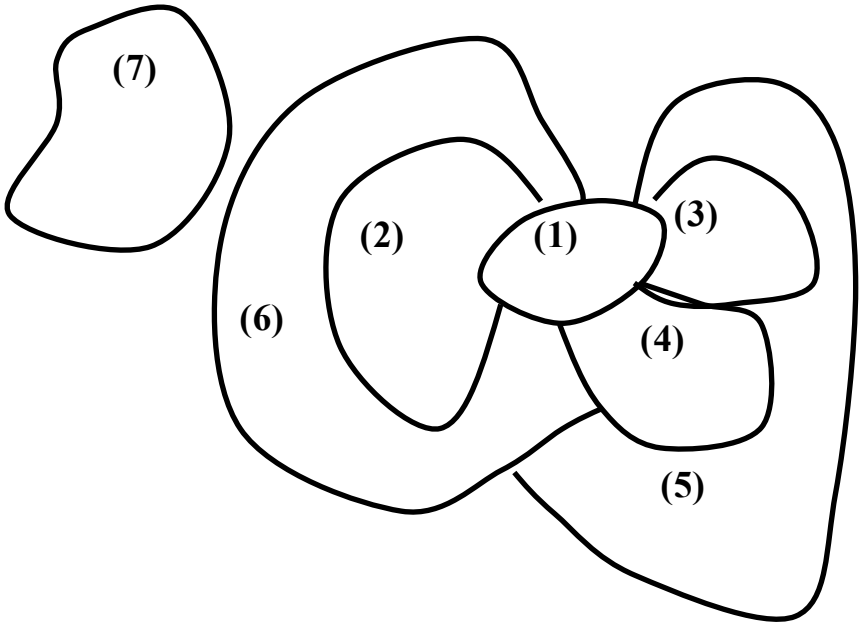
1	A	B	C	6
2	A	C	B	6
3	A	B	C	0



2	A	C	B	6
3	A	B	C	0

思考问题： 地图四染色问题

“四染色”定理是计算机科学中著名的定理之一。“四染色”定理——使地图中相邻的国家或行政区域不重色，最少可用四种颜色对地图着色。证明此定理的结论，利用栈采用回溯法对地图着色。



1# 粉红

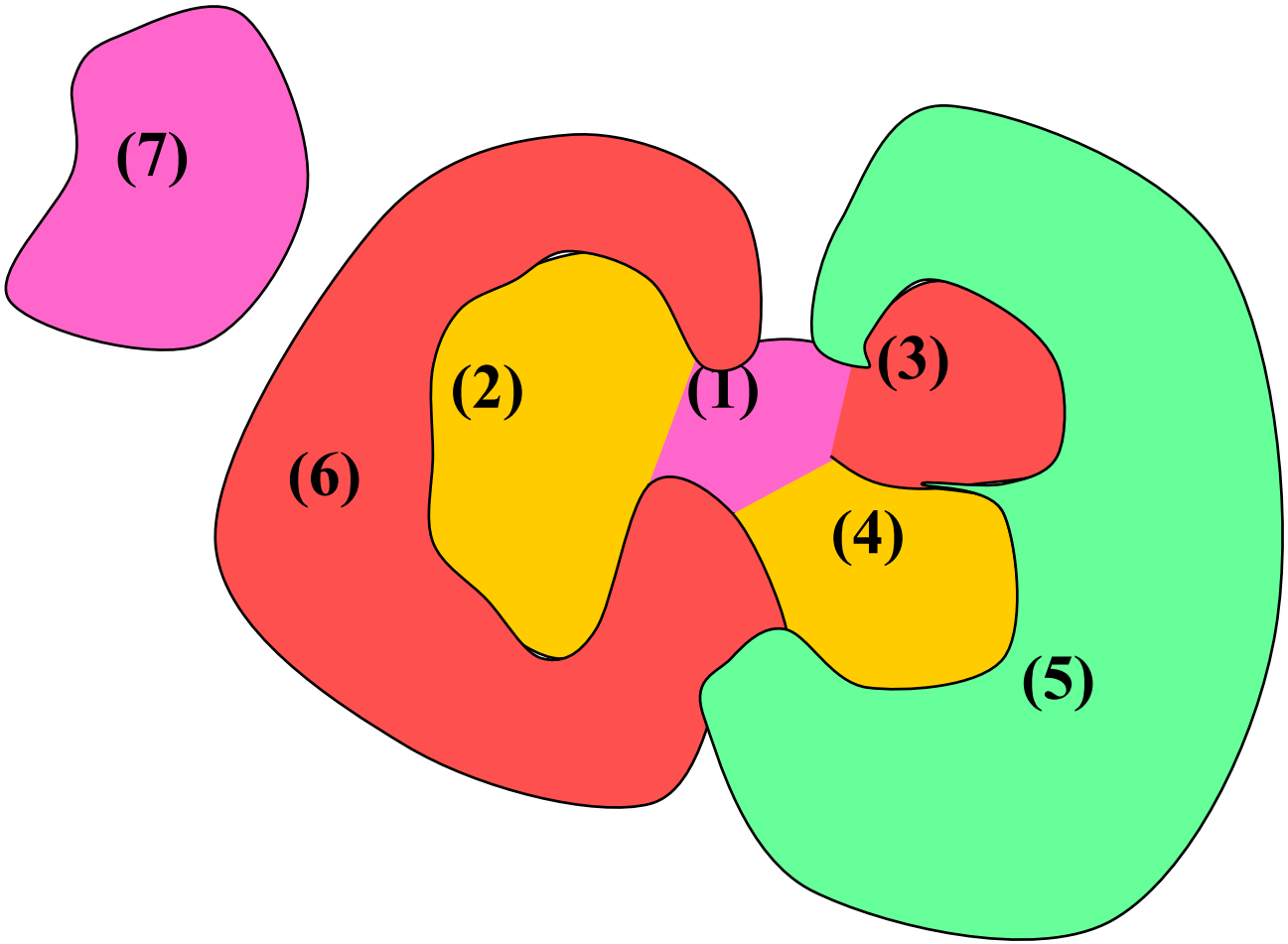
2# 黄色

3# 红色

4# 绿色

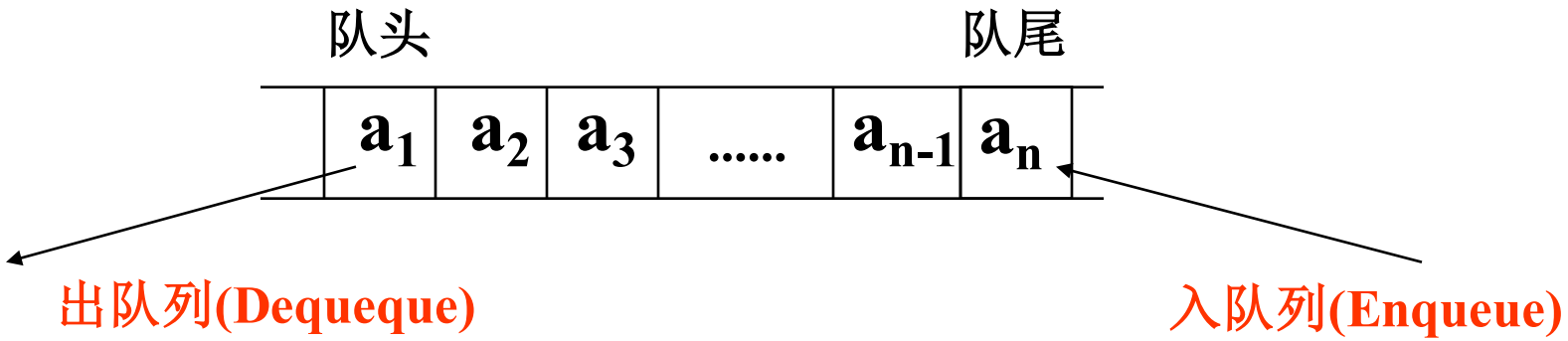
a 粉色　　b 黄色

c 红色　　d 绿色



3.4 队列

- 队列（Queue）：先进先出(First In First Out)(缩写为FIFO）的线性表。
  仅在队尾进行插入和队头进行删除的线性表。
- 队头（front）：线性表的表头端，即可删除端。
- 队尾（rear）：线性表的表尾端，即可插入端。



队列的抽象数据类型

- **ADT Queue {**
- 数 据 对 象 ： D = {a_i | a_i 属 于 Elemset, (i=1, 2, …, n, n≥0)}
- 数 据 关 系 ： R_1 = { < a_{i-1},a_i > |a_{i-1},a_i 属 于 D,(i=2,3,…,n)} 约定a_n为队尾, a_1为队头。
- 基本操作：
  - InitQueue(&Q); DestroyQueue (& Q);
  - QueueLength(Q) ; GetHead(Q,&e);
  - EnQueue (& Q,e); DeQueue (& Q,&e);
  - QueueTraverse(Q ,visit ())
- **}ADT Queue**

队列的基本操作(之一)

- **InitQueue (&Q)**
  - 操作结果:构造一个空的队列Q。
- **DestroyQueue (&Q)**
  - 初始条件: 队列Q已经存在。
  - 操作结果: 销毁队列Q。
- **ClearQueue (&Q)**
  - 初始条件: 队列Q已经存在。
  - 操作结果: 将队列Q重置为空队列。

队列的基本操作(之二)

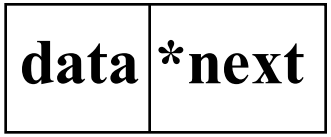
- **QueueEmpty(Q)**
  - 初始条件: 队列Q已经存在。
  - 操作结果: 若队列Q为空队列，则返回TURE; 否则返回FALSE。
- **QueueLength(Q)**
  - 初始条件: 队列Q已经存在。
  - 操作结果: 返回队列Q中的数据元素个数, 即队列Q的长度。
- **GetHead(Q,&e)**
  - 初始条件: 队列Q已经存在且非空。
  - 操作结果: 用e返回队列Q中队头元素的值。

队列的基本操作(之三)

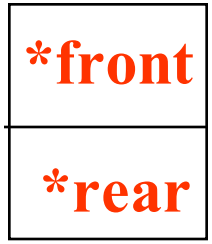
- **EnQueue (&Q,e)**
  - 初始条件: 队列Q已经存在。
  - 操作结果: 插入元素e为新的队尾元素。
- **DeQueue (&Q,&e)**
  - 初始条件: 队列Q已经存在且非空。
  - 操作结果: 删除队列Q 的队头元素并用e返回其值。
- **QueueTraverse(Q,visit ())**
  - 初始条件: 队列Q已经存在且非空。
  - 操作结果: 从队头到队尾依次对队列Q的每个元素调用函数visit ()。

3.4.2 链队列（队列的链式表示与实现）

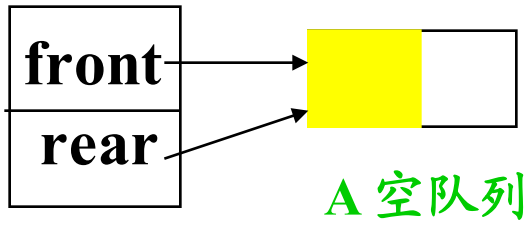
```
typedef struct QNode{
    QElemType data;
    struct QNode *next;
}Qnode, *QueuePtr;
```



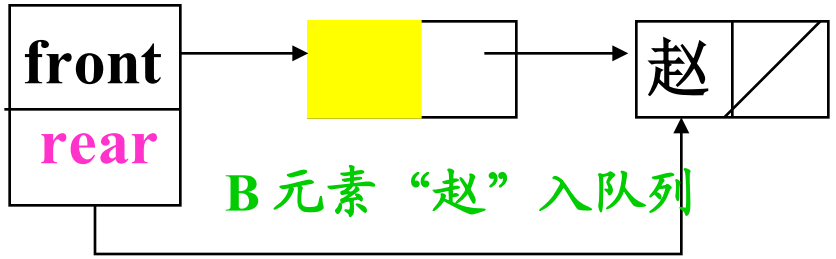

```
typedef struct {
    QueuePtr front; //队头指针
    QueuePtr rear;  //队尾指针
}LinkQueue;
```



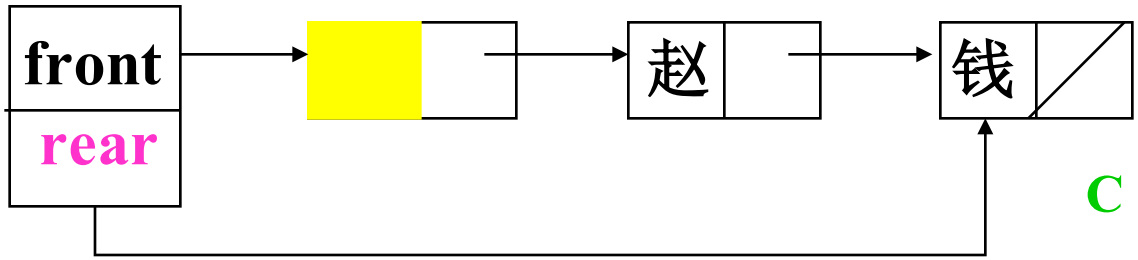
链队列示意图



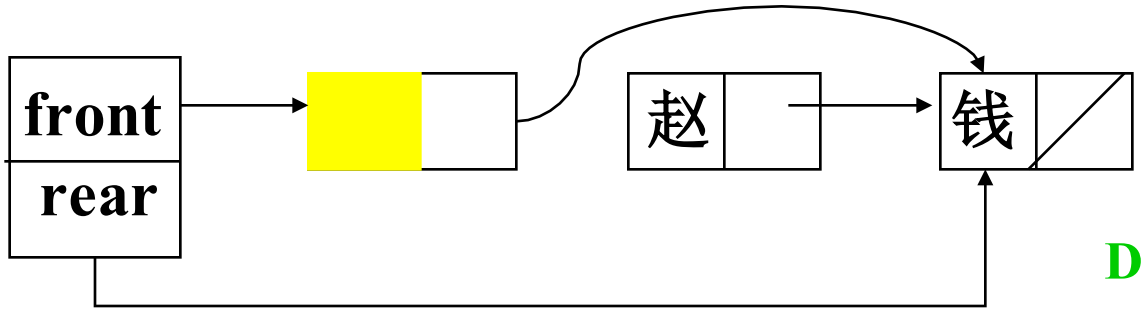
A 空队列



B 元素“赵”入队列



C 元素“钱”入队列



D 元素“赵”出队列

链队列的操作实现举例

```
Status InitQueue(LinkQueue &Q)
{//构造一个空队列Q
  Q.rear=(QueuePtr)malloc(sizeof(QNode));
  Q.front = Q.rear;
  if(!Q.front)  return(OVERFLOW);
  Q.front -> next = NULL;
  return OK;
} // InitQueue
```

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

```
Status EnQueue(LinkQueue &Q, QelemType e)
{ //插入元素e为Q的新的队尾元素
    p = ( QueuePtr )malloc(sizeof(QNode));
    if(!p) return(OVERFLOW);
    p->data = e;
    p -> next = NULL;
    Q.rear ->next = p;
    Q.rear = p;
    return OK;
} // EnQueue
```

```
Status DeQueue(LinkQueue &Q, QelemType &e)
{ //若队列不空，则删除Q的队头元素，用e返回其
    值，并返回OK；否则返回ERROR
    if(Q.front == Q.rear) retrun ERROR;
    p = Q.front -> next;
    e = p->data;
    Q.front->next = p->next;
    if(Q.rear == p) Q.rear = Q.front ;
    free(p);
    return OK;
} // DeQueue
```

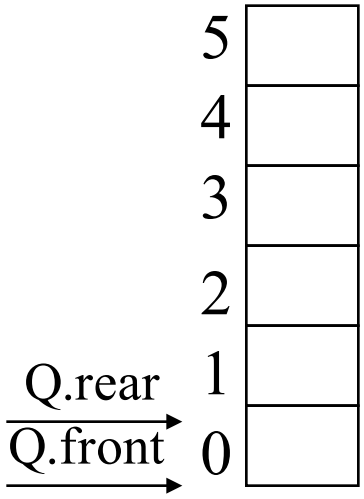
3.4.3 循环队列（队列的顺序表示与实现）

采用顺序存储结构

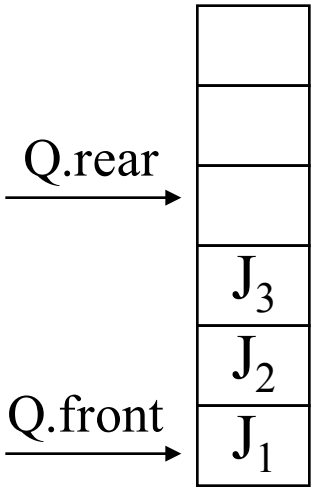
约定: 1)初始空队列： front = rear=0 ；

2)插入新的元素时, rear++;

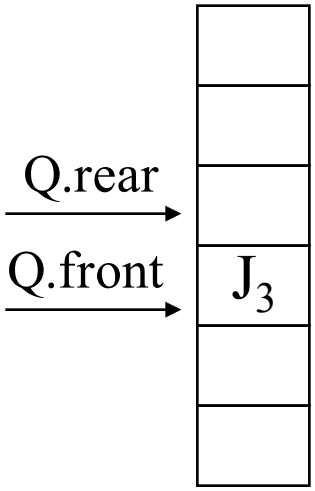
3)删除队头元素时, front++;



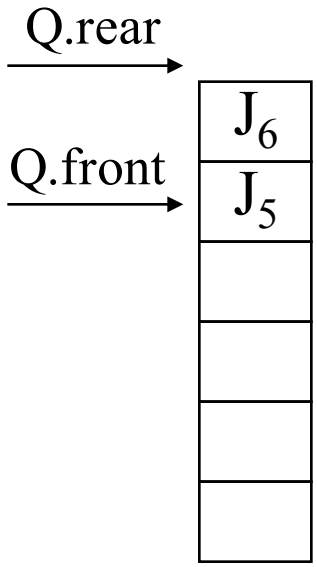
空队列



J1、J2和J3相继入列

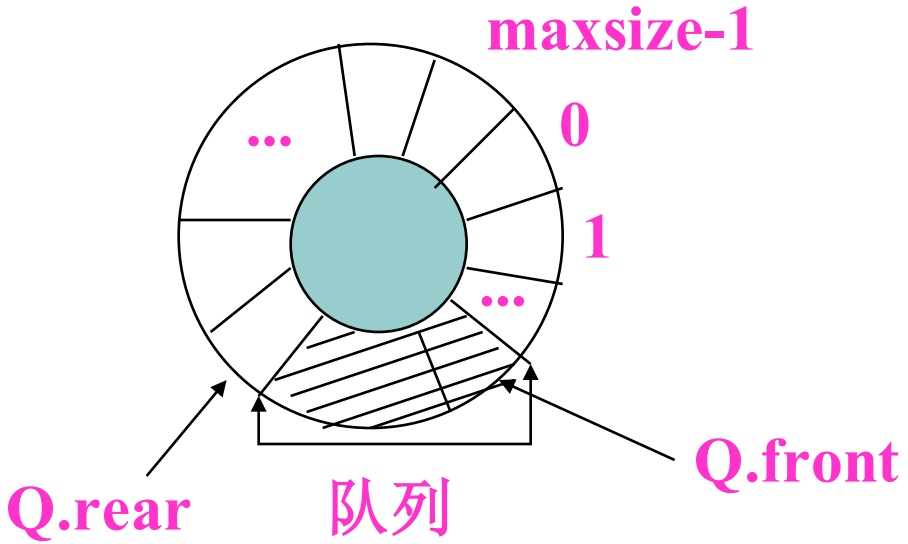


J1、J2相继被删除



J4、J5和J6相继入列后J4被删除

循环队列
（解决数组越界但未占满空间的办法）



当Q.rear > Q.front时

Q.rear – Q.front ＝元素个数

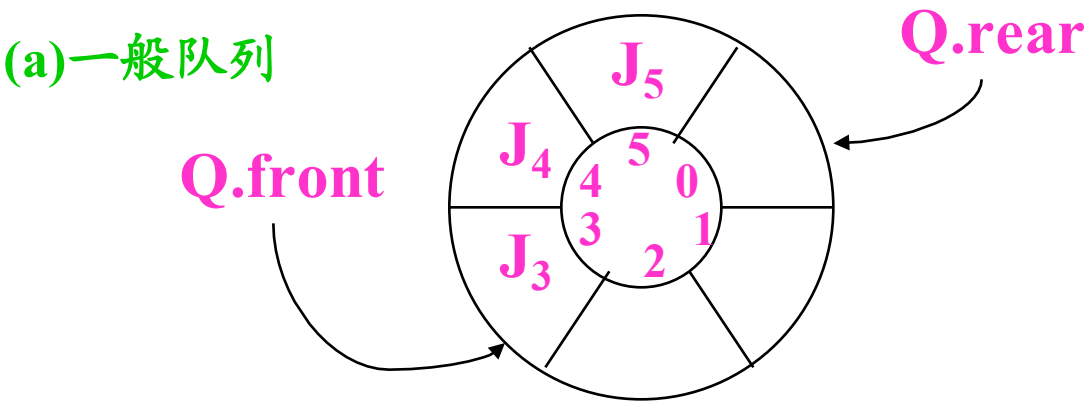
当Q.rear < Q.front时:

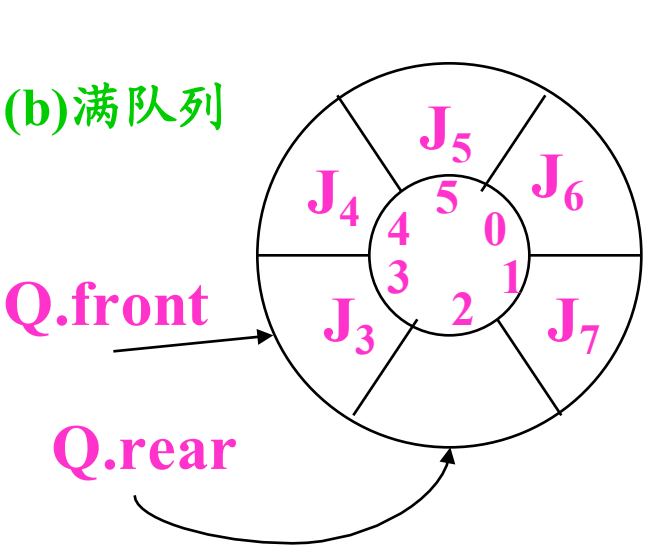
Q.rear – Q.front + maxsize ＝元素个数

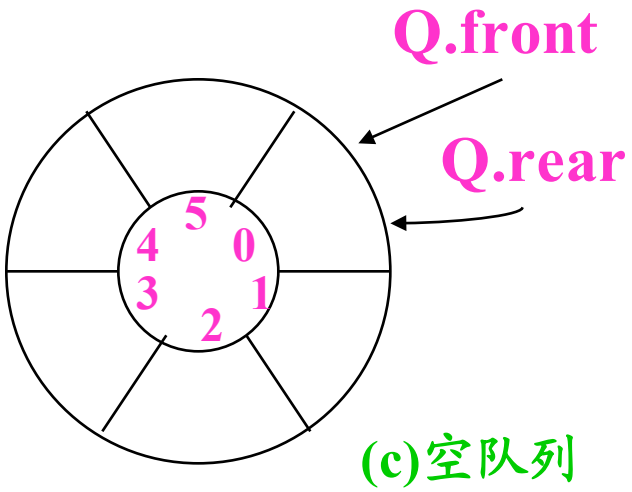
当Q.rear = Q.front时:

队列是’空’或’满’

循环队列的头尾指针







满： Q.rear=Q.front－1

空： Q.rear=Q.front

顺序结构循环队列操作实现举例

```
#define MAXSIZE 100
                                    //最大队列长度
typedef struct {
    QElemType  *base;
                                    //存储空间基地址
    int front;                      //队头指针
    int rear;                       //队尾指针
}SqQueue;
```

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

```
Status EnQueue(SqQueue &Q, QelemType e)
{  //插入元素e为新的队尾元素
   if ((Q.rear+1)% MAXSIZE == Q.front)
       return(ERROR);
   Q.base[Q.rear] = e;
   Q.rear = (Q.rear+1) % MAXSIZE;
   return OK;
} // EnQueue
```

```
Status DeQueue(SqQueue &Q, QelemType &e)
{//若队列不空，则删除Q的队头元素，用e返
   回其值，并返回OK；否则返回ERROR
    if (Q.front == Q.rear) retrun ERROR;
    e = Q.base[Q.front];
    Q.front = (Q.front+1) % MAXSIZE;
    return OK;
} // DeQueue
```