数据结构与算法

Data Structures and Algorithms

第二部分 线性表

回顾：线性表的逻辑结构和物理结构

1. 线性表抽象数据型及操作

线性表 LIST = (D , R)

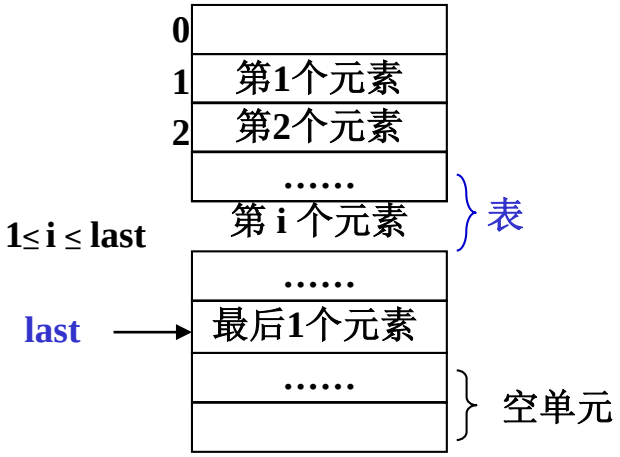
$D = \{ a_i \mid a_i \in \text{Elementset}, i = 1, 2, \ldots, n, n \geq 0 \}$

$R = \{ H \}$

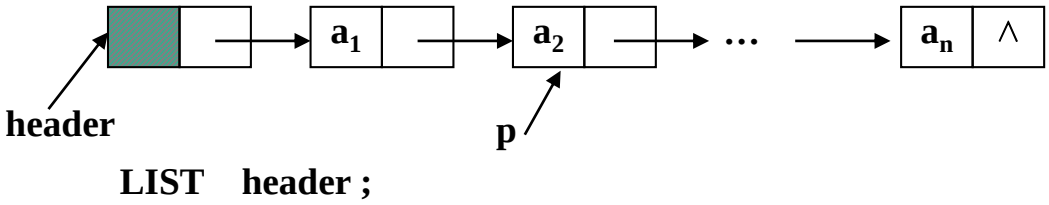
$H = \{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i = 2, \ldots, n \}$

2. 线性表的实现

（1）顺序表--线性表的顺序表示（2）单链表--线性表的指针表示



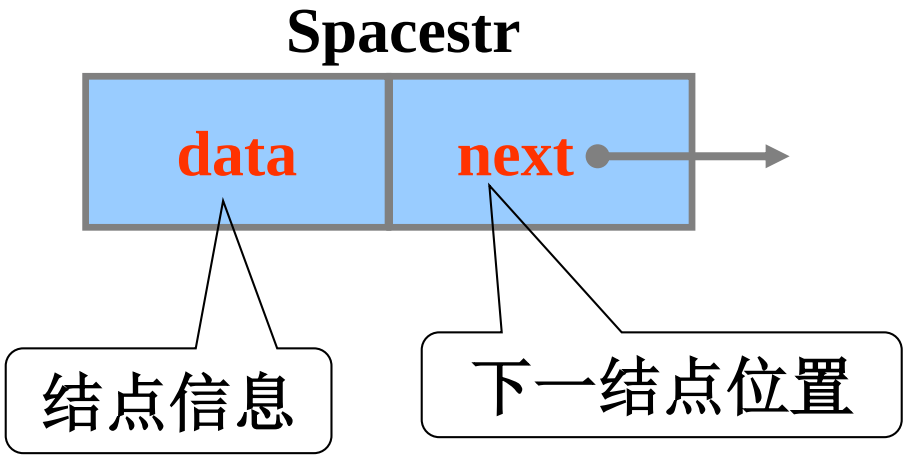
线性表的数组实现



2.2.4 静态链表--线性表的游标实现

借助数组来描述线性表的链式存储结构，节点包含data和next。

结点形式



结点类型

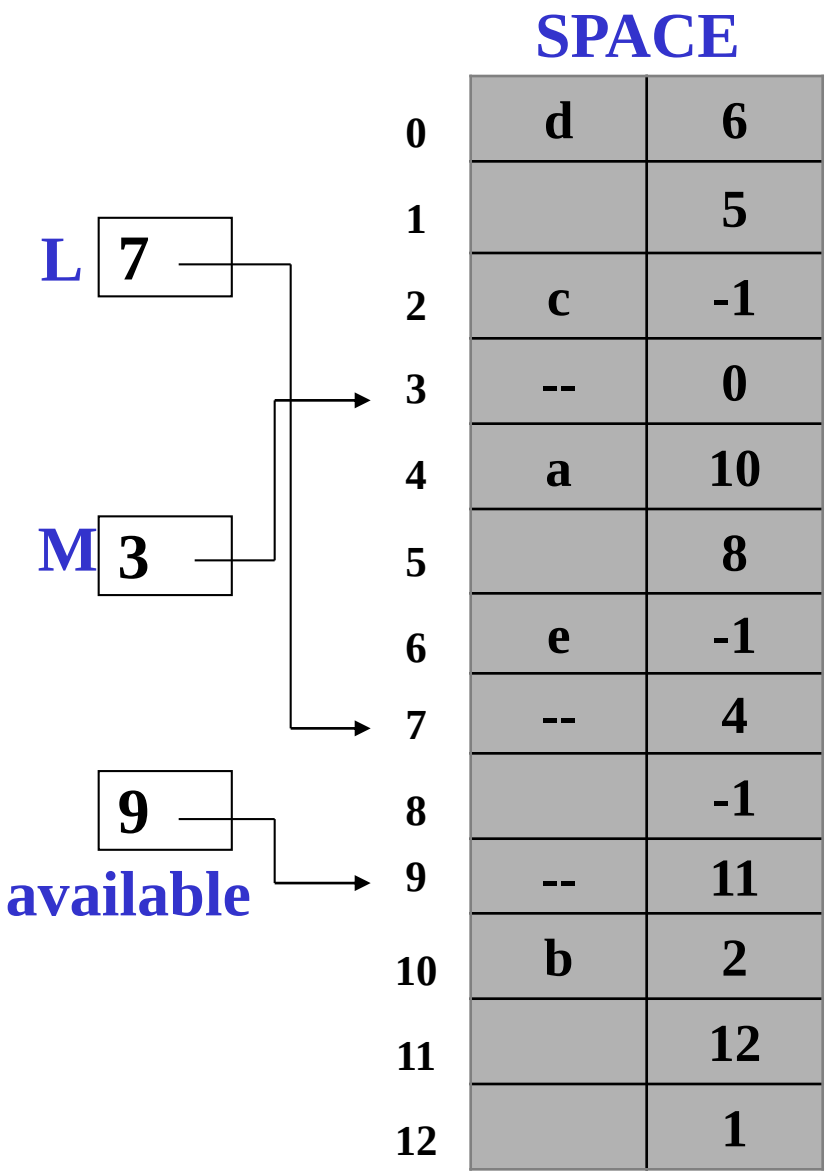
```
struct  Spacestr {
    ElementType  data ;
    int    next ;
} ;
```

线性表：

```
Spacestr  SPACE[ maxsize ] ;
typedef  int  Position;
```

元素：SPACE[i].data → “型”为ElementType(基本、复合)

下一元素位置：SPACE[i].next → “型”为 int



线性表有哪些呢？

线性表1：（a, b, c）　L = 7

线性表2：（d, e）　M = 3

空闲表：available = 9

```
x = New Spacestr ;//自己定义
    x = SPACE[available].next ;
    SPACE[available].next =
        SPACE[ x ].next ;
```

```
Delete  x ;
    SPACE [ x ].next =
        SPACE[available].next ;
    SPACE[available].next = x ;
```

ADT操作：

① Void Insert (ElementType x, Position p, Spacestr *SPACE)

```
{ Position q ;
  q = New Spacestr ;
  SPACE[ q ].data = x ;
  SPACE[ q ].next = SPACE[ p ].next ;
  SPACE[ p ].next = q ;
}
```

② Void Delete (Position p, Spacestr *SPACE)

```
{ Position q ;
  if ( SPACE[ p ].next != -1 )
     { SPACE[ q ] = SPACE[ p ].next ;
       SPACE[ p ].next = SPACE[ q ].next ;
       Delete q ;                    }
} ;
```

例：A=(c, b, e, g, f, d)，B=(a, b, n, f)，求(A-B) U (B-A)。也就是要插入a、删除b、插入n、删除f。S: 1

Space(0:11)

0		8
1		2
2	c	3
3	b	4
4	e	5
5	g	6
6	f	7
7	d	0
8		9
9		10
10		11
11		0

Space(0:11)

0		9
1		2
2	c	3
3	b	4
4	e	5
5	g	6
6	f	7
7	d	8
8	a	0
9		10
10		11
11		0

Space(0:11)

0		3
1		2
2	c	4
3	b	9
4	e	5
5	g	6
6	f	7
7	d	8
8	a	0
9		10
10		11
11		0

Space(0:11)

0		9
1		2
2	c	4
3	n	8
4	e	5
5	g	6
6	f	7
7	d	3
8	a	0
9		10
10		11
11		0

Space(0:11)

0		6
1		2
2	c	4
3	n	8
4	e	5
5	g	7
6		9
7	d	3
8	a	0
9		10
10		11
11		0

例题

已知表头元素是c的单链表在内存中的存储状态如下表所示。

地址	元素	链接地址
1000H	a	1010H
1004H	b	100CH
1008H	c	1000H
100CH	d	NULL
1010H	e	1004H
1014H		

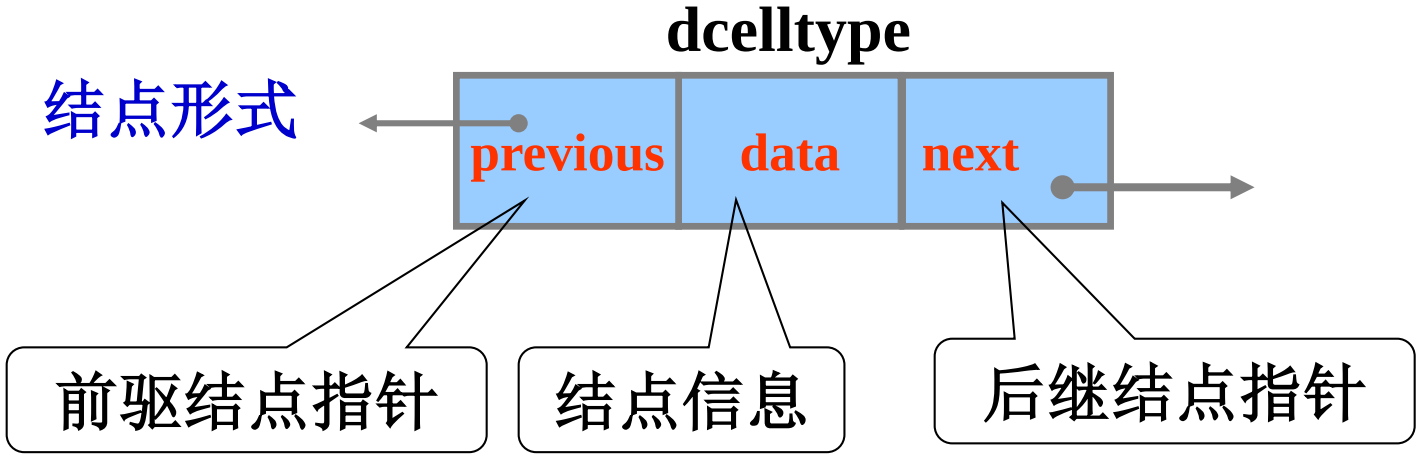
现将f存放于1014H处并插入到单链表中，若f在逻辑上位于a与e之间，则a，e，f的“链接地址”依次是（）

A. 1010H, 1014H, 1004H　　B. 1010H, 1004H, 1014H

C. 1014H, 1010H, 1004H　　D. 1014H, 1004H, 1010H

2.2.5 双向链表

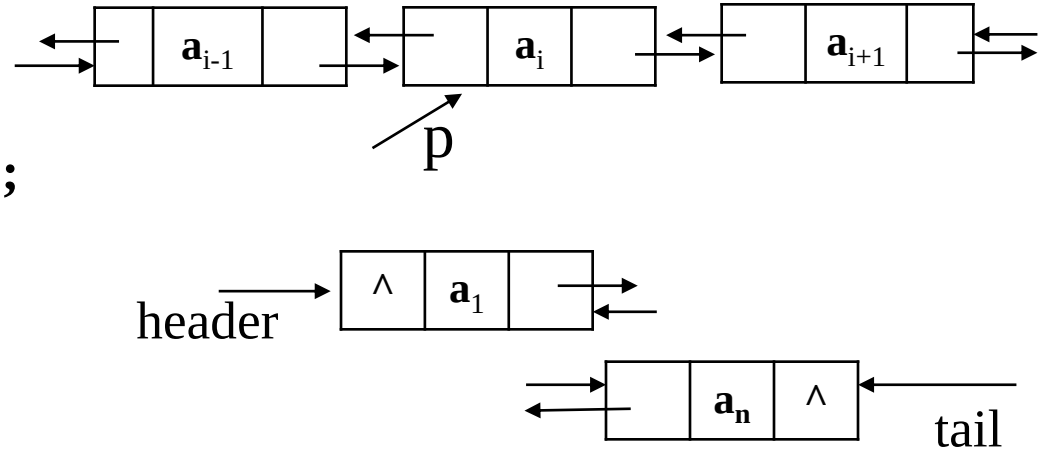
结点形式

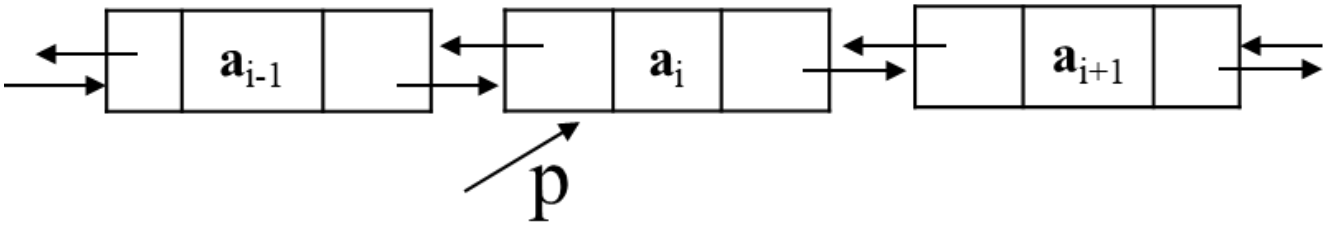


结点类型

```
struct DnodeType {
   ElementType data ;
   DnodeType *next, *previous ;
} ;

typedef DnodeType *DLIST ;
typedef DnodeType *Position ;
```





删除位置p的元素：

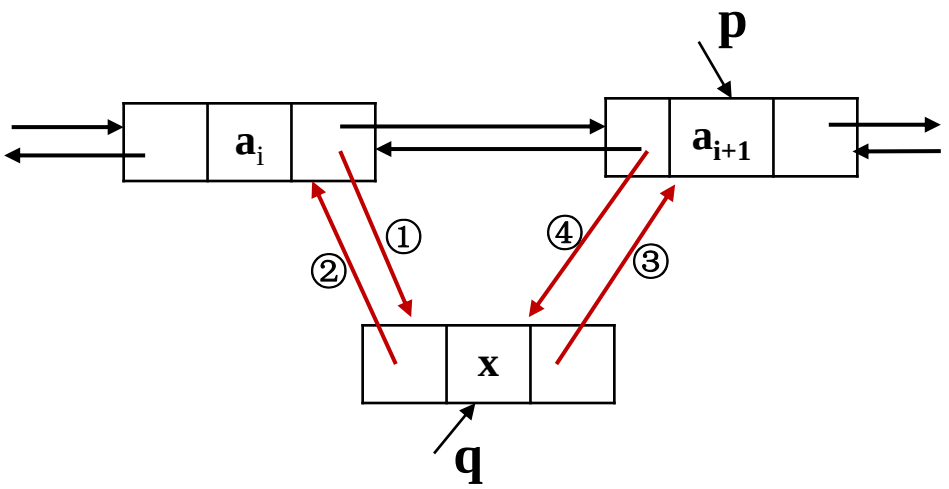
p->previous->next = p->next;

p->next->previous = p->previous ;

free(p) / delete p ;

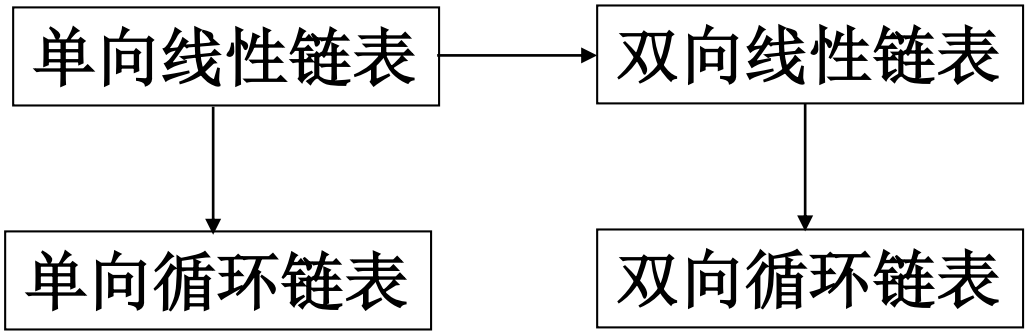
ADT操作：

```
Void  Insert( ElementType x, Position p,  DLIST  &L ) ;
{
   Position  q ;
   q = new DNodeType ;
   q->data = x ;
   ①  p->previous->next = q ;
   ②  q->previous = p->previous ;
   ③  q->next = p ;
   ④  p->previous = q ;
} ;
```



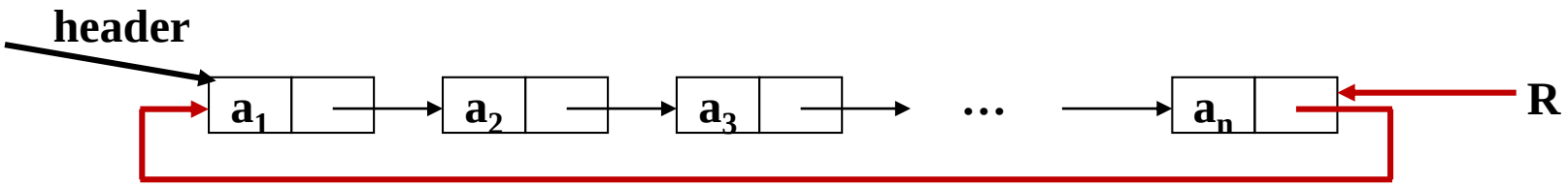
讨论：元素的位置？

2.2.6　环形链表



对线性链表的改进，解决“单向操作”的问题；改进后的链表，能够从任意位置元素开始访问表中的每一个元素。

单向环形链表：



其结构与单向线性链表相同，用表尾指针标识链表，从而省去了表头结点。

表头：R→next　亦即表中的一个元素a_1

ADT操作：①在表左端插入结点Insert(x,First(R),R);

```
Void  LInsert( ElementType  x , LIST  &R )
{   celltype  *p ;
    p = New  NODE ;
    p→data = x ;
    if ( R == Null )
        {   p→next = p ;   R = p ;  }
    else
        {   p→next = R→next ;  R→next = p ;  }
 } ;
```

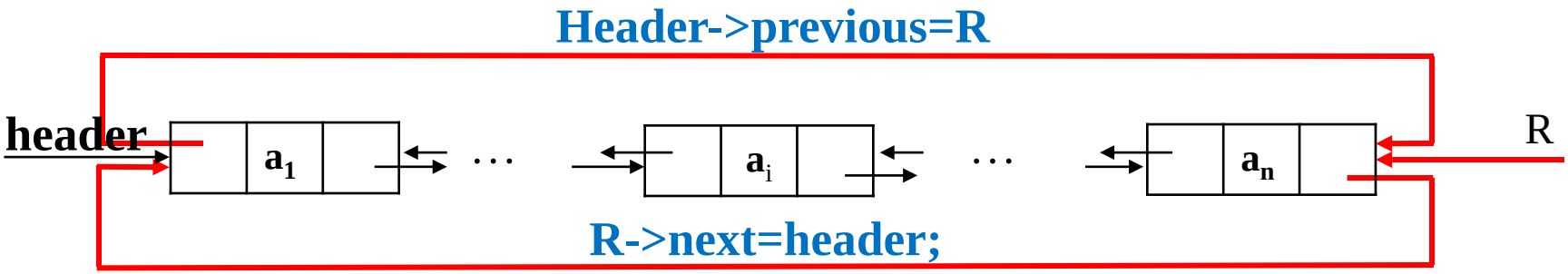
②在表右端插入结点Insert(x,End(R),R); →RInsert(x,R)

```
Void   RInsert( ElementType  x , LIST  R )
{
    LInsert ( x , R ) ;  R = R→next ;
 }
```

③从表左端删除结点Delete(First(R),R); →LDelete(R)

```
Void  LDelete( LIST  &R )
{   struct NODE  *p ;
    if ( R == Null )
        error ( “空表” );
    else
        {   p = R→next ;
            R→next = p→next ;
            if ( p == R )
               R=Null;
            Delete  p ;
        }
};
```

双向环形链表

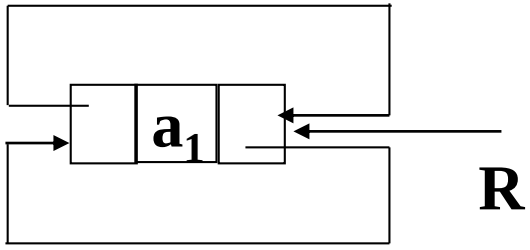


双向环形链表的结构与双向连表结构相同，只是将表头元素的空previous域指向表尾，同时将表尾的空next域指向表头结点，从而形成向前和向后的两个环形链表，对链表的操作变得更加灵活。

空双向环形链表：

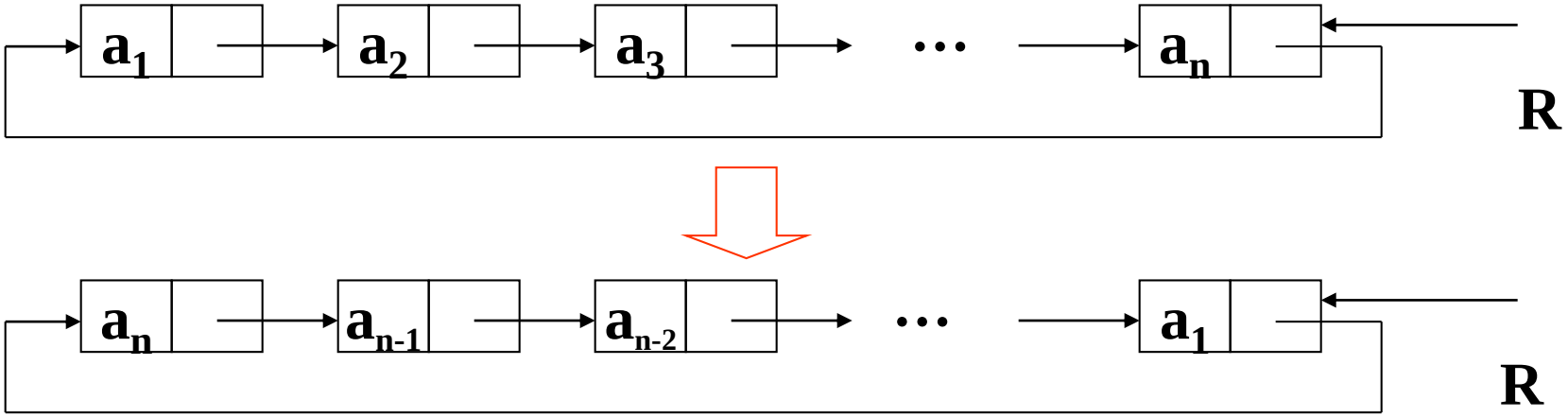
R=Null；

单一结点双向环形链表：



R->next=R;
R->previous=R;

【例2-7】设计算法，将一个单向环形链表反向。头元素变成尾元素，尾元素变成新的头元素，依次类推。



算法如下：

```
Void  Revers( LIST  &R )
{  Position  p, q;
   p = R→next ;
   R→next = p→next ;
   p→next = p ;
   while ( R != Null )
         {  q = R→next ;
            R→next = q→next ;
            q→next = p→next ;
            p→next = q ;
         }
   R = p ;
} ;
```

???r->next!=R
然后单独处理R

例题 已知一个带有表头节点的双向链表L，结点结构为

其中，prev和next分别是指向其直接前驱和直接后继结点的指针。现要删除指针p所指向的结点，正确的语句是（ ）

A. p->next->prev = p->prev; p->prev->next = p->prev; free(p);

B. p->next->prev = p->next; p->prev->next = p->next; free(p);

C. p->next->prev = p->next; p->prev->next = p->prev; free(p);

D. p->next->prev = p->prev; p->prev->next = p-> next; free(p);

一分为二看链表 .....

链表存在的三个问题：

1.首先，每创建一个结点，都要进行一次系统调用分配一块空间，这会浪费一定时间；

2.其次，由于创建结点的时间不固定，会导致结点分配的空间不连续，容易形成离散的内存碎片；

3.最后，由于内存不连续，所以链表的局部访问性较差，容易出现cache（高速缓冲存储器）缺失。

对链表的改进---内存池（静态空间动态化）

特殊线性表或受限制的线性表

栈和队列

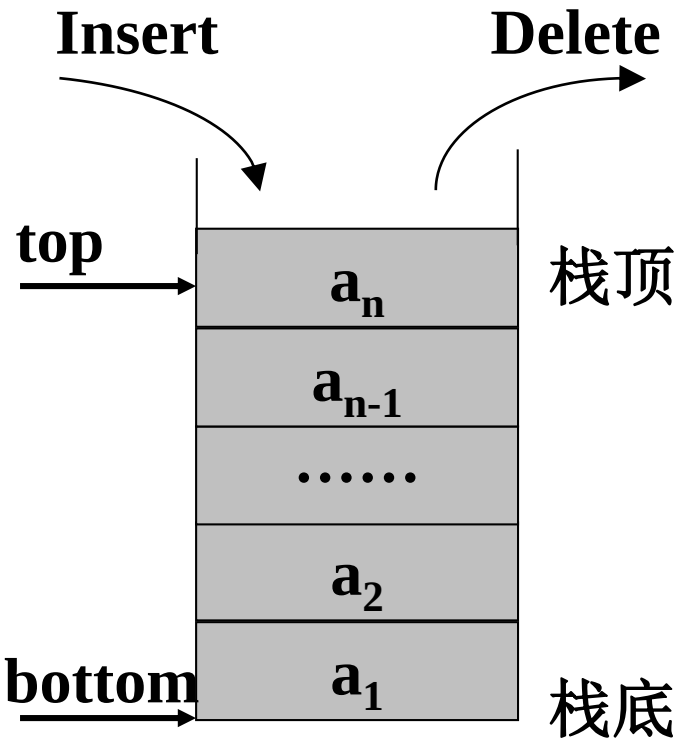
2.3 栈（Stack）

栈是线性表的一种特殊形式，是一种限定性数据结构，也就是在对线性表的操作加以限制后，形成的一种新的数据结构。

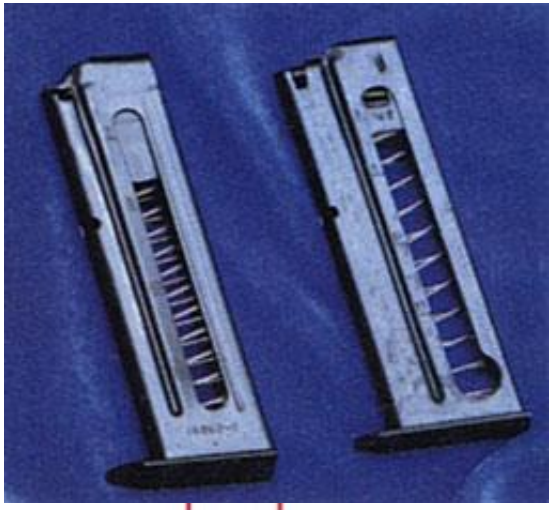
【定义】栈是限定只在表尾进行插入和删除操作的线性表。

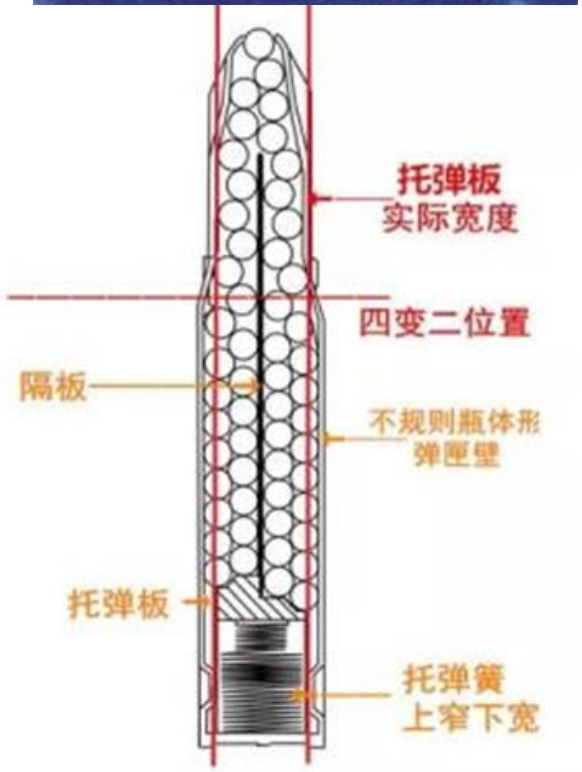
栈又称为后进先出(Last In First Out)的线性表。 简称LIFO结构。

栈举例.....



栈示意图



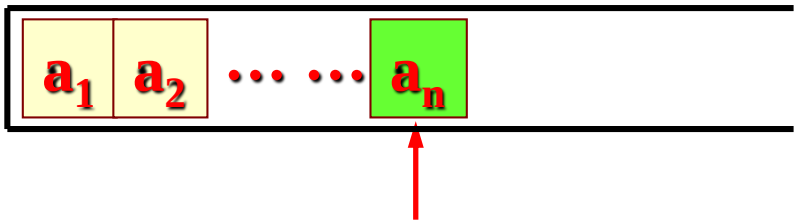

托弹板
实际宽度
四变二位置
隔板
不规则瓶体形
弹匣壁
托弹板
托弹簧
上窄下宽

栈的基本操作：

① MakeNull (S)
② Top (S)
③ Pop (S)
④ Push(S, e)
⑤ Empty (S)

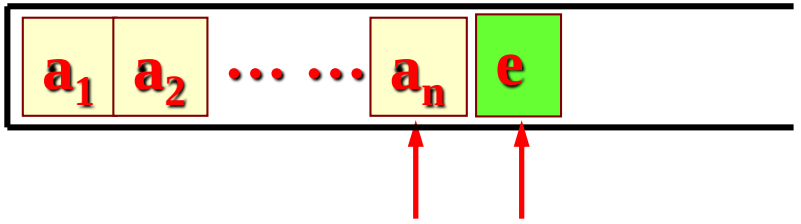
Top(S, &e)

初始条件：栈 S 已存在且非空。
操作结果：用 e 返回 S 的栈顶元素。



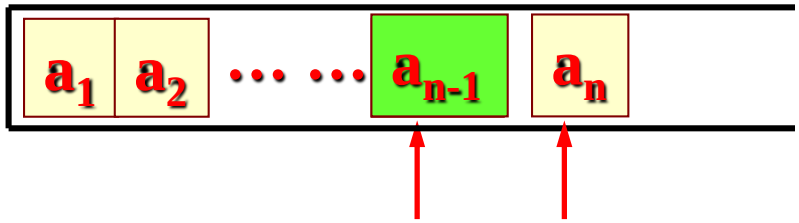
Push(&S, e)

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。



Pop(&S, &e)

初始条件：栈 S 已存在且非空。
操作结果：删除 S 的栈顶元素，并用 e 返回其值。



2.3.1　栈的实现

1、顺序存储

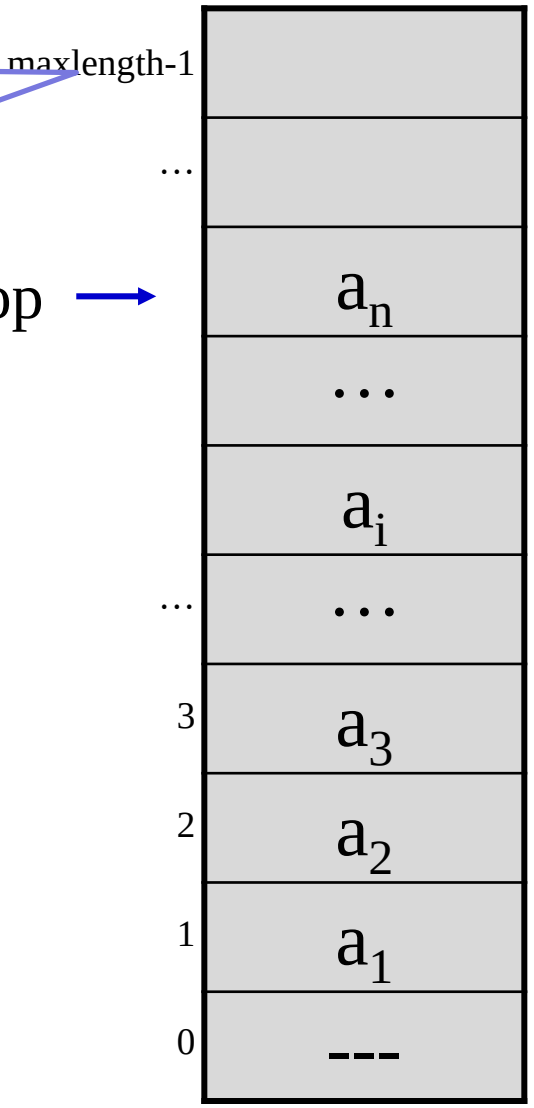
注意约定
栈顶指针
位置

结构类型：

```
typedef   struct {
    ElementType  data[maxlength];
    int  top ;
}  STACK ;

STACK  S ;
```

栈的容量：maxlength – 1 ;
栈空：S.top = 0 ;
栈满：S.top = maxlength – 1 ;
栈顶元素：S.data[S.top] ;



顺序栈示意图

栈的ADT基本操作：

① MakeNull (S)
② Top (S)
③ Pop (S)
④ Push(x , S)
⑤ Empty (S)

```
③ ElementType   Pop( STACK   &S )
  {
     if ( Empty (S ) )
        error ( “栈空” ) ;
     else
        x = S.data[ S.top – –] ;
  } ;
```

```
④ Void  Push( ElementType   x, STACK   &S )
  {
     if  ( S.top == maxlength - 1 )
        error ( “栈满” ) ;
     else
           S.data[++ S.top] = x ;
   } ;
```

问：为什么要设计栈？它有什么独特用途？

1. 调用函数或子程序非它莫属； 2. 递归运算的有力工具；
3. 用于保护现场和恢复现场； 4. 简化了程序设计的问题。

问：栈是什么？ 它与一般线性表有什么不同？

栈是一种特殊的线性表,它只能在表的一端(即栈顶)进行插入和删除运算。
与一般线性表的区别：仅在于运算规则不同。

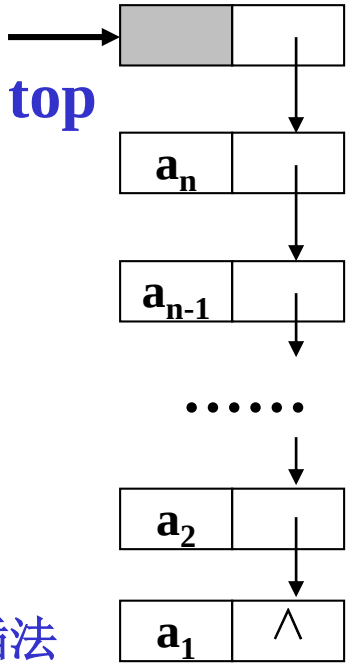
例 对于一个栈,给出输入项A、B、C,如果输入项序列由ABC组成,试给出所有可能的输出序列。

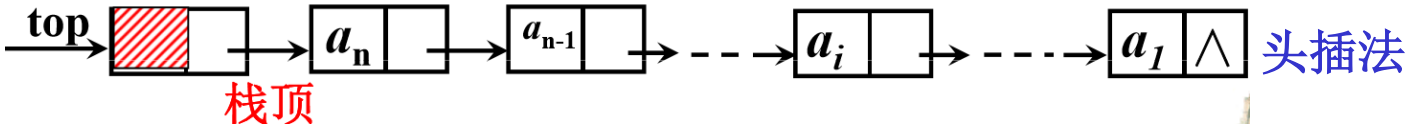
A进 A出 B进 B出 C进 C出	ABC
A进 A出 B进 C进 C出 B出	ACB
A进 B进 B出 A出 C进 C出	BAC
A进 B进 B出 C进 C出 A出	BCA
A进 B进 C进 C出 B出 A出	CBA
不可能产生输出序列CAB	

2、链式存储

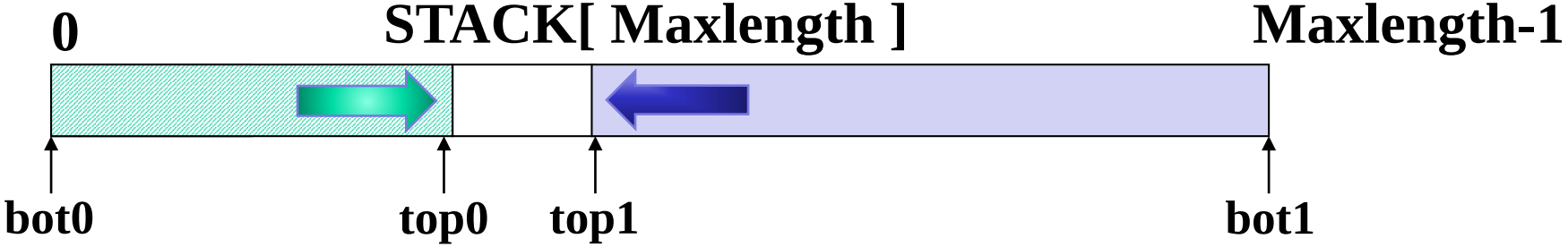
采用由指针形成的线性链表来实现栈的存储,要考虑链表的哪一端实现元素的插入和删除比较方便。

实现的方式如右图所示，其操作与线性链表的表头插入和删除元素相同。





3、共享栈



【例】用 S 表示进栈操作，用 X 表示出栈操作，若元素的进栈顺序是 1234，为了得到1342的出栈顺序，相应的 S 和 X 的操作序列为（ ）

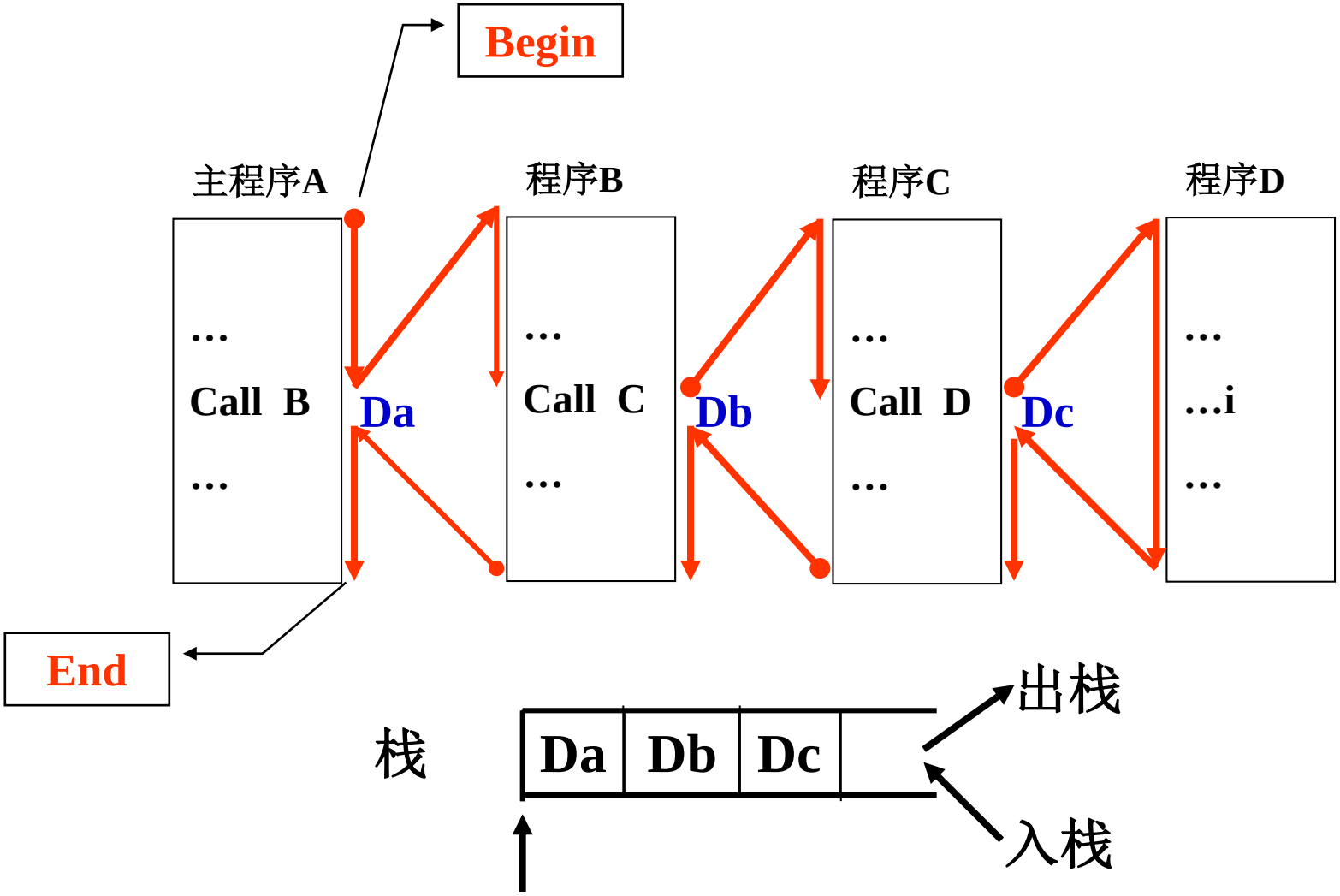
A. SXSXSSXX　　B. SSSXXSXX
C. SXSSXXSX　　D. SXSSXSXX

【例】若元素a, b, c, d, e, f, g依次进栈，允许进栈、退栈操作交替进行，但不允许连续3次进行退栈操作，不可能得到的出栈序列是（ ）

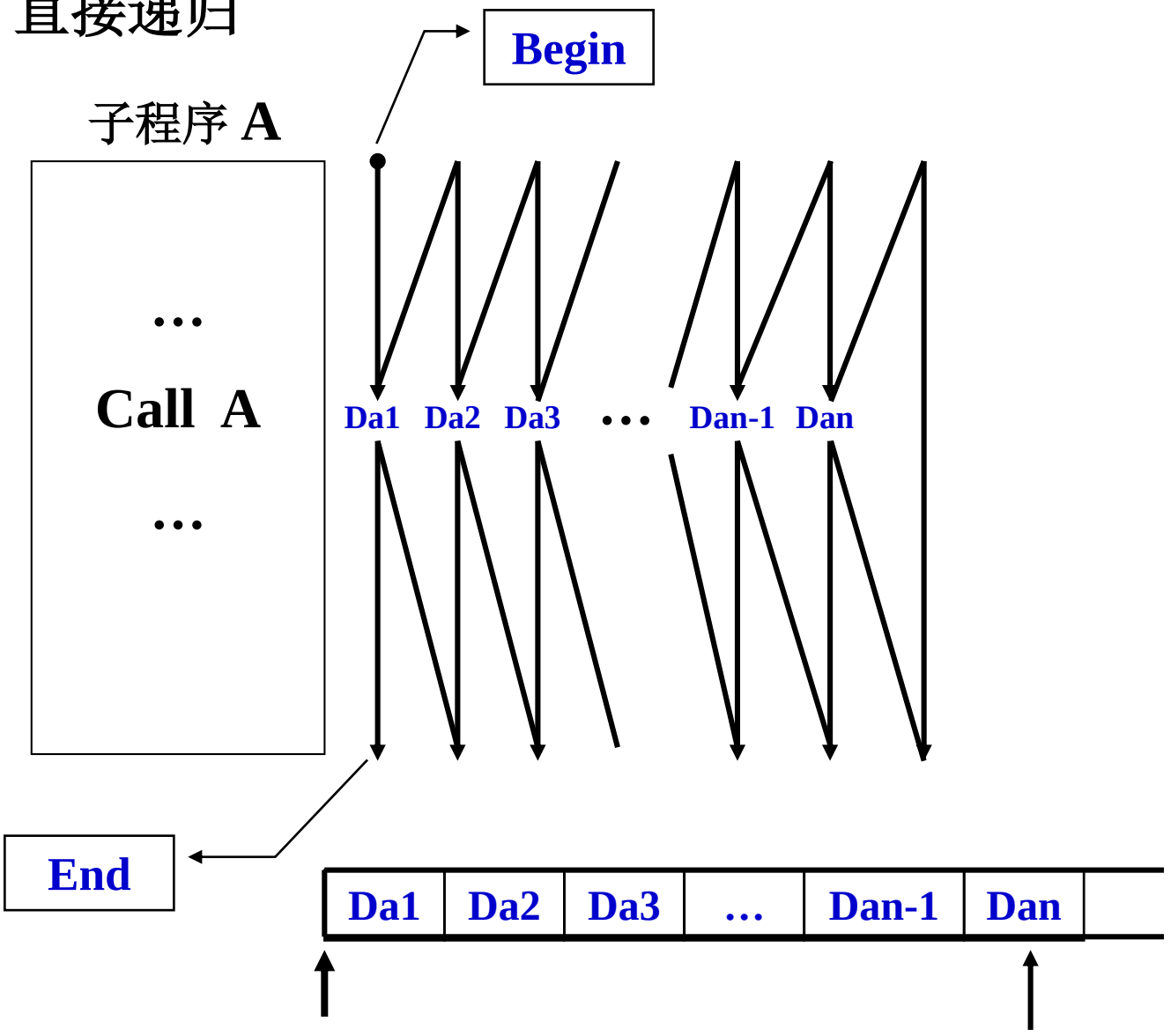
A. dcebfa　　B. cbdaef　　C. bcaefd　　D. afedcb

2.3.2 栈的应用

1、栈和递归过程（程序嵌套调用）



【例2-9】直接递归



判断括号匹配算法：

```
Boolean   Correct(char ext[],int)   //数组ext用于存储表达式
{
    STACK S;
    int j=0;
    MakeNull(S);
    while(ext[j]!='\0')
    {   if(exp[j]=='('
            Push(ext[j],S);
        if(ext[j]==')')
            x=Top(S);
        if(x==')')   Pop(S);
        else
            return FALSE;
        j++;
     }
    if(!Empty(S)) return FALSE;
    else  return TRUE;
}
```

【例2-10】括号匹配问题，对表达式从左到右扫描，遇到左括号进栈，当遇到右括号时退栈，比较是否匹配．当表达式扫描完毕，若栈为空，说明嵌套正确，否则括号匹配错误。

[([ ] [ ])]

问题：
是否有其它方法？

【例2-11】判断字符串是否中心对称。

例如：

xyzzyx

xyzyx

均属中心对称

判断字符串是否中心对称算法：

```
Boolean  Iudge(LIST  *head)
{    STACK S;
     char e;
     MakeNull(S);
     LIST *p=head;
     while(p)  {
          Push(p->data,S);
          p=p->next;
      }
      p=head;
      while(p)  {
          e=Top(S);Pop(S);
          if(p->data==e)    p=p->next;
          else break;
      }
     if(!p)   return TRUE;
     else     return FALSE;
}
```

【例2-12】求阶乘的函数

$$n!=\begin{cases}1 & \text{当}n=0\text{或}n=1\\ n*(n-1)! & \text{当}n>1\end{cases}$$

(1) 非递归算法

```
long fact( int n ）
{
    long f=2;    int i;
    for( i=2; i <= n; i++ )
          f = f*i;
    return(f);
}
```

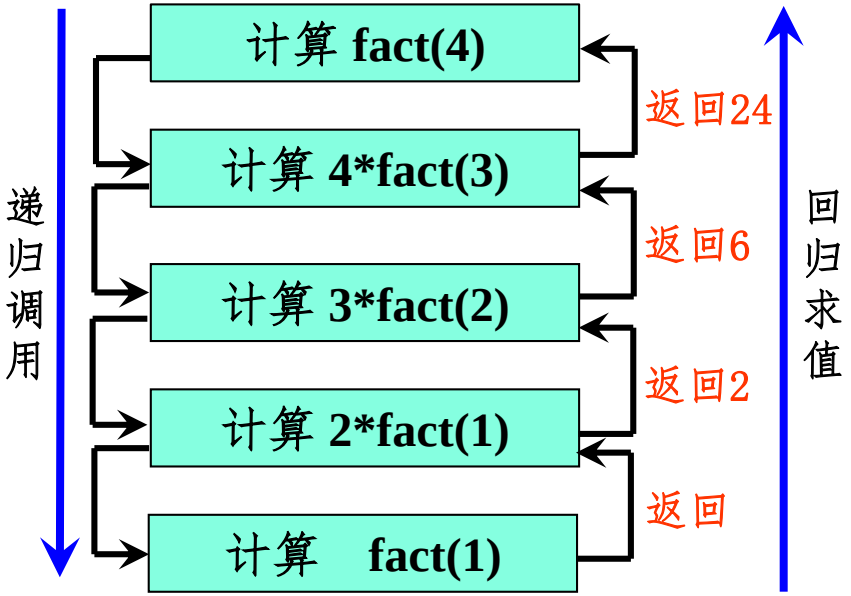
T(n)=o(n)

(2) 递归算法

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

T(n)=o(n) ?

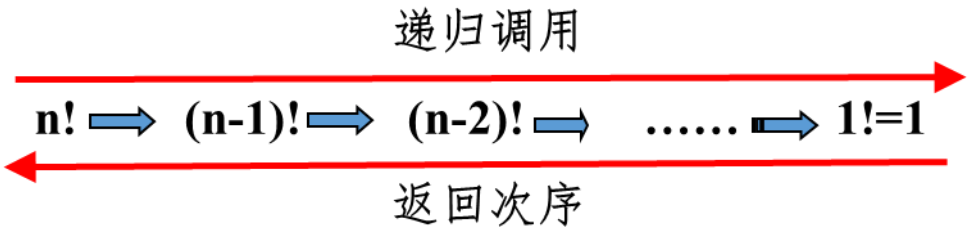
分析递归求解 n! 的过程



递归过程与递归工作栈

①递归过程在实现时，需要自己直接或间接调用自己；

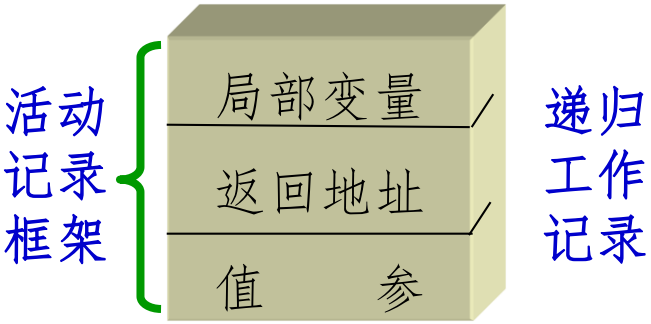
②层层向下递归，返回次序正好相反：



递归过程与递归工作记录

①每一次递归调用时，需要为过程中使用的参数、局部变量和返回地址等另外分配存储空间；

②每层递归调用需分配的空间形成递归工作记录，按栈结构组织，即LIFO。



递归函数的内部执行过程

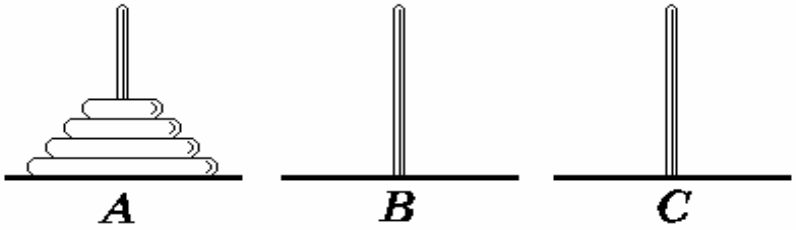
①建栈：运行开始时，首先为递归调用建立一个工作栈，其结构包括值参、局部变量和返回地址；

②压栈：每次执行递归调用之前，把递归函数的值参和局部变量的当前值以及调用后的返回地址压栈；

③出栈：每次递归调用结束后，将栈顶元素出栈，使相应的值参和局部变量恢复为调用前的值，然后转向返回地址指定的位置继续执行。

【例2-13】汉诺塔问题：递归的经典问题

在世界刚被创建的时候有一座钻石宝塔（塔A），其上有64个金碟。所有碟子按从大到小的次序从塔底堆放至塔顶。紧挨着这座塔有另外两个钻石宝塔（塔B和塔C）。从世界创始之日起，婆罗门的牧师们就一直在试图把塔A上的碟子移动到塔C上去，其间借助于塔B的帮助。每次只能移动一个碟子，任何时候都不能把一个碟子放在比它小的碟子上面。当牧师们完成任务时，世界末日也就到了。



汉诺塔问题的递归求解:

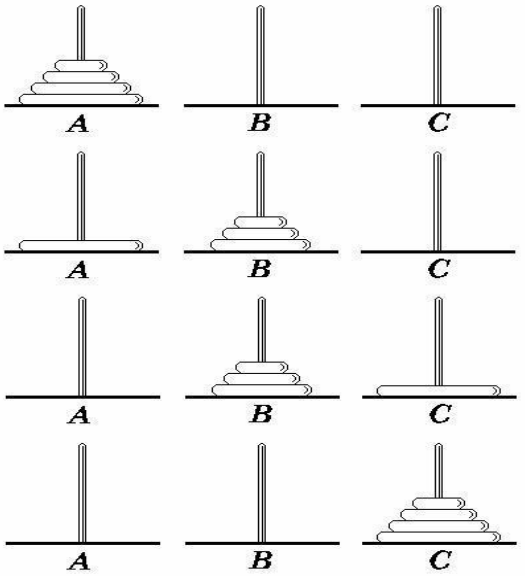
如果 n = 1，则将这一个盘子直接从 塔A移到塔 C 上，
否则，执行以下三步：

① 将塔A上的n-1个碟子借助塔C先移到塔B上；
② 把塔A上剩下的一个碟子移到塔C上；
③ 将n-1个碟子从塔B借助于塔A移到塔C上。

设：

函数：Hanoi(int n, char A, char B, char C);
功能：将n个盘子从塔A移到塔C,塔B为临时；

函数：Move(A, C);
功能：将塔A上的1个盘子直接移到塔C；

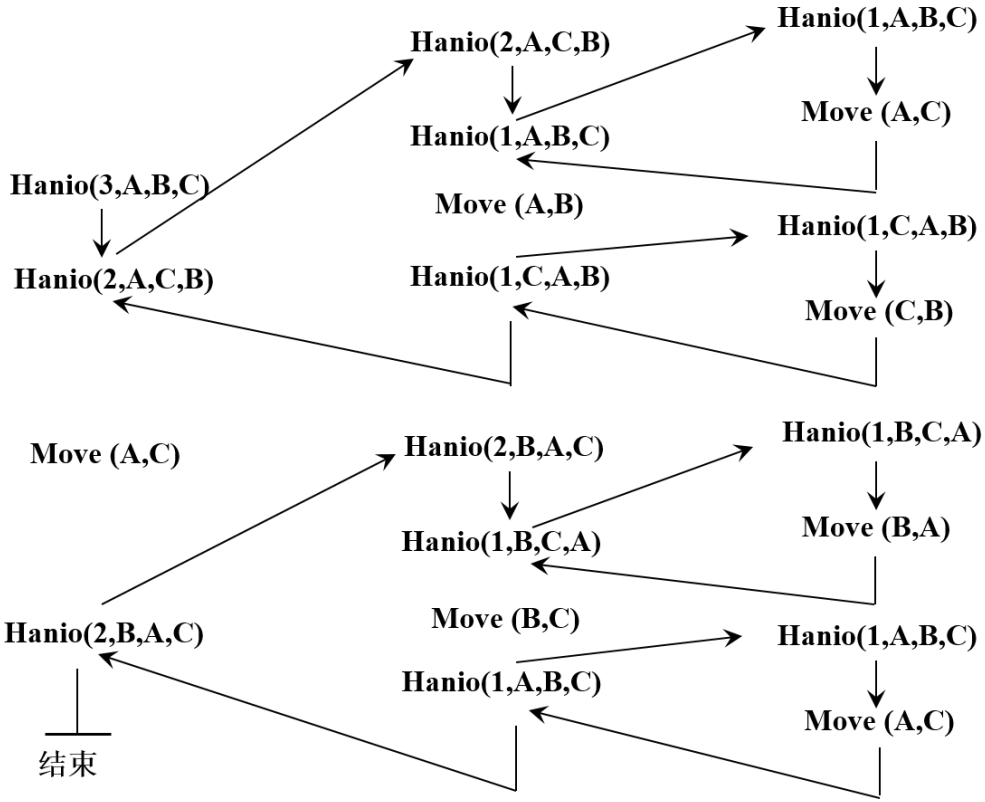


汉诺塔问题的递归求解:
如 n = 1,则将这1个盘子直接从塔A移到塔 C 上。
Move(A, C);
否则，执行以下三步：
(1)将n-1个盘子从塔A借助于塔C移到塔B上；
Hanoi(n-1, A, C, B);
(2)把塔A上剩下的一个碟子移到塔C上；
Move(A, C);
(3)将n-1个碟子从塔B借助于塔A移到塔C上。
Hanoi(n-1, B, A, C);

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

递归函数的运行轨迹

①写出函数当前调用层执行的各语句，并用有向弧表示语句的执行次序；
②对函数的每个递归调用，写出对应的函数调用，从调用处画一条有向弧指向被调用函数入口，表示调用路线，从被调用函数末尾处画一条有向弧指向调用语句的下面，表示返回路线；
③在返回路线上标出本层调用所得的函数值。



4个呢？

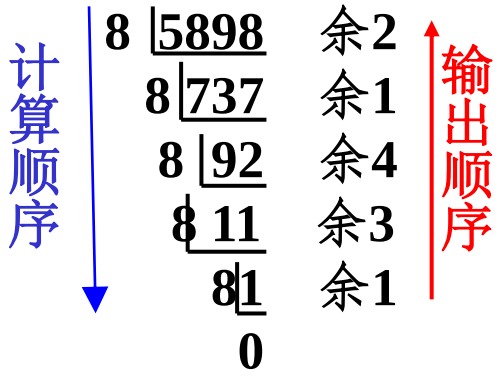
【例2-14】数制转换

计算机实现计算的基本问题。

方法：除留余数法。

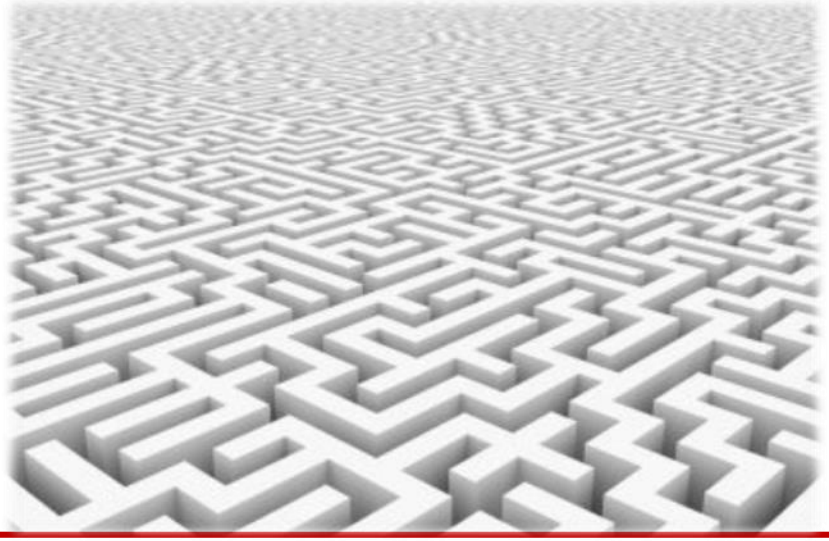
对输入的任意非负十进制整数, 打印输出与其等值的八进制数。

$(5898)_{10} = (13412)_8$




```
void main()
{
    STACK
s=NewStack();
    cin>>n;
   while(n){
        Push(n%8,s);
         n/=8;
    }
   while(! Empty(s)) {
        cout<<Top(s);
        Pop(s) ;
   }
}//时间复杂度?
```

2、迷宫求解

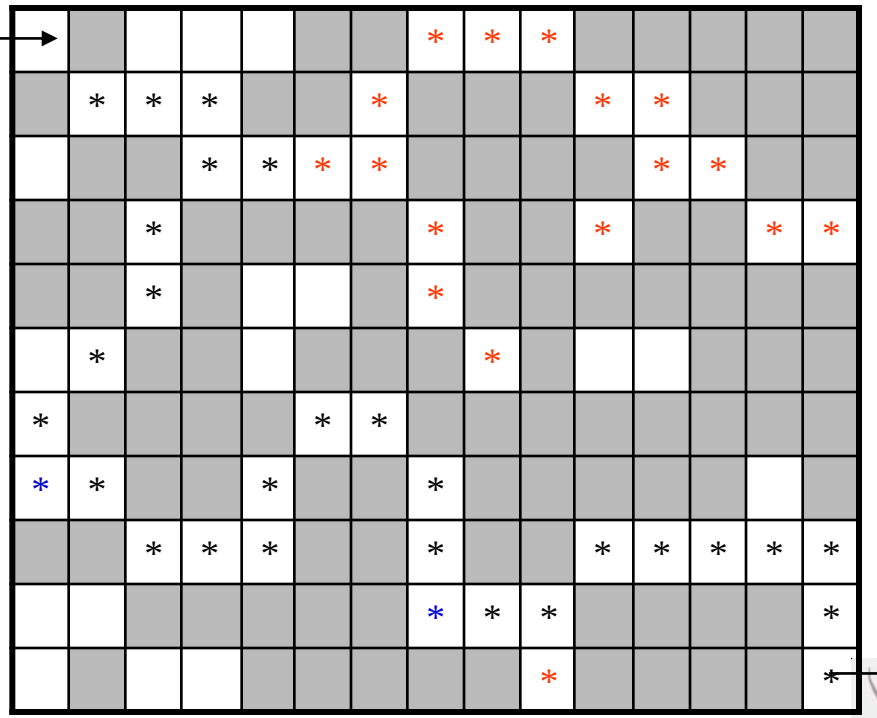


问题：

$$\begin{pmatrix}
0&1&0&0&0&1&1&0&0&0&1&1&1&1&1\\
1&0&0&0&1&1&0&1&1&1&0&0&1&1&1\\
0&1&1&0&0&0&0&1&1&1&1&0&0&1&1\\
1&1&0&1&1&1&1&0&1&1&0&1&1&0&0\\
1&1&0&1&0&0&1&0&1&1&1&1&1&1&1\\
0&0&1&1&0&1&1&1&0&1&0&0&1&1&1\\
0&1&1&1&1&0&0&1&1&1&1&1&1&1&1\\
0&0&1&1&0&1&1&0&1&1&1&1&1&0&1\\
1&1&0&0&0&1&1&0&1&1&0&0&0&0&0\\
0&0&1&1&1&1&1&0&0&0&1&1&1&1&0\\
0&1&0&0&1&1&1&1&1&0&1&1&1&1&0
\end{pmatrix}$$

11×15→m ×n

入口



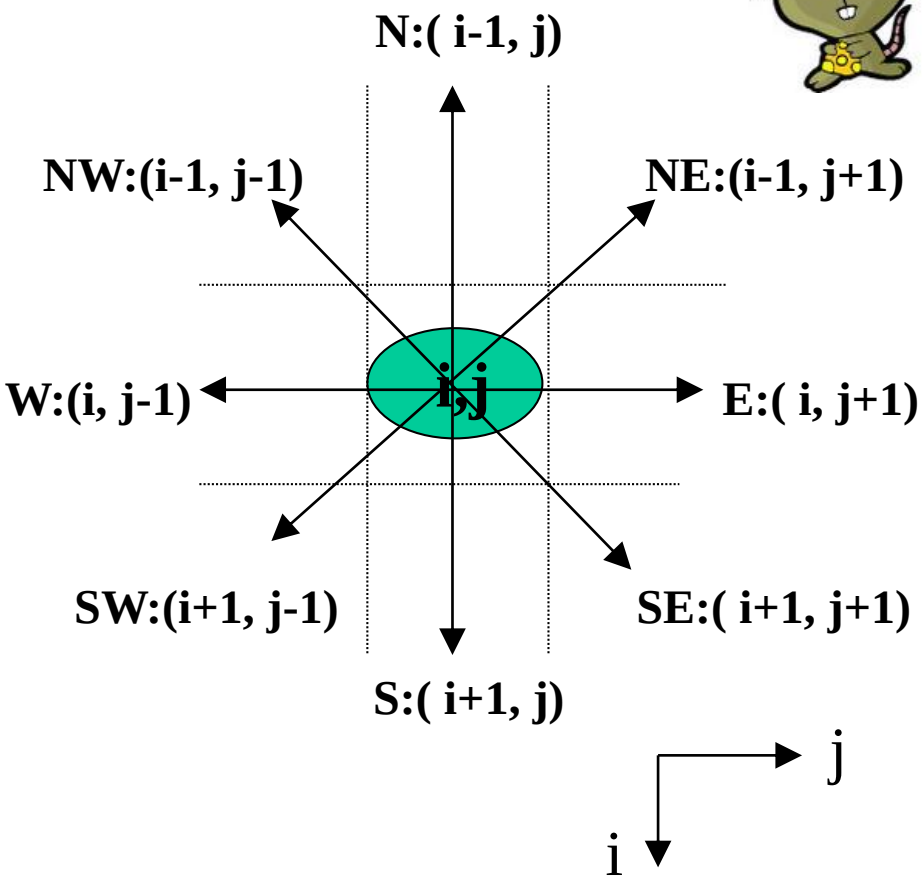
出口

迷宫示例

一个迷宫可用上图所示方阵[m,n]表示，0表示能通过，1 表示不能通过。现假设耗子从左上角[1,1]进入迷宫，编写算法，寻求一条从右下角 [m,n] 出去的路径。

分析：

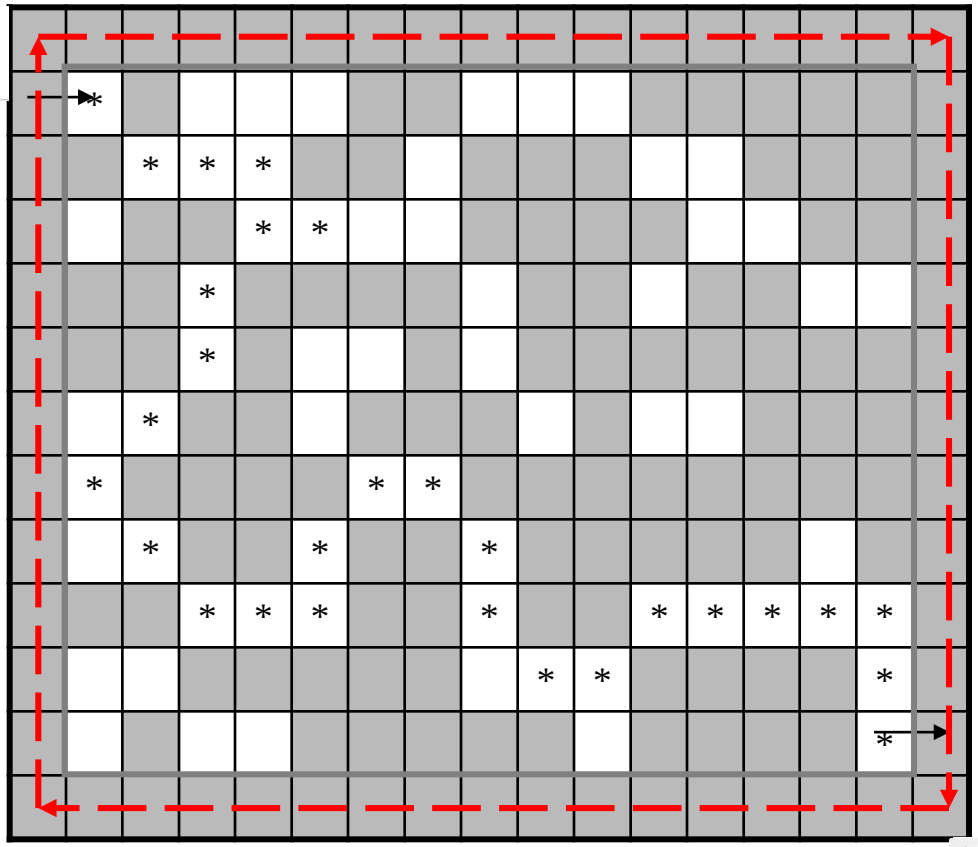
(1) 迷宫可用二维数组 **Maze[i][j] (1≤i≤m, 1≤j≤n)** 表示，入口**maze[1][1] = 0**；耗子在任意位置可用（i, j）坐标表示；



(2) 位置 (i, j) 周围有8个方向可以走通，分别记为：E,SE,S,SW,W,NW,N,NE;如图所示。方向 v 按从正东开始且顺时针分别记为1-8，v=1,8; 设二维数组 move 记下八个方位的增量；

V	i	j	说明
1	0	1	E
2	1	1	SE
3	1	0	S
4	1	-1	SW
5	0	-1	W
6	-1	-1	NW
7	-1	0	N
8	-1	1	NE

入口



出口

迷宫示例

从(i , j) 到 (g , h)
且 v = 2 (东南）则
有：　g = i + move[v][1] = i + 1 ;
　　h = j + move[v][2] = j + 1 ;

(3) 为避免时时监测边界状态，可把二维数组maze[1:m, 1:n]扩大为maze[0:m+1, 0:n+1]，且令0行、0列、m+1行、n+1列的值均为1；

(4) 采用试探的方法，当到达某个位置且周围八个方向走不通时需要回退到上一个位置，并换一个方向继续试探；为解决回退问题，需设一个栈，当到达一个新位置时将（i, j,v）进栈，回退时退栈。

(5) 每次换方向寻找新位置时，需测试该位置以前是否已经过，对已到达的位置，不能重复试探，为此设矩阵mark[][],其初值为0，一旦到达位置(i , j)时，置 mark[i][j] = 1;

文字描述算法：

(1) 耗子在(1, 1)进入迷宫，并向正东（v=1）方向试探。

(2) 检测下一方位(g, h)。若(g, h)=(m, n)且maze[m][n]=0，则耗子到达出口，输出走过的路径；程序结束。

(3) 若(g, h) ≠(m, n),但(g, h)方位能走通且第一次经过,则记下这一步，并从(g, h)出发，再向东试探下一步。否则仍在(i, j)方位换一个方向试探。

(4) 若(i, j)方位周围8个方位阻塞或已经过，则需退一步，并换一个方位试探。若(i, j)=(1, 1)则到达入口，说明迷宫走不通。

迷宫求解算法：

```
Void  GETMAZE ( maze , mark ,move ,S )
{  (i, j, v) = (1,1,1) ;    mark[1][1] = 1 ;   s.top = 0 ;
   do {  g = i+move[v][1] ;  h = j+move[v][2] ;
         if (( g == m) && ( h == n) && (maze[m][n] == 0 ))//出口
            { output( S ) ; return ; }
         if ((maze[g][h] ==0) && mark[g][h] == 0))//第一次走过
            { mark[g][h] = 1 ;  PUSH( i, j, v, S) ; (i, j, v) = (g, h, 1)  ;  }
          else if ( v < 8 )
                 v = v + 1 ;
           else { while (( s.v == 8) && (!Empty(S)))  Pop( S ) ;//连续退栈
                    if ( s.top > 0 )
                      ( i, j, v＋＋ ) = Top( S ); Pop(S);  } ;
      } while (( s.top ) && ( v != 8 )) ;
   cout << “路径不存在！ ”   ;
}
```

波兰逻辑学家J.Lukasiewicz于1929年提出的一种表达式。

3、表达式求值

表达式：
- 前缀表达式（逆波兰式）
- 中缀表达式
- 后缀表达式（波兰式）

如：

$(a+b)*(a-b)$
- $*+ab-ab$
- $(a+b)*(a-b)$
- $ab+ab-*$

高级语言中，采用类似自然语言的中缀表达式，但计算机对中缀表达式的处理是很困难的，而对后缀或前缀表达式则显得非常简单。

后缀表达式的特点：

① 在后缀表达式中，变量（操作数）出现的顺序与中缀表达式顺序相同。

② 后缀表达式中不需要括弧定义计算顺序，而由运算（操作符）的位置来确定运算顺序。

I. 将中缀表达式转换成后缀表达式

对中缀表达式从左至右依次扫描每一个字符，由于操作数的顺序保持不变，当遇到操作数时直接输出；为调整运算顺序，设立一个栈用以保存操作符，扫描到操作符时，将操作符压入栈中，进栈的原则是保持栈顶操作符的优先级要高于栈中其他操作符的优先级，否则，栈顶操作符依次退栈并输出，直到表达式结束。遇到“(”进栈，当遇到“)”时，退栈输出直到“)”为止。

II. 由后缀表达式计算表达式的值

对后缀表达式从左至右依次扫描，与I相反，遇到操作数时，将操作数进栈保留；当遇到操作符时，从栈中退出两个操作数并作相应运算，将计算结果进栈保留；直到表达式结束，栈中唯一元素即为表达式的值。

例 已知操作符包括+、-、*、/、（和）。将中缀表达式a+b-a*((c+d)/e-f)+g 转换成等价的后缀表达式ab+acd+e/f-*-g+时，用栈来存放暂时还不能确定运算次序的操作符。若栈初始为空，则转换过程中同时保存在栈中的操作符的最大个数是（）

A. 5　　B. 7　　C. 8　　D. 11