2.5 特殊线性表—字符串

2.5.1 串的逻辑结构

- 串：零个或多个字符组成的有限序列。
- 串长度：串中所包含的字符个数。
- 空串：长度为0的串，记为：“ ”。
- 非空串通常记为：　　*S*=" *s*1 *s*2 …… *sn* "
  - 其中：*S*是串名，双引号是定界符，双引号引起来的部分是串值，*si*（1≤*i*≤*n*）是一个任意字符。
  - 字符集：ASII码、扩展ASII码、Unicode字符集
- 子串：串中任意个连续的字符组成的子序列。
- 主串：包含子串的串。
- 子串的位置：子串的第一个字符在主串中的序号。

2.5.1 串的逻辑结构

- 串的操作
  - string MakeNull( ) ;
  - bool IsNull (S) ;
  - void In(S, a) ;
  - int Len(S) ;
  - void Concat(S1, S2) ;
  - string Substr(S, m, n) ;
  - int Index(S, S1) ;
- 与其他线性结构相比，串的操作对象有什么特点？
  - 串的操作通常以串的整体作为操作对象。

例一：将串T 插在串S 中第 i 个字符之后INSERT(S, T, i)。

```
Void  INSERT( STRING  &S, STRING  T, int  i )
{   STRING  t1,  t2 ;
    if  ( ( i < 0 ) || ( i > LEN( S ) )
        error ‘指定位置不对’  ;
    else
        if ( ISNULL( S ) )  S = T ;
        else
            if ( ISNULL ( T ) )
            {   t1 = SUBSTR( S, 1, i ) ;
                t2 = SUBSTR( S, i + 1, LEN( S ) );
                S = CONCAT( t1, CONCAT( T, t2 ) );
            }
}
```

例二：从串 S 中将子串 T 删除DELETE(S, T)。

```
Void  DELETE( STRING  &S, STRING  T )
{   STRING  t1,  t2 ;
    int  m, n ;
    m = INDEX( S, T );
    if  ( m==0 )
            error ‘串S中不包含子串T’ ;
    else
        {   n = LEN( T ) ;
            t1 = SUBSTR( S, 1, m - 1 ) ;
            t2 = SUBSTR( S, m + n, LEN( S ) );
            S = CONCAT( t1, t2);
        }
}
```

2.5.2 串的存储结构

- **顺序串：**
  - 用数组来存储串中的字符序列。
- **非压缩形式**
  -

……	*C*	*h*	*i*	*n*	*s*	*e*	……

- **压缩形式**
  -

……		*C* *h* *i* *n*	*e* *s* *e*			……

2.5.2 串的存储结构(Cont.)

- 如何表示串的长度?
  - 方法一：用一个变量来表示串的实际长度，同一般线性表
  - 方法二：在串尾存储一个不会在串中出现的特殊字符作为串的终结符，表示串的结尾。

0	1	2	3	4	5	6	…… …… Max−1
C	*h*	*i*	*n*	*e*	*s*	*e*	空 闲

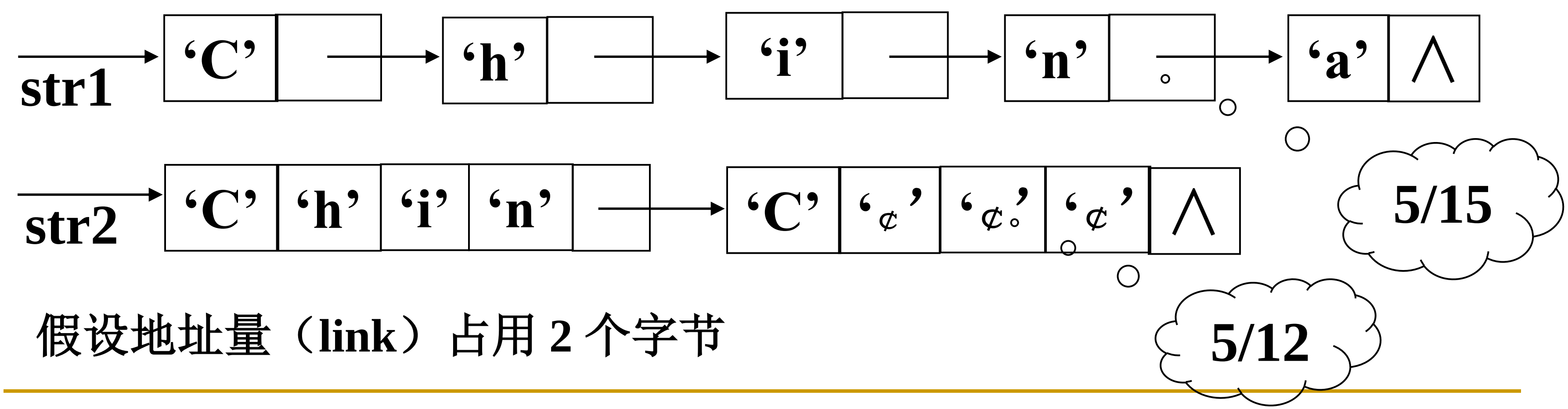
7

0	1	2	3	4	5	6	7	…… …… Max−1
C	*h*	*i*	*n*	*e*	*s*	*e*	\0	空 闲

•2.5.2 链接串

```
struct  node  {
      char  data ;
      node  *link ;
} ;
typedef  node  *STRING1;
STRING1  str1 ;
```

```
struct  node  {
      char  data[4] ;
      node  *link ;
} ;
typedef  node  *STRING2 ;
STRING2  str2 ;
```



假设地址量（link）占用 2 个字节

2.5.3 模式匹配

- 模式匹配 (字符串匹配是计算机的基本任务之一)
  - 给定*S*="$S_0\ S_1 \ldots\ S_{n-1}$"(主串)和*T*="$T_0\ T_1 \ldots T_{m-1}$"(模式)，在*S*中寻找*T*的过程称为模式匹配。如果匹配成功，返回*T*在*S*中的位置；如果匹配失败，返回-1。
- 假设串采用顺序存储结构
- 朴素模式匹配算法(Brute-Force算法)：枚举法(回溯)
  - 基本思想
    - 从主串*S*的第一个字符开始和模式*T*的第一个字符进行比较，若相等，则继续比较两者的后续字符；否则，从主串*S*的第二个字符开始和模式*T*的第一个字符进行比较。
    - 重复上述过程，直到*T*中的字符全部比较完毕，说明本趟匹配成功；或*S*中字符全部比较完，则说明匹配失败。

2.5.3 模式匹配(Cont.)

设主串S=“ababcabcacbab”，模式串T=“abcac”

第1趟匹配	主串	ababcabcacbab	i=2	
	模式串	abc	j=2	匹配失败
第2趟匹配	主串	ababcabcacbab	i=1	
	模式串	abc	j=0	匹配失败
第3趟匹配	主串	ababcabcacbab	i=6	
	模式串	abcac	j=4	匹配失败
第4趟匹配	主串	ababcabcacbab	i=3	
	模式串	abc	j=0	匹配失败
第5趟匹配	主串	ababcabcacbab	i=4	
	模式串	abc	j=0	匹配失败
第6趟匹配	主串	ababcabcacbab	i=9	//返回i-lenT+1
	模式串	abcac	j=4	匹配成功

特点:主串指针需回溯（i=i-j+1），模式串指针需复位（j=0）。

2.5.3 模式匹配(Cont.)

- **BF算法实现的详细步骤：**
  - **1. 在串S和串T中设比较的起始下标i和j；**
  - **2. 循环直到S或T的所有字符均比较完；**
    - **2.1 如果S[i]=T[j]，继续比较S和T的下一个字符；**
    - **2.2 否则，将i回溯(i=i-j+1)，j复位，准备下一趟比较；**
  - **3. 如果T中所有字符均比较完，则匹配成功，返回主串起始比较下标；否则，匹配失败，返回-1。**

2.5.3 模式匹配(Cont.)

```
int StrMatch_BF ( char* S, char* T, int pos=0)
{ /*S为主串T为模式，长度分别为了lenS和lenT；串采用顺序存储结构*/
    i = pos;    j = 0;                        // 从第一个位置开始比较
    while (i<=lenS && j<=lenT) {
        if (S[i] == T[j]) {++i; ++j;}   // 继续比较后继字符
        else {i = i – j + 1;    j = 0; }   // 指针后退重新开始匹配
    }
    // 返回与模式第一字符相等的字符在主串中的序号
    if ( j > lenT)
        return i- lenT+1;
    else
        return -1;                 // 匹配不成功
}
```

2.5.3 模式匹配(Cont.)

Brute-Force算法的时间复杂度

主串S长n,模式串T长m。可能匹配成功的(主串)位置(0~n-m).

①最好的情况下，模式串的第0个字符失配

设匹配成功在S的第i个字符，则在前i趟匹配中共比较了i次，第i趟成功匹配共比较了m次，总共比较了（i+m）次。所有匹配成功的可能共有n-m+1种，所以在等概率情况下的平均比较次数：

$$\sum_{i=0}^{n-m} p_i(i+m)=\frac{1}{n-m+1}\sum_{i=0}^{n-m}(i+m)=\frac{1}{2}(n+m)$$

最好情况下算法的平均时间复杂度O(n+m)。

2.5.3 模式匹配(Cont.)

Brute-Force算法的时间复杂度

主串S长n,模式串T长m。可能匹配成功的(主串)位置(0~n-m).

②最坏的情况下，模式串的最后1个字符失配

- 简单的匹配算法：一旦某个字符匹配失败，从头开始。
- （本次匹配起点后一个字符开始）
- 主串　S=000000000000000000000000000000001
- 52个零

•设匹配成功在S的第i 个字符

- 模式串 T=00000001，指针要回朔45次。

$$\sum_{i=0}^{n-m} p_i(i+1)m = \frac{m}{n-m+1}\sum_{i=0}^{n-m}(i+1) = \frac{1}{2}m(n-m+2)$$

2.5.3 模式匹配(Cont.)

Brute-Force算法的时间复杂度

主串S长n,模式串T长m。可能匹配成功的(主串)位置(0~n-m).

②最坏的情况下，模式串的最后1个字符失配

设匹配成功在S的第i 个字符，则在前i趟匹配中共比较了i*m次，第i趟成功匹配共比较了m次，总共比较了(i+1)*m次。所有匹配成功的可能共有n-m+1种，所以在等概率情况下的平均比较次数：

$$\sum_{i=0}^{n-m} p_i(i+1)m = \frac{m}{n-m+1}\sum_{i=0}^{n-m}(i+1) = \frac{1}{2}m(n-m+2)$$

最坏情况下的平均时间复杂度为O(n*m)。

2.5.3 模式匹配(Cont.)

- KMP算法----改进的模式匹配算法
  - 为什么BF算法时间性能低？
    - 在每趟匹配不成功时存在大量回溯，没有利用已经部分匹配的结果。

第1趟匹配

↓ i=2

a b a b c a b c a c b a b

a b c

↑j=2

第2趟匹配

↓ i=2----6

a b a b c a b c a c b a b

a b c a c

↑ j=0

2.5.3 模式匹配(Cont.)

- KMP算法----改进的模式匹配算法
  - 为什么BF算法时间性能低？
    - 在每趟匹配不成功时存在大量回溯，没有利用已经部分匹配的结果。
  - 如何在匹配不成功时主串不回溯？
    - 主串不回溯，模式就需要向右滑动一段距离。
  - 如何确定模式的滑动距离？
    - 利用已经得到的“部分匹配”的结果
    - 将模式向右“滑动”尽可能远的一段距离后，继续进行比较

2.5.3 模式匹配(Cont.)

- 假设主串ababcabcacbab，模式abcac，KMP算法的匹配过程示例：
- 第1趟匹配

```
    ↓ i=2
a b a b c a b c a c b a b
a b c
    ↑j=2
```

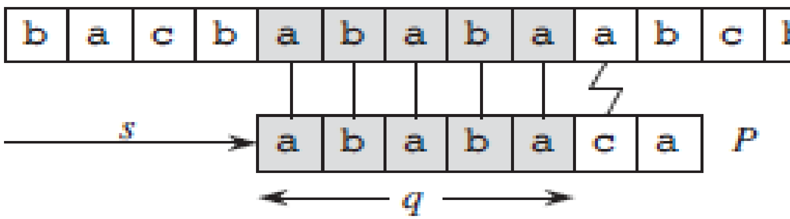
第2趟匹配

```
    ↓ i=2----6
a b a b c a b c a c b a b
    a b c a c
    ↑ j=0
```

第3趟匹配

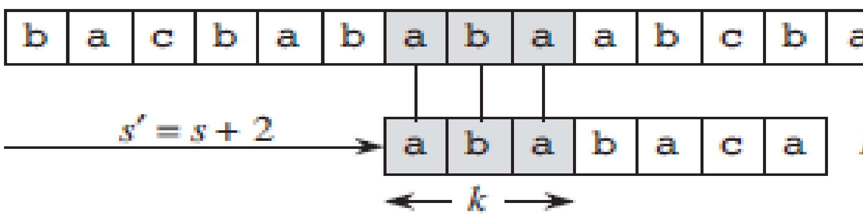
```
            ↓ i=6
a b a b c a b c a c b a b
          a b c a c
            ↑j=1
```

KMP (Knuth-Morris-Pratt)算法（图解）



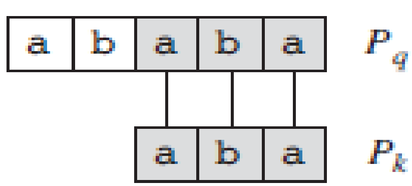
(a)

模式串的部分匹配为下次匹配位置提供信息



(b)

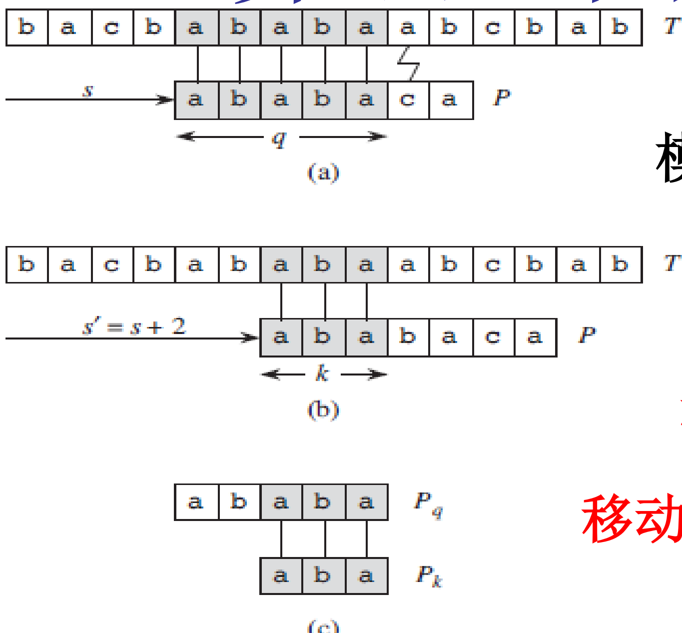
模式应该向右滑多远才是最高效率的?



(c)

移动的位数与模式串某位的自身最大前缀有关

KMP算法（图解）



模式串的部分匹配为下次匹配位置提供信息

模式应该向右滑多远才是最高效率的?

移动的位数与模式串某位的自身最大前缀有关

- （a图）：**前5个字符已经匹配成功**，**naive**算法接着**移到 s + 1**。但是明显的 s + 1 处是明显无效的。
- （b）图：**s + 2前三个字符都可以匹配**，所以**很可能是匹配点**。
- 数组**π** 记录的就是这些信息，比如对于P，上边的例子**π [5] = 3**，则下一个可能的位移是s′= s + （q – π [q]），即s′= s + 2。也就是在匹配过程中，**用π 数组记录下一次可能匹配位置的信息**。

KMP算法（前缀函数）

[前缀函数]：给定模式串P[1..m]，P的前缀函数

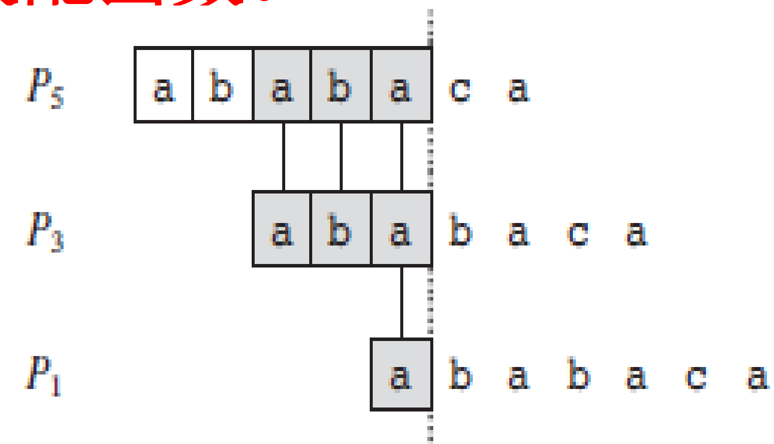
π：{1, 2, …, m}→{0, 1, …, m−1}，满足

π[q]=max{k：k<q，并且P_k是P_q的后缀}

也有称q－π[q]为失配函数。

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

2.5.3 模式匹配(Cont.)

- 假设主串ababcabcacbab，模式abcac，KMP算法的匹配过程示例：
- 第1趟匹配

```
  ↓i=2
a b a b c a b c a c b a b
a b c
    ↑j=2
```

第2趟匹配

```
   ↓i=2----6
a b a b c a b c a c b a b
    a b c a c
    ↑ j=0
```

第3趟匹配

```
            ↓i=6
a b a b c a b c a c b a b
          a b c a c
            ↑j=1
```

KMP算法（匹配算法）

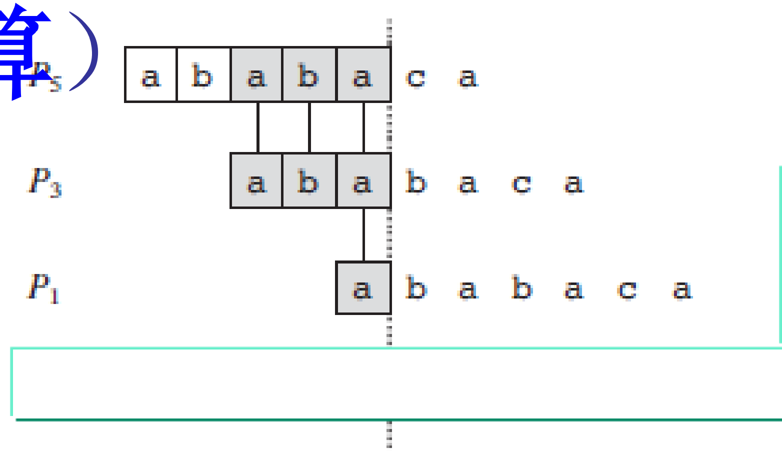
```
KMP-MATCHER(T, P)
n ← length[T]
m ← length[P]
π ← COMPUTE-PREFIX-FUNCTION(P)
q ← 0                               //Number of characters matched.
for i ← 1 to n                      //Scan the text from left to right.
      do while q > 0 and P[q + 1] ≠ T[i]
            do q ← π[q]   //end while //Next character does not match.
      if P[q + 1] = T[i]
          then q ← q + 1            //Next character matches.
      if q = m                      //Is all of P matched?
          then print "Pattern occurs with shift" i – m
                q ← π[q]            //Look for the next match.
```

KMP算法（前缀函数计算）

• $\pi[q]=\max\{k: k<q$，并且P_k是P_q的后缀$\}$

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

$$\pi(q)=\begin{cases}0, & q=\boxed{1}\\ \pi^m(q-1)+1, & \text{其中}m\text{是满足等式}P[\pi^l(q-1)+1]=P[q]\text{ 的最小整数}l\\ 0, & \text{没有满足上式的}l\end{cases}$$

$\Pi^1(q)=\pi(q)$，　　$\pi^l(q)=\pi(\pi^{l-1}(q))$

```
COMPUTE-PREFIX-FUNCTION(P)
m ← length[P]
π[1] ← 0
k ← 0
for q ← 2 to m
     do while k > 0 and P[k + 1] ≠ P[q]
            do k ← π[k]
        if P[k + 1] = P[q]
           then k ← k + 1
        π[q] ← k
return π
```

2.5.3 模式匹配(Cont.)

- 假设主串ababcabcacbab，模式abcac，KMP算法的匹配过程示例：
- 第1趟匹配

```
  ↓ i=2
a b a b c a b c a c b a b
a b c
  ↑j=2
```

第2趟匹配

```
    ↓ i=2----6
a b a b c a b c a c b a b
    a b c a c
    ↑ j=0
```

第3趟匹配

```
            ↓ i=6
a b a b c a b c a c b a b
          a b c a c
            ↑j=1
```

2.6 (多维)数组

- 数组:
  - 是由下标（index）和值（value）组成的序对（index，value）的序列。
  - 也可以定义为是由相同类型的数据元素组成有限序列。
  - 每个元素受$n(n\geq 1)$个线性关系的约束，每个元素在n个线性关系中的序号i_1、i_2、…、i_n称为该元素的下标，并称该数组为 n 维数组。
- 数组的特点:
  - 元素本身可以具有某种结构，属于同一数据类型;
  - 数组是一个具有固定格式和数量的数据集合。
- 示例:

2.6 (多维)数组(Cont.)

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \Rightarrow$$

$A=(A_1, A_2, \ldots\ldots, A_n)$

其中：

$A_i=(a_{1i}, a_{2i}, \ldots\ldots, a_{mi})$

$(1 \le i \le n)$

- 元素a_{22}受两个线性关系的约束，在行上有一个行前驱a_{21}和一个行后继a_{23}，在列上有一个列前驱a_{12}和和一个列后继a_{32}。
- 二维数组是数据元素为线性表的线性表。

2.6 (多维)数组(Cont.)

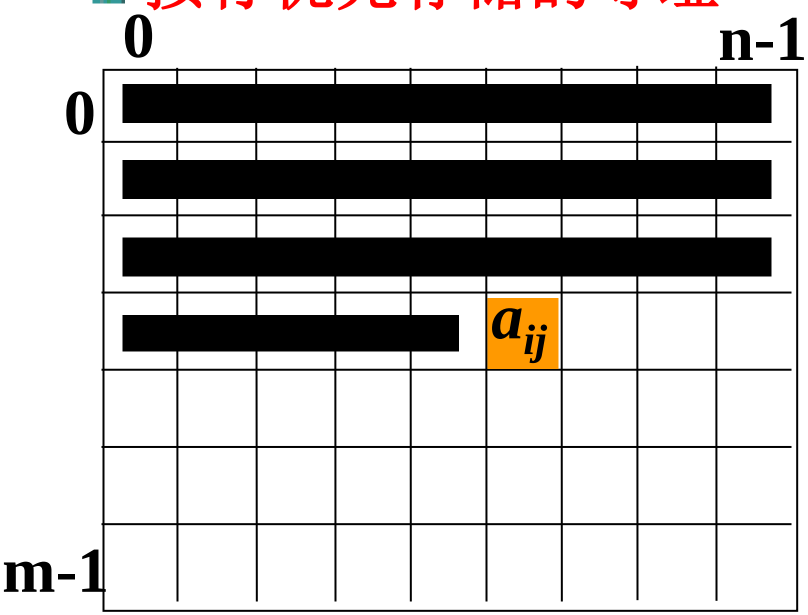
- 数组的基本操作
  - 初始化：Create（ ）
    - 建立一个空数组；
    - int A[ ][ ]
  - 存取： Retrieve（array，index）
    - 给定一组下标，读出对应的数组元素；
    - A[i][j]
  - 修改： Store（array，index，value）：
    - 给定一组下标，存储或修改与其相对应的数组元素。
    - A[i][j]=8
  - 无需插入和删除操作

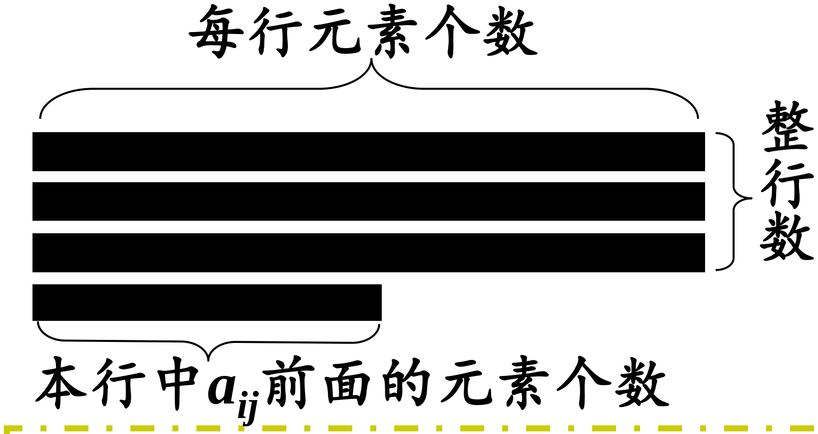
2.6 (多维)数组(Cont.)

- 数组的存储结构
  - 数组没有插入和删除操作，所以，不用预留空间，适合采用顺序存储。
- 数组的顺序存储
  - 用一组连续的存储单元来实现（多维）数组的存储。
  - 高维数组可以看成是由多个低维数组组成的。
- 二维数组的存储与寻址
  - 常用的映射方法有两种：
    - 按行优先：先行后列，先存储行号较小的元素，行号相同者先存储列号较小的元素。
    - 按列优先：先列后行，先存储列号较小的元素，列号相同者先存储行号较小的元素。

2.6 (多维)数组(Cont.)

- 按行优先存储的寻址----二维数组





a_{ij}前面的元素个数k
=整行数×每行元素个数+本行中a_{ij}前面的元素个数 $= i \times n + j$

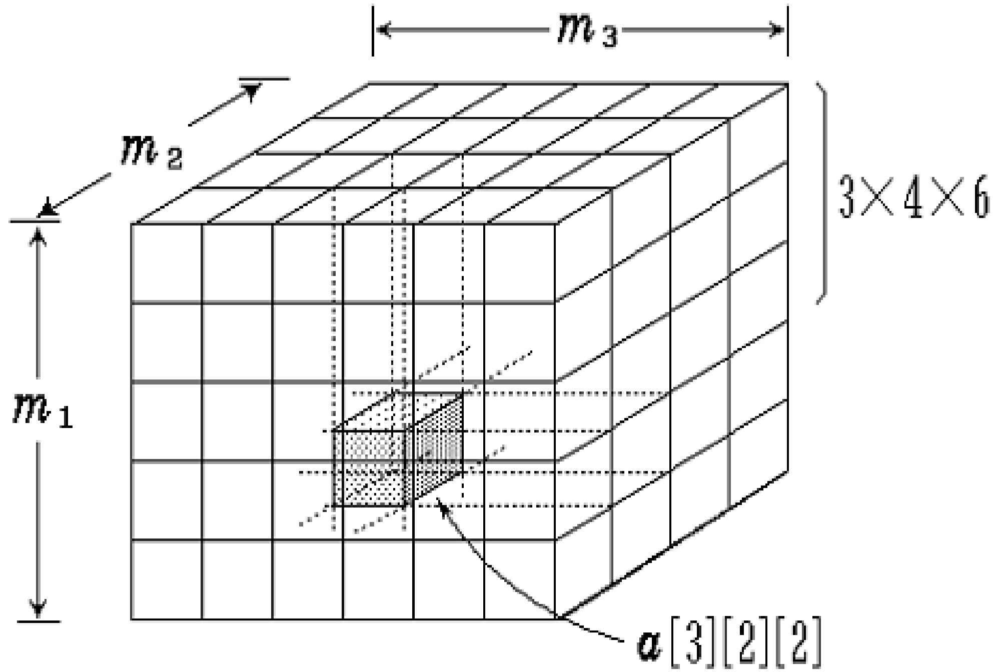
$$Loc(a_{ij}) = Loc(a_{00}) + (i \times n + j) \times c$$

Sa	a_{00}	a_{01}	a_{02}			a_{ij}			$a_{n-1,n-1}$
k=	0	1	2	…		$i \times n+j$			n^2-1

2.6 (多维)数组(Cont.)

- 按行优先存储的寻址----多维数组

n（*n*>2）维数组一般也采用按行优先和按列优先两种存储方法。



$$Loc(a_{ijk}) = Loc(a_{000}) + (i \times m_2 \times m_3 + j \times m_3 + k) \times c$$

更高维的数组呢？

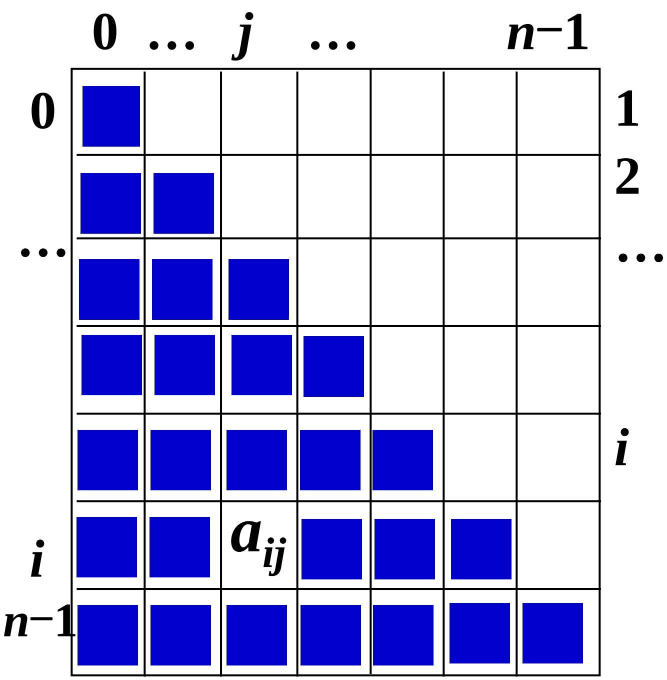
2.6 (多维)数组(Cont.)

- 特殊矩阵的压缩存储
  - 特殊矩阵：矩阵中很多值相同的元素并且它们的分布有一定的规律。
    - 如对称矩阵、上/下三角矩阵、带状(对角)矩阵等
  - 稀疏矩阵：矩阵中有很多特定值的（如零）元素。
    - 分布没有规律
    - 在m*n的矩阵中，有t个元素不为零。令α=t/m*n，称 α为矩阵的稀疏因子。
    - 通常认为α<=0.05时称为稀疏矩阵
- 压缩存储的基本思想是：
  - 为多个值相同的元素只分配一个存储空间；
  - 对特定值（如零）的元素不分配存储空间。

2.6 (多维)数组(Cont.)

- 三角矩阵的压缩存储----下\上三角矩阵
  - 只存储下三角部分的元素。
  - 对角线上方的常数不存或只存一个

$$A=\begin{pmatrix} 3 & c & c & c & c \\ 6 & 2 & c & c & c \\ 4 & 8 & 1 & c & c \\ 7 & 4 & 6 & 0 & c \\ 8 & 2 & 9 & 5 & 7 \end{pmatrix}$$



矩阵中任意一个元素a_{ij}在一维数组中的下标k与i、j的对应关系：

$$k= i\times(i+1)/2+ j \qquad (i \geq j)$$

$$k= n\times(n+1)/2 \qquad 存常数c$$

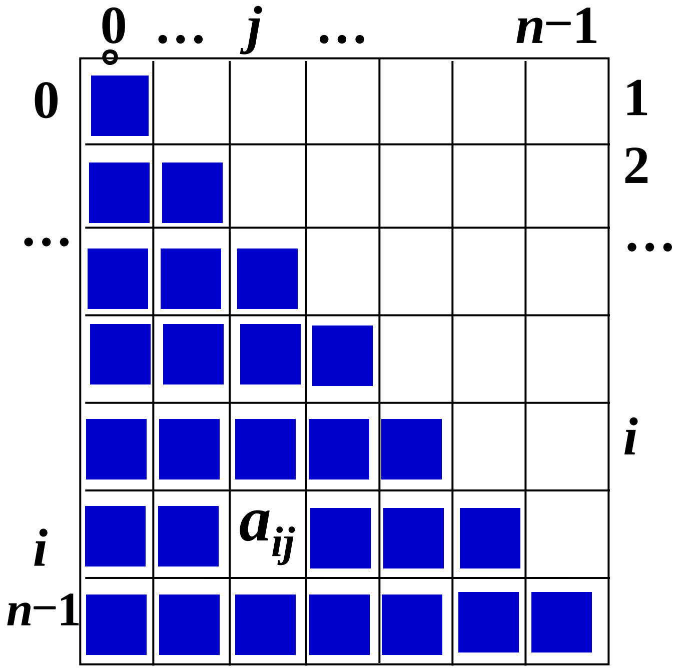
Sa	$a_{0\,0}$	$a_{1\,0}$	$a_{1\,1}$	$a_{2\,0}$	…	$a_{i,j}$	…	$a_{n-1,0}$	…	$a_{n-1,n-1}$	c
k=	0	1	2	3		i(i+1)/2+j		n(n-1)/2		n(n+1)/2-1	*

2.6 (多维)数组(Cont.)

对称矩阵的压缩存储

- 对称矩阵特点：$a_{ij}=a_{ji}$
- 只存储下/上三角部分的元素

$$A=\begin{pmatrix} 3 & 6 & 4 & 7 & 8 \\ 6 & 2 & 8 & 4 & 2 \\ 4 & 8 & 1 & 6 & 9 \\ 7 & 4 & 6 & 0 & 5 \\ 8 & 2 & 9 & 5 & 7 \end{pmatrix}$$



a_{ij}在一维数组中的序号

$= i\times(i+1)/2+j+1$

∵一维数组下标从0开始

∴a_{ij}在一维数组中的下标

$k= i\times(i+1)/2+j \qquad (i\geq j)$

$k= j\times(j+1)/2+i \qquad (i<j)$

Sa	$a_{0\,0}$	$a_{1\,0}$	$a_{1\,1}$	$a_{2\,0}$	…	$a_{i,j}$	…	$a_{n-1,0}$	…	$a_{n-1,n-1}$
k=	0	1	2	3		i(i+1)/2+j		n(n-1)/2		n(n+1)/2-1

2.6 (多维)数组(Cont.)

带状(对角)矩阵的压缩存储

- 所有非零元素都集中在以主对角线为中心的带状区域内
- s = 2，称s为带宽，只存储带区内的非零元素(压缩存储)
- 每行元素最多2s+1个
- 元素个数：(2s+1)n-(s+1)s
- a_{ij}存储位置：k= (2s+1)i+(j-i)？
- 带上元素：|i-j|<=s
- 带宽s<=(n-1)/2

$$\begin{bmatrix} a_{00} & \overbrace{a_{01} & a_{02}}^{s} & 0 & 0 & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & a_{13} & 0 & 0 & 0 & 0 & 0 \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 & 0 & 0 \\ 0 & \alpha & \alpha & \alpha & \alpha & \alpha & 0 & 0 & 0 \\ 0 & 0 & \alpha & \alpha & \alpha & \alpha & \alpha & 0 & 0 \\ 0 & 0 & 0 & \alpha & \alpha & \alpha & \alpha & \alpha & 0 \\ 0 & 0 & 0 & 0 & \lambda & \lambda & \lambda & \lambda & \lambda \\ 0 & 0 & 0 & 0 & 0 & \delta & \delta & \delta & \delta \\ 0 & 0 & 0 & 0 & 0 & 0 & \gamma & \gamma & \gamma \end{bmatrix} n=9$$

2.6 (多维)数组(Cont.)

- 稀疏矩阵的压缩存储 ----三元组顺序表
  - 如何只存储非零元素?
    - 稀疏矩阵中的非零元素的分布没有规律。
  - 将稀疏矩阵中的每个非零元素表示为:
    - (行号，列号，非零元素值)—三元组表

```
typedef  struct {
      int  i, j ;
      ElemType  v ;
}  Triple ;
typedef  struct  {
    Triple  data[MaxSize+1];
     int  mu, nu, tu;     ; //总行号，列号，非0元素个数
} TSMatrix;
```

$$M=\begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix}_{6\times 7}$$

2.6 (多维)数组(Cont.)

- 稀疏矩阵的压缩存储 ----三元组顺序表
  - 如何存储三元组表？
    - 按行优先的顺序存到一个三元组数组。

$$A=\begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix}_{6\times 7}$$

	i	j	v
0	0	1	12
1	0	2	9
2	2	1	-3
3	2	5	14
4	3	2	24
5	4	1	18
6	5	0	15
7	5	3	7
	.	.	.
M-1	.	.	.

data

mu:矩阵行数6

nu:矩阵列数7

tu:非零元数8

2.6 (多维)数组(Cont.)

稀疏矩阵的转置算法

$$A=\begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix}_{6\times 7} \longrightarrow A^T=\begin{bmatrix} 0 & 0 & -3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{7\times 6}$$

0	0	1	12
1	0	2	9
2	2	0	-3
3	2	5	14
4	3	2	24
5	4	1	18
6	5	0	15
7	5	3	-7
	.	.	.
M-1	.	.	.

$\xrightarrow{?}$

0	0	2	-3
1	0	5	15
2	1	0	12
3	1	4	18
4	2	0	9
5	2	3	24
6	3	5	-7
7	5	2	14
	.	.	.
M-1	.	.	.

2.6 (多维)数组(Cont.)

稀疏矩阵的压缩存储 ----十字链表

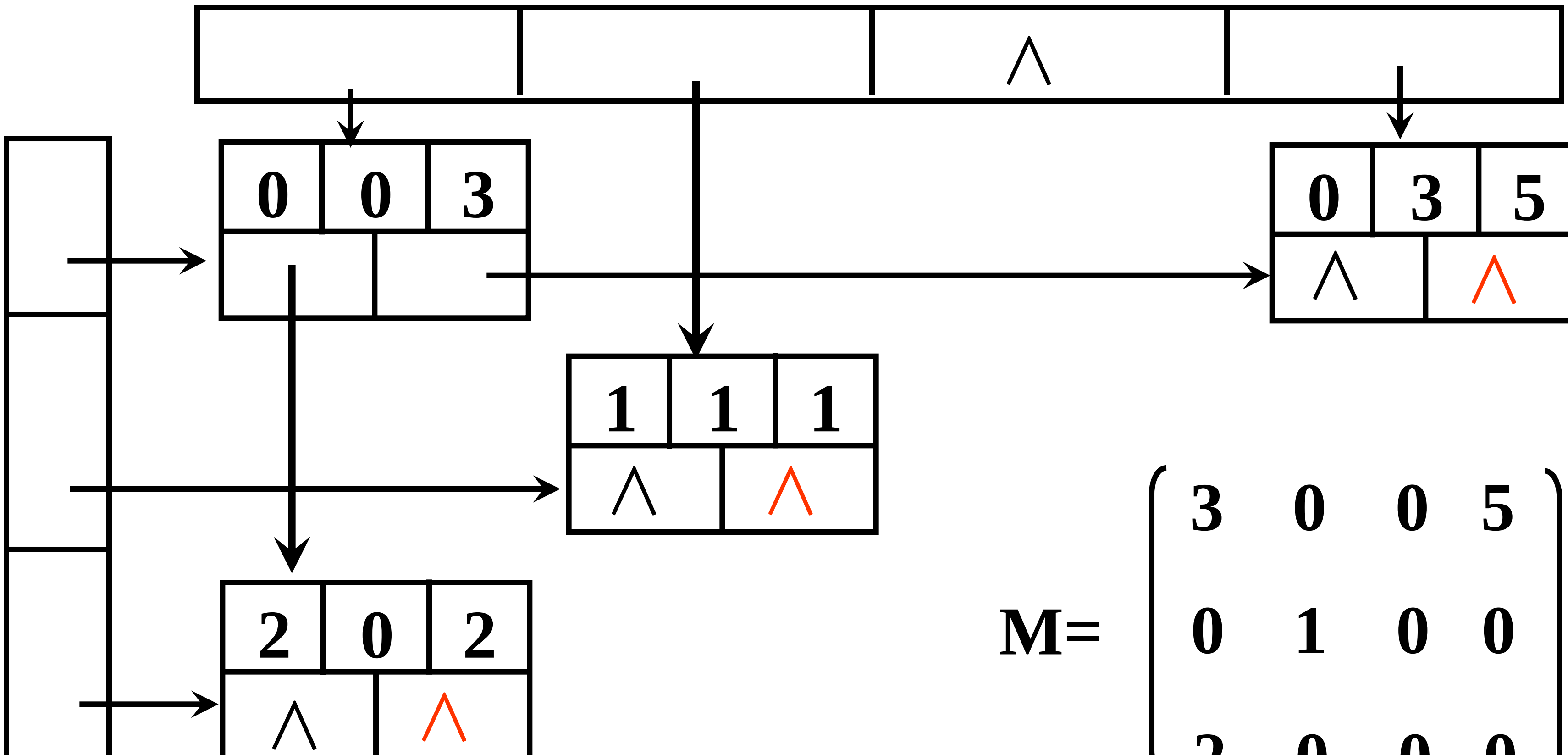
- 采用链接存储结构存储三元组表，每个非零元素对应的三元组存储为一个链表结点，结构为：

<table>
<tr><td colspan="2">row</td><td colspan="2">col</td><td colspan="2">item</td></tr>
<tr><td colspan="3">down</td><td colspan="3">right</td></tr>
</table>

row：存储非零元素的行号
col：存储非零元素的列号
item：存储非零元素的值
right：指针域，指向同一行中的下一个三元组
down：指针域，指向同一列中的下一个三元组

2.6 (多维)数组(Cont.)

- 稀疏矩阵的压缩存储 ----十字链表



$$M=\begin{pmatrix} 3 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

2.7 广义表

- 广义表：*n*（*n*≥0）个数据元素的有限序列，记作：

$$LS=(a_1, a_2, \ldots\ldots, a_n)$$

其中：*LS*是广义表的名称，a_i（$1 \le i \le n$）可以是单个的数据元素，也可以是一个广义表，分别称为*LS*的单个元素（或原子）和子表。

- 长度：广义表*LS*中的直接元素的个数；
- 深度：广义表*LS*中括号的最大嵌套层数。
- 表头：广义表*LS*非空时，称第一个元素为*LS*的表头；
- 表尾：广义表*LS*中除表头外其余元素组成的广义表。

2.7 广义表(Cont.)

- 广义表示例：
  - A =（a，（b，a，b），（），c，（（（2）））)；
  - B =（）；
  - C =（e）；
  - D =（A，B，C）；
  - E =（a，E）；
- 广义表性质：
  - 广义表的元素可以是子表，子表的元素还可以是子表，……，广义表是一个多层次的结构（层次性）；
  - 一个广义表可以被其他广义表所共享（共享性）。
  - 广义表可以是其本身的子表（递归性）。

2.7 广义表(Cont.)

- 广义表基本操作：

①Cal（L）:返回广义表L的第一个元素

②Cdr（L）:返回广义表L除第一个元素以外的所有元素

③Append（L，M）:返回广义表L+M

④Equal（L，M）:判广义表L和M是否相等

⑤Length（L）:求广义表L的长度

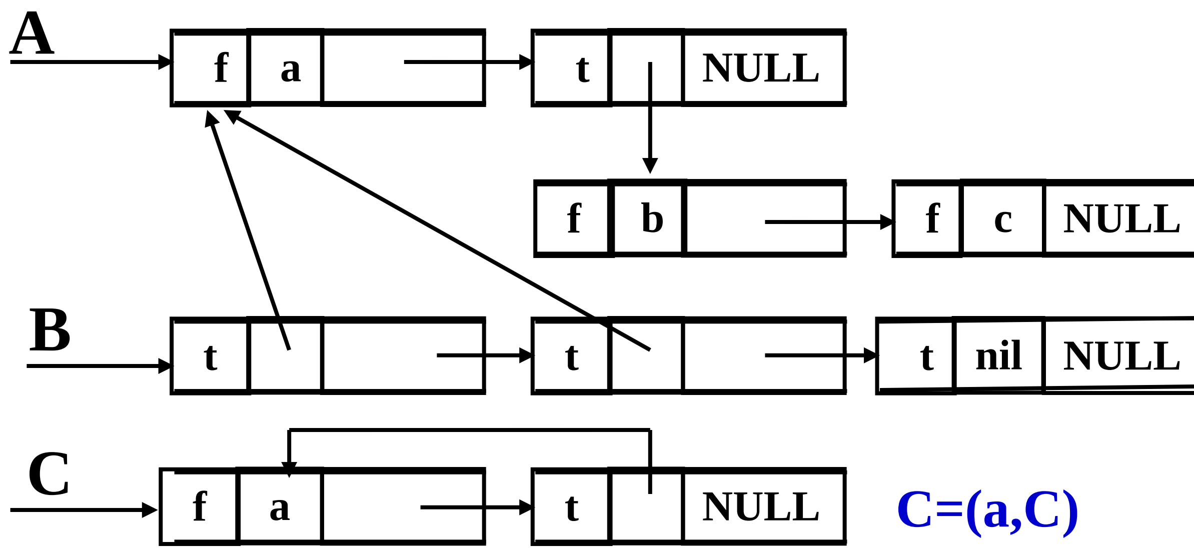
- 广义表存储结构

2.7 广义表(Cont.)

广义表存储结构

A=(a,(b,c))

B=(A,A,( ))



C=(a,C)

```
struct  listnode {
      listnode  *link ;
      boolean  tag ;
      union  {
            char  data ;
            listnode  *dlink ;
      } ;
 } ;
typedef  listnode  *listpointer ;
```

2.7 广义表(Cont.)

广义表操作的实现

```
bool  Equal( listpointer  S, listpointer T )
{    boolean  x, y ;
     y = FALSE ;
     if  ( ( S == NULL ) && ( T == NULL ) )
          y = TRUE ;
     else if  ( ( S != NULL ) && ( T != NULL ) )
          if  ( S→tag == T→tag )
            {  if ( S→tag == FALSE
                 { if ( S→element.data == T→element.data )
                        x = TRUE ;
                     else
                        x = FALSE ;
               else
                    x = Equal( S→element.data,T→element.data );
               if ( x== TRUE )
                    y = Equal( S→link, T→link ) ;
            }
        return y ;
}  //S和T均为非递归的广义表
```

本章小结

- 知识点：
  - 线性表、栈、队列、串、（多维）数组、广义表
- 知识点体系结构

ADT基本数据结构
- ADT定义 逻辑结构
  - 定义及相关术语 } 静态的结构
  - 逻辑结构及其特征 } 静态的结构
  - 基本操作（算法）→ 动态的操作
- ADT实现 存储结构
  - 存储结构（描述） } 静态的结构
  - 存储结构特点 } 静态的结构
  - 存储结构的定义 } 静态的结构
  - 操作（算法）实现→动态的操作
  - 算法的性能
- 应用

本章小结

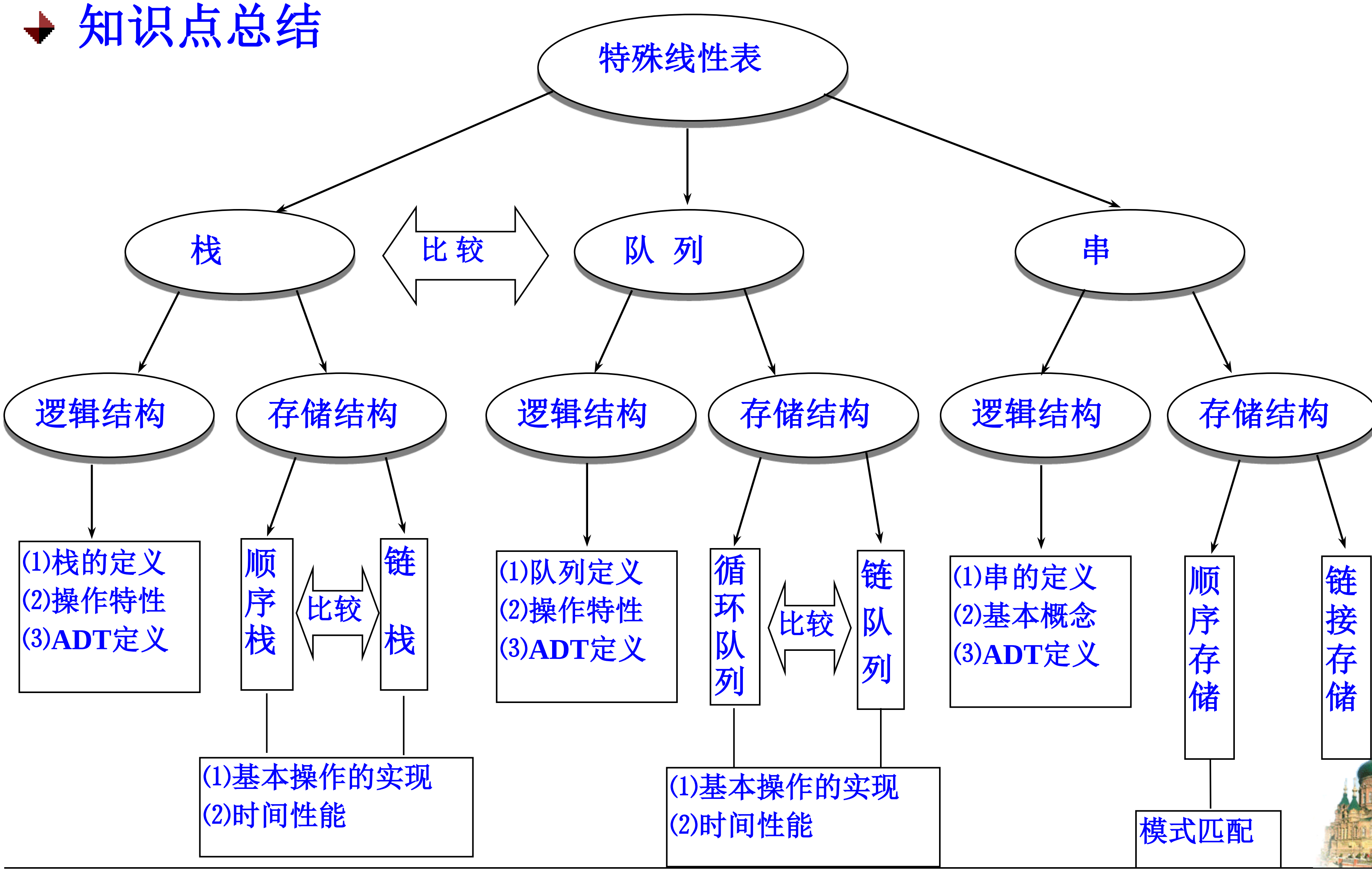
知识点总结

- 线性表
  - 逻辑结构
    - 基本概念
      - (1)线性表定义
      - (2)逻辑特征
    - 抽象数据类型定义
      - (1)ADT定义
      - (2)基本操作
  - 存储结构
    - 顺序存储
      - (1)顺序表的特点
      - (2)顺序表类定义
      - (3)基本操作的实现及时间性能
    - 链接存储
      - (1)单链表的特点
      - (2)单链表类定义
      - (3)基本操作的实现及时间性能
    - 其他存储
      - (1)循环链表
      - (2)双链表
      - (3)静态链表
      - (4)间接寻址

顺序存储 ⟺ 比较 ⟺ 链接存储

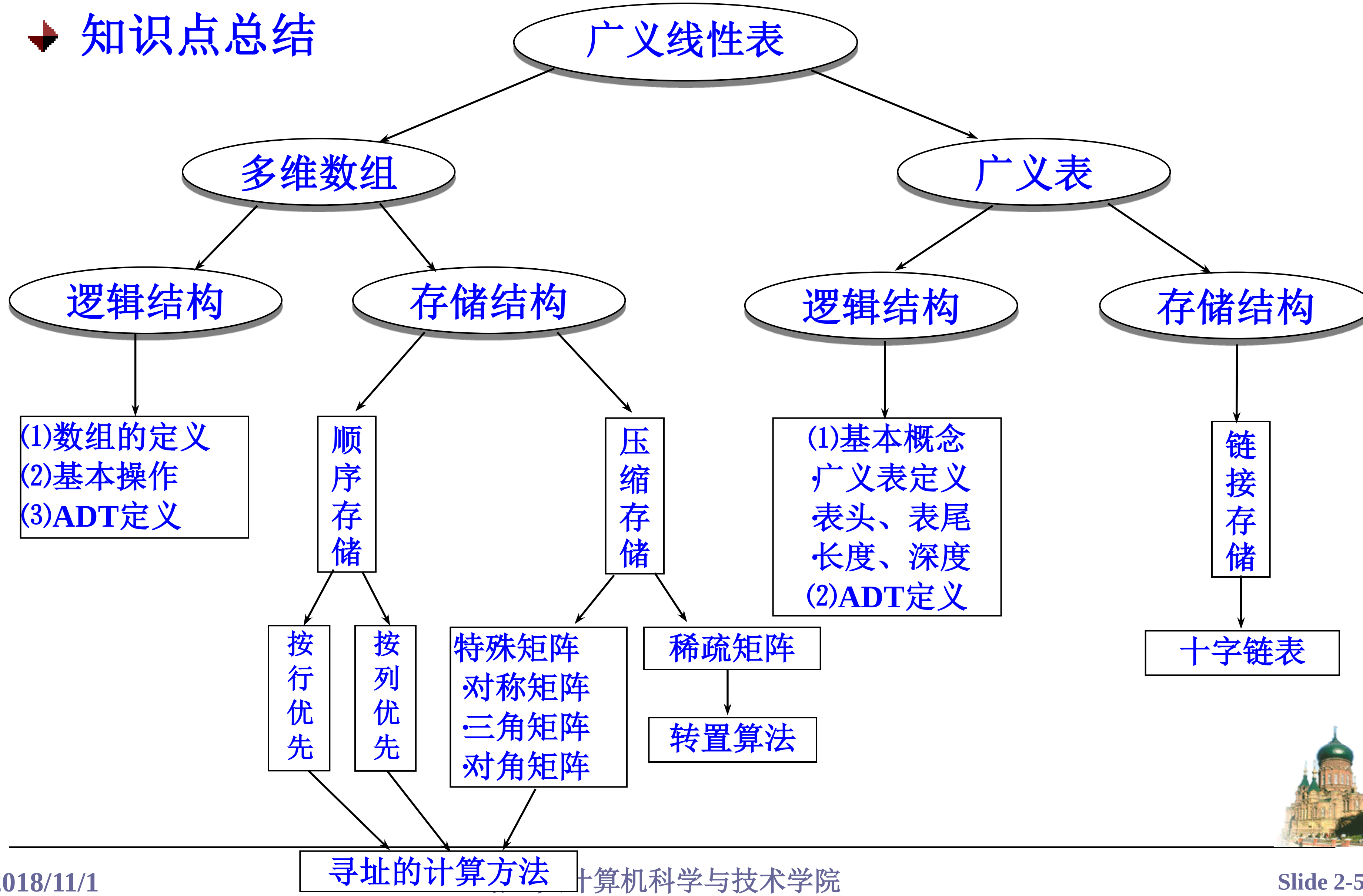
本章小结

- 知识点总结


特殊线性表
栈
比 较
队 列
串
逻辑结构
存储结构
逻辑结构
存储结构
逻辑结构
存储结构
(1)栈的定义
(2)操作特性
(3)ADT定义
顺序栈
比较
链栈
(1)队列定义
(2)操作特性
(3)ADT定义
循环队列
比较
链队列
(1)串的定义
(2)基本概念
(3)ADT定义
顺序存储
链接存储
(1)基本操作的实现
(2)时间性能
(1)基本操作的实现
(2)时间性能
模式匹配

本章小结

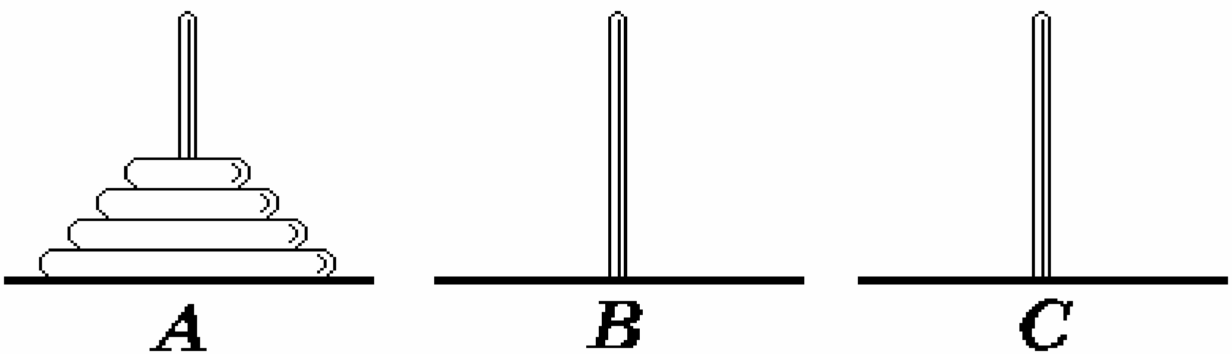
知识点总结


广义线性表
多维数组
广义表
逻辑结构
存储结构
逻辑结构
存储结构
(1)数组的定义
(2)基本操作
(3)ADT定义
顺序存储
压缩存储
(1)基本概念
·广义表定义
·表头、表尾
·长度、深度
(2)ADT定义
链接存储
按行优先
按列优先
特殊矩阵
·对称矩阵
·三角矩阵
·对角矩阵
稀疏矩阵
转置算法
十字链表
寻址的计算方法

2.3.3 栈与递归调用(Cont.)

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。



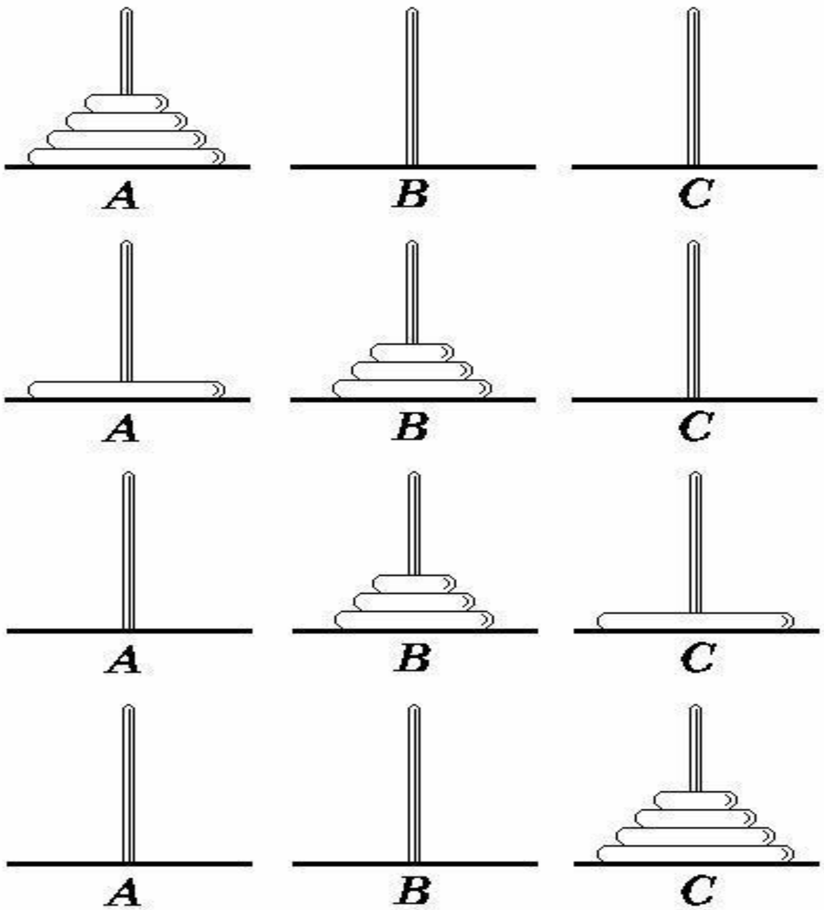
2.3.3 栈与递归调用(Cont.)

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

2.3.3 栈与递归调用(Cont.)

- 汉诺塔问题的递归求解:

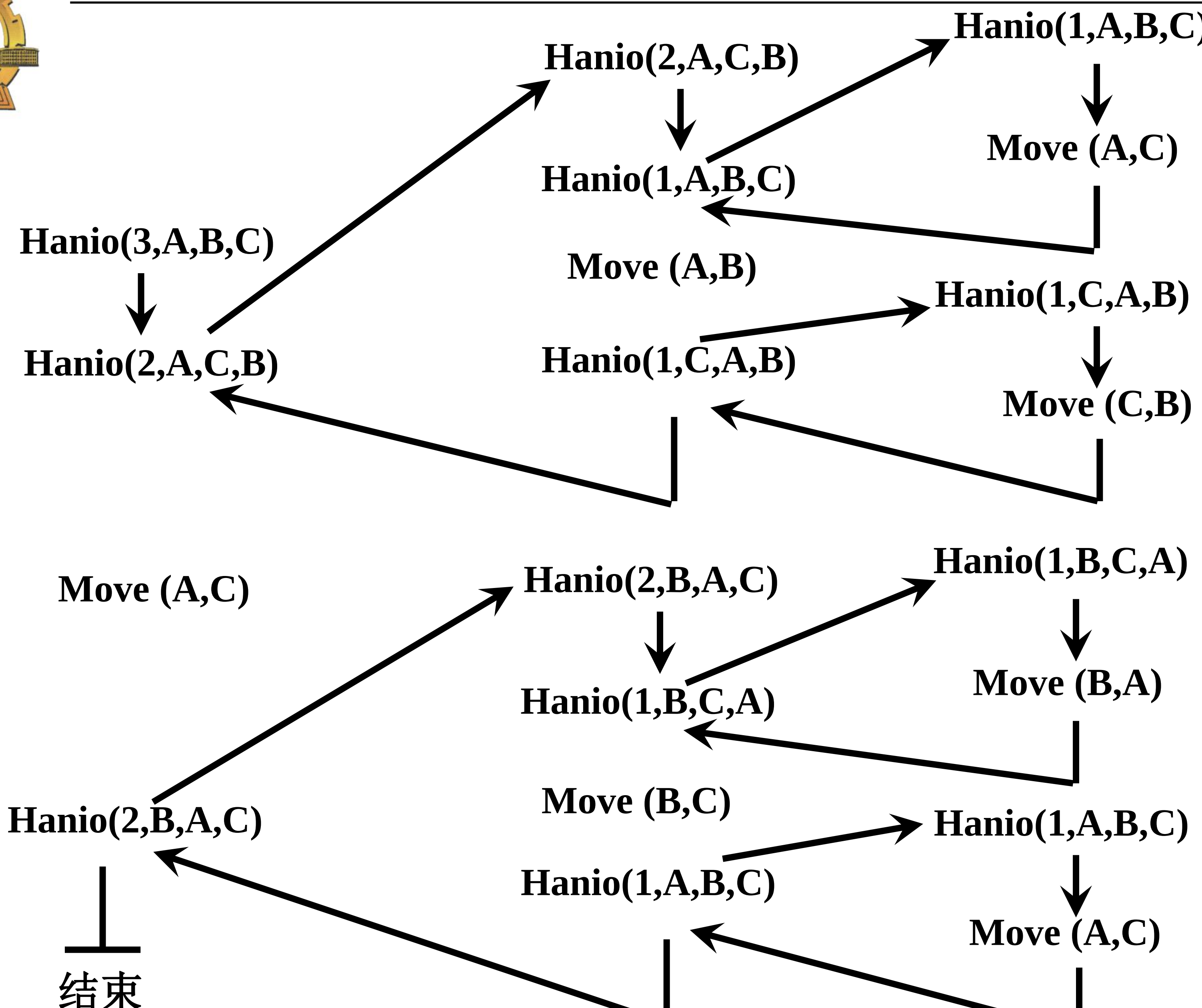
```
void Hanoi(int n, char A, char B, char C)
{
    if (n==1)  Move(A, C);
     else {
              Hanoi(n-1, A, C, B);
              Move(A, C);
              Hanoi(n-1, B, A, C);
     }
}
```



- 时间复杂度?

2.3.3 栈与递归调用(Cont.)

- 递归函数的运行轨迹
  - 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;
  - 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;
  - 在返回路线上标出本层调用所得的函数值。



Hanio(3,A,B,C)
Hanio(2,A,C,B)
Hanio(2,A,C,B)
Hanio(1,A,B,C)
Hanio(1,A,B,C)
Move (A,C)
Move (A,B)
Hanio(1,C,A,B)
Hanio(1,C,A,B)
Move (C,B)
Move (A,C)
Hanio(2,B,A,C)
Hanio(1,B,C,A)
Hanio(1,B,C,A)
Move (B,A)
Move (B,C)
Hanio(1,A,B,C)
Hanio(1,A,B,C)
Move (A,C)
Hanio(2,B,A,C)
结束