数据结构与算法

Data Structures and Algorithms

第二部分 线性表

回顾：栈、队列 和 串

1. 表达式求解

中缀表达式转换成后缀表达式
后缀表达式计算表达式的值

2. 队列

链队列
顺序队列 （循环队列）

3. 串

串的ADT

（1）顺序存储

（2）链式存储

（3）模式匹配
朴素模式匹配

3、模式匹配(朴素模式匹配算法)

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… S_{n-1}”(主串)和T=“T_0 T_1…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)

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。

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-j+1），模式串指针需复位（j=0）。

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。

```
int StrMatch_BF ( char* S, char* T, int pos=0)
{   /*为主串S、模式T长度分别为lenS和lenT，且字符串采用顺序存储*/
    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
}
```

i,j=1开始又怎么样？

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：

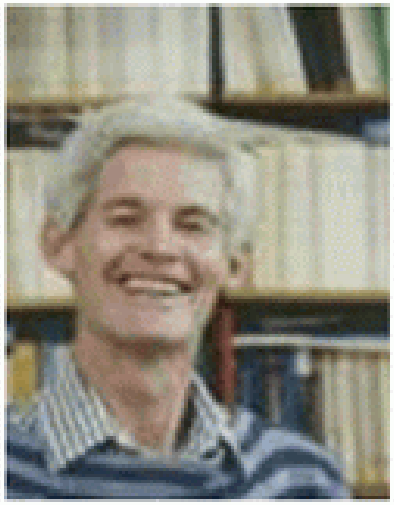
$$\sum_{i=0}^{n-m} p_i(i+1)m = \frac{m}{n-m+1}\sum_{i=0}^{n-m}(i+1) = \frac{1}{2}m(n-m+2)$$

最坏情况下的平均时间复杂度为O(n*m)。

Cook于1970年证明的一个理论得到，任何一个可以使用被称为下推自动机的计算机抽象模型来解决的问题，也可以使用一个实际的计算机（更精确的说，使用一个随机存取机）在与问题规模对应的时间内解决。特别地，这个理论暗示存在着一个算法可以在大约$m+n$的时间内解决模式匹配问题。

D.R.Knuth和V.R.Pratt努力地重建了Cook的证明，由此创建了这个模式匹配算法。大概是同一时间，J.H.Morris在考虑设计一个文本编辑器的实际问题的过程中创建了差不多是同样的算法(克努特-莫里斯-普拉特, KMP算法)。

并不是所有的算法都是“灵光一现”中被发现的，而理论化的计算机科学确实在一些时候会应用到实际的应用中。


斯蒂芬·库克

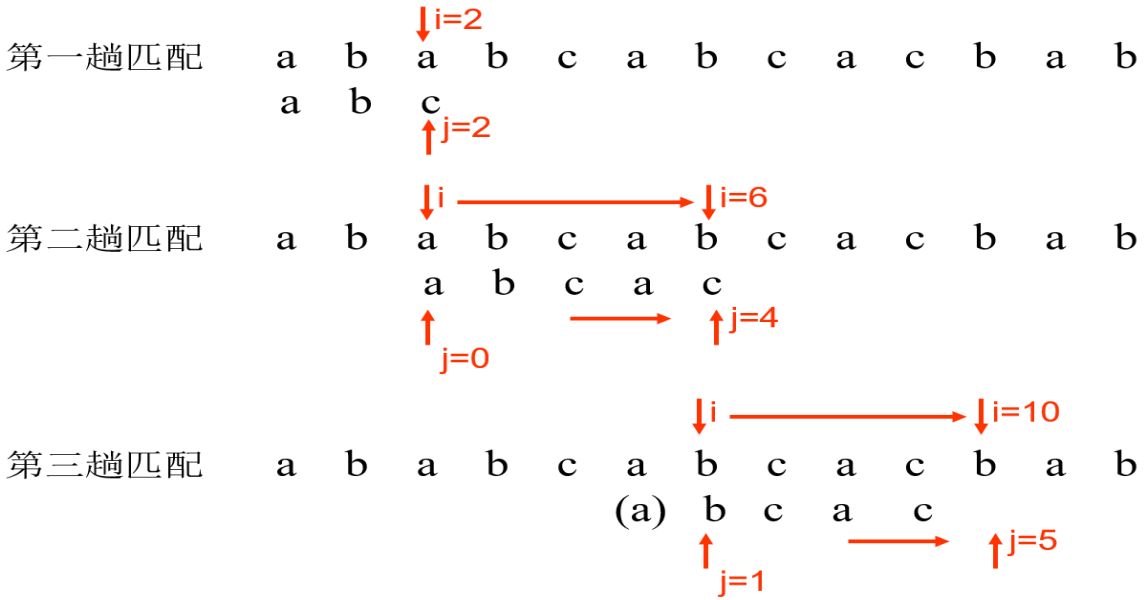
斯蒂芬·库克（Stephen A. Cook）：
1961 年从 University of Michigan 获得其学士学位，于 1962年和1966 年从哈佛大学分别获得其硕士与博士学位。

1966年到1970年，Stephen 在加州 Berkeley分校担任助理教授职务。1970年，Stephen加盟多伦多大学并工作直到现在。他是NP完全性理论的奠基人，1971年发表Cook定理奠定了NP完全理论的基础而获1982年图灵奖。Cook是对计算复杂性理论有突出贡献的计算机科学家之一。

在1998年加盟苹果电脑担任全球业务高级副总裁之前，Cook 先生曾任康柏 (Compaq) 企业材料副总裁,负责采购、管理康柏的产品存货。在这之前，Cook 先生是 Intelligent Electronics 经销商部门的首席运营官。Cook 先生还曾在 IBM 供职12年之久，他在 IBM 最近的职务为北美业务执行主管，负责 IBM 的 Personal Computer Company 在北美和拉美的制造和分销运作。

2.5.2 KMP算法----改进的模式匹配算法

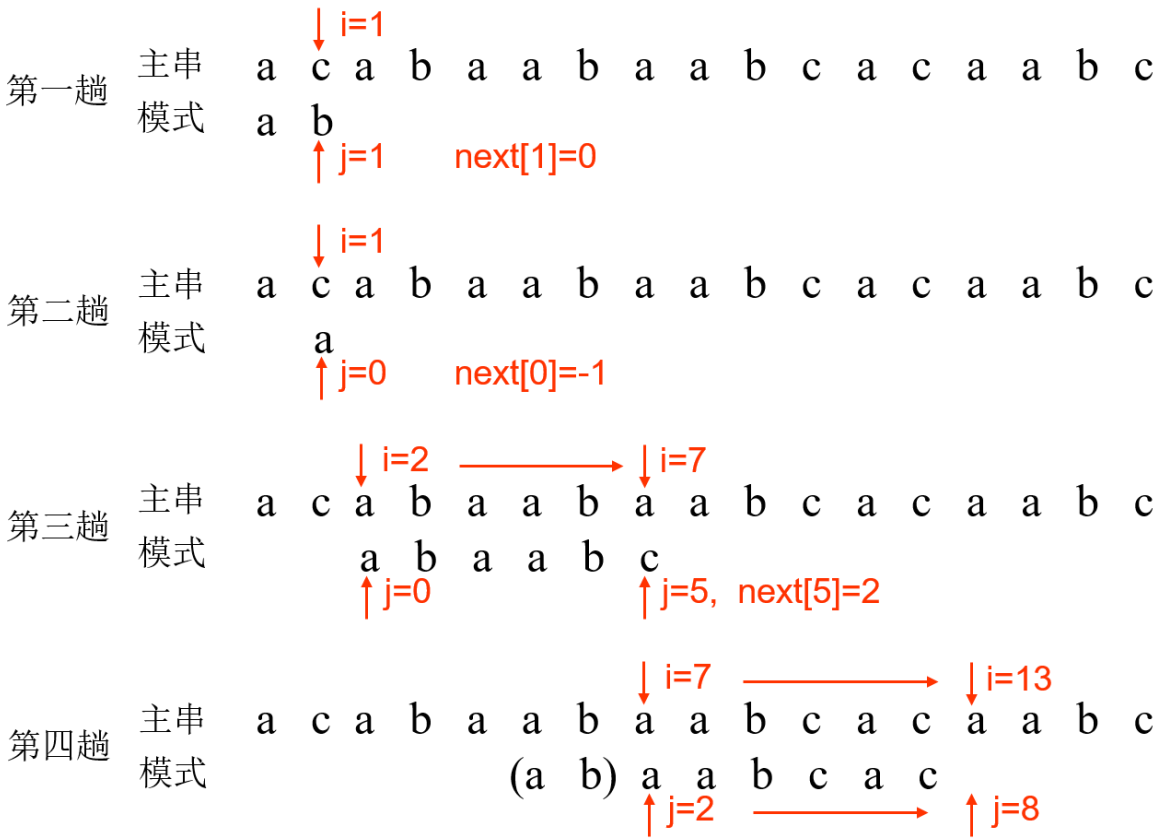
- 为什么BF算法时间性能低？
  在每趟匹配不成功时存在大量回溯，没有利用已经部分匹配的结果；
- 如何在匹配不成功时主串不回溯？
  主串不回溯，模式就需要向右滑动一段距离；
- 如何确定模式的滑动距离？
  利用已经得到的“部分匹配”的结果；
  将模式向右“滑动”尽可能远的一段距离(next[j])后，继续进行比较；
- 字符串的前缀和后缀：（主串ababcabcacbab,模式abcac)



需要解决的问题：

(1)当匹配过程中产生“失配”（$s_i \neq t_j$）时，模式串“向右滑动”可行的距离多远？

(2)当主串中第 *i* 个字符与模式中第 *j* 个字符“失配”时，主串中第 *i* 字符应与模式中哪个字符再比较？



设：s="$s_0s_1s_2\ldots s_{n-1}$"，t="$t_0t_1t_2\ldots t_{m-1}$"，当 $s_i \neq t_j$ 时，存在：

"$t_0t_1\ldots t_{j-1}$" = "$s_{i-j}s_{i-j+1}\ldots s_{i-1}$"

若模式串t中存在可互相重叠的最大真子串满足：

"$t_0t_1\ldots t_{k-1}$" = "$t_{j-k}t_{i-k+1}\ldots t_{j-1}$"　　（$0<k<j$）

则下一次比较可直接从模式的第$k+1$个字符t_k开始与目标串的第$i+1$个字符s_i相对应继续进行下一趟匹配。

若模式串中不存在上述真子串,则下一次比较可直接从模式串的第1个字符t_0开始与目标串s的第$i+1$个字符s_i相对应继续进行下一趟的匹配。

$$Next[j] = \begin{cases} -1，\ j = 0 \\ Max\{ k \mid_{1<k<j} \}，\quad "t_0t_1\ldots t_{k-1}" = "t_{j-k}t_{j-k+1}\ldots t_{j-1}" \text{ 且 } t_k \mathrel{!}= t_j \\ 0，\text{其他情况} \end{cases}$$

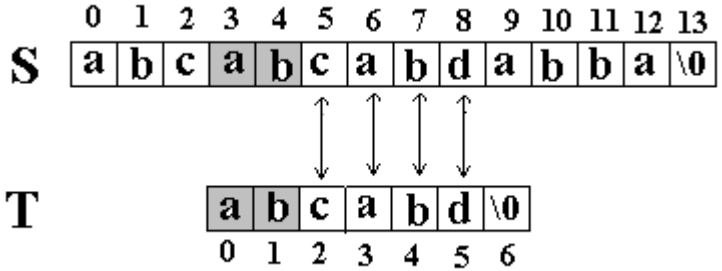
t="abaabcac"

j	0	1	2	3	4	5	6	7
t	a	b	a	a	**b**	c	a	c
next[j]	-1	0	0	1	0	2	0	1

t="abaababc"

j	0	1	2	3	4	5	6	7
t	a	b	a	a	b	**a**	b	c
next[j]	-1	0	0	1	0	0	3	2

如下例所示：当第一次搜索到 S[5] 和 T[5] 不等后，S 下标不是回溯到 1，T 下标也不是回溯到开始，而是根据 T 中T[5] == 'd' 的模式函数值（next[5]=2），直接比较 S[5] 和 T[2] 是否相等，因为相等，S 和T的下标同时增加；因为又相等，S和T的下标又同时增加…最终在 S中找到了T。



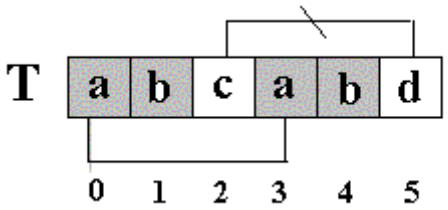
KMP匹配算法和简单匹配算法效率比较，一个极端的例子是：

在S="AAAAAA…AAB"(100个A)中查找T="AAAAAAAAAB"，简单匹配算法每次都是比较到T的结尾，发现字符不同，然后T的下标回溯到开始，S的下标也要回溯相同长度后增1，继续比较。如果使用 KMP 匹配算法，就不必回溯。时间复杂度从 O(m*n)降为 O(m+n) 。

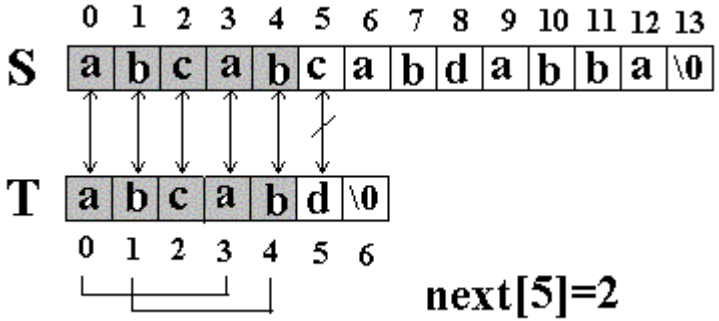
KMP算法的核心思想：

利用已经得到的部分匹配信息来进行后面的匹配过程。

特殊情况： T[5]=='d'的模式函数值等于2, 即(next[5]=2)表示T[5]=='d'的前面有2个字符和开始的两个字符相同，且T[5]=='d'不等于开始的两个字符之后的第三个字符(T[2]='c')。



如果开始的两个字符之后的第三个字符也为' d',那么，尽管T[5]=='d'的前面有2个字符和开始的两个字符相同，T[5]=='d'的模式函数值也不为2，而是为0。



next函数（换一种定义）：

(1)next[0]= -1，任何串的第一个字符的模式值规定为-1

(2)next[j] = -1，模式串t中下标为j的字符，如果与首字符相同，且j的前面的1…k 个字符与开头的1..k 个字符不等(或者相等但t[k]==t[j]), 1≤k<j

如：t=”abCabCad” 则 next[6]=-1，因t[3]=t[6]

(3)next[j] = k，模式串t中下标为j的字符，如果j的前面k个字符与开头的k个字符相等，且$t_j \neq t_k$, 1≤k<j，即

“$t_0t_1t_2\ldots,\ldots,t_{k-1}$ ” = “ $t_{j-k}t_{j-k+1}t_{j-k+2}\ldots t_{j-1}$”

且 $t_j \neq t_k$，1≤k<j

(4)next[j]=0，除(1)、(2)、(3)的其他情况。

【例2-17】求T="abcac"的模式函数的值

next[0]= -1 根据 (1)

next[1]= 0 根据 (4) 因（3）有1<=k<j;不能说，j=1,T[j-1]==T[0]

next[2]= 0 根据 (4) 因（3）有1<=k<j;（T[0]=a）!=（T[1]=b）

next[3]= -1 根据 (2)

next[4]= 1 根据 (3) T[0]=T[3] 且 T[1]=T[4]

下标	0	1	2	3	4
T	a	b	c	a	c
next	-1	0	0	-1	1

若T="abcab"

为什么T[0]==T[3], 还会有next[4]=0 ?

因为T[1]==T[4]

根据 (3)" 且T[j] != T[k]"被划入 (4)。

下标	0	1	2	3	4
T	a	b	c	a	b
next	-1	0	0	-1	0

【例2-18】求T=”ababcaabc” 的模式函数的值

next[0]= -1　根据 (1)

next[1]= 0　根据 (4)

next[2]= -1　根据 (2)

next[3]= 0　根据 (3) 虽T[0]=T[2] 但T[1]=T[3] 被划入 (4)

next[4]= 2　根据 (3) T[0]T[1]=T[2]T[3] 且T[2] !=T[4]

next[5]= -1　根据 (2)

next[6]= 1　根据 (3) T[0]=T[5] 且T[1]!=T[6]

next[7]= 0　根据 (3) 虽T[0]=T[6] 但T[1]=T[7] 被划入 (4)

next[8]= 2　根据 (3) T[0]T[1]=T[6]T[7] 且T[2] !=T[8]

即：

下标	0	1	2	3	4	5	6	7	8
T	a	b	a	b	c	a	a	b	c
next	-1	0	-1	0	2	-1	1	0	2

next[3]=0，而不是=1；

next[6]=1，而不是= -1；

next[8]=2，而不是= 0；

【例2-19】求 T=”abCabCad” 的模式函数的值

下标	0	1	2	3	4	5	6	7
T	a	b	C	a	b	C	a	d
next	-1	0	0	-1	0	0	-1	4

next[5]= 0 根据 (3) 虽T[0]T[1]=T[3]T[4]，但T[2]==T[5]

next[6]= -1 根据 (2) 虽前面有abC=abC， 但T[3]==T[6]

next[7]=4 根据 (3) 前面有abCa=abCa， 且 T[4]!=T[7]

若T[4]==T[7]，即T=” adCadCad”，则

next[7]=0, 而不是= 4，因为T[4]==T[7]

下标	0	1	2	3	4	5	6	7
T	a	d	C	a	d	C	a	d
next	-1	0	0	-1	0	0	-1	0

next 函数值究竟是什么含义？

设在字符串S中查找模式串T，若S[m]!=T[n]，

那么，取T[n]的模式函数值next[n]

(1) next[n]= -1 表示S[m]和T[0]间接比较过了，不相等，下一次比较 S[m+1] 和T[0]；

(2) next[n]=0 表示比较过程中产生了不相等，下一次比较 S[m] 和 T[0]；

(3) next[n]= k >0 但k<n，表示S[m]的前k个字符与T中的，开始k个字符已经间接比较相等了，下一次比较S[m]和T[k]相等吗？

(4) 其他值。

求串T的模式值next[n]的函数：

```
void GetNext( const char *T,  int next[] )
{    // 求模式串T的next函数值
     int j = 0,   k = -1;
     next[0] = -1;
     while ( T[j]  !=  '\0' )
     {
        if ( k == -1 || T[j] == T[k] )
        {
             ++j; ++k;
             if ( T[j] != T[k] )
                  next[j] = k;
             else
                  next[j] = next[k];
        }// if
        else
             k = next[k];
     }// while
}  // GetNext
```

KMP算法：

```
int  KMP( char  *s ,  char  *t , int  next[] )
{
    int  index=0,  i=0,  j=0;
    if(  !s||!t||t[0]=='\0'||s[0]=='\0'  )
       return -1; //空指针或空串，返回-1
    while(s[i]!='\0'&&t[j]!='\0' )
    {
       if(s[i]== t[j])
       {  ++i; ++j; } // 继续比较后继字符
       else
       {
         index += j-next[j];
         if(next[j]!=-1)   j=next[j]; // 模式串右移
         else  { j=0; ++i; }
       }
    }//while
    return((t[j]=='\0') ? Index : -1);  // 匹配失败
}  //KMP
```

第 四 章

数 组

2.6 数组(ARRAY)

2.6.1 抽象数据型数组

- 数组是由下标（index）和值（value）组成的序对（index，value）的集合。
- 也可以定义为是由相同类型的数据元素组成有限序列。
- 数组在内存中是采用一组连续的地址空间存储的，正是由于此种原因，才可以实现下标运算。
- 所有数组都是一个一维向量。

数组1：(a_1, a_2, a_3, …, a_i, …, a_n)；

数组2：(a_{11}, …, a_{1n}, a_{21}, …, a_{2n}, …, a_{ij}, …, a_{m1}, …, a_{mn})；

$1 \le i \le m$, $1 \le j \le n$;

数组3：(a_{111}, …, a_{11n}, a_{121}, …, a_{12n}, …, a_{ijk}, …, a_{mn1}, …, a_{mnp})；

$1 \le i \le m$, $1 \le j \le n$, $1 \le k \le p$;

n 维数组中含有 $\prod_{i=1}^{n} b_i$ 个数据元素,每个元素都受着 n 个关系的约束。在每个关系中，元素$a_{j1j2\cdots jn}$（$0 \le j_i \le b_i-2$）都有一个直接后继元素。

因此，数组仍是一种特殊形式的线性表。对二维数组可以理解成一维数组，其每个元素又是一个一维数组；以此类推，所谓 n 维数组同样如此。

操作：Create （ ）；建立一个空数组
Retrieve（array，index）；返回第 index 个元素
Store（array，index，value）；
在数组array中，为第index个元素赋值value

注：由于高级语言中都提供了数组，本课略去操作。

2.6.2　数组的实现

1、数组的顺序存储

数组的顺序表示，指的是在计算机中，用一组连续的存储单元来实现数组的存储。目前的高级程序设计语言都是这样实现的。

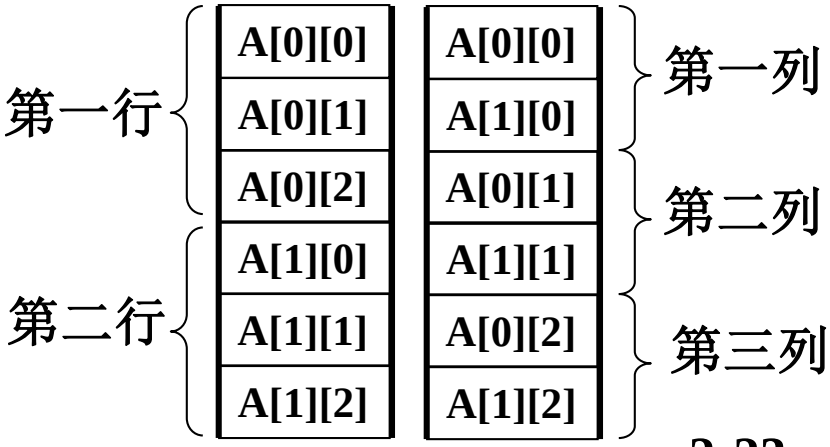
两种存储方式：

一是按行存储（C语言、PASCAL等）；

二是按列存储（FORTRAN等）；

ElementType　A[2][3]；

$$A = \begin{pmatrix} A[0][0] & A[0][1] & A[0][2] \\ A[1][0] & A[1][1] & A[1][2] \end{pmatrix}$$



对二维数组有：

$$LOC(A[i][j]) = LOC(A[0][0]) + n \cdot i + j, \quad 0 \le i \le m-1, \ 0 \le j \le n-1$$

对三维数组有：

$$LOC(A[i_1][i_2][i_3]) = LOC(A[0][0][0]) + d_2 \cdot d_3 \cdot i_1 + d_3 \cdot i_2 + i_3,$$
$$0 \le i_1 \le d_1-1, \ 0 \le i_2 \le d_2-1, \ 0 \le i_3 \le d_3-1$$

同理对n维数组有：

$$LOC(A[i_1][i_2]\cdots[i_n]) = LOC(A[0][0]\cdots[0]) + d_2 \cdot d_3 \cdots i_1 + d_3 \cdot d_4 \cdots d_n \cdot i_2 + d_n \cdot i_{n-1} + i_n$$

或：

$$LOC(A[i_1][i_2]\cdots[i_n]) = LOC(A[0][0]\cdots[0]) + \sum_{r=1}^{n} a_r \cdot i_r$$

而：

$$\begin{cases} a_r = \prod_{a=r+1}^{n} d_a \quad (1 \le r \le n-1) \\ a_n = 1, \ r=n \end{cases}$$

$$0 \le i_1 \le d_1-1, \ 0 \le i_2 \le d_2-1, \ \cdots, \ 0 \le i_n \le d_n-1$$

2、数组的压缩存储

(1) 特殊矩阵

若n 阶矩阵 A 中的元素满足下述性质

$$a_{ij} = a_{ji} \qquad 1 \le i,\ j \le n$$

则称 n 阶对称阵。

对于对称矩阵, 为实现节约存储空间, 我们可以为每一对对称元素分配一个存储空间, 这样, 原来需要的 n^2 个元素压缩存储到 n(n+1)/2 个元素空间。

对称关系：

设sa[1 . . n(n+1)/2] 做为 n 阶对称阵 A 的存储结构, 则 Sa[k] 和 a_{ij} 的一一对应关系为:

$$k = \begin{cases} i(i-1)/2 + j & \text{当 } i \ge j \\ j(j-1)/2 + i & \text{当 } i < j \end{cases}$$

0元素

a_{ij}，（i<j）

$$A_{nn}\begin{bmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ a_{41} & a_{42} & a_{43} & a_{44} & \\ \cdots\cdots & & & & \\ a_{n1} & a_{n2} & a_{n3} & a_{n4} & \cdots\cdots\ a_{nn} \end{bmatrix}$$

0

非0元素

a_{ij}，（i>=j）

```
ElementType   A(ElementType B[ ] , int i, int j)
{
  int k ;
  if ( i >= j )
     {  k = i*(i-1)/2 + j ;
        return ( B[k] ) ;    }
  else
        return( 0 ) ;
}
```

A[i][j]呢？

B[k]	a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	…	a_{ij}	…	a_{n1}	…	a_{nn}
k =	1	2	3	4	5	6		i(i-1)/2+j		n(n-1)/2+1		n(n+1)/2

A[*i*][*j*]=A(*B*, *i*, *j*)

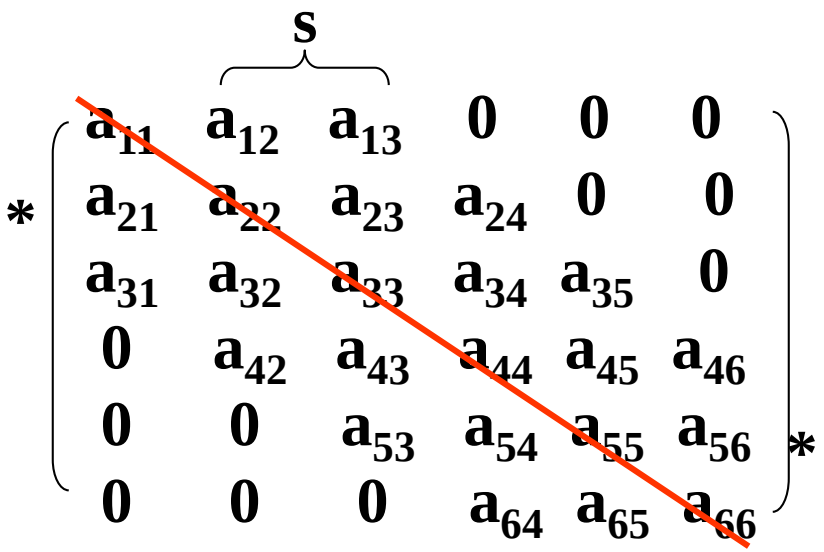
映像关系：

B	a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	…	a_{ij}	…	a_{n1}	…	a_{nn}
k =	1	2	3	4	5	6		i(i-1)/2+j		n(n-1)/2+1		n(n+1)/2

对上（下）三角矩阵可采用同样的方法。

(2) 对角（带状）矩阵

所有非零元素都集中在以主对角线为中心的带状区域内。



如图所示，S = 2，称S为带宽，为实现压缩存储，我们只存储带区内的非零元素。不难总结出：

LOC[4,3]怎么计算？

$$LOC[i,j] = LOC[1,1] + [(2s+1)*(i-1)+(j-i)]*L$$

L = 1; //每个数据元素的字节数

对于特殊矩阵

如：上下三角矩阵、对称矩阵、带状矩阵

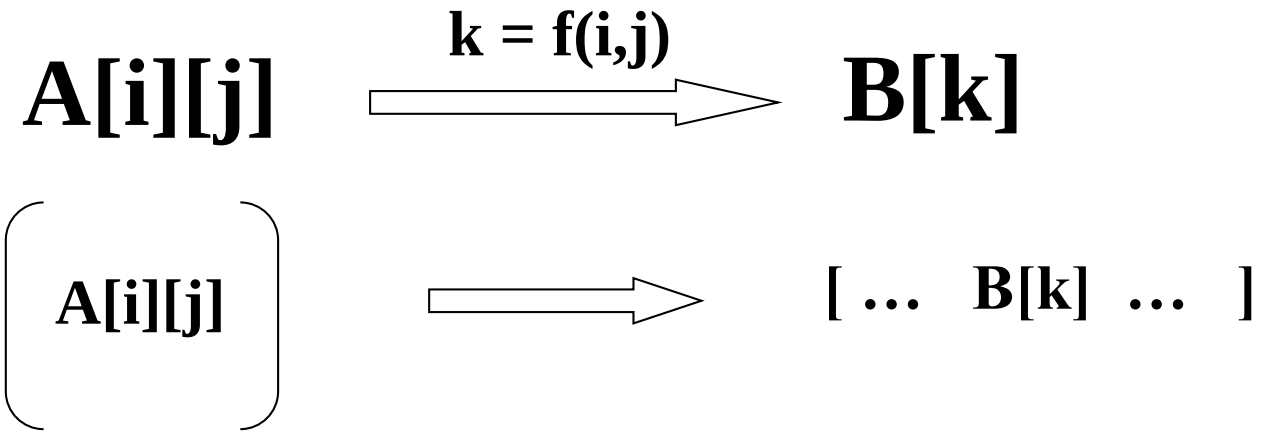
对所有非零元素按行排序，其排列序号k

k = f(i , j) // 对应关系

将所有非零元素存入一维数组B

B[k]=A[i][j]

即：



对称矩阵：

```
ElementType   A( ElementType B[ ] , int i, int j )
{
  int k ;
  if ( i >= j )
     k= i*(i-1)/2 + j ;
  else
     k= j*(j-1)/2 + i ;
  return( B[k] ) ;
}
```

A[i][j] => A(i,j)

```
ElementType   A(ElementType B[ ] , int i, int j)
{
  int k ;
  if ( i >= j )
     {  k = i*(i-1)/2 + j ;
        return ( B[k] ) ;    }
  else
        return( 0 ) ;
}
```

下三角阵

上三角阵

```
ElementType   A(ElementType B[ ] , int i, int j)
{
  int k ;
  if ( i <= j )
     {  k =  i*(i-1)/2 + j ;
        return ( B[k] );     }
  else
        return( 0 ) ;
}
```

带状矩阵，带宽 s ：

```
ElementType   A(ElementType B[ ] , int s, int i, int j)
{
  int k ;
  if ( abs( i-j ) <= s )
      {  k = 1+ (2s+1)*(i-1)+(j-i) ;
         return( B[k] ) ;                }
  else
         return( 0 ) ;
}
```

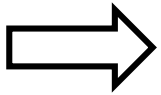
(3) 稀疏矩阵

稀疏矩阵中，零元素的个数远远多于非零元素的个数。为实现压缩存储，我们仍考虑只存储非零元素。行号递增，列号不减。

$$M=\begin{pmatrix}0&12&9&0&0&0&0\\0&0&0&0&0&0&0\\-3&0&0&0&0&14&0\\0&0&24&0&0&0&0\\0&18&0&0&0&0&0\\15&0&0&-7&0&0&0\end{pmatrix}\qquad T=M^T=\begin{pmatrix}0&0&-3&0&0&15\\12&0&0&0&18&0\\9&0&0&24&0&0\\0&0&0&0&0&-7\\0&0&0&0&0&0\\0&0&14&0&0&0\\0&0&0&0&0&0\end{pmatrix}$$

	i	j	v
1	1	2	12
2	1	3	9
3	3	1	-3
4	3	6	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	-7

a.data



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

b.data

```
#define MAXSIZE 100
typedef struct {
    int i, j;
    ElementType e;
} Triple;
typedef struct {
    Triple data[MAXSIZE+1];
    int mu, nu, tu;
} TSMatrix;
```

由稀疏矩阵的三元组表求转置矩阵TransposMatrix；

```
Void TransposMatrix( TSMatrix M，TSMatrix &T )
{ T.mu = M.nu; T.nu = M.mu; T.tu = M.tu;//行列数交换
  if ( T.tu )
    {   q = 1 ; //三元组T的下标
        for ( col = 1; col <= M.nu; ++col ) // M中列号
            for ( p = 1;  p <= M.tu;  ++p ) //M的下标
              { if ( M.data[p].j == col ) // j为列下标
                  {  T.data[q].i = M.data[p].j ;
                     T.data[q].j = M.data[p].i ;
                     T.data[q].e = M.data[p].e ;
                      ++ q ;  }
              }
    }
}
```

分析: T(n) = O(nu · tu)

算法改进:

设向量 num[col], 表示矩阵M中第 col 列中非零元素的个数, cpot[col] 指示M中第 col 列的第一个非零元素在 b 中的恰当位置。

Cpot [1] = 1 ； // 第一列的非零元素

Cpot [col] = cpot[col – 1] + num[col – 1]　　2≤ col ≤a.nu

// 某一列col的第一个非零元素位置 = 上一列首元素位置+非零个数

矩阵M的向量 cpot 的值

col	1	2	3	4	5	6	7
Num[col]	2	2	2	1	0	1	0
Cpot[col]	1	3	5	7	8	8	9

t

	i	j	v
1	1	2	12
2	1	3	9
3	3	1	-3
4	3	6	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	-7

```
for ( col = 1; col <= M.nu; ++col )
        num[ col ] = 0 ;
for ( t = 1; t <= M.tu; ++t )
        ++num[ M.data[ t ].j ] ;//非零个数

cpot [ 1 ] = 1 ;
for(col=2;col<M.nu;col++)
    cpot[col ]=cpot[col -1]+num[col–1];
```

col	1	2	3	4	5	6	7
Num[col]	2	2	2	1	0	1	0
Cpot[col]	1	3	5	7	8	8	9

	i	j	v
1	1	3	-3
2	1	6	15
3	2	1	12
4	2	5	18
5	3	1	9
6	3	4	24
7	4	6	-7
8	6	3	14

	i	j	v
p →	1	2	12
	1	3	9
	3	1	-3
	3	6	14
	4	3	24
	5	2	18
	6	1	15
	6	4	-7

```
Void FastTransposMatrix( TSMatrix M，TSMatrix &T )
{ T.mu = M.nu; T.nu = M.mu; T.tu = M.tu;
  if ( T.tu )
    {  for ( col = 1; col <= M.nu; ++col ) num[ col ] = 0 ;
       for ( t = 1; t <= M.tu; ++t ) ++num[ M.data[ t ].j ] ;
       cpot [ 1 ] = 1 ;
       for(col=2;col<M.nu;col++)
            cpot [ col ] = cpot[ col –1 ] + num[ col – 1 ] ;
       for ( p = 1; p <= M.tu; ++p )
         {  col = M.data[ p ].j ; q = cpot[ col ] ;
            T.data[ q ].i = M.data[ p ].j ;
            T.data[ q ].j = M.data[ p ].i ;
            T.data[ q ].e = M.data[ p ].e ;
            ++cpot[ col ] ;
         }
    }
}
```

转置的改进算法:

$T(n) = O(nu + tu)$

col	1	2	3	4	5	6	7
Num[col]	2	2	2	1	0	1	0
Cpot[col]	1	3	5	7	8	8	9

3、数组的链接式存储

多维数组, 特别是稀疏矩阵可以采用链接式存储。

结点Node:

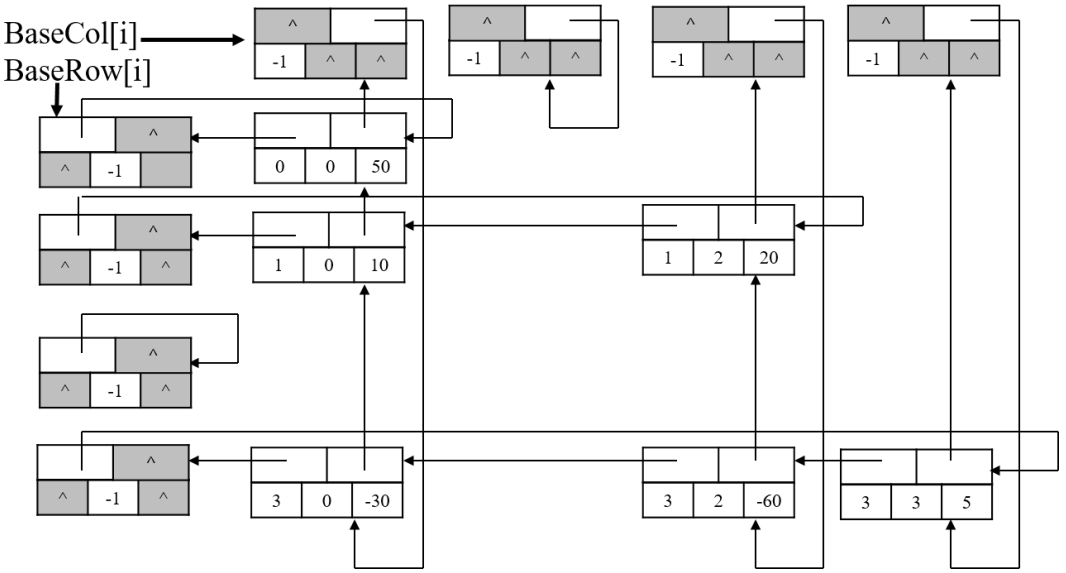
Left		Up
Row	Col	Val

结点类型:

```
strucrt  Node {
        Node  *Left ,  *Up ;
        int   Row ,   Col ;
        ValueType   Val;
      } ;
```

例如:

$$\begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix}$$



2.7 广义表（Lists）

广义表是线性表的一种推广结构，线性表要求是相同的数据元素，而广义表中的元素可以取不同类型，可以是最基本的不可再分割的“原子”，也可以是广义表本身。广义表是由零个原子，或若干个原子或若干个广义表组成的有穷序列。

通常将广义表表示为：

$$A = （a_1，a_2，\cdots，a_n）$$

其中，A 是名称，元素个数 n 是表的长度；若 a_i 不是原子，则称其为A的子表。

注意广义表的递归性。

如：

A = （a，（b，a，b），（），c，（（（2））））；

B = （）；

C = （e）；

D = （A，B，C）；

E = （a，E）；

结论：

① 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；

② 一个广义表可以被其他广义表所共享。

③ 广义表是一个递归的表，即广义表可以是其本身的子表。

操作：

①**Cal（ L ）；** 返回广义表 **L** 的第一个元素

②**Cdr（ L ）；** 返回广义表 **L** 除第一个元素以外的所有元素

③**AppEnd（ L，M ）；** 返回广义表 **L + M**

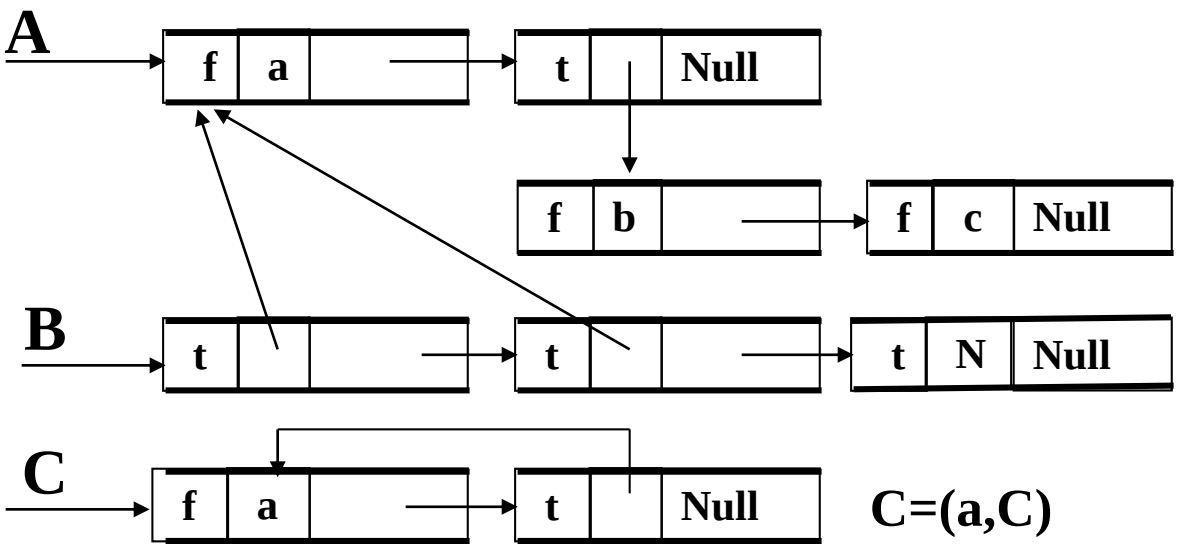
④**Equal（ L，M ）；** 判 广义表 **L** 和 **M** 是否相等

⑤**Length（ L ）；** 求广义表 **L** 的长度

广义表的存储结构

A=(a,(b,c))

B=(A,A,( ))




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

Tag	dlink/data	link

Tag= { FALSE　原子 data
　　 TRUE　 广义表 dlink

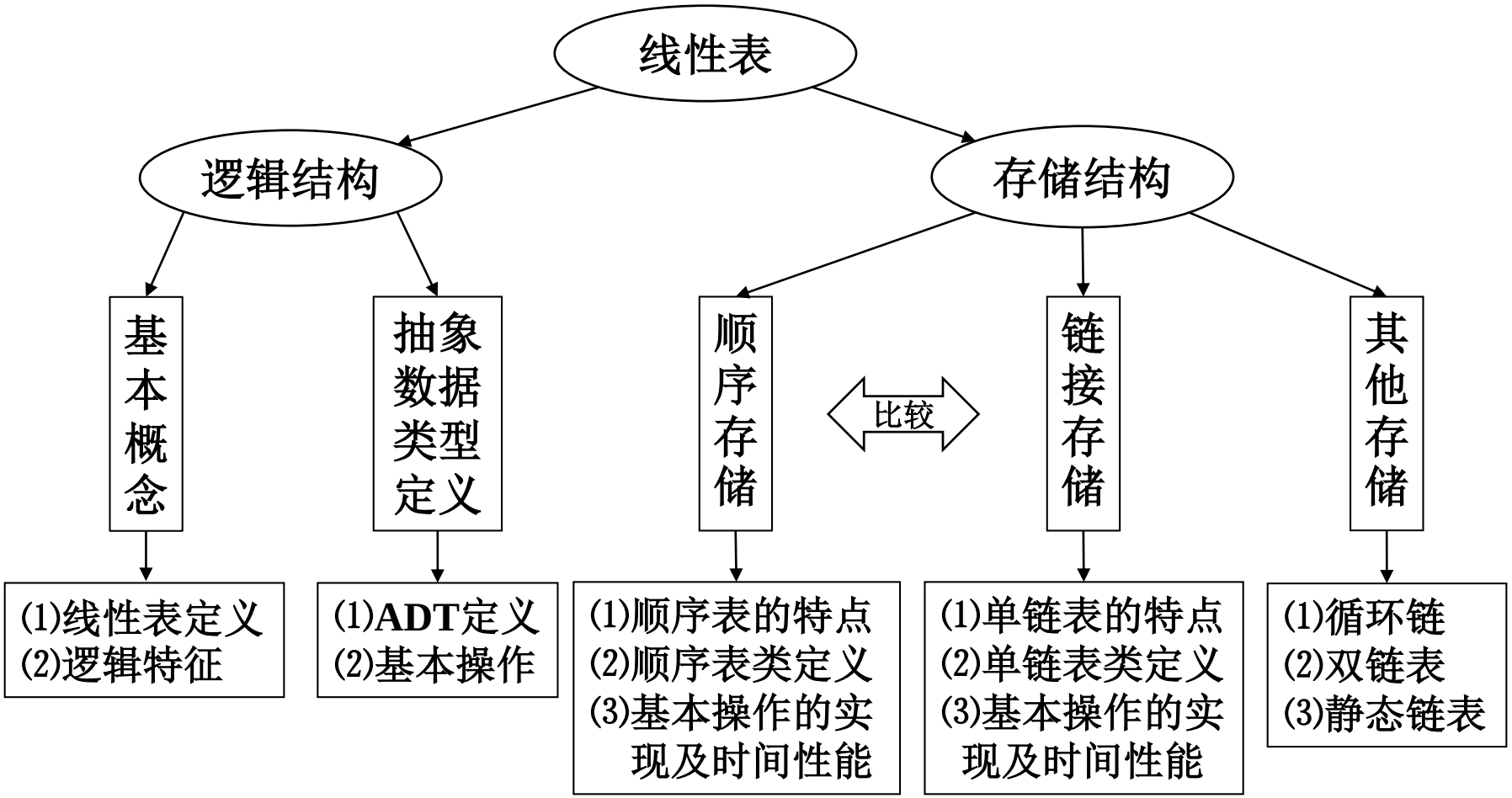
广义表的操作

```
boolean  Equal( listpointer  S, 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 ;
}
```

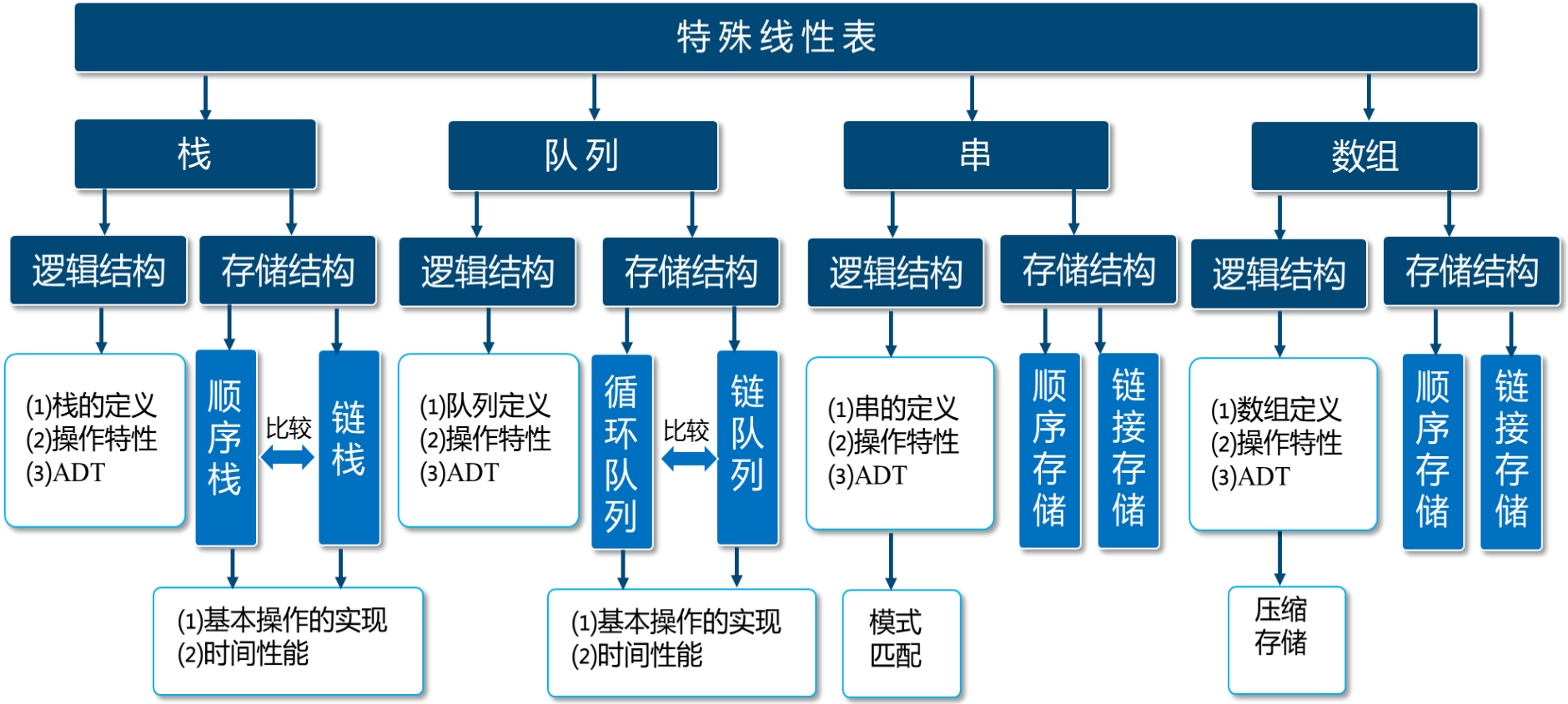
线性表小结

- 线性表的ADT { 数学模型 / 操作集
- 线性表的逻辑结构
  线性表的存储结构 { 顺序存储 / 链式存储 { 单向链表 / 单向循环链表 / 双向链表 / 双向循环链表 } / 游标 }
- 限定性数据结构 { 栈（Stack） / 队（Queue） }
- 串　数组(压缩存储)　广义表
- 顺序存储 与 链式存储 的 比较

知识点总结


线性表
逻辑结构
存储结构
基本概念
抽象数据类型定义
顺序存储
比较
链接存储
其他存储
(1)线性表定义
(2)逻辑特征
(1)ADT定义
(2)基本操作
(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
(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
(1)循环链
(2)双链表
(3)静态链表

线性表小结


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