算法与数据结构

主讲教师：陈 娜
联系方式：chenna@stdu.edu.cn

知识回顾

- 栈的特点、基本操作、存储结构和应用
- 队列的特点、基本操作、存储结构和应用

- 第2章　线性表
- 第3章　栈和队列
- 第4章　串、数组和广义表

线性结构

可表示为：（a_1，　a_2　，　……，　　a_n）

第4章　串、数组和广义表

教学内容

教学目标

1. 了解串的存储方法，理解串的两种模式匹配算法，理解串匹配的KMP算法，熟悉NEXT函数的定义，学会手工计算给定模式串的NEXT函数值和改进的NEXT函数值。
2. 明确数组和广义表这两种数据结构的特点，掌握数组地址计算方法，了解几种特殊矩阵的压缩存储方法。
3. 掌握广义表的定义、性质及其GetHead和GetTail的操作。

4.1 串

串(String)----零个或多个字符组成的有限序列

$$\boxed{s}=''\underline{a_1a_2\cdots a_n}''$$

串名

串值

串长 $\longrightarrow$ n

空串 $\longleftarrow$ n=0

- 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。

如：A=“STUDYING”，B=“DYI”，A为主串，B为子串。

- 位置：字符在序列中的序号。
- 子串在主串中的位置以子串的第一个字符在主串中的位置来表示。
- 串相等的条件：当两个串的长度相等且各个对应位置的字符都相等时才相等。

串的抽象数据类型的定义如下：

ADT String {

数据对象：

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

数据关系：

$$R_1 = \{ < a_{i-1}, a_i > \mid a_{i-1}, a_i \in D,\ i=2,\ldots,n \}$$

基本操作:

StrAssign (&T, chars)

DestroyString(&S)

StrCopy (&T, S)

StrLength(S)

StrCompare (S, T)

Concat (&T, S1, S2)

StrEmpty (S)

SubString (**&**Sub, S, pos, len)

ClearString (**&**S)

Index (S, T, pos)

Replace (**&**S, T, V)

StrInsert (**&**S, pos, T)

StrDelete (**&**S, pos, len)

} ADT String

StrAssign (&T, chars)

初始条件：chars 是字符串常量。

操作结果：生成一个值等于 chars的串 T。

StrCopy (&T, S)

初始条件：串 S 存在。

操作结果：由串 S 复制得串 T。

DestroyString (&S)

初始条件：串 S 存在。

操作结果：串 S 被销毁。

StrEmpty(S)

初始条件： 串S存在。

操作结果： 若 S 为空串，则返回TRUE，否则返回 FALSE。

“” 表示空串，空串的长度为零。

StrCompare (S, T)

初始条件：串 S 和 T 存在。

操作结果：若S > T，则返回值 > 0;

若S = T，则返回值 = 0;

若S < T，则返回值 < 0。

例如：StrCompare("data", "state") < 0

StrCompare("cat", "case") > 0

StrLength(S)

初始条件：串S存在。

操作结果：返回S的元素个数，
称为串的长度。

Concat(&T,S1,S2)

初始条件：串 S1 和 S2 存在。

操作结果：用 T 返回由 S1 和 S2 联接而成的新串。

例如：Concate(T, “man”, “kind”)

求得 T = “mankind”

SubString (&Sub, S, pos, len)

初始条件：

串 S 存在，$1 \le pos \le StrLength(S)$

且$0 \le len \le StrLength(S)-pos+1$。

操作结果：

用 Sub 返回串 S 的第 pos 个字符起长度为 len 的子串。

子串为“串”中的一个字符子序列

例如：

SubString(sub, “commander”, 4, 3)

求得 sub = “man” ;

SubString(sub, “commander”, 1, 9)

求得 sub = “commander”;

SubString(sub, “commander”, 9, 1)

求得 sub = “r”;

Index(S,T,pos)

初始条件：串S和T存在，T是非空串，1≤pos≤StrLength(S)。

操作结果：若主串 S 中存在和串 T 值相同的子串, 则返回它在主串 S 中第pos个字符之后第一次出现的位置; 否则函数值为0。

“**子串在主串中的位置**”意指子串中的第一个字符在主串中的**位序**。

假设 S = “abcaabcaaabc”， T = “bca”

Index(S, T, **1**) = 2;

Index(S, T, **3**) = 6;

Index(S, T, **8**) = 0;

Replace(&S,T,V)

初始条件：串S, T和 V 均已存在，且T 是非空串。

操作结果：用V替换主串S中出现的所有与（模式串）T相等的不重叠的子串。

StrInsert (&S, pos, T)

初始条件： 串S和T存在，

$1 \le pos \le StrLength(S) + 1$。

操作结果： 在串S的第pos个字符之前插入串T。

例如： S = “chater”， T = “rac”，

则执行 StrInsert(S, 4, T)之后得到

S = “character”

ClearString(&S)

初始条件：串S存在。

操作结果：将S清为空串。

StrDelete (&S, pos, len)

初始条件：串S存在

$1 \le pos \le StrLength(S)-len+1$。

操作结果：从串S中删除第pos个字符起长度为len的子串。

串的存储结构

- 顺序存储
- 链式存储

一、串的定长顺序存储表示

```
#define  MAXSTRLEN  255
     // 用户可在255以内定义最大串长
typedef unsigned char  Sstring[MAXSTRLEN + 1];
       // 0号单元存放串的长度
```

特点:

- **串**的实际长度可在这个予定义长度的范围内随意设定，超过予定义长度的串值则被舍去，称之为“**截断**”。
- 按这种串的表示方法实现的串的运算时，其基本操作为“**字符序列的复制**”。

二、串的堆分配存储表示

```
typedef struct {
  char *ch;
    // 若是非空串，则按串长分配存储区，
    //  否则ch为NULL
  int  length;   // 串长度
} HString;
```

链式存储表示

链式存储表示


```
#define CHUNKSIZE 80       //可由用户定义的块大小
typedef struct Chunk{
   char  ch[CHUNKSIZE];
   struct Chunk *next;
}Chunk;

typedef struct{
   Chunk *head,*tail;      //串的头指针和尾指针
   int curlen;             //串的当前长度
}LString;
```

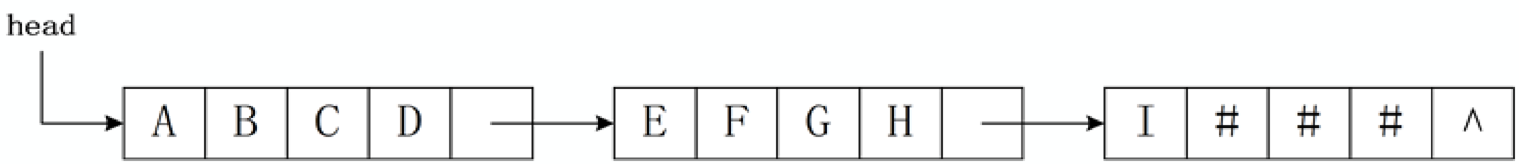
链式存储表示

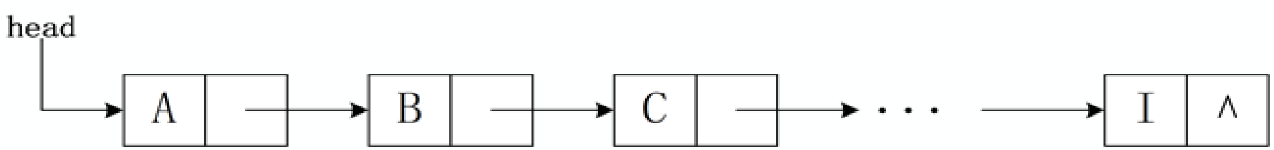
优点：操作方便

缺点：存储密度较低

$$\text{存储密度} = \frac{\text{串值所占的存储位}}{\text{实际分配的存储位}}$$

可将多个字符存放在一个结点中，以克服其缺点





- 串多采用顺序存储结构

串的模式匹配算法

算法目的：

确定主串中所含子串第一次出现的位置（定位）
即如何实现Index(S,T,pos)函数

算法种类：

- BF算法（又称古典的、经典的、朴素的、穷举的）
- KMP算法（特点：速度快）

BF算法设计思想

i

S : a b a b c a b c a c b a b

T : a b c

j

i指针回溯

S : a b a b c a b c a c b a b

T : a b c

S : a b a b c a b c a c b a b

T :   a b c

BF算法设计思想

Index(S,T,pos)

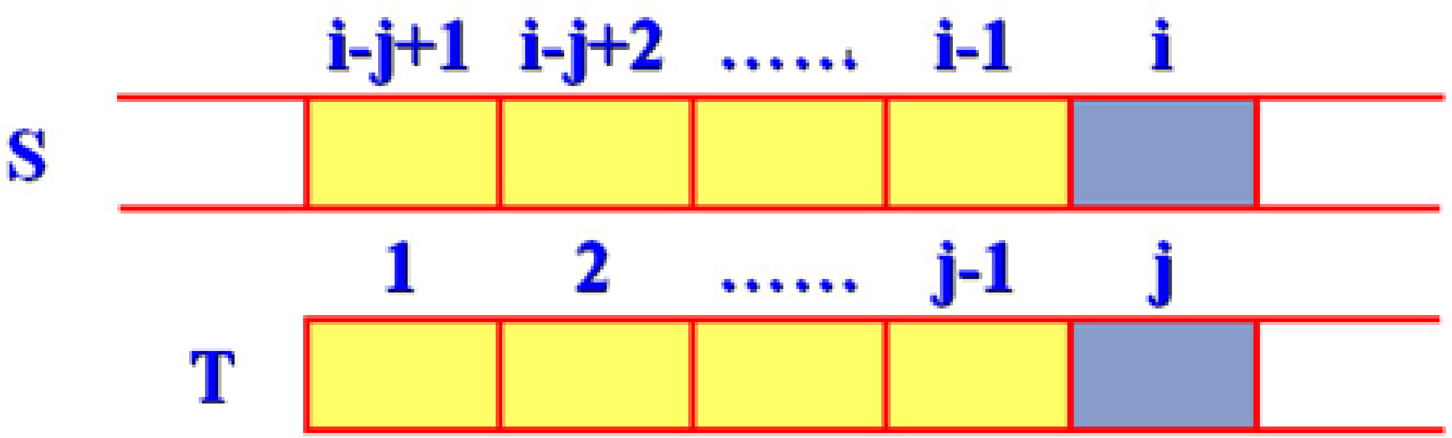
- 将主串的第pos个字符和模式的第一个字符比较，
  若相等，继续逐个比较后续字符；
  若不等，从主串的下一字符起，重新与模式的第一个字符比较。

- 直到主串的一个连续子串字符序列与模式相等 。返回值为S中与T匹配的子序列第一个字符的序号，即匹配成功。

- 否则，匹配失败，返回值 0

BF算法描述（算法4.1）


```
int  Index(Sstring S,Sstring T,int pos){
    i=pos;   j=1;
   while (i<=S[ 0 ] && j <=T[ 0 ]){
      if ( S[ i ]=T[ j ]) {++i;  ++j; }
     else{ i=i-j+2;    j=1; }
   if ( j>T[ 0 ])   return   i－T[0];
   else return 0;
}
```



BF算法时间复杂度

例： S=‘0000000001’，T=‘0001’，pos=1

若n为主串长度，m为子串长度，最坏情况是

✓主串前面n-m个位置都部分匹配到子串的最后一位，即这n-m位各比较了m次

✓最后m位也各比较了1次

总次数为：(n-m)*m+m＝(n-m+1)*m

若m<<n，则算法复杂度O(n*m)

当模式串为’00000001’,主串为’000000000000000000000000000000000000000000000000001’时,整个匹配过程中指针i需回溯45次.

模式匹配算法二：KMP算法

此改进算法是D. E. Knuth与V. R. Pratt和J. H. Morris同时发现的，因此人们称它为克努特-莫里斯-普拉特操作（简称KMP算法）

1、KMP算法设计思想：

尽量利用已经部分匹配的结果信息；尽量让i不要回溯，加快模式串的滑动速度。

例：

S=‘a b a b c a b c a c b a b’
T=‘a b c a c’

S=‘a b a b c a b c a c b a b’
T=‘a b c a c’

S=‘a b a b c a b c a c b a b’
T=‘a b c a c’

需要讨论两个问题：

①如何由当前部分匹配结果确定模式向右滑动的新比较起点k？

② 模式应该向右滑多远才是高效率的?

k是追求的新起点

请抓住部分匹配时的两个特征：

(1)

i

S='a b a b c a b c a c b a b'

T='a b c a c'

k

设目前打算与T的第k字符开始比较

则T的k-1位＝S前i-1～i-k+1)位

'T_1 … T_{k-1}'

'S_{i-k+1} … S_{i-1}'

(2)

i

S='a b a b c a b c a c b a b'刚才肯定是在S的i处和T的第j字符 处失配

T='a b c a c'

k　　j

则T的j-1～j-k+1位＝ S前i-1～i-k+1)位

'T_{j-k+1} …T_{j-1}'　截取一段，但k有限制，0<k<j

两式联立可得："T_1…T_{k-1}"='T_{j-k+1} …T_{j-1}'

注意：j 为当前已知的失配位置，我们的目标是计算新起点 k。
式中仅剩一个未知数k，理论上已可解！

奇妙的结果：k 仅与模式串T有关！

结论：

设主串为'$s_1s_2...s_n$'，模式串为'$p_1p_2...p_m$'，当 S[i] <> T[j] 时，主串中第i个字符与模式中第j个字符“失配”时，仅需从模式中第k个字符和主串第i个字符比较起继续进行。

定义：模式串的*next*函数

若令next[j]=k, 则next[j]表明模式中第j个字符比较不等时，需将模式向右滑动至模式中第k个字符和主串中第i个字符对齐，继续比较。

$$
Next[j]=\begin{cases} 0 & \text{当}j=1\text{时} \\ Max\{k \mid 1<k<j \text{ 且 } `p_1\ldots p_{k-1}\text{'}=`p_{j-k+1}\ldots p_{j-1}\text{'}\} & \text{当此集合不空时} \\ 1 & \text{其他情况} \end{cases}
$$

取P首与Pj处最大的相同子串

注意：

（1）k值仅取决于模式串本身而与相匹配的主串无关。

（2）k值为模式串从头向后及从j向前的两部分的最大相同子串的长度。

（3）这里的两部分子串可以有部分重叠的字符，但不可以全部重叠。

可见，模式中相似部分越多，则next[j]函数越大，它既表示模式T字符之间的相关度越高，也表示j位置以前与主串部分匹配的字符数越多。

即：next[j]越大，模式串向右滑动得越远，与主串进行比较的次数越少，时间复杂度就越低（时间效率）。

匹配过程

假设以指针i和指针j分别指示主串s和模式串p中正待比较的字符，令i的初值为pos，j的初值为1。若在匹配的过程中，$s_i=p_j$，则，i，j分别加1；否则，j退回到某个next值位置，以此类推。直到下列两种可能：

- j退回到某个next值时字符相等，则指针分别加1，继续匹配。
- j=0（即模式的第一个字符失配），则从主串的下一个字符（i+1）起和模式重新匹配。

```
int Index_KMP(SString S, SString T, int pos) {
   // 1≤pos≤StrLength(S)
   i = pos;   j = 1;
   while (i <= S[0] && j <= T[0]) {
      if (j == 0 || S[i] == T[j]) { ++i;  ++j; }
                                        // 继续比较后继字符
     else  j = next[j];           // 模式串向右移动
   }
  if (j > T[0])  return  i-T[0];    // 匹配成功
  else return 0;
} // Index_KMP
```

问 题

- 如何求 next[j]?

求*next*函数值的过程是一个递推过程，分析如下：

已知：next[1] = 0；

设next[j] = k,表明模式串中存在下列关系

‘$p_1..p_{k-1}$’=‘$p_{j-k+1}\cdots p_{j-1}$’，其中k为满足1<k<j的某个值

1．若$p_k=p_j$ 则next[j+1] = k+1即next[j+1]=next[j]+1 (前k个都相同)

主串：……. acbcdacbcf……

1…k-1 j-k+1..j-1

acbcdacbcg

next[j] 0 1 11 1 123 4 5

next[j+1]=k+1=5

2. 若$p_k \neq p_j$ 则表明'$p_1..p_k$' ≠'$p_{j-k+1}...p_j$' 若next[k]=k'
且$p_j = p_{k'}$,则存在'$p_1..p_{k'}$' ＝'$p_{j-k'+1}...p_j$'
则next[j+1]=k'+1即next[j+1]=next[k]+1（前k'个相同）

主串：…… acbddacbaf……

1…. k-1　　j-k+1..j-1

acbddacbag

next[j] 0111 1 1 2 3 4 2

因为k'=1; next[j+1]=next[k]+1=2

3. 若$p_k \neq p_j$ 且$p_j \neq p_{k'}$ next[j+1]=1（没有相同的）

这实际上也是一个匹配的过程，不同在于：主串和模式串是同一个串

主串：…… acbcdacbdf……

1…. k-1　　j-k+1..j-1

acbcdacbdg

next[j] 0111 1 1 2 3 4 1

next[j+1]=1

```
void get_next(SString &T, int &next[] ) {
  // 求模式串T的next函数值并存入数组next
  i = 1;   next[1] = 0;   j = 0;
  while (i < T[0]) {
      if (j == 0 || T[i] == T[j])
            {++i;  ++j; next[i] = j; }
      else  j = next[j];
  }
} // get_next
```

例:

模式串='abcaabbabcabaacbacba'

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	a	b	c	a	a	b	b	a	b	c	a	b	a	a	c	b	a	c	b	a
next[j]	0	1	1	1	2	2	3	1	2	3	4	5	3	2	2	1	1	2	1	1

```
i = 1;   next[1] = 0;   j = 0;
    while (i < T[0]) {
        if (j = 0 || T[i] == T[j])
            {++i;  ++j; next[i] = j; }
        else  j = next[j];
```

知识回顾

- 串的定义
- 模式匹配算法
  - KF算法
  - KMP算法

```
int Index_KMP(SString S, SString T, int pos) {
   //  1≤pos≤StrLength(S)
   i = pos;   j = 1;
   while (i <= S[0] && j <= T[0]) {
      if (j == 0 || S[i] == T[j]) { ++i;  ++j; }
                                          // 继续比较后继字符
     else  j = next[j];          // 模式串向右移动
   }
  if (j > T[0])  return  i-T[0];    // 匹配成功
  else return 0;
} // Index_KMP
```

```
void get_next(SString &T, int &next[] ) {
  // 求模式串T的next函数值并存入数组next
  i = 1;   next[1] = 0;   j = 0;
  while (i < T[0]) {
      if (j == 0 || T[i] == T[j])
          {++i;  ++j; next[i] = j; }
      else  j = next[j];
  }
} // get_next
```

例：

模式串=‘abcaabbabcabaacbacba’

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	a	b	c	a	a	b	b	a	b	c	a	b	a	a	c	b	a	c	b	a
next[j]	0	1	1	1	2	2	3	1	2	3	4	5	3	2	2	1	1	2	1	1

```
i = 1;   next[1] = 0;   j = 0;
    while (i < T[0]) {
        if (j = 0 || T[i] == T[j])
            {++i;  ++j; next[i] = j; }
        else  j = next[j];
```

还有一种特殊情况需要考虑：

例如： http://v.youku.com/v_show/id_XMzc1MTc1MjA=.html

S = 'aaabaaaabaaaabaaaab'

T = 'aaaab'

原因:

next[j]=01234

$p_k=p_j$ 都是'a'

j=4,next[4]=3

nextval[j]=00004

k=next[j]

```
void get_nextval(SString &T, int &nextval[]) {
  i = 1;   nextval[1] = 0;   j = 0;
  while (i < T[0]) {
    if (j = 0 || T[i] == T[j]) {
      ++i;  ++j;
      if (T[i] != T[j])  nextval[i] = j;
      else  nextval[i] = nextval[j];
    }
    else  j = nextval[j];
  }
} // get_nextval
```

例：u=‘abcaabbabcabaacbacba’

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	a	b	c	a	a	b	b	a	b	c	a	b	a	a	c	b	a	c	b	a
next[j]	0	1	1	1	2	2	3	1	2	3	4	5	3	2	2	1	1	2	1	1
nextval[j]	0	1	1	0	2	1	3	0	1	1	0	5	3	2	2	1	0	2	1	0

```
i = 1;   nextval[1] = 0;   j = 0;
    while (i < T[0]) {
        if (j = 0 || T[i] == T[j]) {
            ++i;  ++j;
            if (T[i] != T[j])  nextval[i] = j;
            else  nextval[i] = nextval[j];
        }
        else  j = nextval[j];
    }
```

练习:

s='ADBADABBAABADABBADADA'

pat='ADABBADADA'

	1	2	3	4	5	6	7	8	9	10
	A	D	A	B	B	A	D	A	D	A
next[j]	0	1	1	2	1	1	2	3	4	3
nextval[j]	0	1	0	2	1	0	1	0	4	0

2、KMP算法的时间复杂度

回顾BF的最恶劣情况：S与T之间存在大量的**部分匹配**，比较总次数为：**$(n-m+1)*m=O(n*m)$**

而此时KMP的情况是：由于指针i无须回溯，比较次数仅为n，即使加上计算next[j]时所用的比较次数m，比较总次数也仅为$n+m=O(n+m)$，大大快于BF算法。

> **注意：由于BF算法在一般情况下的时间复杂度也近似于O(n+m)，所以至今仍被广泛采用。**

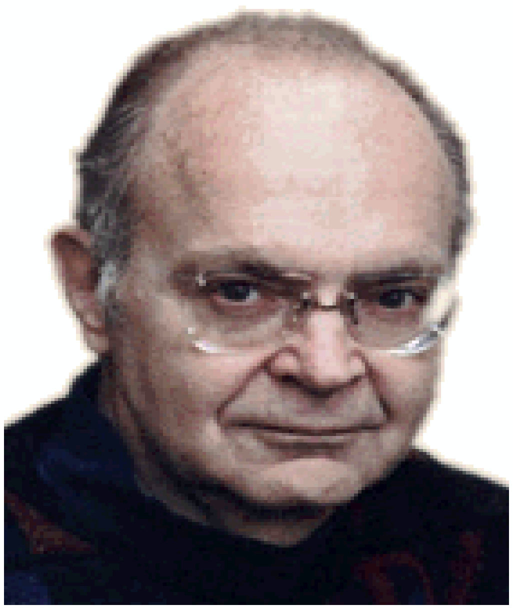
KMP算法的用途：

因为主串指针i不必回溯，所以从外存输入文件时可以做到边读入边查找——“流水作业”！

KMP（Knuth Morris Pratt）算法

《计算机程序设计艺术 第1卷 基本算法》 98元
《计算机程序设计艺术 第2卷 半数值算法》 98元
《计算机程序设计艺术 第3卷 排序与查找》 98元

http://www-cs-faculty.stanford.edu/~knuth/



4.2 数组

数组可以看成是一种特殊的线性表，即线性表中数据元素本身也是一个线性表

数组的抽象数据类型

ADT Array {

数据对象:

$$j_i = 0, \cdots b_i - 1, \mathrm{i} = 1, 2, \cdots, \mathrm{n}$$

$$D = \{\mathrm{a}_{\mathrm{j}\,\mathrm{j}_2 \cdots \mathrm{j}_\mathrm{n}} \mid \mathrm{a}_{\mathrm{j}_1 \mathrm{j}_2 \cdots \mathrm{j}_\mathrm{n}} \in ElemSet$$

数据关系:

基本操作:

(1) InitArray (&A,n,bound1, …boundn)
//构造数组A

(2) DestroyArray (&A) // 销毁数组A

(3) Value(A,&e,index1,…,indexn) //取数组元素值

(4) Assign (A,&e,index1,…,indexn) //给数组元素赋值

}ADT Array

数组的顺序表示和实现

类型特点:

1) 只有引用型操作，没有加工型操作;

2) 数组是多维的结构，而存储空间是一个一维的结构。

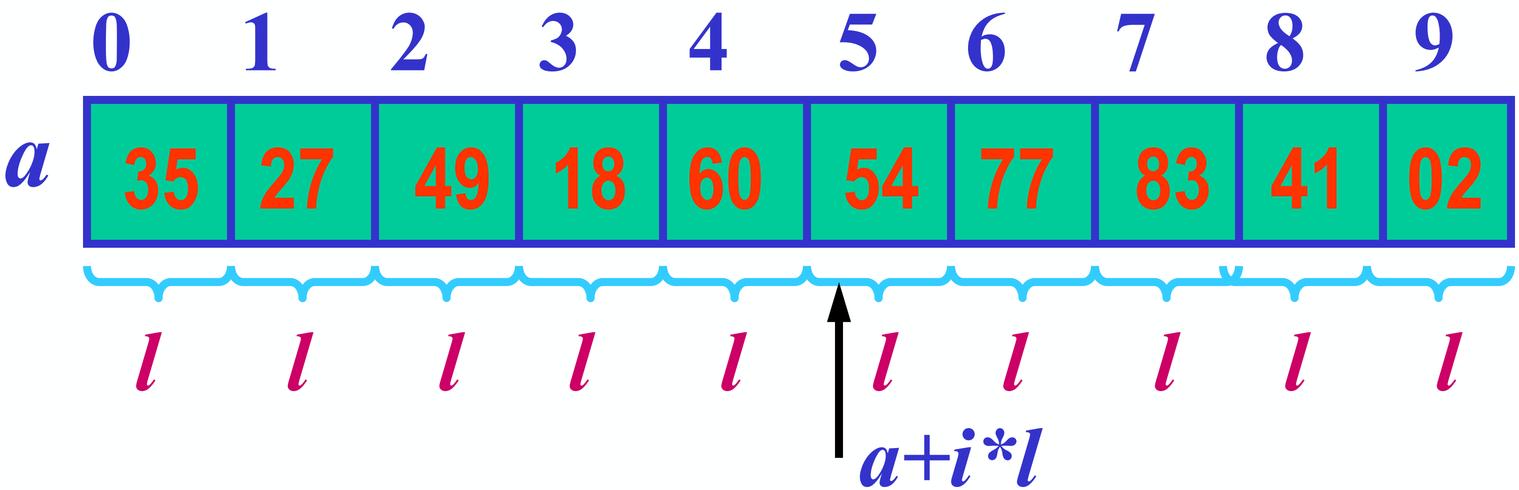
有两种顺序映象的方式:

1)以行序为主序(低下标优先);

2)以列序为主序(高下标优先)。

一维数组

$$\mathrm{LOC}(i) = \begin{cases} a, & i = 0 \\ \mathrm{LOC}(i-1)+l = a+i*l, & i > 0 \end{cases}$$



$$\mathrm{LOC}(i) = \mathrm{LOC}(i-1)+l = a+i*l$$

二维数组

$$A=(\alpha_1,\alpha_2,\cdots,\alpha_p)\quad(\mathrm{p}=\mathrm{m}或\mathrm{n})$$

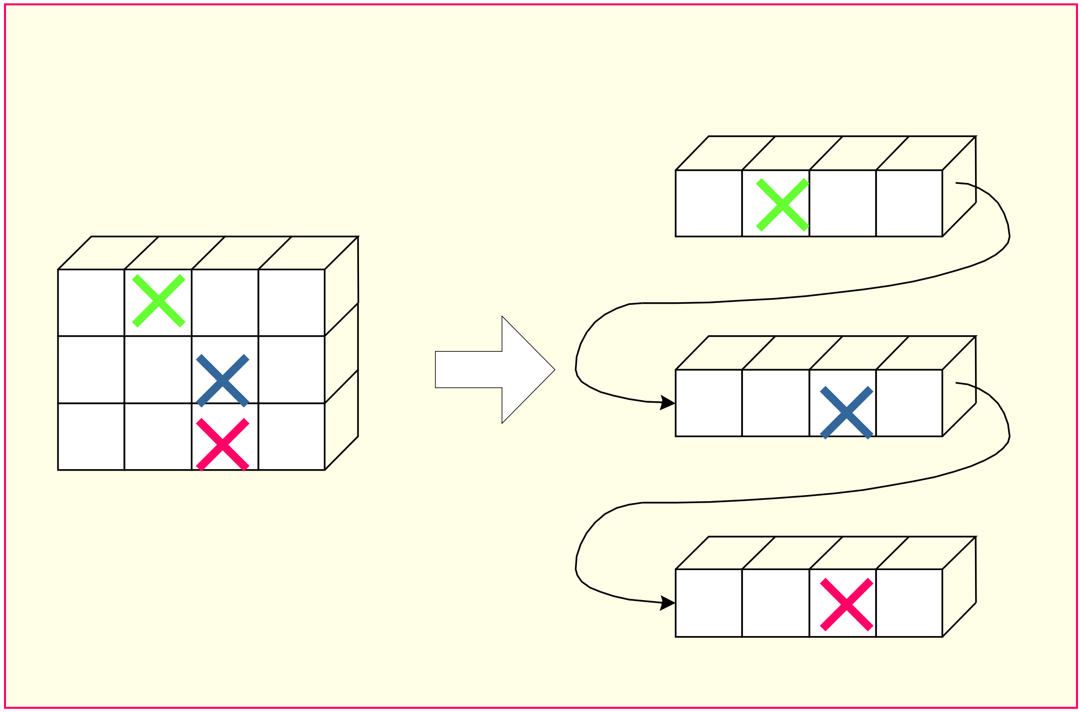
$$\alpha_i=(a_{i1},a_{i2},\cdots,a_{in})\quad 1\le i\le m$$

$$\alpha_j=(a_{1j},a_{2j},\cdots,a_{mj})\quad 1\le \mathrm{j}\le n$$

数组的顺序存储

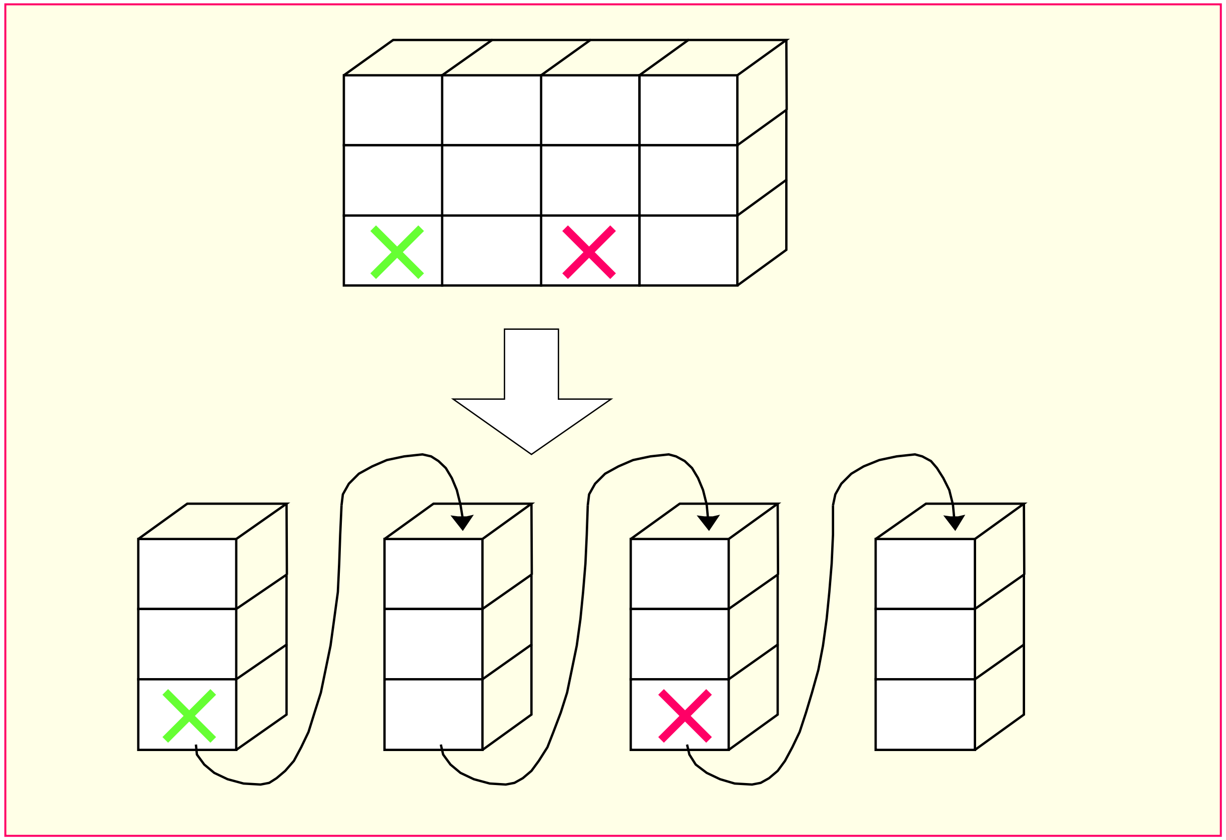
- 以行序为主序

  C, PASCAL



- 以列序为主序

  FORTRAN



二维数组的行序优先表示

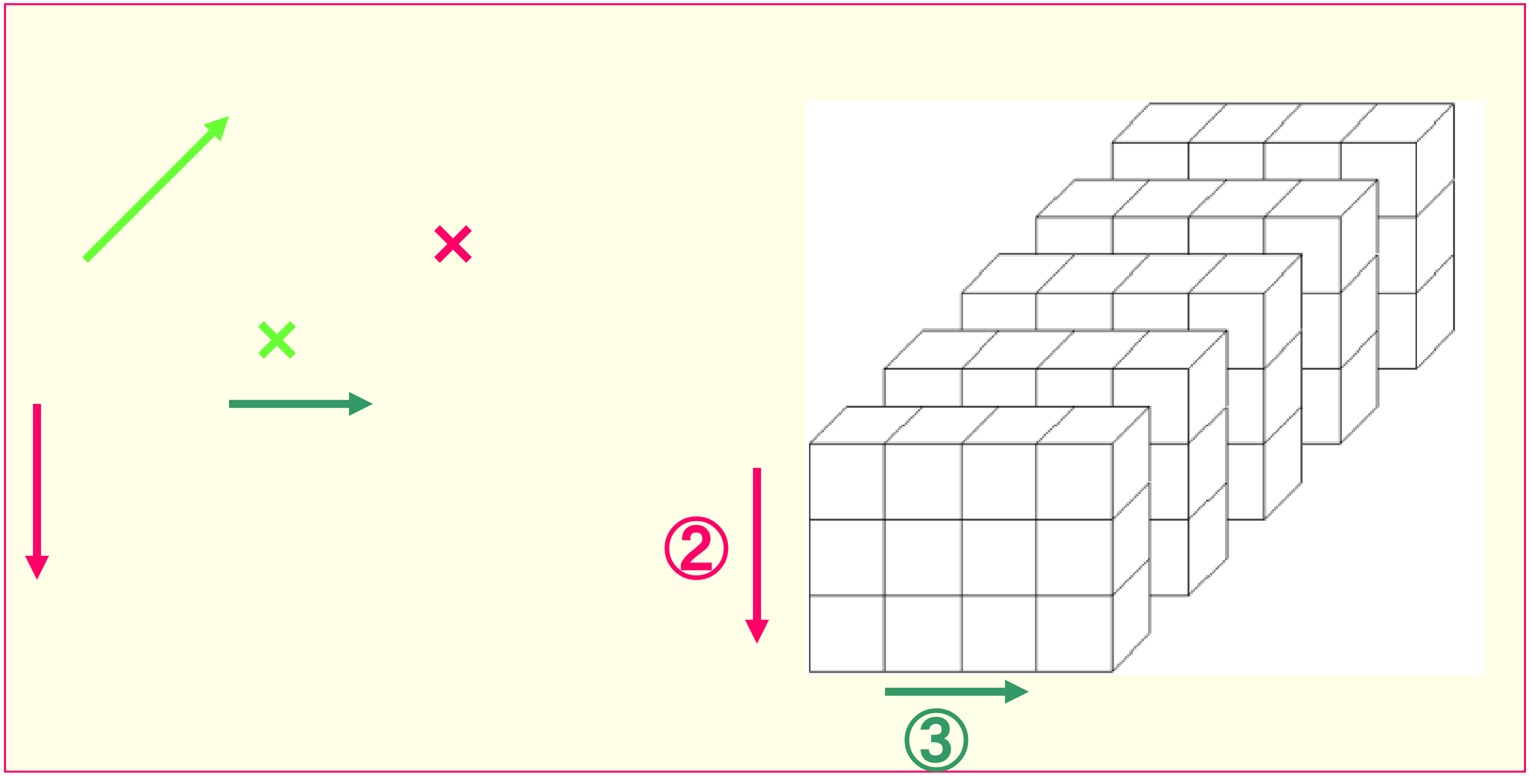
a[n][m]

设数组开始存放位置 LOC(0, 0) = a

$LOC(j, k) = a + j * m + k$

三维数组

按页/行/列存放，页优先的顺序存储



三维数组

- a[m1][m2] [m3] 各维元素个数为 m_1, m_2, m_3
- 下标为 i_1, i_2, i_3的数组元素的存储位置：

$$LOC(i_1, i_2, i_3) = a + \underbrace{i_1 * m_2 * m_3}_{\text{前}i_1\text{页总元素个数}} + \underbrace{i_2 * m_3}_{\text{第}i_1\text{页的前}i_2\text{行总元素个数}} + i_3$$

第 i_2 行前 i_3 列元素个数

n维数组

- ☞ 各维元素个数为 $m_1, m_2, m_3, \ldots, m_n$
- ☞ 下标为 $i_1, i_2, i_3, \ldots, i_n$ 的数组元素的存储位置：

n维数组

$$c_n = L, c_{i-1} = b_i \times c_i,\ \mathbf{1 < i \leq n}$$

练习

设有一个二维数组$A[m][n]$按行优先顺序存储，假设$A[0][0]$存放位置在$644_{(10)}$，$A[2][2]$存放位置在$676_{(10)}$，每个元素占一个空间，问$A[3][3]_{(10)}$存放在什么位置？脚注$_{(10)}$表示用10进制表示。

设数组元素A[i][j]存放在起始地址为Loc (i, j) 的存储单元中

$\because$ Loc (2, 2) = Loc (0, 0) + 2 * n + 2 = 644 + 2 * n + 2 = 676.
$\therefore$ n = (676 - 2 - 644) / 2 = 15
$\therefore$ Loc (3, 3) = Loc (0, 0) + 3 * 15 + 3 = 644 + 45 + 3 = 692.

练习

设有二维数组A[10,20]，其每个元素占两个字节，*A*[0][0]存储地址为100，若按行优先顺序存储，则元素A[6,6]的存储地址为________ 352 按列优先顺序存储，元素A[6,6]的存储地址为__________ 232

$(6*20+6)*2+100=352$

$(6*10+6)*2+100=232$

特殊矩阵的压缩存储

1. 什么是压缩存储?

若多个数据元素的值都相同，则只分配一个元素值的存储空间，且零元素不占存储空间。

2. 什么样的矩阵能够压缩?

一些特殊矩阵，如：对称矩阵，对角矩阵，三角矩阵，稀疏矩阵等。

3. 什么叫稀疏矩阵?

矩阵中非零元素的个数较少（一般小于5%）

n阶对称矩阵： $a_{ij}=a_{ji}$ $1 \le i,j \le n$

$$
\begin{bmatrix}
a_{11} & a_{12} & \cdots & \cdots\cdots & a_{1n} \\
a_{21} & a_{22} & \cdots\cdots & \cdots\cdots & a_{2n} \\
 & \cdots\cdots & \cdots\cdots & \cdots\cdots & \\
a_{n1} & a_{n2} & \cdots\cdots & & a_{nn}
\end{bmatrix}
$$

对称矩阵的压缩存储

以行序为主序

a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	……	a_{n1}	……	a_{nn}
k=0	1	2	3	4		n(n-1)/2		n(n+1)/2-1

问题：

假设以一维数组sa[n(n+1)/2]**作**为n阶对称阵的存储结构，则sa[k]**和矩**阵元素aij**存在怎**样的对应关系？

1.以行序为主序存储下三角中的元

2.以列序为主序存储上三角中的元

下三角

上三角

下（上）三角矩阵：

矩阵的上（下）三角中的元均为常数c或零的n阶矩阵。

$$
\begin{bmatrix}
a_{11} & 0 & 0 \cdots\cdots & 0 \\
a_{21} & a_{22} & 0 \cdots\cdots & 0 \\
\cdots & \cdots & \cdots\cdots & 0 \\
a_{n1} & a_{n2} & a_{n3} \cdots\cdots & a_{nn}
\end{bmatrix}
$$

按行序为主序：

a11	a21	a22	a31	a32	……	an1	……	ann
k=0	1	2	3	4		n(n-1)/2		n(n+1)/2-1

$$Loc(a_{ij}) = Loc(a_{11}) + [\frac{i(i-1)}{2} + (j-1)]$$ 设L为1

前面i-1行元素个数

第i行元素个数

稀疏矩阵

稀疏矩阵： 非零元较零元少,且分布没有一定规律

假设 m 行 n 列的矩阵含 t 个非零元素，

令 $\delta = \dfrac{t}{m \times n}$

则称 δ 为**稀疏因子**。

通常认为 $\delta \le 0.05$ 的矩阵为稀疏矩阵。

以常规方法，即以二维数组表示高阶的稀疏矩阵时产生的**问题:**

1) 零值元素占了很大空间;

2) 计算中进行了很多和零值的运算，遇除法，还需判别除数是否为零。

解决问题的原则:

1) 尽可能少存或不存零值元素;

2) 尽可能减少没有实际意义的运算;

3) 操作方便。即:

能尽可能快地找到与下标值(i，j)对应的元素，

能尽可能快地找到同一行或同一列的非零值元。

– 稀疏矩阵

- 压缩存储原则：只存矩阵的行列维数和每个非零元的行列下标及其值

M由{(1,2,12), (1,3,9), (3,1,-3), (3,6,14), (4,3,24), (5,2,18), (6,1,15), (6,4,-7) } 和矩阵维数（6,7）唯一确定

稀疏矩阵的压缩存储方法:

一、三元组顺序表

二、行逻辑链接的顺序表

三、 十字链表

一、三元组顺序表

```
#define  MAXSIZE  12500
typedef struct {
    int  i, j;       //该非零元的行下标和列下标
    ElemType  e;     // 该非零元的值
} Triple;  // 三元组类型
typedef union {
     Triple  data[MAXSIZE + 1];
     int      mu, nu, tu; // 矩阵行数、列数和非零元个数
} TSMatrix;  // 稀疏矩阵类型
```

稀疏矩阵的压缩存储方法
——三元组顺序表

行列下标

非零元值

m.data[0] 分别存放矩阵行数，列数和非零元个数

	i	j	v
0	6	7	8
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

M.data

如何求转置矩阵？

$$\begin{bmatrix} 0 & 0 & 36 \\ 14 & -7 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 28 \\ -5 & 0 & 0 \end{bmatrix}$$

a　　　　　　　　b

用常规的二维数组表示时的算法

```
for (col=1; col<=nu; ++col)
    for (row=1; row<=mu; ++row)
        T[col][row] = M[row][col];
```

其时间复杂度为: O(mu×nu)

用“三元组”表示时如何实现？

1	2	14
1	5	-5
2	2	-7
3	1	36
3	4	28

1	3	36
2	1	14
2	2	-7
4	3	28
5	1	-5

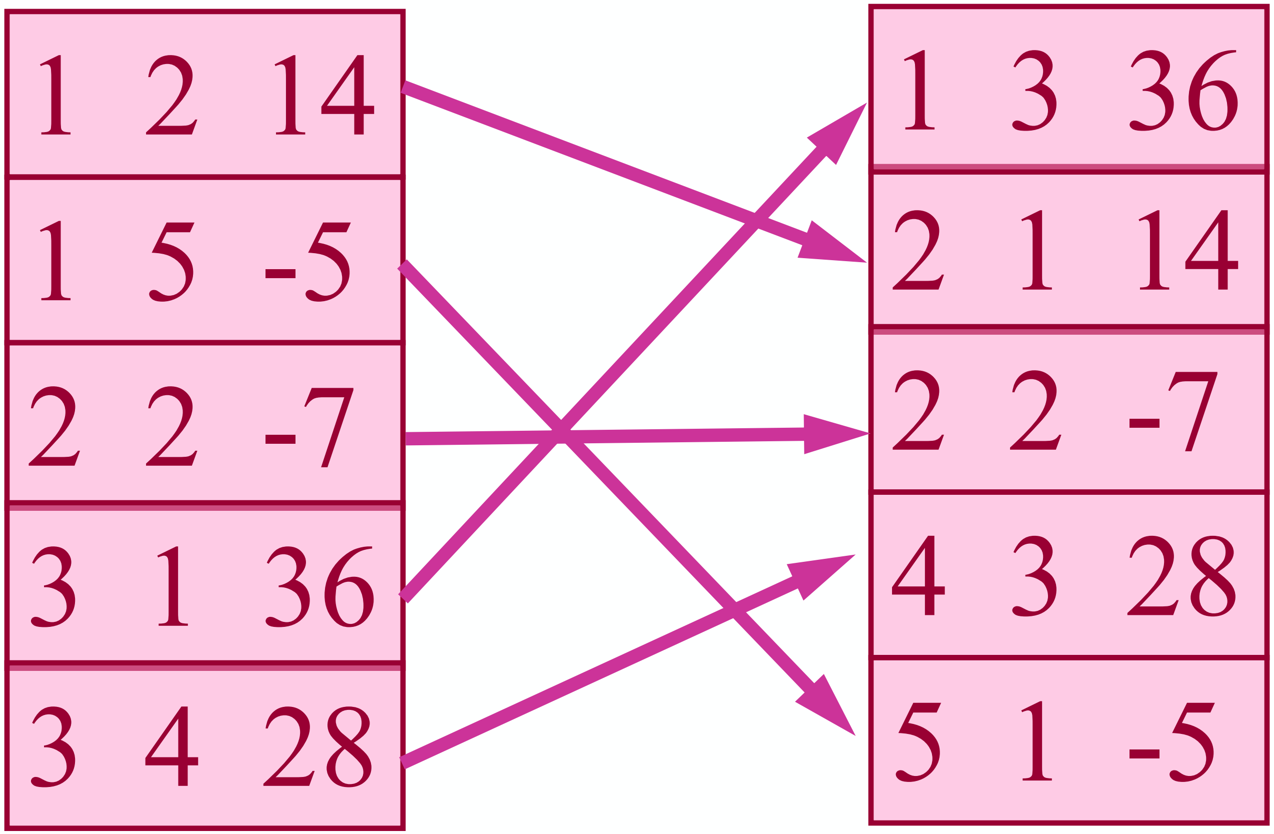
- 矩阵的行列值互换
- 下标i，j互换
- 重排次序

方法1：

- 按照b.data中三元组的次序依次在a.data中找到相应的三元组进行转置。

  即：按照矩阵a的列序来进行转置。

方法1：



方法2：

如果能预先确定矩阵M中每一列（即T中每一行）的第一个非零元在三元组中的位置，则转置时，便可直接放到恰当位置。

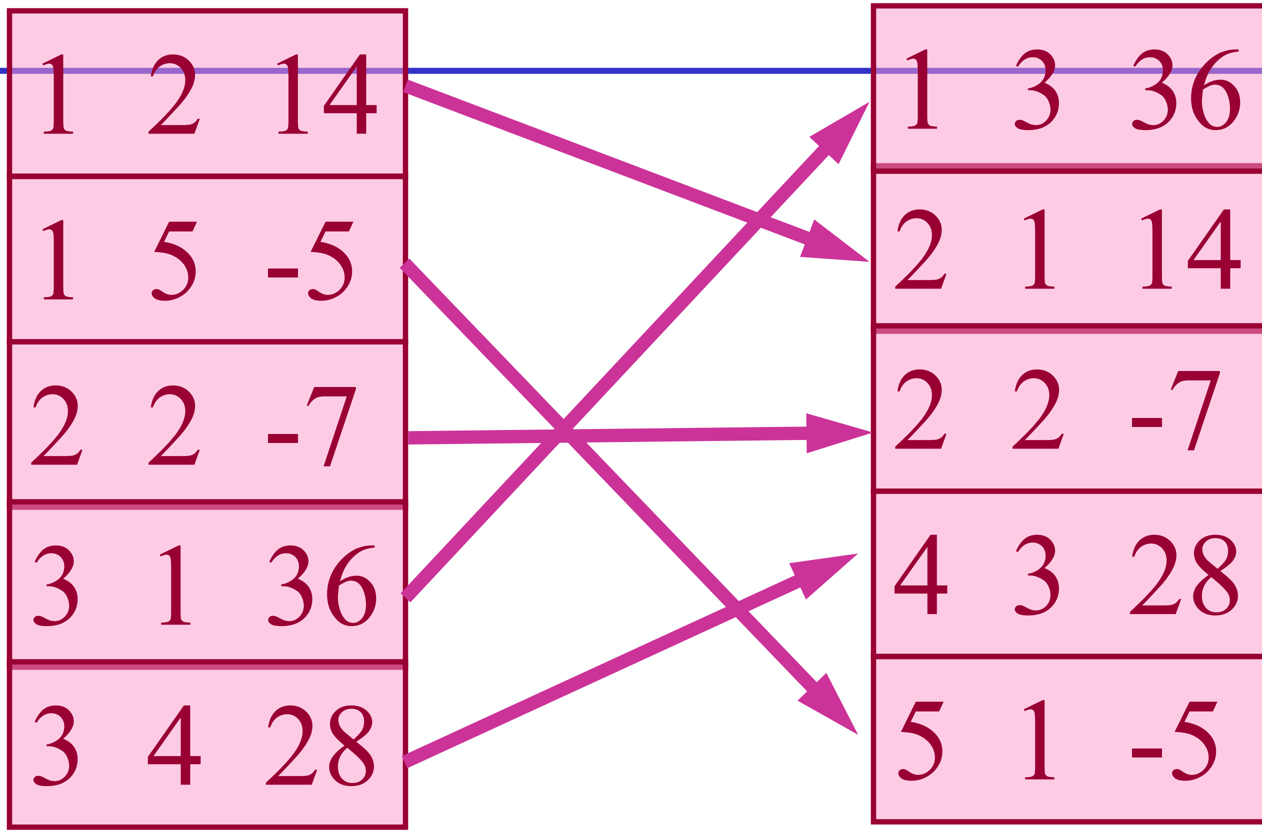
转置前，应先求得M的每一列中非零元个数，进而求得每一列第一个非零元在转置阵T.data中应有的位置。

附设两个向量:num和cpot.

- num[col]：表示矩阵M中第col列中非零元的个数
- cpot[col]：指示M中第col列的第一个非零元在T.data中的恰当位置,显然有

$$cpot[1] = 1$$

$$cpot[col] = cpot[col-1] + num[col-1]$$



col	1	2	3	4	5
Num[col]	1	2	0	1	1
Cpot[col]	**1**	**2**	**4**	**4**	**5**

矩阵的向量cpot的值

三元组顺序表(有序的双下标法)特点:

非零元在表中按行序有序存储，因此便于进行依行顺序处理的矩阵运算。然而，若需按行号存取某一行中的非零元，则需从头开始进行查找。

二、行逻辑链接的顺序表（带行链接信息的三元组表）

为了便于随机存取任意一行的非零元，需知道每一行的第一个非零元在三元组表中的位置，修改前述的稀疏矩阵的结构定义，增加一个数据成员**rpos**。

```
typedef struct {
Triple  data[MAXSIZE + 1]; //非零元三元组表
int  rpos[MAXRC + 1]; //各行第一个非零元的位置表
int  mu, nu, tu; //矩阵的行数、列数、非零元个数
} RLSMatrix; // 行逻辑链接的顺序表类型
```

三、十字链表

基本概念

当矩阵的<u>非零元个数</u>和<u>位置</u>在操作过程中变化较大时，就不宜采用顺序存储结构来表示三元组的线性表，采用链式存储结构表示三元组的线性表更为恰当。

在十字链表中，每个非零元可用一个含5个域的结点表示，其中 i, j 和e 三个域分别表示该非零元所在的行、列和非零元的值，向右域 right 用以链接同一行中下一个非零元，向下域down 用以链接同一列中下一个非零元。

同一行的非零元通过 right 域链接成一个线性链表，同一列的非零元通过 down 域链接成一个线性链表，每个非零元既是某个行链表中的一个结点，又是某个列链表中的一个结点，整个矩阵构成了一个十字交叉的链表，故称这样的存储结构为十字链表，可用两个分别存储行链表的头指针和列链表的头指针的一维数组表示之。

例如：矩阵M的十字链表如下图所示

列链表的头指针

∧

1 1 3

1 4 5

∧ ∧

行链表的头指针

向右域 right 用以链接同一行中下一个非零元，向下域 down 用以链接同一列中下一个非零元

2 2 -1

∧ ∧

3 1 2

∧ ∧

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

稀疏矩阵的十字链表存储表示

```
typedef struct OLNode{
  int    i,j;//该非零元的行和列下标
  ElemType    e;
 Struct    OLNode    *right,*down;
 //该非零元所在行表和列表的后继链域
 }OLNode;*OLink;

typedef struct{
OLink  *rhead,*chead;//行和列链表头指针向量基
//址由CreateSMatrix分配
int mu,nu,tu;//稀疏矩阵的行数，列数和非零元个数
}CrossList;
```

4.3 广义表

- 广义表（列表）： $n\ (\geq 0\ )$个表元素组成的有限序列，

记作$LS = (a_0, a_1, a_2, \ldots, a_{n-1})$

LS是表名，a_i是表元素，它可以是表 (称为子表)，可以是数据元素(称为原子)。

- n为表的长度。$n = 0$ 的广义表为空表。

广义表的类型定义

ADT Glist {

数据对象： $D = \{e_i \mid i=1,2,..,n;\ n \geq 0;$

$e_i \in AtomSet$ **或** $e_i \in GList,$

AtomSet为某个数据对象 $\}$

数据关系：

$LR = \{<e_{i-1}, e_i > \mid e_{i-1}, e_i \in D, 2 \leq i \leq n\}$

} ADT Glist

广义表是**递归**定义的**线性结构**，

$$LS = (\alpha_1, \alpha_2, \cdots, \alpha_n)$$

其中：α_i 或为原子 或为广义表

例如： $A = (\ )$

$F = (d, (e))$

$D = ((a,(b,c)), F)$

$C = (A, D, F)$

$B = (a, B) = (a, (a, (a, \cdots,)))$

广义表与线性表的区别？

- 线性表的成分都是结构上不可分的单元素
- 广义表的成分可以是单元素，也可以是有结构的表
- 线性表是一种特殊的广义表

广义表是一个多层次的线性结构

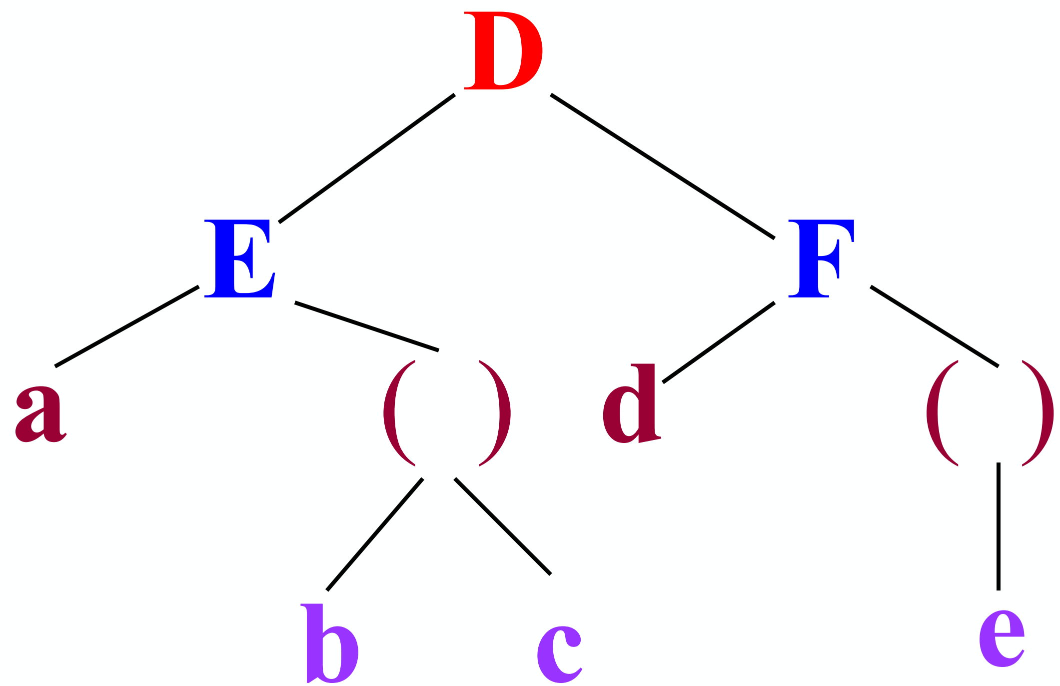
例如：

D=(E, F)

其中：

E=(a, (b, c))

F=(d, (e))



广义表的基本运算

- （1）求表头GetHead(L)：非空广义表的第一个元素，可以是一个单元素，也可以是一个子表
- （2）求表尾GetTail(L)：非空广义表除去表头元素以外其它元素所构成的表。表尾一定是一个表

练习

A=( )	GetHead和GetTail均无定义
A=(a,b)	GetHead(A)=a GetTail(A)=(b)
A=(a)	GetHead(A)=a GetTail(A)=( )
A=((a))	GetHead(A)=(a) GetTail(A)=( )

A=(a,b,(c,d),(e,(f,g)))

GetHead(GetTail(GetHead(GetTail(GetTail(A))))) d

广义表的特点

- 有次序性　一个直接前驱和一个直接后继
- 有长度　＝表中元素个数
- 有深度　＝表中括号的重数
- 可递归　自己可以作为自己的子表
- 可共享　可以为其他广义表所共享

练习：求下列广义表的长度

1）A =( )　　n=0，因为A是空表

2）B = (e)　　n=1，表中元素e是原子

3）C =(a ,(b , c , d))　　n=2，a 为原子，(b,c,d)为子表

4）D=(A , B ,C)　　n=3，3个元素都是子表

5）E=(a, E)　　n=2，a 为原子，E为子表

D=(A,B,C)=(( ),(e),(a,(b,c,d)))，共享表

E=(a,E)=(a,(a,E))= (a,(a,(a,…….)))，E为递归表

广义表的存储结构

通常采用头、尾指针的链表结构

表结点：

tag=1	**hp**	**tp**

原子结点：

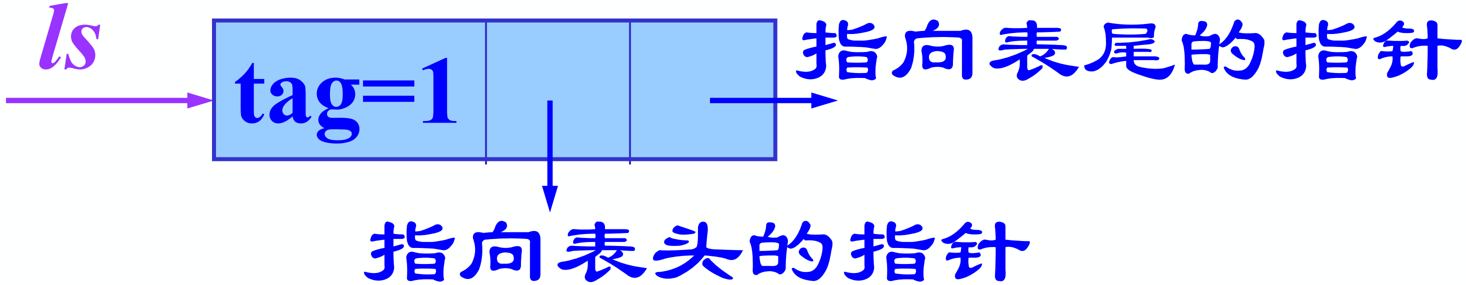
tag=0	**data**

构造存储结构的两种分析方法：

1) 表头、表尾分析法：

空表 *ls=NIL*

非空表 *ls*

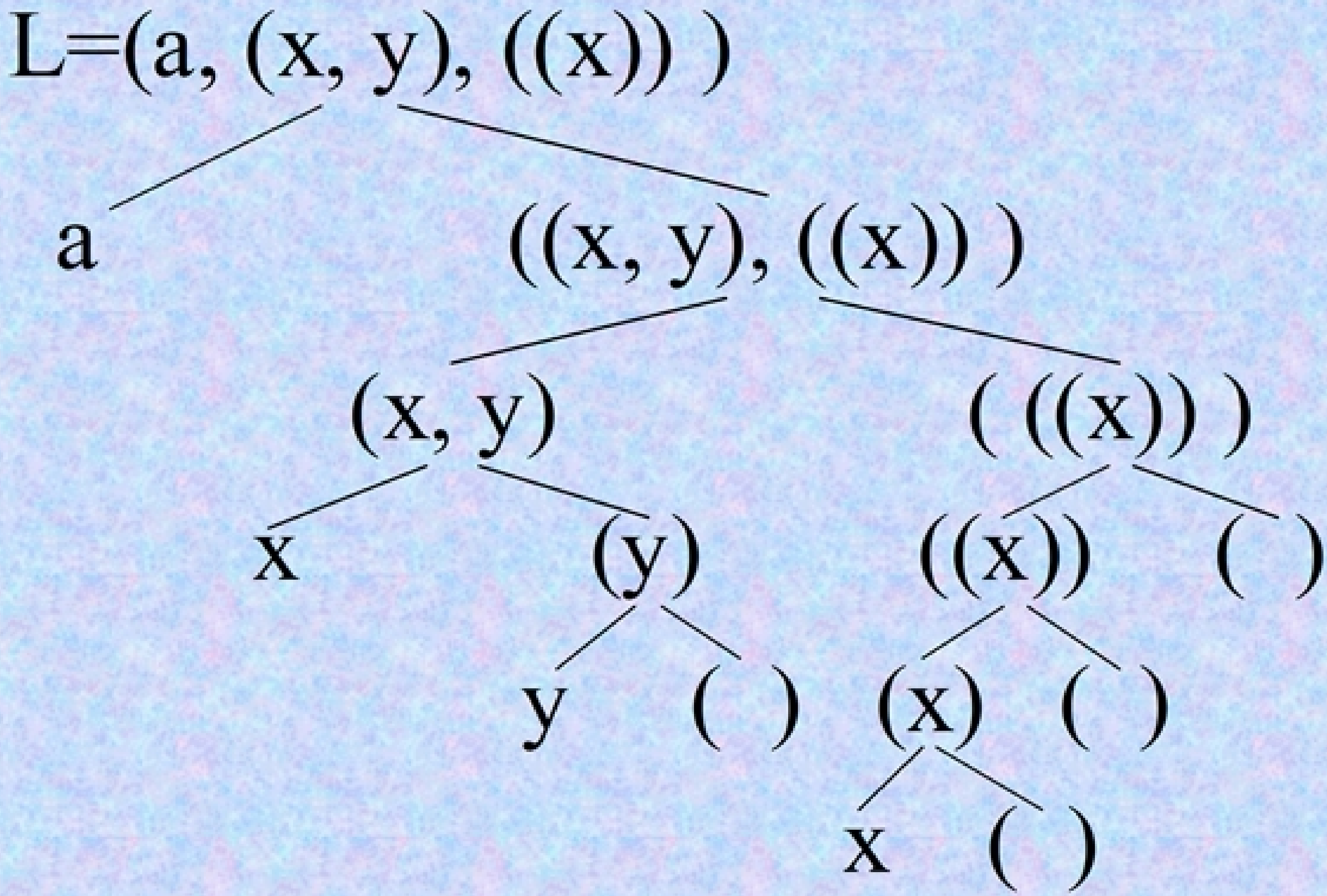


若表头为原子，则为

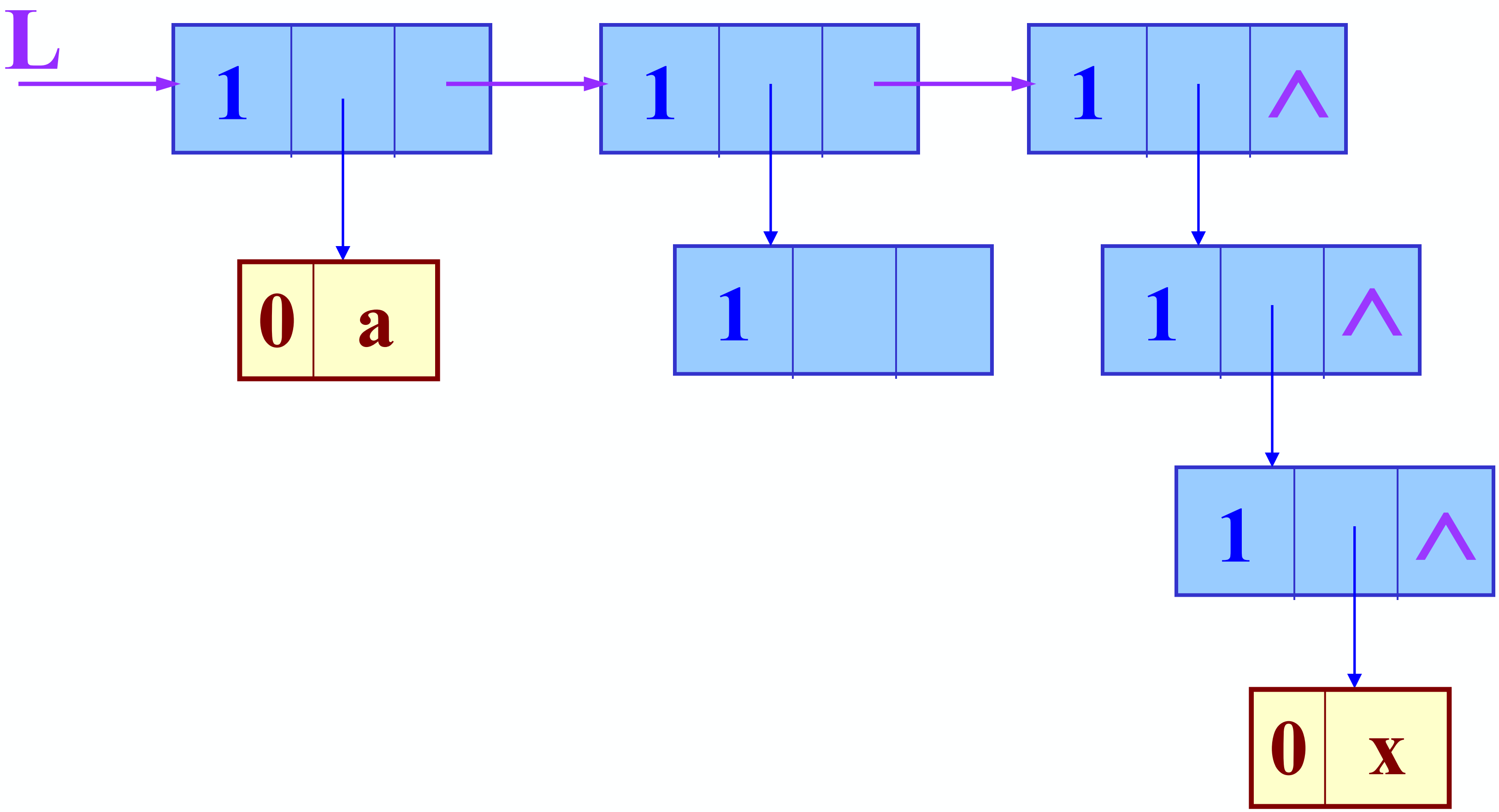
tag=0	data

否则，依次类推。

例如:



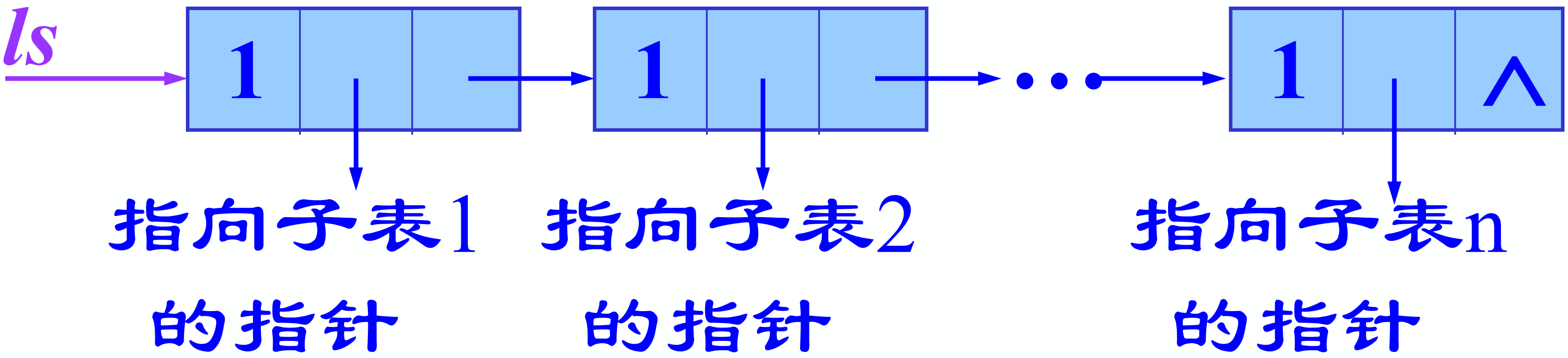
L = (a, (x, y), ((x)))



2) 子表分析法：

空表　　*ls=NIL*

非空表

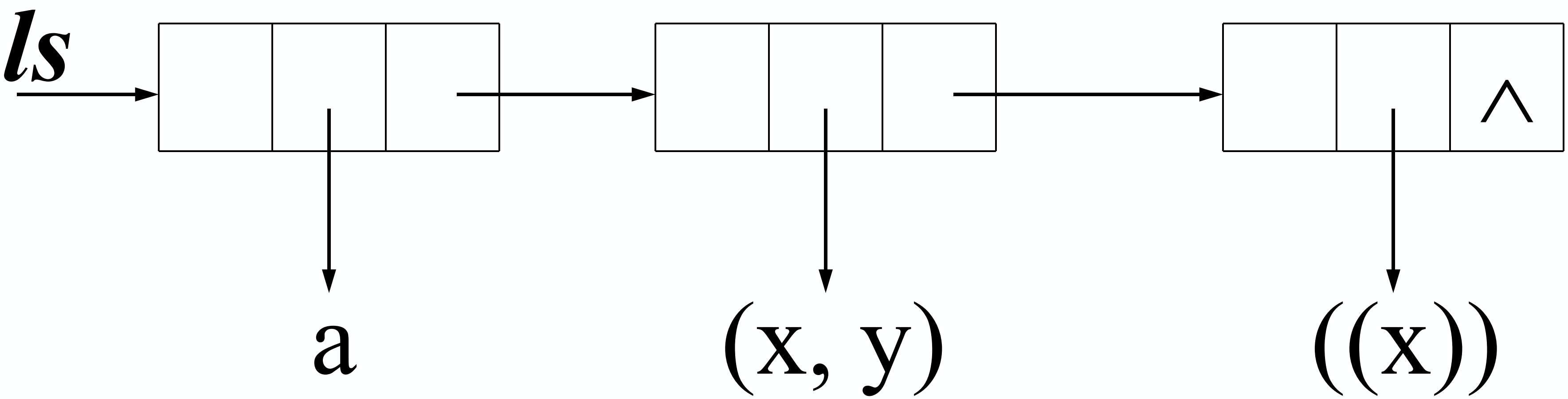


若子表为原子，则为

tag=0	**data**

否则，依次类推。

例如: LS=(a, (x,y), ((x)))



1. 了解串的存储方法，理解串的两种模式匹配算法，重点掌握BF算法。
2. 明确数组和广义表这两种数据结构的特点，掌握数组地址计算方法，了解几种特殊矩阵的压缩存储方法。
3. 掌握广义表的定义、性质及其GetHead和GetTail的操作。

Homework-1

- 2:(1) (2)