第4章 串和数组

- 串 （即字符串）是一种特殊的线性表，它的数据元素仅由一个字符组成。
  - 串及其运算
  - 串的存储结构
  - 串运算的实现
- 数组：可以看成是线性表的扩展：表中的数据元素本身也是一个数据结构。
  - 数组的基本概念
  - 数组的顺序存储结构
  - 矩阵的压缩存储

串及其运算

- 基本概念　串（String）是由零个或多个字符组成的有限序列。一般记为 S="$a_1a_2...a_n$"，其中S是串名，用两个双引号括起的字符序列是串值；a_i可以是字母、数字或其它字符；串中所包含的字符个数成为该串的长度。长度为零的串称为空串，它不包含任何字符。

串及其运算

- 基本概念　串（String）是由零个或多个字符组成的有限序列。一般记为 S="$a_1a_2...a_n$"，其中S是串名，用两个双引号括起的字符序列是串值；a_i可以是字母、数字或其它字符；串中所包含的字符个数成为该串的长度。长度为零的串称为空串，它不包含任何字符。

  串中任意个连续的字符组成的子序列称为该串的子串。包含子串的串称为主串。

串及其运算

- 基本概念　串（String）是由零个或多个字符组成的有限序列。一般记为 S="$a_1a_2\ldots a_n$"，其中S是串名，用两个双引号括起的字符序列是串值；a_i可以是字母、数字或其它字符；串中所包含的字符个数成为该串的长度。长度为零的串称为空串，它不包含任何字符。

  串中任意个连续的字符组成的子序列称为该串的子串。包含子串的串称为主串。

  子串的第一个字符在主串中的序号，定义为子串在主串中的位置（或序号）。特别地，空串是任意串的子串，任意串是其自身的子串。

串的基本操作

- 串的逻辑结构与线性表极为相似，区别仅在于串的数据对象约束为字符集。
- 串的基本操作与线性表有很大差别。通常以“串的整体”作为操作对象，如在串中查找某个子串、取某个子串、删除一个子串等。
- 假设：

  S_1=“$a_1a_2\ldots a_n$”

  S_2=“$b_1b_2\ldots b_m$”

  其中，$1\le m\le n$

串的基本操作-1

1．赋值：strcopy(S,T)表示将串T的值赋给串变量S。T既可以是一个串变量，也可以是串常量。

例如：T=“BEIJING”，strcopy(S,T)的结果使S的值也是“BEIJING”。

2．连接：strcat(S,T)表示将串T紧接着放在串S末尾，组成一个新的串S。例如：

S=“BEIJING”；T=“SHANGHAI”；

strcat(S,T)； 则 S= “BEIJINGSHANGHAI”

3．求串长： strlen(S)表示求串S的长度。

例如： S=“BEIJING”； 则strlen(S)=7

串的基本操作-2

4. 求子串： substr(S,i,j)表示从S中第i个字符开始抽出j个字符构成一个新的串，这个新串是S的子串，其中的参数应满足：
0≤i≤strlen(S)-1,0≤j≤strlen(S)-i

例如：S=“abcdefg”；则substr(S,1,4)= “bcde”

利用求子串和连接运算可以完成对串的插入、删除和修改。

5. 比较串的大小：strcmp(S,T)表示比较两个串S和T的大小，其函数值是一个整数，大于0表示S＞T，小于0表示S＜T，等于0表示S=T。

例如： “there”＜ “this”； “there”＞ “the” 。

6. 插入：strinsert(S,i,T)表示把串T插入到S的第i个字符处。

例如：S=“abcabc”； T=“def”；

strinsert(S,3,T) ； 则结果S=“abcdefabc”

串的基本操作-3

7. 删除：strdelete(S,i,j)表示从串S中删除第i个字符开始的连续j个字符。例如：　S=“Beijing”; strdelete(S,3,4);　则结果S=“Bei”

8. 子串定位：strindex(S,T)是一个求子串在主串中位置的定位函数，表示在主串S中查找是否有等于T的子串，若有，结果为T在S中首次出现的位置；否则，则函数值为-1。例如：

   strindex(“abcdef”, “cd”)=2

   strindex(“abcdef”, “xy”)=-1

9. 置换：replace(S,i,j,T)表示用T置换S中第i个字符开始的连续j个字符。 例如：S=“this is a book”; replace(S,2,2, “at”); 则结果S=“that is a book”

上述串运算都是基本的运算，利用这些基本运算可以完成关于串的各类需求下的操作。

串与线性表

- **Q：** 串的基本操作和线性表一样，无非也就是查找、插入和删除等，那么它们能否用线性表的操作来替代呢？

串与线性表

- **Q：** 串的基本操作和线性表一样，无非也就是查找、插入和删除等，那么它们能否用线性表的操作来替代呢？
- **A：** 串的基本操作和线性表有很大的区别，同样是查找、插入和删除，但对线性表言操作对象是"数据元素"，如在线性表中查找某一个特定的数据元素，或者插入/删除一个数据元素。而对串言，是以整个串作为操作对象，如将两个串联接在一起，在串中查找一个子串，插入/删除一个串等等。

串的存储结构-1

1. 顺序存储：串的顺序存储结构（顺序串）中的字符依次存放在一片连续的单元中。为了清楚起见，以下将用符号"Φ"表示"空格符"。

串的存储结构-1

1. 顺序存储：串的顺序存储结构（顺序串）中的字符依次存放在一片连续的单元中。为了清楚起见，以下将用符号"Φ"表示"空格符"。

顺序串可用以下类型描述：

```
#define  maxsize  100        /* 假设串可能的最大长度为100 */
typedef  struct
{   char  ch[maxsize];       /* 存放串值 */
    int len;                 /* 串的长度 */
}seqstring;
```

串的存储结构-1

1. 顺序存储：串的顺序存储结构（顺序串）中的字符依次存放在一片连续的单元中。为了清楚起见，以下将用符号"Φ"表示"空格符"。

顺序串可用以下类型描述：

```
#define  maxsize  100        /* 假设串可能的最大长度为100 */
typedef  struct
{   char  ch[maxsize];       /* 存放串值 */
    int len;                 /* 串的长度 */
}seqstring;
```

在C语言中用字符'\0'作串的终结符，串S="this is a string"的顺序存储结构如下图

0	1	2	3	4	5	6	7	.....	15	16
t	h	i	s	Φ	i	s	Φ	a	g	\0

串的存储结构-2

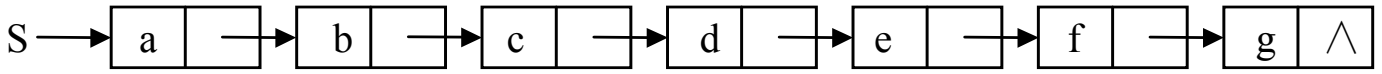
2. 链式存储　串的链式存储结构简称为链串。链串的类型定义和单链表类似：

```
typedef struct  linknode
  { char data;
    struct linknode *next;
  }linkstring;              /*  定义链串类型 */
  linkstring *S;
```

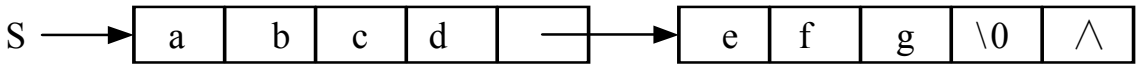
串的存储结构-2

2. 链式存储 串的链式存储结构简称为链串。链串的类型定义和单链表类似：

```
typedef struct  linknode
  { char data;
    struct linknode *next;
  }linkstring;              /*  定义链串类型 */
  linkstring *S;
```



(a) 结点大小为1的链串



(b) 结点大小为4的链串

串的存储结构-3

3. 索引存储

在索引存储中，除了存放串值外，还要建立一个串名和串值之间对应关系的索引表。

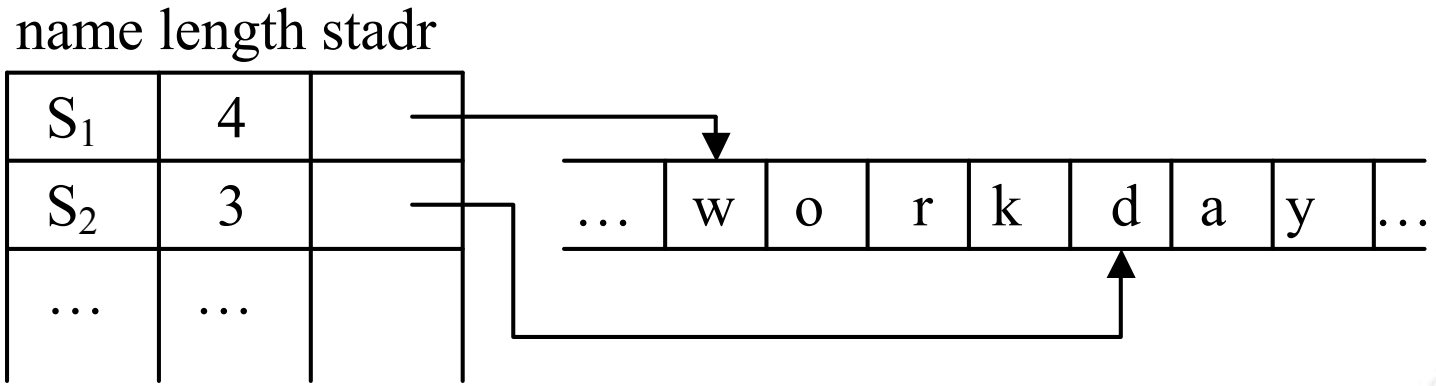
链式存储方式下，索引表中要含有串名及存储串值的链表的头指针；

顺序存储方式下，则索引表中要含有串名以及指示串值存放的起始地址的首指针和指示串值存放结束的末地址。末地址信息可以是串值末尾结束地址、串长等。

索引存储

(1)带长度的索引表 索引表的结点类型是：

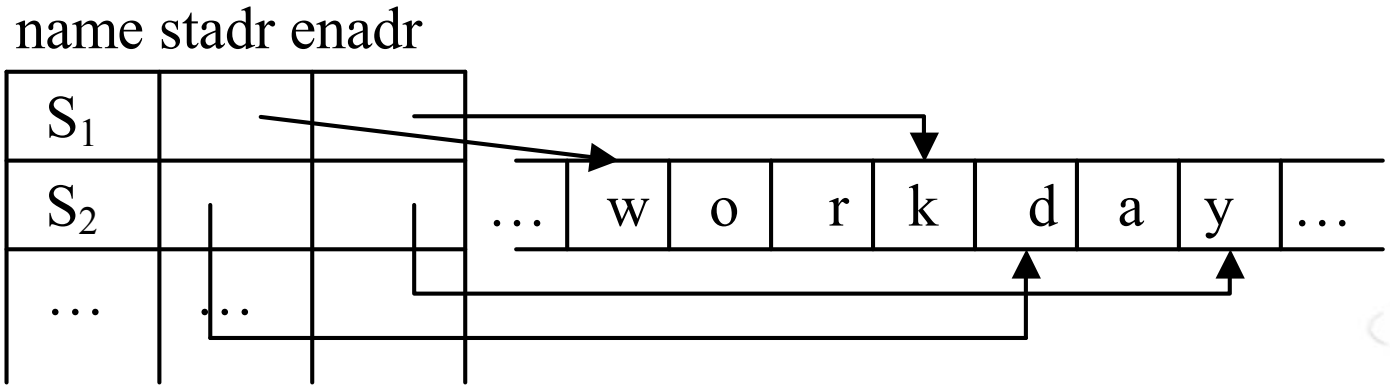
```
typedef struct
{  char name[maxsize];      /*  串名 */
   int length;              /*  串长 */
   char *stadr;             / 串值存入的起始地址 */
}
```



索引存储

(2)带末指针的索引表 用一个指向串值存放的末地址的指针**enadr**来代替长度**length**

```
typedef struct
{  char name[maxsize];
   char *stadr, *enadr;
}enode;
```



索引存储

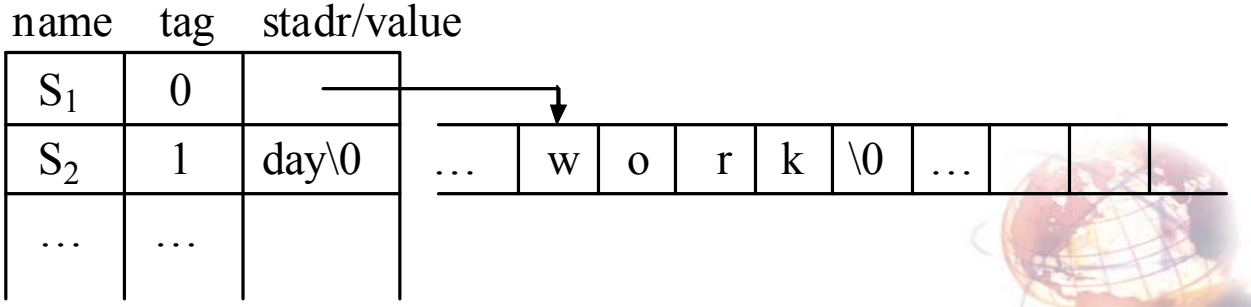
(3)带特征位的索引表 当串值只需要一个指针域的空间就能存放时，可将串值放在**stadr**域中。要增加一个特征位**tag**来指出**stadr**域中是指针还是串值。

```
typedef struct
{  char name[maxsize];
   int tag;        /* 特征位 */
   union
   {  char *stadr;
      char value[4];
   }uval;
}tagnode;
```

索引存储

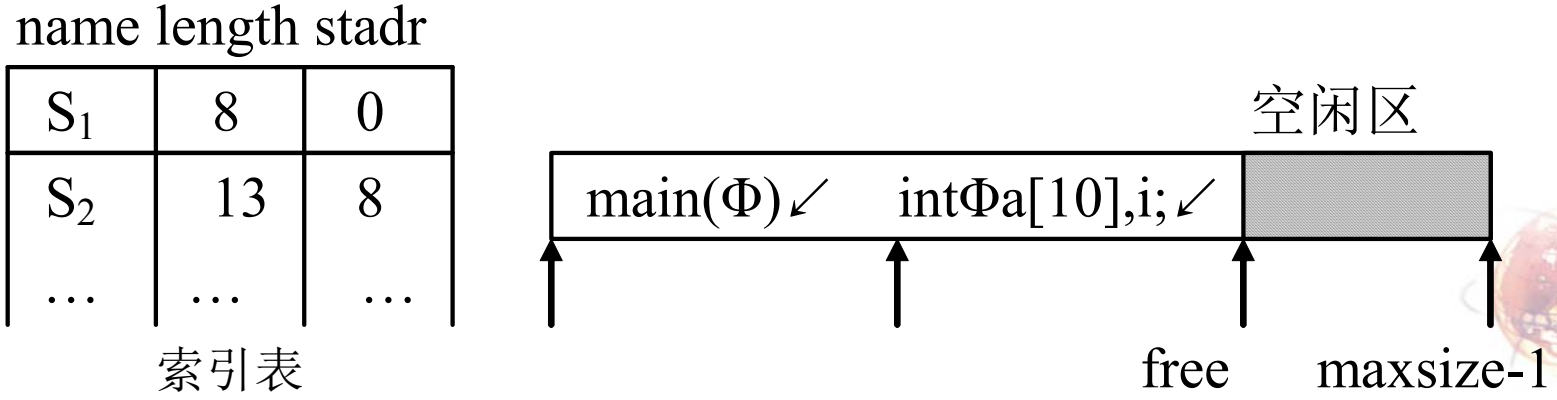
(3)带特征位的索引表 当串值只需要一个指针域的空间就能存放时，可将串值放在**stadr**域中。要增加一个特征位**tag**来指出**stadr**域中是指针还是串值。

```
typedef struct
{  char name[maxsize];
   int tag;        /* 特征位 */
   union
   {  char *stadr;
      char value[4];
   }uval;
}tagnode;
```



索引存储中串的动态分配

- 一个较大的向量**store[maxsize]**表示可供动态分配用的连续的存储空间，用一个指针指示尚未分配存储空间的起始位置，其初值为**0**。
- 当程序执行过程中每产生一个新串，就从**free**指针起进行存储分配，同时在索引表中建立一个相应的结点，在该结点中填入新串的名字、分配到的串值空间的起始位置、串值的长度等信息，然后修改**free**指针。



串运算的实现

- 顺序串的类型定义如下：

```
#define  maxsize  100        /* 假设串可能的最大长度为100 */
typedef  struct
{  char  ch[maxsize];        /* 存放串值 */
    int len;                 /* 串的长度 */
}seqstring;
seqstring *S;
```

串的连接运算

- 将两个串S和T首尾相接连成一个串R，其中S在前，T在后。

```
seqstring *STRCAT(seqstring *S,seqstring *T)
{  int i;
   seqstring *R;
    printf("S=%s T=%s\n",S→ch,T→ch);
    if(S→len+T→len>maxsize)  printf("上溢\n");
     /* 若两串长度之和大于maxsize，则进行溢出处理 */
   else
  {  for(i=0;i<S→len;i++)      /* 将S串传给R */
          R→ch[i]=S→ch[i];
     for(i=0;i<T→len;i++)      /* 将T串传给R */
          R→ch[S→len+i]=T→ch[i];
     R→ch[S→len+i]='\0'        /* 最后一个位置赋'\0' */
     R→len=S→len+T→len;        /* 串长度等于两串之和 */
  }
   return（R）;
} /* STRCAT*/
```

求子串运算

- 设S为主串，现将S中的第i个字符起，抽取j个字符构成一个子串，结果存放在T中。

```
seqstring *SUBSTR(seqstring *S, int i, int j)
{  int k;
   seqstring *T;
   if(i+j>S→len)  printf("超界\n");
   /* 若i,j的值超出允许的范围，则进行“超界”处理 */
   else
  { for(k=0;k<j;k++)
        T→ch[k]=S→ch[i+k];    /* 将S中指定的子串传给T  */
    T→len=j;                  /* 将子串长度赋给T的长度域 */
    T→ch[T→len]= '\0';
  }
   return(T);
} /* SUBSTR */
```

子串定位

- 子串定位运算又称为串的模式匹配，是串处理中最重要的运算之一。
- 设有两个串S和T，S="$s_1s_2\ldots s_n$"，T="$t_1t_2\ldots t_m$"，其中$0<m\leq n$。子串定位是要在主串S中找出一个与子串T相同的子串（第一个）。
- 一般把主串S称为目标，子串T称为模式，把从目标S中查找模式为T的子串的过程称为"模式匹配"。

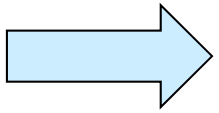
朴素的模式匹配 (布鲁特——福斯算法)

- 从目标串**S**中的第一个字符开始和模式串**T**中的第一个字符比较，**i**和**j**指示**S**串和**T**串正在比较的字符位置。
- 匹配成功，返回模式串**T**中第一个字符在目标串**S**中的位置，否则匹配失败，返回零值。

朴素的模式匹配 (布鲁特——福斯算法)

- 从目标串**S**中的第一个字符开始和模式串**T**中的第一个字符比较，**i**和**j**指示**S**串和**T**串正在比较的字符位置。
- 匹配成功，返回模式串**T**中第一个字符在目标串**S**中的位置，否则匹配失败，返回零值。

S=“abbaba”，T=“aba”为例的模式匹配过程



```
第一趟匹配：  a  b  b  a  b  a      i=2
              ‖  ‖  ≠            j=2    失败
              a  b  a
第二趟匹配：  a  b  b  a  b  a      i=1
                 ≠               j=0    失败
                 a
第三趟匹配：  a  b  b  a  b  a      i=2
                    ≠            j=0    失败
                    a
第四趟匹配：  a  b  b  a  b  a      i=5
                       ‖  ‖  ‖   j=2    成功
                       a  b  a
```

朴素的模式匹配算法实现

- 第k趟比较是从S中的第k-1个字符(i=k-1)开始比较；$t_j \neq s_i$, $t_{j-1}=s_{i-1}$, $t_{j-2}=s_{i-2}$,…$t_0=S_{i-j}$, 下一次t_0对应S开始比较位置是i-j+1。

```
int INDEX(seqstring *S,seqstring *T)  /* 在目标串中找模式T首次出现位置 */
{  int i=0,j=0;
   while((i<S→len)&&(j<T→len))
    { if(S→ch[i]==T→ch[j])
     { i++;j++;}          /* 继续比较后面的字符 */
     else
     {  i=i-j+1; j=0;}
     }  /* 从模式的第一个字符进行新一趟匹配 */
   if(j==T→len)
      return(i-T→len);      /* 匹配成功 */
   else
      return -1;           /* 匹配失败 */
}  /*INDEX */
```

朴素的模式匹配算法实现

- 第k趟比较是从S中的第k-1个字符(i=k-1)开始比较；$t_j \neq s_i$, $t_{j-1}=s_{i-1}$, $t_{j-2}=s_{i-2}$,…$t_0=S_{i-j}$, 下一次t_0对应S开始比较位置是i-j+1。

```
int INDEX(seqstring *S,seqstring *T)  /* 在目标串中找模式T首次出现位置 */
{  int i=0,j=0;
   while((i<S→len)&&(j<T→len))
    { if(S→ch[i]==T→ch[j])
     { i++;j++;}          /* 继续比较后面的字符 */
     else
     { i=i-j+1; j=0;}
     }  /* 从模式的第一个字符进行新一趟匹配 */
   if(j==T→len)
      return(i-T→len);      /* 匹配成功 */
   else
      return -1;           /* 匹配失败 */
} /*INDEX */
```

后面可以加上i＞S→len-T→len，提前终止循环

简单模式匹配算法

- **Q：**你能否举出一个使朴素模式匹配算法运行处最坏情况的例子？

简单模式匹配算法

- **Q：** 你能否举出一个使朴素模式匹配算法运行处最坏情况的例子？
- **A：** 例如：**S =“aaaaaaaaaaaab”，T =“aaab”**，则简单模式匹配算法要执行 **36** 次对应位的比较才匹配成功。

算法的时间复杂性

- 最好的情况下，每趟不成功的匹配都是发生在T的第一个字符与S中相应字符的比较。
- 从S的第i个位置开始与T模式匹配成功的概率为p_i，则在前i-1趟匹配中字符比较了i-1次，若第i趟成功的匹配中字符比较次数为m，则总的比较次数为i-1+m。

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

算法的时间复杂性

- 最好的情况下，每趟不成功的匹配都是发生在T的第一个字符与S中相应字符的比较。
- 从S的第i个位置开始与T模式匹配成功的概率为p_i，则在前i-1趟匹配中字符比较了i-1次，若第i趟成功的匹配中字符比较次数为m，则总的比较次数为i-1+m。

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

最好情况下算法的平均时间复杂度为O(n+m)

算法的时间复杂性

- 在最坏情况下，每一趟不成功的匹配都发生在模式串**T**的最后一个字符与**S**中相应字符的比较不相等。
- 新一趟的起始位置为**i-m+1**，若设第**i**趟匹配成功，则前**i-1**趟不成功的匹配中，每趟都比较了**m**次，总共比较了**i**×**m**次。

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

算法的时间复杂性

- 在最坏情况下，每一趟不成功的匹配都发生在模式串T的最后一个字符与S中相应字符的比较不相等。
- 新一趟的起始位置为i-m+1，若设第i趟匹配成功，则前i-1趟不成功的匹配中，每趟都比较了m次，总共比较了i×m次。

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

由于n＞＞m，则该算法在最坏情况下的时间复杂度O(m×n)。

模式匹配算法 (链式存储结构)

```
linkstring *INDEXL(linkstring *S,linksting *T)  /* 求T在S中首次出现的位置*/
{  linkstring *first,*sptr,*tptr;        /*  S,T是不带头结点的链串 */
   first=S;          /* first指向S的起始比较位置 */
   sptr=first;  tptr=T;
   while(sptr&&tptr)
   {   if(sptr→data==tptr→data)    /* 继续比较后继结点的字符 */
       {  sptr=sptr→next;
          tptr=tptr→next;   }
       else      /* 本趟匹配失败，回溯 */
       {  first=first→next;
          sptr=first; tptr=T; }
   }
  if(tptr==NULL)  return first;      /* 匹配成功 */
  else  return NULL;          /* 匹配失败 */
}  /* INDEXL  */
```

串操作应用—正文编辑

- 正文编辑的实质是修改字符数据的形式或格式。无论是 **Microsoft word** 还是 **WPS**，其工作的基础原理都是正文编辑。
- 虽然各种正文编辑程序的功能强弱不同，但其基本功能大致相同，一般都包括串的查找、插入、删除和修改等基本操作。
- 可以通过换页符和换行符将正文划分为若干页和若干行(或直接划分为若干行)。在编辑程序中，则可将整个正文看成是一个"正文串"，"页"是正文串的子串，而行则是页的子串。

串操作应用—正文编辑

```
main(){

    float a,b,max;
    scanf(“%f,%f”,&a,&b);
    if a>b max=a;
    else max=b;
};
```

将此源程序看成是一个正文串，输入内存后如图所示，图中"↙"为换行符。

m	a	i	n	(	)	{	↙			f	l	o	a	t		a	,	b	,
m	a	x	;	↙			s	c	a	n	f	(	"	%	f	,	%	f	"
,	&	a	,	&	b	)	;	↙			i	f		a	>	b			m
a	x	=	a	;	↙			e	l	s	e			m	a	x	=	b	;
↙	}	;	↙																

串操作应用－正文编辑

编辑程序先为正文串建立相应的页表和行表，页表的每一项列出页号和该页的起始行号，行表的每一项则指示每一行的行号、起始地址和该行子串的长度。

如果正文串只占一页，起始行号为100，则该正文串的行表如右所示。

行号	起始地址	长度
100	200	8
102	208	17
104	225	24
105	249	17
106	266	15
108	281	3

数组的定义和运算

- 数组的定义
  - 数组是由值与下标构成的有序对，结构中的每一个数据元素都与其下标有关。
  - 数组结构的性质：
    - (1)数据元素数目固定：一旦说明了一个数组结构，其元素数目不再有增减变化；
    - (2)数据元素具有相同的类型；
    - (3)数据元素的下标关系具有上下界的约束并且下标有序。

$$A_{mn}=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ & \vdots & & \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

- 数组有两种运算：
  - 给定一组下标，存取相应的数据元素；
  - 给定一组下标，修改相应数据元素中的某个数据项的值。

数组概念思考

- Q：在你的认识中，“数组”是什么？

- Q：为什么顺序表以及其它线性结构的顺序存储结构都可以用“一维数组”来描述？

数组概念思考

- Q：在你的认识中，“数组”是什么？
- A：在学习了C语言之后，可能会认为“数组”是一组地址连续的内存，数组也是一种线性的数据结构，可以看成是线性表的一种扩充。

- Q：为什么顺序表以及其它线性结构的顺序存储结构都可以用“一维数组”来描述？

数组概念思考

- Q：在你的认识中，“数组”是什么？
- A：在学习了C语言之后，可能会认为“数组”是一组地址连续的内存，数组也是一种线性的数据结构，可以看成是线性表的一种扩充。

- Q：为什么顺序表以及其它线性结构的顺序存储结构都可以用“一维数组”来描述？

数组概念思考

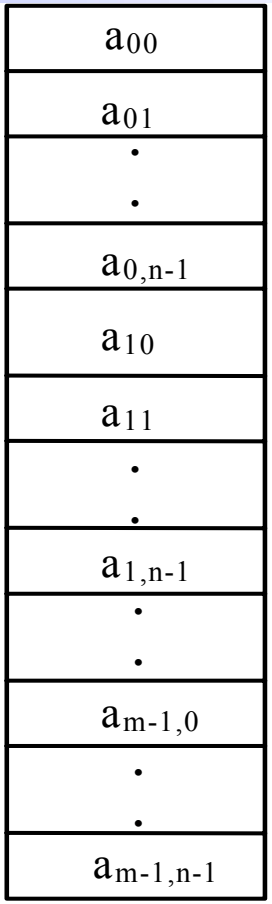
- Q：在你的认识中，“数组”是什么？
- A：在学习了C语言之后，可能会认为“数组”是一组地址连续的内存，数组也是一种线性的数据结构，可以看成是线性表的一种扩充。

- Q：为什么顺序表以及其它线性结构的顺序存储结构都可以用“一维数组”来描述？
- A：因为在高级编程语言中实现的一维数组正是用的这种顺序存储的映象方式。

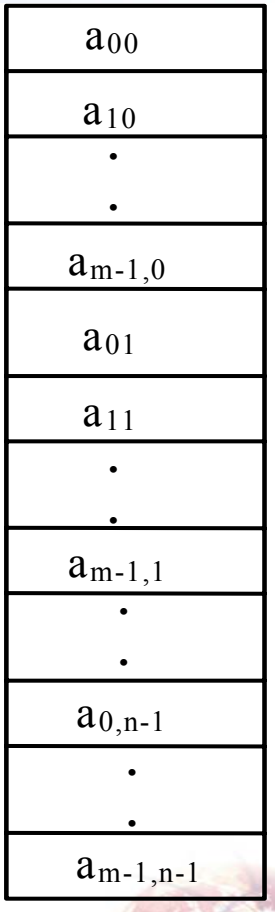
数组的顺序存储结构

- 次序约定问题：
  - 存储单元是一维的结构，而数组是个多维的结构，那么用一组连续存储单元如何存放数组的数据元素？
  - 以行为主序的优先存储：图**(a)**
  - 以列为主序的优先存储：图**(b)**

$$A_{mn}=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ & \vdots & & \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$



(a) 以行为主序



(b) 以列为主序

顺序存储的数组是随机存取结构

- A_{mn}以行优先存储的地址计算公式：

  $$Loc(a_{ij})=Loc(a_{11})+[(i-1)*n+(j-1)]*d$$

  其中：d 是每个数据元素占用的存储单元数
- 以列优先存储的地址计算公式：

  $$Loc(a_{ij})=Loc(a_{11})+[(j-1)*m+(i-1)]*d$$
- 三维数组A_{mnp}按行优先顺序存储，其地址计算公式：

  $$Loc(a_{ijk})=Loc(a_{111})+[(i-1)*n*p+(j-1)*p+(k-1)]*d$$
- 在C语言中，数组下标的下界是0，因此二维数组的地址计算公式为（行优先）：

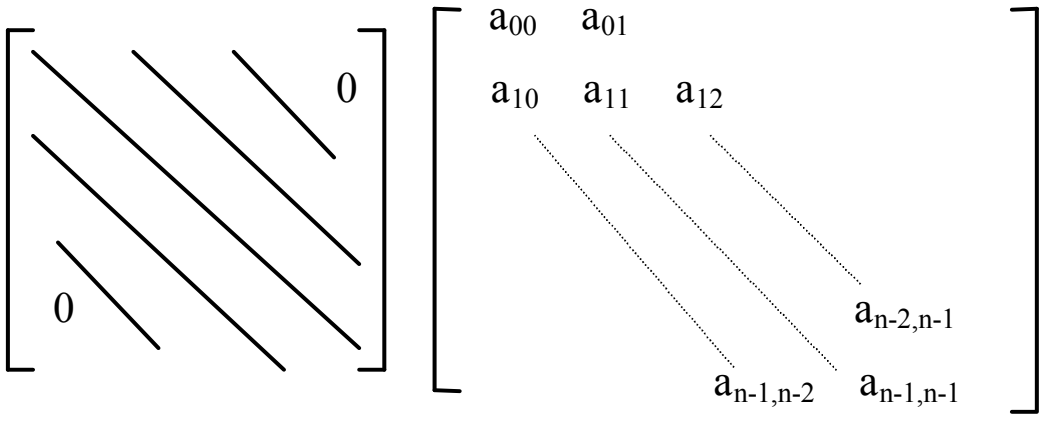
  $$Loc(a_{ij})=Loc(a_{00})+(i*n+j)*d$$

矩阵的压缩存储

- 定义：
  矩阵压缩存储是指矩阵中多个值相同的元素只分配一个空间，对零元素不分配空间。

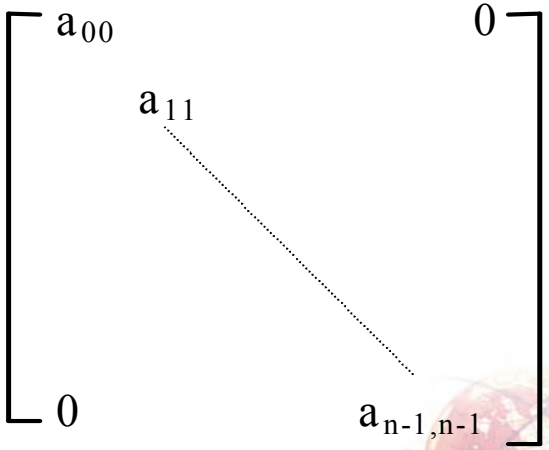
何种矩阵可采用矩阵压缩存储？

- 特殊矩阵
  - 1. 对角矩阵：所有的非零元素都集中在以主对角线为中心的带状区域中。



(a) (b)

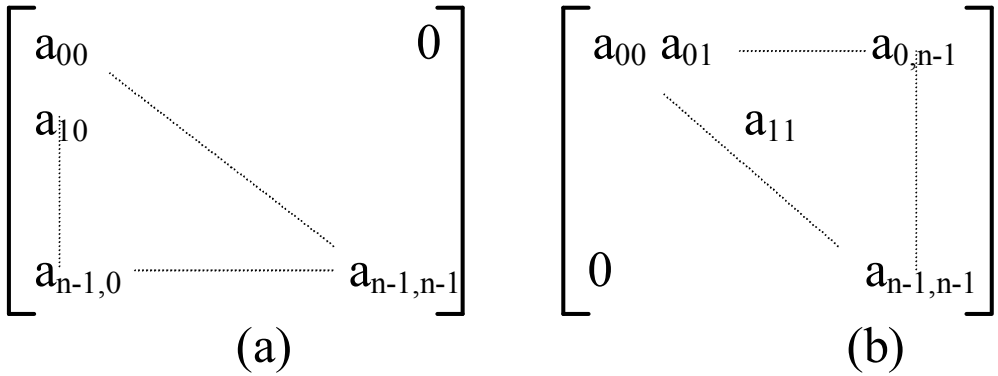
(a) 一般情形 (b) 三对角矩阵



（c）主对角矩阵

三角矩阵

- 定义：
三角矩阵有上三角和下三角两种。上三角矩阵是指矩阵的下三角（不包含对角线）中的元素均为常数（或零）的n阶矩阵，下三角矩阵与之相反。
  - 元素：相同元素占用一个单元，若为0则不存储。
  - 存储：用A[0 ... n*(n+1)/2]数组存储矩阵中的n*(n+1)/2个非零元素。常数占最后一个单元。
  - 数组元素A[k]与a_{ij}的关系



(a)下三角矩阵 (b)上三角矩阵

三角矩阵

- 定义：
三角矩阵有上三角和下三角两种。上三角矩阵是指矩阵的下三角（不包含对角线）中的元素均为常数（或零）的n阶矩阵，下三角矩阵与之相反。
  - 元素：相同元素占用一个单元，若为0则不存储。
  - 存储：用A[0 ... n*(n+1)/2]数组存储矩阵中的n*(n+1)/2个非零元素。常数占最后一个单元。
  - 数组元素A[k]与a_{ij}的关系

$$\begin{bmatrix} a_{00} & & 0 \\ a_{10} & & \\ & \ddots & \\ a_{n-1,0} & \cdots & a_{n-1,n-1} \end{bmatrix} \quad (a)$$

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ & a_{11} & & \\ & & \ddots & \\ 0 & & & a_{n-1,n-1} \end{bmatrix} \quad (b)$$

(a)下三角矩阵 (b)上三角矩阵

下三角矩阵中

$$k = \begin{cases} \dfrac{i*(i+1)}{2} + j & i \geq j \\ \dfrac{n*(n+1)}{2} & i < j \end{cases}$$

k与i、j的关系

对称矩阵

- 定义：在n阶方阵A中，若A中的元素满足$a_{ij}=a_{ji}(0\leqslant i,j\leqslant n-1)$，则称A是对称矩阵，下图给出了一个六阶对称矩阵。
- 元素：对称的元素共享一个存储空间，要存储的元素总数为n*(n+1)/2。
- A[k]与a_{ij}对应关系：
  - k=i*(i+1)/2+j　(0≤k<n*(n+1)/2)　i≥j
  - k=j*(j+1)/2+i　(i<j)
  - 统一的k,i,j的对应关系为：

    k=i*(i+1)/2+j

    其中：i=max(i,j)，　j=min(i,j)。

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

稀疏矩阵

- 定义：

  含有非零元素及较多的零元素，但非零元素的分布没有任何规律，这种矩阵称为稀疏矩阵。即稀疏矩阵$A_{m\times n}$中有s个非零元素，t个零元素，若s<<t，则称A为稀疏矩阵。

- 稀疏矩阵压缩存储方法：
  - 三元组表
  - 十字链表

稀疏矩阵的三元组表

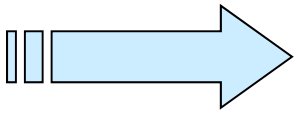
- **定义：**
  将稀疏矩阵的非零元素用三元组按行优先（或列优先）的顺序排列（跳过零元素），则得到一个其结点均是三元组的线性表。
- **三元组：** 是稀疏矩阵的一种顺序存储结构。
  三元组=（行i，列j，非零元素值）——行优先
  三元组=（列j，行i，非零元素值）——列优先
- **数据结构描述：**

```
#define smax 16   /* 最大非零元素个数的常数 */
typedef int datatype;
typedef struct
{ int i,j;      /* 行，列号 */
  datatype v;   /* 元素值 */
} node;
typedef struct
{   int m,n,t; /*行数,列数,非零元素个数*/
    node data[smax];  /* 三元组表 */
} spmatrix;      /* 稀疏矩阵类型 */
```

稀疏矩阵的三元组表示实例

$$A_{4\times5}=\begin{bmatrix}0 & 5 & 0 & 0 & 8\\ 1 & 0 & 3 & 0 & 0\\ 0 & -2 & 0 & 0 & 0\\ 6 & 0 & 0 & 0 & 0\end{bmatrix}$$

稀疏矩阵A



	i	j	v
0	0	1	5
1	0	4	8
⋮	1	0	1
	1	2	3
⋮	2	1	-2
a→ t-1	3	0	6
⋮			
smax-1			

稀疏矩阵A的三元组表

稀疏矩阵三元组表示的转置运算

$$A_{4\times5}=\begin{bmatrix}0&5&0&0&8\\1&0&3&0&0\\0&-2&0&0&0\\6&0&0&0&0\end{bmatrix}$$

转置为

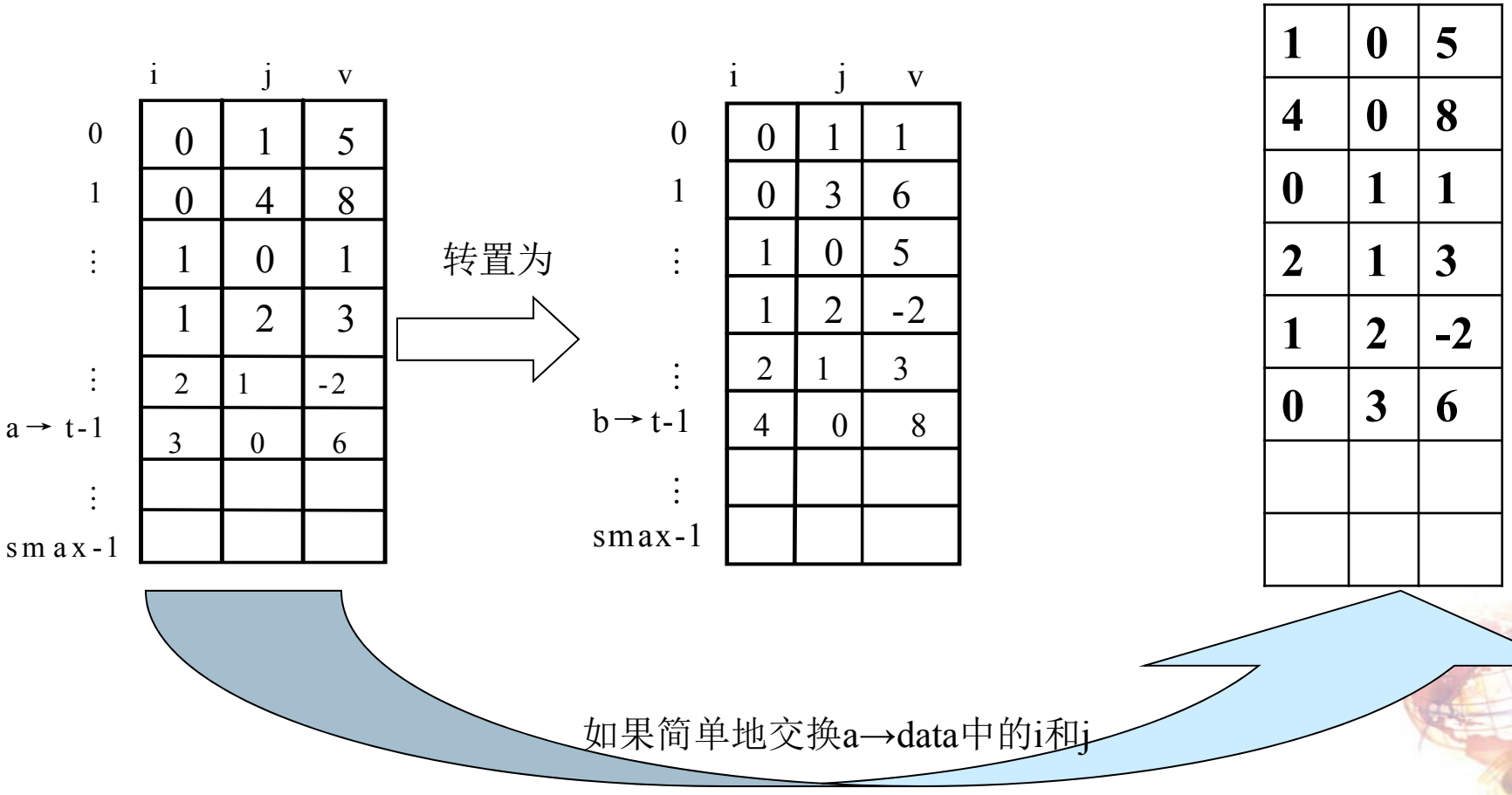
$$B_{5\times4}=\begin{bmatrix}0&1&0&6\\5&0&-2&0\\0&3&0&0\\0&0&0&0\\8&0&0&0\end{bmatrix}$$

	i	j	v
0	0	1	5
1	0	4	8
⋮	1	0	1
	1	2	3
⋮	2	1	-2
a→ t-1	3	0	6
⋮			
smax-1			

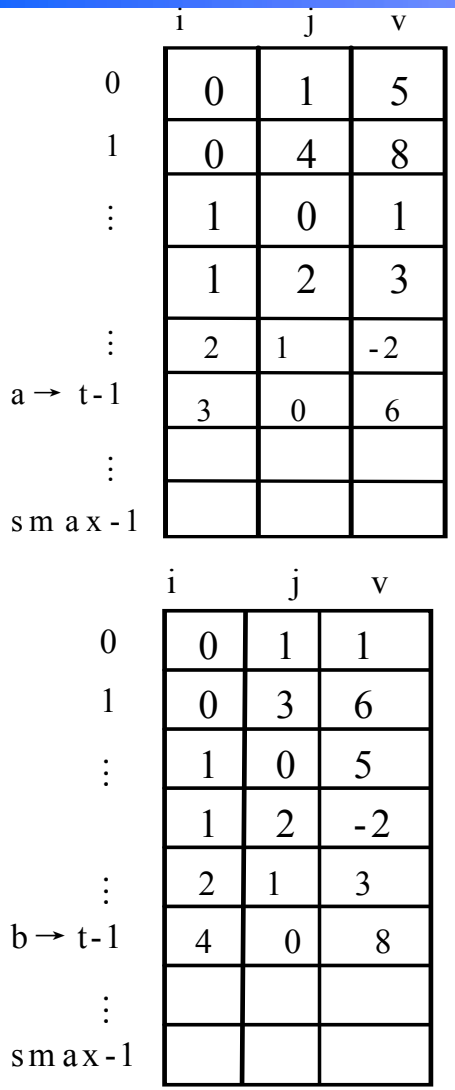
	i	j	v
0	0	1	1
1	0	3	6
⋮	1	0	5
	1	2	-2
⋮	2	1	3
b→ t-1	4	0	8
⋮			
smax-1			

实现转置运算时的注意

- 如果简单地交换a→data中i和j中的内容，那么得到的b→data是一个按列优先顺序存储的稀疏矩阵 B 。



稀疏矩阵三元组表示的转置运算



	i	j	v
0	0	1	5
1	0	4	8
⋮	1	0	1
	1	2	3
⋮	2	1	-2
a→ t-1	3	0	6
⋮			
smax-1			

	i	j	v
0	0	1	1
1	0	3	6
⋮	1	0	5
	1	2	-2
⋮	2	1	3
b→ t-1	4	0	8
⋮			
smax-1			

算法描述如下：

```
spmatrix *TRANSMAT(spmatrix *a)
        /* 返回稀疏矩阵A的转置，ano和bno分*/
        /*别指示a→data和b→data中结点序号 */
{       /* col指示*a的列号(即*b的行号) */
    int ano,bno,col;  spmatrix *b;     /* 存放转置后的矩阵
*/
    b=malloc(sizeof(spmatrix));
    b→m=a→n;  b→n=a→m;       /*行列数交换*/
    b→t=a→t;
    if (b→t>0)                          /* 有非零元素，则转置
*/
    { bno=0;
      for(col=0;col<a→n;col++)      /* 按*a的列序转置 */
      for(ano=0;ano<a→t; ano++)   /* 扫描整个三元组表 */
        if (a→data[ano].j==col)       /* 列号为col则进行置换
*/
          { b→data[bno].i=a→data[ano].j;
            b→data[bno].j=a→data[ano].i;
            b→data[bno].v=a→data[ano].v;
            bno++;              /* b→data结点序号加1 */
          }
     }/* if (b→t>0) */
    return b;  /* 返回转置结果指针 */
}   /* TRANSMAT  */
```

稀疏矩阵转置运算复杂度

- 算法的时间主要耗费在**col**和**ano**的二重循环上，若**A**的列数为**n**，非零元素个数为**t**，则执行时间为O**(n*t)**，即与A的列数和非零元素个数的乘积成正比。
- 通常用二维数组表示矩阵时，其转置算法的执行时间是O**(m*n)**，它正比于行数和列数的乘积。
- 稀疏矩阵转置算法的时间，大于非压缩存储的矩阵转置的时间。

结论和习题

- 结论
  - 串基本概念和基本操作。
  - 串模式匹配算法/实现。
  - 数组的数据结构和顺序存储结构，其核心是数据的下标与存储位置的对应关系。
  - 稀疏矩阵的存储结构，采用三元组。
- 习 题和上机

  19、20、26

