

第四章 数组、串与广义表



一维数组(向量)




1.逻辑结构: $L = (a_1, \dots, a_{i-1}, a_i, \dots, a_n)$

2.存储结构:

```
int A[MaxSize];
```

0	a_1
1	a_2
...	...
$i-1$	a_i
i	a_{i+1}
$n-1$	a_n
...	
...	
$MaxSize-1$	

- 问题:如何求一维数组中第 i 元素的起始地址 ($\text{Loc}(a_i)=?$).



0	a_1
1	a_2
...	...
$i-1$	a_i
i	a_{i+1}
$n-1$	a_n
...	
...	
$\text{MaxSize}-1$	

- 序号为 i 的元素起始地址 = 序号为1的元素起始地址 + $(i-1) * \text{每个元素所需的空间}$

二维数组(矩阵)



1.逻辑结构

a_{00}	a_{01}	a_{02}	\dots	$a_{0,n-1}$
a_{10}	a_{11}	a_{12}	\dots	$a_{1,n-1}$
\dots	\dots	\dots	\dots	\dots
$a_{m-1,0}$	$a_{m-1,1}$	$a_{m-1,2}$	\dots	$a_{m-1,n-1}$

2.存储结构

`int A[m][n];`

在计算机内存中的存储方式



- 二维数组 $A[m][n]$ 可以看成：
 - (1) 由 m 个具有 n 个元素的线性表组成。
 - (2) 由 n 个具有 m 个元素的线性表组成。

(1) 由 m 个具有 n 个元素的线性表组成。



A_0	a_{00}	a_{01}	a_{02}	\dots	$a_{0,n-1}$
A_1	a_{10}	a_{11}	a_{12}	\dots	$a_{1,n-1}$
\dots	\dots	\dots	\dots	\dots	\dots
A_m	$a_{m-1,0}$	$a_{m-1,1}$	$a_{m-1,2}$	\dots	$a_{m-1,n-1}$

(2) 由n个具有m个元素的线性表组成。



a_{00}	a_{01}	a_{02}	\dots	$a_{0,n-1}$
a_{10}	a_{11}	a_{12}	\dots	$a_{1,n-1}$
\dots	\dots	\dots	\dots	\dots
$a_{m-1,0}$	$a_{m-1,1}$	$a_{m-1,2}$	\dots	$a_{m-1,n-1}$

- 问题:如何求二维数组Amn中第ij元素的起始地址($\text{Loc}(a_{ij})=?$).



- 二维数组的存储方法有多少种?
- 每种存储方法的特点是什么?
- 在每种存储方法中 $\text{Loc}(a_{ij})=?$



A_0	a_{00}	a_{01}	a_{02}	\dots	$a_{0,n-1}$
A_1	a_{10}	a_{11}	a_{12}	\dots	$a_{1,n-1}$
\dots	\dots	\dots	\dots	\dots	\dots
A_m	$a_{m-1,0}$	$a_{m-1,1}$	$a_{m-1,2}$	\dots	$a_{m-1,n-1}$

a_{00}	a_{01}	a_{02}	\dots	$a_{0,n-1}$	a_{10}	a_{11}	a_{12}	\dots	$a_{1,n-1}$	\dots	$a_{m-1,0}$	$a_{m-1,1}$	$a_{m-1,2}$	\dots	$a_{m-1,n-1}$
----------	----------	----------	---------	-------------	----------	----------	----------	---------	-------------	---------	-------------	-------------	-------------	---------	---------------

三维数组



- 1.逻辑结构——立体
- 2.存储结构：
 `int A[m][n][k];`

- 问题:如何求三维数组Amnl中第ijk元素的起始地址($\text{Loc}(\text{aijk})=?$).

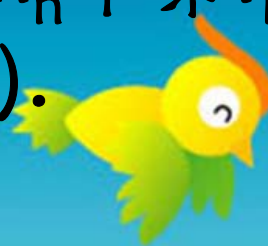


- 三维数组的存储方法有多少种?
- 每种存储方法的特点是什么?
- 在每种存储方法中 $\text{Loc}(\text{aijk})=?$

N维数组



- 问题:如何求n维数组 $A_{m_1 m_2 \dots m_n}$ 中第 $i_1 i_2 \dots i_n$ 元素的起始地址($\text{Loc}(a_{i_1 i_2 \dots i_n})=?$).



- n维数组的存储方法有多少种?
- 每种存储方法的特点是什么?
- 在每种存储方法中 $\text{Loc}(a_{i_1 i_2 \dots i_n})=?$

- 特殊矩阵



1. 对称矩阵

2. 上三角矩阵

3. 下三角矩阵

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

(a)

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

(b)

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

(c)

- 1.对称矩阵的存储及存储地址的计算方法

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



- (1)存储方法有多少种?
- (2)每种存储的存储地址的计算?

- 1.三角矩阵的存储及存储地址的计算方法



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

(b)

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

(c)

- (1)存储方法有多少种?
- (2)每种存储的存储地址的计算?

- 稀疏矩阵

- 什么是稀疏矩阵？

- 表示一个矩阵中元素的要素有哪些？

-

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



- 稀疏矩阵的存储结构
- 1、三元组表



存储单元编号	行	列	值
0	4	5	4
1	1	2	2
2	3	2	1
3	3	4	3
4	4	5	-1
5			
.
.

稀疏矩阵顺序存储类定义



```
struct RCV
{ int row,col;
  float value;
};
```

```
class SMatrix
{ RCV *item;
  int r,c,num;
```

```
public:
```

```
    SMatrix(){    item=NULL; r=0; c=0;num=0;    }
```

```
    SMatrix( RCV a[], int n, int row,int col); //a[]是一个三元组
```

```
    SMatrix& tran();
```

```
    SMatrix& tran1();
```

```
    SMatrix& plus(SMatrix& b);
```

```
    SMatrix& mult(SMatrix& b);
```

```
    void prnt();
```

```
};
```

稀疏矩阵类的构造函数



第一种形态无参数，仅创建一个空的三元组表。

第二种形态设置三元组表a，长度n及行数row、列数col四个参数，创建的三元组表由参数a、n确定，而行数、列数分别由参数row、col确定。

功能:按指定的参数分配存储空间并设置数据成员的初值。

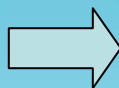
```
SMatrix:: SMatrix(RCV a[],int n, int row,int col)
{ int i;
  r=row; c=col; num=n;
  item=new RCV [num];
  for (i=0;i<num;i++) item[i]=a[i];
}
```

稀疏矩阵的转置操作



a.item

row	col	value
0	0	2
0	6	6
1	3	4
2	2	7
4	0	12
4	4	9
5	7	5



b.item

row	col	value
0	0	2
0	4	12
2	2	7
3	1	4
4	4	9
6	0	6
7	5	5

稀疏矩阵的转置操作



SMatrix& tran()

功能:返回当前矩阵对象的转置矩阵,按以下过程处理:

- (1)创建一个稀疏矩阵 X ,形成 X 的 r, c, num ,并按指定的长度分配存储空间。
- (2)按当前矩阵的列(即 X 的行)进行循环处理:对当前矩阵的每一列扫描一次三元组,找出相应的元素,交换其行号与列号并添加到转置矩阵 X 的三元组表中。
- (3)返回结果矩阵 X 。

稀疏矩阵转置算法



```
SMatrix& SMatrix::tran()
{ SMatrix& x=*new SMatrix; int i,j,k;
  x.r = c; x.c= r; x.num= num;
  x.item=new RCV[num];
  if (num>0)
  { k=0;
    for (i=0;i<c;i++ )
      for (j=0;j<num;j++ )
        if (item[j].col==i)
        { x.item[k].row=item[j].col;
          x.item[k].col=item[j].row;
          x.item[k].value=item[j].value;
          k++;
        }
      }
    return(x);
  }
```

转置算法的分析



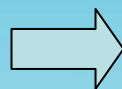
- 时间复杂度过大
- 改进策略

稀疏矩阵转置的改进策略



a.item

row	col	value
0	0	2
0	6	6
1	3	4
2	2	7
4	0	12
4	4	9
5	7	5



b.item

row	col	value
0	0	2
0	4	12
2	2	7
3	1	4
4	4	9
6	0	6
7	5	5

稀疏矩阵快速转置



row	col	value
0	0	2
0	6	6
1	3	4
2	2	7
4	0	12
4	4	9
5	7	5

k	0	1	2	3	4	5	6	7
rnum[k]	2	0	1	1	1	0	1	1
rstart[k]	0	2	2	3	4	5	5	6

稀疏矩阵快速转置



在上述算法中要进行二重循环，算法的效率比较低，如果能确定所求转置矩阵B中每一行的第一个非零元素在对应的三元组表中的位置，那么只要对三元组a进行一次扫描就可以了。为此，设置rnum和rstart两个数组。

rnum[k]表示原矩阵a的第k列中非零元的个数，
rstart[k]则指示a中第k列的第一个非零元在B的三元组表中的恰当位置。

不难看出，rnum和rstart间存在如下关系：

$$rstart[k] = rnum[k-1] + rstart[k-1]$$

稀疏矩阵快速转置



SMatrix& tran1()

功能:使用快速转置法计算并返回当前矩阵的转置矩阵, 其处理过程为:

- (1)创建一个稀疏矩阵 x , 形成 x 的 r, c, num , 并按指定的长度分配存储空间。
- (2)求当前矩阵中各列非零元的个数, 将结果存入数组 $rnum$ 。
- (3)求结果矩阵中各行起始位置, 将结果存入数组 $rstart$ 。
- (4)依次扫描当前矩阵中的三元组表, 对每一个三元组行列置换后按原列号 col 存入 x 中由 $rstart[col]$ 指示的位置, 并使其位置加1。
- (5)返回结果矩阵 x 。

稀疏矩阵快速转置



形成数组rnum;

```
for (i=0;i<c;i++) rnum[i]=0;
```

```
for (i=0;i<num;i++) rnum[item[i].col]++;
```

形成数组rstart;

```
rstart[0]=0;
```

```
for (i=1;i<c;i++)
```

```
    rstart[i]=rnum[i-1]+rstart[i-1];
```

稀疏矩阵快速转置



```
SMatrix& SMatrix::tran1()  
{SMatrix& x=*new SMatrix; int i,j;  
int rnum[100],rstart[100];  
x.r = c; x.c= r; x.num= num;  
x.item=new RCV[num];  
for (i=0;i<c;i++ ) rnum[i]=0;  
for (i=0;i<num;i++ ) rnum[item[i].col]++;  
rstart[0]=0;  
for (i=1;i<c;i++ ) rstart[i]=rnum[i-1]+rstart[i-1];  
for (i=0;i<num;i++ )  
    { j= item[i].col;  
      x.item[rstart[j]].row=j;  
      x.item[rstart[j]].col=item[i].row;  
      x.item[rstart[j]].value=item[i].value;  
      rstart[j]++;  
    }  
return(x);  
}
```

- 2.链式存储

- (1)带行指针的链式存储结构

- (2)带列指针的链式存储结构



- (3) 十字链表



row	col	val
down		right

(a)

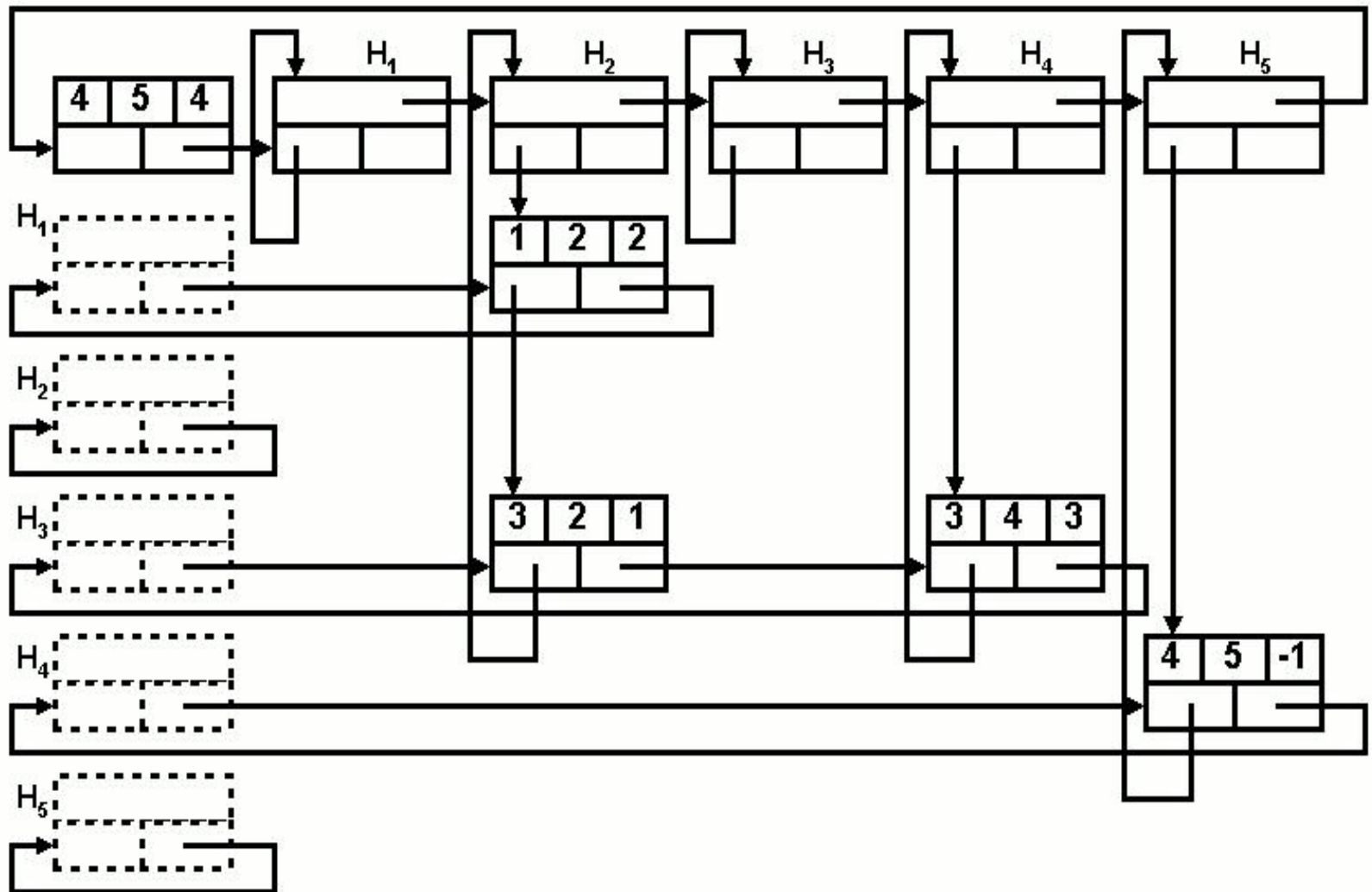
down	right

(b)

- 例子

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





广义表



• 广义表的定义

- 广义表: 是多个元素的有限序列, 一般记作 $LS=(d_1, d_2, \dots, d_n)$, 其中 d_i 可以是原子 (不可再分的元素), 也可以是广义表。
- 长度
- 表头
- 表尾
- 深度
- 空表



- 例1. 广义表 $A = (a, (), (b, (a, b)))$,
- 深度为3, 长度为3; 表头是原子a, 深度为0; 表尾是 $((), (b, (a, b)))$, 深度为3, 长度为2.

- 例2. 广义表 $B = ((a), b, c)$,
- 深度为2, 长度为3; 表头是 (a) , 深度是1, 长度是1; 表尾是 (b, c) , 深度是1, 长度是2.

- 例3. 空表的表头、表尾都是空表。

- 取表头HEAD (LS)
- 取表尾TAIL (LS)。
- 当遍历一个广义表，即按次序逐个访问广义表中的元素时，可以递归地用HEAD、TAIL操作完成。
- $A = (a, (), (b, (a, b)))$
- $HEAD(A) = a$ ，为第一个元素；
- $HEAD(TAIL(A)) = ()$ ，为第二个元素；
- $HEAD(TAIL(TAIL(A))) = (b, (a, b))$ ，为第三个元素。



- 广义表的存储
- 链式存储结构
- 具体存储方法有两种:



- 存储方法一:



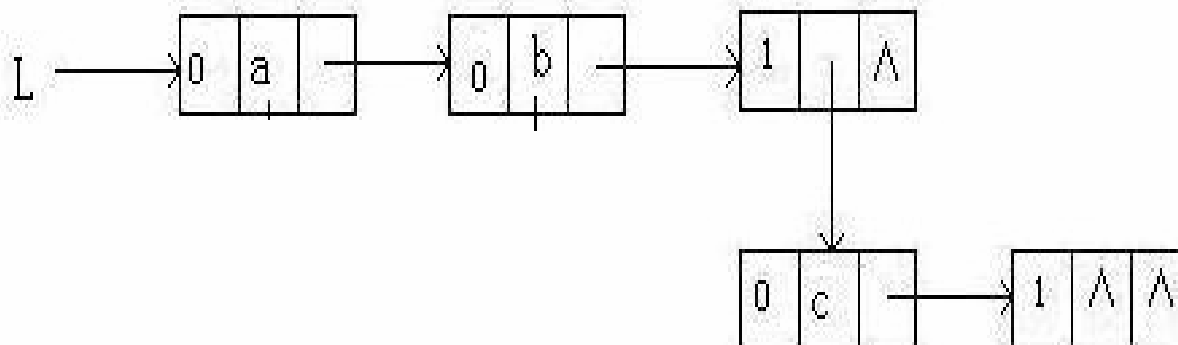
tag=1	head	tail
-------	------	------

表结点

tag=0	data	tail
-------	------	------

$()$: $L = \wedge$

$(a, b, (c, ()))$:





```
class GenList;
class GenListNode{ //结点结构
    friend class GenList;
private:
    bool tag;//FALSE表示原子元素,TRUE表示表元素
    union {
        char data;
        GenListNode *head;
    };
    GenListNode *tail;
};
class GenList{ //广义表类结构
public:    //各种广义表的操作
private:
    GenListNode *first;
};
```

• 存储方法二:



tag=1	head	tail
-------	------	------

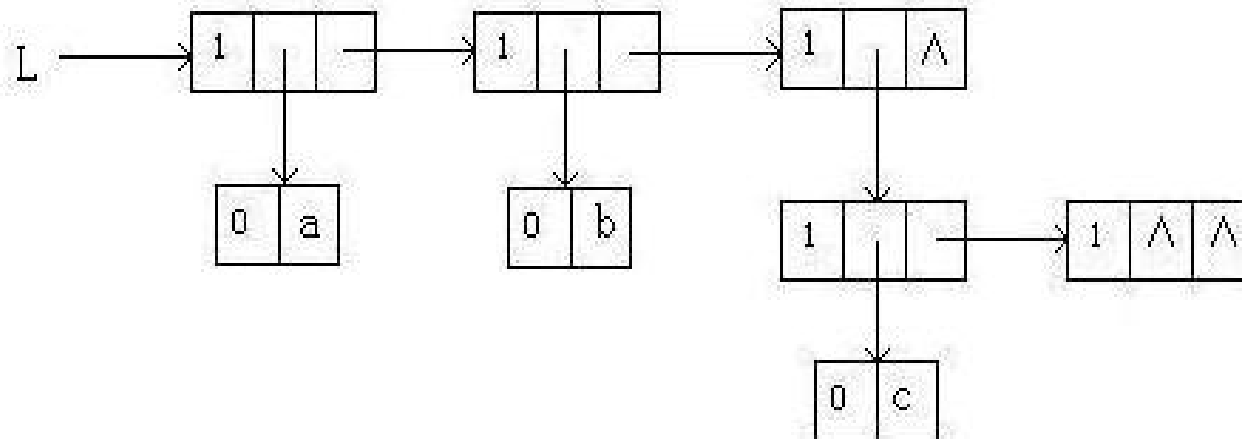
表结点

tag=0	data
-------	------

原子结点

(): L = ^

(a, b, (c, ())) :



```
class GenList;
```

```
class GenListNode{ //结点结构
```

```
    friend class GenList;
```

```
private:
```

```
    Boolean tag;//FALSE表示原子元素,TRUE表示表元素
```

```
    union {
```

```
        char data;
```

```
        struct {
```

```
            GenListNode *head;
```

```
            GenListNode *tail;
```

```
        };
```

```
    };
```

```
};
```

```
class GenList{ //广义表类结构
```

```
public:    //各种广义表的操作
```

```
private:
```

```
    GenListNode *first;
```

```
};
```



广义表算法的实现策略——递归



- 广义表的递归描述：
 - (1)把广义表看作由 n 个元素组成，每个元素可以是原子或广义表；
 - (2)把广义表看作由表头和表尾两部分组成，表头可以是原子或广义表，表尾一定是广义表。

递归算法的补充

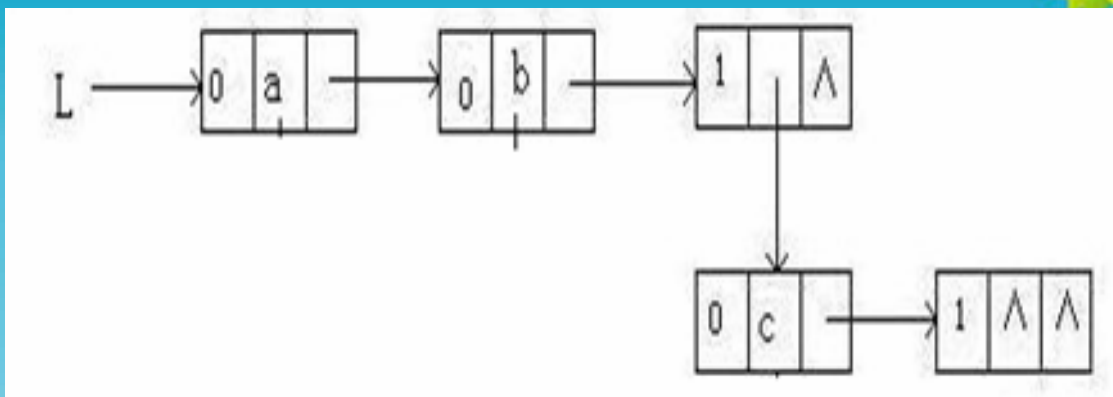


广义表操作的算法实现



- 1.求广义表的表长

• 2 广义表的深度



- (1) 定义基本问题: 空表的深度为1, 原子的深度为0。
- (2) 问题分解的方法:

- 分解问题的方法



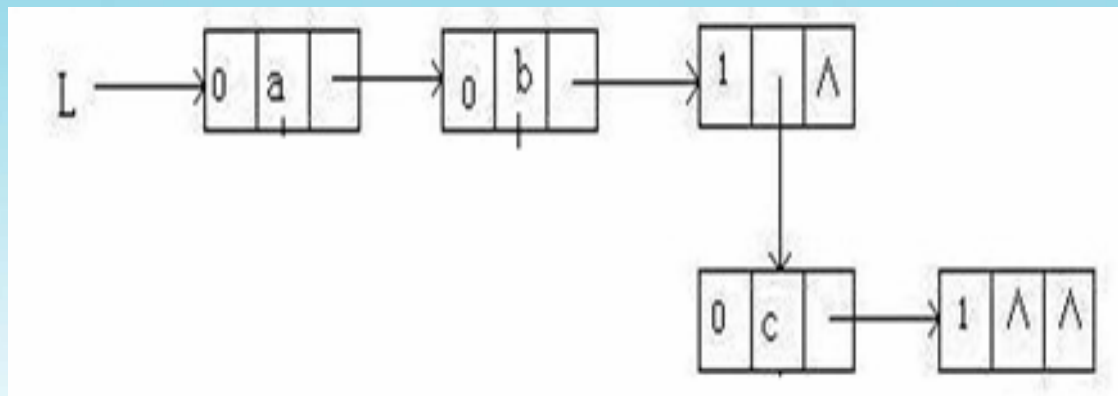
- 1、把广义表分解为 n 个元素 d_1, d_2, \dots, d_n 时:

广义表的深度 =

$\text{MAX}(d_1 \text{ 的深度} + 1, d_2 \text{ 的深度} + 1, \dots, d_n \text{ 的深度} + 1)$

程序 计算广义表的深度(用存储结构一所对应的算法)

```
int GenList::depth(GenListNode *s)
{
    int max=0;
    while (s!=NULL)
    {
        if (s->tag==TRUE)
        {
            dep=depth(s->head);
            if (dep>max) max=dep;
        }
        s=s->tail;
    }
    return(max+1);
}
```



程序 计算广义表的深度(用存储结构二所对应的算法)

```
int GenList::depth(GenListNode *s)
```

```
{
```

```
    if (!s) return(0);
```

```
    if (!s->tag) return(0);
```

```
    if (!s->head&&!s->tail) return(1);
```

```
    GenListNode *p=s; int m=0;
```

```
    while (p) {
```

```
        n=depth(p->head)
```

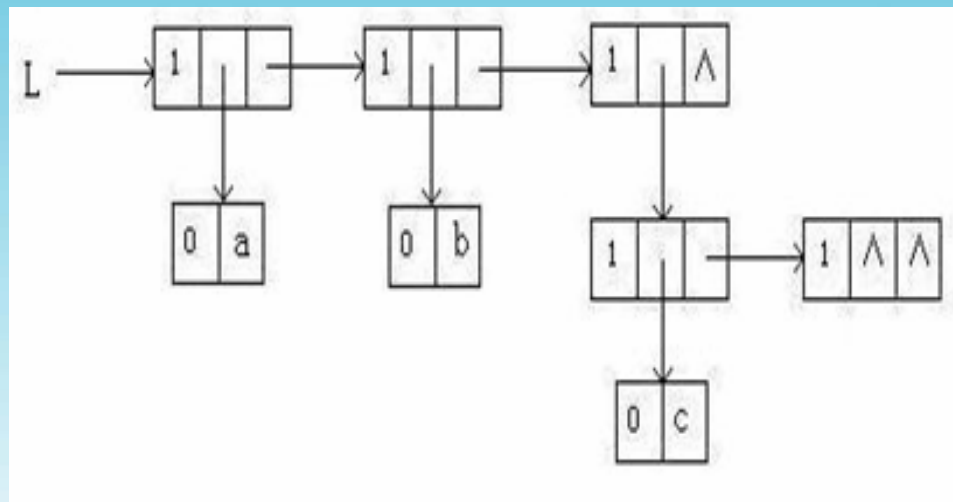
```
        if (n>m) m=n;
```

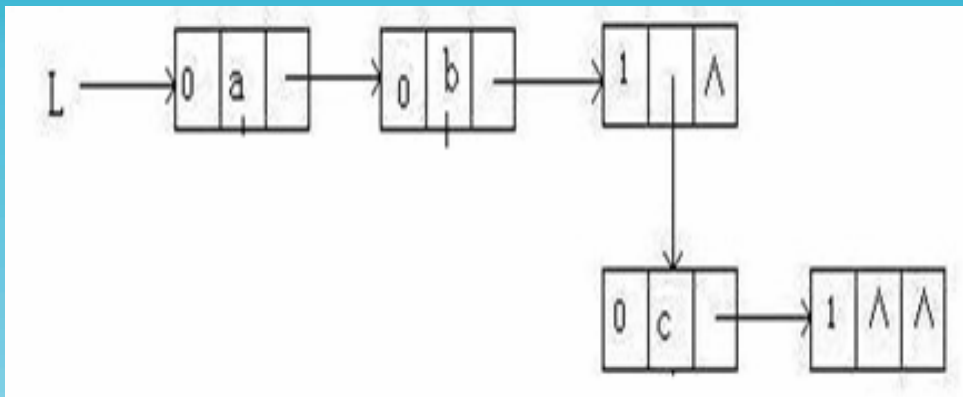
```
        p=p->tail;
```

```
    }
```

```
    return(m+1)
```

```
}
```





- 如果广义表不空，则递归复制表头和表尾；
如果为原子，则直接复制，无需递归。

存储结构一的算法



```
GenListNode *GenList::Copy(GenListNode *p)
```

```
{
```

```
    GenListNode *q=0;
```

```
    if (p!=NULL)
```

```
    { q=new GenListNode;
```

```
      q->tag=p->tag;
```

```
      if (p->tag)
```

```
          q->head=Copy(p->head);
```

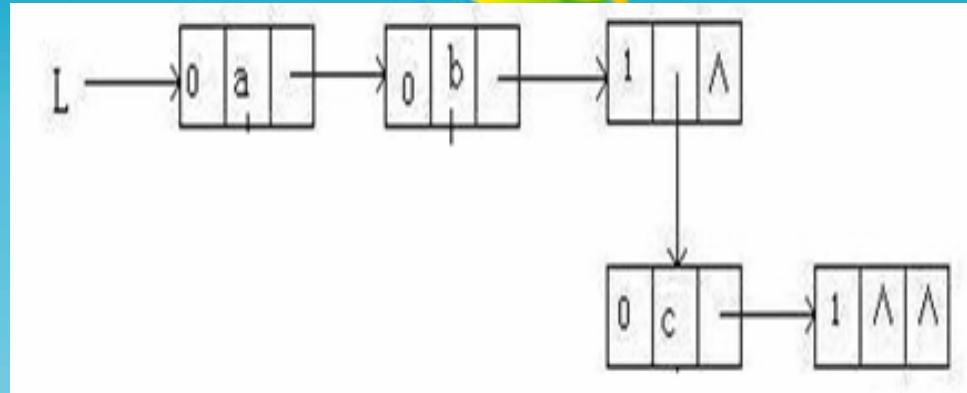
```
      else q->data=p->data
```

```
      q->tail=Copy(p->tail);
```

```
    }
```

```
    return q;
```

```
}
```

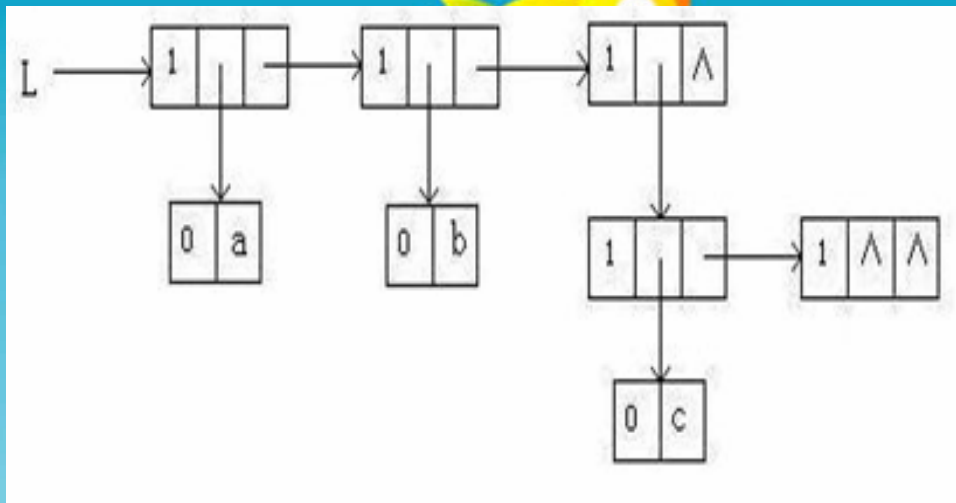


存储结构二的算法



```
GenListNode *GenList::Copy(GenListNode *p)
```

```
{  
    GenListNode *q=0;  
    if (p)  
    { q=new GenListNode;  
      q->tag=p->tag;  
      if (p->tag)  
      {  
          q->head=Copy(p->head);  
          q->tail=Copy(p->tail);  
      }  
      else q->data=p->data  
    }  
    return q;  
}
```



串





- 一、 串的基本概念
- 二、 串的顺序存储结构及操作的实现
- 三、 串的链式存储结构及操作的实现

- 目前，计算机的大量应用是解决非数值计算问题，其对象就是字符串。
- 例如在程序设计语言的编译程序中，源程序和目标程序都是字符串数据；
- 在管理信息系统中，顾客的姓名和地址、商品的名称和规格等都是字符串数据；
- 又如文字编辑、信息检索、人工智能与模式识别等领域中都是以字符串作为处理的对象。
- 正是由于字符串的重要性，所以现在大多数程序设计语言都支持串数据类型，并提供相应的串运算。



• 5.1 基本概念



- 串是由 $n(n \geq 0)$ 个字符组成的有限序列，一般记作 $s = "a_1a_2a_3 \dots a_{n-1}a_n"$ ($n \geq 0$)

(1) 空串，通常用 Φ 表示。

(2) 双引号括起来的字符序列称为串值



- 串相等

- 子串

- 定位(模式匹配)

如: `s="data structure"` `t="ta"`

串t是串s的子串且t在s中的位置为2。

- 注意: 在C/C++语言中, 串的第一个字符的序号为0。

串的操作



从实际的操作中总结出来.

主要的基本操作



1. 求子串 `substring(s,pos,len)` 返回值为串s中第pos个字符起，长度为len的字符序列
2. 插入 `insert(s,pos,t)` 在串s的第pos个字符之后插入串t
3. 删除 `delete(s,pos,len)` 从串s中删去第pos个字符起长度为len的子串
4. 定位 `position(s,t)` 若t在s中存在，则返回t在主串s中的位置，否则函数值为0
5. 替换 `replace(s,t,v)` 操作结果是以串v替换所有在串s中出现的和非空串t相等的子串
6. 判相等 `equal(s,t)` 若s和t相等，则返回true否则返回false。
7. 求长度 `length(s)` 返回s中字符的个数

基本操作的例



s1 = "Data"

s2 = "Structure"

s3 = "DataStructure"

s4 = "Data Structure"

length(s3) 返回值为13

length(s4) 返回值为14

position(s4,s2) 返回值为5

substring(s3,4,3) 返回一个字符串 "Str"

delete(s4,5,3) 执行后串s4的值为 "Data ucture"

replace(s4,s1,s2) 执行后串s4的值为 "Structure Structure"

串的抽象数据类型



```
class Str  
{ public:
```

```
    virtual int  leng()=0; //求长度
```

```
    virtual int  pos(char* t,int k=0)=0; //定位
```

```
    virtual char* subs(int pos,int len)=0; //求子串
```

```
    virtual Str& inst(int pos,char* t)=0; //插入
```

```
    virtual Str& dele(int pos,int len)=0; //删除
```

```
    virtual Str& repl(char* t,char* r)=0; //替换
```

```
};
```

• 5.2 字符串的存储结构



• 5.2.1 顺序存储

顺序存储结构类型定义

```
const maxlen = 允许的串最大长度;  
struct Tstr  
{ int curlen;  
  char str [maxlen];  
};
```

顺序串 的类 定义

```
class Str1 :public Str
{ private:
    char *str;
    int  curlen;
    int  maxlen;
public:
    Str1(int sz=81);
    Str1(char *s);
    Str1(Str1& s);
    ~Str1(){delete []str;};
    int  leng(){return curlen;};
    char* subs(int pos,int len);
    int  pos(char* t,int k=0);
    Str& inst(int pos,char* t);
    Str& dele(int pos,int len);
    Str& repl(char* t,char* r);
};
```



str1 类的构造函数

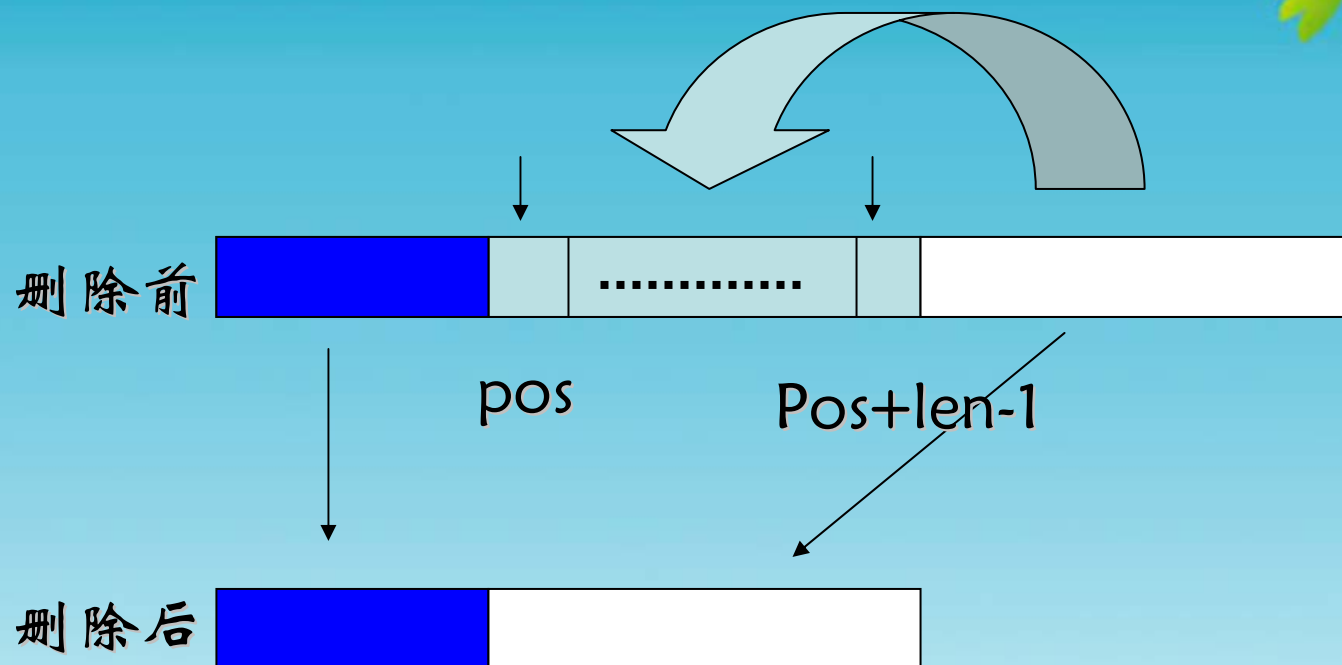
```
Str1(int sz=81) { curlen=0;  
    maxlen=sz;  
    str=new char[maxlen]; }  
Str1(char *s) { curlen=strlen(s);  
    maxlen=curlen+1;  
    str=new char[maxlen];  
    strcpy(str,s);}  
Str1(Str1& s) { curlen=s	curlen;  
    maxlen=s.maxlen;  
    str=new char[maxlen];  
    strcpy(str,s.str);}
```



删除操作



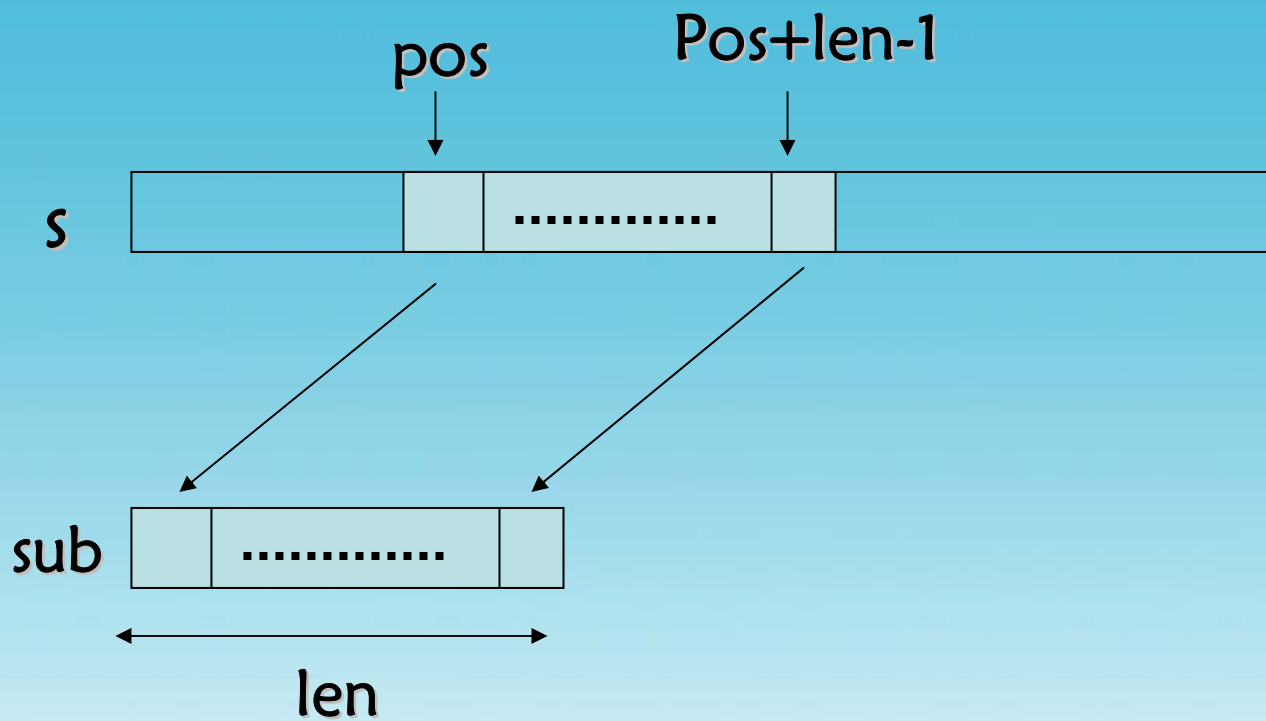
操作的含义



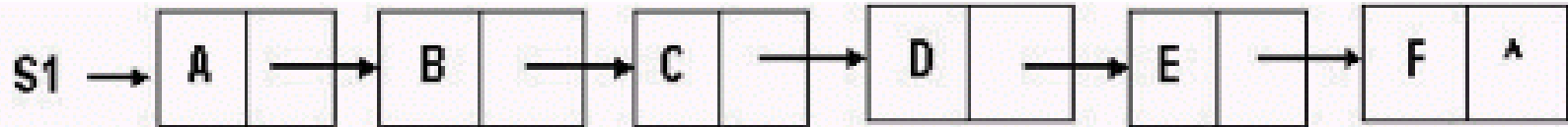
求子串操作



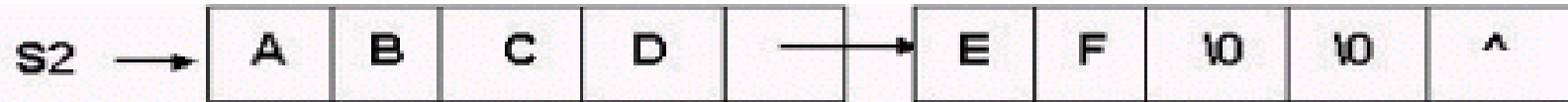
操作的含义



5.2.2 链式存储



(a)



(b)

以字符串“ABCDEF”为例

C++中的串函数及串类



- 串函数 `string.h`
- `strcpy` `strcmp` `strstr` `strcat` `strlen`等等
- 串类
- `String`、`System::String`、`basic_string`



- 作业:
- 请将C/C++语言提供的有关串操作的标准函数及类查找出来，并按以下格式编排。
- C语言中提供有关串的函数

函数名:

功能:

函数头:包括返回值，函数名及参数

函数说明:包括返回值说明，参数说明

返回值说明:

参数说明:

所在头文件:

例子:

函数名: strstr

功能: 在串中查找指定字符串的第一次出现

函数头: char *strstr(char *str1, char *str2)

函数说明:

返回值说明:

参数说明:

所在头文件: string.h

例子:

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main(void)
```

```
{ char *str1 = "Borland International", *str2 = "nation", *ptr;
```

```
  ptr = strstr(str1, str2);
```

```
  printf("The substring is: %s\n", ptr);
```

```
  return 0;
```

```
}
```



- C++语言中提供有关串的类

- 类的描述（包括类名，成员属性及成员函数）

- 各成员函数的详细说明

- 函数名：

功能：

函数头：包括返回值，函数名及参数

函数说明：包括返回值说明，参数说明

返回值说明：

参数说明：

所在头文件：

例子：

