

第5章 数组和广义表

学习目标

- ◆ 掌握数组的存储原理
- ◆ 了解特殊矩阵的存储原理
- ◆ 掌握稀疏矩阵的压缩存储
- ◆ 掌握广义表的存储结构
- ◆ 了解广义表的递归运算

5.1 数组

之前的课程中，我们已经学习过 C 语言，C 语言中明确给出了数组的定义与实现，其中也有很多针对数组的应用和操作，相信大家已经熟练掌握了数组这种数据类型，因此本节只讨论数组的逻辑结构定义，及其在内存中的存储方式。

数组是具有相同数据元素的有序集合。与线性表相似，数组中元素的个数就是数组的长度。假设现在有一个二维数组 Array，那么这个数组的逻辑结构可以表示为：

$$\text{Array} = (\text{Ele}, \text{Row_Col})$$

其中 Ele 代表数组中的一个数据， $\text{Ele} = \{a_{ij} | i=m, m+1, \dots, n, j=p, p+1, \dots, q, a_{ij} \in \text{Ele}_0\}$;

Row_Col 代表数组元素行列关系， $\text{Row_Col} = \{\text{Row}, \text{Col}\}$;

Row 代表数组元素的行间关系， $\text{Row} = \{ \langle a_{ij}, a_{i,j+1} \rangle | m \leq i \leq p, n \leq j \leq q-1, a_{ij}, a_{i,j+1} \in \text{Ele} \}$;

Col 代表数组元素的列间关系， $\text{Col} = \{ \langle a_{ij}, a_{i+1,j} \rangle | m \leq i \leq p-1, n \leq j \leq q, a_{ij}, a_{i+1,j} \in \text{Ele} \}$;

Ele₀ 为某个数据对象，m, n, p, q 均为整数。

二维数组中有 $(p-m+1)(q-n+1)$ 个数据元素。数组中的元素分别受行列关系的约束，在行关系约束中， $a_{i,j+1}$ 是 a_{ij} 的直接后继元素， $a_{i,j-1}$ 是 a_{ij} 的直接前驱元素；在列关系中， $a_{i+1,j}$ 是 a_{ij} 的直接后继元素， $a_{i-1,j}$ 是 a_{ij} 的直接前驱元素。如图 5-1 所示。

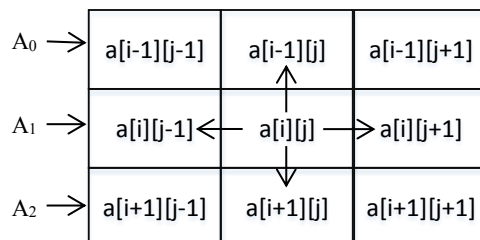


图5-1 二维数组的逻辑存储

图 5-1 中为一个 3*3 的二维数组之间的逻辑存储关系，图中水平方向和竖直方向的箭头所指的数据，分别为行关系和列关系中 a_{ij} 的前驱和后继。

数组中的元素数据类型必须相同，每个数据元素应该对应于唯一的一组下标 (i,j) 。 (m,n) , (p,q) 分别是下标 i 和 j 的下界和上界。

图 5-1 中 3*3 的二维数组，也可以看作是一个一维数组，只不过这个一维数组的元素，

也是一维数组。上图中序号 A_0 、 A_1 、 A_2 所指的一行，分别为这个一维数组的一个元素。如图 5-2 所示。

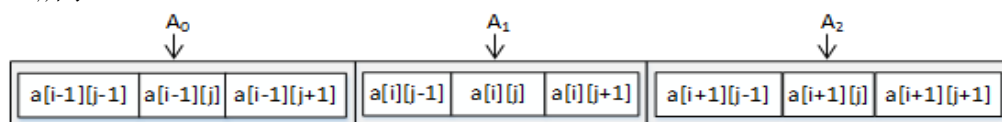


图5-2 视为一维数组的二维数组

同样的，三维数组可以看作是元素为二维数组的一维数组，或者是元素为一维数组的二维数组。由此推广，可以得到以下结论：

N 维数组可以看作是元素为 m 维数组的 $N-m$ 维数组 ($m>0, N>0, m, N \in \mathbb{N}^+$)。

在学习与开发中，如果需要使用数组，一般都是一维数组和二维数组。维数较高的数组操作复杂，容易出错，使用频率相对来说低很多，读者只需要了解即可。

对于数组，通常只有两种操作：

- (1) 将一组类型相同的元素放入一个数组，根据下标进行访问。
- (2) 给定一个数组，根据其下标对相应位置的数据元素进行修改。

虽然从逻辑上，二维数组似乎排列成了一个“面”，但在物理存储中，二维数组的存储方式跟一维数组没有差别，都呈“线”型排列。

一般来说，对数据的操作有四种，即：增加，删除，修改，查询。而数组中常用的操作只有修改和查询两种，这是因为，数组是一种地址连续的数据结构，当往其中增加一个元素，或者删除一个元素时，可能要同时对其余多数甚至全部的元素进行移位操作耗时耗力，所以使用数组保存的数据，一般不进行增删操作。又因为数组的这种特性，数组一旦建立，数据间的相对位置随即确定，所以，一般使用顺序存储结构来存储数组。

数组的存取效率是很高的，在定义的时候系统就为其分配内存，这要求数组在定义的同时显示或隐式地确定所需内存的大小，同时确定数据的存储结构。

存储结构有两种：行序优先存储和列序优先存储。

所谓行序优先，就是按行号递增的顺序一行一行地存取数据；同样地，列序优先就是按列号递增的顺序一列一列地存取数据。在下面的学习内容中我们会以行序优先举例，其内部存储可以参考图 5-4。

存储结构是由语言的内部规则确定的：Fortran 和 matlab 语言中的多维数组存储方式为列优先原则；C 语言中的多维数组存储方式为行优先原则(本书为 C 语言版，其后所讲内容，若无强调，均为行优先原则)。

数组的大小由用户来规定。用户可以根据需要使用显示规则确定数组大小：

数据类型 数组名[行][列]；

或者使用隐式规则确定数组的大小：

数据类型 数组名[] [列]={数组数据}；

在使用隐式规则确定数组大小时，行大小可以省略，但列大小不能省略。

数据在内存中所占内存空间的大小，与数据类型有关(相同的数据类型在不同的操作系统上会有所差异)。如果已知数组的首地址，或者其中某个元素的地址，那么可以根据其数据类型，快速地计算出其它各个元素在内存中存储的位置。

假设已知数组 $ARR[P][Q]$ 中单个元素所占存储单元大小为 len ，数组中元素 ARR_{ij} 的存储位置为 $LOC[i,j]$ ，那么 $ARR_{i+m,j+n}$ 的存储位置计算公式如下：

$$LOC[i+m,j+n]=LOC[i,j]+(m \times Q+n) \times len$$

其中 $0 \leq i < P$, $0 \leq j < Q$, $0 \leq (i+m) < P$, $0 \leq (j+n) < Q$, $i, j, P, Q \in \mathbb{N}^+$, $m, n \in \mathbb{N}$ 。

当数组下标的界偶(上界和下界)确定时，计算各元素在内存中存储位置的时间，仅取决于作乘法运算的时间，因此，数组中任一元素的存取所用时间都相等。我们称具有这一特

点的存储结构为随机存储结构。

下面通过一个简单的例子，对数组在内存中的存储做出说明。

首先创建一个数组，并逐一输出数组元素的内存地址。具体实现如例 5-1 所示：

例 5-1

```
1 #include <stdio.h>
2 int main()
3 {
4     //定义一个数组并初始化
5     int a[2][3] = { 0, 1, 2, 3, 4, 5 };
6     //地址输出
7     int i = 0, j = 0;
8     for (i; i < 2;i++)
9         for (j; j < 3; j++)
10             printf("%x\n", &a[i][j]);
11     return 0;
12 }
```

例 5-1 的运行结果如图 5-3 所示。



图5-3 例 5-1 运行结果

由代码可以看出，这是一个 2*3 的二维数组。数组行号的取值范围为{0,1}，列号的取值范围为{0,1,2}，数据元素的个数为 2*3=6 个。从图 5-3 的运行结果可以得出，内存为数组分配的首地址为 0x43fc68，以行序优先的方式存储。该数组的内存地址取值范围为 0x43fc68~0x43fc7f，大小为一个 int 类型数据所占内存的 6 倍，数组元素在内存中存储的方式如图 5-3 所示：

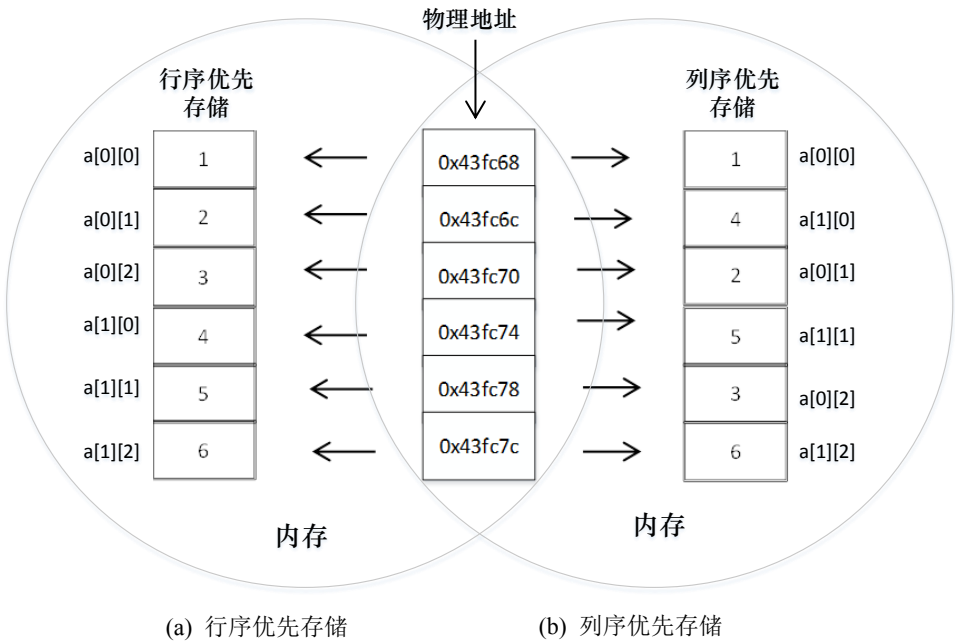


图5-4 二维数组的物理存储

5.2 矩阵的压缩存储

矩阵在诸多科学领域中都有广泛应用。在计算机领域，矩阵研究的意义不在矩阵本身，而在于研究如何存储矩阵，才能让计算机程序运算的效率更高。

计算机程序中，通常使用二维数组来存储矩阵。但是当矩阵阶数很高，并且矩阵中还有大量重复元素，或是零元素时，再使用二维数组来存储矩阵，就会造成很大的内存浪费。此时可对矩阵进行压缩存储。

根据矩阵中非零元分布的规律，可把矩阵分为特殊矩阵和稀疏矩阵。

5.2.1 特殊矩阵

特殊矩阵的元素分布有一定规律，常见的特殊矩阵分为对称矩阵、三角矩阵和对角矩阵。图 5-5 中给出了这些矩阵的样例。

$$A_n = \begin{pmatrix} a & p & b & c & h \\ p & x & e & q & k \\ b & e & d & g & l \\ c & q & g & y & n \\ h & k & l & n & z \end{pmatrix} \quad B_n = \begin{pmatrix} a & & & & \\ p & x & & & \\ b & e & d & & \\ c & q & g & y & \\ h & k & l & n & z \end{pmatrix} \quad C_n = \begin{pmatrix} a & p & b & & \\ p & x & e & q & \\ b & e & d & g & l \\ & q & g & y & n \\ & & l & n & z \end{pmatrix}$$

a-对称矩阵 b-下三角矩阵 c-对角矩阵

图5-5 特殊矩阵举例

特殊矩阵中一般存在着大量的重复元素或者 0 元素，针对这类元素采取的一般方法是：为数据相同的多个元素分配同一块存储空间；不为 0 元素分配存储空间。

1、对称矩阵

若 n 阶方阵中的元素满足： $a_{ij}=a_{ji}$ ， $1 \leq i, j \leq n$ 。则称其为对称矩阵。

对于对称矩阵，为每一对对称元素（如 a_{12} 和 a_{21} ）分配一个存储空间，这样可以将 n^2 个元素压缩存储到 $n(n+1)/2$ 个元素的空间中。

如果按照行序优先存储的存储方式，将 n 阶对称矩阵 M 下三角中的元素（ $i > j$ ）或者上三角中的元素（ $i < j$ ）存储到一维数组 $A[0 \cdots n(n+1)/2]$ 中（ $A[0]$ 不存储数据），那么 $A[k]$ 与矩阵元素 a_{ij} 之间存在着如下的一一对应关系：

$$k = \begin{cases} \frac{i(i-1)}{2} + j & i \geq j \\ \frac{j(j-1)}{2} + i & i < j \end{cases}$$

对于任意给定的一组下标 (i, j) ，均可在 A 中找到矩阵元素 a_{ij} ，反之，对所有的 $k=1, 2, \cdots, \frac{n(n+1)}{2}$ ，都能确定 $A[k]$ 中的矩阵元素在矩阵中的位置 (i, j) 。由此，称 $A[1 \cdots n(n+1)/2]$ 为 n 阶对称矩阵 A_n 的压缩存储。

数据压缩到数组中之后在数组中对应的存储位置如图 5-6 所示。

		a_{11}	a_{21}	a_{22}	a_{31}	...	a_{n1}	...	a_{nn}
$k =$	0	1	2	3	4		$\frac{n(n-1)}{2} + 1$		$\frac{n(n+1)}{2}$

图5-6 对称矩阵的压缩存储

这个存储规律对于三角矩阵同样适用，只需再添加一个存储空间用于存储常数 c 即可。

2、对角矩阵

若 n 阶方阵的元素满足：方阵中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，则称之为 n 阶对角矩阵。若方阵主对角线上下方各有 b 条次对角线，则称 b 为矩阵半带宽， $(2b+1)$ 为矩阵带宽。如图 5-7 就是一个对角矩阵，其主对角线上下各有 1 条次对角线，这个矩阵的半带宽就是 1。

$$A_n = \begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & 0 \\ & \ddots & \ddots & \ddots & \\ & & & a_{n-1,n-1} & \\ 0 & & & & a_{n,n-1} & a_{nn} \end{pmatrix}$$

图5-7 对角矩阵

半带宽为 $b(0 \leq b \leq (n-1)/2)$ 的对角矩阵, 满足 $|i-j| \leq b$ 的元素 a_{ij} 不为零, 其余元素为零。对于这种矩阵, 亦可按照某个原则将其压缩存储到一维数组上, 方式与对称矩阵类似, 读者若感兴趣, 可以自行推演。

以上讨论的特殊矩阵的压缩存储都有一定规律,在需要使用时,只需在算法中按公式映射即可实现矩阵元素的随机存储。

5.2.2 稀疏矩阵的定义

除了特殊矩阵，实际应用中还会遇到这样一种矩阵：它的非零元素较零元少，且分布没有规律，通常称这样的矩阵为稀疏矩阵。

稀疏矩阵的存储要比特殊矩阵复杂。对于稀疏矩阵并没有明确的定义，只是普遍认为：当矩阵中的非零元素 s 远远少于零元 t ($s \ll t$)，并且非零元素的分布没有规律时，我们就称这样的矩阵为稀疏矩阵。如图 5-8 中的矩阵 M 和矩阵 N 就是稀疏矩阵。

$$M = \begin{bmatrix} 0 & 11 & 21 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 92 & 0 & 0 & 0 & 0 & 85 & 0 \\ 0 & 0 & 12 & 0 & 0 & 0 & 0 \\ 0 & 26 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10 & 0 & 0 \end{bmatrix} \quad N = \begin{bmatrix} 0 & 0 & 92 & 0 & 0 & 0 \\ 11 & 0 & 0 & 0 & 26 & 0 \\ 21 & 0 & 0 & 12 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 10 \\ 0 & 0 & 85 & 0 & 10 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

图5-8 稀疏矩阵

稀疏矩阵的压缩存储与特殊矩阵的压缩存储不同：第一，其数据元素分布没有规律，无法根据下标直接获得数组中某个位置对应的元素，所示无法实现随机存储。第二，它只存储非零元素，因此，除了存储非零元素的数值(value)之外，还要存储元素在矩阵中的行列数据

(row,col)，所以我们需要可以唯一确定矩阵中一个非零元素的三元组(row,col,value)。例如图 5-8 中矩阵 M 第一行第二列的元素 11，其三元组表示形式为：(1,2,11)，表示存储的是位于一行二列，值为 11 的元素。

一个三元组可以唯一确定矩阵中的一个非零元素，表示非零元素的三元组集合及矩阵的行列数可以唯一确定一个矩阵。例如图 5-8 中的稀疏矩阵 M 可以由一个线性表

((1,2,11),(1,3,21),(2,4,2),(3,1,92),(3,6,85),(4,3,12),(5,2,26),(6,5,10))

加上(6,7)这一对行、列值来描述。

为了对稀疏矩阵进行操作，首先要了解它在计算机中的表示形式。

构造一个数组 data 来保存稀疏矩阵中每个存储非零元素的三元组，这些三元组是一种用户自定义结构体，其中的元素以 int 型为例。其定义如下：

```
typedef struct
{
    int row;                //矩阵元素所在行
    int col;                //矩阵元素所在列
    int value;              //矩阵元素保存的数据值
}Triples;
```

为了便于矩阵运算，把稀疏矩阵的三元组数据按行序优先的顺序存储结构存储，可以得到稀疏矩阵的一种压缩存储方式，我们称其为三元组顺序表示法。其定义如下：

```
#define ROWS <稀疏矩阵的行数>
#define COLS <稀疏矩阵的列数>
#define MAX_SIZE <稀疏矩阵中非零元素的最大个数>
typedef struct
{
    int rows;               //矩阵的行数
    int cols;               //矩阵的列数
    int nums;               //矩阵中非零元的数量
    Triples data[MAX_SIZE+1]; //存放三元组的数组，data[0]不用
}TSMatrix; //三元组顺序表定义
```

5.2.3 稀疏矩阵的创建

了解了稀疏矩阵的定义和数据结构类型，下面我们来创建一个稀疏矩阵。创建稀疏矩阵与线性表或者堆栈的思路都大同小异：

- (1) 定义一个 struct 来存储结点信息；
- (2) 创建头结点，初始化稀疏矩阵容量；
- (3) 将头结点地址返回。

在 5.2.2 小节稀疏矩阵的定义中已经给出了稀疏矩阵的结构体定义，这里就不再给出。创建稀疏矩阵的代码如下：

```
TSMatrix NewMatrix(int m,int n){
    //新建一个三元组表示的稀疏矩阵
    TSMatrix M;
    M.rows=m;
    M.cols=n;
    M.nums=0;
```

```
    return M;
}
```

这是一个只有容量信息的空矩阵，其非零元素个数为 0，需要往其中插入数据。

在这里使用顺序数组存储数据，这要求我们在数组中为新插入的数据寻找一个合适的位置。依据行序优先存储的方式来存储数据，设当前矩阵为 M ，要插入的三元组数据分别为 row 、 col 、 $value$ ，设保存当前进度的变量 p 。

为了判断插入元素的位置，根据行序优先的原则，首先将当前元素的行值 $M \rightarrow data[p].row$ 与三元组数组的 row 进行比较，比较可能出现以下三种结果：

- (1) 若 $row > M \rightarrow data[p].row$ ，变量 $p+1$ ，让 new 与下一个元素比较；
- (2) 若 $row < M \rightarrow data[p].row$ ，将数组元素依次往后挪一个位置，将新元素放在当前位置；
- (3) 若 $row = M \rightarrow data[p].row$ ，比较其列值 $M \rightarrow data[p].col$ 。

同样的，在比较列值时也可能出现三种结果：

- (1) 若 $col < M \rightarrow data[p].col$ ，将数组元素依次后挪一个位置，将新元素放在当前位置；
- (2) 若 $col > M \rightarrow data[p].col$ ，变量 $p+1$ ，让 new 与下一个元素比较；
- (3) 若 $col = M \rightarrow data[p].col$ ，则修改当前元素的数值 $value$ 。

不同的比较结果流向不同的分支。综上所述，给出元素的插入流程。如下图 5-9 所示：

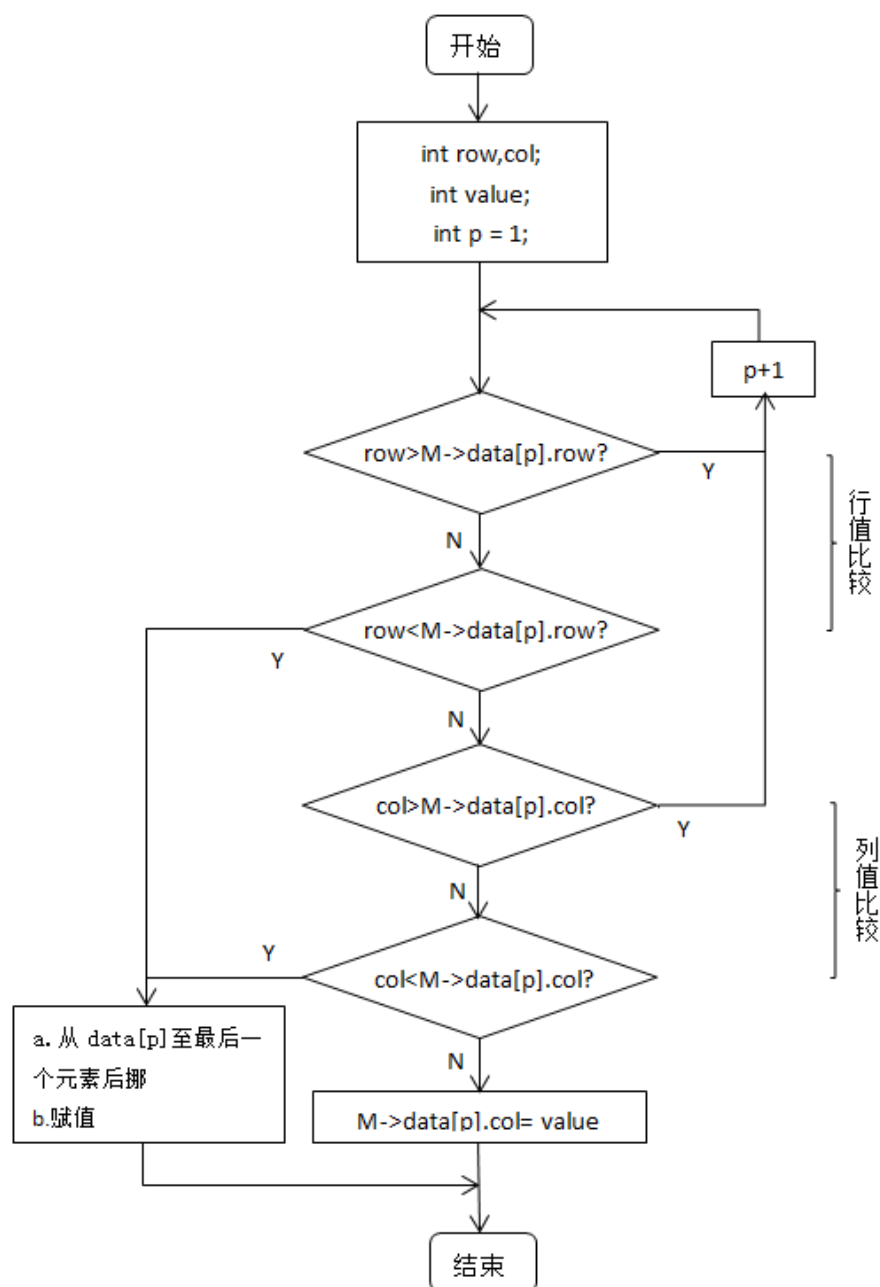


图5-9 元素插入流程图

学习了数据插入的流程，下面使用 C 语言代码来实现插入函数。

```

int InsertElem(TSMatrix *M,int row,int col,int value){
    //在三元组表示的稀疏矩阵M，第 row 行，第 col 列位置插入元素 value
    //插入成功，返回 0，否则返回-1
    int i,t,p;
    if (M->nums>=MAXSIZE) {                //当前三元组表已满
        printf("\nError:There is no space in the matrix;\n");
        return 0;
    }
    //数组越界
    if (row>M->rows||col>M->cols||row<1||col<1){
        printf("\nError:Insert position is beyond the arrange.\n");
    }
}
    
```



```

        return -1;
    }
    p=1; //标志新元素应该插入的位置
    if (M->nums==0) { //插入前矩阵 M 没有非零元素
        M->data[p].row=row;
        M->data[p].col=col;
        M->data[p].value=value;
        M->nums++;
        return 0;
    }
    for (t = 1; t <= M->nums; t++) //寻找合适的插入位置
    {
        //行比当前行大, p++
        if (row > M->data[t].row)
            p++;
        //行相等但是列比当前列大, p++
        if ((row == M->data[t].row) && (col > M->data[t].col))
            p++;
    }
    //插入前, 该位置已有数据, 则更新数值
    if (row==M->data[t-1].row && col==M->data[t-1].col) {
        M->data[t-1].value=value;
        return 0;
    }
    for (i=M->nums; i>=p; i--) { //移动 p 之后的元素
        M->data[i+1].row=M->data[i].row;
        M->data[i+1].col=M->data[i].col;
        M->data[i+1].value=M->data[i].value;
    }
    //插入新元素
    M->data[p].row=row;
    M->data[p].col=col;
    M->data[p].value=value;
    M->nums++;
    return 0;
}

```

数据结构的存储是为其应用服务。以行序为主序进行矩阵存储, 有利于进行矩阵的运算。下面将讨论矩阵的运算。在以下的讨论中, 读者可以清楚地体会到顺序存储的优点。

5.2.4 稀疏矩阵的转置

设 M 为 $m \times n$ 阶矩阵 (即 m 行 n 列), 第 i 行 j 列的元素是 a_{ij} , 即: $M=(a_{ij})_{m \times n}$ 。则定义 M 的转置为这样一个 $n \times m$ 阶的矩阵 N , 满足 $N=(a_{ji})$ (N 的第 i 行第 j 列元素是 M 的第 j 行第 i 列元素)。记作 $M^T=N$ 。例如图 5-8 中, 矩阵 M 转置之后, 就得到了矩阵 N 。

直观看来，矩阵 M 转置之后， M 中第 i 行的元素变为 N 中第 i 列的元素，原矩阵 M 中的数据元素 $a[i][j]$ 在转置后位于矩阵 N 中的 $b[j][i]$ 处。

在创建三元组数组时，我们按照行序优先存储的方式将矩阵的三元组数据存储到了数组 A 中，数组中的数据元素表示，如下图 5-10(a)所示：

	Row	Col	Value		Row	Col	Value
a[0]	1	2	11	b[0]	1	3	92
a[1]	1	3	21	b[1]	2	1	11
a[2]	2	4	2	b[2]	2	5	26
a[3]	3	1	92	b[3]	3	1	21
a[4]	3	6	85	b[4]	3	4	12
a[5]	4	3	12	b[5]	4	2	2
a[6]	5	2	26	b[6]	5	6	10
a[7]	6	5	10	b[7]	6	3	85

(a)数组 A

(b)数组 B

图5-10 稀疏矩阵及其转置矩阵的三元组存储

按照矩阵转置的规则将矩阵转置，并将三元组数据存入数组 B 中，数组 B 中的元素表示如上图 5-10(b)所示。

分析图 5-10 中的(a)和(b)，可以看出：基于三元组数组 A ，经过一系列变化，可获得行列下标互换的、有序的三元组数组 B 。

结合新的三元组数组 B ，再将原矩阵 M 的行列值数据互换，赋值给转置矩阵 N ，就获得了转置后的矩阵。

表 5-1 矩阵转置

序号	矩阵 M	步骤	矩阵 N
①	三元组 A	$N.B[i].row = M.A[i].col$ $N.B[i].col = M.A[i].row$ $N.B[i].value = M.A[i].value$	三元组 B
②	行列值 row_m 、 col_m	$N.row_n = M.col_m$ $N.col_n = M.row_m$	行列值 row_n 、 col_n

表 5-1 中给出了转置过程中数据交叉赋值的主要步骤。步骤①和②是简单的赋值过程，都不难理解，重要的是怎么实现三元组数组的有序，从而实现矩阵的转置。在这里，通常有两种方案。（这两种方案讲解参考图 5-8 和图 5-10，读者可以结合例图进行学习。）

1、方案 a

按照图 5-10(b)中三元组的次序，依次在 5-10(a)中找到相应的三元组进行转置，也就是根据矩阵 M 中的列序进行转置。每查找 M 中的一列，都要完整地扫描其三元组数组 A 。因为 A 中存储的数据行列是有序的，所以得到的 B 也是有序的。

假设此时开始扫描矩阵 M 的第二列，并将其存放到 N 中。那么实际的操作是：根据 col 值扫描数组 A ，每扫描到一个列号为 2 的元素，就依次放到数组 B 中。扫描结果如图 5-11 所示。

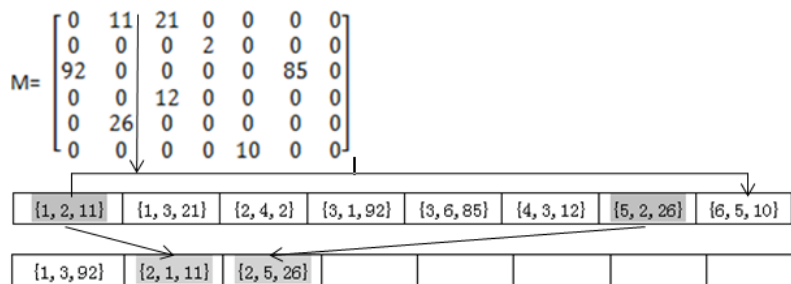


图5-11 方案 a 图示

接下来我们使用 C 语言代码来实现这种方案。

```
//矩阵转置
int sMatrixTranspose(const TSMatrix *M, TSMatrix *N)
{
    //采用三元组表示法存储表示，求稀疏矩阵 M 的转置矩阵 N
    int col, p, q;
    N->rows=M->cols;           //将 M 的行值赋给 N 作为列值
    N->cols=M->rows;           //将 M 的列值赋给 N 作为行值
    N->nums=M->nums;           //将 M 的非零元素个数赋给 N
    //如果 N 中非零元素个数不为 0，就施行图 5-11 中的转置
    if(N->nums) {
        q=1;
        for(col=1; col<=M->rows; col++)
            for(p=1; p<=M->nums; p++)
                if(M->data[p].col==col) {
                    N->data[q].row=M->data[p].col;
                    N->data[q].col=M->data[p].row;
                    N->data[q].value=M->data[p].value;
                    q++;
                }
    }
    return 0;
}
```

以上矩阵转置的代码，其时间的消耗主要在双层 for 循环中。第一层循环的时间复杂度只与矩阵 M 的列数 rows 有关，为 $O(\text{rows})$ 。第二层循环的时间复杂度与矩阵中非零元的个数 nums 有关，为 $O(\text{cols} \times \text{nums})$ 。当非零元素个数 nums 接近矩阵的容量 $\text{rows} \times \text{cols}$ 时，其时间复杂度为 $O(\text{cols}^2 \times \text{rows})$ 。即便使用经典算法逐个遍历矩阵元素进行转置，时间复杂度也不过是 $O(\text{rows} \times \text{cols})$ ，所以当矩阵非零元素个数 $\text{nums} > \text{rows}$ 时，这种算法的性能显然不能达到要求。

有没有什么方法在节省空间的同时，又能节约时间呢？在方案 a 中，转置每列数据都要扫描整个数组，转置的过程中，许多数据都被重复扫描了数次。假如能知道转置后的三元组在数组 B 中的位置，那么就可以直接将数据放进数组，如此数据重复扫描带来的时间消耗就可以被避免。

2、方案 b

为了确定 B 中元素的位置，我们需要事先知道 M 中每一列非零元素的个数，进而求得每一列的第一个元素在数组 B 中的位置，然后按照数组 A 中数据元素的顺序进行转置，将

数据放到 B 中合适的位置。

为了记录每列非零元素的个数，和每一列中第一个非零元素在数组 B 中的位置，此处设置了辅助变量数组 num 和 cpot，对个数和位置分别进行存储。num[col]表示第 col 列中非零元素的个数，cpot[col]表示第 col 列中第一个非零元素在数组 B 中的位置。对于图 5-8 中的矩阵 M，这两组辅助变量的值如下表 5-2 所示：

表 5-2 辅助变量

col	1	2	3	4	5	6	7
num	1	2	2	1	1	1	0
cpot	1	2	4	6	7	8	9

方案 b 中表述的方法通常被称为快速转置。下面使用 C 语言代码来实现方案 b：

```
//快速转置
#define COLS 7
int sMatrixFastTranspose(const TSMatrix *M, TSMatrix *N)
{
    //矩阵 N 的成员初始化
    N->rows = M->cols;
    N->cols = M->rows;
    N->nums = M->nums;
    if (N->nums) //如果矩阵中有非零元素
    {
        int col; //辅助数组的下标
        int nums[COLS] = { 0 }, cpot[COLS] = { 0 };
        int t;
        for (t = 1; t <= M->cols; t++)
            nums[M->data[t].col]++; //求矩阵 M 中每一列非零元素个数
        cpot[1] = 1;
        for (col = 2; col <= M->nums; col++)
            cpot[col] = cpot[col - 1] + nums[col - 1];
        int p, q;
        for (p = 1; p <= M->nums; p++)
        {
            col = M->data[p].col;
            q = cpot[col];
            N->data[q].row = M->data[p].col;
            N->data[q].col = M->data[p].row;
            N->data[q].value = M->data[p].value;
            cpot[col]++;
        }
    }
    return 0;
}
```

与方案 a 相比，这个方案多了两个辅助变量。从时间上看，这段代码包含三个单层 for 循环，循环次数为 cols 和 nums，因此其时间复杂度为 $O(\text{cols} + \text{nums})$ 。当矩阵中非零元素个数 nums 接近矩阵的容量 $\text{rows} \times \text{cols}$ 时，其时间复杂度与经典算法相同，为 $O(\text{cols} \times \text{rows})$ 。

稀疏矩阵的创建和稀疏矩阵转置的两种方法已经分析完毕。为了方便学习，下面给出完整的测试代码。具体如例 5-2 所示。

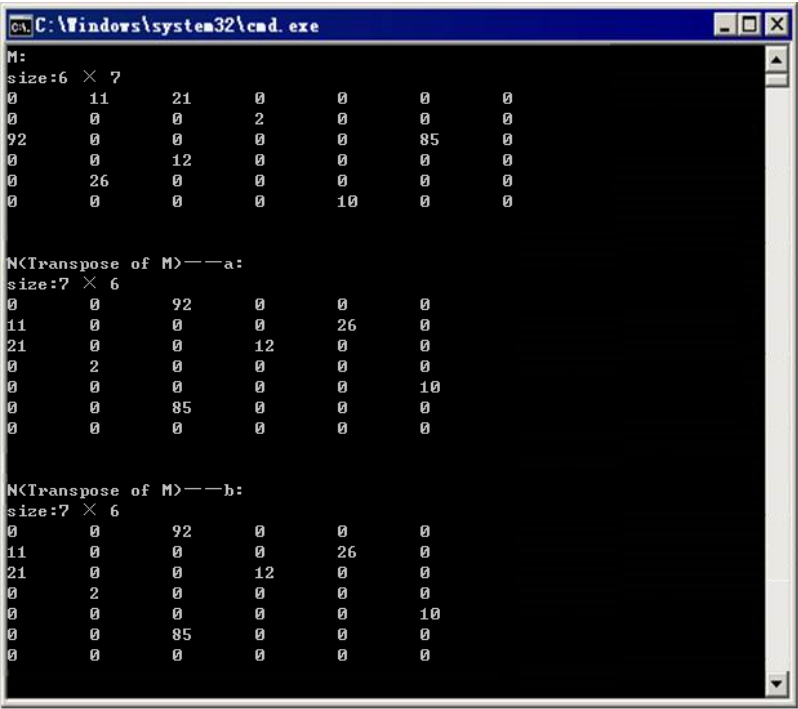
例 5-2

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #define ROWS 6
4  #define COLS 7
5  #define MAX_SIZE 100
6  int main()
7  {
8      //创建矩阵
9      TSMatrix M=NewMatrix(ROWS, COLS);
10     TSMatrix N;
11     //向矩阵中插入数据
12     InsertElem(&M, 1, 2, 11);
13     InsertElem(&M, 1, 3, 21);
14     InsertElem(&M, 2, 4, 2);
15     InsertElem(&M, 3, 1, 92);
16     InsertElem(&M, 3, 6, 85);
17     InsertElem(&M, 4, 3, 12);
18     InsertElem(&M, 5, 2, 26);
19     InsertElem(&M, 6, 5, 10);
20     //打印矩阵
21     printf("\nM:");
22     sMatrixPrint(&M);
23     //打印使用方案 a 转置的矩阵
24     sMatrixTranspose(&M, &N);
25     printf("\nN(Transpose of M)——a:");
26     sMatrixPrint(&N);
27     //打印使用方案 b 转置的矩阵
28     sMatrixFastTranspose(&M, &N);
29     printf("\nN(Transpose of M)——b:");
30     sMatrixPrint(&N);
31     return 0;
32 }

```

测试代码的运行结果如图 5-12 所示。



```
C:\Windows\system32\cmd.exe
M:
size:6 × 7
0 11 21 0 0 0 0
0 0 0 2 0 0 0
92 0 0 0 0 85 0
0 0 12 0 0 0 0
0 26 0 0 0 0 0
0 0 0 0 10 0 0

N(Transpose of M)---a:
size:7 × 6
0 0 92 0 0 0
11 0 0 0 26 0
21 0 0 12 0 0
0 2 0 0 0 0
0 0 0 0 0 10
0 0 85 0 0 0
0 0 0 0 0 0

N(Transpose of M)---b:
size:7 × 6
0 0 92 0 0 0
11 0 0 0 26 0
21 0 0 12 0 0
0 2 0 0 0 0
0 0 0 0 0 10
0 0 85 0 0 0
0 0 0 0 0 0
```

图5-12 例 5-2 运行结果

例 5-2 中的代码，实现了矩阵的创建和矩阵元素的插入。两种转置方法的代码在前文已经给出，这里只进行函数调用。

5.2.5 稀疏矩阵的十字链表表示

矩阵进行加、减、乘运算，会获得一个新矩阵。但由于新矩阵中元素数量会在很大的范围内变动，元素数量不确定，其存储空间也不确定，因此如果使用顺序结构存储三元组数据，在进行增加和删除时，会有大量的数据移动，效率非常低。学习线性表时也遇到过这种情况。线性表中我们选择使用链式存储结构来解决这个问题，同样的，在这里我们也可以使用链式存储结构来存储矩阵。

与线性表有所不同。线性表中只需存储数据域和一个指针域，而在矩阵中，首先需要三个三元组来存储数据，同时还需要有行列指针分别连接一行的数据和一列的数据。设这两个指针分别为 **right** 和 **down**，每一行的数据通过 **right** 指针与其右数据加上表头指针连接成带有表头结点的循环链表，每一列的元素通过 **down** 指针与其下数据加上表头指针连接成带有表头结点的循环链表。这样一来，数据既存在于某一个行链表，又存在于某一列链表，数据好像处于一个十字路口，所以这样的链表称为十字链表。这个用于存储数据信息的结构我们称为结点结构，其表示如图 5-13(a)所示。

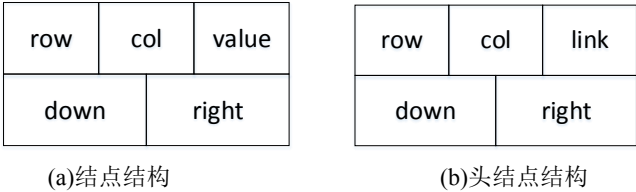


图5-13 十字链表结点结构

结点结构中提到，将数据连接成循环链表时需要表头指针，所以，除了数据，还需要为矩阵定义表头指针，指向每一行和每一列的头部，同时需要一个矩阵头指针，指向行列指针

的头部，即指向整个矩阵的头部。这就还需要额外的一种结构，即头结点结构。为了与结点结构统一，保留其中的 row 与 col，只是将值设为 0。表头指针通过结点中的 link 域连成链表。其表示如图 5-13(b)所示。

假设现有一个矩阵

$$M = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

其十字链表结构如图 5-14 所示。

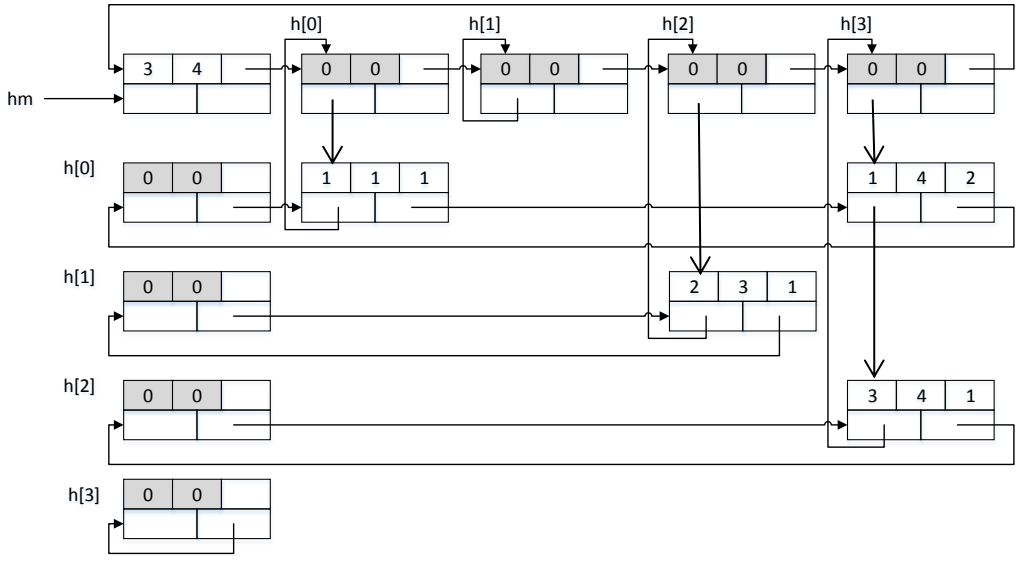


图5-14 矩阵的十字链表图示

图 5-14 中，矩阵的行列指针实际上是同一组，为了构图清晰，把行列指针拆分开来。从图中可以看出：头指针 **h[0]~h[3]**通过头结点结构的 link 域链接起来；在行指针链表中，row 域和 col 域值都为 0，使用了 right 域按行与矩阵数据组成多个循环链表；在列指针链表中，row 域和 col 域同样为 0，使用了 down 域按列与矩阵数据组成多个循环链表；头结点指针的数量为行列中的较大值。所以在结构体中使用一组链表来表示矩阵行列链表的头指针。在指向头结点链表的头结点 **hm**中，存储了矩阵的行列数值，这样一个完整的十字链表就完成了，此时只需要知道矩阵结点链表的头指针 **hm**，就可以方便地访问矩阵的数据了。

下面给出十字链表结点结构与头结点的数据类型定义，其中的元素类型以 int 型为例。

```
#define ROWS <稀疏矩阵的行数>
#define COLS <稀疏矩阵的列数>
#define Max ((ROWS)>(COLS)?(ROWS):(COLS)) //矩阵行列较大者
typedef struct mtxn
{
    int row; //行号
    int col; //列号
    struct mtxn *right, *down; //向右和向下的指针
    union
    {
        int value;
        struct mtxn *link;
    } tag;
};
```

```
} MatNode;          //十字链表类型定义
```

因为结点结构与头结点结构只有一个域（value/link）不同，所以将它们合并到一起，使用联合 union 来表示这个域。

如同前面学习到的链式结构一样，要构造一个矩阵的十字链表，首先应该创建一个头指针，即图 5-14 中的 hm 指针。其次确定其中的数据，我们可以在初始化的时候传入一组数据，根据其行列号，依次插入十字链表中。

```
//创建一个十字链表，并使用数组 a 初始化
void cmatCreate(MatNode *&hm, int a[ROWS][COLS])
{
    int i, j;
    //头结点指针
    MatNode *h[Max], *p, *q, *r;
    hm = (MatNode*)malloc(sizeof(MatNode)); //矩阵指针 hm
    hm->row = ROW;                          //初始化行
    hm->col = COL;                          //初始化列
    r = hm;
    for (i = 0; i < Max; i++)
    {
        h[i] = (MatNode*)malloc(sizeof(MatNode));
        h[i]->right = h[i];                //构成循环
        h[i]->down = h[i];
        r->tag.link = h[i];                //将头指针链接起来
        r = h[i];
    }
    r->tag.link = hm;                      //将指针重新指向矩阵头结点
    for (i = 0; i < ROWS; i++)
    {
        for (j = 0; j < COLS; j++)
        {
            if (a[i][j] != 0)              //赋值初始化
            {
                p = (MatNode*)malloc(sizeof(MatNode));
                p->row = i;
                p->col = j;
                p->tag.value = a[i][j];
                q = h[i];
                //插入行链表
                while (q->right != h[i] && q->right->col < j)
                    q = q->right;
                p->right = q->right;
                q->right = p;
                q = h[j];
                //插入列链表
                while (q->down != h[j] && q->down->row < i)
```



```

        q = q->down;
        p->down = q->down;
        q->down = p;
    }
}
}

```

在函数 `cmatCreate` 中，有一个单层 `for` 循环和一个三层循环。单层循环的时间复杂度为 $O(\text{Max})$ ；三层循环由双层 `for` 循环和两个并列的单层 `while` 循环构成，时间复杂度分别为 $O(\text{ROWS})$ ， $O(\text{COLS})$ ， $O(\text{Max})$ ，所以总时间复杂度为 $O(\text{ROWS} \times \text{COLS} \times \text{Max})$ 。假设非零元素由用户手动输入，每插入一个元素，都要寻找它在行表和列表中的插入位置，那么时间复杂度为 $O(\text{nums} \times \text{Max})$ ，这种算法对元素的输入顺序无要求。

以行序优先遍历矩阵，并将其输出。

```

//遍历输出函数
void cmatPrint(MatNode *hm)
{
    MatNode *p, *q;
    printf("行 = %d, 列 = %d\n", hm->row, hm->col);
    p = hm->tag.link;
    while (p != hm)
    {
        q = p->right;
        while (p != q)
        {
            printf("(%d,%d,%d)\n", q->row+1, q->col+1, q->tag.value);
            q = q->right;
        }
        p = p->tag.link;
    }
}

```

下面使用函数 `cmatCreate()` 创建矩阵，使用函数 `cmatPrint()` 输出矩阵。给出测试代码，如例 5-3 所示

例 5-3

```

1 int main()
2 {
3     ElemType a[ROW][COL] = { { 1, 0, 0, 2 }, { 0, 0, 1, 0 }, { 0, 0, 0, 1 } };
4     MatNode *mat;
5     cmatCreate(mat, a);
6     cmatPrint(mat);
7     return 0;
8 }

```

例 5-3 的运行结果如图 5-15 所示。

```

C:\Windows\system32\cmd.exe
行 = 3, 列 = 4
<1,1,1>
<1,4,2>
<2,3,1>
<3,4,1>
    
```

图5-15 例 5-3 运行结果

5.3 广义表

5.3.1 广义表的定义

广义表 (Lists, 简称表) 是一种非线性的数据结构, 它是线性表的推广。广义表放宽了表中对原子级元素的限制, 它可以存储线性表中的数据, 也可以存储广义表其自身结构。广义表被广泛的应用于人工智能等领域的表处理语言 LISP 语言中。

广义表的表示与线性表相似, 也是 n 个元素的有限序列。设 a_i 是广义表中的第 i 个元素, 则广义表可以表示为

$$LS = (a_1, a_2, a_3, \dots, a_n)$$

其中 n ($n \geq 0$) 为广义表 LS 中的元素个数, 表元素 a_i 可以是如线性表中的单元素, 也可以是一个子表。习惯上, 用小写字母表示单元素, 用大写字母表示子表。广义表可以为空。当广义表不为空时, 称表中第一个元素为表头 (Head), 称其余元素组成的表为表尾 (Tail)。广义表还有如下定义:

- 广义表中的数据元素有相对次序。
- 广义表的长度为表中的元素个数, 即最外层括弧包含的元素个数。
- 广义表的深度为表中所含括弧的重数。其中单元素的深度为 0, 空表的深度为 1。
- 广义表可以共享, 一个广义表可以被其它广义表共享, 这种共享广义表称为再入表。
- 广义表可以是一个递归的表, 即一个广义表的子表可以是它自身。递归广义表的深度无穷, 长度有限。

接下来通过一些举例来了解这些性质的含义。

例 5-4

$A = ()$

$B = (p)$

$C = ((x,y,z), p)$

$D = (A,B,C)$

$E = (q,E)$

在例 5-4 中:

表 A 是一个空表, 其长度为 0, 深度为 1。

表 B 只含有一个单元素 p , 其长度为 1, 深度为 1。 $\text{head}(B) = p$, $\text{tail}(B) = ()$ 。

表 C 包含单元素 p 和表元素 (x,y,z) , 其长度为 2, 深度为 2。 $\text{head}(C) = (x,y,z)$, $\text{tail}(C) = (p)$ 。

表 D 包含 A 、 B 、 C 三个子表, 长度为 3, 深度为 3。其展开表示为 $(((), (p), ((x,y,z), p)))$ 。
 $\text{head}(D) = A$, $\text{tail}(D) = (B,C)$ 。

表 E 是一个递归表, 它包含了它自身, 长度为 2, 深度为无穷。 $\text{head}(E) = q$, $\text{tail}(E) = (E)$ 。

需要注意的是, 表头即可以是单元素, 又可以是子表, 而表尾是由其余元素组成的表, 所以无论表尾中包含什么样的数据, 表尾都只可能是一个表。

如果用圆圈表示单元素，方框表示子表，则例 5-4 中五个表的图形表示如图 5-16 所示。

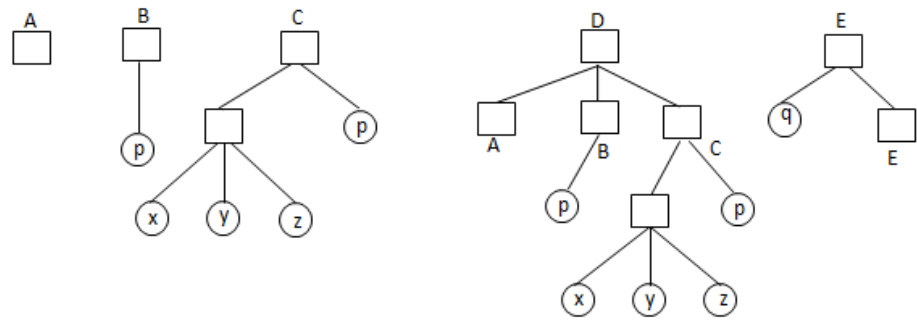


图5-16 广义表的图形表示

5.3.2 广义表的存储结构

通常使用链式存储来存储广义表。因为广义表中既可以存储单元素，又可以存储表，其数据元素结构不固定，每个元素需要的存储空间也不尽相同，难以为其分配固定的存储空间，所以不使用顺序存储。

我们已经了解到，广义表中元素分为单元素和表，所以在定义其数据结构时要加以区分。但是为了在结构上保持一致，使用下面的形式来定义广义表的结点。

tag	atom/Lists	link
-----	------------	------

其中 tag 域为标志位， atom/Lists 为数据域， link 域为指针域。

我们定义当 tag = 0 时，表示该结点为单元素结点， atom/Lists 域存储单元素 atom；当 tag = 1 时，表示该结点为子表结点， atom/Lists 域存储子表中第一个元素的地址。link 指针域用来存储与当前元素处于同一级的后继元素所在的地址。若当前元素为该级元素中的最后一个，那么其 link 域置为 NULL。

在这种存储结构中，所有的表都有一个表头指针，若该表不为空，这个表头指针总是指向列表表头（单元素结点/子表结点）。这种存储结构可以清晰地把握广义表的层次。使用 link 指针链接起来的结点，都处于同一层；第一层中结点的个数即为表的长度。图 5-17 给出了图 5-16 中广义表 A、B、D、E 的存储结构示例。

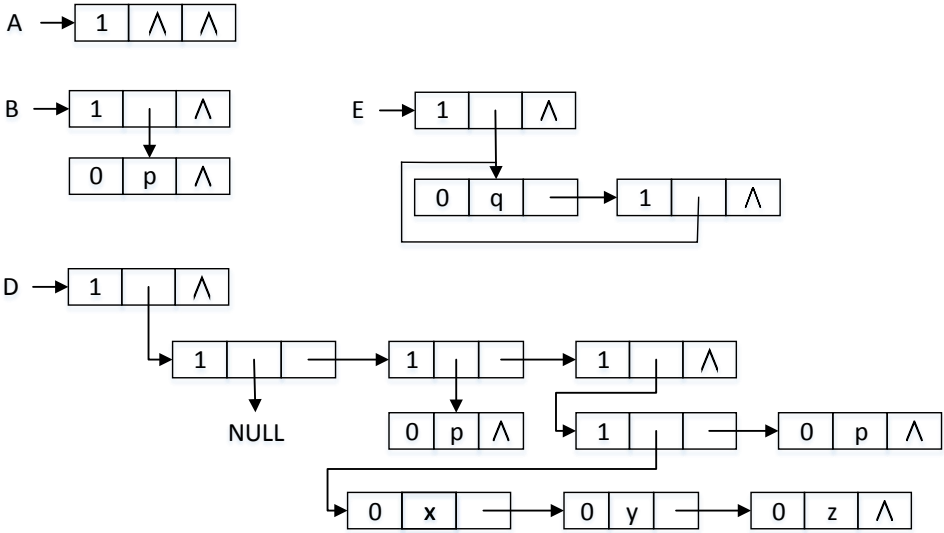


图5-17 广义表存储结构示例

综上所述，使用以下定义描述广义表结点类型。其中的数据元素以 `char` 型为例。

```
typedef struct LinkListsNode
{
    int tag; //标志位
    union
    {
        char atom; //单元素
        struct LinkListsNode *Lists; //指向子表的指针
    }value;
    struct LinkListsNode *link; //指向同一层中的后继元素
}LSNode; //广义表结点类型定义
```

5.3.3 广义表的递归运算

1、递归的定义

递归是一种编程技巧，具体指一个过程或函数在其定义或说明中有直接或间接调用自身的一种方法。作为一种算法思想在程序设计语言中应用广泛。

递归只需少量代码，便可描述出求解过程所需要的多次重复运算，大大减少了程序代码量，有效优化代码结构。但一般不提倡使用递归，因为递归执行效率很低，并且在每一层递归调用中，系统都会为局部变量和返回点开辟栈用于存储数据，递归次数过多容易造成栈溢出。所以除非一个问题在其求解分析过程中呈现明显的递归规律，使用递归相较于栈更符合思维逻辑，否则无须追求递归。

递归可以用有限的代码，定义对象的无限集合。但是没有结果的程序意义不大，我们要求算法要有零个或多个输入，至少一个输出，递归也一样。递归必须满足以下两个条件：

- (1) 子问题与原问题性质相同，且更为简单。
- (2) 存在一种可以使递归退出的简单情境，即递归必须要有终止条件。

在设计递归代码的时候，需要确定两点：一是递归公式，二是边界条件。递归公式是递归求解过程中的归纳项，用于处理原问题以及与原问题规律相同的子问题。边界条件即终止条件，用于终止递归。

递归一般用于解决三类问题：

- (1) 数据本身是按递归定义的。如斐波那契（Fibonacci）数列。
- (2) 问题求解过程是使用递归算法实现的。如 Hanoi 问题。
- (3) 数据的结构形式是按递归定义的。如二叉树、广义表等。

在本章中，我们通过学习广义表的递归运算，来了解递归，掌握递归规律。

2、求广义表的深度

5.3.1 节中提及，广义表的深度等于其括号的重数。设有一非空广义表

$$LS = (a_1, a_2, a_3, \dots, a_n)$$

其中 a_i ($i=1,2,3,\dots,n$) 为单元素或者是子表，那么 LS 的深度就是最大子表深度加 1。求广义表深度的问题，可以转化为求表中 n 个数据元素深度的问题。其中单元素深度为 0，空表深度为 1。对于子表的深度，同样以上述方法求解。其求解过程明显符合递归规律。

综上所述，归纳出递归求广义表深度的两个重要因素：

- (1) 递归公式： $\text{Depth}(LS) = \max \{\text{Depth}(a_i)\} + 1 \quad (1 \leq i \leq n, n \geq 1)$
- (2) 边界条件： $\text{Depth}(LS)=1$ 当 LS 为空表时

Depth(LS)=0 当 LS 为单元素时

根据以上分析，可以总结出求深度的基本思路：假设 ls 是一个 LSNODE* 类型的变量，当 ls->tag 为 0 时，该表只有一个单元素，函数返回其深度 0；当 ls->tag 为 1 时，表明这是一个表，根据 ls->value.Lists 来判断这是否是一个空表：如果是空表，则返回其深度 1；如果不是空表，遍历表中的每一个元素，当元素为子表时，进行递归调用求出当前层中子表深度。如果元素还有后继结点，调用递归对其后继结点进行前面所述操作。

下面用 C 语言代码来实现这个算法。

```
//递归求广义表的深度
int LSDepth(LSNODE *ls)
{
    if (ls->tag == 0)                //为单元素时返回 0
        return 0;
    int max = 0, dep;
    LSNODE *gls = ls->value.Lists;   //gls 指向第一个元素
    if (gls == NULL)                 //空表返回 1
        return 1;
    while (gls != NULL)              //不为空遍历表中的每一个元素
    {
        if (gls->tag == 1)            //元素为子表的情况
        {
            dep = LSDepth(gls);       //递归调用求出子表的深度
            if (dep > max)              //max 为同一层所求过的子表深度最大值
                max = dep;
        }
        gls = gls->link;              //使 gls 指向下一个元素
    }
    return (max + 1);                //返回表的深度
}
```

算法中 while 循环内第二层的 if 语句将 dep 的值与记录中的 max 值比较，获取较大的一个；最后一条语句“gls=gls->link;”将指针指向当前元素在本层的后继。若把循环体中的递归调用当作一条普通语句，则可以将 while 循环体中求一层数据中子表深度的最大值，视为求链表中元素最大值的问题。这类问题在以前的学习中已经了解，为了简化对递归的分析，降低学习难度，此处就不再详述。

以广义表 ls=(e,(a,b,c),z)为例分析 LSDepth()函数，求解过程如图 5-18 所示。

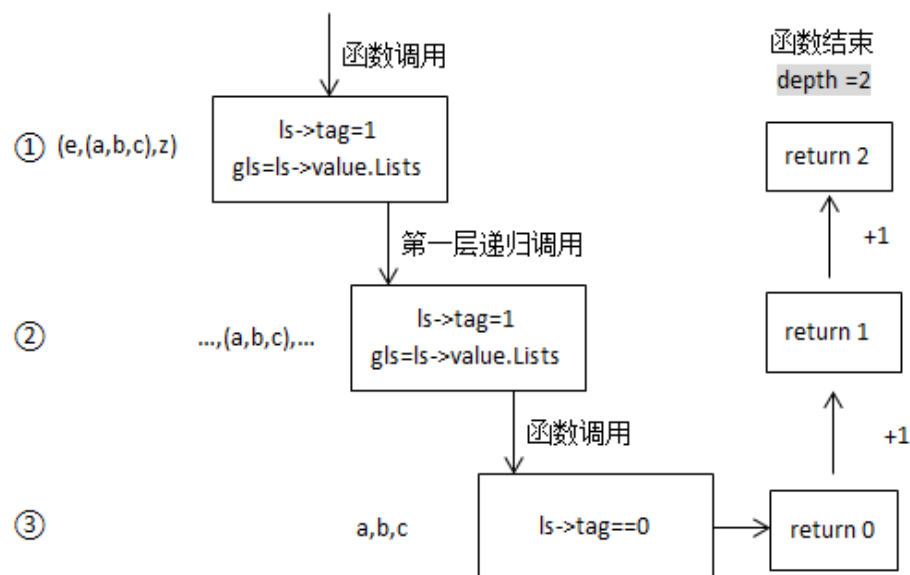


图5-18 递归调用举例

如图 5-18 所示：从函数调用处进入函数，①首先对 ls 的 tag 进行检测，其 tag 为 1，进入 $while$ 循环体；②然后对其中子表 (a,b,c) 的 tag 进行检测，其 tag 为 1，调用 $LSDepth()$ 函数进行递归；③进入函数，对子表中元素检测，发现其元素皆为单元素， tag 为 0。此时 $return$ 0，并将结果 0 返回到步骤②；步骤②获取步骤③的结果，并执行 $max+1$ ，将获得的结果 1 返回到步骤①；步骤①获取步骤③的结果，并执行 $max+1$ ，将结果 2 返回给函数调用处，函数体执行完毕。

3、输出广义表

遍历输出广义表时必然会有一对括号输出。假设要输出的广义表为 ls ，那么会有以下步骤：

- (1) 若 $ls->tag$ 为 0，表示该表为单元素，直接输出元素；
- (2) 若 $ls->tag$ 为 1，表示这是一个表，输出左括号 “(”，然后根据 $ls->value.Lists$ 判断：
 - a. 若 $ls->value.Lists$ 为 $NULL$ ，说明这是一个空表，输出右括号 “)”；
 - b. 若 $ls->value.Lists$ 不为空，进行递归调用，将 $ls->value.Lists$ 作为变量传入函数。
- (3) 在子表的结点输出完成之后，函数会回到递归调用处，然后根据 $ls->link$ 判断当前结点在本层之中是否有后继结点：
 - a. 若 $ls->link$ 为 $NULL$ ，说明本层遍历结束，返回函数调用处；
 - b. 若 $ls->link$ 不为空，说明本层中当前元素还有后继，输出一个 “，”，之后进行递归调用，将 $ls->link$ 作为变量传入函数。

下面给出遍历输出广义表的 C 语言代码。

```

//输出广义表 ls
void LSDis(LSNode *ls)
{
    if (ls->tag == 0)
        printf("%c", ls->value.atom); //输出单元素值
    else
    {
        printf("(");
        if (ls->value.Lists == NULL)
            printf(""); //空表什么也不输出
    }
}
    
```

```

        else
            LSDis(ls->value.Lists);    //递归输出子表
        printf(")");
    }
    if (ls->link != NULL)
    {
        printf(",");
        LSDis(ls->link);
    }
}

```

4、广义表的创建

假设广义表中的元素类型为 `char` 型，每个单元元素的值为除了左右括号之外的其他字符，广义表中每层数据使用 “,” 分割，广义表的元素包含在一对小括号之中，单元元素表中只有一个字符，空表中不含任何字符。如 `(e,(a,b,c),z)` 表示一个非空广义表。

显然创建广义表也是一个递归的过程：在创建表时，需要逐个地创建其中的单元元素和子表；在创建子表时，重复前面的步骤，只是将要创建的表的参数变为子表。假设要创建的广义表为 `ls`。

算法执行之时，根据定义的格式，从键盘输入一个字符串，之后算法会从头到尾扫描字符串中的每一个字符。字符可以分为四种：

(1) “(”。当遇到左括号，表示遇到了一个表，需要申请一个结点空间存放数据，并将结点中的 `tag` 置为 1，然后进行递归调用，将结点的 `ls->value.Lists` 指针地址作为参数传入函数。

(2) “)”。当遇到右括号时，表示前面的字符串已经处理完毕，应将当前传入的参数指针置空。这个传入的参数指针可能为 `ls->value.Lists` 指针地址，或 `ls->link` 指针地址。

(3) “,”。当遇到逗号时，表示当前的结点处理完毕，应该处理后继结点，此时进行递归调用，传入 `ls->link` 指针地址。

(4) 其它字符。遇到其它字符，表示的是结点中存储的数据，将 `ls->tag` 置为 0，将当前字符赋值给 `ls->value.atom`。

创建的广义表大体上可以分为三类：单元元素表、空表、非空广义表。其中单元元素表和空表，是递归的边界条件；在创建非空广义表时，递归创建子表，是其条件归纳。

下面给出创建广义表的 C 语言代码。

```

//广义表的递归创建
void LSCreate(LSNode** ls)
{
    char ch;
    ch = getchar();
    if (ch == ')')
        *ls = NULL;
    else if (ch == '(')                //当前字符为左括号时
    {
        *ls = (LSNode*)malloc(sizeof(LSNode)); //创建一个新结点
        (*ls)->tag = 1;                      //新结点作为表头结点
        LSCreate(&((*ls)->value.Lists));    //递归构造子表并链接到表头结点
    }
}

```

```

else
{
    *ls = (LSNode*)malloc(sizeof(LSNode)); //创建一个新结点
    (*ls)->tag = 0;                        //是单元素
    (*ls)->value.atom = ch;                //新结点作为单元素结点
}
ch = getchar();                          //取下一个字符
if ((*ls) == NULL);                      //串未结束，继续构造兄弟结点
else if (ch == ',')                      //当前字符为", "
    LSCreate(&((*ls)->link));            //递归构造兄弟结点
else                                     //没有兄弟了，将兄弟指针置为 NULL
    (*ls)->link = NULL;
return;                                  //返回
}

```

LSCreate()函数在创建表时，如果是单元素，则代码自顶向下执行一遍；如果创建表，在代码执行过程中发生一次递归调用。求广义表长度的运算是对于广义表的一层进行操作，不涉及递归，这里直接给出代码，以展示运算结果，作为与广义表深度的对比学习。

完整的测试代码如下。

例 5-5

```

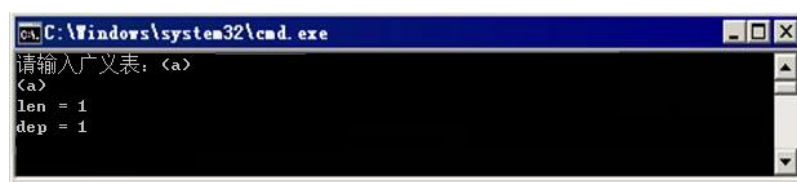
1 //求广义表的长度
2 int LSLength(LSNode *ls)
3 {
4     int n = 0;
5     if (ls->tag == 0)                //为单元素时返回 0
6         return 1;
7     ls = ls->value.Lists;            //gls 指向广义表的第一个元素
8     while (ls != NULL)
9     {
10         n++;                        //累加元素个数
11         ls = ls->link;
12     }
13     return n;
14 }
15 //测试代码主函数
16 int main()
17 {
18     LSNode *ls;
19     printf("请输入广义表: ");
20     LSCreate(&ls);
21     LSDis(ls);
22     int len = LSLength(ls);
23     printf("\nlen = %d\n", len);
24     int dep = LSDepth(ls);
25     printf("dep = %d\n", dep);

```



```
26     return 0;
27 }
```

从键盘分别输入单元素表(a)、空表()、广义表(e,(a,b,c),z)，代码执行结果依次如下图 5-19 所示。



(a) 单元素表(a)



(b) 空表()



(c) 非空广义表(e,(a,b,c),z)

图5-19 例 5-5 执行结果

对比图 5-19 中的执行结果可以看到，相比于非空广义表，空表在输入与输出之间多一个回车换行。这是因为，每调用一次 LSCreate()代码，要执行两次 getchar()函数。创建空表时输入了两个有效字符“(”、“)”以及一个回车符共三个字符，所以需要再吸收一个回车符才能执行下面的语句。

为了解决这个问题，我们可以在函数中设置一个参数 flag，用来记录当前创建的层数，也就是当前操作的深度。如果层数为 0，则不再执行后面的代码。改良后的算法在下面给出。

```
void LSCreate(LSNode** ls,int flag)
{
    char ch;
    ch = getchar();
    if (ch == '\n')
        *ls = NULL;
    else if (ch == '(') //当前字符为左括号时
    {
        *ls = (LSNode*)malloc(sizeof(LSNode)); //创建一个新结点
        (*ls)->tag = 1; //新结点作为表头结点
        //递归构造子表并链接到表头结点
        LSCreate(&((*ls)->value.Lists),flag+1);
    }
    else if (ch == '\n')
    {
```

```

        (*ls)->link = NULL;
        return;
    }
    else
    {
        *ls = (LSNode*)malloc(sizeof(LSNode)); //创建一个新结点
        (*ls)->tag = 0;                        //是单元素
        (*ls)->value.data = ch;                //新结点作为单元素结点
    }
    if (flag == 0)                             //若当前层为 0
    {
        (*ls)->link = NULL;                  //将结点 link 域置空再返回
        return;
    }
    ch = getchar();                            //取下一个字符
    if ((*ls) == NULL);                       //串未结束，继续构造后继结点
    else if (ch == ',')                       //当前字符为","
        LSCreate(&((*ls)->link), flag);      //递归构造后继结点
    else
        (*ls)->link = NULL;                  //若无后继，将 link 指针置为 NULL
    return;                                    //返回
}

```

在主函数中调用 LSCreate(&ls,0)来创建广义表，将层数标记 flag 输出，可以看出递归调用创建广义表的特征：结点从最深层开始创建，逐层返回。代码执行结果如图 5-20 所示。



(a) 空表()



(b) 单元素表(a)



(c) 非空广义表(e,(a,b,c),z)

图5-20 LSCreate()改良结果

5.4 本章小结

本章主要介绍了几种数据结构在内存中的存储。首先介绍了数组的基本存储形式，然后介绍了特殊矩阵和稀疏矩阵的压缩存储，以及稀疏矩阵的十字链表表示方法，即链式结构存储，最后介绍了广义表的基本知识和广义表的存储结构，以及广义表的递归运算。

通过本章的学习，读者应对数据结构在内存中的存储有初步了解，并能掌握文中所讲的几种数据类型的存储方式。亦要对广义表的概念有所把握，通过广义表的递归运算，学习递归算法的结构和基本使用方法。

【思考题】

- 1、请简述一下稀疏矩阵的十字链表存储结构。
- 2、请简述一下广义表的递归运算思想。

扫描右方二维码，查看思考题答案！

