第 5 章　数组和广义表

5.1　数　　组

1. 数组的基本概念

数组是一个元素可直接按序号寻址的线性表：

$$a=(a_0,a_1,\cdots,a_{m-1})$$

若 $a_i(i=0,1,\cdots,m-1)$是简单元素，则 a 是一维数组；当一维数组的每个元素 a_i 本身又是一个一维数组时，则一维数组扩充为二维数组。同样道理，当 a_i 是一个二维数组时，则二维数组扩充为三维数组。以此类推，若 a_i 是 $k-1$ 维数组，则 a 是 k 维数组。

可以看出，在 n 维数组中，每个元素受 n 个线性关系的约束($n\geqslant 1$)，若它在第 1～n 个线性关系中的序号分别为 $i_1,i_2,\cdots,i_n$，则称它的下标为 $i_1,i_2,\cdots,i_n$。如果数组名为 a，则记下标为 $i_1,i_2,\cdots,i_n$ 的元素为 $a_{i_1,i_2,\cdots,i_n}$。

从上面的定义可以看出，如果一个 $n(n>0)$维数组的第 i 维长度为 b_i，则此数组中共含有 $\prod_{i=1}^{n}b_i$ 个数据元素，每个元素都受 n 个关系的约束，就其单个关系而言，这 n 个关系都是线性关系。

数组的基础操作如下：

```
ElemType &operator()(int sub0,…)
```

初始条件：数组已存在。

操作结果：重载函数运算符。

2. 数组的顺序存储结构

设 n 维数组共有 $m\left(=\prod_{i=1}^{n}b_i\right)$个元素，数组存储的首地址为 base，数组中的每个元素需要 size 个存储单元，则整个数组共需 $m\cdot$size 个存储单元。为了存取数组中某个特定下标的元素，必须确定下标为 $i_1,i_2,\cdots,i_n$ 的元素的存储位置。实际上就是把下标 $i_1,i_2,\cdots,i_n$ 映射到[0，$m-1$]中的某个数 Map($i_1,i_2,\cdots,i_n$)，使得该下标所对应的元素值存储在以下位置：

$$\text{Loc}(i_1,i_2,\cdots,i_n)=\text{base}+\text{Map}(i_1,i_2,\cdots,i_n)\cdot\text{size}$$

其中，Loc($i_1,i_2,\cdots,i_n$)表示下标为 $i_1,i_2,\cdots,i_n$ 的数组元素的存储地址。可见，如果已经知道数组的首地址，要确定其他元素的存储位置，只需求出 Map($i_1,i_2,\cdots,i_n$)即可。具体如下：

$$\begin{aligned}\text{Map}(i_1,i_2,\cdots,i_n)&=i_1b_2b_3\cdots b_n+i_2b_3\cdots b_n+\cdots+i_{n-1}b_n+i_n\\&=\sum_{j=1}^{n-1}i_j\prod_{k=j+1}^{n}b_k+i_n=\sum_{j=1}^{n}c_ji_j\end{aligned}$$

进一步可得如下公式：

$$\begin{aligned}\text{Loc}(i_1, i_2, \cdots, i_n) &= \text{base} + (i_1 b_2 b_3 \cdots b_n + i_2 b_3 \cdots b_n + \cdots + i_{n-1} b_n + i_n)\text{size} \\ &= \text{base} + \sum_{j=1}^{n} c_j i_j \cdot \text{size}\end{aligned}$$

其中，$c_n = 1$，$c_{j-1} = b_j c_j$，$1 < j \leqslant n$。

n 维数组 $a[b_1][b_2]\cdots[b_n]$的列优先映射函数为

$$\begin{aligned}\text{Map}(i_1, i_2, \cdots, i_n) &= i_1 + b_1 i_2 + \cdots + b_1 b_2 \cdots b_{n-2} i_{n-1} + b_1 b_2 \cdots b_{n-1} i_n \\ &= i_1 + \sum_{j=2}^{n} \left(\prod_{k=1}^{j-1} b_k \right) i_j = \sum_{j=1}^{n} c_j i_j\end{aligned}$$

$$\begin{aligned}\text{Loc}(i_1, i_2, \cdots, i_n) &= \text{base} + (i_1 + b_1 i_2 + \cdots + b_1 b_2 \cdots b_{n-2} i_{n-1} + b_1 b_2 \cdots b_{n-1} i_n)\text{size} \\ &= \text{base} + \sum_{j=1}^{n} c_j i_j \text{size}\end{aligned}$$

其中，$c_1 = 1$，$c_{j+1} = b_j c_j$，$1 \leqslant i < n$。

5.2 矩　　阵

1. 矩阵的定义和操作

矩阵与二维数组有很多相似之处，一般用如下的二维数组来描述一个 $m \times n$ 矩阵 $\boldsymbol{a}_{m \times n}$：

ElemType $a[m][n]$；

矩阵中的元素 $a(i, j)$对应于二维数组的元素 $a[i-1][j-1]$。这种形式要求使用数组的下标[][]来指定每个矩阵元素。这种变化降低了应用代码的可读性，也增加了出错的概率。可以通过定义一个类 Matrix 来克服这个问题。在 Matrix 类中，将矩阵元素按照行优先次序存储到一个一维数组 elems 中，另外通过重载函数运算符()实现使用(i, j)来指定每个元素并且根据矩阵的约定，其行列下标值都是从 1 开始。

矩阵的基础操作如下：

1) int GetRows() const

初始条件：矩阵已存在。

操作结果：返回矩阵行数。

2) int GetCols() const

初始条件：矩阵已存在。

操作结果：返回矩阵列数。

3) ElemType &operator()(int i, int j)

初始条件：矩阵已存在。

操作结果：重载函数运算符。

2. 特殊矩阵

如果值相同的元素或零元素在矩阵中按一定的规律分布，这样的矩阵称为特殊矩阵，可以用特殊方法进行存储和处理，以便提高空间和时间效率。下面首先介绍相关的

几个概念。

(1) 方阵(square matrix)：是行数和列数相同的矩阵。下面介绍的特殊矩阵都是方阵。

(2) 对称(symmetric)矩阵：$\boldsymbol{a}$ 是一个对称矩阵当且仅当对于所有的 i 和 j 有 $a(i, j)=a(j, i)$，如图 1.5.1(a)所示。

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

(a) 对称矩阵　(b) 三对角矩阵　(c) 下三角矩阵　(d) 上三角矩阵

图 1.5.1　特殊矩阵示例

(3) 三对角(tridiagonal)矩阵：$\boldsymbol{a}$ 是一个三对角矩阵当且仅当 $|i-j|>1$ 时有 $a(i, j)=0$(或常数 c)，如图 1.5.1 (b)所示。

(4) 下三角(lower triangular)矩阵：$\boldsymbol{a}$ 是一个下三角矩阵当且仅当 $i<j$ 时有 $a(i, j)=0$(或常数 c)，如图 1.5.1(c)所示。

(5) 上三角(upper triangular)矩阵：$\boldsymbol{a}$ 是一个上三角矩阵当且仅当 $i>j$ 时有 $a(i, j)=0$(或常数 c)，如图 1.5.1(d)所示。

特殊矩阵的基础操作如下：

1) int GetOrder() const

初始条件：特殊矩阵已存在。

操作结果：返回特殊矩阵阶数。

2) ElemType &operator()(int row, int col)

初始条件：特殊矩阵已存在。

操作结果：重载函数运算符。

3. 稀疏矩阵

如一个矩阵中有许多元素为 0，则称该矩阵为稀疏(sparse)矩阵。对每个非零元素，用三元组(行号，列号，元素值)来表示，这样每个元素的信息就全部记录下来了。

各非零元素对应的三元组及其行列数可唯一确定一个稀疏矩阵。

稀疏矩阵具有如下的一些基本操作：

1) int GetRows() const

初始条件：稀疏矩阵已存在。

操作结果：返回稀疏矩阵行数。

2) int GetCols() const

初始条件：稀疏矩阵已存在。

操作结果：返回稀疏矩阵列数。

3) int GetNum() const

初始条件：稀疏矩阵已存在。

操作结果：返回稀疏矩阵非零元素个数。

4) bool Empty() const

初始条件：稀疏矩阵已存在。

操作结果：如稀疏矩阵为空，则返回 true，否则返回 false。

5) bool SetElem(int r, int c, const ElemType &v)

初始条件：稀疏矩阵已存在。

操作结果：设置指定位置的元素值。

6) bool GetElem(int r, int c, ElemType &v)

初始条件：稀疏矩阵已存在。

操作结果：求指定位置的元素值。

稀疏矩阵包含三元组顺序表与十字链表两种存储结构，下面分别加以介绍。

(1) 三元组顺序表。以顺序表存储三元组表，可得到稀疏矩阵的顺序存储结构——三元组顺序表。在三元组顺序表中，用三元组表表示稀疏矩阵时，为避免丢失信息，增设了一个信息元组，形式如下：

（行数，列数，非零元素个数）

将它作为三元组表的第一个元素。

*(2) 十字链表：稀疏矩阵中的非零元素个数或位置在操作过程中经常发生变化时，就不适合采用三元组顺序表来表示稀疏矩阵的非零元素了，这时可采用链式存储方式表示稀疏矩阵。由于稀疏矩阵的链式存储表示最终形成了一个十字交叉的链表，所以这种存储结构叫作十字链表。十字链表是一种特殊的链表，它不仅可以用来表示稀疏矩阵，事实上，一切具有正交关系的结构，都可用十字链表存储。在此，我们基于稀疏矩阵来介绍十字链表的相关内容。

在稀疏矩阵的十字链表表示中，每个非零元素对应十字链表中的一个节点，各节点的结构如图 1.5.2 所示。

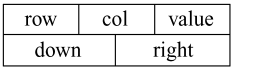

图 1.5.2　十字链表节点结构

节点中的 row、col、value 分别记录各非零元素的行号、列号和元素值，down、right 是两个指针，分别指向同一列和同一行的下一个非零元素节点。这样，每个非零元素既是某个行链表中的一个节点，又是某个列链表中的一个节点。

5.3 广　义　表

1. 基本概念

广义表通常简称为表，是由 $n(n\geqslant 0)$个表元素组成的有限序列，记作

$$GL = (a_1, a_2, \cdots, a_n)$$

其中，GL 为表名，n 为表的长度，$n=0$ 时为空表。a_i为表元素($i = 1, 2, \cdots, n$)，简称为元素，它可以是单个数据元素(称为原子元素，或简称为原子)，也可以是满足本定义的广义表(称为子表元素，或简称为子表)。

广义表具有如下基本操作。

1) GenListNode<ElemType> * First() const

初始条件：广义表已存在。

操作结果：返回广义表的第一个元素。

2) GenListNode<ElemType> * Next(GenListNode<ElemType> * elemPtr) const

初始条件：广义表已存在，elemPtr 指向广义表的元素。

操作结果：返回 elemPtr 指向的广义表元素的后继。

3) bool Empty() const

初始条件：广义表已存在。

操作结果：如广义表为空，则返回 true，否则返回 false。

4) void Push(const ElemType &e)

初始条件：广义表已存在。

操作结果：将原子元素 e 作为表头加入广义表最前面。

5) void Push(GenList<ElemType> &subList)

初始条件：广义表已存在。

操作结果：将子表 subList 作为表头加入广义表最前面。

6) int Depth() const

初始条件：广义表已存在。

操作结果：返回广义表的深度。

*** 2. 广义表的存储结构**

广义表的链式存储可以有多种形式，具体使用时，应根据具体问题的要求选择不同的存储结构。下面给出一种常用的借助引用数的链式存储结构——引用数法广义表。在这种方法中，每个表节点由 3 个成员组成，如图 1.5.3 所示。

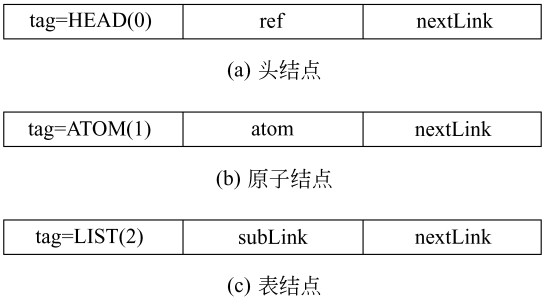

图 1.5.3　引用数法广义表节点结构

上面引用数法广义表节点结构中，nextLink 用于存储指向后继节点的指针，这样将广义表的各元素连接成一个链表，为方便起见还在链表的前面加上头节点，这样广义表的节点可分 3 种类型。

(1) 头节点，用标志 tag＝HEAD 标识，ref 用于存储引用数，子表的引用数表示能访问此子表的广义表或指针个数。

(2) 原子节点，用标志 tag＝ATOM 标识，原子元素用原子节点存储，atom 用于存储原子元素的值。

(3) 表节点,用标志 tag=LIST 标识,subLink 用于存储指向子表头节点的指针。

这种存储结构的广义表具有如下特点。

(1) 广义表中的所有表,不论是哪一层的子表,都带有一个头节点,空表也不例外,其优点是便于操作。特别是当一个广义表被其他表共享的时候,如果要删除这个表中的第一个元素,则需删除此元素对应的节点。如果广义表的存储中不带表头节点,则必须检测所有的子表节点,逐一修改那些指向被删节点的指针,这样修改既费时,又容易发生遗漏。如果所有广义表都带有表头节点,在删除表中第一个表元素所在节点时,由于头节点不会发生变化,从而也就不用修改任何指向该子表的指针。

(2) 表中节点的层次分明。所有位于同一层的表元素,在其存储表示中也在同一层。

(3) 可以很容易计算出表的长度。从头节点开始,沿 nextLink 链能够找到的节点个数即为表的长度。

在释放广义表节点时,如直接在物理上释放广义表节点,这时由于广义表具有元素共享性,可能还有其他广义表要引用被释放广义表的节点,因此在逻辑上释放广义表并不表示一定要在物理上释放节点。为了判断是否能在物理上释放一个广义表节点,可用“引用数”识别,引用数就是能访问广义表的广义表或指针个数。由于头节点的数据成分是空闲的,正好用来存放引用数。在释放广义表时,首先让引用数自减 1,如果引用数为 0,则在物理上释放节点。

虽然用头节点和引用数解决了表共享的释放问题,但对于递归表,引用数不会为 0,这样就无法实现释放递归表的目的,因此如果不改变思想,递归表会出现问题;进而可以这样来解决释放广义表的问题,建立一个全局广义表使用空间表对象,专门用于收集指向广义表中节点的指针,用析构函数在程序结束时统一释放所有广义表节点,这样实现时,不再需要引用数,头节点的数据部分为空,这样的广义表称为使用空间法广义表。