第 5 章　数组与广义表

教学要点

数组是一种基本而重要的数据集合类型，一个数组是由一组具有相同数据类型的数据元素组成的集合，数据元素按次序存储于一个地址连续的内存空间中。数组元素的类型可以是简单的基本类型，也可以是复杂的自定义类型。数组是其他数据结构实现顺序存储的基础，一维数组可以看作是一个顺序存储结构的线性表，二维数组则可视为数组的数组。一般采用二维数组存储矩阵，但这种方法存储特殊矩阵和稀疏矩阵的效率较低，需采用一些特殊方法进行压缩存储。

线性表结构具有弹性，既可以是简单的数组，也可以扩展为复杂的数据结构—广义表。

本章介绍数组、稀疏矩阵和广义表的基本概念，并详细讨论稀疏矩阵和广义表的存储结构。

本章在 Visual Studio 中用名为 matrix 的类库型项目实现有关数据结构的类型定义，用名为 matrixtest 的应用程序型项目实现相应类型数据结构的测试和演示程序。

建议本章授课 4 学时，实验 3 学时。

5.1　数组

数组（array）是一种重要的基础性数据集合类型，是其他数据结构实现顺序存储的基础。一个数组是由一组相同数据类型的数据元素组成的集合，元素的类型可以是简单的基本类型，也可以是复杂的用户自定义类型，但不论元素类型如何，数组元素都按次序存储于一个地址连续的内存空间中。某个数组元素在数组中的位置可以通过该元素的序号确定，这个序号称之为数组元素的下标，简称数组下标。为了物理上访问某数组元素，可以通过它的下标，再加上数组的起始地址，就可找到存放该元素的存储地址。逻辑上，数组可以看成二元组<下标，值>的一个集合，以后我们还会看到二元组<键，值>的集合。

数组下标的个数称为数组的维数，有一个下标的数组是一维数组，有两个下标的数组就是二维数组，以此类推。C#编程语言中数组的工作方式与在大多数其他流行语言中的工作方式类似，但还是有一些差异应引起注意。C#支持一维数组、多维数组（矩形数组）和数组的数组（交错的数组）。C#中数组变量是引用类型变量，只有用 new 操作符为数组分配空间后，数组才真正占有实际的存储空间，数组元素的下标从零开始计算。

5.1.1　一维数组

一维数组是由 n（n>1）个相同数据类型的数据元素 $a_0, a_1, \cdots, a_{n-1}$ 构成的有限序列，其中 n 称为数组的长度。数组记作：

$$\text{Array} = \{ a_0，\ a_1，\ a_2，\ \cdots，\ a_{n-1}\}$$

数组元素依次占用一块地址连续的内存空间，每两个相邻数据元素之间都有直接前驱和直接后继的关系。当系统为一个数组分配内存空间时，数组所需空间的大小及其首地址就确定下来。只需一个

数组下标即可标识任意一个元素在序列中的位置，通过数组名加下标的形式，可以访问数组中任意一个指定的数组元素。假设数组的首地址为 Addr(a_0)，每个数据元素占用 c 个存储单元，则第 i 个数据元素的地址为：

$$\text{Addr}(a_i) = \text{Addr}(a_0) + i \times c$$

根据数组元素的下标就可计算出该元素的存储地址，因而可存取数组元素的值，并且该操作的复杂度是 O(1)，具有这种特性的存储结构称为随机存储结构，可见，数组是一种随机存储结构。

高级程序语言中存在两种为数组分配内存空间的方式：编译时分配数组空间和运行时分配数组空间：

- 编译时分配数组空间：程序声明数组时给出数组元素类型和元素个数，编译程序为数组分配好存储空间。当程序开始运行时，数组即获得系统分配的一块地址连续的内存空间。
- 运行时分配数组空间：程序声明时，仅需说明数组元素类型，不指定数组长度。当程序运行中需要使用数组时，向系统申请指定长度数组所需的存储单元空间。当不再需要这个数组时，需要向系统归还所占用的内存空间。

在 C 语言中，以上两种方式都存在。例如，在 C 语言某个函数中，通过声明语句 int a[10]定义局部数组变量 a，并为数组 a 分配 10 个单元的内存空间。第二种方式的例子是，在某个函数中通过声明语句 int* a 将变量 a 定义为局部整型指针，再通过语句 a = (int*)malloc(10*sizeof(int)) 向系统申请 10 个单元的内存空间。

在 C#语言中，数组都是在运行时分配所需空间。例如，通过语句 int[] a = new int[10]声明变量 a 为整型数组变量，并在程序运行时向系统申请 10 个单元的内存空间。

5.1.2 二维数组

1. 二维数组的概念

如果将数组及其元素的概念加以推广，就可得到所谓的多维数组，多维数组被视为数组的数组，其中的数组元素本身就是一个数组，例如二维数组可以看作是元素为一维数组的数组，而三维数组可以看成是由二维数组组成的数组。n 维数组需要 n 个下标来确定具体元素的位置。

二维数组常用来表示一个矩阵

$$\boldsymbol{A}_{m\times n} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

$\mathbf{A}_{m\times n}$ 表示由 $m\times n$ 个元素 $a_{i,j}$ 组成的矩阵，可以看成是由 m 行一维数组组成的（行）数组，或是 n 列一维数组组成的（列）数组。

矩阵 $\mathbf{A}_{m\times n}$ 也可以视为一种特殊的双重线性表，矩阵中的每个元素 $a_{i,j}$ 同时属于两个线性表：第 i 行的线性表和第 j 列的线性表。一般情况下，元素 $a_{i,j}$ 有 1 个行前驱 $a_{i-1,j}$ 和 1 个列前驱 $a_{i,j-1}$ 以及 1 个行后继 $a_{i+1,j}$ 和 1 个列后继 $a_{i,j+1}$。矩阵的首元素 $a_{0,0}$ 没有前驱；矩阵的最后一个元素 $a_{m-1,n-1}$ 没有后继。矩阵边界上的元素 $a_{0,j}$（j=0，…，n-1）只有列后继，没有列前驱；$a_{i,0}$（i=0，…，m-1）只有行后继，没有行前驱；$a_{m-1,j}$（j =0，…，n-1）只有列前驱，没有列后继；$a_{i,n-1}$（i=0，…，m-1）只有行前驱，没有行后继。

2. 二维数组的遍历

遍历一种数据结构，就是按照某种次序访问该数据结构中的所有元素，并且每个数据元素恰好访问一次，这样将得到一个由所有数据元素组成的线性序列。

一维数组只有一种遍历次序，而二维数组则有两种次序：行优先遍历和列优先遍历。

（1）行优先次序遍历：

对二维数组依行序逐行访问每个数据元素，得到的线性序列是将二维数组元素依次按行的一个排列，第 i+1 行紧跟在第 i 行后面。对于二维数组 $\mathbf{A}_{m\times n}$，行优先遍历可以得到如下线性序列：

$$a_{0,0}，a_{0,1}，\cdots，a_{0,n-1}，a_{1,0}，a_{1,1}，\cdots，a_{1,n-1}，\cdots，a_{m-1,0}，a_{m-1,1}，\cdots，a_{m-1,n-1}$$

（2）列优先次序遍历：

对二维数组依列序逐列访问每个数据元素，得到的线性序列是将二维数组元素依次按列的一个排列，第 j+1 列紧跟在第 j 列后面。对于二维数组 $\mathbf{A}_{m\times n}$，列优先遍历可以得到如下线性序列：

$$a_{0,0}，a_{1,0}，\cdots，a_{m-1,0}，a_{0,1}，a_{1,1}，\cdots，a_{m-1,1}，\cdots，a_{0,n-1}，a_{1,n-1}，\cdots，a_{m-1,n-1}$$

3. 二维数组的顺序存储结构

二维数组的顺序存储可以有两种方式：一种是按行优先次序存储，或称行主序（row major order）；另一种是按列优先次序存储，或称列主序（column major order）。

假设每个数据元素占用 c 个存储单元，Addr($a_{i,j}$)为元素 $a_{i,j}$ 的存储地址，Addr($a_{0,0}$)为元素 $a_{0,0}$ 的地址，也就是数组的起始地址。如果按行优先存储二维数组 $\mathbf{A}_{m\times n}$，则元素 $a_{i,j}$ 的地址计算函数为：

$$\mathrm{Addr}(a_{i,j}) = \mathrm{Addr}(a_{0,0}) + (i \times n + j) \times c$$

如果按列优先存储数组，则元素 $a_{i,j}$ 的地址计算函数为：

$$\mathrm{Addr}(a_{i,j}) = \mathrm{Addr}(a_{0,0}) + (j \times m + i) \times c$$

在 Pascal，C/C++/C#和 Java 语言中，二维数组都是行优先存储，而在 FORTRAN 和 Matlab 语言中，二维数组都是列优先存储。

不管是以上那种方式，存储地址与数组下标之间仍然存在着简单的线性关系，可见二维数组的顺序存储结构也具有随机存储特性，对数组元素进行随机存取的时间复杂度为 O(1)。

5.1.3　C#中的数组

1. 一维数组

C#中声明一维数组变量的格式是: 〈类型〉[ ] 数组名;

例如：语句

```
int[] a;
```

定义 a 为一个整型数组。

数组元素既可以是简单数据类型，也可以是自定义类型。

在 C# 中，数组是一种引用类型的变量，声明数组并没有实际创建它们，必须进行实例化后才能使用它们。只有用 new 操作符为数组分配空间后，数组才真正占有实际的存储单元。使用 new 创建一维数组的语法如下：

```
〈数组名〉 = new 〈类型〉[〈长度〉]
```

例如：

```
a = new int[10];
```

为整型数组 a 分配 10 个单元的空间。

可以在声明数组的同时为数组进行初始化，例如：

```
int[] a = {1,2,3,4,5};
```

通过下标可以访问数组中的任何元素。数组元素的访问格式为：<数组名>[<下标>]。

可见 C#中的数组是在运行时分配所需空间，声明数组变量时不用指定数组长度，使用 new 运算符为数组分配空间后，数组才真正占用一片地址连续的存储单元空间。而当数组使用完之后，不需要立即向系统归还所占用的内存空间。因为.NET 平台的垃圾回收机制将自动判断对象是否在使用，并能够自动销毁不再使用的对象，收回对象所占的资源。

C#中的所有数组都是类（class）类型的数据类型，任何数组类型都隐含继承自基类型 System.Array，因此数组实例都具有对象特性。System.Array 类中定义的属性以及其他类成员都可以供所有数组实例使用，例如 Length 属性可以获得数组元素的个数，即数组的长度。System.Array 类还提供了许多用于排序、搜索和复制数组的方法。

公共属性

```
virtual int Length {get;}                    //返回数组的所有维数中元素的总数。
virtual int Rank {get;}                      //获取数组的维数。
```

公共方法

```
static void Copy(Array source, int srcIndex, Array destination, int dstIndex, int length)
//从指定位置开始，复制源数组中指定长度的元素到目数组中的指定位置，有多个重载方法。
void CopyTo(Array dstArray, int dstIndex)
//将当前一维数组的所有元素复制到目数组中指定位置
static int IndexOf<T>(T[] a, T k)            //返回给定数据首次出现位置
static void Sort(Array a)                    //对整个一维数组元素进行排序，有多个重载方法。
int GetLength( int dimension);               //获取数组指定维中的元素数
```

【例5.1】　数组的搜索与排序。

```
using System;
namespace matrixtest {
    class ArrayTest {
        static void Main(string[] args) {
            double[] d = { 3.0, 4.0, 1.0, 2.0, 5.0 };
            int i = Array.IndexOf<double>(d, 5.0);
            Console.WriteLine("{0}'s index is: {1}",5.0,i);
            Console.Write("Sorted Array: ");
            Array.Sort(d);
            foreach(double f in d) Console.Write("{0} ", f);
            Console.WriteLine();
        }
    }
}
```

程序运行结果如下：

```
5's index is: 4
Sorted Array: 1 2 3 4 5
```

这里使用了 Array 类的静态泛型方法 IndexOf<T>来进行查找操作，在调用时需指明数组元素的类型，上例用 double 代替符号“<T>”中的 T。.NET Framework 和 C#语言 2.0 版开始增加了泛型类型和泛型方法，泛型利用类型参数将一个或多个类型的指定推迟到用户代码声明并实例化该类或方法的时候。使用泛型可以最大限度地重用代码、保护类型的安全以及提高性能。

2. 多维数组

C#中的多维数组通过说明多个下标的形式来定义，例如：

```
int[,] items = new int[5,4];
```

上述语句声明了一个二维数组 items，并分配 5×4 个存储单元。

同样也可以初始化多维数组，例如：

```
int[,] numbers = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
```

初始化多维数组时也可以省略数组的大小，如下所示：

```
int[,] numbers = new int[,] { {1, 2}, {3, 4}, {5, 6} };
```

C#中的二维数组按行优先顺序存储数组的元素。

【例5.2】　自定义矩阵类及矩阵的相加操作。

几乎所有的程序设计语言，如 C，C++和 C#，都是用一维或二维数组来处理矩阵，这种方法不够自然，有时显得繁琐。本例声明 Matrix 类表示矩阵，类中成员 items 是一个元素类型为整型 int 的一维数组，成员变量 rows 记录矩阵的行数，成员变量 cols 记录矩阵的列数。设计了多个构造方法，以方便构造和初始化矩阵对象。Add()方法实现与另一个矩阵的相加操作，类中对'+'运算符进行了重载，提供一种完成两个矩阵的相加操作的简洁形式。Transpose()方法实现矩阵的转置操作。

程序如下：

```
public class Matrix{
    private int[] items;
    private int rows, cols;

    public Matrix(int nRows,int nCols) {
        rows = nRows; cols = nCols;
        items = new int[rows * cols];
    }
    public Matrix(int nSize): this(nSize, nSize) {
    }
    public Matrix(): this(1){ }
    public Matrix(int nRows,int nCols,int[] mat) {
        rows = nRows; cols = nCols;
        items = new int[rows * cols];
        Array.Copy(mat, items, mat.Length);
    }
    public Matrix(Matrix omat) {
        rows = omat.Rows; cols = omat.Columns;
        int size = rows * cols;
        items = new int[size];
        Array.Copy(omat.items, this.items, size);
    }
    public int Rows {
        get{ return rows; }
    }

    public int Columns {
```

```
        get{ return cols; }
    }

    //获得或设置第i行第j列的元素
    public int this[int i, int j]{
        get{ return items[i*rows+j];  }
        set{ items[i*rows+j] = value; }
     }

  //两个矩阵相加
  public void Add(Matrix b) {
       for(int i=0;i<Rows;i++)
            for(int j=0;j<Columns;j++)
                 items[i*rows+j] +=  b[i,j];
  }

  // '+'运算符重载
  public static Matrix operator +(Matrix a, Matrix b){
  Matrix c = new Matrix(a.Rows,a.Columns);
       for(int i=0;i<a.Rows;i++)
            for(int j=0;j<a.Columns;j++)
                 c[i,j] = a[i,j] + b[i,j];
       return c;
  }
 public void Transpose() {
     Matrix trans = new Matrix(Columns,Rows);
     int t = 0;
     for (int i = 0; i < Rows; i++) {
         for (int j = i+1; j < Columns; j++) {
             t = this[i, j];
             this[i, j] = this[j, i];
             this[j, i] = t;
          }
      }
  }
  //遍历,输出各元素值
  public void Show() {
       int i,j;
       for(i=0;i<Rows;i++) {
            for(j=0;j<Columns;j++)
                 Console.Write(" " + items[i,j]);
            Console.WriteLine();
       }
```

```
            Console.WriteLine();
        }
    }
```

Matrix 类定义在 Matrix.cs 源文件中，同样也声明为 DSAGL 名字空间中的类；源程序 MatrixTest.cs（定义在 matrixtest 项目中，因而缺省处在 matrixtest 名字空间中），引用在 DSAGL 名字空间中定义的 Matrix 类定义两个矩阵 a 和 b，并进行加法运算以及转置操作。程序如下：

```
using System; using DSAGL;
namespace matrixtest {
    class MatrixTest{
        public static void Main(string[] args) {
            int[ ]  m1 = {1,2,3,4,5,6,7,8,9};
            Matrix a = new Matrix(3, 3, m1); a.Show();
            int[]  m2 = {1,0,0,0,1,0,0,0,1};
            Matrix b = new Matrix(3, 3, m2); b.Show();
            a.Add(b); a.Show();
            Matrix c = a + b; c.Show();
            c.Transpose();
            Matrix d = new Matrix(c);
            d.Show();
        }
    }
}
```

程序运行结果如下：

```
1 2 3
4 5 6
7 8 9

1 0 0
0 1 0
0 0 1

2 2 3
4 6 6
7 8 10

3 2 3
4 7 6
7 8 11

 3 4 7
 2 7 8
```

3 6 11

5.2 稀疏矩阵

在科学与工程计算中经常出现一些阶数很高的矩阵，在这类矩阵中常常存在许多值相同的元素或零元素，如果对这类矩阵按常规方法存储，就会占用很大的存储空间并有较多的信息冗余。在这类应用中，应该采用特殊方式进行压缩存储以节省存储空间。

设矩阵 $\mathbf{A}_{m\times n}$ 中有 t 个非零元素，则矩阵中非零元素所占比例为 $\delta = t\times(m\times n)^{-1}$，当 $\delta \leqslant 0.1$ 时，称这类矩阵为稀疏矩阵（sparse matrix）。

在存储稀疏矩阵时，如果仍然用顺序存储的方法将每个元素都存起来，就会占用许多存储单元去存储重复的零值，这无疑会造成存储空间的浪费。为了节省存储空间，可以采用只存储其中的非零元素的压缩存储方式。这种压缩存储方式，可以压缩掉重复的零元素的存储空间，但可能也会失去数组的随机存取特性。

如果矩阵中有很多零且非零元素具有某种分布规律时，可以只对非零元素进行顺序存储，此时仍可以进行随机存取。例如，下三角矩阵

$$\boldsymbol{A}_{m\times n}=\begin{bmatrix} a_{0,0} & 0 & \cdots & 0 \\ a_{1,0} & a_{1,1} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

当 i<j 时，上三角元素 $a_{i,j}$=0。如果按行优先次序遍历矩阵中的下三角元素，便可得到如下的线性序列：

$$a_{0,0}，\quad a_{1,0}，a_{1,1}，\cdots，a_{\text{m-1, 1}}，a_{\text{m-1, 1}}，\cdots，a_{\text{m-1, n-1}}$$

如果按行优先次序只将矩阵中的下三角元素顺序存储，第 0 行到第 i-1（$i\geqslant 1$）行元素的个数为：

$$\sum_{k=0}^{i-1}(k+1)=\frac{i(i+1)}{2}$$

因此，元素 $a_{\text{i,j}}$（$i\geqslant j$）的地址可用下式计算：

$$\text{Addr}(a_{i,j})=\text{Addr}(a_{0,0})+[\frac{i(i+1)}{2}+j]\times c, 0\leq j\leq i\leq n-1$$

如果矩阵中大多数元素值为零且非零元素的分布没有规律时，可以用顺序存储结构或链式存储结构存储表示非零元素的三元组。

5.2.1 稀疏矩阵的三元组

稀疏矩阵的一个非零元素可以由一个三元组<行下标，列下标，矩阵元素值>来表示，一个稀疏矩阵则可以用它的三元组集合表示。例如，稀疏矩阵

$$\boldsymbol{A}=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 3 \\ 0 & 4 & 0 & 5 \end{bmatrix}$$

可以用三元组序列表示为：

{{0,0,1}, {2,0,2}, {2,3,3}, {3,1,4}, {3,3,5}}

如果只存储稀疏矩阵的三元组集合，也就是只存储矩阵中的非零元素，就可以达到压缩存储稀疏矩阵的目的。稀疏矩阵的三元组集合可以用顺序存储结构和链式存储结构两种方法实现。

5.2.2 三元组的顺序存储结构

稀疏矩阵的三元组集合的顺序存储结构是将表示稀疏矩阵非零元素的三元组，按照行优先（或列优先）的原则，依次存储在一个占据连续存储空间的数组中，该数组元素的类型为稀疏矩阵三元组，每个稀疏矩阵非零元素三元组对应于该数组中的一个元素。例如，对于稀疏矩阵 A 三元组序列{{0,0,1}, {2,0,2}, {2,3,3}, {3,1,4}, {3,3,5}}，其顺序存储结构如表 5-1 所示。

表 5-1　稀疏矩阵三元组的顺序存储结构

三元组数组下标	行下标	列下标	数据元素值
0	0	0	1
1	2	0	2
2	2	3	3
3	3	1	4
4	3	3	5

1. 稀疏矩阵的顺序存储结构三元组类

为描述顺序存储结构的稀疏矩阵中表示非零元素的三元组，定义如下的 TripleEntry 类：

```
public class TripleEntry {
    private int row;                          //行下标
    private int column;                       //列下标
    private int data;                         //值

    public TripleEntry(int i, int j, int k) {
        row = i;
        column = j;
        data = k;
    }

    public TripleEntry() : this(0, 0, 0) { }

    public int Row {  get { return row; }   set { row = value; } }

    public int Column { get { return column; } set { column = value; } }

    public int Data { get { return data; } set { data = value; } }

    //输出一个元素的三元组值
    public void Show() {
```

```
            Console.WriteLine("r: " + row + "\tc: " + column + "\tv: " + data);
        }
    }
```

用 TripleEntry 类型定义的实例表示稀疏矩阵的一个三元组，用来记录稀疏矩阵中的一个非零元素的行列位置及其值。

2. 基于三元组顺序存储结构的稀疏矩阵类

下面声明的 SSparseMatrix 类表示基于三元组顺序存储结构的稀疏矩阵对象。

```
using System;  using System.Collections.Generic;
namespace DSAGL{
    public class SSparseMatrix{
        private int rows, cols;
        protected List<TripleEntry> items;                    // 三元组线性表

        public int Rows { get { return rows; } set { rows = value; } }
        public int Columns { get { return cols; } set { cols = value; } }

        public SSparseMatrix(int[,] mat){
            Console.WriteLine("稀疏矩阵:");
             rows = mat.GetLength(0); cols = mat.GetLength(1);
             items = new List<TripleEntry>();
             for (int i = 0; i < rows; i++) {
                 for (int j = 0; j < cols; j++) {
                     Console.Write("  " + mat[i,j]);
                     if(mat[i,j]!=0) {
                         items.Add( new TripleEntry(i,j,mat[i,j]) );
                     }
                 }
                 Console.WriteLine();
             }
        }

        //输出一个稀疏矩阵中所有元素的三元组值
        public void Show(){
            Console.WriteLine("{0}x{1} 稀疏矩阵三元组的顺序表示:",rows,cols);
            Console.WriteLine("\t行下标\t列下标\t值");
            for(int i=0; i<items.Count; i++) {
                Console.Write("items["+i+"] = ");
                items[i].Show();
            }
        }
    }
```

```
}
```

在 SSparseMatrix 类中，成员 items 是一个用线性表表示的动态数组，元素类型为三元组 TripleEntry 类。稀疏矩阵 SSparseMatrix 类的构造方法将一个常规稀疏矩阵转换成三元组的顺序存储结构表示法。

三元组 TripleEntry 类和稀疏矩阵 SSparseMatrix 类中都定义了成员 Show()方法。TripleEntry 类中的 Show ()方法输出一个矩阵元素的三元组值，SSparseMatrix 类中的 Show ()方法输出一个稀疏矩阵中所有的三元组，其中每个非零元素均调用 TripleEntry 类的 Show ()方法输出一个三元组值。

【例5.3】　测试基于三元组顺序存储结构的稀疏矩阵类。

下面的程序调用声明在 DSAGL 名字空间中的类 SSparseMatrix 实现稀疏矩阵三元组的顺序存储结构，SSparseMatrixTest.cs 源程序定义在 matrixtest 项目中，因而缺省处在 matrixtest 名字空间中。程序如下：

```
using System; using DSAGL;
namespace matrixtest {
    public class SSparseMatrixTest {
        public static void Main(string[] args) {
            //稀疏矩阵
            int[,] mat = {{1,0,0,0}, { 0,0,0,0}, {2,0,7,0}, {0,0,8,9}};
            SSparseMatrix ssm = new SSparseMatrix(mat);
            ssm.Show();
        }
    }
}
```

程序运行结果如下：

```
稀疏矩阵:
  1  0  0  0
  0  0  0  0
  2  0  0  3
  0  4  0  5

4x4 稀疏矩阵三元组的顺序表示:
          行下标 列下标    值
items[0] = r: 0  c: 0    v: 1
items[1] = r: 2  c: 0    v: 2
items[2] = r: 2  c: 3    v: 3
items[3] = r: 3  c: 1    v: 4
items[4] = r: 3  c: 3    v: 5
```

在上面的三元组顺序存储结构稀疏矩阵的实现中，我们用一个动态数组（线性表）保存稀疏矩阵的非零元素三元组序列，它适合于非零元素的数目发生变化的情况。可以看到顺序存储结构的稀疏矩阵结构简单，但插入、删除操作不方便。若矩阵元素的值发生变化，一个值为零的元素变为非零元素，就要向线性表中插入一个三元组；若非零元素变成零元素，就要从线性表中删除一个三元组。为了保持线性表元素间的相对次序，进行插入和删除操作时，就必须移动其他元素。这方面的不足可以通过采用后一节将介绍的三元组链式存储结构加以克服。

5.2.3 三元组的链式存储结构

稀疏矩阵的三元组链式存储结构可以有几种常用的方式，例如，基于行的单链的表示、基于列的单链的表示和十字链表示方法，下面介绍稀疏矩阵基于行的单链的表示方法。

将稀疏矩阵每一行上的非零元素作为结点链接成一个单向链表，用一个数组记录这些链表，从上到下，数组的元素依次指向各行所对应的链表的第一个结点。对于前述稀疏矩阵 A，其行的单链表示如图 5.1 所示。

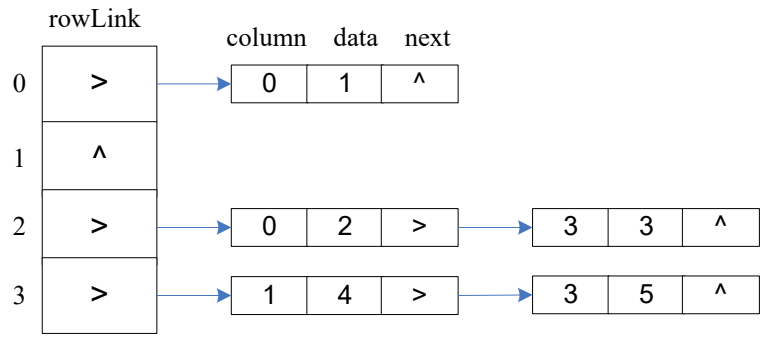


图 5.1 稀疏矩阵的行的单链表示

为了以行的单链表示稀疏矩阵，可以声明如下的两个类：三元组结点类 LinkedTriple 和链式存储结构稀疏矩阵类 LSparseMatrix。

```
public class LinkedTriple {
    private int column;                          //列下标
    private int data;                            //值
    private LinkedTriple next;

    public int Column {get { return column; } set { column = value; } }
    public int Data { get { return data; } set { data = value; } }
    public LinkedTriple Next { get { return next; } set { next = value; }}

    public LinkedTriple(int i, int k) {
        column = i;
        data = k;
        next = null;
    }

    public LinkedTriple(): this(0, 0) { }

    //输出当前元素的三元组值，并传导输出当前行的下一个三元组
    public void Show() {
        LinkedTriple p = this;
        while (p != null) {
            Console.Write(" " + p.Column + " " + p.Data + " -> ");
```

```
                p = p.Next;
            }
            Console.WriteLine(".");
        }
    }
```

三元组结点类 LinkedTriple 定义链表结点的类型，它由 3 个成员组成：column（列下标），data（值）和 next（用来引用后继结点）。一个 LinkedTriple 类型的对象表示链表中的一个结点，对应于稀疏矩阵中的一个非零元素。LinkedTriple 类中的 Show()方法输出结点及本结点后链接的其他结点的值。

下面定义的 LSparseMatrix 类实现稀疏矩阵的行单链表示，它的成员 rowLink 是一个数组，其元素类型为 LinkedTriple 类，rowLink 数组元素依次存放每条链表第 1 个结点的引用。

LSparseMatrix 矩阵类的一个构造方法是将一个常规矩阵转换成行的单链表示，Show()方法依次输出矩阵各行链表中的全部结点值。

```
public class LSparseMatrix {
    LinkedTriple[] rowLink;
    private int rows, cols;

    public int Rows { get { return rows; } set { rows = value; } }
    public int Columns { get { return cols; } set { cols = value; } }

    public LSparseMatrix(int[,] mat) {
        rows = mat.GetLength(0);
        cols = mat.GetLength(1);
        rowLink = new LinkedTriple[rows];
        int i,j;
        LinkedTriple p = null, q;
        for (i = 0; i < rows; i++) {
            p = rowLink[i];
            for (j = 0; j < cols; j++) {
                if( mat[i,j]!=0 ){
                    q = new LinkedTriple(j, mat[i,j]);
                    if ( p==null )
                        rowLink[i] = q; // rowLink数组存放链表第1个结点的引用
                    else
                        p.Next = q;
                    p = q;
                }
            }
        }
    }

    public void Show(){
        int i;
```

```
            Console.WriteLine("{0}x{1}稀疏矩阵行的单链表示:",rows,cols);
             for (i = 0; i < rows; i++) {
                Console.Write("Row Triples["+i+"] = ");
                if(rowLink[i]!=null)
                    rowLink[i].Show();
                else
                    Console.WriteLine(".");
            }
        }
    }
```

【例5.4】　基于行单链的稀疏矩阵实现。

下面的程序调用 LSparseMatrix 类实现稀疏矩阵行的单链表示。

```
    using System; using DSAGL;
    namespace matrixtest {
        public class LSparseMatrixTest {
            public static void Main(string[] args) {
                int[,] mat= {{1,0,0,0}, {0,0,0,0}, {2,0,0,3}, {0,4,0,5}};
                LSparseMatrix lsm = new LSparseMatrix(mat);
                lsm.Show();
            }
        }
    }
```

程序运行结果如下：

```
    4x4稀疏矩阵行的单链表示:
    Row Triples[0] =  0 1 -> .
    Row Triples[1] = .
    Row Triples[2] =  0 2 ->  3 3 -> .
    Row Triples[3] =  1 4 ->  3 5 -> .
```

在基于行单链的稀疏矩阵的实现中，存取一个元素的时间复杂度为 $O(n)$，其中 n 为矩阵的列数。按行的单链表示的稀疏矩阵，每个结点可以很容易地找到行方向上的后继结点，但较难找到列方向上的后继结点。将行的单链表示和列的单链表示结合起来存储稀疏矩阵的十字链表示方法可以带来更大的灵活性。

5.3　广义表

5.3.1　广义表的概念及定义

线性表结构可以是简单的数组，也可以扩展为复杂的数据结构—广义表（general list）。广义表是 n

（$n \geqslant 0$）个数据元素 $a_0, a_1, \cdots, a_{n-1}$ 组成的有限序列，记为：

$$\text{GeneralList} = \{a_0, a_1, \cdots, a_{n-1}\}$$

与第二章介绍的普通线性表不同，这里的广义表在结构复杂性上可以进行扩展。元素 a_i 可以是称为原子的、不可再分的单元素，也可以是还可再分的线性表或广义表，这些可再分的元素称作子表。广义表所包含的数据元素的个数 n 称为广义表的长度，当 n=0 时的广义表为空表。

广义表的元素或为原子或为子表，为了便于区分，在下面的描述中用小写字母表示原子，用大写字母表示表和子表。例如，

L1＝( )：L1 为空表，长度为 0。

L2＝(L1)＝(( ))：广义表 L2 包含一个子表元素 L1，L2 的长度为 1。

L＝(1, 2)：线性表 L 包含两个（原子）元素，表的长度为 2。

T＝(3, L)＝(3, (1, 2))：广义表 T 包含（原子）元素 3 和子表元素 L，T 的长度为 2。

G＝(4, L, T)＝(4, (1, 2), (3, (1, 2)))：广义表 G 包含元素 4、子表 L 和 T，G 的长度为 3。

Z＝(e, Z)＝(e, (e, (e, (⋯))))：广义表 Z 包含元素 e 和子表 Z，Z 是一个递归表，其长度为 2。

在上面的例子中 L1，L2，L 和 Z 等分别是各广义表变量的名字，简称表名。在表示广义表时，可以将表名写在对应的括号前，这样既标明了每个表的名字，又说明了它的组成，于是在上面的示例中的各表又可以表示成：

L1( )，L2 (L1( ))，L(1, 2)，T(3, L (1, 2))，G(4, L(1, 2), T(3, L(1，2)))，Z(e, Z (e, Z(e, Z(⋯))))

由上面的定义可见，广义表可以表示一种多层次的结构，它是用递归的方式进行定义的。广义表层次的深度即是广义表的深度。在前面的例子中，各个广义表的深度等于表中所含括号的层数。例如，表 L 的深度为 1，表 T 的深度为 2，表 G 的深度为 3。容易看出，空表的深度为 1，原子的深度为 0。

如果广义表的某个子表元素是其自身，如前面例子中的广义表 Z，则称该广义表为递归表。递归表的长度是有限值，深度却可能是无穷值。

5.3.2　广义表的特性和操作

1. 广义表的特性

（1）广义表可作为其他广义表的子表元素

例如在前面的例子中，广义表 L 分别是广义表 T 和 G 不同层次上的元素，我们称表 T 和 G 共享子表 L，共享又可看成引用。在算法中，通过子表的引用，可以避免在母表中重复列出子表的值，这样就利用了广义表的共享特性达到减少存储结构中的数据冗余和节约存储空间的目的。

（2）广义表是一种多层次的结构

广义表中的元素可以是广义子表，因此广义表可以表示线性表、树和图等多种基本的数据结构。树结构和图结构都是某种多层次的结构，有关它们的基本概念将在以后的章节中介绍，这里仅指出，当广义表的数据元素中包含子表时，该广义表就是一种多层次的结构。

如果限制广义表中成分的共享和递归，所得到的结构就是树结构，树中的叶结点对应广义表中的原子，非叶结点对应子表。例如 T(3, L (1, 2))表示一种树形的层次结构。

（3）广义表是一种广义的线性结构

广义表同一层次的数据元素之间有着固定的相对次序，是线性关系，如同普通线性表。线性表是广义表的特例，而广义表则是线性表的扩展，当广义表的数据元素全部是原子时，该广义表就是线性

表。例如，广义表 L(1, 2)其实已简化为一个线性表。

（4）广义表可以递归

广义表中如果有共享或递归成分的子表，就会演变为图结构。

通常将与树结构对应的广义表称为纯表，将允许数据元素共享的广义表称为再入表，将允许递归的广义表成为递归表，它们之间的关系满足：

递归表 ⊃ 再入表 ⊃ 纯表 ⊃ 线性表

2. 广义表的操作

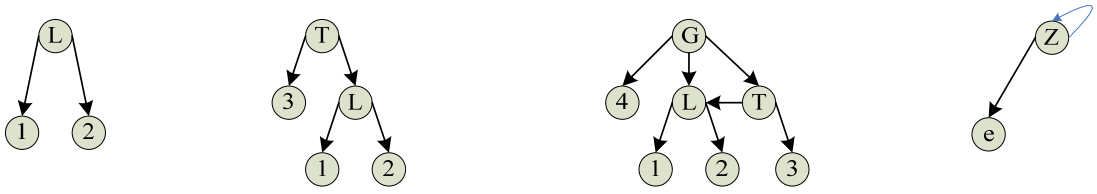
广义表具有弹性，用广义表的形式可以表示线性表、树和图等多种基本的数据结构，因此广义表的操作既包括与线性表、树和图等数据结构类似的基本操作，也包括一些特殊操作，主要有：

- Initialize：初始化。建立一个广义表。
- IsAtom：判别某数据元素是否为原子。
- IsList：判别某数据元素是否为子表。
- Insert：插入。在广义表中插入一个数据元素（原子或子表）。
- Remove：删除。从广义表中删除一个数据元素（原子或子表）。
- Equals：判别两个广义表是否相等。
- Copy：复制。复制一个广义表。

5.3.3 广义表的图形表示

用广义表的形式可以表达线性表、树和图等基本的数据结构，这些数据结构可以分别与相应的有向图建立对应关系。在这种对应中，主表对应于树的根结点或图的起始结点，广义表中的各数据元素依次对应于与根结点相邻接的各结点，如果某个数据元素是原子，则对应的结点称为原子结点；如果某元素是子表，则可继续上述对应过程来处理，直到所有层次。广义表和有向图之间的这种对应关系构成了广义表的图形表示。

用广义表的形式表达线性表、树和图等基本的数据结构如图 5.2 所示。



(a) 线性结构L(1, 2) (b) 树结构: 纯表T(3, L(1, 2)) (c) 图结构: 再入表G(4, L(1,2), T(3,L(1,2))) (d) 图结构: 递归表Z(e, Z)

图 5.2 广义表表示的多种结构对应的图形表示

在图 5.2 中可见线性表、树和图结构的若干特性：

- 广义表 L(1, 2)的数据元素全部是原子，元素对应的结点都是原子结点，该广义表为具有线性特性的线性表。
- 广义表 T(3, L)的数据元素中有原子，也有子表，但表中不存在共享和递归成分，该广义表为具有树结构特性的纯表。原子元素用原子结点表示，子表用分枝结点表示。
- 广义表 G(4, L, T)的数据元素中有子表，并且表中有共享成分，该广义表为具有图结构特性的再入表。

- 广义表 Z(e, Z)的数据元素中有子表且有递归成分，该广义表为具有图结构特性的递归表。

5.3.4　广义表的存储结构

具有线性特性的普通线性表有顺序存储结构和链式存储结构两种实现方式，非线性结构的广义表则通常采用链式存储结构。本章简要说明广义表链式存储结构的一般方法，有关树结构和图结构的存储表示的专门问题将在相关章节中讨论。

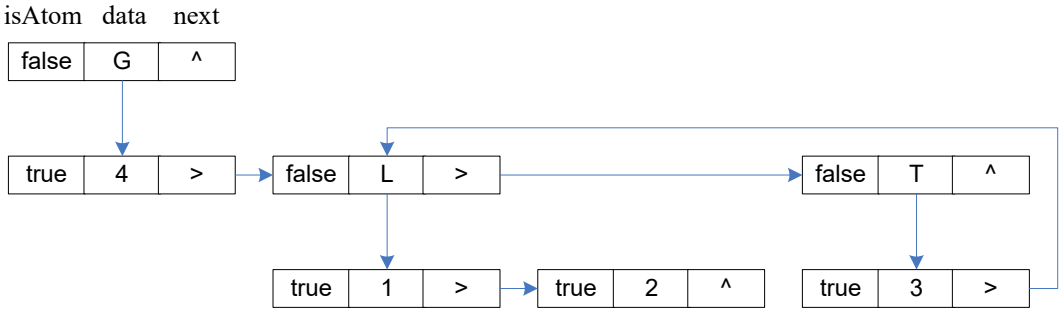
1. 广义表的单链表示

广义表可以用单向链表结构存储。单向广义链表的每个结点由如下 3 个域组成：

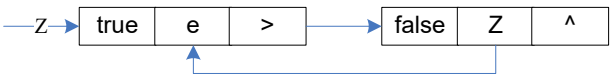
```
public class GSLinkedNode{
    public bool isAtom;
    public object data;
    public GSLinkedNode next;
    其他成员
}
```

域 isAtom 是一个标志域，表示数据元素（结点）是否为原子，当 isAtom 等于 true 时，表明当前结点为原子，data 存放当前原子结点的数据值；当 isAtom 等于 false 时，表明当前结点为子表，data 存放子表中第一个数据元素所对应结点的引用。next 成员存放与当前数据元素处于同层的下一个数据元素所对应结点的引用，当本数据元素是所在层的最后一个数据元素时，next 为 null。用单链方式表示的再入表和递归表如图 5.3 所示。

当广义表中有共享成分时，被共享的结点只需出现一次，但可能被重复引用。例如，再入表 G 中有子表 L 和 T，而 T 中也有子表 L，所以在图中子表 L 的结点被引用了两次。



(a) 再入表



(b) 递归表

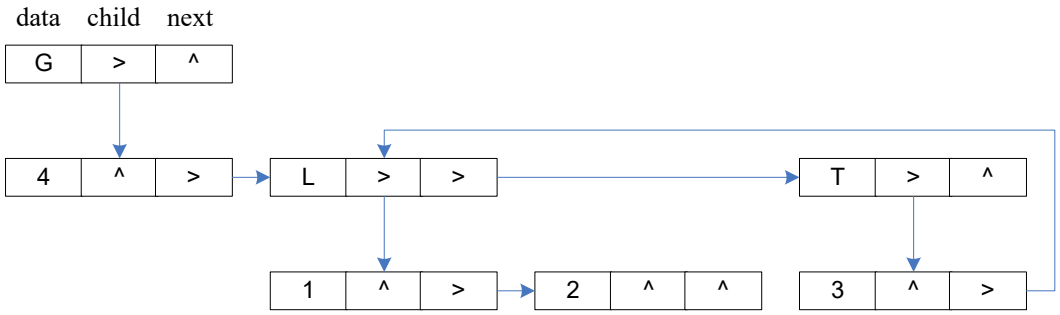
图 5.3　广义表的单链表示

2. 广义表的双链表示

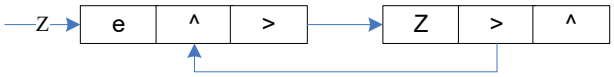
广义表也可以用双向链表结构存储。双向广义链表的每个结点由如下 3 个域组成：

```
public class GDLinkedNode<T>{
    public T data;
    public GDLinkedNode<T> child;
    public GDLinkedNode<T> next;
    ……
}
```

域 data 存放数据元素信息，域 child 是子表中第一个数据元素所对应结点的引用，next 则引用与本数据元素处于同层的下一个数据元素所对应的结点。当本数据元素是所在层的最后一个数据元素时，next 为 null。如果域 child 为 null，则表明本结点是原子结点。用双链方式表示的再入表和递归表如图 5.4 所示。



(a) 再入表



(b) 递归表

图 5.4 广义表的双链表示

习题 5

5.1 在二维矩阵 Matrix 类中增加下列功能：

1) 求一个矩阵的转置矩阵，方法声明为：public void Transpose( );

2) 两个矩阵相减，方法声明为：public void Substract(Matrix b);。

3) 两个矩阵相乘，方法声明为：public void Multiply(Matrix b);。

5.2 在表示稀疏矩阵的三元组顺序存储结构 SSparseMatrix 类中，增加以下功能：

1) 稀疏矩阵的转置矩阵。

2) 两个稀疏矩阵相加。

3) 两个稀疏矩阵相乘。

5.3 在表示稀疏矩阵的三元组行单链 LSparseMatrix 类中，增加以下功能：

1) 稀疏矩阵的转置矩阵。

2) 两个稀疏矩阵相加。

3) 两个稀疏矩阵相乘。

5.4 定义用双链表示的广义表的结点类与广义表类。

实习 5

1. 实习目的：理解稀疏矩阵的表示及操作实现。
2. 题意：在表示稀疏矩阵的三元组顺序存储结构中，实现稀疏矩阵的基本运算操作。