电子信息学院

算法与数据结构
(基于现代C++的方法及实践)
ALGORITHM & DATA STRUCTURE IN MODERN C++

第8章 数组与广义表

王文伟 Wang Wenwei, Dr.-Ing.
Tel: 189-71562600
Email: wwang@aliyun.com
课程QQ群：珞珈EIS数据结构与算法，668792335

电子信息学院 Table of Contents

本章位置

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。

电子信息学院 Table of Contents

8.0 Introduction

- 数组是一种基本而重要的数据集合类型，它是由一组具有相同类型的元素组成的集合，元素依次存储于一个连续的内存空间。数组是其他数据结构实现顺序存储的基础。
  - 一维数组可以看成一个顺序存储结构的线性表。
  - 二维数组可以定义为“数组的数组”。
- 矩阵一般采用二维数组存储，但特殊矩阵和稀疏矩阵可采用特殊方法进行压缩存储。
- 广义表是一种复杂的数据结构，它是线性表结构的扩展。

8.1 数组

8.1.1 一维数组 8.1.2 二维数组 8.1.3 在C++中自定义矩阵类

- **数组**（array）是一组相同类型的数据元素的集合，其元素依次存储于一个地址连续的内存空间中。
- 数组元素在数组中的位置称为**数组的下标（index）**，通过下标，可以找到元素的物理存储地址，从而访问该元素。逻辑上，数组可以看成二元组<下标，值>的一个集合。<键，值>的集合称为哈希表。
- 数组下标的个数就是**数组的维数**，有一个下标的数组是一维数组，有两个下标的就是二维数组，以此类推。
- C++支持一维数组、多维数组（矩形数组）。C++（函数内）的数组在栈中分配存储空间，但数组变量的作用同指针，数组元素的下标从零开始。通过使用指针变量和new操作符可以在堆中开辟和使用数组。

8.1.1 一维数组

- 一维数组是由n（$n>0$）个相同类型的数据元素a_0, a_1, …, a_{n-1}构成的，占用一块地址连续的内存空间的有限序列，记作：

  Array = { a_0, a_1, a_2, …, a_{n-1}}

  其中n称为数组长度。
- 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。通过数组名+下标的形式，可以访问数组中任意一个指定的数组元素。第i个数据元素的地址为：

$$\mathrm{Addr}(a_i) = \mathrm{Addr}(a_0) + (i-0) \times c$$

数组是一种随机存储结构，对数组元素进行随机存放的时间复杂度为O(1)。

两种为数组分配内存空间的方式

- 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所需的空间。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。
  例：int a[10]; //C语言
- 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空间。当数组使用完成之后，需要向系统归还所占用的内存空间。

```
int* a; //C语言
a = (int*)malloc(10*sizeof(int));
a = new int[10]; //C++
```

```
int[] a; //C#
a = new int[10];
```


C++中的一维数组

- 声明数组：<类型> 数组名[len]，例：int a[10];
- 声明数组的同时进行初始化，例：int a[] ={1,2,3,4,5};
- 对于元素数量事先未知且可能多变的情况，常以动态内存分配的方式声明和使用数组。数组是通过指针操作的对象，必须进行实例化。只有用new操作符为数组分配空间后，数组才真正占有实在的存储单元：
  <指针变量名> = new <类型>[len]，例：int* p= new int[10];
- 两种方式下，数组均占用一片地址连续的存储空间，数组长度一旦确定即不可改变。
- 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>]。例如：a[3];

C++中的一维数组(II)

- 数组具有随机访问特性：数组的每个元素占据相同大小的存储空间，任一元素的地址可以通过数组的首地址和元素的下标计算出来。因此，访问某一元素所需的时间与该元素的位置以及数组的大小没有关系，数组的这种特性称为随机访问特性。
- 数组既可基于下标又可基于迭代器遍历
- C++标准库算法模块中包括许多用于排序、搜索和复制数组的方法，例如：copy、find、for_each和transform等模板函数。

【例】 数组的搜索与排序

```
#include <algorithm>
const int CNT = 5; using namespace std;
int main(int argc, char* argv[]) {
    double d[] = { 3.6, 7.5, 1.1, 2.3, 5.0 };
    double* pd = find(d, d + CNT, 5.0);
    if(pd<d+CNT)cout<<"数组d包含数值:"<< 5.0 << endl;
    pd = max_element(d, d + CNT);
    if(pd<d+CNT)cout<<"数组d中的最大值是:"<<*pd << endl;
    cout<<"Sorted Array: "; sort(d,d+CNT);
    for(const double elem:d)
        cout<< elem << " ";
    cout<<endl; }
```

8.1.2 二维数组

- 多维数组是一维数组的推广，二维数组可看作“其元素为一维数组”的数组，它可以表示一个矩阵。二维数组需要两个下标来确定具体元素的位置。

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

- 矩阵$\mathbf{A}_{m\times n}$由$m\times n$个元素$a_{i,j}$组成，可看成是由m行一维数组组成的数组，或是n列一维数组组成的数组。
- 双重线性表：$A_{m\times n}$中的元素$a_{i,j}$同时属于两个线性表：第i行和第j列的线性表。一般，$a_{i,j}$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$a_{i,j+1}$。但$a_{0,0}$是起点，没有前驱；$a_{m-1,n-1}$是终点，没有后继。

二维数组的顺序存储结构

- 二维数组可以按行优先或列优先的次序进行顺序存储。
- 按行优先次序存储二维数组$\mathbf{A}_{m\times n}$，则元素$a_{i,j}$的地址计算函数为：

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

Pascal C/C++/C# Java

- 按列优先次序存储，则元素$a_{i,j}$的地址计算函数为：

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

FORTRAN Matlab

- 可见二维数组的顺序存储结构也是随机存储结构，可以对数组元素进行随机存放，对数组元素进行随机存放的时间复杂度为O(1)。

多维数组的遍历

- 按照某种次序访问一个数据结构中的所有元素，并且每个元素恰好访问一次，称为对数据结构的遍历。遍历一种数据结构，将得到一个由所有元素组成的线性序列。
- 一维数组只有一种基本遍历次序，而二维数组则有两种基本遍历次序：
  - **行优先次序**：对二维数组依行序逐行访问每个数据元素。将数组元素按行排列，第i+1行紧跟在第i行后面。
  - **列优先次序**：对二维数组依列序逐列访问每个数据元素。将数组元素按列排列，第j+1列紧跟在第j列后面。

$a_{0,0}$, $a_{0,1}$, …, $a_{m-1,0}$, $a_{m-1,1}$, …, $a_{m-1,n-1}$
$a_{0,0}$, $a_{1,0}$, …, $a_{0,n-1}$, $a_{1,n-1}$, …, $a_{m-1,n-1}$

C++的多维数组

- 用说明多个下标的形式来定义多维数组，例如：
  int items [3][2];
  声明了一个二维数组items，并分配3×2个存储单元。
- 可以声明并初始化多维数组，例如：
  int numbers[][2] = { {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。

$a_{0,0}$, $a_{0,1}$, …, $a_{0,n-1}$, $a_{1,0}$, $a_{1,1}$, …, $a_{1,n-1}$,
…, $a_{m-1,0}$, $a_{m-1,1}$, …, $a_{m-1,n-1}$

- 例8.1 自定义Matrix类，封装矩阵：
  - 成员_items是一个一维int数组。
  - 多个构造函数，以方便构造和初始化矩阵对象。
  - 对'+'和'*'运算符进行了重载，以提供完成两个矩阵的相加或相乘操作的简洁形式。
  - 矩阵的转置等……。

Matrix c = a + b;

```
class Matrix{
private: int _rows, _cols; int* _items;
public:
  Matrix(int nRows,int nCols,const int* mat){
    int* p=(int*)mat;_rows=nRows; _cols=nCols;
    int es=_rows*_cols; _items=new int[es];
    for(int i=0; i<es;i++)_items[i]=*p++; }
…… }
```

```
//copy constructor. 构造矩阵，它复制另一个矩阵
Matrix(const Matrix& a);
// copy assignment operator,复制另一个矩阵
const Matrix& operator=(const Matrix& rhs);
// move constructor
Matrix(Matrix&& om);
// move assignment operator
Matrix& operator = (Matrix&& rhs);
// 析构函数
~Matrix() {delete[] _items;}
int Rows() const{    return _rows;}
int Cols() const {return _cols;}
const int get(int i, int j) const{
    return _items[i * _cols + j];}
int& set(int i, int j){ return _items[i * _cols + j];}
```

Matrix b…
x= b.get(2,3);

```
Matrix operator+(const Matrix& A,
          const Matrix& B) {
  Matrix C(A);
  for(int i=0;i<A.Rows();i++)
    for(int j=0;j<A.Cols();j++)
      C.set(i,j)=C.get(i,j)+B.get(i,j);
  return C;
}
```

Matrix c = a + b

从函数中返回一个（复合类型的）矩阵对象，其过程往往伴随多个低效重复的数据拷贝负荷。为了提高效率，在矩阵类设计中应用了现代C++引入的移动语义，定义了移动构造函数及重载移动赋值运算符。自动调用

MatrixTest测试、应用Matrix类

```
int m1[]={ 1, 2, 3 ,4, 5, 6 , 7, 8, 9 };
Matrix a(3, 3, m1); a.show();
int m2[]={1, 0, 0, 0, 1, 0, 0, 0, 1 };
Matrix b(3, 3, m2); b.show();
Matrix c = a + b;    c.show();
int d1[] = { 1, 2, 3, 4, 5, 6, 7, 8};
Matrix d(2, 4, d1); d.show();
d.Transpose(); d.show();
d.Transpose(); d.show();
Matrix x = a * c; x.show();
```

程序运行结果

a

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

b

$$\begin{matrix}1&0&0\\0&1&0\\0&0&1\end{matrix}$$

c=a+b

$$\begin{matrix}2&2&3\\4&6&6\\7&8&10\end{matrix}$$

d = c+b

$$\begin{matrix}3&2&3\\4&7&6\\7&8&11\end{matrix}$$

8.2 稀疏矩阵

- 设矩阵$\mathbf{A}_{m\times n}$中有t个非零元素，则称$\delta = t/(m\times n)$为矩阵的稀疏因子，当$\delta \le 0.1$时，称为稀疏矩阵(sparse matrix)。
- 在存储稀疏矩阵时，可以只存储其中的非零元素。这种方式可以压缩掉零元素的存储空间，但往往也会失去数组的随机存取特性。

以下三角矩阵为例: 压缩并保持随机存取性

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

非零元素具有某种分布规律

- 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（$i\ge 1$）行元素的个数为：

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

- 下三角元素$a_{i,j}$（$i\ge j$）的地址可用下式计算：

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

8.2.1 稀疏矩阵的三元组

- 如果稀疏矩阵中非零元素分布没有规律，要压缩存储，基本方法是只存储非零元素。
- 非零元素由三部分组成：行下标、列下标和元素值，这称为稀疏矩阵的非零元素三元组。一个稀疏矩阵可由其三元组集合唯一地确定。
- 表示所有非零元素的三元组集合，可以用顺序存储结构或链式存储结构存储。

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

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

8.2.2 稀疏矩阵三元组集合的顺序存储结构

- 按照行优先（或列优先）的原则，将稀疏矩阵三元组存储在一个线性表中。

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

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

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

声明三元组类

```
struct TripleEntry {
  int row;          //行下标
  int column;       //列下标
  int data;         //值
  TripleEntry(int i, int j, int k) {
    row = i; column = j; data = k; }
  ...   }
```

- TripleEntry对象表示稀疏矩阵的一个三元组实例，用来记录稀疏矩阵中的一个（非零）元素的行列位置及其值。

基于三元组顺序存储结构的稀疏矩阵类

◆SSparseMatrix类表示基于三元组顺序存储结构的稀疏矩阵，其成员_items是一个用线性表表示的动态数组，元素类型为三元组类TripleEntry。本类的构造函数将一个常规矩阵转换成基于三元组顺序存储结构的表示法。

```
class SSparseMatrix{
private:
  int _rows, _cols;
  vector<TripleEntry> _items;//三元组线性表
  ......}
```


构造函数

```
SSparseMatrix(int nRows,int nCols, const int* mat) {
  cout << "稀疏矩阵（二维数组）:" << endl;
  int* p = (int*)mat; int v;
  _rows = nRows; _cols = nCols;
  for (int i = 0; i < _rows; i++) {
    for (int j = 0; j < _cols; j++) {
      v = *p; cout << "  " << v;
      if(v!=0)_items.push_back(TripleEntry(i, j, v));
      p++; }
    cout << endl;  }   }
```


【例8.2】测试基于三元组顺序存储结构的稀疏矩阵类

```
#include "../dsa/SSparseMatrix.h"
using namespace std;
int main() { // SSparseMatrixTest.cpp
  int mat[][4]={{1,0,0,0},{0,0,0,0},{2,0,0,3},{0,4,0,5}};
  SSparseMatrix ssm(4,4,mat[0]);//稀疏矩阵对象
  ssm.show();
```

```
稀疏矩阵（二维数组）:
  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
```


顺序存储结构的缺点

◆用一个动态数组保存稀疏矩阵的三元组序列，它适合于非零元素的数目发生变化的情况。

◆插入、删除操作不方便。若矩阵元素的值发生变化，一个零元素变为非零元素，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。

8.2.3 稀疏矩阵三元组集合的链式存储结构

◆以链式存储结构存储稀疏矩阵的三元组集合便于插入和删除操作。

◆常用的链式存储结构有两种：

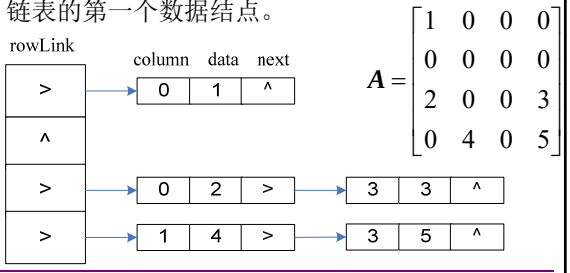
➢行的单链表示

➢列的单链表示

➢十字链表示

行的单链表示方法

◆将稀疏矩阵每行上的非零元素作为结点链接成一个单向链表，而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数据结点。



链表结点：三元组结点结构LinkedTriple

- 该结构由3个成员组成：column（列下标），data（值）和next（后继结点指针）。LinkedTriple对象表示链表中的一个结点，对应矩阵某行的一个非零元素。

```
struct LinkedTriple {
  int column;      //列下标
  int data;        //值
  LinkedTriple* next;
  LinkedTriple(int j, int k) {
    column=j; data=k; next= nullptr; }
  LinkedTriple(): LinkedTriple(-1,0) { }
};
```


稀疏矩阵的行的单链表示类

- 类LSparseMatrix实现稀疏矩阵行的单链表示，成员_rowLink是一个指针数组，其元素指向每条链表的第1个结点。

```
class LSparseMatrix{
private:
  LinkedTriple** _rowLink;
  int _rows, _cols;
  LSparseMatrix(int nRows,int nCols,int* mat){...}
  void show(){... }
}
```


构造函数

```
LSparseMatrix(int nRows,int nCols, const int* mat) {
  _rows = nRows; _cols = nCols;
  _rowLink = new LinkedTriple*[_rows];
  LinkedTriple *p, *q;
  int* pmat = (int*)mat; int v;
  for (int i = 0; i < _rows; i++) {
    p = _rowLink[i] = nullptr;
    for (int j = 0; j < _cols; j++) {
      v = *pmat++;
      if(v!=0){q = new LinkedTriple(j, v);
        if (p == nullptr) _rowLink[i] = q;
        else p->next = q;
        p = q; } } }  }
```

【例8.3】稀疏矩阵单链表示

```
#include "../dsa/LSparseMatrix.h"
using namespace std;
int main() { // LSparseMatrixTest.cpp
  int mat[][4]={{1,0,0,0},{ 0,0,0,0},{2,0,0,3}, {0,4,0,5} };
  LSparseMatrix lsm(4, 4, mat[0]);
  lsm.show();
  return 0;
}
```


程序运行结果

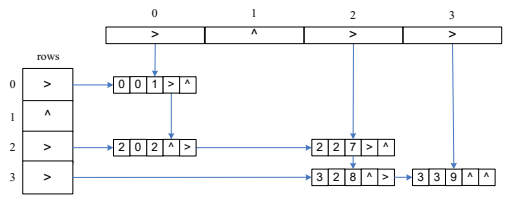
$$A=\begin{bmatrix}1&0&0&0\\0&0&0&0\\2&0&0&3\\0&4&0&5\end{bmatrix}$$

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

- 按行的单链表示的稀疏矩阵，存取一个元素的时间复杂度为$O(n)$。
- 每个结点可以很容易地找到行的后继结点，但很难找到列的后继结点。

十字链表示

- 每个结点表示一个非零元素。结点有5个成员：行下标，列下标，值，行后继引用以及列后继引用。
- 各行的非零元素和各列的非零元素都分别链接在一起，最多有$m+n$条链。



8.3 广义表

- 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）。
- 广义表是一种复杂的数据结构，它是线性表结构的扩展，其元素或为原子或为子表，可以表示多层次的结构。

8.3.1. 广义表的概念及定义

- 广义表是n（$n\geq 0$）个数据元素$a_0, a_1, \ldots, a_{n-1}$组成的有限序列，记为：

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

- 其中，a_i或为不可分的单元素（称为**原子**），或为可再分的**广义表**（称为**子表**）。
- 广义表可以表示多层次的结构，它是用递归的方式进行定义的。

广义表示例

```
L＝(1, 2)          //常规线性表，长度为2
T＝(3, L)＝(3, (1, 2) )//L为T的子表，T的长度为2
L1＝( )            //空表，长度为0
L2＝(L1)＝(( ))//非空表，元素是一个子表，L2的长度为1
G＝(4, L, T)＝(4, (1, 2), (3, (1, 2)))
                   //L、T为G的子表，G的长度为3
Z＝(e, Z)＝(e, (e, (e, (...))))//递归表，Z的长度为2
```

- 在表示广义表时，可以将表名写在对应的括号前：

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(...))))

8.3.2 广义表的特性和操作

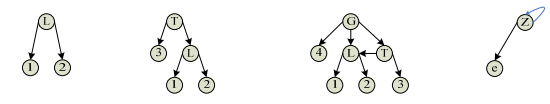
特性

- 广义表可作为其他广义表的子表元素。共享或引用，避免在母表中重复列出子表的值。表T和G共享子表L。
- 广义表是一种多层次的结构。例如T(3, L (1, 2))表示一种树形的层次结构。树中的叶结点对应广义表中的原子，非叶结点对应子表。
- 广义的线性结构。即同层次数据元素之间有着固定的相对次序。线性表是广义表的特例，而广义表则是线性表的扩展。
- 广义表可以是一个递归表。广义表中有共享或递归成分的子表就是图结构。
- 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**。 递归表 > 再入表 > 纯表 > 线性表

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：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：在广义表中插入一个数据元素。
  - Remove：删除一个数据元素。
  - Equals：判别两个广义表是否相等。
  - Copy：复制一个广义表。

8.3.3 广义表的图形表示



(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)

- 线性表：数据元素全部是原子，用原子结点表示。
- 树结构：数据元素中有原子，也有子表，但没有共享和递归成分，该广义表T为纯表。
- 图结构：数据元素中有子表，并且有共享成分，该广义表G为再入表。
- 图结构：数据元素中有子表且有递归成分时，该广义表Z为递归表。

8.3.4 广义表的存储结构

- 线性表有顺序存储结构和链式存储结构两种，层次结构的广义表通常采用链式存储结构。树结构和图结构的存储结构表示将在相关章节中讨论。此处简要说明广义表链式存储结构的一般方法。
  - 广义表的单链表示
  - 广义表的双链表示

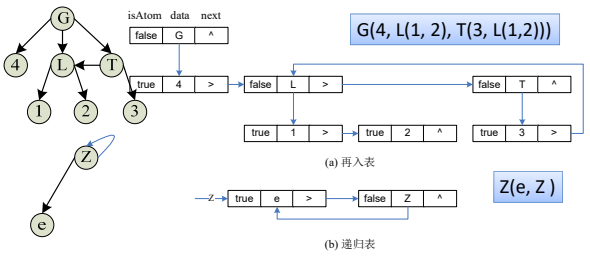
广义表的单链表示

isAtom	data	next

```
struct GSLinkedNode {
  bool isAtom;
  unsigned int data;
  GSLinkedNode* next;
  ......
}
```

- 当isAtom等于true时，表明data存放本原子的信息；当isAtom等于false时，data存放子表第一个数据元素所对应结点的引用。
- next成员存放与本数据元素处于同层的下一个数据元素结点的指针。

广义表的单链表示



- 再入表G中有子表L和T，T中也有子表L，在图中子表L的结点仅出现一次，但被引用两次。

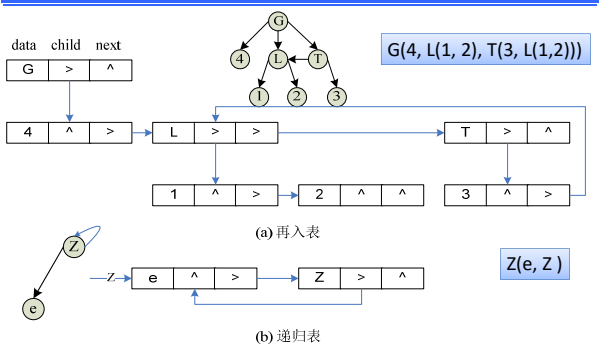
广义表的双链表示

child	data	next

```
template <typename T> struct
GDLinkedNode {
  T data;
  GDLinkedNode<T>* child;
  GDLinkedNode<T>* next;
  ......
}
```

- 域data存放数据元素信息。
- child存放子表第一个数据元素所对应结点的指针， child = = null时，表明本结点是原子。
- next存放与本数据元素同层的下一个数据元素所对应结点的指针。

广义表的双链表示



本章学习要点

1. 了解数组类型的特点以及在高级编程语言中的两种存储表示和实现方法，并熟练掌握多维数组在以行为主的存储结构中的地址计算方法。
2. 掌握稀疏矩阵的结构特点及其存储表示方法。
3. 掌握广义表的结构特点及其存储表示方法。

作业

8.1 在Matrix类中增加下列功能：
1. 求一个矩阵的转置矩阵。
2. 两个矩阵相减/相乘。

8.2 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。

8.3 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。

8.4 定义用双链表示的广义表的结点类与广义表类。

实习

- 实验目的
  理解稀疏矩阵的表示及操作实现。
- 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操作。