山东大学　　计算机科学与技术　　学院

数据结构与算法　课程实验报告

学号：201700130033	姓名：　武学伟	班级：　2017 级 2 班
实验题目：数组和矩阵		
实验学时：4	实验日期：　2018.11.4	

实验目的：
掌握稀疏矩阵的描述和操作的实现

软件环境：
Win10home，codeblocks

1. 实验内容（题目内容，输入要求，输出要求）

1.创建稀疏矩阵类，采用行主顺序把稀疏矩阵非 0　元素映射到一维数组中，提供操作：两个稀疏矩阵相加、两个稀疏矩阵相乘、输出矩阵（以通常的阵列形式输出）。

2.键盘输入矩阵的行数、列数；按行输入矩阵的各元素值，建立矩阵；

3.对建立的矩阵执行相加、相乘的操作，输出操作的结果矩阵。

2. 数据结构与算法描述　（整体思路描述，所需要的数据结构与算法）

数据结构：
数组描述的线性表
稀疏矩阵
三元法描述矩阵元素

算法：

加法：

①a=b+c 分别定义变量表示矩阵 a，b，c 的下标，用于表示结果，加法元素 1 和 2

②判断矩阵 b 和 c 的行列数是否匹配，不匹配报错

③遍历 b，c，当行列值匹配时，进行加法，若是不匹配，则将 b，c 的元素直接插入到 a 中

④检查 b，c，若非空，则插入剩余元素

转置：

①交换行列值

②重新排序（及时终止的冒泡排序）

乘法：

①a=b*c，先将 c 转置得到 d，这样可以直接行乘行。

②对每行每列的元素值进行对应的计算，先统计出每行的非零元素数

③行数跳转，累加 b，d 每行的非零元素值需要做乘法的前一行，得到 b，d 做乘法的对应索引值。

④在两层循环中，若是 b 或者 d 的某行非零元素数为零，则不需要计算。在检查每行的元素是否查询完时，可以定义两个遍历记录 b，d 每行剩余的非零元素数，初始值为之前记录的每行的非零元素数，每进行一次操作时，减一，当等于零时，停止查询。

④行数在一开始已经跳转好，所以仅需列数对应即可相乘并累积结果，若不匹

配则比较 b，d 的列值，若是 b 在前，b 向后移动一次，若是 c 在前，c 向后移动一次。

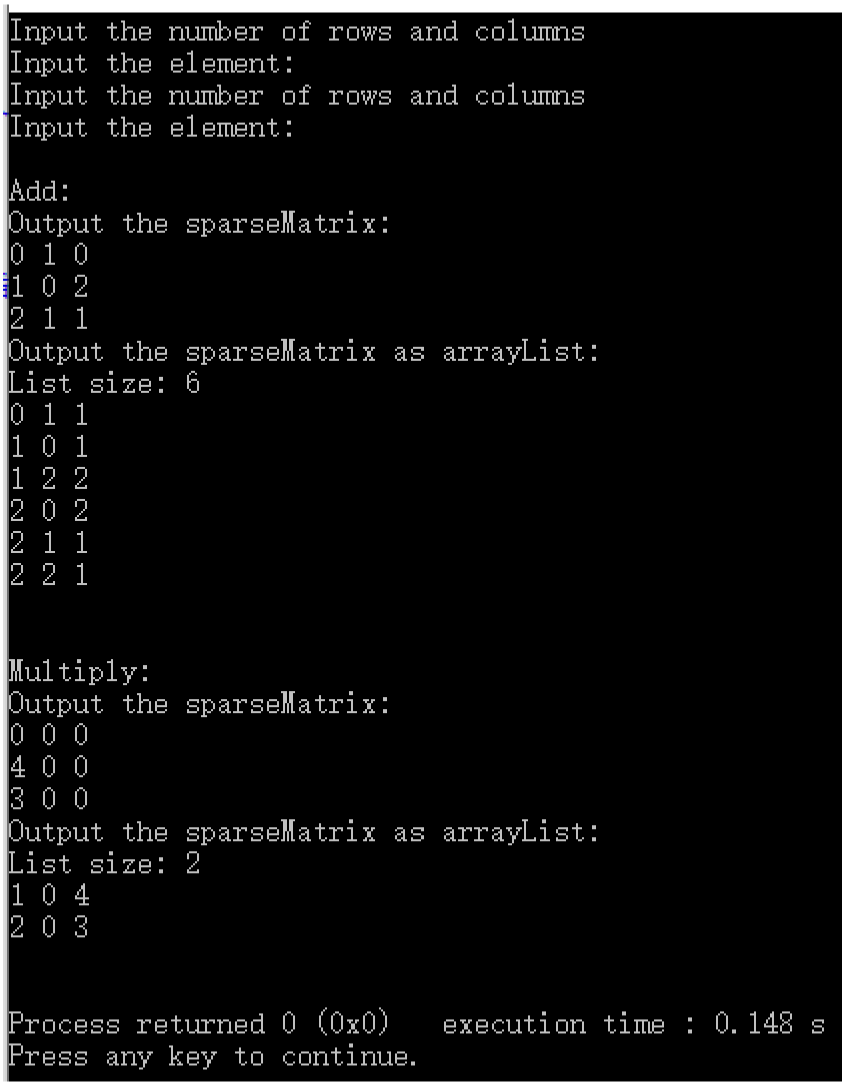
3. 测试结果（测试输入，测试输出，结果分析）

测试一：

输入：in.in 文件

```
3 3
0 0 0
0 0 2
0 1 1
3 3
0 1 0
1 0 0
2 0 0
```

输出：



```
Input the number of rows and columns
Input the element:
Input the number of rows and columns
Input the element:

Add:
Output the sparseMatrix:
0 1 0
1 0 2
2 1 1
Output the sparseMatrix as arrayList:
List size: 6
0 1 1
1 0 1
1 2 2
2 0 2
2 1 1
2 2 1


Multiply:
Output the sparseMatrix:
0 0 0
4 0 0
3 0 0
Output the sparseMatrix as arrayList:
List size: 2
1 0 4
2 0 3


Process returned 0 (0x0)   execution time : 0.148 s
Press any key to continue.
```

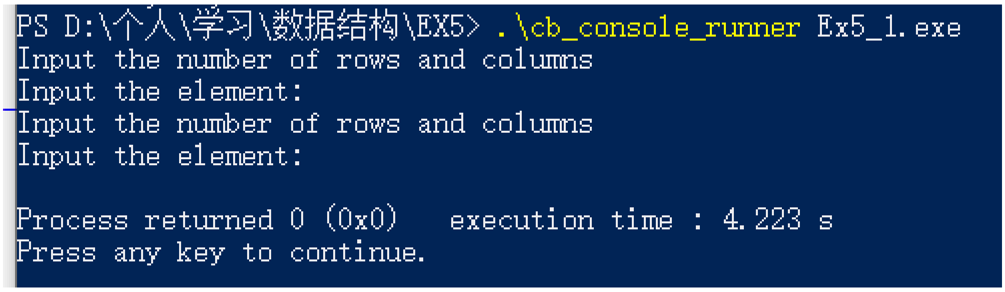
结果正确

测试二：

输入：

data_structure_ex5.in 文件

输出（省去了数据的打印）



```
PS D:\个人\学习\数据结构\EX5> .\cb_console_runner Ex5_1.exe
Input the number of rows and columns
Input the element:
Input the number of rows and columns
Input the element:

Process returned 0 (0x0)   execution time : 4.223 s
Press any key to continue.
```

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

结果分析：

数据正确，但是运行时间较长，应该与数组的容量的扩充有关，每次都加倍，并且复制旧的数据比较耗费时间。

无法重载 operator << 可能和编译器有关

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

文件 1：/*MatrixTerm*/

```
//#include <iostream>
using namespace std;

template <class T>
struct  MatrixTerm      //三元法存储稀疏矩阵中非零元素的行列值以及元素信息
{
    public:
        long long int row;  //行
        long long int col;  //列
        T value;  //元素值

        MatrixTerm()
        {
            row = 0;
            col = 0;
            value = 0;
        }
        MatrixTerm(long long int ro, long long int co, T va) //构造函数
        {
            row = ro;
            col = co;
            value = va;
        }
       void OutputM()   //输出
       {
           cout << row << " " << col << " " << value;
```

```
            cout << endl;
        }
};
文件 2：/*arrayList.h*/
//#include <iostream>
using namespace std;

template <class T>
class arrayList
{
    protected:
        T* Element;
        long long int arrayLength;
        long long int listSize;
        void checkIndex(long long int theIndex) const;  //检查索引值
    public:
        arrayList(long long int initialCapacity = 1000);  //构造函数
        ~arrayList() {delete []Element;}      //析构函数
        arrayList(const arrayList<T>&);       //复制构造函数
        bool Empty() {return listSize==0;}    //判断是否为空
        long long int Size() {return listSize;}         //返回数组的大小
        T& Get(long long int theIndex);                 //获取索引值对应元素
        long long int indexOf(T& theElement);           //获取元素索引值
        void Insert(long long int theIndex, const T& theElement); //插入元素
        void Erase(long long int theIndex);   //删除元素
        void reSet(long long int theSize);   //保证充足的空间
        void Set(long long int theIndex, const T& theElement); //重设索引值
        void OutputA();    //输出线性表的元素
};

/*构造函数*/
template <class T>
arrayList<T>::arrayList(long long int initialCapacity)
{
    if (initialCapacity < 1)
    {
        cout << "wrong capacity" << endl;
        return;
    }
    arrayLength = initialCapacity;
    Element = new T[arrayLength];
    listSize = 0;
}

/*复制构造函数*/
template <class T>
arrayList<T>::arrayList(const arrayList<T>& theList)
{
    arrayLength = theList.arrayLength;
    listSize = theList.listSize;
    Element = new T[arrayLength];
    for (long long int i=0; i<listSize; i++)
        Element[i] = theList.Element[i];
}

/*检查索引*/
template <class T>
void arrayList<T>::checkIndex(long long int theIndex) const
{
    if(theIndex<0 || theIndex>=listSize)
    {
        cout << "error Index" << endl;
        return;
    }
}

/*获取索引值对应元素*/
template <class T>
T& arrayList<T>::Get(long long int theIndex)
{
    checkIndex(theIndex);
    return Element[theIndex];
}

/*获取元素索引值*/
template <class T>
long long int arrayList<T>::indexOf(T& theElement)
{
    for(long long int i=0; i<listSize; i++)
        if(Element[i] == theElement)
            return i;
    return -1;
}

/*改变数组长度*/
template <class T>
void changeLength(T*&a, long long int oldLength, long long int
newLength)
{
    if (newLength < 0)
    {
        cout << "error: wrong capacity" << endl;
        return;
    }
    T* temp = new T[newLength];
    long long int number = min(oldLength,newLength);
    for (long long int i=0; i<number; i++)
        temp[i] = a[i];
    delete []a;
    a = temp;
}

/*插入*/
template <class T>
void arrayList<T>::Insert(long long int theIndex, const T&
theElement)
{
    if (theIndex<0 || theIndex>listSize)
    {
        cout << "wrong index" << endl;
        cout << "index = " << theIndex << " size = " << listSize <<
endl;
        return;
    }   //检查
    if (listSize == arrayLength)
    {
        changeLength(Element, arrayLength, 2*arrayLength);
        arrayLength *= 2;
    }
    for (long long int i=listSize-1; i>=theIndex; i--)//插入位置之
后的元素后移
        Element[i+1] = Element[i];
    Element[theIndex] = theElement;
    listSize++;
}
```

```
/*删除元素*/
template <class T>
void arrayList<T>::Erase(long long int theIndex)
{
    checkIndex(theIndex);
    for (long long int i=theIndex; i<listSize; i++)//元素前移
        Element[i-1] = Element[i];
    Element[--listSize].~T();
}

/*保证充足的空间*/
template <class T>
void arrayList<T>::reSet(long long int theSize)
{
    if (theSize < 0)
    {
        cout << "wrong capacity" << endl;
        return;
    }
    if (theSize > arrayLength)
    {
        delete []Element;
        Element = new T[theSize];
        arrayLength = theSize;
    }
    listSize = theSize;
}

/*重设索引值*/
template <class T>
void arrayList<T>::Set(long long int theIndex, const T& theElement)
{
    if (theIndex < 0 || theIndex >= listSize)
    {
        cout << "wrong index" << endl;
        cout << "index = " << theIndex << " size = " << listSize <<
endl;
        return;
    }//检查
    Element[theIndex]  = theElement;
}

/*输出线性表的元素*/
```

```
template <class T>
void arrayList<T>::OutputA()
{
    for (long long int i=0; i<listSize; i++)
        Element[i].OutputM();
    cout << endl;
}
```

文件 3：/*实验五*/

```
#include <iostream>
#include "MatrixTerm.h"
#include "arrayList.h"
#include <stdio.h>
using namespace std;



template <class T>
class sparseMatrix
{
    private:
        long long int rows;     //稀疏矩阵的行列总数
        long long int cols;
        arrayList<MatrixTerm<T> > terms;     //存放稀疏矩阵的数组
    public:
        void Add(sparseMatrix<T>&, sparseMatrix<T>&);  //加法 a+b=c
        void Multiply(sparseMatrix<T>&, sparseMatrix<T>&);    //乘法 a*b=c
        void Input();     //输入
        void Output();    //输出稀疏矩阵
        void Output_as_arratList();    //按线性表格式输出
        void Transpose(sparseMatrix<T> &b); //矩阵转置
};

/*输入*/
template <class T>
void sparseMatrix<T>::Input()
{
    cout << "Input the number of rows and columns" << endl;
    cin >> rows >> cols;
    T notZero;
    cout << "Input the element:" << endl;
    long long int num = 0;
    for (long long i=0; i<rows; i++)
        for (long long int j=0; j<cols; j++)
```

```
            {
                cin >> notZero;
                if(notZero!=0)  //非零时存储
                {
                    MatrixTerm<T> term(i,j,notZero);
                    terms.Insert(num++,term);
                }
            }
}

/*输出稀疏矩阵*/
template <class T>
void sparseMatrix<T>::Output()
{
    long long int num = 0;
    cout << "Output the sparseMatrix:" << endl;
    for (long long int i=0; i<rows; i++)
    {
        for (long long int j=0; j<cols; j++)
        {
            if (num < terms.Size() && terms.Get(num).row==i &&
terms.Get(num).col==j )
                cout << terms.Get(num++).value << " " ;
            else
                cout << "0" << " " ;
        }
        cout << endl;
    }
}

/*按线性表格式输出*/
template <class T>
void sparseMatrix<T>::Output_as_arratList()
{
    cout << "Output the sparseMatrix as arrayList:" << endl;
    cout << "List size: " << terms.Size() << endl;
    terms.OutputA();
}

/*矩阵转置*/
template <class T>
void sparseMatrix<T>::Transpose(sparseMatrix<T> &b)
{
    cols = b.rows;
    rows = b.cols;  //设置行列特征
    terms = b.terms;
    for (long long int i=0; i<terms.Size(); i++)
    {
        long long int temp = terms.Get(i).row;
        terms.Get(i).row = terms.Get(i).col;
        terms.Get(i).col = temp;
    }
    bool swapped = true;   //及时终止的冒泡排序
    for (long long int i=terms.Size(); i>1&&swapped; i--)
    {
        swapped = false;
        for (long long int j=0; j<i-1; j++)
        {
            if      ((terms.Get(j).row==terms.Get(j+1).row      &&
terms.Get(j).col>terms.Get(j+1).col)                             ||
terms.Get(j).row>terms.Get(j+1).row)
            {//j 的位置应该在 j+1 的位置之后，进行交换
                MatrixTerm<T> temp = terms.Get(j);
                terms.Set(j,terms.Get(j+1));
                terms.Set(j+1,temp);
                //swap(terms[j],terms[j+1]);
                swapped  =  true;    //当无序时进行了交换，swapped 为
真，继续循环
            }
        }
    }
}

/*矩阵加法*/
template <class T>
void sparseMatrix<T>::Add(sparseMatrix<T> &b,sparseMatrix<T> &c)
{
    if (b.rows!=c.rows || b.cols!=c.cols)
    {
        cout << "Not match" << endl;
        return ;
    }
    rows = b.rows;
    cols = b.cols;
    long long int num_a = 0;
    long long int num_b = 0;
    long long int num_c = 0;//分别标记 a，b，c 的数组元素下标
    while (b.terms.Size()>num_b || c.terms.Size()>num_c)
    {
        if  ((b.terms.Get(num_b).row==c.terms.Get(num_c).row  &&
b.terms.Get(num_b).col<c.terms.Get(num_c).col)                ||
b.terms.Get(num_b).row<c.terms.Get(num_c).row)
            terms.Insert(num_a++,b.terms.Get(num_b++));   //b的元素
在前，插入b的元素
        else if ((b.terms.Get(num_b).row==c.terms.Get(num_c).row &&
b.terms.Get(num_b).col>c.terms.Get(num_c).col)                ||
b.terms.Get(num_b).row>c.terms.Get(num_c).row)
            terms.Insert(num_a++,c.terms.Get(num_c++));   //c的元素
在前，插入c的元素
        else   //b和c的元素位置匹配，插入两者之和
        {
            T  sum  =  b.terms.Get(num_b++).value  +
c.terms.Get(num_c++).value;
            MatrixTerm<T> temp;
            temp.row = b.terms.Get(num_b-1).row;
            temp.col = b.terms.Get(num_b-1).col;
            temp.value = sum;
            terms.Insert(num_a++,temp);
        }
        while (c.terms.Size()==num_c && b.terms.Size()!=num_b)
            terms.Insert(num_a++,b.terms.Get(num_b++));   //c中没有
元素
        while (b.terms.Size()==num_b && c.terms.Size()!=num_c)
            terms.Insert(num_a++,c.terms.Get(num_c++));   //b中没有
元素
    }


}

/*矩阵乘法*/
template <class T>
void  sparseMatrix<T>::Multiply(sparseMatrix<T>  &b,  sparseMatrix<T>
&c)
{
    if (b.rows!=c.cols)
    {
        cout << "Not match" << endl;
        return ;
    }
    rows = b.rows;
    cols = c.cols;
    sparseMatrix<T> d;
    d.Transpose(c);   //将 c 转置，这样就可以与 b 行行相乘
    //统计出 b,d 两个矩阵每行有多少非零元素
    long long int b_rows[b.rows] = {0};
    long long int d_rows[d.rows] = {0};
    for (long long int i=0; i<b.terms.Size(); i++)
        b_rows[b.terms.Get(i).row]++;
    for (long long int i=0; i<d.terms.Size(); i++)
        d_rows[d.terms.Get(i).row]++;

    //逐个计算元素值
    long long int no_a = 0;  //数组的下标

    for (long long int i=0; i<rows; i++)
        for (long long int j=0; j<cols; j++)
        {
            long long int no_b = 0;  //数组的下标
            long long int no_d = 0;  //数组的下标
            if (b_rows[i]==0 || d_rows[j]==0)
                continue;
            MatrixTerm<T> temp(i,j,0);  //temp 存储计算值
            for (long long int k=0; k<i; k++)   //使稀疏矩阵跳转到做
乘法的那一行
                no_b += b_rows[k];
            for (long long int k=0; k<j; k++)
                no_d += d_rows[k];
            long long int num_b = b_rows[i];
            long long int num_d = d_rows[j];//b,d 两个矩阵在 i 行剩余
的元素数量
            while (num_b>0 && num_d>0)  //当 b,d 矩阵在这行还有元素
            {
                if (b.terms.Get(no_b).col == d.terms.Get(no_d).col)
                {
                    temp.value    +=    b.terms.Get(no_b).value    *
d.terms.Get(no_d).value;
                    num_b--;
                    num_d--;
                    no_b++;
                    no_d++;
                }
                else if (b.terms.Get(no_b).col <
d.terms.Get(no_d).col)
                {//b 的位置靠前，移动到下一个 b
                    no_b++;
                    num_b--;
                }
                else
                {//d的位置靠前，移动到下一个d
                    no_d++;
                    num_d--;
                }
            }
            if (temp.value != 0)  //非零值插入
                terms.Insert(no_a++,temp);
        }

}

int main()
{
    int uuuu;
    freopen("data_structure_ex5.in","r",stdin);
    sparseMatrix<long long int> a,b,c,d;
    a.Input();
    //a.Output();
    //a.Output_as_arratList();
    b.Input();
    //b.Output();
    //b.Output_as_arratList();
    c.Add(a,b);
    //cout << endl << "Add: " << endl;
    //c.Output();
    //c.Output_as_arratList();
    d.Multiply(a,b);
    //cout << endl << "Multiply: " << endl;
    //d.Output();
    //d.Output_as_arratList();
}
```