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：201700140056	姓名：李港	班级：跟 18.2（17.4）
实验题目：实验五 数组和矩阵		
实验学时：2h	实验日期：2019.10.18	

实验目的：

掌握稀疏矩阵结构的描述及加、乘等操作的实现。

软件开发工具：

Virtual Studio 2019

1. 实验内容

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）

总体思路：

1. 实现默认构造、复制构造、析构功能。
2. “稀疏”的实现：采用三元组表示方式。
3. 加法：采用迭代器，大体思路类似链表合并，先同时对两个加数矩阵的数组循环进行大小判断并赋值给*this；再将剩余非空链表的内容复制到*this 中。
4. 乘法：使用迭代器。首先准备左右矩阵，将右矩阵转置，简化后续代码的编写；然后分别对于每一行每一列的元素进行求值。
5. 枚举常见错误，如下标越界，加乘时两矩阵不匹配等。

数据结构：

1. 采用之前编写的 arrayList 类作为底层数据结构
2. 使用 element 结构存储矩阵元素所在位置的的行、列与数值

```
typedef struct element {
    int row;
    int col;
    T value;
    element () {}
    element ( int irow, int icol, T ivalue )
        :row ( irow ), col ( icol ), value ( ivalue ) {}
} element;
```

算法：

1. “稀疏”的具体实现
   1. `void set ( int irow, int icol, T ivalue );`
      1. 下标判断
      2. 若元素在元素数组中存在，则修改元素数值
      3. 若元素不在元素数组中，且输入值不为 0，则建立元素并赋值；输入为 0 直接返回
      4. 需要注意对 0 值的判断，不能 value==0 就直接返回，有可能矩阵中原来的元素不等于 0，直接返回就不会修改原来的非零数，从而造成错误。

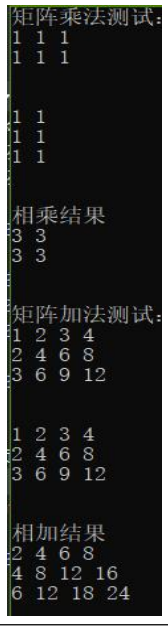
2. T get (int irow, int icol)const;
   1. 下标判断
   2. 若元素在元素数组中存在，则返回其数值
   3. 若元素不在元素数组中，则返回 0
2. 加法：与链表合并有异曲同工之妙。
   1. 使用迭代器。
   2. 对元素分别进行相加与赋值。
   3. 复制剩余元素进行扫尾。
3. 乘法
   1. 判断能否进行相乘。
   2. 准备左右矩阵，清空*this，转置右矩阵以简化后续运算。。
   3. 对每行每列依次使用迭代器对其进行求值。
   4. 迭代器应注意初始化的时机与初始化的值。

稀疏矩阵类的成员及简要注释：

```
public:
  enum  err;                                  //常见错误的枚举
  struct element;                             //三元组
private:
  int _rows;                                  //矩阵行数
  int _cols;                                  //矩阵列数
  arrayList<element > _terms;                 //矩阵三元组数组
  typedef  typename arrayList<element >::iterator arriterator;
                                              //迭代器类型
  bool _checkIndex(int irow, int icol );//检查下标是否越界
  void _swap ( T& _Left, T& _Right )          //交换元素
public:
  spareMatrix() {}                                                    //构造函数
  spareMatrix ( int irow, int icol );                                 //指定大小的构造函数
  void set ( int irow, int icol, T ivalue );                          //设置某元素的值
  T get ( int irow, int icol )const;                                  //获得某元素的值
  void transpose ();                                                  //转置
  void add (const spareMatrix& in );                                  //加法
  void multi(spareMatrix<T> &in);                                     //矩阵乘法
  void input (istream& icin);                                         //从标准输入读取
  friend istream& operator >> (istream& icin, spareMatrix& in);       //重载
  void out (ostream& icout)const;                                     //输出到标准输出
  friend ostream& operator << (ostream& icout,const spareMatrix& in);
                                                                      //重载
  const spareMatrix& operator =  ( const spareMatrix& in );           //重载等号运算符
```

3. 测试结果（测试输入，测试输出）

1.课堂检查：



2.平台提交

题目 1 简单数据：通过

题目 2 复杂数据：通过

4. **分析与探讨（结果分析，若存在问题，探讨解决问题的途径）**

本实验花费时间较长，在实验中遇到过较大困难：

1. 本实验对数组线性表与 C++的类机制进行了更深层次的使用。
2. 在实验过程中遇到了一些问题，其中一部分是对 C/C++语言本身理解的的问题，一部分是编译器相关的问题；对编程语言的使用越深入，就越容易遇到一些古怪问题，如微软实现的函数与自己编写的函数发生冲突等。
3. 在可能导致错误的地方提前进行检查与报错可以极大地方便后续调试。
4. 随着矩阵功能的不断添加，线性表也要增加新的接口，如 reset，“=”重载等。
5. 加、乘等运算符的重载返回矩阵类常量，而重载等号运算符返回矩阵的常量引用：
   1. 在加，乘中，结果的载体既不是左值也不是右值，而是临时建立的矩阵，其存活时长较短，应以值，而不是引用的形式来返回运算结果。
   2. 在重载“=”运算符中，被修改的是左值，该左值从外部传入，生命周期较长，可以以引用的形式返回，以期实现连续等号赋值。
6. 被 const 调用的函数也需要有 const 属性，等号运算符两边都是 const，复制构造函数接受 const 引用不返回值
7. 由于要兼容 C，C++的各种特性都打了折扣，比如编译器明明可以多次分析源码而不必关注所谓“前向声明”问题，但是为了兼容 C，C++必须“假装”自己只便利了一次源码。
8. 各种编译器对语言标准支持不一，如 MSVC 至今不支持变长数组。这一定程度上加大了使用 VS 写代码的难度。
9. 本实验耗时极长
   1. 在超时问题上花费了大量时间。
   2. 重构之后任然有一个测试点错误，多次修改代码才通过。
   3. 测试通过之后仍然没有精确定位错误位置，然而本地测试样例全部通过，只能通过不断修改并提交查看对错的方式一点一点地定位错误，该过程又花费大量时间。
   4. 总结经验，本次实验最大的失误在于最初思路错误，没有采用迭代器；其次是没有保存代码的各个版本，导致有时需要对代码进行重复修改。

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）**

最终版本源码

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

/*class arrayList {
public:
    enum err;                                       //常见错误
    class iterator;                                 //数组迭代器
protected:
    inline void _checkIndex ( int index )const ;    //检查下标是否越界
    void _exLength ();                              扩展数组长度
    T* _head;                                       //数组头指针
    int _maxLength;                                 //数组最大长度
    int _size;                                      //数组元素个数
public:
    arrayList ( int ilength=10 );               //指定长度的构造函数
```

```
    ~arrayList ();                              //析构函数
    arrayList ( const arrayList& ilist );       //拷贝构造函数
    T& get ( int index ) const;                 //获取某元素的值
    void insert (const int index, T in );       //插入元素
    void clear ();                              //清空内容
    const arrayList& operator = ( const arrayList& in );//等号运算符重载
    void set ( int index, const T& in );        //设置某元素值
    int size () const;                          //获取数组长元素个数
    iterator begin () const;                    //返回头部迭代器
    iterator end () const;                      //返回尾部迭代器
*/
template<class T>
class arrayList {
public:
    typedef enum {
        index_out_of_range
    } err;
    class iterator {
    public:
        typedef bidirectional_iterator_tag iterator_category;
        typedef T value_type;
        typedef ptrdiff_t difference_type;
        typedef T* pointer;
        typedef T& reference;

        iterator ( T* ipos ) {
            datap = ipos;
        }

        T& operator*()const {
            return *datap;
        }
        T* operator&()const {
            return datap;
        }
        bool operator!=( const iterator in ) const {
            return datap != in.datap;
        }
        bool operator==( const iterator in )const {
            return datap == in.datap;
        }
        iterator& operator++() {
            ++datap;
            return *this;
        }
        iterator operator++( int ) {
            iterator old = *this;
            ++datap;
            return old;
        }
        iterator operator+( int n ) {
            datap += n;
            return *this;
        }
        iterator& operator--() {
            --datap;
            return *this;
        }
        iterator operator--( int ) {
            iterator old = *this;
            --datap;
            return old;
        }
        iterator operator-( int n ) {
            datap -= n;
            return *this;
        }
```

```
    protected:
        T* datap;
    };
protected:
    inline void _checkIndex ( int index )const {
        if ( index<0 || index>_size ) {
            throw index_out_of_range;
        }
    }
    void _exLength () {
        T* newhead = new T[_maxLength * 2];
        memcpy ( newhead, _head, sizeof ( T ) * ( _maxLength ) );

        _maxLength = _maxLength * 2;

        delete[] _head;
        _head = newhead;
    }
    T* _head;
    int _maxLength;
    int _size;
public:
    ~arrayList () {
        delete[] _head;
    }
    arrayList ( int ilength = 10 ) :_maxLength ( ilength ), _size ( 0 ) {
        _head = new T[_maxLength];
    }

    arrayList ( const arrayList& ilist ) {
        _maxLength = ilist._maxLength;
        _size = ilist._size;
        _head = new T[_maxLength];
        for ( int i = 0; i < _size; i++ ) {
            _head[i] = ilist._head[i];
        }
    }

    T& get ( int index ) const {
        _checkIndex ( index );
        return _head[index];
    }

    void insert ( const int index, T in ) {
        _checkIndex ( index );
        if ( _size == _maxLength ) {
            _exLength ();
        }
        if ( _size - 1 == index ) {
            copy_backward ( _head + index, _head + _size, _head + _size + 1 );
        }
        _head[index] = in;
        _size++;
    }

    void clear () {
        delete[]_head;
        _head = new T[_maxLength];
        _size = 0;
    }

    const arrayList& operator =  ( const arrayList& in ) {
        delete[]_head;
        _maxLength = in._maxLength;
        _size = in._size;
        T* x = in._head;
        _head = new T[_maxLength];
        for ( int i = 0; i < in.size (); i++ ) {
            _head[i] = x[i];
```

```
        }
    }
    void set ( int index, const T& in ) {
        _head[index] = in;
    }

    int size () const {
        return _size;
    }
    iterator begin () const {
        return iterator ( _head );
    }
    iterator end () const {
        return iterator ( _head + _size );
    }
};


/*class spareMatrix
public:
    enum  err;                                  //常见错误的枚举
    struct element;                             //三元组
private:
    int _rows;                                  //矩阵行数
    int _cols;                                  //矩阵列数
    arrayList<element > _terms;                 //矩阵三元组数组
    typedef  typename arrayList<element >::iterator arriterator;//迭代器类型
    bool _checkIndex(int irow, int icol );//检查下标是否越界
    void _swap ( T& _Left, T& _Right )          //交换元素
public:
    spareMatrix() {}                                                    //构造函数
    spareMatrix ( int irow, int icol );                                     //指定大小的构造函数
    void set ( int irow, int icol, T ivalue );                          //设置某元素的值
    T get ( int irow, int icol )const;                                      //获得某元素的值
    void transpose ();                                                      //转置
    void add (const spareMatrix& in );                                      //加法
    void multi(spareMatrix<T> &in);                                         //矩阵乘法
    void input (istream& icin);                                             //从标准输入读取
    friend istream& operator >> (istream& icin, spareMatrix& in);   //重载
    void out (ostream& icout)const;                                         //输出到标准输出
    friend ostream& operator << (ostream& icout,const spareMatrix& in);//重载
    const spareMatrix& operator =  ( const spareMatrix& in );       //重载等号运算符，使用另一个矩
阵重新初始化此矩阵
*/
template<class T>
class spareMatrix {
public:
    typedef enum {
        not_match,
        index_out_of_range
    } err;
    typedef struct element {
        int row;
        int col;
        T value;
        element () {}//如果使用了带参构造函数，则需要显式声明此无参构造函数；带参构造函数中的 T 类型无
法确定类型，不能指定默认形参
        element ( int irow, int icol, T ivalue ) :row ( irow ), col ( icol ), value ( ivalue ) {}
    } element;
private:
    int _rows;
    int _cols;
    arrayList<element > _terms;
    typedef  typename arrayList<element >::iterator arriterator;
    bool _checkIndex ( int irow, int icol ) {
        if ( irow > _rows || irow<0 || icol>_cols || icol < 0 ) {
```

```
            throw index_out_of_range;
        }
        return true;
    }
    void _swap ( T& _Left, T& _Right ) {
        T _Tmp = _Left;
        _Left = _Right;
        _Right = _Tmp;
    }
public:
    spareMatrix () {}
    spareMatrix ( int irow, int icol ) :_rows ( irow ), _cols ( icol ) {}
    ~spareMatrix () {}

    int getRows () const {return _rows;}
    int getCols ()const {return _cols;}
    void set ( int irow, int icol, T ivalue ) {
        _checkIndex ( irow, icol );
        arriterator it = _terms.begin ();
        arriterator itend = _terms.end ();

        //获取插入值应当插入的位置 ,有可能提前终止
        int insert_index = 0;
        int target_index = irow * _cols + icol;//在实际矩阵中的目标位置
        while ( it != itend ) {
            int cur_element_index = ( *it ).row * _cols + ( *it ).col;//当前元素在矩阵中的的实际
位置

            //当前元素位置在目标元素位置前面，目标元素位置应该加一
            if ( cur_element_index < target_index ) {
                insert_index++;
            } else if ( cur_element_index > target_index ) {
                //当前元素位置在目标位置后边，则目标元素应插入到该元素前面
                _terms.insert ( insert_index, element ( icol, irow, ivalue ) );
                return;
            } else if ( cur_element_index == target_index ) {
                //当前元素位置等于目标元素位置，目标元素就在此处
                ( *it ).value = ivalue;
                return;
            }
            it++;
        }


    }
    T get ( int irow, int icol )const {
        _checkIndex ( irow, icol );
        arriterator it = _terms.begin ();
        arriterator itend = _terms.end ();

        //依次循环对比
        while ( it != itend ) {
            if ( ( *it ).row == irow && ( *it ).col == icol ) {
                return ( *it ).value;
            }
            it++;
        }
        return 0;
    }

    void transpose () {
        spareMatrix temp = *( this );
        arriterator it = temp._terms.begin ();
        arriterator end = temp._terms.end ();
        //统计各列非零元素数
        int colindex[_cols + 1];
        memset ( colindex, 0, sizeof ( colindex ) );
        while ( it != end ) {
```

```
            colindex[( *it++ ).col]++;
        }

        //计算转置后各行元素起始位置
        int new_raw_index[_cols + 1];
        new_raw_index[1] = 0;
        for ( int i = 2; i <= _cols; i++ ) {
            new_raw_index[i] = new_raw_index[i - 1] + colindex[i - 1];
        }

        //进行转置
        it = temp._terms.begin ();//重置迭代器
        end = temp._terms.end ();
        while ( it != end ) {
            _swap ( ( *it ).col, ( *it ).row );
            _terms.set ( new_raw_index[( *it ).row]++, *it );
            it++;
        }
        _swap ( _rows, _cols );
    }
    void add ( const spareMatrix& in ) {
        if ( _rows != in.getRows () || _cols != in.getCols () ) {
            throw not_match;
        }
        spareMatrix<T> temp = *( this );
        int insert_index = 0;
        _terms.clear ();

        arriterator lit = temp._terms.begin ();
        arriterator rit = in._terms.begin ();
        arriterator litend = temp._terms.end ();
        arriterator ritend = in._terms.end ();

        while ( lit != litend && rit != ritend ) {
            int lindex = ( *lit ).row * _cols + ( *lit ).col;
            int rindex = ( *rit ).row * _cols + ( *rit ).col;
            if ( lindex > rindex ) {
            //右元素靠后，则插入右元素
                _terms.insert ( insert_index++, *rit++ );
            } else if ( lindex == rindex ) {
            //两项在同一个位置则相加并插入到临时矩阵中
                if ( ( *lit ).value + ( *rit ).value != 0 ) {//提前判断0，提高效率
                    _terms.insert ( insert_index++, element (
                        ( *lit ).row,
                        ( *lit ).col,
                        ( *lit ).value + ( *rit ).value
                    )
                    );
                }
                lit++;
                rit++;
            } else {
            //左元素靠后，则插入左元素
                _terms.insert ( insert_index++, *lit++ );
            }
        }

        //剩下的，未复制完成的那个矩阵继续复制 ，只有一个矩阵会执行如下代码
        while ( rit != ritend ) {
            _terms.insert ( insert_index++, *rit++ );
        }
        while ( lit != litend ) {
            _terms.insert ( insert_index++, *lit++ );
        }
    }
    void multi ( spareMatrix<T>& in ) {
        if ( _cols != in.getRows () ) {
            throw not_match;
```

```
        }

        //首先构造左右矩阵，并清空*this，准备将结果赋给*this
        spareMatrix lm = *this;
        _rows = _rows;
        _cols = in.getCols ();
        _terms.clear ();
        spareMatrix rm = in;
        rm.transpose ();

        //描述没行/列各有多少元素，供迭代器初始化使用
        int each_row_num[_rows + 1] = { 0 };
        int each_col_num[_rows + 1] = { 0 };
        for ( arriterator i = lm._terms.begin (); i != lm._terms.end (); i++ ) {
            each_row_num[( *i ).row]++;
        }
        for ( arriterator i = rm._terms.begin (); i != rm._terms.end (); i++ ) {
            each_col_num[( *i ).row]++;
        }
        arriterator lit = lm._terms.begin ();
        arriterator rit = rm._terms.begin ();

        int insert_index = 0;

        //对于每行
        for ( int insert_row = ( *lit ).row; insert_row <= lm._rows; insert_row++ ) {
            //初始化迭代器
            rit = rm._terms.begin ();
            lit = lit + each_row_num[insert_row - 1];
            //对于每列
            for ( int insert_col = ( *rit ).row; insert_col <= rm.getRows (); insert_col++ ) {
                int sum = 0;

                //计算该行列的结果并赋给*this
                for ( int current_col = 1; current_col <= lm._cols; current_col++ ) {
                    int lvalue = 0;
                    int rvalue = 0;
                    if ( ( *lit ).col == current_col && ( *lit ).row == insert_row ) {
                        lvalue = ( *lit++ ).value;
                    }
                    if ( ( *rit ).col == current_col && ( *rit ).row == insert_col ) {
                        rvalue = ( *rit++ ).value;
                    }
                    sum = sum + lvalue * rvalue;
                }
                lit = lit - each_row_num[insert_row];
                if ( sum != 0 ) {
                    _terms.insert ( insert_index++, element ( insert_row, insert_col, sum ) );
                }
            }
        }
    }
    void input ( istream& icin ) {
        _terms.clear ();
        icin >> _rows >> _cols;
        int tindex = 0;
        element temp;
        for ( int i = 1; i <= _rows; i++ )
            for ( int j = 1; j <= _cols; j++ ) {
                temp.row = i;
                temp.col = j;
                icin >> temp.value;
                if ( temp.value != 0 ) {
                    _terms.insert ( tindex++, temp );
                }
            }
    }
    friend istream& operator >> ( istream& icin, spareMatrix& in ) {
```

```
        in.input ( icin );
        return icin;
    }
    void out ( ostream& icout )const {
        icout << _rows << " " << _cols << endl;
        arriterator it = _terms.begin ();
        for ( int row = 1; row <= _rows; row++ ) {
            for ( int col = 1; col <= _cols; col++ ) {
                if ( ( *it ).row == row && ( *it ).col == col ) {
                    icout << ( *it++ ).value << " ";
                } else {
                    icout << "0" << " ";
                }
            }
            icout << endl;
        }
    }
    friend ostream& operator << ( ostream& icout, const spareMatrix& in ) {
        in.out ( icout );
        return icout;
    }
    const spareMatrix& operator =  ( const spareMatrix& in ) {
        _rows = in._rows;
        _cols = in._cols;
        _terms = in._terms;
    }

};

int main () {
    //FILE* fileh = freopen ( "input.txt", "r", stdin );
    int num = 0;
    int func = 0;
    spareMatrix<int> p, q;
    cin >> num;
    for ( int i = 0; i < num; i++ ) {
        cin >> func;
        switch ( func ) {
        case 1:
            cin >> p;
            break;
        case 2:
            cin >> q;
            if ( p.getCols () != q.getRows () ) {
                p = q;
                cout << "-1" << endl;
            } else {
                p.multi ( q );
            }
            break;
        case 3:
            cin >> q;
            if ( p.getRows () != q.getRows () || p.getCols () != q.getCols () ) {
                p = q;
                cout << "-1" << endl;
            } else {
                p.add ( q );
            }
            break;
        case 4:
            cout << p;
            break;
        }
    }
    return 0;
}
```