**32.下上三角矩阵乘法**

按照题目要求，结果用二维数组存储，使用matrix类存储的代码版本见文末附录。

上下三角矩阵的实现亦见附录。

const int\*\* operator \* ( LowerTriangularMatrix <int> & l, LowerTriangularMatrix<int> & r ) {

/\*首先判断能否相乘\*/

if ( l.getN () != r.getN () ) {

throw not\_match;

}

int size = l.getN ();

/\*建立结果数组\*/

int\*\* result = new int\* [size];

for ( int i = 0; i < size; i++ ) {

result[i] = new int[size];

}

/\*乘积第m行第n列的元素等于左矩阵的第m行元素与右矩阵的第n列对应元素乘积之和\*/

for ( int m = 1; m <= size; m++ ) {

for ( int n = 1; n <= size; n++ ) {

int value = 0;

for ( int i = 1; i <= size; i++ ) {

value += l.get ( m, i ) \* r.get ( i, n );

}

result[m][n] = value;

}

}

return result;

}

复杂度：

O(n^3)

结果矩阵与三角矩阵的赋值取值操作时间复杂度均为O(1)

矩阵乘法本身需要进行三层嵌套循环，即进行n^3次元素加法操作

故总体复杂度为O(n^3)

**34.C形矩阵**

1)给出一个4\*4C形矩阵及其压缩表示方式

1 1 1 1

1 0 0 0

1 0 0 0

1 1 1 1

压缩表示：

使用3n-2长度的一维数组：1 1 1 1 1 1 1 1 1 1

当i==1 L(i,j)=element[j-1]

当i==n L(i,j)=element[2\*n-2+j]

当j==1 L(i,j)=element[n+i-1]

其他情况，L(i,j)=0

2）证明最多有3n-2个非0元素

第一行，最后一行，第一列都无0的情况下非零元素最多，此时两行一列有3n个元素，减去行列重合处的两个元素，即非零元素最多有3n-2个。

**35.反对角矩阵**

1)给出一个4\*4反对角矩阵的例子

0 0 0 1

0 0 1 0

0 1 0 0

1 0 0 0

1. 证明最多有n个非0元素

由 “下标符合 i+j != n+1 的元素值为0” 可知，所有不位于反对角线上的元素都为0

又因为反对角线有n个元素，当这些元素都不为0时，非零元素最多，有n个。

3）用n个元素的一维矩阵表示反对角矩阵

一维数组表示反对角线，由于一行只有一个元素，因此只用行数作为数组的索引-1就可以。

当i+j!=n+1 L(i,j)=0

当i+j==n+1 L(i,j)=element[i-1]

**36.等对角矩阵**

1)证明n\*n等对角矩阵最多有2n-1个不同元素

由题意得，除第一行与第一列外，每个元素都与其左上方元素相等

也即当第一行与第一列元素各不相同时，其数目就是该题所指的最大不同元素数，为2n-1

2）设计映射，将等对角矩阵映射到2n-1长度数组。

当i<=j L(i,j)=element[j-i]

当i>j L(i,j)=element[n+i-j-1]

4)等对角矩阵乘法

按照题目要求，结果用二维数组存储，使用matrix类存储的代码版本见文末附录。

/\*等对角矩阵乘法\*/

const int\*\* operator \* ( EquidiagonalMatrix <int>& l, EquidiagonalMatrix<int>& r ) {

/\*首先判断能否相乘\*/

if ( l.getN () != r.getN () ) {

throw not\_match;

}

int size = l.getN ();

/\*建立结果数组\*/

int\*\* result= new int \* [size];

for ( int i = 0; i < size; i++ ) {

result[i] = new int[size];

}

/\*乘积第m行第n列的元素等于左矩阵的第m行元素与右矩阵的第n列对应元素乘积之和\*/

for ( int m = 1; m <= size; m++ ) {

for ( int n = 1; n <= size; n++ ) {

int value = 0;

for ( int i = 1; i <= size; i++ ) {

value += l.get ( m, i ) \* r.get ( i, n );

}

result[m][n]=value;

}

}

return result;

}

两个等对角矩阵相乘，其结果不一定是等对角矩阵，所以结果矩阵应用普通矩阵存储

复杂度同32题，为O(n^3)

**41.稀疏矩阵的赋值与取值**

使用数组线性表的find函数需要为node编写==运算符重载，此题选择手动判断相等而不采用find函数

get方法：

(由于零值判断是类型相关的，编写适合所有数据类型的零值判断函数可能需要用到type\_traits，较为复杂，所以此处默认类型T均为数值类型，其零值为数字零)

/\*获取值\*/

T get(int r, int c)const {

/\*如果矩阵没有非0元素，则直接返回0\*/

if (terms.getlength() == 0) {

return 0;

}

/\*寻找元素并返回值\*/

for (int i = 0; i < terms.getlength(); i++) {

if (terms[i].col == c && terms[i].row == r) {

return terms[i].num;

}

}

/\*如果找不到元素，则直接返回0\*/

return 0;

}

set方法：

/\*设置某位置元素的值\*/

void set(int r, int c, T value) {

/\*检查下标是否越界\*/

\_checkrc(r, c);

if (terms.getlength() == 0) {//如果矩阵还没有非0元素

if (value == 0) {//如果要赋值0，则不进行操作，直接返回

return;

}else {//如果赋的数值不为0，则建立新结点并加入到terms中

node temp;

temp.col = c;

temp.row = r;

temp.num = value;

terms.push(temp);

return;

}

/\*如果矩阵已经有非0元素，则在terms中找到相应行、列的元素并修改\*/

}else {

for (int i = 0; i < terms.getlength(); i++) {

if (terms[i].col == c && terms[i].row == r) {

/\*如果元素要赋的值为0，则从vector中删除该元素\*/

if (value == 0) {

terms.del(i);

/\*如果元素要赋的值不为0，则赋值\*/

}else {

terms[i].num = value;

}

return;

}

}

/\*如果数组中找不到该元素\*/

if (value == 0) {//如果元素要赋的值为0，则直接返回

return;

}else {//如果元素要赋的值不为0，则直接加入该新元素

node temp;

temp.col = c;

temp.row = r;

temp.num = value;

for ( int i = 0; i < terms.getlength ();i++ ) {

if ( terms[i].col > temp.col ) {

terms.insert ( i, temp );

return;

}

}

terms.push(temp);

return;

}

}

}

设terms当前长度为n；

最坏情况下，每次赋值或取值都要遍历整个数组进行检索，遍历检索复杂度为O(n)；

除insert外，其他操作复杂度均为O(1)；

insert函数与寻找元素的代码的复杂度呈现互补状态：因为元素越靠前越容易找到，但insert也就需要移动更多元素，元素靠后则相反；所以寻找元素耗时，则insert不耗时，寻找元素不耗时，则insert耗时。此处复杂度总和仍然为O(1)。

故两个函数复杂度均为O(n)，n代表terms当前长度。

**第8章 9.1**

typedef enum { pointer\_is\_null, newLength\_less\_than\_zero } arrayStack\_err;

/\*改变容量的函数\*/

void changeLength ( T\*& p, int oldLength, int newLength ) {

/\*错误检查\*/

//新长度小于0则报错

if ( newLength < 0 ) {

throw newLength\_less\_than\_zero;

}

//指针为空则报错

if ( p == NULL || p == nullptr ) {

throw pointer\_is\_null;

}

/\*进行复制\*/

T\* temp = new T[newLength];

memcpy ( temp, p, min ( oldLength, newLength ) \* sizeof ( T ) );

delete[] p;

p = temp;

}

void pop () {

if ( stackTop == -1 )

throw stackEmpty ();

stack[stackTop--].~T ();

/\*用于缩小stack空间的代码，判断并进行空间缩小操作\*/

if ( size () < arrayLength / 4 ) {

changeLength ( stack, arrayLength, arrayLength/2 );

}

}

**附录：**

上下三角矩阵乘法matrix版：

/\*矩阵乘法\*/

const Matrix<int> operator \* ( LowerTriangularMatrix<int>& l, UpperTriangularMatrix<int>& r ) {

/\*首先判断能否相乘\*/

if ( l.getN () != r.getN () ) {

throw not\_match;

}

int size = l.getN ();

/\*建立临时数组，拥有左值的行数与右值的列数\*/

Matrix<int> temp ( size );

/\*乘积第m行第n列的元素等于左矩阵的第m行元素与右矩阵的第n列对应元素乘积之和\*/

for ( int m = 1; m <= size; m++ ) {

for ( int n = 1; n <= size; n++ ) {

int value = 0;

for ( int i = 1; i <= size; i++ ) {

value += l.get ( m, i ) \* r.get ( i, n );

}

temp.set ( m, n, value );

}

}

return temp;

}

等对角矩阵乘法 matrix版：

/\*对角矩阵乘法\*/

const Matrix<int> operator \* ( EquidiagonalMatrix <int>& l, EquidiagonalMatrix<int>& r ) {

/\*首先判断能否相乘\*/

if ( l.getN () != r.getN () ) {

throw not\_match;

}

int size = l.getN ();

/\*建立临时数组\*/

Matrix<int> temp ( size );

/\*乘积第m行第n列的元素等于左矩阵的第m行元素与右矩阵的第n列对应元素乘积之和\*/

for ( int m = 1; m <= size; m++ ) {

for ( int n = 1; n <= size; n++ ) {

int value = 0;

for ( int i = 1; i <= size; i++ ) {

value += l.get ( m, i ) \* r.get ( i, n );

}

temp.set ( m, n, value );

}

}

return temp;

}

上下三角矩阵的实现：

template<class T>

class LowerTriangularMatrix {

private:

T\* arr;

int n;

bool \_checkIndex ( int i, int j ) {

cout << i << " " << j << endl;

if ( i < 1 || j < 1 || i > n||j>n) {

throw index\_err;

}

}

public:

typedef enum err{ index\_err }err;

explicit LowerTriangularMatrix (int in) {

n = in;

arr = new T[n\* ( n + 1 ) / 2];

}

~LowerTriangularMatrix () {

delete[] arr;

}

T get ( int i, int j ) {

\_checkIndex ( i, j );

if ( i < j ) {

return 0;

}

return arr[i \*( i - 1 ) / 2 + j - 1];

}

void set ( int i, int j,T item ) {

\_checkIndex (i,j);

if ( i < j ) {

return;

}

cout << i \* ( i - 1 ) / 2 + j - 1 << endl;

arr[i\* ( i - 1 ) / 2 + j - 1]=item;

}

int getN ()const { return n; }

};

template<class T>

class UpperTriangularMatrix {

private:

T\* arr;

int n;

bool \_checkIndex ( int i, int j ) {

cout << i << " " << j << endl;

if ( i < 1 || j < 1 || i > n || j>n ) {

throw index\_err;

}

}

public:

typedef enum err { index\_err }err;

explicit UpperTriangularMatrix ( int in ) {

n = in;

arr = new T[n\* ( n + 1 ) / 2];

}

~UpperTriangularMatrix () {

delete[] arr;

}

T get ( int i, int j ) {

\_checkIndex ( i, j );

if ( i > j ) {

return 0;

}

cout << n \* ( n + 1 ) / 2 - ( n - i + 2 ) \* ( n - i + 1 ) / 2 + ( j - i );

return arr[n \* ( n + 1 ) / 2 - ( n - i + 2 )\*( n - i + 1 ) / 2 + ( j - i )];

}

void set ( int i, int j, T item ) {

\_checkIndex ( i, j );

if ( i > j ) {

return;

}

cout << n \* ( n + 1 ) / 2 - ( n - i + 2 ) \* ( n - i + 1 ) / 2 + ( j - i ) << endl;

arr[n \* ( n + 1 ) / 2 - ( n - i + 2 )\*( n - i + 1 ) / 2 + ( j - i )] = item;

}

int getN ()const { return n; }

};

等对角矩阵的实现：

template<class T>

class EquidiagonalMatrix {

private:

T\* arr;

int n;

bool \_checkIndex ( int i, int j ) {

cout << i << " " << j << endl;

if ( i < 1 || j < 1 || i > n || j>n ) {

throw index\_err;

}

}

public:

typedef enum err { index\_err }err;

explicit EquidiagonalMatrix ( int in ) {

n = in;

arr = new T[2 \* n - 1];

}

~EquidiagonalMatrix () {

delete[] arr;

}

T get ( int i, int j ) {

\_checkIndex ( i, j );

if ( i <= j ) {

return arr[j - i];

}

if ( i > j ) {

return arr[n + i - j - 1];

}

}

void set ( int i, int j, T item ) {

\_checkIndex ( i, j );

if ( i <= j ) {

arr[j - i] = item;

}

if ( i > j ) {

arr[n + i - j - 1] = item;

}

}

int getN ()const { return n; }

};

matrix实现：

template<class T>

class Matrix {

private:

T\*\* arr;

int n;

bool \_checkIndex ( int i, int j )const {

cout << i << " " << j << endl;

if ( i < 1 || j < 1 || i > n || j>n ) {

throw index\_err;

}

}

public:

typedef enum err { index\_err }err;

explicit Matrix ( int in ) {

n = in;

arr = new T \* [in];

for ( int i = 0; i < in; i++ ) {

arr[i] = new T[in];

}

}

Matrix ( Matrix& raw ) {

n = raw.getN ();

arr = new T \* [n];

for ( int i = 0; i < n; i++ ) {

arr[i] = new T[n];

}

for ( int r = 0; r < n; r++ ) {

for ( int c = 0; c < n; c++ ) {

set ( r + 1, c + 1, raw.get ( r + 1, c + 1 ) );

}

}

}

~Matrix () {

for ( int i = 0; i < n; i++ ) {

delete[] arr[i];

}

delete[] arr;

}

const T get ( int i, int j ) const {

\_checkIndex ( i, j );

return arr[i - 1][j - 1];

}

void set ( int i, int j, T item ) {

\_checkIndex ( i, j );

arr[i - 1][j - 1] = item;

}

int getN ()const { return n; }

/\*输出矩阵，传入输出流的引用\*/

void show ( ostream& out )const {

for ( int r = 0; r < n; r++ ) {

for ( int c = 0; c < n; c++ ) {

out << arr[r][c] << " ";

}

out << endl;

}

out << endl;

}

/\*= 运算符重载，该函数与复制构造函数同功能\*/

const Matrix<int>& operator = ( const Matrix<int>& raw ) {

for ( int i = 0; i < n; i++ ) {

delete[] arr[i];

}

delete[] arr;

n = raw.getN ();

arr = new T \* [n];

for ( int i = 0; i < n; i++ ) {

arr[i] = new T[n];

}

for ( int r = 0; r < n; r++ ) {

for ( int c = 0; c < n; c++ ) {

set ( r + 1, c + 1, raw.get ( r + 1, c + 1 ) );

}

}

return \*this;

}

};