

## 数据结构实验报告

张恒硕      2212266      智科 2 班

实验目标：实现稠密矩阵和稀疏矩阵的矩阵类，包含访问元素、加法、乘法、转置操作等。

### 一、稀疏矩阵

实验原理：

稀疏矩阵是一种特殊类型的矩阵，其中大部分元素都是零，其存储可采用特殊的压缩技术，以减少存储非零元素所需的空间，从而提高内存使用效率，三元组是一种常用方法。

在三元组中，稀疏矩阵中的每一个非零元素由一个三元组(i,j,a<sub>ij</sub>)唯一确定。其中，i 和 j 分别表示该非零元素所对应的行下标和列下标，a<sub>ij</sub> 表示该元素的值。为了压缩存储稀疏矩阵，我们只需要存储这些非零元素以及它们的行下标和列下标，而不需要存储大量的零元素。这样就可以大大减少存储空间的需求，并且可以在处理稀疏矩阵时提高效率。

代码：

```
#include<iostream>
using namespace std;
class element { //三元组类
public:
    int x;
    int y;
    int value;
    void display() { //输出三元组
        cout << "(" << x + 1 << ", " << y + 1 << ", " << value << ")" << endl;
    }
};
class matrix { //矩阵类
private:
    int row;
    int col;
    int size;
    element* ele;
public:
    matrix() {
        row = col = size = 0;
        ele = NULL;
    }
    matrix(int m, int n, int num) { //初始化
        row = m;
        col = n;
        size = num;
        ele = new element[size];
        for (int i = 0; i < size; i++) {
```

```

        ele[i].x = -1;
        ele[i].y = -1;
        ele[i].value = 0;
    }
}

void SetElement(int m, int n, int val){ //在适当位置插入元素
    for (int i = 0; i < size; i++){ //如果输入的是某个已存在的元素则将其覆盖
        if (m == ele[i].x && n == ele[i].y){
            ele[i].value = val;
            return;
        }
    }
    int i;
    for (i = 0; i < size && ele[i].value != 0; i++){ //不存在则找到正确位置插入
        if (m < ele[i].x || (m == ele[i].x && n < ele[i].y)){
            break;
        }
    }
    for (int j = size - 1; j >= i; j--){ //将后面所有元素后移一位
        ele[j] = ele[j - 1];
    }
    ele[i].x = m;
    ele[i].y = n;
    ele[i].value = val;
}

int GetElement(int m, int n) { //获取适当位置的元素
    for (int i = 0; i < size; i++) {
        if (ele[i].x == m && ele[i].y == n) {
            return ele[i].value;
        }
    }
}

void Print(){ //输出
    int num = 0;
    while (ele[num].value != 0 && num < size){ //先打印一维数组中元素的存储次序
        ele[num].display();
        num++;
    }
    for (int i = 0; i < row; i++){
        for (int j = 0; j < col; j++){
            for (int k = 0; k < size; k++){
                if (i == ele[k].x && j == ele[k].y){
                    cout << ele[k].value << " ";
                    break;
                }
            }
        }
    }
}

```

```

        }
        if (k == size - 1) {
            cout << 0 << " ";
        }
    }
}
cout << endl;
}
}

matrix transpose() { //转置，先将矩阵从行次序变列次序，在转置
    int mark = 0;
    matrix ret(col, row, size);
    for (int i = 0; i < col && mark < size; i++) {
        for (int j = 0; j < size; j++) {
            if (ele[j].y == i) {
                ret.ele[mark] = ele[j];
                mark++;
            }
        }
    }
    for (int i = 0; i < ret.size; i++) {
        int temp = 0;
        temp = ret.ele[i].x;
        ret.ele[i].x = ret.ele[i].y;
        ret.ele[i].y = temp;
    }
    return ret;
}

friend matrix Addition(matrix& m1, matrix& m2);
friend matrix Multiplication(matrix& m1, matrix& m2);
};

matrix Addition(matrix& m1, matrix& m2) { //矩阵加法
    matrix ret(m1.row, m1.col, m1.row * m1.col);
    if (m1.col != m2.col || m1.row != m2.row) {
        cout << "无法进行加法" << endl;
    }
    else {
        for (int i = 0; i < ret.size; i++) {
            ret.ele[i].value = m1.ele[i].value + m2.ele[i].value;
        }
    }
    return ret;
}

matrix Multiplication(matrix& m1, matrix& m2) { //矩阵乘法

```

```

matrix ret(m1.row, m2.col, m1.row * m2.col);
if (m1.col != m2.row) {
    cout << "无法进行乘法" << endl;
}
else {
    for (int i = 0; i < m1.size; i++) {
        for (int j = 0; j < m2.size; j++) {
            if (m1.ele[i].y == m2.ele[j].x) {
                element temp{ m1.ele[i].x, m2.ele[j].y, m1.ele[i].value *
m2.ele[j].value };

                bool check = false;
                for (int k = 0; k < ret.size && ret.ele[i].x != -1; k++) {
                    if (ret.ele[k].x == temp.x && ret.ele[k].y == temp.y) {
                        ret.ele[k].value += temp.value;
                        check = true;
                        break;
                    }
                }
                if (!check) {
                    ret.SetElement(m1.ele[i].x, m2.ele[j].y, m1.ele[i].value *
m2.ele[j].value);
                }
            }
        }
    }
}

return ret;
}

int main() {
    cout << "矩阵的左上角第一个元素的位置输出为 (1, 1) , 但输入时为 (0, 0) " << endl;
    matrix a(4, 5, 7);
    a.SetElement(0, 2, 2);
    a.SetElement(1, 3, 3);
    a.SetElement(0, 0, 1);
    a.SetElement(3, 4, 7);
    a.SetElement(2, 1, 4);
    a.SetElement(3, 2, 6);
    a.SetElement(2, 2, 5);
    cout << "矩阵A:" << endl;
    a.Print();
    cout << "矩阵A (2, 4) 位置上的元素:" << endl;
    cout << a.GetElement(1, 3)<<endl;
    matrix b(5, 3, 5);
    b.SetElement(0, 1, 1);

```

```
b.SetElement(1, 2, 2);
b.SetElement(3, 1, 5);
b.SetElement(2, 2, 4);
b.SetElement(2, 0, 3);
cout << "矩阵B:" << endl;
b.Print();
cout << "和:" << endl;
matrix ret1 = Addition(a, b);
ret1.Print();
cout << "积:" << endl;
matrix ret2 = Multiplication(a, b);
ret2.Print();
cout << "转置:" << endl;
matrix ret3 = a.transpose();
ret3.Print();
return 0;
}
```

运行结果：

```
Microsoft Visual Studio × + -
矩阵的左上角第一个元素的位置输出为 (1, 1) , 但输入时为 (0, 0)
矩阵A:
(1,1,1)
(1,3,2)
(2,4,3)
(3,2,4)
(3,3,5)
(4,3,6)
(4,5,7)
1 0 2 0 0
0 0 0 3 0
0 4 5 0 0
0 0 6 0 7
矩阵A (2, 4) 位置上的元素:
3
矩阵B:
(1,2,1)
(2,3,2)
(3,1,3)
(3,3,4)
(4,2,5)
0 1 0
0 0 2
3 0 4
0 5 0
0 0 0
和:
无法进行加法
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
积:
(1,1,6)
(1,2,1)
(1,3,8)
(2,2,15)
(3,1,15)
(3,3,28)
(4,1,18)
(4,3,24)
6 1 8
0 15 0
15 0 28
18 0 24
转置:
(1,1,1)
(2,3,4)
(3,1,2)
(3,3,5)
(3,4,6)
(4,2,3)
(5,4,7)
1 0 0 0
0 0 4 0
2 0 5 6
0 3 0 0
0 0 0 7
C:\Users\zhs20\Desktop\Study\数据结构 (刘进超, 大二上) \数据结构\稀疏矩阵\x64\Debug\稀疏矩阵.exe (进程 3076)已退出, 代码为 0。
按任意键关闭此窗口...
```

实现操作:

构造函数 (初始化和在指定位置插入元素一起完成)

获取适当位置的元素

输出 (三元组形式+矩阵形式)

转置

加法

乘法

分析:

本代码实现了要求的操作, 但是在转置部分与课堂所讲的快速转置不同, 本方法代码简单, 但时间复杂度高。

以下给出出自自己理解的快速转置代码。

```
void sort() { //排序
    int a[max_size] = { 0 };
```

```

    for (int i = 0; i < number; i++) { //统计各列元素数
        a[value_[i].row]++;
    }
    for (int i = 2; i <= rows; i++) { //统计该列及之前列总元素个数
        a[i] += a[i - 1];
    }
    value_1[max_size];
    for (int i = 0; i < number; i++) { //排序
        int n = a[value_[i].row - 1]; //定位该列的第一个位置
        if (value_1[n].val == 0) { //空就直接放
            value_1[n] = value_[i];
        }
        else {
            while (value_1[n].col <= value_[i].col && value_1[n].col != -1) { //不
空，先看原位置行数是否小于等于该元素行数，小于等于则看下一个
                n++;
            }
            if (value_1[n].col == -1) { //当到空位时，放入
                value_1[n] = value_[i];
            }
            if (value_1[n].col > value_[i].col) { //当该位置行数大于该元素行数，向
后移空出一个位置
                for (int j = a[value_[i].row]; j > n; j--) {
                    value_1[j] = value_1[j - 1];
                }
                value_1[n] = value_[i];
            }
        }
    }
}
}

```

## 二、稠密矩阵

实验原理：

稠密矩阵通过使用二维数组来存储矩阵元素。每个元素在数组中对应一个特定的位置，其值可以直接通过该位置的数组元素进行访问和修改。在稠密矩阵中，非零元素的存储需要占用内存空间，并且这些元素在内存中的位置是紧密相邻的。这种存储方式可以有效地支持矩阵的各种运算，特别是对于那些大部分元素非零的矩阵。

代码：

```

#include <iostream>
using namespace std;
const int ROWS = 3;
const int COLS = 3;
void add(int A[][COLS], int B[][COLS], int C[][COLS]) { //矩阵加法

```

```

        for (int i = 0; i < ROWS; i++) {
            for (int j = 0; j < COLS; j++) {
                C[i][j] = A[i][j] + B[i][j];
            }
        }
    }

void multiply(int A[][COLS], int B[][COLS], int C[][COLS]) { //矩阵乘法
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            C[i][j] = 0;
            for (int k = 0; k < COLS; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void transpose(int A[][COLS], int AT[][ROWS]) { //矩阵转置
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            AT[j][i] = A[i][j];
        }
    }
}

void printMatrix(int A[][COLS]) { //输出矩阵
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            cout << A[i][j] << " ";
        }
        cout << endl;
    }
}

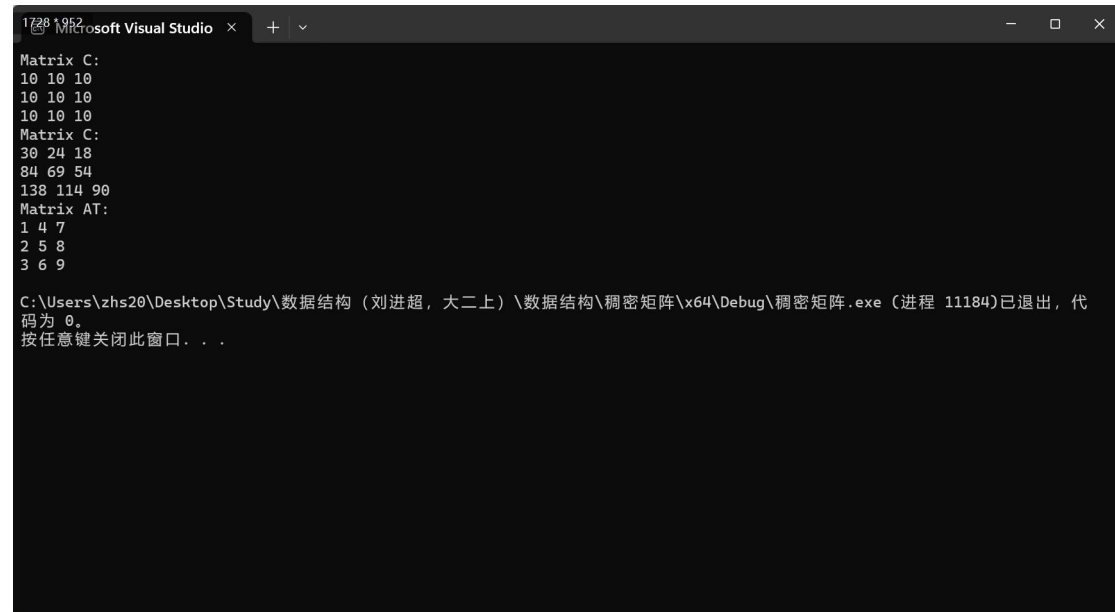
int main() {
    int A[ROWS][COLS] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
    int B[ROWS][COLS] = { {9, 8, 7}, {6, 5, 4}, {3, 2, 1} };
    int C[ROWS][COLS];
    int AT[COLS][ROWS];
    add(A, B, C);
    cout << "Matrix C: " << endl;
    printMatrix(C);
    multiply(A, B, C);
    transpose(A, AT);
    cout << "Matrix C: " << endl;
    printMatrix(C);
    cout << "Matrix AT: " << endl;
}

```



```
    printMatrix(AT);  
    return 0;  
}
```

运行结果：



```
Microsoft Visual Studio x  + v  
Matrix C:  
10 10 10  
10 10 10  
10 10 10  
Matrix C:  
30 24 18  
84 69 54  
138 114 90  
Matrix AT:  
1 4 7  
2 5 8  
3 6 9  
C:\Users\zhs20\Desktop\Study\数据结构（刘进超，大二上）\数据结构\稠密矩阵\x64\Debug\稠密矩阵.exe（进程 11184）已退出，代  
码为 0。  
按任意键关闭此窗口...
```

实现操作：

- 加法
- 乘法
- 转置
- 输出

分析：

本人先前使用二维动态数组实现了本题目，功能更全面，可以用于更多的矩阵而非只限于 3\*3 矩阵，但代码未保存上。因为时间紧迫，只能先提交本简单的代码。