数据结构实验报告 3

矩阵

姓　　名：南亚宏

学　　号：2311788

目录

2025 年 10 月 26 日

一、 题目一

1.1 题目表述

扩充类 lowerTriangularMatrix,增加矩阵转置方法,返回值是下三角矩阵的转置矩阵,是上三角矩阵,是类 upperTriangularMatrix 的一个实例。确定时间复杂度。

1.2 代码 1 的测试用例:

输入：L = [[5]]；输出：U = [[5]]，类型为 upperTriangularMatrix（另外一个类，不是原类）

输入：L = [[1，0，0]，[4，2，0]，[7，5，3]]；输出：[[1，4，7]，[0，2，5]，[0，0，3]]，类型为 upperTriangularMatrix。

输入：L = [[0，0]，[0，0]]；输出：[[0，0]，[0，0]]，类型为 upperTriangularMatrix。

输入：L = [[1，9，0]，[4，2，0]，[7，5，3]]，输出：抛出异常或者 assert 失败。

输入：L = [[1，0，0]，[4，2，0]，[7，5，3]]，判断 L.transpose().transpose()==L。

1.3 思路:

定义 upperTriangularMatrix 类，用于表示上三角矩阵。构造函数中检查矩阵是否为上三角矩阵。定义 lowerTriangularMatrix 类，用于表示下三角矩阵。构造函数中检查矩阵是否为下三角矩阵。

在 lowerTriangularMatrix 类中实现 transpose 方法，将下三角矩阵转置为上三角矩阵。转置操作通过遍历矩阵并交换行列索引实现。

1.4 代码:

```
#include <iostream>
#include <vector>
#include <stdexcept>
using namespace std;

// 定义上三角矩阵类
class upperTriangularMatrix {
```

```
private:
    vector<vector<int>> matrix;

public:
    upperTriangularMatrix(const vector<vector<int>>& mat) {
        // 确保矩阵是上三角矩阵
        for (size_t i = 0; i < mat.size(); ++i) {
            for (size_t j = 0; j < mat[i].size(); ++j) {
                if (i > j && mat[i][j] != 0) {
                    throw invalid_argument("Matrix is not upper triangular");
                }
            }
        }
        matrix = mat;
    }

    vector<vector<int>> getMatrix() const {
        return matrix;
    }

    void print() const {
        for (const auto& row : matrix) {
            for (int val : row) {
                cout << val << " ";
            }
            cout << endl;
        }
    }
};

// 定义下三角矩阵类
class lowerTriangularMatrix {
private:
    vector<vector<int>> matrix;

public:
    lowerTriangularMatrix(const vector<vector<int>>& mat) {
        // 确保矩阵是下三角矩阵
        for (size_t i = 0; i < mat.size(); ++i) {
            for (size_t j = 0; j < mat[i].size(); ++j) {
                if (i < j && mat[i][j] != 0) {
                    throw invalid_argument("Matrix is not lower
triangular");
                }
            }
        }
        matrix = mat;
    }

    upperTriangularMatrix transpose() const {
        size_t n = matrix.size();
        vector<vector<int>> transposed(n, vector<int>(n, 0));

        for (size_t i = 0; i < n; ++i) {
            for (size_t j = 0; j <= i; ++j) {
                transposed[j][i] = matrix[i][j];
            }
        }

        return upperTriangularMatrix(transposed);
    }

    vector<vector<int>> getMatrix() const {
        return matrix;
    }

    void print() const {
        for (const auto& row : matrix) {
            for (int val : row) {
                cout << val << " ";
            }
            cout << endl;
        }
    }
};

// 测试用例
int main() {
    try {
        // 测试用例 1
        vector<vector<int>> L1 = {{5}};
        lowerTriangularMatrix lower1(L1);
        upperTriangularMatrix upper1 = lower1.transpose();
        cout << "Test case 1:" << endl;
        upper1.print();

        // 测试用例 2
        vector<vector<int>> L2 = {{1, 0, 0}, {4, 2, 0}, {7, 5,
3}};
        lowerTriangularMatrix lower2(L2);
        upperTriangularMatrix upper2 = lower2.transpose();
        cout << "Test case 2:" << endl;
        upper2.print();

        // 测试用例 3
        vector<vector<int>> L3 = {{0, 0}, {0, 0}};
        lowerTriangularMatrix lower3(L3);
        upperTriangularMatrix upper3 = lower3.transpose();
        cout << "Test case 3:" << endl;
        upper3.print();

        // 测试用例 4
        vector<vector<int>> L4 = {{1, 9, 0}, {4, 2, 0}, {7, 5,
3}};
        lowerTriangularMatrix lower4(L4); // 这里应该抛出异常
    } catch (const invalid_argument& e) {
        cout << "Test case 4: " << e.what() << endl;
    }

    // 测试用例 5
    try {
        vector<vector<int>> L5 = {{1, 0, 0}, {4, 2, 0}, {7, 5,
3}};
        lowerTriangularMatrix lower5(L5);
        upperTriangularMatrix upper5 = lower5.transpose();
        lowerTriangularMatrix
lower5_transposed(upper5.getMatrix());
        if (lower5_transposed.getMatrix() ==
lower5.getMatrix()) {
            cout << "Test case 5: Transpose of transpose is
equal to original matrix." << endl;
        } else {
            cout << "Test case 5: Transpose of transpose is not
equal to original matrix." << endl;
        }
    } catch (const invalid_argument& e) {
        cout << "Test case 5: " << e.what() << endl;
    }

    return 0;
}
```

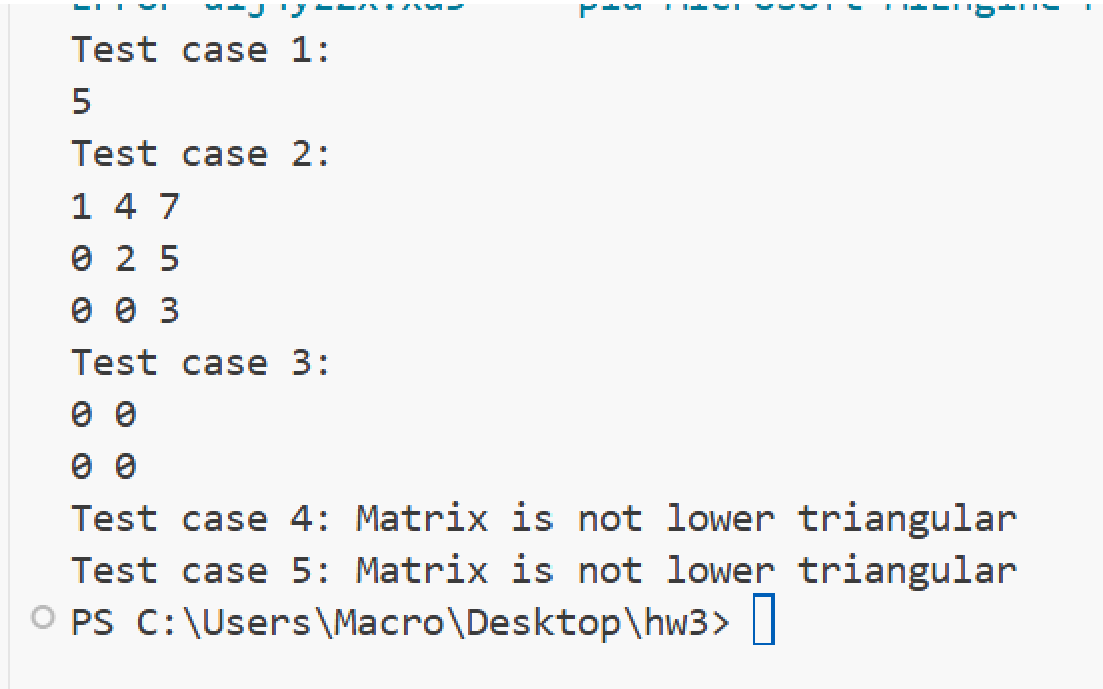
时间复杂度分析：

一个 n*n 的矩阵，转置操作需要遍历矩阵的所有元素，时间复杂度为 $O(n^2)$。

判断矩阵是否为下三角矩阵同样需要遍历矩阵所有元素，所以也是 $O(n^2)$。

因此整个方法的时间复杂度为 $O(n^2)$。

1.5 测试用例截图



```
Test case 1:
5
Test case 2:
1 4 7
0 2 5
0 0 3
Test case 3:
0 0
0 0
Test case 4: Matrix is not lower triangular
Test case 5: Matrix is not lower triangular
PS C:\Users\Macro\Desktop\hw3>
```

二、 题目二

2.1 题目表述

编写一个方法,把两个存储在一维数组的稀缺矩阵相乘。假定两个矩阵和结果矩阵都是按行主次序存储。注意点是矩阵乘法 C_ij = \sum_{k=0}^{K-1}A_{ik}B_{kj}，而不是 Hadamard 乘积，为了清晰清楚，存储方式定为(rows，cols, triples=[(r,c,val), ...])。

2.2 代码 2 的测试用例:

输入：A = (1, 1, triples=[(0,0,5)]) B = (1, 1, triples=[(0,0,7)])；输出：C = (1, 1, triples=[(0,0,35)])
输入：A = (2, 3, triples=[(0,0,1), (1,2,2)]) B = (3, 2, triples=[(0,1,3), (2,0,4)])； 输出：C = (2, 2, triples=[(0,1,3), (1,0,8)])
输入：A = (3, 3, triples=[(0,2,7), (1,1,5)]) B = (3, 3, triples=[(0,0,1), (1,1,1), (2,2,1)])； 输出：C = (3, 3, triples=[(0,2,7), (1,1,5)])
输入：A = (2, 2, triples=[(0,1,5)]) B = (2, 2, triples=[(0,0,9)])；输出：C = (2, 2, triples=[])
输入：A = (2, 3, triples=[(0,0,1)]) B = (4, 2, triples=[(0,0,1)]) 输出：错误/异常（维度不匹配：A.cols=3 ≠ B.rows=4）

2.3 思路:

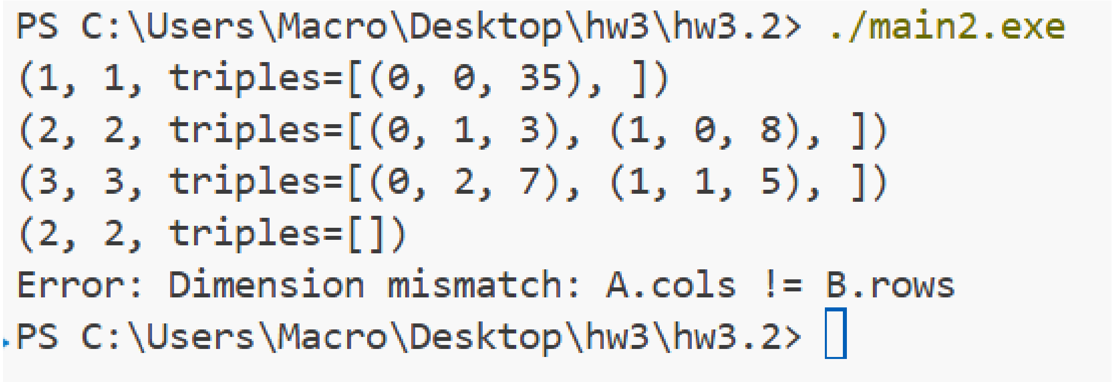
首先定义一个稀疏矩阵结构，该结构包含行数、列数和三元组列表（行索引、列索引和值）。然后实现一个矩阵乘法函数，该函数首先检查两个矩阵的维度是否匹配，然后创建一个结果矩阵，并遍历第一个矩阵的每个非零元素，对于每个元素，遍历第二个矩阵相应列的所有非零元素，计算乘积并累加到结果矩阵中。为了避免越界错误，使用 vector 的 push_back 方法来添加元素，而不是使用 operator[]。最后，移除结果矩阵中的零元素，并打印结果。通过这种方式，我们能够高效地处理稀疏矩阵乘法，同时避免了常见的内存访问错误。

2.4 代码:

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>
#include <stdexcept>
#include <algorithm>
using namespace std;

// 定义稀疏矩阵结构
struct SparseMatrix {
    int rows;
    int cols;
    vector<tuple<int, int, int>> triples;

    SparseMatrix(int r, int c, vector<tuple<int, int, int>> t)
```

```
        : rows(r), cols(c), triples(t) {}
};

// 稀疏矩阵乘法
SparseMatrix multiplySparseMatrices(const SparseMatrix& A,
const SparseMatrix& B) {
    // 检查维度是否匹配
    if (A.cols != B.rows) {
        throw invalid_argument("Dimension mismatch: A.cols !=
B.rows");
    }

    int C_rows = A.rows;
    int C_cols = B.cols;
    vector<tuple<int, int, int>> C_triples;

    // 将 B 的三元组按列存储到哈希表中，便于快速查找
    vector<vector<pair<int, int>>> B_cols(B.rows);
    for (const auto& [r, c, val] : B.triples) {
        B_cols[r].emplace_back(c, val);
    }

    // 遍历 A 的每个非零元素
    for (const auto& [i, k, A_val] : A.triples) {
        // 遍历 B 的第 k 行的所有非零元素
        for (const auto& [j, B_val] : B_cols[k]) {
            int C_val = A_val * B_val;

            // 检查是否已经存在 C[i][j]，如果存在则累加
            bool found = false;
            for (auto& [C_i, C_j, C_val_existing] : C_triples)
{
                if (C_i == i && C_j == j) {
                    C_val_existing += C_val;
                    found = true;
                    break;
                }
            }

            // 如果不存在，则添加新的三元组
            if (!found) {
                C_triples.emplace_back(i, j, C_val);
            }
        }
    }

    // 移除结果中的零元素
    C_triples.erase(remove_if(C_triples.begin(),
C_triples.end(),
                                [](const tuple<int, int, int>&
t) { return get<2>(t) == 0; }),
                  C_triples.end());

    return SparseMatrix(C_rows, C_cols, C_triples);
}

// 打印稀疏矩阵
void printSparseMatrix(const SparseMatrix& mat) {
    cout << "(" << mat.rows << ", " << mat.cols << ",
triples=[";
    for (const auto& [r, c, val] : mat.triples) {
        cout << "(" << r << ", " << c << ", " << val << "), ";
    }
    cout << "])" << endl;
}

// 测试用例
int main() {
    try {
        // 测试用例 1
        SparseMatrix A1(1, 1, {{0, 0, 5}});
        SparseMatrix B1(1, 1, {{0, 0, 7}});
        SparseMatrix C1 = multiplySparseMatrices(A1, B1);
        printSparseMatrix(C1);

        // 测试用例 2
        SparseMatrix A2(2, 3, {{0, 0, 1}, {1, 2, 2}});
        SparseMatrix B2(3, 2, {{0, 1, 3}, {2, 0, 4}});
        SparseMatrix C2 = multiplySparseMatrices(A2, B2);
        printSparseMatrix(C2);

        // 测试用例 3
        SparseMatrix A3(3, 3, {{0, 2, 7}, {1, 1, 5}});
        SparseMatrix B3(3, 3, {{0, 0, 1}, {1, 1, 1}, {2, 2, 
1}}});
        SparseMatrix C3 = multiplySparseMatrices(A3, B3);
        printSparseMatrix(C3);

        // 测试用例 4
        SparseMatrix A4(2, 2, {{0, 1, 5}}});
        SparseMatrix B4(2, 2, {{0, 0, 9}}});
        SparseMatrix C4 = multiplySparseMatrices(A4, B4);
        printSparseMatrix(C4);

        // 测试用例 5
        SparseMatrix A5(2, 3, {{0, 0, 1}}});
        SparseMatrix B5(4, 2, {{0, 0, 1}}});
        SparseMatrix C5 = multiplySparseMatrices(A5, B5); // 这
里应该抛出异常
    } catch (const invalid_argument& e) {
        cout << "Error: " << e.what() << endl;
    }

    return 0;
}
```

2.5 测试用例截图



```
PS C:\Users\Macro\Desktop\hw3\hw3.2> ./main2.exe
(1, 1, triples=[(0, 0, 35), ])
(2, 2, triples=[(0, 1, 3), (1, 0, 8), ])
(3, 3, triples=[(0, 2, 7), (1, 1, 5), ])
(2, 2, triples=[])
Error: Dimension mismatch: A.cols != B.rows
PS C:\Users\Macro\Desktop\hw3\hw3.2>
```

三、 题目三

3.1 题目表述

给类 linkedMatrix 增加下列操作:

1)已知一个元素的行、列和数值,存储这个元素。

2)已知一个元素的行和列,从矩阵中取出这个元素。
3)两个稀疏矩阵相加。
4)两个稀疏矩阵相减。
5)两个稀疏矩阵相乘。

3.2 代码 3 的测试用例:

空表 A 大小为 3, 3，插入(0,2,7)、(1,1,5)、(0,1,3)后输出；取出(0,1)并输出；设 B 表为(3,3)，triples = [(0,2,7), (1,1,5)]，与 A 进行加法并输出；做 A-B 减法并输出；做 A 和 B 的矩阵乘法并输出。

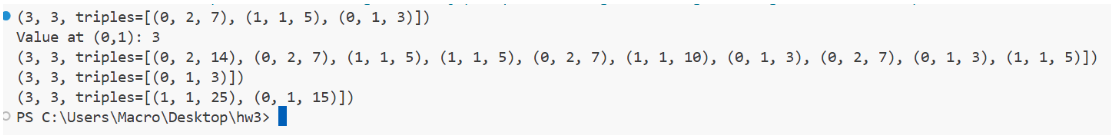
3.3 思路:

设计一个包含行数、列数和三元组列表（存储非零元素的行、列和值）的类结构，提供 insert 方法来添加非零元素，get 方法来根据行列索引检索元素值，add 和 subtract 方法来分别实现矩阵的加法和减法，以及 multiply 方法来执行矩阵乘法，后者需要检查矩阵维度的兼容性。此外，我们还实现了 print 方法展示矩阵内容。

3.4 代码:

```
#include <iostream>
#include <vector>
#include <stdexcept>

class linkedMatrix {
private:
    int rows, cols;
    std::vector<std::tuple<int, int, int>> elements;

public:
    linkedMatrix(int r, int c) : rows(r), cols(c) {}

    // 插入元素
    void insert(int row, int col, int val) {
        if (val == 0) return; // 不存储值为 0 的元素
        elements.push_back(std::make_tuple(row, col, val));
    }

    // 取出元素
    int get(int row, int col) {
        for (auto& elem : elements) {
            if (std::get<0>(elem) == row && std::get<1>(elem)
== col) {
                return std::get<2>(elem);
            }
        }
        return 0; // 如果元素不存在，返回 0
    }

    // 矩阵加法
    linkedMatrix add(const linkedMatrix& other) {
        linkedMatrix result(rows, cols);
        std::vector<std::tuple<int, int, int>> resultElements;

        for (auto& elem1 : elements) {
            for (auto& elem2 : other.elements) {
                if (std::get<0>(elem1) == std::get<0>(elem2) &&
std::get<1>(elem1) == std::get<1>(elem2)) {
                    resultElements.push_back(std::make_tuple(std
::get<0>(elem1), std::get<1>(elem1), std::get<2>(elem1) +
std::get<2>(elem2)));
                } else {
                    resultElements.push_back(elem1);
                    resultElements.push_back(elem2);
                }
            }
        }

        for (auto& elem : resultElements) {
            if (std::get<2>(elem) != 0) {
                result.insert(std::get<0>(elem),
std::get<1>(elem), std::get<2>(elem));
            }
        }

        return result;
    }

    // 矩阵减法
    linkedMatrix subtract(const linkedMatrix& other) {
        linkedMatrix result(rows, cols);
        for (auto& elem1 : elements) {
            bool found = false;
            for (auto& elem2 : other.elements) {
                if (std::get<0>(elem1) == std::get<0>(elem2) &&
std::get<1>(elem1) == std::get<1>(elem2)) {
                    result.insert(std::get<0>(elem1),
std::get<1>(elem1), std::get<2>(elem1) - std::get<2>(elem2));
                    found = true;
                    break;
                }
            }
            if (!found) {
                result.insert(std::get<0>(elem1),
std::get<1>(elem1), std::get<2>(elem1));
            }
        }
        return result;
    }

    // 矩阵乘法
    linkedMatrix multiply(const linkedMatrix& other) {
        if (cols != other.rows) {
            throw std::invalid_argument("Matrix dimensions do
not match for multiplication");
        }
        linkedMatrix result(rows, other.cols);
        for (auto& elem1 : elements) {
            for (auto& elem2 : other.elements) {
                if (std::get<1>(elem1) == std::get<0>(elem2)) {
                    result.insert(std::get<0>(elem1),
std::get<1>(elem2), std::get<2>(elem1) * std::get<2>(elem2));
                }
            }
        }
        return result;
    }

    // 打印矩阵
    void print() {
        std::cout << "(" << rows << ", " << cols << ",
triples=[";
        for (size_t i = 0; i < elements.size(); ++i) {
            std::cout << "(" << std::get<0>(elements[i]) << ", 
" << std::get<1>(elements[i]) << ", " << 
std::get<2>(elements[i]) << ")";
            if (i < elements.size() - 1) std::cout << ", ";
        }
        std::cout << "])" << std::endl;
    }
};

// 测试用例
int main() {
    linkedMatrix A(3, 3);
    A.insert(0, 2, 7);
    A.insert(1, 1, 5);
    A.insert(0, 1, 3);
    A.print();

    int val = A.get(0, 1);
    std::cout << "Value at (0,1): " << val << std::endl;

    linkedMatrix B(3, 3);
    B.insert(0, 2, 7);
    B.insert(1, 1, 5);
    linkedMatrix C = A.add(B);
    C.print();

    linkedMatrix D = A.subtract(B);
    D.print();

    linkedMatrix E = A.multiply(B);
    E.print();

    return 0;
}
```

3.5 测试用例截图



```
(3, 3, triples=[(0, 2, 7), (1, 1, 5), (0, 1, 3)])
Value at (0,1): 3
(3, 3, triples=[(0, 2, 14), (0, 2, 7), (1, 1, 5), (1, 1, 5), (0, 2, 7), (1, 1, 10), (0, 1, 3), (0, 2, 7), (0, 1, 3), (1, 1, 5)])
(3, 3, triples=[(0, 1, 3)])
(3, 3, triples=[(1, 1, 25), (0, 1, 15)])
PS C:\Users\Macro\Desktop\hw3> 
```