数据结构 ： 判别回文字符串 实验报告

毛子恒 李臻 张梓靖

2020 年 10 月 25 日

小组成员

班级：2019211309	姓名：毛子恒	学号：2019211397	分工：代码 文档
班级：2019211310	姓名：李臻	学号：2019211458	分工：测试 文档
班级：2019211308	姓名：张梓靖	学号：2019211379	分工：文档

Contents

1 需求分析

1.1 题目描述

输入稀疏矩阵 A 和 B，检测 A 和 B 能否相加/相乘。
如能，做矩阵相加/相乘运算，并打印运算结果；如不能，显示原因。

1.2 输入描述

程序从标准输入中读入数据。输入两行两个整数，用空格分隔，分别表示矩阵的行数和列数 n,m。
每行整数后紧接着输入 n*m 个整数，用空格隔开，表示稀疏矩阵中的数据。

1.3 输出描述

程序向标准输出中输出结果。
输出分为两种情况：

1. 输入合法，程序正常运行结束。此时输出矩阵运算的结果。
2. 如矩阵不能相加，输出”Cannot add matrix A and B, An != Bn.” 或”Cannot add matrix A and B, An != Bn.”
3. 如矩阵不能相乘，输出”Cannot multiply matrix A and B, Am != Bn.”

1.4 样例输入输出

1.4.1 样例输入输出 1

【输入】

【输出】

1.4.2 样例输入输出 2

【输入】

【输出】

1.4.3 样例输入输出 3

【输入】

【输出】

1.4.4 样例输入输出 4

【输入】

【输出】

1.5 程序功能

程序判别稀疏矩阵 A 和 B 能否相加/相乘，并完成计算。

2 概要设计

2.1 问题解决的思路

2.2 矩阵的定义

```
//数据对象
typedef struct
{
    int i, j, val;
} Tuple;

typedef struct
{
    Tuple * data;
    int * pos; // 每一行中首个非零元素的位置
    int n, m, tot; // 为非空元素总数tot
    int sizeOfMatrix; // 三元组存储单位的个数
} Matrix;

/*
 * 操作：初始化矩阵
 * 前件：指向一个空矩阵，an，>0m>0
 * 后件：指向一个an*的零矩阵m
 */
void initMatrix(Matrix * a, int n, int m);

/*
 * 操作：扩展矩阵的存储空间
 * 前件：指向一个矩阵c
 * 后件：该矩阵扩展个三元组存储单位MATRIXINCREASESIZE
 */
void expandMatrix(Matrix * c);

/*
 * 操作：把由一维数组存储的矩阵转化为由三元组表存储的矩阵
 * 前件：n，>0m，>0中存储一个矩阵，矩阵元素vala[i][j存储在]val[(i−1)*m+j中]
 * 后件：函数返回由三元组表作为存储形式的矩阵
 */
Matrix array2Matrix(int n, int m, int val[]);

/*
 * 操作：把两个矩阵相加
 * 前件：a,为两个矩阵，并且行数和列数都相等b
 * 后件：函数返回两个矩阵相加的结果
```

```
 */
Matrix addMatrix(Matrix a, Matrix b);

/*
 * 操作：把两个矩阵相乘
 * 前件：a,为两个矩阵，并且的列数和的行数相等bab
 * 后件：函数返回两个矩阵相乘的结果
 */
Matrix mulMatrix(Matrix a, Matrix b);

/*
 * 操作：把由三元组表存储的矩阵转化为由一维数组存储的矩阵
 * 前件：为一个矩阵，指向一个至少有avaln*m个存储单位的+1类型的数组int
 * 后件：指向由一维数组存储的矩阵，矩阵元素vala[i][j存储在]val[(i-1)*m+j中]
 */
void matrix2Array(Matrix a, int val[]);

/*
 * 操作：释放矩阵空间
 * 前件：指向一个矩阵a
 * 后件：指向一个空矩阵a
 */
void destroyMatrix(Matrix * a);
```

2.3 主程序的流程

1. 输入，元素进入一维数组
2. 一维数组存储的矩阵转化为由三元组表存储的矩阵
3. 判断矩阵能否相加
4. 输出
5. 判断矩阵能否相乘
6. 输出
7. 释放空间

2.4 各程序模块之间的层次关系

程序模块层次关系图如图 1。

3 详细设计

3.1 栈的实现

三元组设计中基本操作的伪代码算法如下：

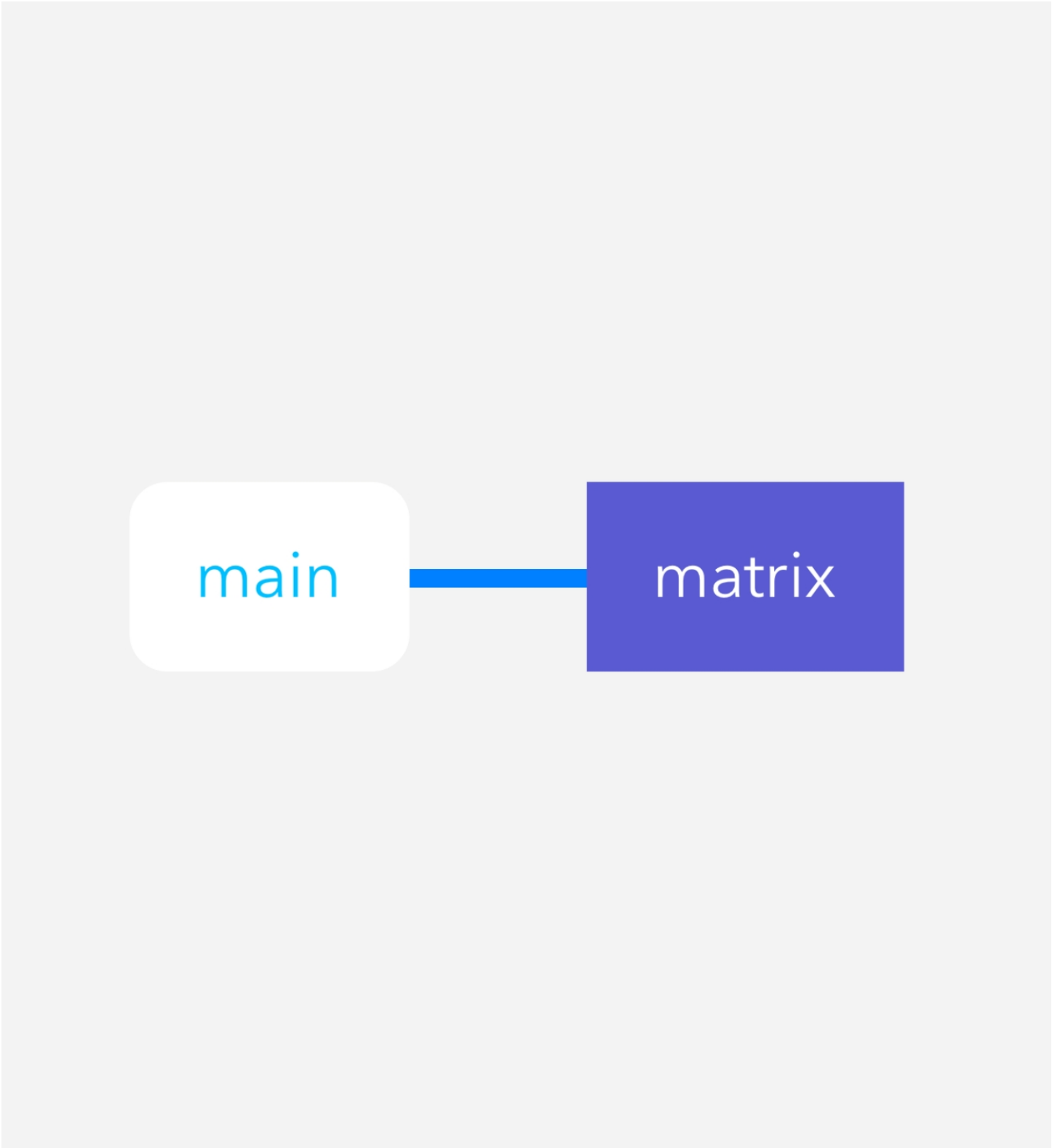

Figure 1: 程序模块层次关系

```
// 初始化矩阵
void initMatrix(Matrix * a, int n, int m)
{
    a—>n <— n;
    a—>m <— m;
    a—>tot <— 0;
    if (a—>内存分配失败pos)异常退出

    a—>sizeOfMatrix <— MATRIXINCREASESIZE;
    if (a—>内存分配失败data)异常退出

}

// 扩展矩阵的存储空间
void expandMatrix(Matrix * c)
{
    // 分配更多存储空间
    if (c—>内存分配失败data)异常退出

    c—>sizeOfMatrix <— c—>sizeOfMatrix + MATRIXINCREASESIZE
}

// 把由一维数组存储的矩阵转化为由三元组表存储的矩阵
Matrix array2Matrix(int n, int m, int val[])
{定义矩阵
     a 并初始化
    for (int i <— 1; i <= n; ++i)
    {
        a.pos[i] <— a.tot + 1
        for (int j <— 1; j <= m; ++j)
        {
            if (val[(i — 1) * m + j不为空])
            {
                if 矩阵(的存储空间不足a)扩展矩阵的存储空间
                    a
                a.data[++a.tot] <— (Tuple) {i, j, val[(i — 1) * m + j]}
            }
        }
    }
    a.pos[n + 1] <— a.tot + 1返回
     a
}

// 两个矩阵相加
Matrix addMatrix(Matrix a, Matrix b)
{
    if 两个矩阵的行列大小不相等()异常退出定义矩阵

```

```
    c 并初始化
    if 有一个矩阵为空()返回空矩阵
        c
    for (int i <- 1; i 小于等于a.n; ++i)
    {
        c.pos[i] <- c.tot + 1定义整型
        p1 <- a.pos[i]定义整型
        p2 <- b.pos[i]
        while (p1 小于a.pos[i + 1] 或p2 小于b.pos[i + 1]) // 枚举矩阵和矩阵第行的非零元素abi
        {定义临时整型
            tempj,tempy // 用于记录矩阵中每一个位置加法的结果c
            if (矩阵本行没有元素或者矩阵非零元素列数小于矩阵非零元素的列数bab)
            {
                c[i][j] <- a[i][j]
            }
            else if (矩阵本行没有元素或者矩阵非零元素列数小于矩阵非零元素的列数aba)
            {
                c[i][j] <- b[i][j]
            }
            else
            {
                c[i][j] <- a[i][j]+b[i][j]
            }
            if (c[i][j不为]0)
            {
                if 矩阵(存储空间不足c)扩展矩阵的存储空间

                c.data[++c.tot] <- (Tuple) {i, tempj, tempv}
        }
    }
    c.pos[c.n + 1] <- c.tot + 1返回
    c
}

// 两个矩阵相乘
Matrix mulMatrix(Matrix a, Matrix b)
{
    if 第一个矩阵的列数不等于第二个矩阵的行数() // 两个矩阵不可相乘异常退出定义矩阵

    c 并初始化
    if 有一个矩阵为空()返回空矩阵
        c定义临时数组并分配内存
    temp // 临时数组，用于记录矩阵中每一行的结果c
    for (int i <- 1; i 小于等于a.n; ++i)
    {
        memset(temp, 0, (a.m + 1) * sizeof(int));
        c.pos[i] <- c.tot + 1;
        for (int p <- a.pos[i]; p 小于a.pos[i + 1]; ++p) // 枚举矩阵第行的非零元素ai
```

```
97        {
98            int k <— a.data[p].j // 矩阵的该非零元素为aa[i][k]
99            for (int q <— b.pos[k]; q 小于b.pos[k + 1]; ++q) // 枚举矩阵第行的非零元素bk
100           {
101               int j <— b.data[q].j // 矩阵的该非零元素为bb[k][j]
102               c[i][j] <— c[i][j] + a[i][k]*b[k][j]
103           }
104       }
105       for (int j <— 1; j 小于等于a.m; ++j)
106       {
107           if (c[i][j不等于]0)继续下一轮循环
108
109               if 矩阵(存储空间不足c)扩展矩阵的存储空间
110
111           c.data[++c.tot] <— (Tuple) {i, tempj, tempv}
112       }
113   }
114   c.pos[c.n + 1] <— c.tot + 1;释放临时数组
115   temp返回
116    c
117}
118
119// 把由三元组表存储的矩阵转化为由一维数组存储的矩阵
120void matrix2Array(Matrix a, int val[])
121{
122   memset(val, 0, sizeof(int) * (a.n * a.m + 1));
123   for (int i <— 1; i 小于等于a.tot; ++i)
124       val[(a.data[i].i — 1) * a.m + a.data[i].j] <— a.data[i].val; // a[i][j存储在]val[(i—1]*m+j中]
125}
126
127// 释放矩阵空间
128void destroyMatrix(Matrix * a)
129{释放
130   a—>pos释放
131   a—>data
132   a—>n <— 0
133   a—>m <— 0
134   a—>tot <— 0
135   a—>sizeOfMatrix <— 0
136}
```

3.2 函数的调用关系图

函数调用关系图如图 2。

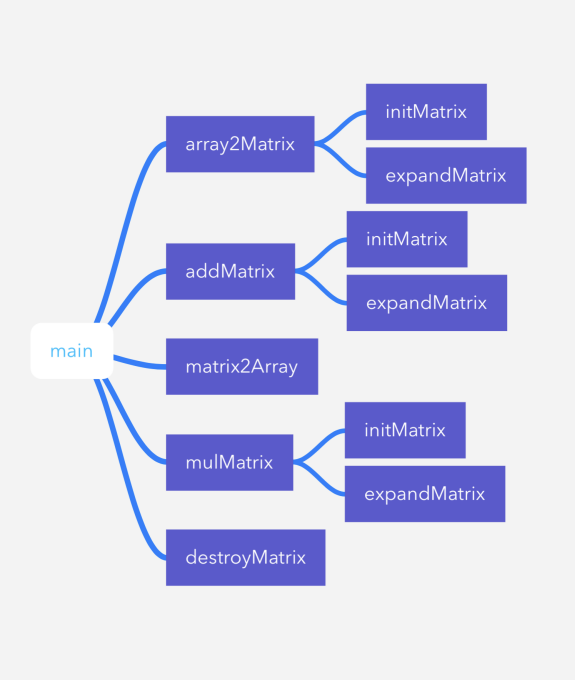

Figure 2: 函数调用关系图

4 调试分析报告

4.1 调试过程中遇到的问题和思考

4.2 设计实现的回顾讨论

4.3 算法复杂度分析

4.4 改进设想的经验和体会

4.4.1 改进 1

5 用户使用说明

5.1 栈和队列的顺序实现

使用 gcc 编译生成可执行文件。

```
gcc -o main -std=c11 main.c stack.c queue.c
```

执行可执行文件：

```
./main
```

在 Windows cmd 下：

```
main
```

之后通过标准输入输入数据，输入格式参考 1.2 节的输入描述，结果通过标准输出返回。如果输入合法并且程序正常运行结束，主函数返回值为 0。

5.2 栈和队列的链表实现

使用 gcc 编译生成可执行文件。

```
gcc -o main -std=c11 main1.c stack.c queuelist.c
```

执行可执行文件：

```
./main
```

在 Windows cmd 下：

```
main
```

之后通过标准输入输入数据，输入格式参考 1.2 节的输入描述，结果通过标准输出返回。如果输入合法并且程序正常运行结束，主函数返回值为 0。

6 测试结果

本题的两种实现方式均通过以下测试。
测试环节分为三个步骤。

6.1 测试第一部分

对 1.4 节给出的样例进行测试。

6.2 测试第二部分

测试边界条件。
【输入】

```
#
```

（认为空串是回文串）
【输出】

```
YES
```

【输入】

```
1#
```

【输出】

```
YES
```

【输入】

```
00#
```

【输出】

```
YES
```

6.3 测试第三部分

将原解法与 4.4.1 节改进解法比对。

测试在 macOS Catalina 10.15.6 下进行。

在 $LEN <= 10$，$LEN <= 1000$，$LEN <= 1000000$ 的范围下分别随机生成 1000 组测试数据，分别传入 main 和 test，并且比对两程序的输出。

3000 组数据中两程序的输出均相同。

数据生成程序 (testing/data.cpp) 如下：

```cpp
#include <bits/stdc++.h>

using namespace std;

const int LEN = 1e6;
char a[LEN + 10];

int main()
{
    srand(time(0));
    int n = rand() % LEN + 1;
    for (int i = 1; i <= n / 2; ++i)
        printf("%c", a[i] = rand() % 26 + 'a');
    if (rand() % 2 == 1)
    {
        for (int i = 1; i <= n / 2; ++i)
            printf("%c", a[i]);
    }
    else
    {
        for (int i = n / 2; i >= 1; --i)
            printf("%c", a[i]);
    }
    puts("#");
    return 0;
}
```

比对脚本 (testing/chk.sh) 如下：

```sh
for i in {1..100}
do
    sleep 1
    ./data >in.in
    ./main <in.in >out.out
    ./test <in.in >out1.out
    if ! diff out.out out1.out
    then
        break
    fi
    echo "Correct"
done
```