2023—2024 学年第二学期

《数据结构与算法导论》实验报告

班级：2023211119

姓名：XXX

学号：2023210XXX

班内序号：07

报告日期：2024-05-12

数据结构实验报告

实验名称： 实验 2——稀疏矩阵

学生姓名： XXX

班　　级： 2023211119

班内序号： 07

学　　号： 2023210XXX

日　　期： 2024 年 5 月 12 日

1．实验要求

[正文格式要求]

字体：汉字宋体、英文 Times New Roman

字号：五号

颜色：黑色

行距：单倍行距

[内容要求]

要求：实现线性表的基本功能

三元组的基本功能：

三元组的建立

三元组转置

三元组相乘

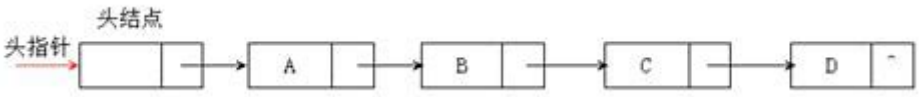
其他

编写测试 main()函数测试三元组的正确性

2. 程序分析

2.1 存储结构

不带头节点的单链表



2.2 关键算法分析

1.关键算法：

a.构造函数

a.1.构造函数 Matrix(int rows, int cols)

```
function Matrix(rows, cols) {
    // 初始化行数为 rows
    this.rows = rows
```

```
        // 初始化列数为 cols
        this.cols = cols
        // 初始化链表的头指针为 nullptr
        this.data = nullptr
    }
```

a.2.从数组生成

```
    function Matrix(data, rows, cols) {
        // 调用基本构造函数初始化行数和列数
        this.rows = rows
        this.cols = cols
        this.data = nullptr
        // 遍历数组 data
        for i from 0 to rows * cols - 1 {
            // 如果元素非零
            if data[i] != 0 {
                // 调用 set 方法添加元素
                set(i / cols, i % cols, data[i])
            }
        }
    }
```

b.析构函数

```
    function ~Matrix() {
        // 使用指针 p 遍历链表
        p = this.data
        while p != nullptr {
            // 保存当前节点到 q
            q = p
            // 移动 p 到下一个节点
            p = p.next
            // 删除当前节点 q
            delete q
        }
    }
```

c.插入元素

```
    function set(row, col, value) {
        // 创建新节点 t
        t = new tuple<T>
        // 设置节点值
        t.x = row
        t.y = col
        t.value = value
        // 如果 data 为空
        if this.data == nullptr {
            // data 指向新节点 t
```

```
            this.data = t
        } else {
            // 找到链表尾部
            p = this.data
            while p.next != nullptr {
                p = p.next
            }
            // 在尾部添加新节点 t
            p.next = t
        }
    }
```

d.乘法

```
    function operator*(other) {
        // 检查是否可以进行矩阵乘法
        if this.cols != other.rows {
            throw "Matrices are not compatible for multiplication."
        }
        // 创建结果矩阵 result
        result = new Matrix<T>(this.rows, other.cols)
        // 遍历当前矩阵的每一行
        for i from 0 to this.rows - 1 {
            // 遍历 other 矩阵的每一列
            for j from 0 to other.cols - 1 {
                // 初始化乘积和为 0
                sum = 0
                // 遍历当前矩阵的列数
                for k from 0 to this.cols - 1 {
                    // 使用指针 p 遍历当前矩阵
                    p = this.data
                    // 使用指针 q 遍历 other 矩阵
                    q = other.data
                    // 找到匹配的行和列的元素
                    while p != nullptr && q != nullptr {
                        if p.x == i && p.y == k && q.x == k && q.y == j {
                            // 计算乘积和
                            sum += p.value * q.value
                            // 移动指针 p 和 q 到下一个节点
                            p = p.next
                            q = q.next
                        } else {
                            // 根据条件移动指针 p 或 q
                            if p.x == i && p.y == k {
                                q = q.next
```

```
                                } else if q.x == k && q.y == j {
                                    p = p.next
                                } else {
                                    p = p.next
                                    q = q.next
                                }
                            }
                        }
                    }
                    // 将乘积和设置到结果矩阵
                    result.set(i, j, sum)
                }
            }
            // 调用 result 的 deleteZero 方法
            result.deleteZero()
            // 返回结果矩阵
            return result
        }
```

e.输出

```
        function print() {
            // 遍历矩阵的每一行
            for i from 0 to this.rows - 1 {
                // 遍历矩阵的每一列
                for j from 0 to this.cols - 1 {
                    // 使用指针 p 遍历链表
                    p = this.data
                    // 找到匹配的行和列的元素
                    while p != nullptr && p.x != i && p.y != j {
                        p = p.next
                    }
                    // 如果找到匹配的行和列
                    if p != nullptr {
                        // 打印元素
                        printf("%3d ", p.value)
                    } else {
                        // 打印 0
                        printf("%3d ", 0)
                    }
                }
                // 打印换行符
                printf("\n")
            }
        }
```

f.转置

```
function transpose() {
    // 创建结果矩阵 result，行列数与当前矩阵转置
    result = new Matrix<T>(this.cols, this.rows)
    // 使用指针 p 遍历链表
    p = this.data
    while p != nullptr {
        // 将当前节点的值设置到 result 矩阵对应的转置位置
        result.set(p.y, p.x, p.value)
        // 移动到下一个节点
        p = p.next
    }
    // 返回结果矩阵
    return result
}
```

g.删除链表里的 0

```
function deleteZero() {
    // 使用指针 p 遍历链表
    p = this.data
    while p != nullptr {
        // 使用指针 q 跟踪前一个节点
        q = p
        // 移动 p 到下一个节点
        p = p.next
        // 如果当前节点的值为 0
        if q.value == 0 {
            // 如果 q 是头节点
            if q == this.data {
                // 更新头节点为下一个节点
                this.data = p
            } else {
                // 更新 q 的前一个节点的 next 为 p
                q.prev.next = p
            }
            // 删除当前节点 q
            delete q
        }
    }
}
```

2.代码详细分析

a.构造函数

a.1.构造函数 1

这个构造函数初始化一个稀疏矩阵对象，设置其行数和列数，并初始化数据指针 data 为 nullptr，表示矩阵开始时是空的。

a.2.构造函数 2

这个构造函数接收一个二维数组的指针，行数和列数作为参数。它首先调用上面的构造函数来初始化行数、列数和数据指针，然后遍历二维数组，如果发现非零元素，就使用 set 方法将它们添加到稀疏矩阵中。

b.析构函数

析构函数用于在对象生命周期结束时清理资源。它遍历链表，删除所有动态分配的 tuple 节点，以防止内存泄漏。

c.插入元素

set 方法创建一个新的 tuple 节点，并将其添加到链表的末尾。如果链表为空，新节点直接成为头节点。

d.乘法

重载了乘法运算符以计算两个稀疏矩阵的乘积。它首先检查两个矩阵是否可以相乘（即第一个矩阵的列数是否等于第二个矩阵的行数）。如果可以，它创建一个新的结果矩阵，并遍历第一个矩阵的每一行和第二个矩阵的每一列，计算乘积。在计算过程中，它使用两个指针分别遍历两个矩阵的链表，寻找匹配的元素进行乘法操作。最后，它调用 deleteZero 方法删除结果矩阵中值为 0 的元素。

e.输出

打印矩阵的当前内容。它遍历矩阵的行和列，使用链表遍历找到非零元素并打印它们。如果某个位置上没有非零元素，它打印 0。

f.转置

计算矩阵的转置。它创建一个新的矩阵对象，其行数和列数与原矩阵互换，并遍历原矩阵的链表，将每个元素的值设置到新矩阵的对应转置位置上。

g. 删除链表里的 0

删除矩阵中所有值为 0 的元素。它遍历链表，检查每个元素的值。如果值为 0，它将从链表中移除该元素。

3.时间和空间复杂度（n 是数组的大小）

a.构造函数

a.1.构造函数 1

时间复杂度：O(1)，空间复杂度：O(1)

a.2.构造函数 2

时间复杂度：O(n)，空间复杂度：O(n)

b. 析构函数

时间复杂度: O(n)，空间复杂度: O(1)

c. 插入元素

时间复杂度: O(n)，空间复杂度: O(1)

d. 乘法

时间复杂度: O(r1 * c1 * r2 * c2)，空间复杂度: O(r1 * r2)

其中 r1 和 c1 是当前矩阵的行数和列数，r2 和 c2 是另一个矩阵的行数和列数

e. 输出

时间复杂度: O(r * c * n)，空间复杂度: O(1)

其中 r 是行数，c 是列数，n 是链表中非零元素的数量

f. 转置

时间复杂度: O(n)， 空间复杂度: O(n)

g. 删除链表里的 0

时间复杂度: O(n)，空间复杂度: O(1)

3. 程序运行结果

测试主函数流程：

```
int main()
{
    // 设置矩阵 A 和 B 的元素
    int arrayA[6][7] = {
            1, 12,  9, 0, 0,  0, 0,
            0,  0,  0, 0, 5,  0, 0,
           -3,  0,  0, 0, 0, 14, 0,
            0,  0, 13, 0, 0,  0, 0,
            0, 18,  0, 0, 0,  0, 0,
           15,  0,  0, 0, 0,  0, 0};
    Matrix<int> A(&arrayA[0][0], 6, 7);
    int arrayB[7][8] = {
      1, 9, 0, 0, 0, 0, 2, 0,
     -1, 0, 0, 0, 3, 0, 0, 0,
      0, 0, 0, 8, 0, 0, 0, 0,
      0, 0, 4, 0, 0, 0, 0, 0,
      0, 7, 0, 0, 0, 0, 0, 1,
      0, 0, 0, 0, 0, 6, 0, 0,
      5, 0, 0, 0, 0, 0, 0, 0,
    };
    Matrix<int> B(&arrayB[0][0], 7, 8);
    // 计算矩阵 A 和 B 的乘积
    Matrix<int> C = A * B;
    // 打印矩阵 C
    A.print();
    printf("\n");
    B.print();
    printf("\n");
    C.print();
    printf("\n");
    Matrix<int> D=C.transpose();
    D.print();
    return 0;
}
```

测试结果：

```
  1   12    9    0    0    0    0
  0    0    0    0    5    0    0
 -3    0    0    0    0   14    0
  0    0   13    0    0    0    0
```

```
  0  18   0   0   0   0   0
 15   0   0   0   0   0   0

  1   9   0   0   0   0   2   0
 -1   0   0   0   3   0   0   0
  0   0   0   8   0   0   0   0
  0   0   4   0   0   0   0   0
  0   7   0   0   0   0   0   1
  0   0   0   0   0   6   0   0
  5   0   0   0   0   0   0   0

-11   9   0  72  36   0   2   0
  0  35   0   0   0   0   0   5
 -3 -27   0   0   0  84  -6   0
  0   0   0 104   0   0   0   0
-18   0   0   0  54   0   0   0
 15 135   0   0   0   0  30   0

-11   0  -3   0 -18  15
  9  35 -27   0   0 135
  0   0   0   0   0   0
 72   0   0 104   0   0
 36   0   0   0  54   0
  0   0  84   0   0   0
  2   0  -6   0   0  30
  0   5   0   0   0   0
```

```
PS C:\Users\18055\Desktop\Data_Structur_Algo>  & 'c:\Users\18055\.vscode\extensions\ms-vscode.cpptools-1.20.5-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-z00j40kf.cpz' '--stdout=Microsoft-MIEngine-Out-5peoq0y4.ad5' '--stderr=Microsoft-MIEngine-Error-0nhs5awp.yjo' '--pid=Microsoft-MIEngine-Pid-wifvzrha.0yj' '--dbgExe=C:\Program Files (x86)\mingw64\bin\gdb.exe' '--interpreter=mi'
  1  12   9   0   0   0   0
  0   0   0   0   5   0   0
 -3   0   0   0   0  14   0
  0   0  13   0   0   0   0
  0  18   0   0   0   0   0
 15   0   0   0   0   0   0

  1   9   0   0   0   0   2   0
 -1   0   0   0   3   0   0   0
  0   0   0   8   0   0   0   0
  0   0   4   0   0   0   0   0
  0   7   0   0   0   0   0   1
  0   0   0   0   0   6   0   0
  5   0   0   0   0   0   0   0

-11   9   0  72  36   0   2   0
  0  35   0   0   0   0   0   5
 -3 -27   0   0   0  84  -6   0
  0   0   0 104   0   0   0   0
-18   0   0   0  54   0   0   0
 15 135   0   0   0   0  30   0

-11   0  -3   0 -18  15
  9  35 -27   0   0 135
  0   0   0   0   0   0
 72   0   0 104   0   0
 36   0   0   0  54   0
  0   0  84   0   0   0
  2   0  -6   0   0  30
  0   5   0   0   0   0
PS C:\Users\18055\Desktop\Data_Structur_Algo>
```

测试结论：功能实现完全。

实验名称： 实验 5——表达式求值

学生姓名： XXX

班　　级： 2023211119

班内序号： 07

学　　号： 2023210XXX

日　　期： 2024 年 5 月 12 日

1．实验要求

[正文格式要求]

字体：汉字宋体、英文 Times New Roman

字号：五号

颜色：黑色

行距：单倍行距

[内容要求]

表达式求值是程序设计语言编译中最基本的问题，它要求把一个表达式翻译成能够直接求值的序列。

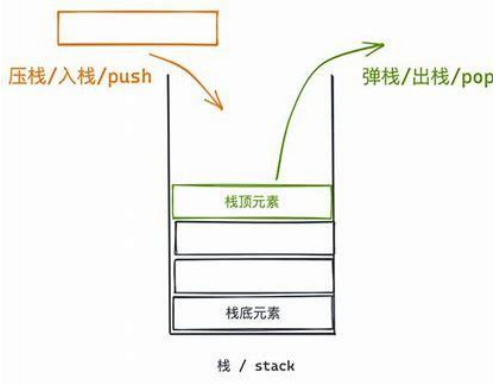
例如输入字符串“14+((13-2)*2-11*5)*2”，程序可以自动计算得到最终的结果。在这里，我们将问题简化，假定算数表达式的值均为非负整数常数，不包含变量、小数和字符常量。

试设计一个算术四则运算表达式求值的简单计算器。

2. 程序分析

2.1 存储结构

栈



2.2 关键算法分析

1.关键算法：

a.基本初始化判断

```
// 定义操作符优先级枚举类型
enum PRIO {
    NONE = 0,
    ADD_SUB,
    MUL_DEV,
    LEFT_BR
};
// 定义表达式解析状态枚举类型
enum PRE {
    START,
    NUM,
    OPER
};
// 函数：判断字符是否为操作符
function isOper(c) returnS BOOLEAN {
    return x[c] 不等于 NONE
}
// 定义字符到操作符优先级的映射数组
ARRAY x[255] 初始化为 0;
// 函数：设置操作符的优先级
function setOper() {
    for 每个操作符 c IN '+','-','*','/'：
        x[c] = 对应的操作符优先级
}
// 函数：判断字符是否为数字
function isDigital(c) returnS BOOLEAN {
    IF c 在 '0' 到 '9' 之间 THEN
        return true
    ELSE
        return false
    ENDIF
}
```

b. 执行栈顶运算

```
// 函数：执行栈顶的操作符所代表的运算
function run(栈 s, 栈 v) {
    o = s.pop()   // 弹出操作符栈顶的操作符
    x[0] = v.pop()   // 弹出值栈顶的值
    x[1] = v.pop()   // 弹出值栈次顶的值
    SWITCH o {
        case '+': v.push(x[1] + x[0])   // 执行加法
        case '-': v.push(x[1] - x[0])   // 执行减法
        case '*': v.push(x[1] * x[0])   // 执行乘法
        case '/': v.push(x[1] / x[0])   // 执行除法
```

```
            DEFAULT: 抛出 "oper error"  // 如果操作符无效，抛出错误
        }
    }
```

c.计算表达式的值

```
    // 函数：计算表达式的值
    function calc(字符串 k) returnS FLOAT {
        栈 s 初始化为空  // 操作符栈
        栈 v 初始化为空  // 值栈
        i = 0  // 字符串索引
        value = 0  // 当前数字的值
        status = START  // 当前状态

        WHILE k[i] 存在 DO
            c = k[i]  // 获取当前字符
            IF isDigital(c) THEN  // 如果当前字符是数字
                IF status 是 NUM THEN
                    value = value * 10 + c - '0'  // 累加数字
                ELSE
                    value = c - '0'  // 开始新的数字
                    status = NUM  // 设置状态为数字
                ENDIF
            ELSE  // 如果当前字符不是数字
                IF status 是 NUM THEN
                    v.push(value)  // 将数字压入值栈
                    status = OPER  // 设置状态为操作符
                ENDIF
                IF c 是 ')' THEN  // 如果当前字符是右括号
                    WHILE s 不为空 且 s.top() 不是 '(' DO
                        run(s, v)  // 执行栈顶的操作符
                    ENDWHILE
                    IF s 为空 THEN 抛出 "s empty error"  // 如果操作符栈为空，抛出错
    误
                    s.pop()  // 弹出左括号
                ELSE IF isOper(c) THEN  // 如果当前字符是操作符
                    IF s 不为空 且 s.top() 不是 '(' 且 x[s.top()] 大于等于 x[c] THEN
                        run(s, v)  // 执行优先级高的运算
                    ENDIF
                    s.push(c)  // 将操作符压入操作符栈
                ELSE
                    抛出 "error"  // 如果字符既不是数字也不是操作符，抛出错误
                ENDIF
            ENDIF
            i = i + 1  // 移动到下一个字符
        ENDWHILE
```

```
    IF status 是 NUM THEN
        v.push(value)   // 如果字符串以数字结束，将其压入值栈
    ENDIF

    WHILE v 的大小不等于 1 或 s 的大小不等于 0 DO
        run(s, v)   // 执行所有剩余的运算
    ENDWHILE

    return v.top()   // 返回值栈顶的元素，即为表达式的结果
}
```

2.代码详细分析

a.基本初始化

1. 引入库和命名空间

#include <stack>：引入栈的库。

#include <iostream>：引入输入输出流的库。

using namespace std;：使用标准命名空间。

2. 定义枚举类型

PRIO：定义操作符的优先级。

PRE：定义解析表达式时的状态。

3. 定义全局数组

char x[255]：用于存储字符和对应操作符优先级的映射。

4. 初始化操作符优先级映射

setOper()函数初始化 x 数组，将加、减、乘、除和左括号字符映射到它们对应的优先级。

5. 辅助函数

isDigital(char c)：判断字符 c 是否为数字。

isOper(char c)：判断字符 c 是否为操作符。

b.执行栈顶运算

如果两个栈都不为空，函数将从操作符栈 s 中弹出栈顶的操作符 o，然后从值栈 v 中连续弹出两个栈顶的值 x[0]和 x[1]。

根据弹出的操作符 o，函数将执行对应的运算。这里是一个 switch 语句，根据操作符的不同，执行加法、减法、乘法或除法。

无论执行哪种运算，结果都会被压回值栈 v 中。

c.表达式求值

calc(char *k)函数是程序的核心，它接收一个字符串 k 作为输入，表示一个数学表达式，并返回该表达式的计算结果。

计算过程：

初始化两个栈：操作符栈 s 和值栈 v。

初始化变量 i 为 0，用于遍历字符串；value 为 0，用于累积数字；status 为 START，表示初始状态。

遍历字符串 k 的每个字符 c：

如果 c 是数字，根据 status 的状态更新 value：

如果 status 是 NUM，将 c 累加到 value。

如果 status 不是 NUM，设置 status 为 NUM 并开始累加 value。

如果 c 不是数字：

如果 status 是 NUM，将当前 value 压入值栈 v，并将 status 更新为 OPER。

如果 c 是右括号)，执行操作符栈顶的运算直到遇到左括号(，然后弹出左括号。

如果 c 是操作符，检查操作符栈顶是否有优先级更高的操作符或左括号。如果有，先执行栈顶操作符的运算，再将 c 压入操作符栈。

如果 c 既不是数字也不是有效的操作符，抛出错误。

在遍历结束后，如果最后一个字符是数字，将其压入值栈。

执行所有剩余的运算，直到操作符栈为空且值栈只剩下一个元素。

返回值栈顶的元素，即为表达式的计算结果。3.时间和空间复杂度（n 是数组的大小）

3.时间和空间复杂度（n 是数组的大小）

a.基本初始化

时间复杂度 O(1)，空间复杂度 O(1)。

b.执行栈顶运算

时间复杂度 O(1)，空间复杂度 O(n)

c.表达式求值

时间复杂度 O(n)，空间复杂度 O(n)。

3. 程序运行结果

测试主函数流程：

```
int main()
{
    char s[100000];
    while (1)
    {
        try
        {
            cin >> s;
            setOper();
            cout << calc(s) << endl;
        }
        catch (const char *e)
        {
            cout << e << endl;
        }
    }
}
```

测试结果：

```
PS C:\Users\18055\Desktop\Data_Structur_Algo>  & 'c:\Users\18055\.vscode\extensions\ms-vscode.cpptools-1.20.5-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-qnq5pcud.kux' '--stdout=Microsoft-MIEngine-Out-rzopmjub.02r' '--stderr=Microsoft-MIEngine-Error-bbbtw2d5.ucf' '--pid=Microsoft-MIEngine-Pid-w4g1g2ll.kqn' '--dbgExe=C:\Program Files (x86)\mingw64\bin\gdb.exe' '--interpreter=mi'
14+((13-2)*2-11*5)*2
-52
12+(5+3)*2
28
```

测试结论：功能实现完全。

4. 总结

心得体会：

模板类的应用：通过使用模板类，Matrix 可以处理不同数据类型的矩阵运算，提高了代码的通用性和灵活性。
稀疏矩阵的表示：采用三元组 tuple 链表的形式来表示稀疏矩阵，有效节省了存储空间，并且对于稀疏矩阵的乘法运算，这种方法可以减少不必要的零元素乘法，提高了计算效率。
异常处理：在重载的乘法运算符中，通过抛出异常来处理矩阵尺寸不兼容的问题，这是一种良好的错误处理方式。
栈的使用：栈与递归有着天然的联系。递归函数的调用可以通过系统栈来实现，而迭代的递归算法则可以通过手动管理栈来完成。