数据结构

实验报告（三）

串和数组

学号：3020205015

姓名：石云天

班级：智能机器平台 2 班

日期：2022.12.4

目　录

一、实验内容描述 3
二、实验步骤 4
三、程序设计 4
（一）抽象数据类型 ADT 4
（二）算法简述 5
（三）程序代码 6
四、调试分析 6
（一）调试过程和主要错误 6
（二）时间复杂度 7
五、程序测试 7
六、实验总结 8
附录 1：程序源代码：myStr and myMatrix 9

一、实验内容描述

本次实验主题是串和数组，主要包含两个内容：实现字符串中的 KMP 算法、实现稀疏矩阵。下面是详细的实验内容：

（1）实现字符串 myStr，存储方式不限。完成以下功能：

- 字符串初始化，提示：字符串以\0 结尾
- 字符串销毁
- 字符串输出
- 其他必要的成员函数
- 实现串的替换，即要求在主串 S 中，从位置 start 开始查找是否存在子串 T，若主串 S 中存在子串 T，则用子串 V 替换子串 T，且函数返回 1；若主串 S 中不存在子串 T，则函数返回 0， start 取值从 1 开始
- kmp 辅助数组 next 的计算
- kmp 辅助数组 next 的输出
- 实现简单字符串匹配算法，目标串 S 和模式串 T，求 T 在 S 中的位置。匹配失败返回-1，匹配成功返回匹配位置（字符串的位置从 1 开始）
- 实现改进 KMP 算法的字符串匹配算法，目标串 S 和模式串 T，求 T 在 S 中的位置。匹配失败返回-1，匹配成功返回匹配位置（字符串的位置从 1 开始）

（2）实现稀疏矩阵 myMatrix，使用三元组存储稀疏矩阵元素，实现以下功能：

- 初始化稀疏矩阵,参数依次为行数、列数、三元组元素个数、三元组初始化数据，数组元素为 3 的倍数，每 3 个数一组，分别为（row，col，value）
- 初始化稀疏矩阵
- 销毁稀疏矩阵
- 其他必要的成员函数
- 实现快速转置算法,转置结果存在 T 中
- 打印矩阵，打印格式为：行数，列数，元素数

二、实验步骤

（1）根据上课所讲，回顾串和数组的基本概念，考虑字符串和数组的数据类型，认识到字符串其实就是字符数组。串和数组均使用顺序存储结构。随后总结所有关于串和数组的必要成员函数，构造抽象数据类型。
（2）仔细阅读实验要求，考虑 KMP 算法和稀疏矩阵快速转置算法的核心思想，并进一步考虑 KMP 算法的改进算法。列写伪代码，体会算法思路。
（3）利用 CodeBlocks 编译器，配置环境，基于 C++语言将算法用程序实现。
（4）编译运行程序，使用样例进行程序测试，观察所编程序是否实现要求的功能。
（5）考察算法的时间复杂度和空间复杂度，评价算法的优劣，进一步优化程序。
（6）撰写实验报告，进行实验总结与反思。

三、程序设计

（一）抽象数据类型 ADT

串的抽象数据类型：

ADT myStr{
数据对象：D={$< a_i, a_{i+1} > | a_i, a_{i+1} \in char, i = 1,2,3 \ldots$}
数据关系：每个节点都有唯一的前驱节点和唯一的后继节点，是一种线性关系。
基本操作：
`InitString (&T, chars)`
操作结果：字符串的初始化
`DestroyString(&S)`
操作结果：字符串的销毁
`StrCopy (&T,S)`
操作结果：字符串的复制
`StrLength(S)`
操作结果：求字符串的长度
`StrCompare (S, T)`
操作结果：字符串比较函数
`Concat (&T, S1,S2)`
操作结果：二叉树的初始化函数
`StrEmpty (S)`
操作结果：判断字符串是否为空串
`KMPmatch(&S,pos)`

操作结果：对字符串使用 KMP 算法
Getnext(S)
操作结果：获取当前字符串的 next 数组
Replace(&S,&T,pos)
操作结果：字符串的替换
}ADT myStr;

稀疏矩阵的抽象数据类型：

ADT myMatrix{
数据对象：D={$< a_i, a_{i+1} > | a_i, a_{i+1} \in int, i = 1,2,3 \ldots$}
数据关系：每个节点都有唯一的前驱节点和唯一的后继节点，是一种线性关系。
基本操作：
InitMatrix(M)
操作结果：初始化建立一个稀疏矩阵
DestroyMatrix(M)
操作结果：销毁一个稀疏矩阵
PrintMatirx(M)
操作结果：稀疏矩阵的打印
TransposeMatirx (M)
操作结果：实现稀疏矩阵的快速转置
}ADT myMatrix;

（二）算法简述

下面分别对 KMP 算法及其改进算法、稀疏矩阵的快速转置算法的算法思想进行介绍。

一、KMP 算法

（1）暴力算法与 KMP

对于暴力字符串匹配算法，分别设置两个指针采用两次循环，如果出现不匹配字符，将两个指针同时回退再进行循环。根据暴力匹配算法的思路，若假设现在文本串 S 匹配到 i 位置，模式串 P 匹配到 j 位置，则有：

如果当前字符匹配成功（即 S[i] == P[j]），则 i++，j++，继续匹配下一个字符；如果匹配失败（即 S[i]! = P[j]），令 i = i - (j - 1)，j = 0，相当于每次匹配失败时，i 回溯，j 被置为 0。

暴力算法时间复杂度高，因为其没有很好的利用已经匹配的信息，如何合理高效的利用已经匹配的信息便是 KMP 算法的出发点和核心思路。

（2）next 数组

KMP 算法的核心在于求出每个前缀的最长相同前缀后缀。在 KMP 算法中，这个集合被称作 next 数组，所以求出 next 数组就是整个 KMP 算法的核心步骤。根据前面的分析，next[j]的值应该是前 j-1 个字符构成的字符串的最长相同前缀后缀的长度，可通过如下公式进行计算：

$$\text{next}[j]=\begin{cases}-1, j=0\\ \max\{k \mid 0<k<j \text{且} Q_0\cdots Q_{k-1}=Q_{j-k}\cdots Q_{j-1}\}\end{cases}$$

（3）算法流程

假设目前文本串 S 匹配到 i 位置，模式串 P 匹配到 j 位。如果 j = -1 或当前字符匹配成功（即 S[i] == P[j]），都令 i++，j++，继续匹配下一个字符；如果 j != -1，且当前字符匹配失败（即 S[i] != P[j]），则令 i 不变，j = next[j]。此举意味着失配时，模式串 P 相对于文本串 S 向右移动了 j - next [j] 位。

如果发现如果某个字符匹配成功，模式串首字符的位置保持不动，仅仅是 i++、j++；如果匹配失配，i 不变（即 i 不回溯），模式串会跳过匹配过的 next [j]个字符。整个算法最坏的情况是，当模式串首字符位于 i - j 的位置时才匹配成功，算法结束。

二、稀疏矩阵转置算法

（1）一般的稀疏矩阵转置算法

由于稀疏矩阵存储使用三元组的结构，故可以通过下面三个步骤实现：

1. 将矩阵行列值相互转换

2. 将每个三元组中的 i 和 j 交换

3. 重排三元组的顺序实现矩阵转置

（2）快速转置算法

如果能预先确定矩阵 M 中每一列（即 T 中每一行）的第一个非零元在 b.data 中恰当位置。那么在对 a.data 中的三元组一次做转置时，便可直接放到 b.data 中恰当的位置上去。

据此原理，可设两个向量：x 和 y。x[col]表示矩阵 M 中第 col 列中的非零元素个数。y[col]指 M 中第 col 列的第一个非零元在 b.data 中的恰当位置。有下面两个公式：

$$y[1]=1$$

$$y[col]=y[col-1]+x[col-1]\ 2<=col<=a.nu$$

因此便可找到矩阵元素转置后应该放置的位置，从而能够快速实现转置。

（三）程序代码

为保证实验报告的清晰和可读性，将源代码以**附录形式**附于文末。

四、调试分析

（一）调试过程和主要错误

在程序编写完成后，使用 CodeBlocks 编译并运行。发现初代程序主要存在

以下问题：

在编写 KMP 算法时，写 next 数组的时候编译器报错 Segmentation Fault,我仔细检查了程序代码，发现还是因为没有深刻理解匹配算法所导致。按照正常的思路，一旦匹配失配，便有 j = next [j]，模式串向右移动的位数为：j - next[j]。换言之，当模式串的后缀 pj-k pj-k+1, ..., pj-1 跟文本串 si-k si-k+1, ..., si-1 匹配成功，但 pj 跟 si 匹配失败时，由于 next[j] = k，相当于在不包含 pj 的模式串中有最大长度为 k 的相同前缀后缀。而在编写程序时将失配右移的位数写成了 j-next[j]+1，忽略了初始值已经赋好了这一前提，所以程序才会出现数组越界的问题。

（二）时间复杂度

假设文本串长度为 N，模式串长度为 M，此时计算 next 数组的时间复杂度为 O（M），匹配过程的时间复杂度为 O（N），因此 KMP 算法的整体时间复杂度为 O（M+N）。对于普通的矩阵转置算法，需要双层嵌套循环，时间复杂度为 O（MN），应用快速转置算法后，多使用了两个辅助向量，为并列的循环，循环次数分别为 M 和 N，因此整体时间复杂度为 O（M+N）。

五、程序测试

在完成全部程序编写后，输入测试样例进行测试，所得结果均满足要求。myStr 基本操作及算法测试结果见图 1，myMatrix 基本操作及算法测试结果见图 2。

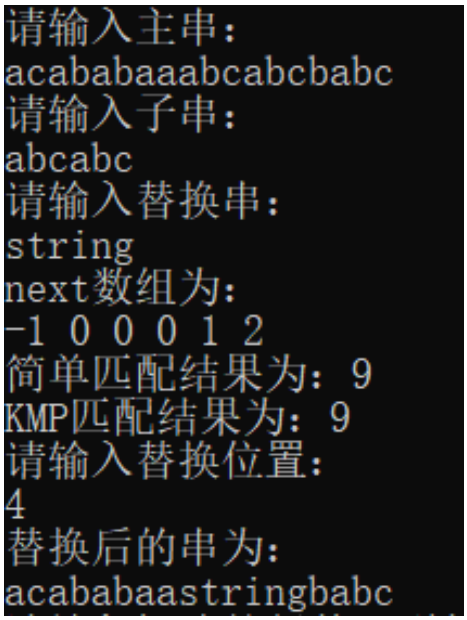

图 1 myStr 基本操作及算法测试结果

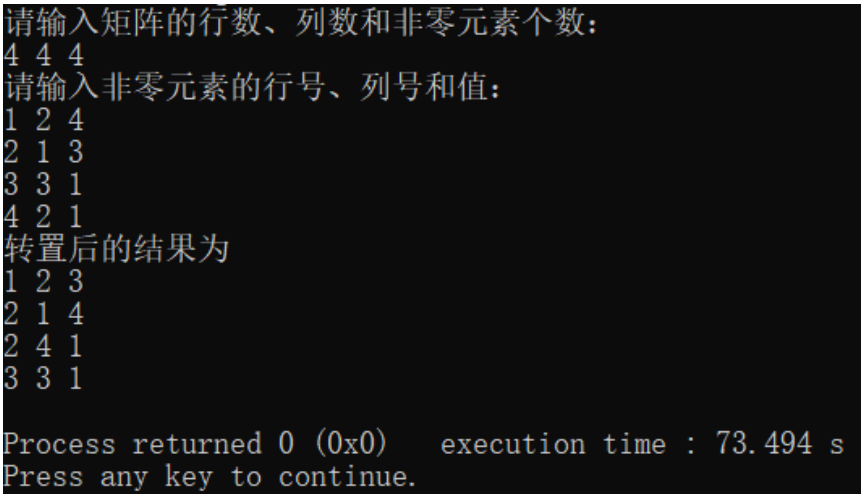

图 2 myMatrix 基本操作及算法测试结果

六、实验总结

通过这次试验，我发现我对串与数组这一部分的理解不够深入全面，需要不断巩固学习，加深理解。同时。在编程过程中需要完成某些特定目标时，我不能很快的想出其对应的操作，需要课下不断练习以熟能生巧，还可以多查阅一些资料以开阔自己的思路。在本次实验中，我编写并实现了字符串及稀疏矩阵的各项基本操作，同时利用以上数据结构实现了 KMP 算法，在此过程中不断调试，寻找问题，并不断简化代码，提升函数执行速度。除此之外，在本次实验过程中，编写、调试程序花费了很长时间：首先是 KMP 算法的编写，由于之前对于递归算法的思想理解不够深刻，起初使用起来较为生疏，错误频出，经过不断思考，我发现其与普通的暴力匹配算法不同，它利用已匹配的信息进行后一步的匹配，由于 next 数组的存在，主串的指针在循环中永远不会后移。其次是对于空指针的理解更加深刻，我最初以为 Segmentation Fault 类型报错是因为数组越界，经过不断查阅资料发现是因为调用了空指针，空指针不会指向任何实体，因此在程序编写过程中需要格外注意各指针变量指向的变化，在 delete 操作完成后，最好在后面加一行将指针置为 NULL 的代码，这可以有效避免调用空指针的错误。最后是关于结果部分不正确的情况，这一问题往往是比较棘手的，此时需要在边界条件上入手，寻找没有关注到的情况，让思考更加全面周到，有助于顺利解决问题。在本次实验后，我还需要精益求精，不断改进程序，优化函数性能，实现预期目标与功能所需。

附录 1：程序源代码：myStr and myMatrix

```
#include <iostream>
#include <cstring>
#define maxnum  100
using namespace std;

class myStr
{
public:
    char str[maxnum];
    //字符串初始化，以'\0'结尾
    int next[maxnum];
    myStr()
    {
        str[0] = '\0';
    }
    //字符串销毁
    ~myStr()
    {
        str[0] = '\0';
    }
    //next 数组的计算
    void Getnext()
    {
        int i = 0, j = -1;
        next[0] = -1;
        while(str[i] != '\0')
        {
            if(j == -1 || str[i] == str[j])
            {
                i++;
                j++;
                next[i] = j;
```

```
        }
        else
        {
            j = next[j];
        }
    }
}
//打印 next 数组
void Printnext()
{
    int i = 0;
    while(str[i] != '\0')
    {
        cout << next[i] << " ";
        i++;
    }
    cout << endl;
}
//简单字符串匹配
int Simplematch(myStr &s, int pos = 1)
{
    int i = pos-1, j = 0;
    while(str[i] != '\0' && s.str[j] != '\0')
    {
        if(str[i] == s.str[j])
        {
            i++;
            j++;
        }
        else
        {
            i = i - j + 1;
            j = 0;
        }
```

```
    }
    if(s.str[j] == '\0')
    {
        return i - j+1;
    }
    else
    {
        return -1;
    }
}
//改进 KMP 字符串匹配
int KMPmatch(myStr &s, int pos = 1)
{
    int i = pos-1, j = 0;
    while((str[i] != '\0' && s.str[j] != '\0')||j == -1)
    {
        if(j == -1 || str[i] == s.str[j])
        {
            i++;
            j++;
        }
        else
        {
            j = s.next[j];
        }
    }
    if(s.str[j] == '\0')
    {
        return i - j + 1;
    }
    else
    {
        return -1;
    }
}
```

```
//串的替换
int Replace(myStr &s, myStr &t, int pos)
{
    int i = pos, j = 0;
    i = KMPmatch(s, i);
    if(i != -1)
    {
        int lt = strlen(t.str);
        int ls = strlen(s.str);
        int k = i + ls - 1;//保留的位置
        char temp[maxnum];
        int f = 0;
        while(t.str[f] != '\0')
        {
            temp[f] = t.str[f];
            f++;
        }
        while(str[k] != '\0')
        {
            temp[f] = str[k];
            k++;
            f++;
        }
        temp[f] = '\0';
        k = i - 1;
        while(f >= 0)
        {
            str[k] = temp[j];
            k++;
            j++;
            f--;
        }
        return 1;
    }
```

```
        else
        {
            return 0;
        }

    }
    //打印字符串
    void StrPrint()
    {
        int i = 0;
        while(str[i] != '\0')
        {
            cout << str[i];
            i++;
        }
        cout << endl;
    }

};

//稀疏矩阵
class myMatrix
{
public:
    struct element
    {
        int row, col, value;
    };
    element data[maxnum];
    int row, col, num;
    myMatrix()
    {
        row = col = num = 0;
    }
```

```
    ~myMatrix()
    {
        row = col = num = 0;
    }
    //稀疏矩阵初始化
    void InitMatrix()
    {
        cout << "请输入矩阵的行数、列数和非零元素个数: " << endl;
        cin >> row >> col >> num;
        cout << "请输入非零元素的行号、列号和值: " << endl;
        for(int i = 0; i < num; i++)
        {
            cin >> data[i].row >> data[i].col >> data[i].value;
        }
    }
    //打印矩阵: 行, 列, 值
    void PrintMatrix()
    {
        for(int i = 0; i < num; i++)
        {
            cout << data[i].row << " " << data[i].col << " "
<< data[i].value << endl;
        }
    }
    //稀疏矩阵的快速转置
    void TransposeMatrix(myMatrix &m)
    {
        int i, j, k, colnum[maxnum], rowpos[maxnum];
        m.row = col;
        m.col = row;
        m.num = num;
        if(num > 0)
```

```
        {
            for(i = 0; i < col; i++)
            {
                colnum[i] = 0;
            }
            for(i = 0; i < num; i++)
            {
                colnum[data[i].col]++;
            }
            rowpos[0] = 0;
            for(i = 1; i < col; i++)
            {
                rowpos[i] = rowpos[i - 1] + colnum[i - 1];
            }
            for(i = 0; i < num; i++)
            {
                j = data[i].col;
                k = rowpos[j];
                m.data[k].row = data[i].col;
                m.data[k].col = data[i].row;
                m.data[k].value = data[i].value;
                rowpos[j]++;
            }
        }
    }

};

int main()
{
    myStr a,b,c;
    int num;
    cout << "请输入主串: " << endl;
    cin >> a.str;
```

```
    cout << "请输入子串: " << endl;
    cin >> b.str;
    cout << "请输入替换串: " << endl;
    cin >> c.str;
    b.Getnext();
    cout << "next 数组为: " << endl;
    b.Printnext();
    cout << "简单匹配结果为: " << a.Simplematch(b) << endl;
    cout << "KMP 匹配结果为: " << a.KMPmatch(b) << endl;
    cout << "请输入替换位置: " << endl;
    cin >> num;
    a.Replace(b,c,num);
    cout << "替换后的串为: " << endl;
    a.StrPrint();

    myMatrix m, n;
    m.InitMatrix();
    m.TransposeMatrix(n);
    cout << "转置后的结果为" << endl;
    n.PrintMatrix();
    return 0;
}
```