天津大学

数据结构实验报告

实验名称：串和数组

学院名称 智能与计算学部

专　　业 软件工程

学生姓名 陈昊昆

学　　号 3021001196

年　　级 2021 级

班　　级 软工 3 班

时　　间 2023 年 5 月 18 日

1. 实验内容

1. 实现字符串类 myStr，存储方式不限。实现以下功能：
   - 字符串初始化 myStr(const char*)，提示：字符串以\0 结尾
   - 字符串销毁~myStr()
   - 字符串输出 void print()
   - 其他必要的成员函数
   - 实现 myStr 的友元函数，串的替换 bool replaceStr(myStr& S, const int& start, const myStr& T, const myStr& V)，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，void kmp_next()
   - kmp 辅助数组 nextVal 的计算，void kmp_nextVal()
   - kmp 辅助数组 next 的输出 void printNext()
   - kmp 辅助数组 nextVal 的输出 void printNextVal()
   - 实现 myStr 的友元函数，简单字符串匹配算法 int simpleMatch(const myStr& S, const myStr& T)，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 myStr 的友元函数，改进 KMP 算法的字符串匹配算法 int kmpMatch(const myStr& S, const myStr& T)，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. 实现稀疏矩阵类 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，myMatrix (const int& rNum, const int& cNum, const int& nNum, const int*),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 myMatrix（）
   - 销毁稀疏矩阵，~ myMatrix()
   - 其他必要的成员函数
   - 实现快速转置算法 void FastTransposeSMatrix(myMatrix& T),转置结果存在 T 中
   - 打印矩阵 void printMatrix()，打印格式为：行数，列数，元素数
     行，列，元素值

……

2. 程序实现

```
#include "string_array.h"
#include <iostream>
using namespace std;

//字符串初始化 myStr(const char*)，提示：字符串以\0 结尾
myStr::myStr(const char* s){
        int len = 0;
        while (s[len] != '\0') len++;
        data = new char[len + 1];
        length = len;
        for (int i = 0; i <= len; i++) {
            data[i] = s[i];
        }

}

// 字符串销毁~myStr()
myStr::~myStr(){
    delete data;
    length = 0;
}

// 寻找匹配子串的位置
int myStr::findStr(const myStr& T, const int& pos) {
    if (pos <= 0 || pos > length) {
        return -1;
    }

    for (int i = pos-1 ; i < length - T.length + 1; i++) {
        bool found = true;
        for (int j = 0; j < T.length; j++) {
            if (data[i + j] != T.data[j]) {
                found = false;
                break;
            }
        }
        if (found) {
            return i + 1;
        }
    }
```

```
    return -1;
}


bool replaceStr(myStr& S, const int& start, const myStr& T, const
myStr& V) {
    if (start <= 0 || start > S.length || T.length == 0) {
        return false;
    }

    int pos = S.findStr(T, start);
    if (pos == -1) {
        return false;
    }
    pos -= 1;

    int newLength = S.length - T.length + V.length;
    char* newData = new char[newLength + 1];

    for (int i = 0; i < pos; i++) {
        newData[i] = S.data[i];
    }

    for (int i = 0; i < V.length; i++) {
        newData[pos + i] = V.data[i];
    }

    for (int i = pos + T.length; i < S.length; i++) {
        newData[V.length + i - T.length] = S.data[i];
    }

    newData[newLength] = '\0';
    delete[] S.data;
    S.data = newData;
    S.length = newLength;

    return true;
}



void myStr::kmp_next(){
    for(int i = length; i > 0; i--){
        data[i] = data[i-1];
    }
```

```
    next = new int [length+1];
    next[1] = 0;
    next[2] = 1;
    int j = 2;
    int k = 1;
    while(j < length){
        if(data[j] == data[k]){
            next[j+1] = k + 1;
            j++;
            k++;
        }
        else k = next[k];
        if(k == 0){
            next[j+1] = 1;
            j++;
            k = 1;
        }
    }

    for(int i = 0; i < length; i++){
        data[i] = data[i+1];
        next[i] = next[i+1];
    }


}


void myStr::kmp_nextVal(){
    for(int i = length; i > 0; i--){
        data[i] = data[i-1];
    }

    nextVal = new int [length+1];
    nextVal[1] = 0;

    int j = 1;
    int k = 0;

    while(j < length){
        if(k == 0 || data[j] == data[k]){
            j++;
            k++;
            if(data[k] != data[j]) nextVal[j] = k;
            else nextVal[j] = nextVal[k];
```

```
        }
        else k = nextVal[k];
    }

    for(int i = 0; i < length; i++){
        data[i] = data[i+1];
        nextVal[i] = nextVal[i+1];
    }

}

int simpleMatch(const myStr& S, const myStr& T){
    for (int i = 0 ; i < S.length - T.length + 1; i++) {
        bool found = true;
        for (int j = 0; j < T.length; j++) {
            if (S.data[i + j] != T.data[j]) {
                found = false;
                break;
            }
        }
        if (found) {
            return i + 1;
        }
    }

    return -1;
}

int kmpMatch(const myStr& S, const myStr& T){
    for(int i = S.length; i > 0; i--){
        S.data[i] = S.data[i-1];
    }
    for(int i = T.length; i > 0; i--){
        T.data[i] = T.data[i-1];
    }
    for(int i = T.length; i > 0; i--){
        T.nextVal[i] = T.nextVal[i-1];
    }

    int i = 1;
    int j = 1;
    while(i <= S.length && j <=T.length){
        if(j == 0 || S.data[i] == T.data[j]){
            i++;
            j++;
```

```
        }
        else j = T.nextVal[j];
    }

    for(int i = 0; i < S.length; i++){
        S.data[i] = S.data[i+1];
    }

    for(int i = 0; i < T.length; i++){
        T.data[i] = T.data[i+1];
        T.nextVal[i] = T.nextVal[i+1];
    }
    if(j > T.length) return i - T.length;
    else return -1;
}




void myStr::print(){
    for(int i = 0; i < length; i++)
        cout << data[i];
    cout << endl;
}

void myStr::printNext(){
    for(int i = 0; i < length; i++)
        cout << next[i];
    cout << endl;
}

void myStr::printNextVal(){
    for(int i = 0; i < length; i++)
        cout << nextVal[i];
    cout << endl;
}



myMatrix::myMatrix(const int& rNum, const int& cNum, const int& nNum,
const int*data){
        rowNum = rNum;
        colNum = cNum;
        nodeNum = nNum;
        nodeList = new matrixNode[nNum];
        for (int i = 0; i < nNum; i++) {
```

```
            nodeList[i].row = data[i*3];
            nodeList[i].col = data[i*3+1];
            nodeList[i].value = data[i*3+2];
        }
}

myMatrix::~myMatrix(){
    rowNum = 0;
    colNum = 0;
    nodeNum = 0;
    delete nodeList;
}

myMatrix::myMatrix(){
}


void myMatrix::FastTransposeSMatrix(myMatrix& T){
    T.rowNum=colNum;
    T.colNum=rowNum;
    T.nodeNum=nodeNum;

    T.nodeList = new matrixNode[nodeNum];

    int* num = new int [colNum+1];
    int* cpot = new int [colNum+1];


    if(T.nodeNum){
        for(int col = 1; col <= colNum; col++)
            num[col]=0;
        for(int i = 0; i < nodeNum; i++) {
            num[nodeList[i].col]++;
        }

        cpot[1]=1;
        for(int col = 2; col <= colNum; col++)
            cpot[col] = cpot[col-1] + num[col-1];


        for(int p = 0 ; p < nodeNum; p++){
            int col = nodeList[p].col;
            int q = cpot[col] - 1;
            T.nodeList[q].row=nodeList[p].col;
            T.nodeList[q].col=nodeList[p].row;
```

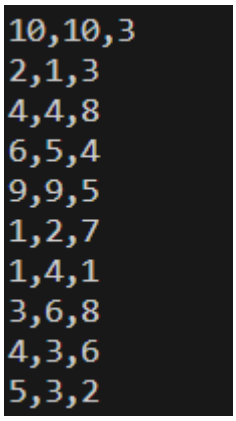
```
            T.nodeList[q].value=nodeList[p].value;
            cpot[col]++;
        }
    }
}

void myMatrix::printMatrix() {
cout << rowNum << "," << colNum << "," << nodeNum << endl;

for(int i = 1; i <= rowNum; i++){
    for(int j = 1; j <= colNum; j++){
        for(int u = 0; u < nodeNum; u++){
            if(nodeList[u].row == i && nodeList[u].col == j)
            cout << nodeList[u].row << "," << nodeList[u].col << "," <<
nodeList[u].value << endl;
        }

    }
}
}
```

3. 实验结果

字符串

```
1 0
1
11
-1
-1
011
011
01112323112311111121
01101313101311110 20
8
```

稀疏矩阵



4. 实验中遇到的问题及解决方法

实现了 KMP 算法,用于字符串匹配。KMP 算法是一种进步的字符串匹配算法,相比朴素匹配算法具有更高的效率。理解 KMP 算法的原理和实现过程,对学习字符串匹配很重要。

实现了稀疏矩阵的快速转置,理解了稀疏矩阵存储结构和相关操作。稀疏矩阵通过存储非零元素的位置和值来实现矩阵的压缩存储,本程序采用三元组{行,列,值}来表示一个非零元素。

实现过程中也注意到一些细节,如字符串中‘\0’的使用、递归结束条件等。这些地方在实现字符串和矩阵时需要注意。

实现一个简单的字符串匹配和矩阵转置的应用,可以更好地理解字符串和矩阵在实际问题中的应用。