

数据结构课程设计

南京航空航天大学
计算机科学与技术学院

班 级 : 1618403

学 号 : 161840225

姓 名 : 宋 鹏 霄

指导老师: 孙 涵

指导助教: 林 磊

目录

(总代码量 3811 行)

1	系统进程设计	4
1.1	数据结构	4
1.2	设计思想	4
1.3	核心代码	4
1.4	测试数据及结果	5
1.5	算法结果与分析	6
2	算术表达式求值	6
2.1	数据结构	6
2.2	设计思想	6
2.3	核心代码	6
2.4	测试数据及结果	9
3	公共钥匙盒	9
3.1	数据结构	9
3.2	设计思想	9
3.3	核心代码	9
3.4	测试数据及结果	10
3.5	算法结果与分析	11
4	家谱管理系统	11
4.1	数据结构	11
4.2	设计思想	11
4.3	核心代码	11
4.4	测试数据及结果	13
5	哈夫曼编码	13
5.1	数据结构	13
5.2	设计思想	13
5.3	源代码	14
5.4	测试数据及结果	20
5.5	改进策略	20
6	最小生成树	21
6.1	数据结构	21
6.2	设计思想	21
6.3	核心代码	21
6.4	测试数据及结果	26
6.5	改进策略	27
7	公交线路规划	27

7.1	数据结构	27
7.2	设计思想	27
7.3	源代码	27
7.4	测试数据集结果	38
7.5	算法结果与分析:	38
8	排序算法比较	39
8.1	数据结构	39
8.2	设计思想	39
8.3	各排序算法分析	39
8.4	源代码:	40
8.5	测试数据及结果	51
8.6	算法结果分析	52
9	用扑克牌计算 24 点.....	52
9.1	数据结构	52
9.2	设计思想	52
9.3	源代码	52
9.4	测试数据及结果	57
10	跳一跳.....	57
10.1	设计思想	57
10.2	源代码	57
10.3	测试数据及结果	58
11	URL 映射	58
11.1	设计思想	58
11.2	源代码	58
11.3	测试数据及结果	60
12	迷宫问题.....	61
12.1	数据结构	61
12.2	设计思想	61
12.3	源代码	61
12.4	测试数据及结果	65
13	连连看.....	65
13.1	算法思想	65
13.2	源代码	65
13.3	测试数据及结果	76
14	树的应用.....	77
14.1	数据结构	77
14.2	算法思想	77
14.3	源代码	77
14.4	测试数据及结果	80

1 系统进程设计

1.1 数据结构

1.1.1 链表

1.2 设计思想

1.2.1 一个链表按照内存使用自多到少存储当前活动进程，另外一个链表

按照结束时间离当前时间的关系存储已结束进程；

1.2.2 每秒在窗口内更新一次当前系统进程情况，输出内容包括：进程名，

持续时间，内存使用情况。

1.2.3 每秒在窗口内更新一次已结束进程情况，输出内容包括：进程名，

持续时间，结束时间。

1.2.4 其中进程信息由

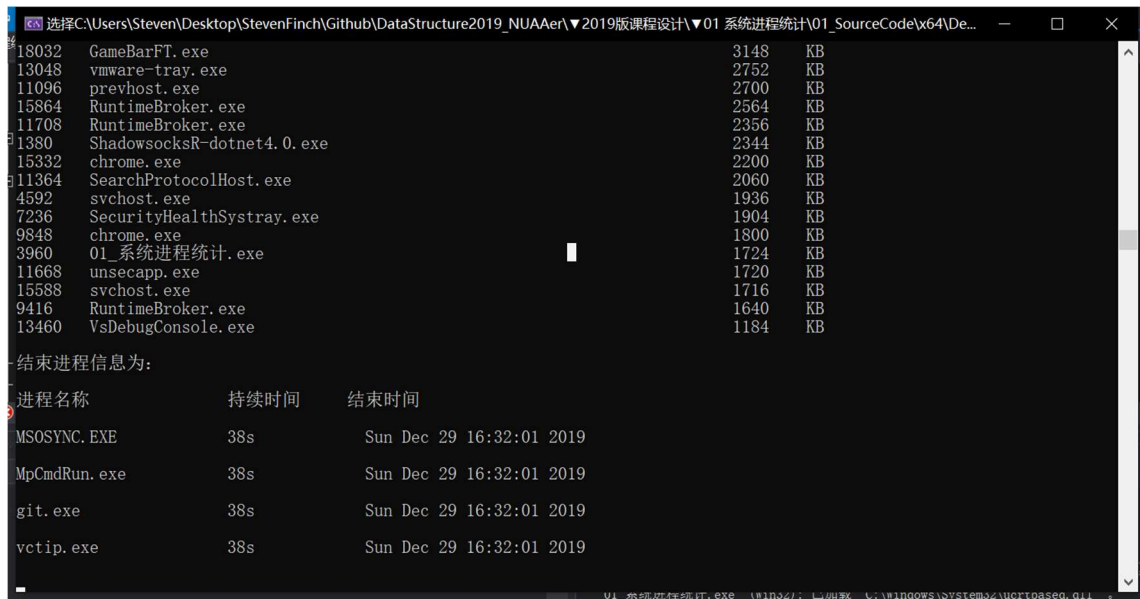
1.3 核心代码

```
ListInfo* DList = new ListInfo;

    InitList(DList);
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(pe32);
    HANDLE hProcessSnap = ::CreateToolhelp32Snapshot(TH32CS_SNAPPRO
CESS, 0);

    BOOL bMore = ::Process32First(hProcessSnap, &pe32);
    while (bMore)
    {
        Process p(pe32);
        InsertElemAfterCurNode(DList, &p);
        bMore = ::Process32Next(hProcessSnap, &pe32);
    }
```

1.4 测试数据及结果



The screenshot shows a Windows command prompt window with the following content:

```
选择C:\Users\Steven\Desktop\StevenFinch\Github\DataStructure2019_NUAAer\▼2019版课程设计\▼01 系统进程统计\01_SourceCode\vx64\De...  — □ ×
```

18032	GameBarFT.exe	3148	KB
13048	vmware-tray.exe	2752	KB
11096	prevhost.exe	2700	KB
15864	RuntimeBroker.exe	2564	KB
11708	RuntimeBroker.exe	2356	KB
1380	ShadowsocksR-dotnet4.0.exe	2344	KB
15332	chrome.exe	2200	KB
11364	SearchProtocolHost.exe	2060	KB
4592	svchost.exe	1936	KB
7236	SecurityHealthSystray.exe	1904	KB
9848	chrome.exe	1800	KB
3960	01_系统进程统计.exe	1724	KB
11668	unsecapp.exe	1720	KB
15588	svchost.exe	1716	KB
9416	RuntimeBroker.exe	1640	KB
13460	VsDebugConsole.exe	1184	KB

结束进程信息为:

进程名称	持续时间	结束时间
MSOSYNC.EXE	38s	Sun Dec 29 16:32:01 2019
MpCmdRun.exe	38s	Sun Dec 29 16:32:01 2019
git.exe	38s	Sun Dec 29 16:32:01 2019
vctip.exe	38s	Sun Dec 29 16:32:01 2019

任务栏显示: 01_系统进程统计.exe (Win32) | 已加载 C:\Windows\System32\UCRT0ASeg.dll

1.5 算法结果与分析

1.5.1 排序利用选择排序，时间复杂度为 $O(n^2)$ ；

1.5.2 对 API 的运行机制不熟练，导致代码逻辑不清晰，结构设置也有待优化。

2 算术表达式求值

2.1 数据结构

2.1.1 栈

2.1.2 顺序表

2.2 设计思想

2.2.1 首先将算术表达式去除所有空格，然后进行合法字符检验，最后进行分割处理，将得到的运算符和运算数存入 Vector 中。

2.2.2 第二步，建立运算数栈和运算符栈，将 Vector 中的元素依次入栈；当当前运算符优先级不大于栈顶运算符优先级时，将两个运算数和一个运算符弹栈并进行运算，将运算结果压栈；循环往复，直到 Vector 数组为空，这时运算数栈的元素即表达式值。

2.2.3 在操作过程中，进行合法检验。

2.3 核心代码

```
//栈操作
double CalExpress(string str)
{
    vector<string> exp;
    if (SeperateStr(exp, str) == 0)
        return FLT_MAX;

    stack<double> opdSt;//运算数栈
    stack<string> optSt;//运算符栈
```

```

string topOpt;

for (int i = 0; i < exp.size(); ++i)
{
    string c = exp[i];

    if (IsOpt(c[0]))
    {
        //空栈则压栈
        if (optSt.size() == 0)
        {
            optSt.push(c);
            PrintStack(opdSt, optSt);
        }

        else
        {
            string topOpt = optSt.top();

            if (CompareOpt(topOpt[0], c[0]) == '>' || CompareOpt(topOpt[0], c[0]) == '=')//栈顶优先级不小于当前优先级
            {
                while (CompareOpt(topOpt[0], c[0]) == '>' || CompareOpt(topOpt[0], c[0]) == '=')
                {
                    optSt.pop();
                    PrintStack(opdSt, optSt);

                    Calculate(opdSt, topOpt[0]);
                    PrintStack(opdSt, optSt);

                    if (optSt.size() > 0)
                    {
                        topOpt = optSt.top();
                    }
                    else
                    {
                        break;
                    }
                }
                //当前运算符入栈
                optSt.push(c);
                PrintStack(opdSt, optSt);
            }
        }
    }
}

```

```

        else//栈顶优先级大于当前优先级
        {
            optSt.push(c);
            PrintStack(opdSt, optSt);
        }
    }//end else
} //end if
else if (c == "(")
{
    optSt.push(c);
    PrintStack(opdSt, optSt);
}
else if (c == ")")
{
    topOpt = optSt.top();

    while (topOpt != "(")
    {
        Calculate(opdSt, topOpt[0]);
        PrintStack(opdSt, optSt);

        optSt.pop();
        PrintStack(opdSt, optSt);

        topOpt = optSt.top();
    }
    optSt.pop();//左括号出栈
    PrintStack(opdSt, optSt);
}
else//操作数
{
    opdSt.push(atof(c.c_str()));
    PrintStack(opdSt, optSt);
}
}

//处理剩余
while (!optSt.empty())
{
    topOpt = optSt.top();

    Calculate(opdSt, topOpt[0]);
    PrintStack(opdSt, optSt);
}

```



```

        optSt.pop();
        PrintStack(opdSt, optSt);
    }

    return opdSt.top();
}

```

2.4 测试数据及结果

该表达式结果为: $1.1 + (2.2 * 3.3 + 4.4 / (1.2 - 0.111)) = 12.4004$

该表达式结果为: $1 + 2 + 3.2 = 6.2$

该表达式结果为: $3/0 = \text{inf}$

3 公共钥匙盒

3.1 数据结构

3.1.1 顺序结构顺序表

3.1.2 队列

3.2 设计思想

3.2.1 开辟钥匙盒数组，每个钥匙盒设置 status 位记录该位置是否为空；

3.2.2 开辟借还事件数组，根据题意按照时间进行排序，然后依次入队；

3.2.3 根据时间进程，将取走钥匙和归还钥匙分别视为事件，放入队列中，

然后通过每个事件的先后发生对钥匙盒的状态进行变更；

3.2.4 在控制端通过显示钥匙盒的即时状态，以及事件队列的状态。

3.3 核心代码

```

//模拟事件发生
while (q.length)
{
    PrintStatus(q, ks, n);
}

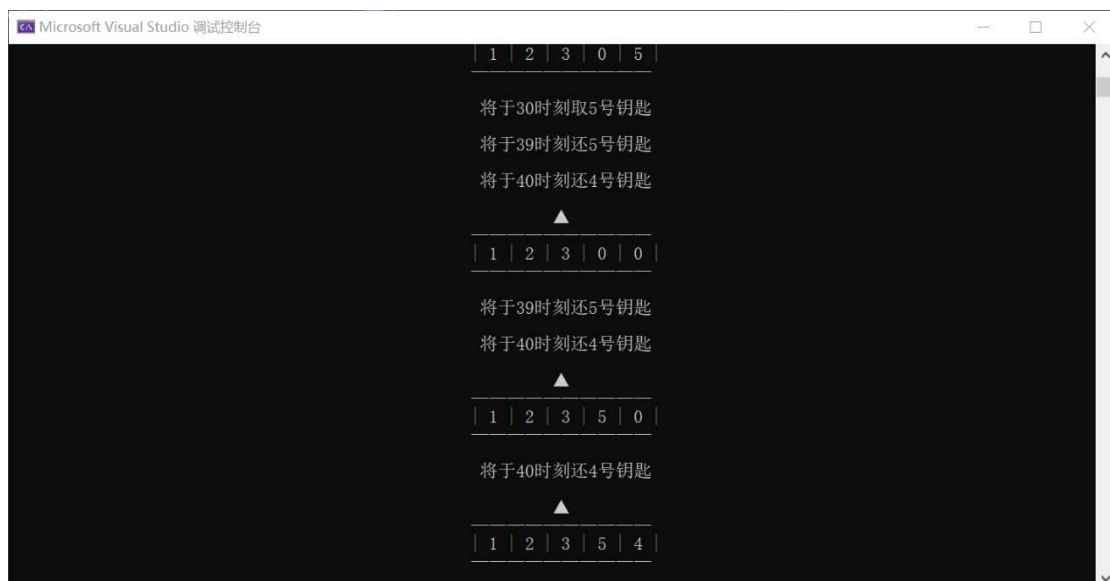
```

```

temp = q.Dequeue();
if (temp.flag == 0)
{
    for (j = 1; j < n + 1; j++)
    {
        if (ks[j].num == temp.num)
        {
            ks[j].num = 0;
            ks[j].status = 0;
        }
    }
}
else
{
    for (j = 1; j < n + 1; j++)
    {
        if (ks[j].status == 0)
        {
            ks[j].num = temp.num;
            ks[j].status = 1;
            break;
        }
    }
}
}
}

```

3.4 测试数据及结果



3.5 算法结果与分析

3.5.1 时间复杂度：主要操作在模拟事件队列，两层循环，故时间复杂度为 $O(n^2)$ 。

3.5.2 改进策略

1) 可在事件发生时间排序问题上进行优化。

4 家谱管理系统

4.1 数据结构

4.1.1 二叉树

4.2 设计思想

4.2.1 将森林转换为左孩子右兄弟树，进行模拟；

4.3 核心代码

```
//输出家谱成员信息
void OutputFamily(Tree T)

//先序序列创建 CSTree
void CreateTree(Tree& T, istream& fin, int generation)

//销毁该树
void DestroyTree(Tree& T)

//通过姓名查找成员信息
bool SearchByName(Tree T, string name, tn& m)
bool Search_FaS(Tree T, string name)

//通过年份查找成员信息
bool SearchByYear(Tree T, int year)

//打印某代人全部成员信息
bool PrintGenMember(Tree T, int i)

//添加孩子
bool AddChild(Tree& T, string name)
```

```
//删除某成员及其后代
```

```
bool DeleteMember(Tree& T, string name)
```

```
//查询某代所有成员
```

```
void Generation(Tree T, string name, int& gen)
```

4.4 测试数据及结果

4.4.1 略

5 哈夫曼编码

5.1 数据结构

5.1.1 二叉树

5.2 设计思想

5.2.1 采用面向对象设计思路，数据即用户提供待压缩文件；各个函数完成哈夫曼编码过程。

5.2.2 读取英文文档，统计各字符出现频度，构造数组存储各字符即频度；

5.2.3 构造哈夫曼树：

- 1) 根据统计的频度 $\{w_1, w_2, w_3 \dots w_n\}$ ，构造 n 棵只有根节点的二叉树，令其权值为 w_j ；
- 2) 在森林中选取两棵根节点权值最小的树作为左右子树，构造一颗新的二叉树，置新二叉树根节点权值为其左右子树根节点权值之和。注意，左子树的权值应小于右子树的权值；
- 3) 从森林中删除这两棵树，同时将新得到的二叉树加入森林中；
- 4) 重复上述两步，直到只含一棵树为止，即哈夫曼树；
- 5) 遍历哈夫曼树叶结点，并将编码存取。

5.2.4 编码：再次遍历英文文档将所有字符以字符串 01 文本形式存储；
并利用位运算将文本格式 01 字符串转化为二进制文件形式，编码完毕；

5.2.5 解码：利用逆位运算将二进制文件解码。

5.3 源代码

```
#include "HuffmanCoding.h"
using namespace std;

void HuffmanCoding::Coding()
{
    this->WriteCodeTxt();
    this->WriteSourceCodeTxt();
    int left = this->WriteSourceCodeDat();
    this->DecodeTargetCodeTxt(left);
    this->DecodeTargetTxt();
}

HuffmanCoding::HuffmanCoding(string sourcefile)
{
    n = 0;
    hufcode_ = (HufCode*)malloc((128 + 1) * sizeof(HufCode));
    sourcefile_ = sourcefile;
    ReadSource();
    CreateHufTree();
    Encoding();
}

void Select(HufNode*& hufree_, int m, int& s1, int& s2)
{
    int i, j;
    int* a = (int*)malloc((m + 1) * sizeof(int));

    for (i = 1, j = 1; i < m + 1; i++)
    {
        //把 parent 为 0 的结点复制到 a[] 中
        if (hufree_[i].parent == 0)
        {
            a[j++] = hufree_[i].fre;
        }
    }
    for (i = 1; i < j - 1; i++)
    {
        //对 a[] 冒泡排序
        for (int k = i + 1; k < j; k++)
        {
            if (a[i] > a[k])
                swap(a[i], a[k]);
        }
    }
}
```

```

//找出 s1, s2
for (i = 1; i <= m; i++)
{
    if (huftree_[i].fre == a[1] && huftree_[i].parent == 0)
    {
        s1 = i;
        break;
    }
}
for (i = 1; i <= m; i++)
{
    if (huftree_[i].fre == a[2] && huftree_[i].parent == 0 && i !=
s1)
    {
        s2 = i;
        break;
    }
}
free(a);
}

//统计
void HuffmanCoding::ReadSource()
{
    ifstream fin(sourcefile_);
    unsigned char now;//当前字符
    while (fin)
    {
        now = fin.get();
        int i;
        //已经出现过
        for (i = 1; i < n + 1; i++)
        {
            if (hufcode_[i].val == now)
            {
                hufcode_[i].fre++;
                break;
            }
        }
        //新字符
        if (i == n + 1)
        {
            hufcode_[i].val = now;
            hufcode_[i].fre = 1;
        }
    }
}

```

```

        n++;
    }
}
fin.close();
}

//建树
void HuffmanCoding::CreateHufTree()
{
    hufmtree_ = (HufNode*)malloc((2 * n) * sizeof(HufNode)); //一共有 2n-1
    个结点, 其中 0 号位置闲置
    if (NULL == hufmtree_)
        exit(OVERFLOW);
    int i;
    //初始化叶节点
    for (i = 1; i < n + 1; i++)
    {
        hufmtree_[i].val = hufcode_[i].val;
        hufmtree_[i].fre = hufcode_[i].fre;
        hufmtree_[i].parent = 0;
        hufmtree_[i].lchild = 0;
        hufmtree_[i].rchild = 0;
    }
    //建树
    for (i; i < 2 * n; i++)
    {
        int s1, s2;
        Select(hufmtree_, i - 1, s1, s2);
        hufmtree_[s1].parent = hufmtree_[s2].parent = i;
        hufmtree_[i].val = '\0';
        hufmtree_[i].fre = hufmtree_[s1].fre + hufmtree_[s2].fre;
        hufmtree_[i].parent = 0;
        hufmtree_[i].lchild = s1;
        hufmtree_[i].rchild = s2;
    }
}

//字符编码
void HuffmanCoding::Encoding()
{
    int i;
    for (i = 1; i < n + 1; i++)
    { //字符编码
        int t = 0;

```



```

        for (int c = i, p = hufree_[i].parent; p != 0; c = p, p = hufree_[p].parent)
        {
            if (hufree_[p].lchild == c)
                hufree_[i].code[t++] = '0';
            else
                hufree_[i].code[t++] = '1';
        }
        reverse(hufree_[i].code, hufree_[i].code + t);
        hufree_[i].code[t++] = '\0';
    }
}

//字符编码文本
void HuffmanCoding::WriteCodeTxt()
{
    ofstream fout("code.txt");
    int i;
    for (i = 1; i < n + 1; i++)
    {
        fout << left;
        fout << setw(5) << hufree_[i].val << setw(20) << hufree_[i].code << endl;
    }
    fout.close();
}

//文档编码文本
void HuffmanCoding::WriteSourceCodeTxt()
{
    ifstream fin("Source.txt");
    ofstream fout("SourceCode.txt");
    unsigned char t;
    while (fin)
    {
        t = fin.get();
        for (int i = 1; i < n + 1; i++)
        {
            if (hufree_[i].val == t)
            {
                fout << hufree_[i].code;
            }
        }
    }
}

```

```

        fout.close();
        fin.close();
    }

    //文档编码二进制文本
    int HuffmanCoding::WriteSourceCodeDat()
    {
        ifstream fin("SourceCode.txt");
        ofstream fout("SourceCode.dat", ios::binary);
        unsigned char c = '\0';
        unsigned char now;
        int m = 0;
        int left = 0;
        fin >> now;
        while (fin)
        {
            for (int i = 1; i < 9; i++)
            {
                if (now == '1')
                    c = (c << 1) | 0x01;
                else
                    c = (c << 1);
                fin >> now;
                if (fin.eof())
                {
                    left = 8 - i;
                    for (int j = 1; j < left + 1; j++)
                    {
                        c = (c << 1);
                    }
                    break;
                }
            }
            //end for
            fout.write((char*)&c, sizeof(c));
        }
        //end while
        fout.close();
        fin.close();
        return left;
    }

    //解码后编码文本
    void HuffmanCoding::DecodeTargetCodeTxt(int left)
    {
        ifstream fin("SourceCode.dat", ios::binary);
    }

```

```

    ofstream fout("TargetCode.txt");
    unsigned char c;
    fin.read((char*)&c, sizeof(c));
    while (fin)
    {
        for (int i = 1; i < 9; i++)
        {
            if ((c & 0x80) == 0x80)
                fout << 1;
            else
                fout << 0;
            c <<= 1;
        }
        fin.read((char*)&c, sizeof(c));
    }
    fout.seekp(-left * (int)sizeof(char), fout.end);
    fout << "#####";
    fout.close();
    fin.close();
}

//解码后文本
void HuffmanCoding::DecodeTargetTxt()
{
    ifstream fin("TargetCode.txt");
    ofstream fout("Target.txt");
    unsigned char t = '\0';
    //从根节点走到叶节点然后输出当前字符
    while (fin)
    {
        int i = 2 * n - 1;
        while (0 != huftree_[i].lchild && 0 != huftree_[i].rchild)
        {
            fin >> t;
            if ('0' == t)
                i = huftree_[i].lchild;
            else if ('1' == t)
                i = huftree_[i].rchild;
            else if '#'
                break;
        }
        if ('#' == t)
            break;
        fout << huftree_[i].val;
    }
}

```

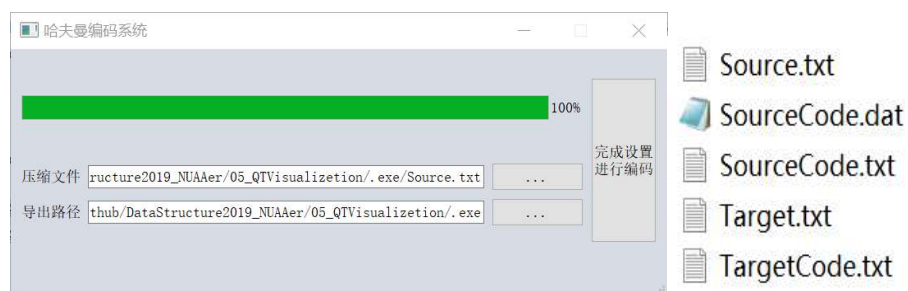
```

    }
    fout.close();
    fin.close();
}

HuffmanCoding::~HuffmanCoding()
{
    free(huftree_);
    free(hufcode_);
}

```

5.4 测试数据及结果



5.5 改进策略

- 1) QT 界面只完成了基本操作，尚不能达到完美的用户体验；
- 2) QT 界面没有进行编码改进，暂不支持中文路径。

6 最小生成树

6.1 数据结构

6.1.1 带权无向图

6.1.2 树

6.2 设计思想

6.2.1 Prim 算法：从点集开始出发，每次从已经生成的树中找到离该树权值最小的一条边，若没有构成回路则并入树中。

6.2.2 Kruskal 算法：从边集出发，每次都选取最短的边，若不构成回路则进入树中，利用并查集的思想。相对来说较简单。

6.3 核心代码

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <algorithm>
using namespace std;
const int MAX_N = 10; // max vertex in array
const int MAX_E = 11;

//Prim
struct Graph_P
{
    struct vertex
    {
        int num;
        string name;
    } v[MAX_N];
    float w[MAX_N][MAX_N];
    int n; // vertex
};
struct edge
{
    int v;
    float w;
};
```

```

void CreateGraph_P(Graph_P& G)
{
    fstream fin("Graph.txt");
    if (!fin)
    {
        cout << "file failed" << endl;
        exit(1);
    }
    int i, j;
    float k;
    fin >> G.n; //vertex number
    if (G.n > MAX_N)
        exit(OVERFLOW);
    for (i = 0; i < G.n; i++)
    {
        fin >> G.v[i].name;
        G.v[i].num = i;
    }
    for (i = 0; i < MAX_N; i++)
    {
        for (j = 0; j < MAX_N; j++)
        {
            G.w[i][j] = FLT_MAX;
        }
    }
    while (!fin.eof())
    {
        fin >> i >> j >> k;
        G.w[i][j] = G.w[j][i] = k;
    }
    fin.close();
}

void Prim(const Graph_P& G, int v0, float& c)
{ //v0: beginning vertex, c: lowest cost
    edge lowcost[MAX_N];
    int vset[MAX_N]; //vertex set
    int i, j;
    c = 0;
    for (i = 0; i < G.n; i++)
    {
        lowcost[i].v = v0;
        lowcost[i].w = G.w[v0][i];
        vset[i] = 0;
    }
}

```

```

vset[v0] = 1;//visited

cout << "Prim 算法" << endl;
cout << " 1.该最小生成树各边为: ";
for (i = 0; i < G.n - 1; i++)
{
    float min = FLT_MAX;
    int k;
    for (j = 0; j < G.n; j++)
    {
        if (vset[j] == 0 && lowcost[j].w < min)
        {
            k = j;
            min = lowcost[j].w;
        }
    }
    cout << G.v[lowcost[k].v].name << ' ' << G.v[k].name << " ";
    vset[k] = 1;
    c += min;
    for (j = 0; j < G.n; j++)
    {
        if (vset[j] == 0 && G.w[k][j] < lowcost[j].w)
        {
            lowcost[j].w = G.w[k][j];
            lowcost[j].v = k;
        }
    }
}
cout << endl << " 2.该最小生成树权值为: ";
cout << c << endl;
}

//Kruskal
struct Road
{
    int a, b;
    float w;
};
struct Graph_K
{
    struct vertex
    {
        int num;
        string name;
    };
};

```

```

    }v[MAX_N];
    Road road[MAX_E];
    int n;//vertex
    int e;//edge
};

void CreateGraph_K(Graph_K& G)
{
    fstream fin("Graph.txt");
    if (!fin)
    {
        cout << "file failed" << endl;
        exit(1);
    }
    int i, j;
    float k;
    fin >> G.n;//vertex number
    G.e = 0;
    if (G.n > MAX_N)
        exit(OVERFLOW);
    for (i = 0; i < G.n; i++)
    {
        fin >> G.v[i].name;
        G.v[i].num = i;
    }
    i = 0;
    while (!fin.eof())
    {
        fin >> G.road[i].a >> G.road[i].b >> G.road[i].w;
        i++;
        G.e++;
    }
    fin.close();
}

int getRoot(int Father[], int a)
{
    while (a != Father[a])
        a = Father[a];
    return a;
}

void Kruskal(Graph_K& G, float& c)
{
    int Father[MAX_E];
    int i, j;
    int ra, rb;

```



```

    for (i = 0; i < G.n; i++)
    { //初始化并查集
        Father[i] = i;
    }
    for (i = G.e - 1; i >= 1; i--)
    { //按边数由小到大排列
        Road temp;
        for (j = 1; j <= i; j++)
        {
            if (G.road[j].w < G.road[j - 1].w)
            {
                temp = G.road[j - 1];
                G.road[j - 1] = G.road[j];
                G.road[j] = temp;
            }
        }
    }

    cout << "Kruskal 算法" << endl;
    cout << "  1.该最小生成树各边为: ";
    c = 0;
    for (i = 0; i < G.e; i++)
    {
        ra = getRoot(Father, G.road[i].a);
        rb = getRoot(Father, G.road[i].b);
        if (ra != rb)
        {
            Father[ra] = rb;
            c += G.road[i].w;
            cout << G.v[G.road[i].a].name << ' ' << G.v[G.road[i].b].name
e << "    ";
        }
    }
    cout << endl << "  2.该最小生成树权值为: " << c << endl;
}

int main()
{
    float c;

    char choice;
    bool flag = 1;
    cout << "请输入您将使用的算法 (A.Prim   B.Kruskal) : ";
    cin >> choice;

```

```

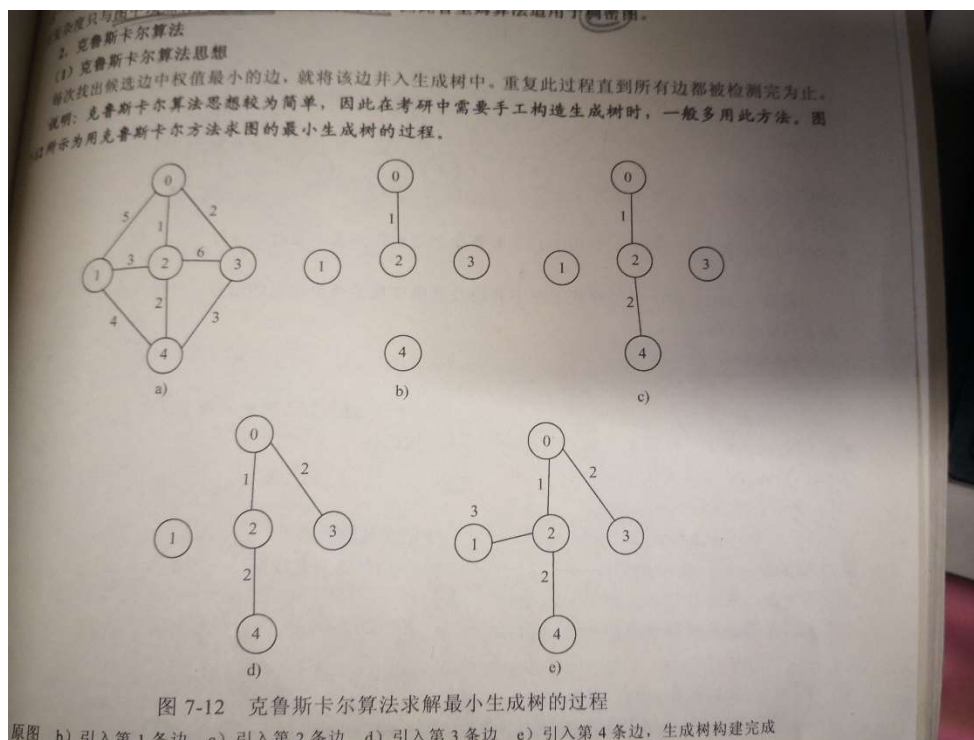
while (flag)
{
    switch (choice)
    {
        case 'A':
        {
            Graph_P G_P;
            CreateGraph_P(G_P);
            Prim(G_P, 0, c);
            flag = 0;
            break;
        }
        case 'B':
        {
            Graph_K G_K;
            CreateGraph_K(G_K);
            Kruskal(G_K, c);
            flag = 0;
            break;
        }
        default:
            cin >> choice;
    }
}
return 0;
}

```

6.4 测试数据及结果

请输入您将使用的算法 (A. Prim B. Kruskal) : A
 Prim算法
 1. 该最小生成树各边为: A C A D C E C B
 2. 该最小生成树权值为: 8

请输入您将使用的算法 (A. Prim B. Kruskal) : B
 Kruskal算法
 1. 该最小生成树各边为: A C C E A D



6.5 改进策略

6.5.1 Prim 算法的时间复杂度为 $O(n^2)$ ，适合稠密图（边数较多）；

6.5.2 Kruscal 算法的时间复杂度依赖于 `sort()` 函数，适合稀疏图（边数较少）。

7 公交线路规划

7.1 数据结构

7.1.1 图

7.1.2 顺序表

7.2 设计思想

7.2.1 开辟堆内存存储公交线路数据

7.2.2 采用迪杰斯特拉算法求最短路径。

7.3 源代码

```
#include <iostream>
#include <fstream>
```

```

#include <sstream>
#include <vector>
#include <map>
#include <algorithm>
#include <iomanip>

using namespace std;
const int inf = 99999;
struct stop
{
    int num;
    string name;
    string bus;
    stop(int num_, string name_, string bus_)
    {
        num = num_;
        name = name_;
        bus = bus_;
    }
};

vector<vector<stop>> rt; // 公交线路信息
map<string, int> stophash; // 公交编号信息
map<int, vector<string>> stopbus; // 能到该站点的公交
int M[6000][6000]; // 邻接矩阵

// 从文件中读取线路信息
void InputRoutine(fstream& fin)
{
    string s;
    stringstream ss;
    string busname, stopname;

    while (fin.peek() != EOF)
    {
        int i = 0;
        vector<stop> r;

        getline(fin, s);
        ss << s;

        ss >> busname;
        r.push_back(stop(i++, busname, busname)); // 0 号位置存储巴士名称
    }
}

```

```

        while (ss)
        {
            getline(ss, stopname, ',');

            if (isascii(stopname[0]) && isspace(stopname[0]))
                stopname.erase(stopname.begin(), stopname.begin() + 1);
//清除空格

            r.push_back(stop(i++, stopname, busname)); //之后每个位置存储
            站点编号及名称
        }
        rt.push_back(r);

        ss.clear();
        ss.str("");
    }
}

void OutputRoutine()
{
    int i, j;
    for (i = 0; i < rt.size(); i++)
    {
        for (j = 0; j < rt[i].size(); j++)
        {
            cout << rt[i][j].name << ' ' << rt[i][j].num << ' ';
        }
        cout << endl;
    }
}

//建立站点名称到编号的映射，建立站点名称到能够到该站点的公交的映射
void Init()
{
    //编号
    int i, j;
    int k = 1;

    for (int i = 0; i < rt.size(); i++)
    {
        for (int j = 1; j < rt[i].size(); j++)
        {
            string name = rt[i][j].name;

```

```

        //map 中没有该站的记录，则进行编号
        if (stophash.find(name) == stophash.end())
        {
            stophash[name] = k++;
        }
        rt[i][j].num = stophash[name];

        stopbus[rt[i][j].num].push_back(rt[i][j].bus);
    }
}

//最小换乘 迪杰斯特拉算法
void CreateM1()
{
    int i, j, k;

    for (i = 1; i < stophash.size() + 1; i++)
    {
        for (j = 1; j < stophash.size() + 1; j++)
        {
            if (i != j)
            {
                M[i][j] = inf;
            }
            else
            {
                M[i][j] = 0;
            }
        }
    }

    for (i = 0; i < rt.size(); i++)
    {
        for (j = 1; j < rt[i].size() - 1; j++)
        {
            for (k = j + 1; k < rt[i].size(); k++)
            {
                M[rt[i][j].num][rt[i][k].num] = 1;
                M[rt[i][k].num][rt[i][j].num] = 1;
            }
        }
    }
}

```

```

}
bool TaskOne(string s1, string s2)
{
    CreateM1();//建立邻接矩阵

    int n = stophash.size();//站点总数

    int* set = new int[n + 1]; //标记数组
    int* dist = new int[n + 1]; //记录到各点最短路径
    int* path = new int[n + 1]; //记录上一个顶点

    //对应两个站点的编号
    int v = stophash[s1];
    int u = stophash[s2];

    //站点不存在
    if (u < 1 || u > n || v < 1 || v > n)
    {
        return false;
    }
    int min, i, j, k;

    //初始化各数组
    for (i = 1; i < n + 1; i++)
    {
        dist[i] = M[v][i];
        set[i] = 0;
        if (M[v][i] < inf)
            path[i] = v;
        else
            path[i] = -1;
    }
    set[v] = 1;
    path[v] = -1;

    //核心操作：每次从剩余顶点中选出一个顶点，通往这个顶点的路径在通往所有剩余
    顶点的路径中是长度最短的
    for (i = 1; i < n; i++)
    {
        min = inf;
        for (j = 1; j < n + 1; j++)
        {
            if (set[j] == 0 && dist[j] < min)

```

```

        {
            k = j;
            min = dist[j];
        }
    }
    set[k] = 1; //并入最短路径

    //再次更新，类似于前面的初始化
    for (j = 1; j < n + 1; j++)
    {
        if (set[j] == 0 && dist[j] > dist[k] + M[k][j])
        {
            dist[j] = dist[k] + M[k][j];
            path[j] = k;
        }
    }
}

//核心操作结束

//输出操作,利用栈逆序输出
int* stack = new int[n];
int top = -1;
while (path[u] != -1)
{
    stack[++top] = u;
    u = path[u];
}
stack[++top] = u;

cout << left;
cout << setw(20) << "最小换乘路线为:";
while (top - 1 != -1)
{
    int rnum = stack[top];
    int lnum = stack[top - 1];
    string bus;
    vector<string> bt;
    for (int i = 0; i < stopbus[rnum].size(); i++)
    {
        for (int j = 0; j < stopbus[lnum].size(); j++)
        {
            if (stopbus[rnum][i] == stopbus[lnum][j])
            {
                bt.push_back(stopbus[rnum][i]);
            }
        }
    }
}

```



```

    }
}
}
sort(bt.begin(), bt.end());
bt.erase(unique(bt.begin(), bt.end()), bt.end());

if (rnum != v)
    cout << setw(20) << "";
for (int i = 0; i < bt.size(); i++)
{
    cout << bt[i];
    if (i != bt.size() - 1)
        cout << " / ";
}
for (auto it = stophash.begin(); it != stophash.end(); it++)
{
    if (it->second == rnum)
        cout << " " << it->first;
}
cout << " ->";
for (auto it = stophash.begin(); it != stophash.end(); it++)
{
    if (it->second == lnum)
        cout << " " << it->first;
}
cout << endl << endl;
top--;
}
cout << endl;

//释放堆内存
delete[] set;
delete[] dist;
delete[] path;
}

//最短路径 迪杰斯特拉算法
void CreateM2(string* from)
{
    int i, j, k;

    for (i = 1; i < stophash.size() + 1; i++)
    {
        for (j = 1; j < stophash.size() + 1; j++)

```

```

        {
            if (i != j)
                M[i][j] = inf;
            else
                M[i][j] = 0;
        }
    }

    for (i = 0; i < rt.size(); i++)
    {
        for (j = 1; j < rt[i].size() - 1; j++)
        {
            for (k = j + 1; k < rt[i].size(); k++)
            {
                int d = k - j;
                int d0 = M[rt[i][j].num][rt[i][k].num];

                if (d < d0)
                {
                    M[rt[i][j].num][rt[i][k].num] = d;
                    M[rt[i][k].num][rt[i][j].num] = d;

                    //当前的公交
                    from[rt[i][j].num] = rt[i][0].name;
                    from[rt[i][k].num] = rt[i][0].name;
                }
            }
        }
    }
}

bool TaskTwo(string s1, string s2)
{
    string* from = new string[stophash.size() + 1]; //保存最短路径下到达该
    站点的公交
    CreateM2(from); //建立邻接矩阵

    int n = stophash.size(); //站点总数

    int* set = new int[n + 1]; //标记数组
    int* dist = new int[n + 1]; //记录到各点最短路径
    int* path = new int[n + 1]; //记录上一个顶点

    //对应两个站点的编号
    int v = stophash[s1];

```

```

int u = stophash[s2];

//站点不存在
if (u < 1 || u > n || v < 1 || v > n)
{
    return false;
}
int min, i, j, k;

//初始化各数组
for (i = 1; i < n + 1; i++)
{
    dist[i] = M[v][i];
    set[i] = 0;
    if (M[v][i] < inf)
        path[i] = v;
    else
        path[i] = -1;
}
set[v] = 1;
path[v] = -1;

//核心操作：每次从剩余顶点中选出一个顶点，通往这个顶点的路径在通往所有剩余
//顶点的路径中是长度最短的
for (i = 1; i < n; i++)
{
    min = inf;
    for (j = 1; j < n + 1; j++)
    {
        if (set[j] == 0 && dist[j] < min)
        {
            k = j;
            min = dist[j];
        }
    }
    set[k] = 1; //并入最短路径

    //再次更新，类似于前面的初始化
    for (j = 1; j < n + 1; j++)
    {
        if (set[j] == 0 && dist[j] > dist[k] + M[k][j])
        {
            dist[j] = dist[k] + M[k][j];

```

```

        path[j] = k;
    }
}

//核心操作结束

//输出操作,利用栈逆序输出
int* stack = new int[n];
int top = -1;
while (path[u] != -1)
{
    stack[++top] = u;
    u = path[u];
}
stack[++top] = u;

cout << left;
cout << setw(20) << "最短乘车路线为:";
while (top - 1 != -1)
{
    int rnum = stack[top];
    int lnum = stack[top - 1];
    string bus;
    if (rnum != v)
        cout << setw(20) << "";
    cout << from[rnum] << " ";
    for (auto it = stophash.begin(); it != stophash.end(); it++)
    {
        if (it->second == rnum)
            cout << it->first;
    }
    cout << " ->";
    for (auto it = stophash.begin(); it != stophash.end(); it++)
    {
        if (it->second == lnum)
            cout << " " << it->first;
    }
    cout << endl << endl;
    top--;
}
cout << endl;

//释放堆内存
delete[] set;

```

```

        delete[] dist;
        delete[] path;
        return true;
    }
}

int main()
{
    fstream fin("routine.txt");

    InputRoutine(fin);
    //OutputRoutine();

    Init();

    string s1, s2;
    while (true)
    {
        cout << "请输入起点站: ";
        cin >> s1; //定坊工业园站

        cout << endl;

        cout << "请输入终点站: ";
        cin >> s2; //双龙路站

        cout << endl;

        if (!TaskOne(s1, s2) || !TaskTwo(s1, s2))
            cout << "输入有误, 请确认站点名称" << endl << endl;
    }
}

```

7.4 测试数据集结果



```
C:\Users\Steven\Desktop\StevenFinch\Github\DataStructure2019_NUAAer\2019版课程设计\07 公交线路规划\Debug\07 公交线路规划.e...  
请输入终点站: 新街口站  
输入有误, 请确认站点名称  
请输入起点站: 南京南站  
请输入终点站: 莫干路站  
输入有误, 请确认站点名称  
请输入起点站: 南京南站站  
请输入终点站: 莫干路站  
最小换乘路线为: 19路 南京南站站 -> 水西门大街大士茶亭站  
78路 水西门大街大士茶亭站 -> 莫干路站  
最短乘车路线为: 792路 南京南站站 -> 岔路口站  
866路 岔路口站 -> 新庄广场东站  
309路 新庄广场东站 -> 新模范马路虹桥站  
557路 新模范马路虹桥站 -> 莫干路站  
请输入起点站:
```

7.5 算法结果与分析:

7.5.1 算法结果



```
C:\Users\Steven\Desktop\StevenFinch\Github\DataStructure2019_NUAAer\2019版课程设计\07 公交线路规划\Debug\07 公交线路规划.e...  
请输入终点站: 新街口站  
输入有误, 请确认站点名称  
请输入起点站: 南京南站  
请输入终点站: 莫干路站  
输入有误, 请确认站点名称  
请输入起点站: 南京南站站  
请输入终点站: 莫干路站  
最小换乘路线为: 19路 南京南站站 -> 水西门大街大士茶亭站  
78路 水西门大街大士茶亭站 -> 莫干路站  
最短乘车路线为: 792路 南京南站站 -> 岔路口站  
866路 岔路口站 -> 新庄广场东站  
309路 新庄广场东站 -> 新模范马路虹桥站  
557路 新模范马路虹桥站 -> 莫干路站  
请输入起点站:
```

7.5.2 算法分析

- 1) TaskOne 和 TaskTwo 中迪杰斯特拉算法部分可以继续抽象出来, 将输出部分作为第三个模块, 这样代码逻辑会更清楚;
- 2) 在做题之前没有整体考虑整个题目的要求, 导致开辟了太多的堆内存, 空间复杂度较高;

8 排序算法比较

8.1 数据结构

8.1.1 顺序表

8.2 设计思想

8.2.1 随机生成十个数据集，每个数据集 20000 个元素；分别记录各排序算法的耗时最终得出平均耗时，进行可视化后比较分析。

8.3 各排序算法分析

8.3.1 直接插入排序：将一个记录插入到已排好序的序列中，从而得到一个新的有序序列。起初将序列的第一个数据看成是一个有序的子序列，然后从第二个记录逐个向该有序的子序列进行有序的插入，直至整个序列有序。

8.3.2 希尔排序：把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至 1 时，整个文件恰被分成一组，算法终止。

8.3.3 冒泡排序：重复地走访过要排序的元素列，依次比较两个相邻的元素，如果顺序错误就把他们交换过来。走访元素的工作是重复地进行直到没有相邻元素需要交换，也就是说该元素列已经排序完成。

8.3.4 快速排序：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

8.3.5 简单选择排序：设所排序序列的记录个数为 n 。 i 取 $1, 2, \dots, n-1$, 从所有 $n-i+1$ 个记录 $(R_i, R_{i+1}, \dots, R_n)$ 中找出排序码最小的记录，与第 i 个记录交换。执行 $n-1$ 趟 后就完成了记录序列的排序。

8.3.6 堆排序：

- 3) 最大堆调整：将堆的末端子节点作调整，使得子节点永远小于父节点；
- 4) 创建最大堆：将堆中的所有数据重新排序；
- 5) 深度堆排序：移除位在第一个数据的根节点，并做最大堆调整的递归运算。

8.3.7 归并排序：将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，即二路归并。

8.3.8 基数排序：

- 6) 最高位优先法：先按 k_1 排序分组，同一组中记录，关键码 k_1 相等，再对各组按 k_2 排序分成子组，之后，对后面的关键码继续这样的排序分组，直到按最次位关键码 k_d 对各子组排序后。再将各组连接起来，便得到一个有序序列。
- 7) 最低位优先：先从 k_d 开始排序，再对 k_d-1 进行排序，依次重复，直到对 k_1 排序后便得到一个有序序列。

8.4 源代码

```
#include<iostream>
#include<fstream>
#include<string>
#include<ctime>
```



```

#include<iomanip>
#include<queue>

using namespace std;
const int n = 20000;//每个数据集 20000 个元素
const int m = 10;//10 个数据集

void PutData();

void InsertSort(int a[]);
void ShellSort(int a[], int derta[], int t);

void BubbleSort(int a[]);
void QuickSort(int a[], int low, int high);

void SelectSort(int a[]);
void Sift(int a[], int low, int high);
void HeapSort(int a[]);

void Merge(int a[], int low, int mid, int high);
void MergeSort(int a[]);

void RadixSort(int a[]);

int main()
{
    PutData();//建立数据集
    int i;
    clock_t start, stop;
    float duration;
    string file;
    //直接插入排序
    cout << "▲直接插入排序: " << endl << endl;
    for (i = 1; i < 11; i++)
    {
        file = "data" + to_string(i) + ".txt";
        fstream fin(file);
        int a[n + 1];
        for (int j = 1; j < n + 1; j++)
        {
            //读取数据集
            fin >> a[j];
        }
        start = clock();
        InsertSort(a);
    }
}

```

```

        stop = clock();
        duration = ((float)(stop - start) / CLK_TCK);    /* CLK_TCK 是机器
没秒经过的 tick */
        cout.fill('0');
        cout << "Time" << setw(2) << i << ": ";
        cout.fill(' ');
        cout << setw(5) << duration << endl << endl;
    }

    //希尔排序
    cout << "▲希尔排序: " << endl << endl;
    for (i = 1; i < 11; i++)
    {
        file = "data" + to_string(i) + ".txt";
        fstream fin(file);
        int a[n + 1];
        for (int j = 1; j < n + 1; j++)
        { //读取数据集
            fin >> a[j];
        }
        start = clock();
        int derta[3] = { 5, 3, 1 };
        ShellSort(a, derta, 3);
        stop = clock();
        duration = ((float)(stop - start) / CLK_TCK);    /* CLK_TCK 是机器
没秒经过的 tick */
        cout.fill('0');
        cout << "Time" << setw(2) << i << ": ";
        cout.fill(' ');
        cout << setw(5) << duration << endl << endl;
    }

    //冒泡排序
    cout << "▲冒泡排序: " << endl << endl;
    for (i = 1; i < 11; i++)
    {
        file = "data" + to_string(i) + ".txt";
        fstream fin(file);
        int a[n + 1];
        for (int j = 1; j < n + 1; j++)
        { //读取数据集
            fin >> a[j];
        }
        start = clock();

```

```

        BubbleSort(a);
        stop = clock();
        duration = ((float)(stop - start) / CLK_TCK);    /* CLK_TCK 是机器
没秒经过的 tick */
        cout.fill('0');
        cout << "Time" << setw(2) << i << ": ";
        cout.fill(' ');
        cout << setw(5) << duration << endl << endl;
    }

    //快速排序
    cout << "▲快速排序: " << endl << endl;
    for (i = 1; i < 11; i++)
    {
        file = "data" + to_string(i) + ".txt";
        fstream fin(file);
        int a[n + 1];
        for (int j = 1; j < n + 1; j++)
        { //读取数据集
            fin >> a[j];
        }
        start = clock();
        QuickSort(a, 1, n);
        stop = clock();
        duration = ((float)(stop - start) / CLK_TCK);    /* CLK_TCK 是机器
没秒经过的 tick */
        cout.fill('0');
        cout << "Time" << setw(2) << i << ": ";
        cout.fill(' ');
        cout << setw(5) << duration << endl << endl;
    }

    //选择排序
    cout << "▲选择排序: " << endl << endl;
    for (i = 1; i < 11; i++)
    {
        file = "data" + to_string(i) + ".txt";
        fstream fin(file);
        int a[n + 1];
        for (int j = 1; j < n + 1; j++)
        { //读取数据集
            fin >> a[j];
        }
        start = clock();

```

```

        SelectSort(a);
        stop = clock();
        duration = ((float)(stop - start) / CLK_TCK);    /* CLK_TCK 是机器
没秒经过的 tick */
        cout.fill('0');
        cout << "Time" << setw(2) << i << ": ";
        cout.fill(' ');
        cout << setw(5) << duration << endl << endl;
    }

//堆排序
cout << "▲堆排序: " << endl << endl;
for (i = 1; i < 11; i++)
{
    file = "data" + to_string(i) + ".txt";
    fstream fin(file);
    int a[n + 1];
    for (int j = 1; j < n + 1; j++)
    { //读取数据集
        fin >> a[j];
    }
    start = clock();
    HeapSort(a);
    stop = clock();
    duration = ((float)(stop - start) / CLK_TCK);    /* CLK_TCK 是机器
没秒经过的 tick */
    cout.fill('0');
    cout << "Time" << setw(2) << i << ": ";
    cout.fill(' ');
    cout << setw(5) << duration << endl << endl;
}

//基数排序
cout << "▲基数排序: " << endl << endl;
for (i = 1; i < 11; i++)
{
    file = "data" + to_string(i) + ".txt";
    fstream fin(file);
    int a[n + 1];
    for (int j = 1; j < n + 1; j++)
    { //读取数据集
        fin >> a[j];
    }
    start = clock();

```

```

        RadixSort(a);
        stop = clock();
        duration = ((float)(stop - start) / CLK_TCK);    /* CLK_TCK 是机器
没秒经过的 tick */
        cout.fill('0');
        cout << "Time" << setw(2) << i << ": ";
        cout.fill(' ');
        cout << setw(5) << duration << endl << endl;
    }

    //归并排序
    cout << "▲归并排序: " << endl << endl;
    for (i = 1; i < 11; i++)
    {
        file = "data" + to_string(i) + ".txt";
        fstream fin(file);
        int a[n + 1];
        for (int j = 1; j < n + 1; j++)
        { //读取数据集
            fin >> a[j];
        }
        start = clock();
        MergeSort(a);
        stop = clock();
        duration = ((float)(stop - start) / CLK_TCK);    /* CLK_TCK 是机器
没秒经过的 tick */
        cout.fill('0');
        cout << "Time" << setw(2) << i << ": ";
        cout.fill(' ');
        cout << setw(5) << duration << endl << endl;
    }
}

//生成数据集
void PutData()
{
    string str_temp;
    int i, j;
    int int_temp; //temp
    srand((unsigned)time(NULL));
    for (i = 1; i < 11; i++)
    {
        str_temp = "data" + to_string(i) + ".txt";
        fstream f(str_temp, fstream::out);
        for (j = 1; j < 20001; j++)

```

```

        {
            int_temp = rand() % 20000;
            f << int_temp << endl;
        }
        f.close();
    }
}

//插入类排序
void InsertSort(int a[])
{
    int i, j;
    int temp;
    for (i = 2; i < n + 1; i++)
    { //core
        temp = a[i];
        j = i - 1;
        while (j >= 1 && temp < a[j])
        {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = temp;
    }
}

void ShellSort(int a[], int derta[], int t)
{ //derta[]为增量数组, t为其中元素个数
    int i, j, k;
    int temp;
    for (k = 0; k < t; k++)
    { //遍历增量数组
        int d = derta[k];
        for (i = 1 + d; i < n + 1; i++)
        {
            if (a[i] < a[i - d])
            {
                temp = a[i];
                for (j = i; j > d; j -= d)
                {
                    if (temp < a[j - d])
                        a[j] = a[j - d];
                    else
                        break;
                }
                a[j] = temp;
            }
        }
    }
}

```

```

    }
}
}
}
//交换类排序
void BubbleSort(int a[])
{
    int i, j;
    int temp;
    for (i = n; i >= 2; i--)
    {
        for (j = 2; j <= i; j++)
        {
            if (a[j - 1] > a[j])
            {
                temp = a[j - 1];
                a[j - 1] = a[j];
                a[j] = temp;
            }
        }
    }
}
void QuickSort(int a[], int low, int high)
{
    int temp;
    int i = low;
    int j = high;
    if (low < high)
    {
        temp = a[low];
        while (i < j)
        {
            while (i < j && a[j] >= temp)
                j--;
            if (i < j)
                a[i++] = a[j];

            while (i < j && a[i] < temp)
                i++;
            if (i < j)
                a[j--] = a[i];
        }
        a[i] = temp;
        QuickSort(a, low, i - 1);
    }
}

```

```

        QuickSort(a, i + 1, high);
    }
}
//选择类排序
void SelectSort(int a[])
{
    int i, j, k;
    int temp;
    for (i = 0; i < n; i++)
    {
        k = i;
        for (j = i + 1; j < n; j++)
        {
            if (a[k] > a[j])
                k = j;
        }
        if (k != i)
        {
            temp = a[i];
            a[i] = a[k];
            a[k] = temp;
        }
    }
}

void Sift(int a[], int low, int high)
{
    int i = low;
    int j = 2 * i;
    int temp = a[i];
    while (j <= high)
    {
        if (j < high && a[j] < a[j + 1])
        { //找出较大的孩子下标
            j++;
        }
        if (temp < a[j])
        { //较大的孩子放到双亲结点的位置上，继续向下寻找
            a[i] = a[j];
            i = j;
            j = 2 * i;
        }
        else //调整完毕
            break;
    }
}

```



```

    }
    a[i] = temp;
}
void HeapSort(int a[])
{
    int i;
    int temp;
    for (i = n / 2; i >= 1; i--)//从第一个非叶节点开始向前调整，数组下标从 1
开始
    {
        //建立大顶堆
        Sift(a, i, n);
    }
    for (i = n; i >= 2; i--)
    {
        //堆排序
        temp = a[1];
        a[1] = a[i];
        a[i] = temp;
        Sift(a, 1, i - 1);
    }
}
//归并排序
void Merge(int a[], int low, int mid, int high)
{
    int* b = (int*)malloc(sizeof(int) * (high - low + 1));
    int i = low;
    int j = mid + 1;
    int k = 0;
    while (i <= mid && j <= high)
    {
        if (a[i] <= a[j])
        {
            b[k++] = a[i++];
        }
        else
        {
            b[k++] = a[j++];
        }
    }
    while (i <= mid)
    {
        b[k++] = a[i++];
    }
    while (j <= high)
    {

```

```

        b[k++] = a[j++];
    }
    for (k = 0, i = low; i <= high; k++, i++)
    {
        a[i] = b[k];
    }
}
void MergeSort(int a[])
{
    int len;
    for (len = 1; len < n + 1; len *= 2)
    {
        int i = 1;
        while (i + 2 * len < n + 1)
        { //归并长为 len 的两个子序列
            Merge(a, i, i + len - 1, i + 2 * len - 1);
            i = i + 2 * len;
        }
        if (i + len <= n)
        {
            Merge(a, i, i + len - 1, n);
        }
    }
}
//基数排序
void RadixSort(int a[])
{
    queue<int> Q[10];
    int radix = 1;
    int i, j, k;
    int m;
    for (k = 1; k <= 5; k++)
    {
        radix *= 10;
        for (i = 1; i < n + 1; i++)
        {
            m = (a[i] % radix) / (radix / 10);
            Q[m].push(a[i]);
        }
        for (i = 1, m = 0; m < 10; m++)
        {
            while (!Q[m].empty())
            {
                a[i] = Q[m].front();

```

```
        Q[m].pop();
        i++;
    }
}
}
```

8.5 测试数据及结果

data1.txt	2019/12/12 21:24	文本文档	124 KB
data2.txt	2019/12/12 21:24	文本文档	124 KB
data3.txt	2019/12/12 21:24	文本文档	124 KB
data4.txt	2019/12/12 21:24	文本文档	124 KB
data5.txt	2019/12/12 21:24	文本文档	124 KB
data6.txt	2019/12/12 21:24	文本文档	124 KB
data7.txt	2019/12/12 21:24	文本文档	124 KB
data8.txt	2019/12/12 21:24	文本文档	124 KB
data9.txt	2019/12/12 21:24	文本文档	124 KB
data10.txt	2019/12/12 21:24	文本文档	124 KB

data1.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

12173
2391
10579
14211
3899
10128
203
3387
12600
10217
7788
12685
17051
8751
9284
12615
16427
5701
10844
19960
6732

data2.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

6597
163
8058
16918
12431
10030
8256
1033
11746
14637
5239
4866
19831
1511
1117
13339
11797
1463
3374
12585
9670

08_排序算法比较

排序方法	平均耗时	Push
1. 直接插入排序	0.0474	<input type="button" value="Begin"/>
2. 希尔排序	0.0127	<input type="button" value="Begin"/>
3. 冒泡排序	0.5174	<input type="button" value="Begin"/>
4. 快速排序	0.0011	<input type="button" value="Begin"/>
5. 选择排序	0.4735	<input type="button" value="Begin"/>
6. 堆排序	0.0017	<input type="button" value="Begin"/>
7. 基数排序	0.0009	<input type="button" value="Begin"/>
8. 归并排序	0.0018	<input type="button" value="Begin"/>

注意:

1. 部分算法时间复杂度较高, 出现卡顿请耐心等待
2. 数据集文件请见根目录

8.6 算法结果分析

8.6.1 内部排序方法比较（引用自孙老师个人主页）：

表 8.1 内部排序方法的比较

排序方法	平均时间	最坏情况	辅助存储	稳定性
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
快速排序	$O(n \log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	不稳定
归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$	稳定
基数排序	$O(d \cdot n)$	$O(d \cdot n)$	$O(rd)$	稳定

8.6.2 改进策略

- 8) 可视化部分可采用多线程进行 QT 编程，可优化程序框架减缓卡顿；
- 9) 优化版的冒泡排序已经掌握。

9 用扑克牌计算 24 点

9.1 数据结构

9.1.1 顺序表

9.2 设计思想

- 9.2.1 首先最外层循环为遍历四个数字和三个符号；
- 9.2.2 第二层构造五个函数用来模拟括号位置；
- 9.2.3 第三层用一个函数进行两个数字的计算并返回计算结果
- 9.2.4 暴力。

9.3 源代码

```
#include <stdio>
#include <iostream>
```

```

#include <ctime>

char op[5] = { '#', '+', '-', '*', '/', }; //0 号位闲置

//计算两个数的运算结果

double cal(double x, double y, int op)
{
    switch (op)
    {
        case 1:
            return x + y;
        case 2:
            return x - y;
        case 3:
            return x * y;
        case 4:
            return x / y;
    }
}

//每一种遍历情况下的运算顺序（括号的位置）

double cal_m1(double i, double j, double k, double t, int op1, int op2,
int op3)
{
    double r1, r2, r3;
    r1 = cal(i, j, op1);
    r2 = cal(r1, k, op2);
    r3 = cal(r2, t, op3);
    return r3;
}

double cal_m2(double i, double j, double k, double t, int op1, int op2,
int op3)
{
    double r1, r2, r3;
    r1 = cal(i, j, op1);
    r2 = cal(k, t, op3);
    r3 = cal(r1, r2, op2);
    return r3;
}

```

```

double cal_m3(double i, double j, double k, double t, int op1, int op2,
int op3)
{
    double r1, r2, r3;
    r1 = cal(j, k, op2);
    r2 = cal(i, r1, op1);
    r3 = cal(r2, t, op3);
    return r3;
}

double cal_m4(double i, double j, double k, double t, int op1, int op2,
int op3)
{
    double r1, r2, r3;
    r1 = cal(k, t, op3);
    r2 = cal(j, r1, op2);
    r3 = cal(i, r2, op1);
    return r3;
}

double cal_m5(double i, double j, double k, double t, int op1, int op2,
int op3)
{
    double r1, r2, r3;
    r1 = cal(j, k, op2);
    r2 = cal(r1, t, op3);
    r3 = cal(i, r2, op1);
    return r3;
}

//符号位全排列

int get_24(int i, int j, int k, int t)
{
    for (int op1 = 1; op1 <= 4; op1++)
    {
        for (int op2 = 1; op2 <= 4; op2++)
        {
            for (int op3 = 1; op3 <= 4; op3++)
            {
                if (cal_m1(i, j, k, t, op1, op2, op3) == 24)
                {
                    printf("(%d%c%d)%c%d)%c%d\n", i, op[op1], j, op[op
2], k, op[op3], t);

```

```

        return 1;
    }
    if (cal_m2(i, j, k, t, op1, op2, op3) == 24)
    {
        printf("(%d%c%d)%c(%d%c%d)\n", i, op[op1], j, op[op
2], k, op[op3], t);
        return 1;
    }
    if (cal_m3(i, j, k, t, op1, op2, op3) == 24)
    {
        printf("(%d%c(%d%c%d))%c%d\n", i, op[op1], j, op[op
2], k, op[op3], t);
        return 1;
    }
    if (cal_m4(i, j, k, t, op1, op2, op3) == 24)
    {
        printf("(%d%c(%d%c(%d%c%d))\n", i, op[op1], j, op[op
2], k, op[op3], t);
        return 1;
    }
    if (cal_m5(i, j, k, t, op1, op2, op3) == 24)
    {
        printf("(%d%c((%d%c%d)%c%d)\n", i, op[op1], j, op[op
2], k, op[op3], t);
        return 1;
    }
    }
}
}
return 0;
}

//数据读取及遍历数字位

int main()
{
    int a[4];
    int t1, t2, t3, t4;
    int flag;
    for (int i = 0; i < 4; i++)
    {
        a[i] = (rand() % 13) + 1;
        printf("%d ", a[i]);
    }
}

```

```

}
for (int i = 0; i < 4; i++)
{
    for (int j = 0; j < 4; j++)
    {
        if (j == i)
            continue;
        for (int k = 0; k < 4; k++)
        {
            if (i == k || j == k)
                continue;
            for (int t = 0; t < 4; t++)
            {
                if (t == i || t == j || t == k)
                    continue;
                t1 = a[i], t2 = a[j], t3 = a[k], t4 = a[t];

                flag = get_24(t1, t2, t3, t4);

                if (flag == 1)
                    break;
            }
            if (flag == 1)
                break;
        }
        if (flag == 1)
            break;
    }
    if (flag == 1)
        break;
}
if (flag == 0)
    printf("-1\n");

return 0;
}

```


9.4 测试数据及结果

```
Microsoft Visual Studio 调试控制台
3 8 4 7      8-((3-7)*4)
8 8 13 5     ((8+8)+13)-5
1 12 12 1    ((1+12)+12)-1
12 6 4 11    ((12-11)*6)*4
6 9 5 3      6+(9*(5-3))
9 6 3 11     9-((6-11)*3)
7 7 2 10     7*(2+(10/7))
11 6 1 4     ((11-6)+1)*4
4 8 11 10    ((4-11)+10)*8
6 1 6 2      (6+(1*6))*2
```

10 跳一跳

10.1 设计思想

10.1.1 模拟

10.2 源代码

```
#include <iostream>
#include <algorithm>
#include <fstream>
using namespace std;

int main()
{
    /*
    得分和输入不同就这么处理
    */

    int x; //本次输入
    int t = 1; //本次得分
    int sum = 0; //总分

    fstream fin("text.txt");
    while (fin >> x && x)
    {
        if (x == 1)
        {
```

```

        t = 1;
        sum += t;
    }
    else if (x == 2)
    {
        if (t == 1)
        {
            t = 2;
        }
        else
        {
            t += 2;
        }
        sum += t;
    }
}
cout << sum << endl;
return 0;
}

```

10.3 测试数据及结果



11 URL 映射

11.1 设计思想

11.1.1 正则表达式匹配

11.2 源代码

```

#include<iostream>
#include<vector>
#include<string>
#include<regex>
#include<fstream>
using namespace std;

typedef struct Node
{
    string rule;
    string name;
}

```

```

Node(string _rule, string _name)
{
    rule = _rule;
    name = _name;
}
} UrlMapping;
bool match(string url, UrlMapping urlmappings)
{
    smatch result;//匹配数组
    if (!regex_match(url, result, regex(urlmappings.rule)))
        return false;

    cout << urlmappings.name;
    for (int i = 1; i < result.size(); i++)
    {
        if (regex_match(result.str(i), regex("\\d+")))
        {
            cout << ' ' << stoi(result.str(i));//去掉前导0
        }
        else
        {
            cout << ' ' << result.str(i);
        }
    }
    cout << endl;
    return true;
}
int main()
{
    fstream fin("text.txt");
    int n, m;
    fin >> n >> m;
    vector<UrlMapping> mappings;
    string rule, name;
    while (n--)
    {
        fin >> rule >> name;
        rule = regex_replace(rule, regex("<int>"), "\\d+");
        rule = regex_replace(rule, regex("<str>"), "[-\\w\\.]+");
        rule = regex_replace(rule, regex("<path>"), "(.+)");
        mappings.push_back(UrlMapping(rule, name));
    }
    string url;

```

```

while (m--)
{
    fin >> url;
    bool find = false;
    for (int i = 0; i < mappings.size(); i++)
    {
        if (match(url, mappings[i]))
        {
            find = true;
            break;
        }
    }
    if (!find)
    {
        cout << 404 << endl;
    }
}
return 0;
}

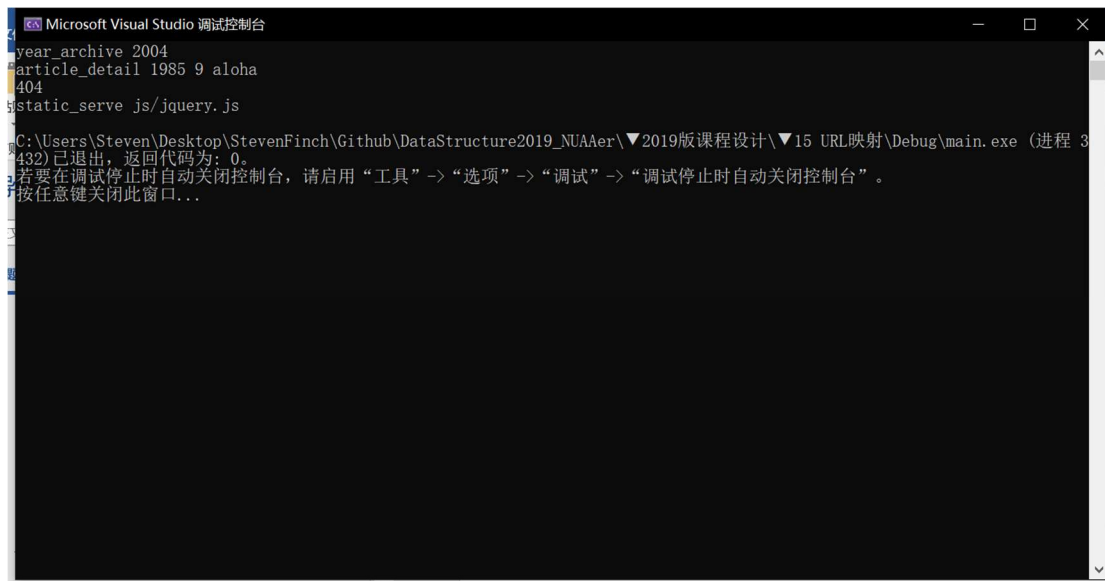
```

11.3 测试数据及结果

```

5 4
/articles/2003/ special_case_2003
/articles/<int>/ year_archive
/articles/<int>/<int>/ month_archive
/articles/<int>/<int>/<str>/ article_detail
/static/<path> static_serve
/articles/2004/
/articles/1985/09/aloha/
/articles/hello/
/static/js/jquery.js

```



12 迷宫问题

12.1 数据结构

12.1.1 栈

12.2 设计思想

12.2.1 不断将可能路径点入栈并进行标记，如果走到不可走的地点则出栈，接下来尝试新的路径；循环往复直到终点。

12.3 源代码

```
#include<bits/stdc++.h>
using namespace std;

const int X = 10;
const int Y = 10;
const int MaxSize = 100;

int M[X][Y]; //maze

struct
{
    int i; //x
    int j; //y
    int di; //direction
}S[MaxSize];
```

```

int top = -1;

void GetPath(int xi, int yi, int xe, int ye)//入口: (xi,yi)    出口:
(xe,ye)
{
    int i, j, di, k;

    //初始方块进栈
    top++;
    S[top].i = xi;
    S[top].j = yi;
    S[top].di = -1;
    M[xi][yi] = -1;//来过防止重复走

    while (top > -1)
    {
        i = S[top].i;
        j = S[top].j;
        di = S[top].di;

        //抵达出口
        if (i == xe && j == ye)
        {
            cout << "迷宫路径如下: " << endl << endl;
            for (k = 0; k <= top; k++)
            {
                cout << "(" << S[k].i << "," << S[k].j << ") ";
                if ((k + 1) % 5 == 0) cout << endl;
            }
            cout << endl << endl;
            return;
        }

        //寻找通路
        int find = 0;
        while (di < 4 && find == 0)
        {
            di++;
            switch (di)
            {
                case 0://left
                    i = S[top].i - 1;
                    j = S[top].j;
                    break;
            }
        }
    }
}

```

```

        case 1://up
            i = S[top].i;
            j = S[top].j + 1;
            break;
        case 2://right
            i = S[top].i + 1;
            j = S[top].j;
            break;
        case 3://down
            i = S[top].i;
            j = S[top].j - 1;
            break;
    }
    if (M[i][j] == 0)
        find = 1;
}
//找到通路下一个方块入栈
if (find == 1)
{
    S[top].di = di;//明确上一方块的 di

    //下一个方块进栈
    top++;
    S[top].i = i;
    S[top].j = j;
    S[top].di = -1;
    M[i][j] = -1;//来过防止重复走
}

//走投无路该方块出栈
else
{
    M[S[top].i][S[top].j] = 0;//该位置可以走
    top--;
}
}
cout << "没有可走路径! " << endl << endl;
}

int main()
{
    //读取迷宫, 0: 通道, 1: 墙
    fstream fin("maze.txt");
    for (int i = 0; i < X; i++)

```

```

{
    for (int j = 0; j < Y; j++)
    {
        fin >> M[i][j];
    }
}
fin.close();

int xi, xe, yi, ye;
while (1)
{
    cout << "请输入迷宫入口(1-8), 输入 0 结束读取: "; // 1 1
    cin >> xi >> yi;
    cout << endl;

    cout << "请输入迷宫出口(1-8), 输入 0 结束读取: "; // 8 8
    cin >> xe >> ye;
    cout << endl;

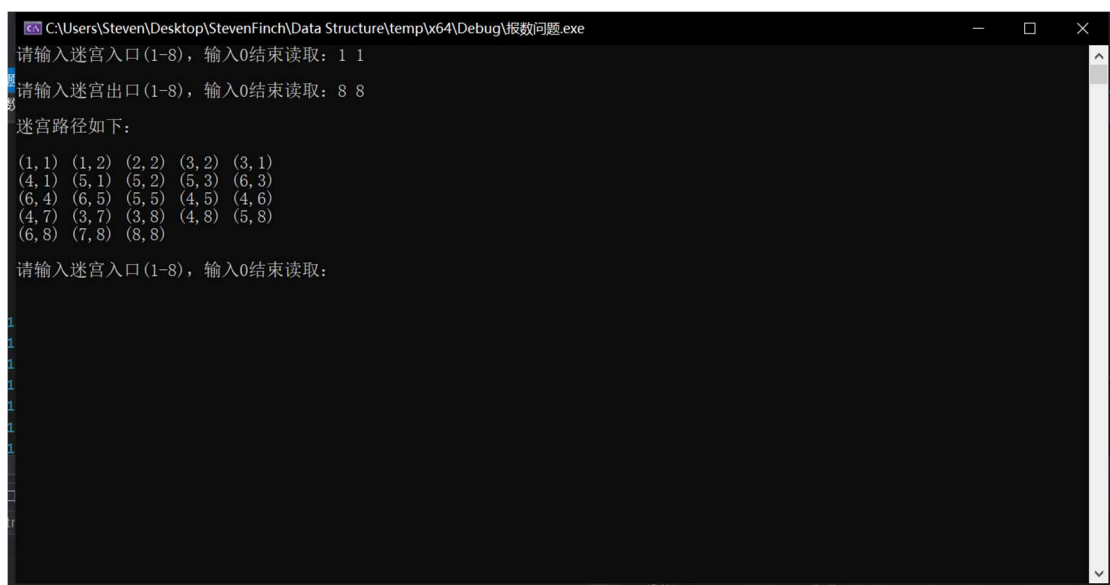
    if (xi == 0 || yi == 0 || xe == 0 || ye == 0)
    {
        cout << "程序结束! " << endl;
        return 0;
    }

    if (1 <= xi && xi <= 8 && 1 <= yi && yi <= 8 && 1 <= xe && xe <
= 8 && 1 <= ye && ye <= 8)
    {
        GetPath(xi, yi, xe, ye);
    }
    else
    {
        cout << "读取失败,请重新输入!" << endl << endl;
    }
}
}

```


12.4 测试数据及结果

```
1 1 1 1 1 1 1 1 1 1
1 0 0 1 1 0 0 1 0 1
1 0 0 1 0 0 0 1 0 1
1 0 0 0 0 1 1 0 0 1
1 0 1 1 1 0 0 0 0 1
1 0 0 0 1 0 0 0 0 1
1 0 1 0 0 0 1 0 0 1
1 0 1 1 1 0 1 1 0 1
1 1 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1
```



13 连连看

13.1 算法思想

13.1.1 第一种情况是直线两个图像直线可以连接;

13.1.2 第二种情况是拐一下可以连接;

13.1.3 第三种情况是拐两下可以连接。

13.1.4 将三种情况进行模拟即可。

13.2 源代码

```
#include "stdafx.h"
#include <graphics.h>
#include <algorithm>
```

```

#include <string>
#include <time.h>
#include <vector>
#include <ctime>

#define WIDTH 10
#define HEIGHT 6

#define GRID_WIDTH 60 // 像素 60 * 60

int data[HEIGHT][WIDTH] = { 0 }; // 随机分配数据
const int picCount = 4; // 每种图片的显示次数
IMAGE img[8];

// 初始化数据
void InitData()
{
    int tmpData[HEIGHT - 2][WIDTH - 2] = { 0 };

    int picNum = 1;
    int curPicCount = 0;
    for (int j = 0; j < HEIGHT - 2; ++j)
    {
        for (int i = 0; i < WIDTH - 2; ++i)
        {
            // 填入数据
            tmpData[j][i] = picNum;
            curPicCount++;

            if (curPicCount == picCount)
            {
                picNum++; // 图片用完，换下一种图片
                curPicCount = 0;
            }
        }
    }

    // 随机数据
    std::random_shuffle((int*)tmpData, (int*)tmpData + (HEIGHT - 2) * (
WIDTH - 2));

    // 再填入 20*10 数组中
    for (size_t i = 1; i < WIDTH - 1; i++)
    {

```

```

        for (size_t j = 1; j < HEIGHT - 1; j++)
        {
            data[j][i] = tmpData[j - 1][i - 1];
        }
    }
}

void LoadImage()
{
    for (size_t i = 0; i < _countof(img); i++)
    {
        std::string file = "./img/" + std::to_string(i + 1) + ".bmp";
        loadimage(&img[i], file.c_str());
    }
}

// 连接情况
bool IsResembling(int x1, int y1, int x2, int y2)
{
    bool s1 = x1 != x2 || y1 != y2; // 不能自己连自己

    bool s2 = data[y1][x1] == data[y2][x2] && data[y1][x1] != 0 && data[y2][x2] != 0; // 图案相同

    return s1 == s2 ? true : false;
}

bool IsHLinked(int x1, int y1, int x2, int y2, std::vector<POINT>& line)
{
    if (y1 != y2)
    {
        // 横向不在一条线
        return false;
    }

    int minX = min(x1, x2); // 找到左边的点
    int maxX = max(x1, x2);

    line.push_back({ x1, y1 });
    line.push_back({ x2, y2 });

    for (int i = minX + 1; i <= maxX - 1; i++)

```

```

    {
        if (data[y1][i] != 0)
        {
            line.clear();
            return false;
        }

        line.push_back({ i, y1 });
    }

    return true;
}

bool IsVLinked(int x1, int y1, int x2, int y2, std::vector<POINT>& line)
{
    if (x1 != x2)
    {
        // 横向不在一条线
        return false;
    }

    int minY = min(y1, y2); // 找到左边的点
    int maxY = max(y1, y2);

    line.push_back({ x1, y1 });
    line.push_back({ x2, y2 });

    for (int i = minY + 1; i <= maxY - 1; i++)
    {
        if (data[i][x1] != 0)
        {
            line.clear();
            return false;
        }
        line.push_back({ x1, i });
    }

    return true;
}

bool IsZeroTurnLinked(int x1, int y1, int x2, int y2, std::vector<POINT>& line)
{

```

```

    if (IsHLinked(x1, y1, x2, y2, line))
    {
        return true;
    }

    if (IsVLinked(x1, y1, x2, y2, line))
    {
        return true;
    }

    return false;
}

bool IsOneTurnLinked(int x1, int y1, int x2, int y2, std::vector<std::vector<POINT> >& lines)
{
    int tmpPointX[2] = { x1, x2 };
    int tmpPointY[2] = { y2, y1 }; // 找到两个黄色点的坐标

    for (size_t i = 0; i < _countof(tmpPointX); i++)
    {
        if (data[tmpPointY[i]][tmpPointX[i]] != 0)
        {
            continue;
        }
        if (IsZeroTurnLinked(tmpPointX[i], tmpPointY[i], x1, y1, lines[0])
            && IsZeroTurnLinked(tmpPointX[i], tmpPointY[i], x2, y2, lines[1]))
        {
            return true;
        }
    }

    return false;
}

bool IsTwoTurnLinked(int x1, int y1, int x2, int y2, std::vector<std::vector<POINT> >& lines)
{
    // 顺着图 1 的延长线纵向遍历所有点
    for (size_t j = 0; j < HEIGHT; j++)

```

```

{
    int tmpX1 = x1;
    int tmpY1 = j;

    if (j == y1)
    {
        continue;    // 与图 1 重合
    }

    if (tmpX1 == x2 && tmpY1 == y2)
    {
        continue;    // 与图 2 重合
    }

    int tmpX2 = x2;
    int tmpY2 = tmpY1;    // 另一个点的坐标

    if (data[tmpY1][tmpX1] != 0
        || data[tmpY2][tmpX2] != 0)
    {
        continue;
    }
    lines[0].clear();
    lines[1].clear();
    lines[2].clear();

    if (IsZeroTurnLinked(tmpX1, tmpY1, tmpX2, tmpY2, lines[0])
        && IsZeroTurnLinked(tmpX1, tmpY1, x1, y1, lines[1])
        && IsZeroTurnLinked(tmpX2, tmpY2, x2, y2, lines[2]))
    {
        return true;
    }
}

// 顺着图 1 的延长线横向遍历所有点
for (size_t j = 0; j < WIDTH; j++)
{
    int tmpX1 = j;
    int tmpY1 = y1;

    if (j == x1)
    {
        continue;    // 与图 1 重合
    }
}

```

```

        if (tmpX1 == x2 && tmpY1 == y2)
        {
            continue;    // 与图 2 重合
        }

        int tmpX2 = tmpX1;
        int tmpY2 = y2;    // 另一个点的坐标

        if (data[tmpY1][tmpX1] != 0
            || data[tmpY2][tmpX2] != 0)
        {
            continue;
        }
        lines[0].clear();
        lines[1].clear();
        lines[2].clear();
        if (IsZeroTurnLinked(tmpX1, tmpY1, tmpX2, tmpY2, lines[0])
            && IsZeroTurnLinked(tmpX1, tmpY1, x1, y1, lines[1])
            && IsZeroTurnLinked(tmpX2, tmpY2, x2, y2, lines[2]))
        {
            return true;
        }
    }

    return false;
}

bool IsLinked(int x1, int y1, int x2, int y2, std::vector<std::vector<POINT> >& lines)
{
    lines.resize(3);

    if (IsZeroTurnLinked(x1, y1, x2, y2, lines[0]) && IsResembling(x1,
y1, x2, y2))
    {
        return true;
    }

    if (IsOneTurnLinked(x1, y1, x2, y2, lines) && IsResembling(x1, y1,
x2, y2))
    {
        return true;
    }
}

```

```

        if (IsTwoTurnLinked(x1, y1, x2, y2, lines) && IsResembling(x1, y1,
x2, y2))
        {
            return true;
        }

        return false;
    }

// 鼠标点击
bool Click(int& x, int& y)
{
    static bool isLDown = false;

    x = -1;
    y = -1;

    bool isClicked = false;

    while (MouseHit())
    {
        MOUSEMSG msg = GetMouseMsg();

        if (msg.mklButton)
        {
            isLDown = true;
            continue;
        }

        if (!msg.mklButton && isLDown)
        {
            // 点击
            isLDown = false;

            x = msg.x / GRID_WIDTH;
            y = msg.y / GRID_WIDTH;

            isClicked = true;
        }
    }
}

```



```

    return isClicked;
}

int main()
{
    initgraph(800, 500);

    srand(time(0));
    InitData();
    LoadImage();

    setlinecolor(0x15C8FC);
    setlinestyle(PS_SOLID, 5);

    int picNum = 0;
    int x1 = 0;
    int y1 = 0;
    int x2 = 0;
    int y2 = 0;

    clock_t start, stop;
    double duration;

    start = clock();
    while (true)
    {
        clearrectangle(0, 0, 2000, 1000);

        for (int j = 0; j < HEIGHT; ++j)
        {
            for (int i = 0; i < WIDTH; ++i)
            {
                // 显示图片
                if (data[j][i] == 0)
                {
                    continue;
                }

                putimage(i * GRID_WIDTH, j * GRID_WIDTH, &img[data[j][i]
] - 1]);
            }
        }
    }
}

```

```

int mouseX = 0;
int mouseY = 0;
if (Click(mouseX, mouseY))
{
    if (picNum == 0)
    {

        if (data[mouseY][mouseX] != 0)
        {
            x1 = mouseX;
            y1 = mouseY;
            picNum = 1;
        }

    }
    else if (picNum == 1)
    {
        if (data[mouseY][mouseX] != 0)
        {
            x2 = mouseX;
            y2 = mouseY;

            picNum = 0;

            std::vector<std::vector<POINT> > lines;

            if (IsLinked(x1, y1, x2, y2, lines))
            {
                for (size_t i = 0; i < lines.size(); i++)
                {

                    for (size_t j = 0; j < lines[i].size(); j++)
                )
                    {
                        if (j == 0)
                        {
                            moveto(lines[i][j].x * GRID_WIDTH +
GRID_WIDTH / 2, lines[i][j].y * GRID_WIDTH + GRID_WIDTH / 2);
                        }
                        else
                        {
                            lineto(lines[i][j].x * GRID_WIDTH +
GRID_WIDTH / 2, lines[i][j].y * GRID_WIDTH + GRID_WIDTH / 2);
                        }
                    }
                }
            }
        }
    }
}

```

```

    }
}

Sleep(100);

data[y1][x1] = 0;
data[y2][x2] = 0;
}

}

}

//判断是否结束
bool flag = true;
for (int j = 0; j < HEIGHT; ++j)
{
    for (int i = 0; i < WIDTH; ++i)
    {
        if (data[j][i] != 0)
        {
            flag = false;
            break;
        }
    }
}
if (flag)
{
    stop = clock();
    duration = ((double)(stop - start) / CLK_TCK); /* CLK_TCK
是机器每秒经过的 tick */

    clearrectangle(0, 0, 2000, 1000);

    RECT r = { 0, 0, 639, 479 };
    std::string str = "Hello World. " + std::to_string(duration
) + "s is your time, " + "You're too young too simple.";

    drawtext(_T(str.c_str()), &r, DT_CENTER | DT_VCENTER | DT_S
INGLELINE);

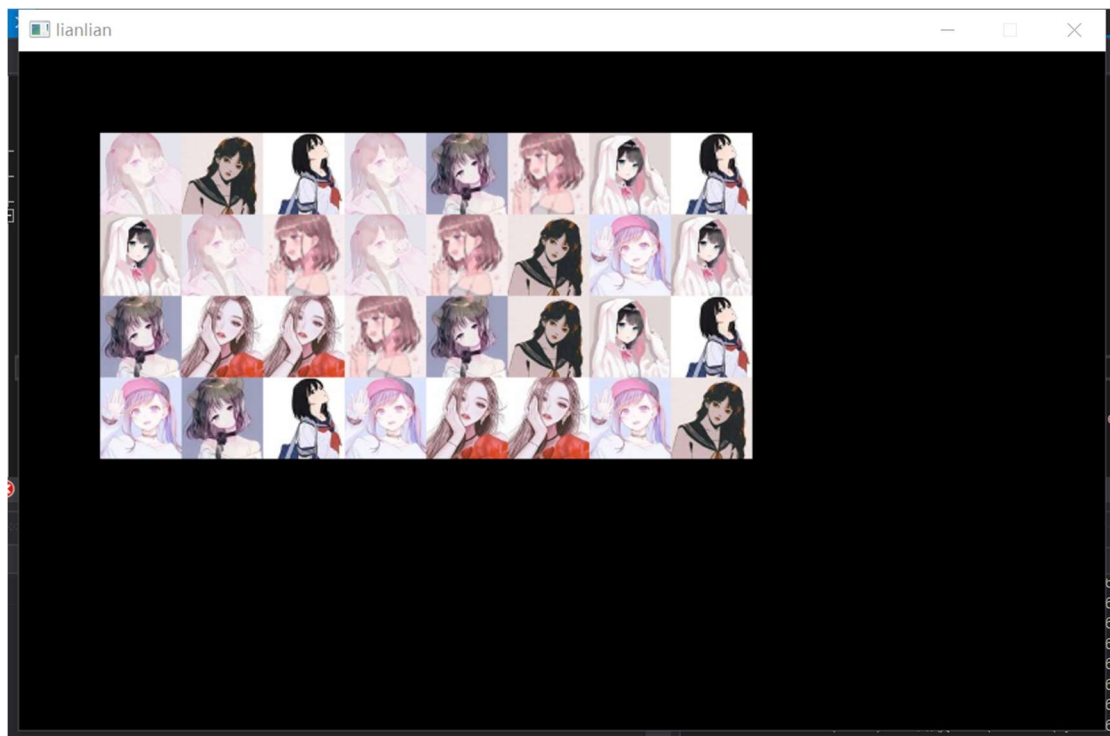
    Sleep(7000);

```

```
        break;
    }
    Sleep(30);
} //end while

closegraph();
return 0;
}
```

13.3 测试数据及结果



14 树的应用

14.1 数据结构

14.1.1 map<string, string>

14.2 算法思想

14.2.1 对输入预处理, 将多行去空白, 整理成一个字符串 json 用于处理,

向 deal()函数传入 json。

14.2.2 从第一个双引号 (第一个字符串的开端) 开始, 先找“:”获取 key,

接着再获取 value。如果“:”的下一位是双引号, 说明 value 是字符串, 找“, ” (如果没有“, ”直接定位到末尾); 反之则下一位一定是“{” (输入保证合法), 说明 value 是对象, 去找下一个与其匹配的大括号。

14.2.3 如果 value 是字符串, 对 key 和 value 进行标准化; 如果 value 是

对象, 对该对象递归调用 deal()函数。无论 value 是什么, 将<key, value>加入 map。

14.2.4 对每个查询, 在 map 中查找对应的键是否存在, 输出相应的信息。

14.3 源代码

```
#include<bits/stdc++.h>
using namespace std;

map<string, string> json;

//从下标 i 开始, 寻找键名或者键值, 并去掉首尾的引号、中间的转义符号'\ '
string get(int& i, string str) {
    string key;
    i++;
    while (i < str.size() && str[i] != '"') {
        //若遇到转义符号, 则跳过一位
        if (str[i] == '\\')
```

```

        {
            key += str[++i];
        }
        else
        {
            key += str[i];
        }
        i++;
    }
    return key;
}

// parent: 当前处理的 json 串的 key
// str: 为尚未解析的字符串
// return: 本次解析的 json 串的长度
int parseJSON(string parent, string str)
{
    string key;
    for (int i = 0; i < str.size(); i++)
    {
        if (str[i] == '}')
        {
            return i;
        }

        if (str[i] == ' ' || str[i] == ',')
        {
            continue;
        }
        // 获取键名
        if (str[i] == '"')
        {
            key = get(i, str);
            continue;
        }
        // 获取键值
        if (str[i] == ':')
        {
            // 去掉可能存在的空格
            while (str[++i] == ' ');

            // 若值为字符串
            if (str[i] == '"') {
                string value = get(i, str);
            }
        }
    }
}

```

```

        if (!parent.empty())
        {
            json[parent + '.' + key] = value;
        }
        else
        {
            json[key] = value;
        }
        continue;
    }
    // 若值为对象
    // 去掉左花括号{后面可能存在的空格
    while (str[++i] == ' ');
    string newParent;
    if (!parent.empty())
    {
        newParent = parent + '.' + key;
    }
    else
    {
        newParent = key;
    }
    // 用{}标记 newParent 的值为对象
    json[newParent] = "{}";
    // 解析子串
    i += parseJSON(newParent, str.substr(i));
}
}
return str.size();
}

int main()
{
    //读 json
    fstream fin("text.txt");
    int n, m;
    fin >> n >> m;
    fin.get();

    string line;
    string str;
    while (n--)
    {
        getline(fin, line);
    }
}

```

```

        str += line;
    }
    parseJSON("", str); //处理 json 字符串

    //判断读入
    while (m--)
    {
        fin >> line;
        if (!json.count(line))
        {
            cout << "NOTEXIST" << endl;
        }
        else if (json[line] == "{}")
        {
            cout << "OBJECT" << endl;
        }
        else
        {
            cout << "STRING " << json[line] << endl;
        }
    }

    return 0;
}

```

14.4 测试数据及结果

```

10 5
{
    "firstName": "John",
    "lastName": "Smith",
    "address": {
        "streetAddress": "2ndStreet",
        "city": "NewYork",
        "state": "NY"
    },
    "esc\\aped": "\\hello\\"
}
firstName
address
address.city
address.postal
esc\\aped

```

