

中国矿业大学计算机学院实验报告

| | | | | | |
|------|------------------|----|------|-------------|----------|
| 课程名称 | 数据结构实践 | | 实验名称 | 第五十八章编程练习 | |
| 班级 | 信息安全 2019-1 班 | 姓名 | 李春阳 | 学号 | 10193657 |
| 仪器组号 | | | 实验日期 | 2021. 1. 17 | |

实验报告要求：1.实验目的 2.实验内容（题目描述，源代码，运行截图，调试情况） 3.实验体会

一、实验目的

- 1 理解树结构的逻辑特性
- 2 熟练掌握二叉树的逻辑结构特性及各种存储方法
- 3 熟练掌握二插树的各种基本操作，尤其是三种递归遍历算法
- 4 熟练掌握图的存储结构
- 5 掌握图的邻接矩阵和邻接表表示分别进行深度和广度优先搜索遍历的算法。

二、实验内容

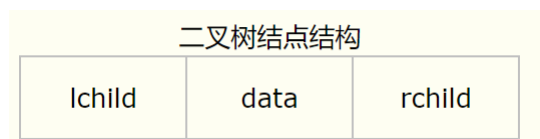
1、第一题

1.1 题目描述

编写一个二叉链表类，试写出求二叉树结点数目和二叉树叶子节点的数目。（只要写二叉链表的前序输入、先序中序后序输出、求节点数目和求叶子节点数目的方法）

1.2 设计思路

二叉树一般多采用二叉链表（binary linked list）存储，其基本思想是：令二叉树的每一个结点对应一个链表结点链表结点除了存放与二叉树结点有关的数据信息外，还要设置指示左右孩子的指针。二叉链表的结点结构如下图所示：



其中，**data** 为数据域，存放该结点的数据信息；
lchild 为左指针域，存放指向左孩子的指针，当左孩子不存在时空指针；
rchild 为右指针域，存放指向右孩子的指针，当右孩子不存在时空指针；

深度遍历采用递归的形式实现，层次遍历引入队列进行选择。在进行层序遍历时，对某一层的结点访问完后，再按照它们的访问次序对各个结点的左孩子和右孩子顺序访问，这样一层一层进行，先访问的结点其左右孩子也要先访问，这符合队列的操作特性，因此，在进行层序遍历时，可设置一个队列存放已访问的结点。

构造函数对二叉树的特殊处理：将二叉树中每个结点的空指针引出一个

虚结点，其值为一特定值，如‘#’，以标识其为空。二叉链表属于动态内存分配，需要在析构函数中释放二叉链表的所有结点。在释放某结点时，该结点的左右都子树已经释放，所以应该采用后序遍历。

1.3 运行截图



```
Microsoft Visual Studio 调试控制台
请输入树:
A
B
D
#
G
#
#
#
C
E
#
#
F
#
#
树高: 4
结点数目: 7
叶节点个数: 3
前序遍历: ABDGCEF
中序遍历: DGBAECF
后序遍历: GDBEFCA
层次遍历: ABCDEFG

C:\Users\lenovo\Desktop\数据结构实践\数据结构作业三\第三次作业\Debug\第三次作业.exe (进程 20188) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。...
```

2、第二题

2.1 题目描述

对图的邻接矩阵和邻接表表示分别进行深度优先搜索遍历算法的实现。

2.2 设计思路

```
#define DefaultVertices 20
#define maxWeight 2147483647
```

//邻接矩阵

```
template <class T, class E>
class Graph1
{
public:
    Graph1(int sz = DefaultVertices);
    ~Graph1();

    //插入点
    void addVertices(T v);
    //插入边
    void addEdge(int i, int j, E e);

    //深度优先查找
    void DFSTraverse();
    void DFS(int i);
```

```

friend istream& operator >> (istream& in, Graph1<T, E>& G);    //输入
friend ostream& operator << (ostream& out, Graph1<T, E>& G);    //输出

//顶点表
T* VerticesList;
//邻接矩阵
E** Edge;
//访问标志数组;
int* visited;
//图中最大顶点数
int maxVertices;
//当前顶点数
int numVertices;
//当前边数
int numEdges;
//给出顶点 vertex 在图中位置
int getVertexPos(T vertex)
{
    for (int i = 0; i < numVertices; i++)
    {
        if (VerticesList[i] == vertex)
        {
            return i;
        }
    }
    return -1;
};

template <class T, class E>
Graph1<T,E>::Graph1(int sz)
{
    maxVertices = sz;
    numVertices = 0;
    numEdges = 0;
    visited = new int[maxVertices];
    int i, j;
    VerticesList = new T[maxVertices];    //创建顶点表
    //邻接矩阵
    Edge = (E**) new E * [maxVertices];
    for (i = 0; i < maxVertices; i++)
    {
        Edge[i] = new E[maxVertices];
    }
}

```

```

    }
    //矩阵初始化
    for (i = 0; i < maxVertices; i++)
    {
        for (j = 0; j < maxVertices; j++)
        {
            Edge[i][j] = (i == j) ? 0 : maxWeight;
        }
    }
}

template <class T, class E>
Graph1<T, E>::~~Graph1()
{
    delete[] VerticesList;
    for (int i = 0; i < numVertices; i++)
    {
        delete[] Edge[i];
    }
    delete[] Edge;
}

template <class T, class E>
void Graph1<T, E>::addVertices(T v)
{
    VerticesList[numVertices] = v;
    numVertices += 1;
}

template <class T, class E>
void Graph1<T, E>::addEdge(int i, int j, E e)
{
    Edge[i - 1][j - 1] = e;
    Edge[j - 1][i - 1] = e;
    numEdges++;
}

```

```

template <class T, class E>
void Graph1<T, E>::DFS(int i)
{
    int j;
    visited[i] = 1;
    cout << VerticesList[i] << " ";
    for ( j = 0; j < numVertices; j++)
    {
        if (Edge[i][j] == 1 && Edge[i][j] != maxWeight && !visited[j])
        {
            DFS(j);
        }
    }
}

template <class T, class E>
void Graph1<T, E>::DFS Traverse()
{
    int i;
    for ( i = 0; i < numVertices; i++)
    {
        visited[i] = 0;
    }
    for ( i = 0; i < numVertices; i++)
    {
        if (!visited[i])
        {
            DFS(i);
        }
    }
}

```

```

template <class T, class E>
ostream& operator << (ostream& out, Graph1<T, E>& G)
{
    int i, j;
    out << "节点个数" << G.numVertices << "边个数是: " << G.numEdges
<< "\n";
    out << "节点为: " << "\n";
    for ( i = 0; i < G.numVertices; i++)
    {
        out << G.VerticesList[i] << " ";
    }
    out << "\n";
    out << "邻接矩阵为: " << "\n";
    for ( i = 0; i < G.numVertices; i++)
    {
        for ( j = 0; j < G.numVertices; j++)
        {
            out << G.Edge[i][j] << " ";
        }
        out << "\n";
    }
    return out;
}

template <class T, class E>
void print(Graph1<T, E>& G)
{
    int i, j;
    cout << "节点个数" << G.numVertices << "边个数是: " << G.numEdges
<< "\n";
    cout << "节点为: " << "\n";
    for (i = 0; i < G.numVertices; i++)
    {
        cout << G.VerticesList[i] << " ";
    }
    cout << "\n";
    cout << "邻接矩阵为: " << "\n";
    for (i = 0; i < G.numVertices; i++)
    {
        for (j = 0; j < G.numVertices; j++)
        {
            cout << G.Edge[i][j] << " ";
        }
        cout << "\n";
    }
}

```

```

    }
}

int main()
{
    int n = 0;
    Graph1<char, int> G;
    cout << "输入的节点数: ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        char str;
        cin >> str;
        G.addVertices(str);
    }
    cout << "输入的边数: ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        int a, b;
        cin >> a >> b;
        G.addEdge(a, b, 1);
    }
    print(G);
    G.DFSTraverse();

    return 0;
}
// 定义边表结点
template <class T, class E>
struct ArcNode
{
    int adjvex; // 邻接点域
    E wight;
    ArcNode<T,E>* next;

    ArcNode()
    {
        next = NULL;
    }
}

```

```

    }

    ArcNode(E e,int i, ArcNode<T, E>* n = NULL)
    {
        wight = e;
        adjvex = i;
        next = n;
    }

};
// 定义顶点表结点
template <class T, class E>
struct VertexNode
{
    T data;
    ArcNode<T, E>* firstedge;

    VertexNode()
    {
        firstedge = NULL;
    }
};

template <class T, class E>
class Graph2
{
public:
    Graph2(int sz = DefaultVertices);
    ~Graph2();

    //插入点
    void addVertices(T v);
    //插入边
    void addEdge(int i, int j, E e);

    //深度优先查找
    void DFSTraverse();
    void DFS(int i);

    //顶点表
    VertexNode<T,E>* VerticesList;

```



```

//访问标志数组;
int* visited;
//图中最大顶点数
int maxVertices;
//当前顶点数
int numVertices;
//当前边数
int numEdges;
//给出顶点 vertex 在图中位置
int getVertexPos(T vertex)
{
    for (int i = 0; i < numVertices; i++)
    {
        if (VerticesList[i].data == vertex)
        {
            return i;
        }
    }
    return -1;
};

};

template <class T, class E>
Graph2<T,E>::Graph2(int sz)
{
    maxVertices = sz;
    numVertices = 0;
    numEdges = 0;
    visited = new int[maxVertices];
    VerticesList = new VertexNode<T, E>[maxVertices]; //创建顶点表

}

template <class T, class E>
Graph2<T, E>::~~Graph2()
{

}

template <class T, class E>
void Graph2<T, E>::addVertices(T v)
{
    VerticesList[numVertices].data = v;

```

```

        numVertices++;
    }

template <class T, class E>
void Graph2<T, E>::addEdge(int i, int j, E e)
{
    ArcNode<T, E>* ptredge1 = NULL;
    ptredge1 = VerticesList[i].firstedge;
    while (ptredge1 != NULL)
    {
        ptredge1 = ptredge1->next;
    }
    ptredge1 = new ArcNode<T, E>(e, j);

    ArcNode<T, E>* ptredge2 = NULL;
    ptredge2 = VerticesList[j].firstedge;
    while (ptredge2 != NULL)
    {
        ptredge2 = ptredge2->next;
    }
    ptredge2 = new ArcNode<T, E>(e, i);
    numEdges++;
}

template <class T, class E>
void Graph2<T, E>::DFSTraverse()
{
    int i;
    for ( i = 0; i < numVertices; i++)
    {
        visited[i] = 0;
    }
    for (i = 0; i < numVertices; i++)
    {
        if (!visited[i])
        {
            DFS(i);
        }
    }
}

template <class T, class E>
void Graph2<T, E>::DFS(int i)
{

```

```

        ArcNode<T, E>* ptredge = NULL;
        visited[i] = 1;
        cout << VerticesList[i].data << " ";
        ptredge = VerticesList[i].firstedge;
        while (ptredge != NULL)
        {
            if (!visited[ptredge->adjvex])
            {
                DFS(ptredge->adjvex);
            }
            ptredge = ptredge->next;
        }
    }

int main()
{

    int n = 0;
    Graph2<char, int> G;
    cout << "输入的节点数: ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        char str;
        cin >> str;
        G.addVertices(str);
    }
    cout << "输入的边数: ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        int a, b;
        cin >> a >> b;
        G.addEdge(a, b, 1);
    }
    cout << "深度优先 DFS: " << endl;
    G.DFSTraverse();
    cout << endl;

    return 0;
}

```

}

2.3 运行截图

```
Microsoft Visual Studio 调试控制台
输入的节点数: 9
A
B
C
D
E
F
H
I
G
输入的边数: 15
1 2
1 6
2 9
6 5
5 7
5 4
4 7
4 9
9 7
2 8
8 4
8 3
3 4
2 3
节点个数9边个数是: 15
节点为:
A B C D E F H I G
邻接矩阵为:
0 1 2147483647 2147483647 2147483647 1 2147483647 2147483647 2147483647
1 0 1 2147483647 2147483647 2147483647 2147483647 1 1
2147483647 1 0 1 2147483647 2147483647 2147483647 1 2147483647
2147483647 2147483647 1 0 1 2147483647 1 1 1
2147483647 2147483647 2147483647 2147483647 1 0 1 2147483647 2147483647
1 2147483647 2147483647 2147483647 1 0 2147483647 2147483647 1
2147483647 2147483647 2147483647 1 1 2147483647 0 2147483647 1
2147483647 1 1 1 2147483647 2147483647 2147483647 0 2147483647
2147483647 1 2147483647 1 2147483647 1 1 2147483647 0
A B C D E F H I G
C:\Users\lenovo\Desktop\数据结构实践\数据结构作业三\第三次作业\Debug\第三次作业.exe (进程 20008) 已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。...
```

```
Microsoft Visual Studio 调试控制台
输入的节点数: 9
A
B
C
D
E
F
H
I
G
输入的边数: 15
1 2
1 6
2 9
6 5
5 7
5 4
4 7
4 9
9 7
2 8
8 4
8 3
3 4
2 3
深度优先DFS:
A B C D E F H I G
C:\Users\lenovo\Desktop\数据结构实践\数据结构作业三\第三次作业\Debug\第三次作业.exe (进程 18324) 已退出, 代码为 0。
```

3、第三题

3.1 题目描述

实现霍夫曼编、解码

(1) 输入一系列字符及其出现频率并以此构造霍夫曼树进行编码并输出码表, 另输入一段文字, 对其进行霍夫曼编码。例: CASTCASTSATATATASA

(2) 在 1 中已构成的霍夫曼树的基础上, 输入一段 01 编码, 要求输出其解码的原文。例: 111011001110110011001001001001100

3.2 源代码

CAST CAST SAT AT A TASA

字符集合是 {C,A,S,T}, 各个字符出现的频度 (次数) 是 $W=\{2,7,4,5\}$ 。

若给每个字符以等长编码 (2 位二进制足够)

A : 00 T : 10 C : 01 S : 11

则总编码长度为 $(2+7+4+5) * 2 = 36$ 。

若按各个字符出现的概率不同而给予不等长编码, 可望减少总编码长度。

各字符出现概率为 $\{2/18, 7/18, 4/18, 5/18\}$, 化整为 $\{2, 7, 4, 5\}$ 。

以它们为各叶结点上的权值, 建立 Huffman 树。左分支赋 0, 右分支赋 1, 得 Huffman 编码(变长编码)。

A : 0 T : 10 C : 110 S : 111

它的总编码长度: $71+52+(2+4)*3 = 35$ 。比等长编码的情形要短。

总编码长度正好等于 Huffman 树的带权路径长度 WPL。

Huffman 编码是一种前缀编码, 即任一个二进制编码不是其他二进制编码的前缀。解码时不会混淆。

代码如下:

```
#include<iostream>
#include<string>
#include<algorithm>
#include<map>
using namespace std;
const int DefaultSize = 20;
struct HuffmanNode {
    int count;
    char data;
    HuffmanNode* leftChild, * rightChild;
    HuffmanNode(char d = '', int p = 0, HuffmanNode* left = NULL,
HuffmanNode* right = NULL)
        :count(p), data(d), leftChild(left), rightChild(right) {}
};

bool cmp(HuffmanNode* a, HuffmanNode* b) {
    return a->count > b->count;
}

class HuffmanTree {
public:
    ~HuffmanTree() { destroy(root); }
    HuffmanNode* root;
```

```

map<char, string> code;
string s1;
HuffmanTree(string s) {
    s1 = s;
    int size = 0;
    HuffmanNode* w[100];
    for (int i = 0; i < s.length(); i++) {
        int j = 0, flag = 1;
        for (; j < size; j++) {
            if (w[j]->data == s[i]) {
                flag = 0;
                w[j]->count++;
                break;
            }
        }
        if (flag) {
            w[j] = new HuffmanNode;
            w[j]->data = s[i];
            w[j]->count++;
            size++;
        }
    }
    for (int i = 0; i < size - 1; i++) {
        sort(w, w + size - i, cmp);
        w[size - i - 2] = new HuffmanNode(' ', w[size - i - 1]->count +
w[size - i - 2]->count, w[size - i - 1], w[size - i - 2]);
    }
    root = w[0];
}
void destroy(HuffmanNode*& subTree) {
    if (subTree != NULL) {
        destroy(subTree->leftChild);
        destroy(subTree->rightChild);
        delete subTree;
    }
}
void output() {
    makecode(root, "");
}
void makecode(HuffmanNode* subtree, string s) {
    if (subtree->data != ' ') {
        code[subtree->data] = s;
        cout << subtree->data << " " << s << endl;
    }
    return;
}

```

```

    }
    makecode(subtree->leftChild, s + "0");
    makecode(subtree->rightChild, s + "1");
}
void decode(string s) {
    string ans = "";
    for (int i = 0; i < s.length(); i) {
        HuffmanNode* now = root;
        while (now->data == '') {
            if (s[i] == '0') {
                now = now->leftChild;
            }
            else {
                now = now->rightChild;
            }
            i++;
        }
        ans += now->data;
    }
    cout << ans;
}
string tocode() {
    string ans = "";
    for (int i = 0; i < s1.length(); i++) {
        ans += code[s1[i]];
    }
    return ans;
}
};

```

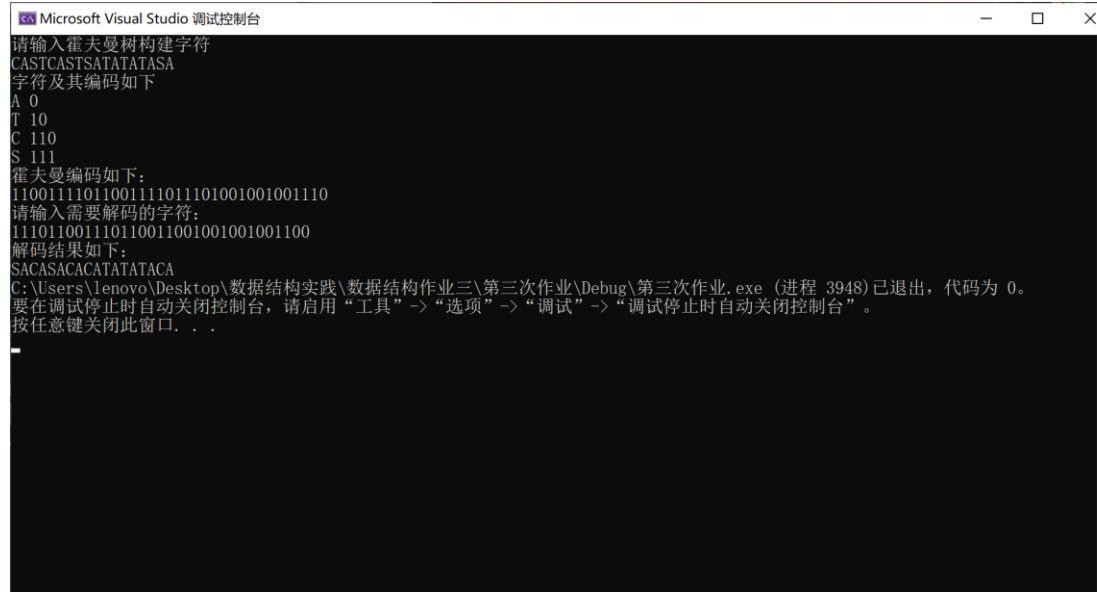
```

int main()
{
    string s;
    cout << "请输入霍夫曼树构建字符" << endl;
    cin >> s;
    HuffmanTree a(s);
    cout << "字符及其编码如下" << endl;
    a.output();
    cout << "霍夫曼编码如下: " << endl;
    cout << a.tocode() << endl;
    cout << "请输入需要解码的字符: " << endl;
    cin >> s;
    cout << "解码结果如下: " << endl;
}

```

```
a.decode(s);  
return 0;  
}
```

3.3 运行截图



```
Microsoft Visual Studio 调试控制台  
请输入霍夫曼树构建字符  
CASTCASTATATATASA  
字符及其编码如下  
A 0  
T 10  
C 110  
S 111  
霍夫曼编码如下:  
11001111011001111011101001001001110  
请输入需要解码的字符:  
111011001110110011001001001001100  
解码结果如下:  
SACASACATATATACA  
C:\Users\lenovo\Desktop\数据结构实践\数据结构作业三\第三次作业\Debug\第三次作业.exe (进程 3948) 已退出, 代码为 0。  
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。  
按任意键关闭此窗口. . .
```

4、第四题

4.1 题目描述

校园导游咨询。要求：

(1) 设计一个校园的平面图，所含景点不少于 8 个。以图中的顶点表示校内各景点，存放景点名称、代号、简介等信息。以边表示路径，存放路径长度等相关信息。

(2) 为来访客人图中任意景点相关信息的查询。

(3) 为来访客人提供图中任意景点的问路查询，即查询任意两个景点之间的一条最短路。

4.2 源代码

信息结点：


```
struct InformationNode
{
    int SerialNumber;
    string Name;
    string Information;

    InformationNode()
    {
        ;
    }

    InformationNode(int n, string name, string information)
    {
        SerialNumber = n;
        Name = name;
        Information = information;
    }
};
```

地图类:

```

template <class T, class E>
class Graph1
{
public:
    Graph1(int sz = DefaultVertices):
        ~Graph1():

        //插入点
        void addVertices(T v):
        //插入边
        void addEdge(int i, int j, E e):

        //深度优先查找
        void DFSTraverse():
        void DFS(int i):

        //最短路径
        void Dijkstra(int v0, Patharc* p, ShortPathTable* D):

        friend istream& operator >> (istream& in, Graph1<T, E>& G): //输入
        friend ostream& operator << (ostream& out, Graph1<T, E>& G): //输出

        //顶点表
        T* VerticesList:
        //邻接矩阵
        E** Edge:
        //访问标志数组;
        int* visited:
        //图中最大顶点数
        int maxVertices:
        //当前顶点数
        int numVertices:
        //当前边数
        int numEdges:
        //给出顶点vertex在图中位置
        int getVertexPos(T vertex)
        {
            for (int i = 0; i < numVertices; i++)
            {
                if (VerticesList[i] == vertex)
                {
                    return i:
                }
            }
            return -1:
        }

```

最短路径实现:

```

#define MAXVEX 9
//最短路径
template <class T, class E>
void Graph1<T, E>::Dijkstra(int v0, Patharc *p, ShortPathTable *D)
{
    int v, w, k = v0, min;
    int final[MAXVEX];
    for (v = 0; v < numVertices; v++)
    {
        final[v] = 0;
        (*D)[v] = Edge[v0][v];
        (*p)[v] = 0;
    }
    (*D)[v0] = 0;
    final[v0] = 1;
    //(*p)[v0] = v0;

    for (v = 1; v < numVertices; v++)
    {
        min = INFINITY;
        for (w = 0; w < numVertices; w++)
        {
            if (!final[w] && (*D)[w] < min)
            {
                k = w;
                min = (*D)[w];
            }
        }
        final[k] = 1;
        for (w = 0; w < numVertices; w++)
        {
            if (!final[w] && (min + Edge[k][w] < (*D)[w]))
            {
                (*D)[w] = min + Edge[k][w];
                (*p)[w] = k;
            }
        }
    }
    //(*p)[v0] = v0;
}

```

函数主体

```

int main()
{
    int m;
    cout << "shuasho" << endl;
    Graph1<InformationNode, int> G;
    cout << "请输入图的信息: " << endl;
    cout << "输入的节点数: ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        InformationNode node;
        string name, information;
        node.SerialNumber = i + 1;
        cin >> node.Name >> node.Information;
        G.addVertices(node);
    }
    cout << "输入的边数: ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        int a, b, w;
        cin >> a >> b >> w;
        G.addEdge(a, b, w);
    }
    Print(G);
    int d[9];
    int m[9];
    Patharc* p = &m;
    ShortPathTable* D = &d;
    string strpath;

    while (true)
    {
        cout << "\n*****\n" << endl;
        print(G, -1);
        cout << "[1]查询信息" << " " << "[2]查询路径" << "[3]退出" << endl;
        cin >> n;
        switch (n)
        {
            case 1:
                int m;
                cout << "请输入查找的序号: ";
                cin >> m;
                print(G, m);
                break;
            case 2:
                int m1, m2;
                cout << "请输入出发的序号和到达的序号: ";
                cin >> m1 >> m2;
                cout << "Dijkstra最短路径, " << G.VerticesList[m1 - 1].Name << "出发: " << endl;
                G.Dijkstra(m1 - 1, p, D);
                for (int i = 0; i < G.numVertices; i++)
                {
                    cout << i + 1 << " " << G.VerticesList[i].Name << " " << G.VerticesList[(p)[i]].Name << " " << (*D)[i] << endl;
                }
                strpath = path(G, m1 - 1, m2 - 1, p, D);
                cout << "路径为: " << strpath << endl;
                break;
            default:
                exit(1);
                break;
        }
    }

    return 0;
}

```

示例输入

*

示例

9

A ThiSisA

B ThiSisB

C ThiSisC

D ThiSisD

E ThiSisE

F ThiSisF

H ThiSisH

I ThiSisI

G ThiSisG

15

1 2 3

1 6 4

2 9 6

6 9 4

6 5 9

5 7 7

5 4 5

4 7 3

4 9 4

9 7 6

2 8 5

8 4 3

8 3 6

3 4 4

2 3 6

9

A ThiSisA

B ThiSisB

C ThiSisC

D ThiSisD

E ThiSisE

F ThiSisF

H ThiSisH

I ThiSisI

G ThiSisG

16

1 2 1

1 3 5

2 3 3

2 4 7

2 5 5

3 5 1

3 6 7

4 5 2

4 7 3

5 6 3

5 7 6

5 8 9

6 7 5

7 8 2

7 9 7

8 9 4

4.3 运行截图

```
C:\Users\lenovo\Desktop\数据结构实践\数据结构作业三\第三次作业\Debug\第三次作业.exe
shuahe
请输入图的信息：
输入的节点数： 9
A ThiSisA
B ThiSisB
C ThiSisC
D ThiSisD
E ThiSisE
F ThiSisF
H ThiSisH
I ThiSisI
G ThiSisG
输入的边数： 16
1 2 1
1 3 5
2 3 3
2 4 7
2 5 5
3 5 1
3 6 7
4 5 2
4 7 3
5 6 3
5 7 6
5 8 9
6 7 5
7 8 2
7 9 7
8 9 4
节点个数9边个数是： 16
节点为：
A B C D E F H I G
邻接矩阵为：
0 1 5 410065408 410065408 410065408 410065408 410065408 410065408 410065408
1 0 3 7 5 410065408 410065408 410065408 410065408
5 3 0 410065408 1 7 410065408 410065408 410065408
410065408 7 410065408 0 2 410065408 3 410065408 410065408
410065408 5 1 2 0 3 6 9 410065408
410065408 410065408 7 410065408 3 0 5 410065408 410065408
410065408 410065408 410065408 3 6 5 0 2 7
410065408 410065408 410065408 410065408 9 410065408 2 0 4
410065408 410065408 410065408 410065408 410065408 7 4 0

*****

C:\Users\lenovo\Desktop\数据结构实践\数据结构作业三\第三次作业\Debug\第三次作业.exe
序号 名称 介绍
1 A ThiSisA
2 B ThiSisB
3 C ThiSisC
4 D ThiSisD
5 E ThiSisE
6 F ThiSisF
7 H ThiSisH
8 I ThiSisI
9 G ThiSisG
[1]查询信息 [2]查询路径[3]退出
1
请输入查找的序号： 2
序号 名称 介绍
2 B ThiSisB

*****

序号 名称 介绍
1 A ThiSisA
2 B ThiSisB
3 C ThiSisC
4 D ThiSisD
5 E ThiSisE
6 F ThiSisF
7 H ThiSisH
8 I ThiSisI
9 G ThiSisG
[1]查询信息 [2]查询路径[3]退出
2
请输入出发的序号和到达的序号： 1 9
Dijkstra最短路径，A出发：
1 A A 0
2 B A 1
3 C B 4
4 D E 7
5 E C 5
6 F E 8
7 H D 10
8 I H 12
9 G I 16
路径为： A->B->C->E->D->H->I->G

*****
```

三、程序附件

// 第三次作业.cpp :

```
#include <iostream>
#include <queue>
#include <string>
#include <cstring>
#include <vector>
```

```
using namespace std;
```

```

/**
 *
 * （一）基本题
 * 1、编写一个二叉链表类，试写出求二叉树结点数目和二叉树叶子节点
 * 的数目。（只要写二叉链表的前序输入、先序中序后序输出、求节点
 * 数目和求叶子节点数目的方法）
 * 2、对图的邻接矩阵和邻接表表示分别进行深度优先搜索遍历算法的实现。
 */

/*
#define BTN(T) BinTreeNode<T>*

template <class T>
struct BinTreeNode
{
    T data;
    BTN(T) leftChild;
    BTN(T) rightChild;

    BinTreeNode()
    {
        leftChild = NULL;
        rightChild = NULL;
    }
    BinTreeNode(T x, BTN(T) lChild = NULL, BTN(T) rChild = NULL)
    {
        data = x;
        leftChild = lChild;
        rightChild = rChild;
    }
};

template <class T>
class BinaryTree
{
public:

    BinaryTree(T value);
    ~BinaryTree();

```

```

//返回树高
int Hight() { return hight; };
//返回节点数
int Size() { return size; };
//返回叶节点数
int NumberOfLeafNodes() { return numberOfLeafNodes; };
//设置空标准
void setRefValue(T value);

// 递归前序遍历二叉树
void PreOrder() { PreOrder(root); };

// 递归中序遍历二叉树
void InOrder() { InOrder(root); };

// 递归后序遍历二叉树
void PostOrder() { PostOrder(root); };

// 层序遍历二叉树
void LeverOrder();

private:
//叶子数
int numberOfLeafNodes;
//个数
int size;
//树高
int hight;
//根节点
BTN(T) root;
//停止输入标志
T RefValue;
//树高
int treeDepth(BTN(T) bt);
//个数
void set(BTN(T) bt);
// 构造函数调用
BTN(T) Creat(BTN(T)bt);
// 析构函数调用
void Release(BTN(T)bt);
// 前序遍历函数调用
void PreOrder(BTN(T) bt);
// 中序遍历函数调用
void InOrder(BTN(T) bt);

```



```

    // 后序遍历函数调用
    void PostOrder(BTN(T) bt);
};

template <class T> BinaryTree<T>::BinaryTree(T value)
{
    RefValue = value;
    root = Creat(root);
    hight = treeDepth(root);
    size = 0;
    numberOfLeafNodes = 0;
    set(root);
}

template <class T> BinaryTree<T>::~~BinaryTree()
{
    Release(root);
}

template<class T>
int BinaryTree<T>::treeDepth(BTN(T) bt)
{
    if (bt == NULL)
    {
        return 0;
    }
    int nLeft = treeDepth(bt->leftChild);
    int nRight = treeDepth(bt->rightChild);
    return nLeft > nRight ? nLeft + 1 : nRight + 1;
}

template<class T>
void BinaryTree<T>::set(BTN(T) bt)
{
    // 递归调用的结束条件
    if (bt == NULL)
    {
        return;
    }
    // 访问根节点 bt 的数据域
    size++;
    if (bt->leftChild == NULL && bt->rightChild == NULL)

```

```

    {
        numberOfLeafNodes++;
    }
    // 前序递归遍历 bt 的左子树
    set(bt->leftChild);
    // 前序递归遍历 bt 的右子树
    set(bt->rightChild);
}

template<class T>
void BinaryTree<T>::setRefValue(T value)
{
    RefValue = value;
}

template<class T>
inline void BinaryTree<T>::LeverOrder()
{
    // 定义一个队列
    queue<BTN(T)> Q;
    // 顺序队列
    Q.push(root);
    while (!Q.empty())
    {
        BTN(T) bt = Q.front();
        Q.pop();
        cout << bt->data;
        if (bt->leftChild != NULL)
        {
            Q.push(bt->leftChild);
        }
        if (bt->rightChild != NULL)
        {
            Q.push(bt->rightChild);
        }
    }
}

template<class T>
inline BTN(T) BinaryTree<T>::Creat(BTN(T) bt)
{
    T value;
    // 输入结点的数据信息

```

```

    cin >> value;
    // 建立一棵空树
    if (value == RefValue)
        bt = NULL;
    else
    {
        // 生成一个结点，数据域为 ch
        bt = new BinTreeNode<T>;
        bt->data = value;
        // 递归建立左子树
        bt->leftChild = Creat(bt->leftChild);
        // 递归建立右子树
        bt->rightChild = Creat(bt->rightChild);
    }
    return bt;
}

```

```

template<class T>
inline void BinaryTree<T>::Release(BTN(T) bt)
{
    if (bt != NULL)
    {
        // 释放左子树
        Release(bt->leftChild);
        // 释放右子树
        Release(bt->rightChild);
        // 释放根节点
        delete bt;
    }
}

```

```

template<class T>
inline void BinaryTree<T>::PreOrder(BTN(T) bt)
{
    // 递归调用的结束条件
    if (bt == NULL)
    {
        return;
    }
    // 访问根节点 bt 的数据域
    cout << bt->data;
}

```

```

    // 前序递归遍历 bt 的左子树
    PreOrder(bt->leftChild);
    // 前序递归遍历 bt 的右子树
    PreOrder(bt->rightChild);
}

template<class T>
inline void BinaryTree<T>::InOrder(BTN(T) bt)
{
    if (bt == NULL)
    {
        return;
    }
    InOrder(bt->leftChild);
    cout << bt->data;
    InOrder(bt->rightChild);
}

template<class T>
inline void BinaryTree<T>::PostOrder(BTN(T) bt)
{
    if (bt == NULL)
    {
        return;
    }
    PostOrder(bt->leftChild);
    PostOrder(bt->rightChild);
    cout << bt->data;
}

int main()
{
    std::cout << "Hello World!\n";
    BinaryTree<char> BT('#');

    cout << "树高: ";
    cout << BT.Hight() << endl;
    cout << "结点数目: ";
    cout << BT.Size() << endl;
    cout << "叶节点个数: ";
    cout << BT.NumberOfLeafNodes() << endl;
    cout << "前序遍历: ";

```

```

    BT.PreOrder();
    cout << endl;
    cout << "中序遍历: ";
    BT.InOrder();
    cout << endl;
    cout << "后序遍历: ";
    BT.PostOrder();
    cout << endl;
    cout << "层次遍历: ";
    BT.LeverOrder();
    cout << endl;

    return 0;
}

*/

/*
* 示例:
A
B
D
#
G
#
#
#
C
E
#
#
F
#
#
*/

/*
* 2、对图的邻接矩阵和邻接表表示分别进行深度优先搜索遍历算法的实现。
*/

#define DefaultVertices 20
#define maxWeight 2147483647

```

```

//邻接矩阵
/*
template <class T, class E>
class Graph1
{
public:
    Graph1(int sz = DefaultVertices);
    ~Graph1();

    //插入点
    void addVertices(T v);
    //插入边
    void addEdge(int i, int j, E e);

    //深度优先查找
    void DFSTraverse();
    void DFS(int i);

    friend istream& operator >> (istream& in, Graph1<T, E>& G);    //输入
    friend ostream& operator << (ostream& out, Graph1<T, E>& G);    //输出

    //顶点表
    T* VerticesList;
    //邻接矩阵
    E** Edge;
    //访问标志数组;
    int* visited;
    //图中最大顶点数
    int maxVertices;
    //当前顶点数
    int numVertices;
    //当前边数
    int numEdges;
    //给出顶点 vertex 在图中位置
    int getVertexPos(T vertex)
    {
        for (int i = 0; i < numVertices; i++)
        {
            if (VerticesList[i] == vertex)
            {
                return i;
            }
        }
    }
}

```

```

        }
        return -1;
    };
};

template <class T, class E>
Graph1<T,E>::Graph1(int sz)
{
    maxVertices = sz;
    numVertices = 0;
    numEdges = 0;
    visited = new int[maxVertices];
    int i, j;
    VerticesList = new T[maxVertices]; //创建顶点表
    //邻接矩阵
    Edge = (E**) new E * [maxVertices];
    for (i = 0; i < maxVertices; i++)
    {
        Edge[i] = new E[maxVertices];
    }
    //矩阵初始化
    for (i = 0; i < maxVertices; i++)
    {
        for (j = 0; j < maxVertices; j++)
        {
            Edge[i][j] = (i == j) ? 0 : maxWeight;
        }
    }
}

template <class T, class E>
Graph1<T, E>::~~Graph1()
{
    delete[] VerticesList;
    for (int i = 0; i < numVertices; i++)
    {
        delete[] Edge[i];
    }
    delete[] Edge;
}

```

```

template <class T, class E>
void Graph1<T, E>::addVertices(T v)
{
    VerticesList[numVertices] = v;
    numVertices += 1;
}

```

```

template <class T, class E>
void Graph1<T, E>::addEdge(int i, int j, E e)
{
    Edge[i - 1][j - 1] = e;
    Edge[j - 1][i - 1] = e;
    numEdges++;
}

```

```

template <class T, class E>
void Graph1<T, E>::DFS(int i)
{
    int j;
    visited[i] = 1;
    cout << VerticesList[i] << " ";
    for ( j = 0; j < numVertices; j++)
    {
        if (Edge[i][j] == 1 && Edge[i][j] != maxWeight && !visited[j])
        {
            DFS(j);
        }
    }
}

```

```

template <class T, class E>
void Graph1<T, E>::DFS Traverse()
{
    int i;
    for ( i = 0; i < numVertices; i++)
    {
        visited[i] = 0;
    }
    for ( i = 0; i < numVertices; i++)

```



```

    {
        if (!visited[i])
        {
            DFS(i);
        }
    }
}

```

```

template <class T, class E>
ostream& operator << (ostream& out, Graph1<T, E>& G)
{
    int i, j;
    out << "节点个数" << G.numVertices << "边个数是: " << G.numEdges <<
"\n";
    out << "节点为: " << "\n";
    for ( i = 0; i < G.numVertices; i++)
    {
        out << G.VerticesList[i] << " ";
    }
    out << "\n";
    out << "邻接矩阵为: " << "\n";
    for ( i = 0; i < G.numVertices; i++)
    {
        for ( j = 0; j < G.numVertices; j++)
        {
            out << G.Edge[i][j] << " ";
        }
        out << "\n";
    }
    return out;
}

```

```

template <class T, class E>

```

```

void print(Graph1<T, E>& G)
{
    int i, j;
    cout << "节点个数" << G.numVertices << "边个数是: " << G.numEdges <<
"\n";
    cout << "节点为: " << "\n";
    for (i = 0; i < G.numVertices; i++)
    {
        cout << G.VerticesList[i] << " ";
    }
    cout << "\n";
    cout << "邻接矩阵为: " << "\n";
    for (i = 0; i < G.numVertices; i++)
    {
        for (j = 0; j < G.numVertices; j++)
        {
            cout << G.Edge[i][j] << " ";
        }
        cout << "\n";
    }
}
*/

/*

int main()
{
    int n = 0;
    cout << "niuashdu" << endl;
    Graph1<char, int> G;
    cout << "输入的节点数: ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        char str;
        cin >> str;
        G.addVertices(str);
    }
    cout << "输入的边数: ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        int a, b;
        cin >> a >> b;
    }
}

```

```

        G.addEdge(a, b, 1);
    }
    print(G);
    G.DFSTraverse();

    return 0;
}
*/

//邻接表

/*
// 定义边表结点
template <class T, class E>
struct ArcNode
{
    int adjvex;// 邻接点域
    E wight;
    ArcNode<T,E>* next;

    ArcNode()
    {
        next = NULL;
    }

    ArcNode(E e,int i, ArcNode<T, E>* n = NULL)
    {
        wight = e;
        adjvex = i;
        next = n;
    }
};

// 定义顶点表结点
template <class T, class E>
struct VertexNode
{
    T data;
    ArcNode<T, E>* firstedge;

    VertexNode()

```

```

    {
        firstedge = NULL;
    }
};

template <class T, class E>
class Graph2
{
public:
    Graph2(int sz = DefaultVertices);
    ~Graph2();

    //插入点
    void addVertices(T v);
    //插入边
    void addEdge(int i, int j, E e);

    //深度优先查找
    void DFSTraverse();
    void DFS(int i);

    //顶点表
    VertexNode<T,E>* VerticesList;
    //访问标志数组;
    int* visited;
    //图中最大顶点数
    int maxVertices;
    //当前顶点数
    int numVertices;
    //当前边数
    int numEdges;
    //给出顶点 vertex 在图中位置
    int getVertexPos(T vertex)
    {
        for (int i = 0; i < numVertices; i++)
        {
            if (VerticesList[i].data == vertex)
            {
                return i;
            }
        }
    }
}

```

```

        return -1;
    };

};

template <class T, class E>
Graph2<T,E>::Graph2(int sz)
{
    maxVertices = sz;
    numVertices = 0;
    numEdges = 0;
    visited = new int[maxVertices];
    VerticesList = new VertexNode<T, E>[maxVertices]; //创建顶点表
}

template <class T, class E>
Graph2<T, E>::~~Graph2()
{
}

template <class T, class E>
void Graph2<T, E>::addVertices(T v)
{
    VerticesList[numVertices].data = v;
    numVertices++;
}

template <class T, class E>
void Graph2<T, E>::addEdge(int i, int j, E e)
{
    ArcNode<T, E>* ptredge1 = NULL;
    ptredge1 = VerticesList[i].firstedge;
    while (ptredge1 != NULL)
    {
        ptredge1 = ptredge1->next;
    }
    ptredge1 = new ArcNode<T, E>(e, j);

    ArcNode<T, E>* ptredge2 = NULL;
    ptredge2 = VerticesList[j].firstedge;
    while (ptredge2 != NULL)
    {

```

```

        ptredge2 = ptredge2->next;
    }
    ptredge2 = new ArcNode<T, E>(e, i);
    numEdges++;
}

```

```

template <class T, class E>
void Graph2<T, E>::DFS Traverse()
{
    int i;
    for ( i = 0; i < numVertices; i++)
    {
        visited[i] = 0;
    }
    for (i = 0; i < numVertices; i++)
    {
        if (!visited[i])
        {
            DFS(i);
        }
    }
}

```

```

template <class T, class E>
void Graph2<T, E>::DFS(int i)
{
    ArcNode<T, E>* ptredge = NULL;
    visited[i] = 1;
    cout << VerticesList[i].data << " ";
    ptredge = VerticesList[i].firstedge;
    while (ptredge != NULL)
    {
        if (!visited[ptredge->adjvex])
        {
            DFS(ptredge->adjvex);
        }
        ptredge = ptredge->next;
    }
}

```

```

*/

```

```

/*

```

```

int main()

```

```

{
    cout << "niuashdu" << endl;

    int n = 0;
    cout << "niuashdu" << endl;
    Graph2<char, int> G;
    cout << "输入的节点数: ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        char str;
        cin >> str;
        G.addVertices(str);
    }
    cout << "输入的边数: ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        int a, b;
        cin >> a >> b;
        G.addEdge(a, b, 1);
    }
    cout << "深度优先 DFS: " << endl;
    G.DFS_Traverse();
    cout << endl;

    return 0;
}
*/

```

```

/*
* 示例
9
A
B
C
D
E
F

```

H
I
G
15
12
16
29
69
65
57
54
47
49
97
28
84
83
34
23

*/

/*

******/

/*

* 1、实现霍夫曼编、解码

(1) 输入一系列字符及其出现频率并以此构造霍夫曼树进行编码并输出码表，另输入一段文字，

对其进行霍夫曼编码。例：CASTCASTSATATATASA

CAST CAST SAT AT A TASA

(2) 在 1 中已构成的霍夫曼树的基础上，输入一段 01 编码，要求输出其解码的原文。例：111011001110110011001001001001100

*/

/*

*

CAST CAST SAT AT A TASA

字符集合是 {C,A,S,T}，各个字符出现的频度（次数）是 $W=\{2,7,4,5\}$ 。

若给每个字符以等长编码 (2 位二进制足够)

A : 00 T : 10 C : 01 S : 11

则总编码长度为 $(2+7+4+5) * 2 = 36$ 。

若按各个字符出现的概率不同而给予不等长编码, 可望减少总编码长度。

各字符出现概率为 $\{2/18, 7/18, 4/18, 5/18\}$, 化整为 $\{2, 7, 4, 5\}$ 。

以它们为各叶结点上的权值, 建立 Huffman 树。左分支赋 0, 右分支赋 1, 得 Huffman 编码(变长编码)。

A : 0 T : 10 C : 110 S : 111

它的总编码长度: $71+52+(2+4)*3 = 35$ 。比等长编码的情形要短。

总编码长度正好等于 Huffman 树的带权路径长度 WPL。

Huffman 编码是一种前缀编码, 即任一个二进制编码不是其他二进制编码的前缀。解码时不会混淆。

*/

//最小堆的类定义

/*

template <class Type>

class MinHeap

{

public:

MinHeap(int maxSize);

MinHeap(Type arr[], int n);

~MinHeap() { delete[] heap; }

const MinHeap<Type>& operator =(const MinHeap& R);

int Insert(const Type& x);

int RemoveMin(Type& x);

int IsEmpty() const

{

return CurrentSize == 0;

}

int IsFull() const

{

return CurrentSize == MaxHeapSize;

}

void MakeEmpty() { CurrentSize = 0; }

private:

enum { DefaultSize = 10 };

Type* heap;

```

    int CurrentSize;
    int MaxHeapSize;
    void FilterDown(int i, int m);
    void FilterUp(int i);
};

template <class Type> MinHeap <Type> ::MinHeap(int maxSize)
{//根据给定大小 maxSize,建立堆对象
    MaxHeapSize = DefaultSize < maxSize ? maxSize : DefaultSize;           // 确定堆大小
    heap = new Type[MaxHeapSize]; //创建堆空间
    CurrentSize = 0;                                                         //初始化
}

//堆的建立
template <class Type> MinHeap <Type> ::MinHeap(Type arr[], int n)
{//根据给定数组中的数据和大小,建立堆对象
    MaxHeapSize = DefaultSize < n ? n : DefaultSize;
    heap = new Type[MaxHeapSize];
    heap = arr;                                                             //数组传送
    CurrentSize = n;                                                         //当前堆大小
    int currentPos = (CurrentSize - 2) / 2;    //最后非叶
    while (currentPos >= 0)
    {
        //从下到上逐步扩大,形成堆
        FilterDown(currentPos, CurrentSize - 1);
        //从 currentPos 开始,到 CurrentSize 为止,调整
        currentPos--;
    }
}

//最小堆的向下调整算法
template <class Type> void MinHeap<Type> ::FilterDown(int start, int EndOfHeap)
{
    int i = start, j = 2 * i + 1;      //j 是 i 的左子女
    Type temp = heap[i];
    while (j <= EndOfHeap) {
        if (j < EndOfHeap && heap[j].key >
            heap[j + 1].key) j++;      //两子女中选小者
        if (temp.key <= heap[j].key) break;
        else { heap[i] = heap[j]; i = j; j = 2 * j + 1; }
    }
    heap[i] = temp;
}

//堆的插入

```

```

template <class Type> int MinHeap<Type> ::Insert(const Type& x)
{
    //在堆中插入新元素 x
    if (CurrentSize == MaxHeapSize)    //堆满
    {
        cout << "堆已满" << endl;    return 0;
    }
    heap[CurrentSize] = x;              //插在表尾
    FilterUp(CurrentSize);              //向上调整为堆
    CurrentSize++;                      //堆元素增一
    return 1;
}

//最小堆的向上调整算法
template <class Type> void MinHeap<Type> ::FilterUp(int start)
{
    //从 start 开始,向上直到 0,调整堆
    int j = start, i = (j - 1) / 2;    // i 是 j 的双亲
    Type temp = heap[j];
    while (j > 0) {
        if (heap[i].key <= temp.key) break;
        else { heap[j] = heap[i];    j = i;    i = (i - 1) / 2; }
    }
    heap[j] = temp;
}

//最小堆的删除算法
template <class Type> int MinHeap <Type> ::RemoveMin(Type& x)
{
    if (!CurrentSize)
    {
        cout << "堆已空" << endl;
        return 0;
    }
    x = heap[0];                      //最小元素出队列
    heap[0] = heap[CurrentSize - 1];
    CurrentSize--;                    //用最小元素填补
    FilterDown(0, CurrentSize - 1);
    //从 0 号位置开始自顶向下调整为堆
    return 1;
}

```

//Huffman 树的类定义

const int DefaultSize = 20; //缺省权值集合大小

template <class T, class E>

struct HuffmanNode

{ //树结点的类定义

E data; //结点数据

HuffmanNode<T, E>* parent;

HuffmanNode<T, E>* leftChild, * rightChild;

//左、右子女和父结点指针

HuffmanNode() : parent(NULL), leftChild(NULL), rightChild(NULL) {} // 构造函数

HuffmanNode(E elem, HuffmanNode<T, E>* pr = NULL, HuffmanNode<T, E>* left = NULL, HuffmanNode<T, E>* right = NULL): data(elem), parent(pr), leftChild(left), rightChild(right) {}

};

template <class T, class E>

class HuffmanTree

{ //Huffman 树类定义

public:

HuffmanTree(E w[], int n); //构造函数

//~HuffmanTree() { deleteTree(root); } //析构函数

protected:

HuffmanNode<T, E>* root; //树的根

void deleteTree(HuffmanNode<T, E>* t); //删除以 t 为根的子树

void mergeTree(HuffmanNode<T, E>& ht1, HuffmanNode<T, E>& ht2, HuffmanNode<T, E>* & parent);

};

//建立 Huffman 树的算法

template <class T, class E>

HuffmanTree<T, E>::HuffmanTree(E w[], int n)

{

//给出 n 个权值 w[1]~w[n], 构造 Huffman 树

MinHeap<T, E> hp; //使用最小堆存放森林

HuffmanNode<T, E>* parent, & first, & second;

HuffmanNode<T, E>* NodeList =

new HuffmanNode<T, E>[n]; //森林

for (int i = 0; i < n; i++) {

NodeList[i].data = w[i + 1];

NodeList[i].leftChild = NULL;

NodeList[i].rightChild = NULL;

```

        NodeList[i].parent = NULL;
        hp.Insert(NodeList[i]);    //插入最小堆中
    }
    for (int i = 0; i < n - 1; i++) {    //n-1 趟, 建 Huffman 树
        hp.Remove(first);    //根权值最小的树
        hp.Remove(second);    //根权值次小的树
        merge(first, second, parent);    //合并
        hp.Insert(*parent);    //重新插入堆中
        root = parent;    //建立根结点
    }
};

template <class T, class E>
void HuffmanTree<T, E>::mergeTree(HuffmanNode<T, E>& bt1,HuffmanNode<T,
E>& bt2,HuffmanNode<T, E>*& parent)
{
    parent = new E;
    parent->leftChild = &bt1;
    parent->rightChild = &bt2;
    parent->data.key = bt1.root->data.key + bt2.root->data.key;
    bt1.root->parent = bt2.root->parent = parent;
};

*/

//采用静态链表的 Huffman 树
/*
const int n = 20;
const int m = 2 * n - 1;

typedef struct
{
    float weight;
    int parent, lchild, rchild;
} HTNode;

typedef HTNode HuffmanTree[m];
//建立 Huffman 树的算法
void CreateHuffmanTree(HuffmanTree T,float fr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
    {

```

```

        T[i].weight = fr[i];
    }
    for (i = 0; i < m; i++)
    {
        T[i].parent = T[i].lchild = T[i].rchild = -1;
    }
    for (i = n; i < m; i++)
    {
        //求 n-1 次根
        int min1, min2, MaxNum;
        int pos1, pos2;
        for (int j = 0; j < i; j++)    //检测前 i 棵树
            if (T[j].parent == -1)    //可参选的树根
                if (T[j].weight < min1) {    //选最小
                    pos2 = pos1; min2 = min1;
                    pos1 = j; min1 = T[j].weight;
                }
                else if (T[j].weight < min2)    //选次小
                {
                    pos2 = j; min2 = T[j].weight;
                }
        T[i].lchild = pos1; T[i].rchild = pos2;
        T[i].weight = T[pos1].weight + T[pos2].weight;
        T[pos1].parent = T[pos2].parent = i;
    }
};

*/

```

/*1、实现霍夫曼编、解码

(1) 输入一系列字符及其出现频率并以此构造霍夫曼树进行编码并输出码表，另输入一段文字，对其进行霍夫曼编码。例：CASTCASTSATATATASA

(2) 在 1 中已构成的霍夫曼树的基础上，输入一段 01 编码，要求输出其解码的原文。例：111011001110110011001001001001100

*/

/*

```

CASTCASTSATATATASA
111011001110110011001001001001100

```

*/

/*

```

#include<iostream>

```

```

#include<string>

```

```

#include<algorithm>
#include<map>
using namespace std;
const int DefaultSize = 20;
struct HuffmanNode {
    int count;
    char data;
    HuffmanNode* leftChild, * rightChild;
    HuffmanNode(char d = '\0', int p = 0, HuffmanNode* left = NULL,
HuffmanNode* right = NULL)
        :count(p), data(d), leftChild(left), rightChild(right) {}
};

bool cmp(HuffmanNode* a, HuffmanNode* b) {
    return a->count > b->count;
}

class HuffmanTree {
public:
    ~HuffmanTree() { destroy(root); }
    HuffmanNode* root;
    map<char, string> code;
    string s1;
    HuffmanTree(string s) {
        s1 = s;
        int size = 0;
        HuffmanNode* w[100];
        for (int i = 0; i < s.length(); i++) {
            int j = 0, flag = 1;
            for (; j < size; j++) {
                if (w[j]->data == s[i]) {
                    flag = 0;
                    w[j]->count++;
                    break;
                }
            }
            if (flag) {
                w[j] = new HuffmanNode;
                w[j]->data = s[i];
                w[j]->count++;
                size++;
            }
        }
        for (int i = 0; i < size - 1; i++) {

```

```

        sort(w, w + size - i, cmp);
        w[size - i - 2] = new HuffmanNode(' ', w[size - i - 1]->count + w[size -
i - 2]->count, w[size - i - 1], w[size - i - 2]);
    }
    root = w[0];
}
void destroy(HuffmanNode*& subTree) {
    if (subTree != NULL) {
        destroy(subTree->leftChild);
        destroy(subTree->rightChild);
        delete subTree;
    }
}
void output() {
    makecode(root, "");
}
void makecode(HuffmanNode* subtree, string s) {
    if (subtree->data != ' ') {
        code[subtree->data] = s;
        cout << subtree->data << " " << s << endl;
        return;
    }
    makecode(subtree->leftChild, s + "0");
    makecode(subtree->rightChild, s + "1");
}
void decode(string s) {
    string ans = "";
    for (int i = 0; i < s.length(); i++) {
        HuffmanNode* now = root;
        while (now->data == ' ') {
            if (s[i] == '0') {
                now = now->leftChild;
            }
            else {
                now = now->rightChild;
            }
            i++;
        }
        ans += now->data;
    }
    cout << ans;
}
string tocode() {
    string ans = "";

```



```

        for (int i = 0; i < s1.length(); i++) {
            ans += code[s1[i]];
        }
        return ans;
    }
};

int main()
{
    string s;
    cout << "请输入霍夫曼树构建字符" << endl;
    cin >> s;
    HuffmanTree a(s);
    cout << "字符及其编码如下" << endl;
    a.output();
    cout << "霍夫曼编码如下: " << endl;
    cout << a.tocode() << endl;
    cout << "请输入需要解码的字符: " << endl;
    cin >> s;
    cout << "解码结果如下: " << endl;
    a.decode(s);
    return 0;
}

*/

/*****
*****/

/*****
*****/

/*
* 2、校园导游咨询。要求：
    (1) 设计一个校园的平面图，所含景点不少于 8 个。以图中的顶点表示校内各景点，
        存放景点名称、代号、简介等信息。以边表示路径，存放路径长度等相关信

```

息。

(2) 为来访客人图中任意景点相关信息的查询。

(3) 为来访客人提供图中任意景点的问路查询，即查询任意两个景点之间的一条最短路。

*/

```
#define INFINITY 65535
```

```
typedef int Patharc[9];
```

```
typedef int ShortPathTable[9];
```

```
struct InformationNode
```

```
{
```

```
    int SerialNumber;
```

```
    string Name;
```

```
    string Information;
```

```
    InformationNode()
```

```
{
```

```
}
```

```
    InformationNode(int n, string name, string information)
```

```
{
```

```
        SerialNumber = n;
```

```
        Name = name;
```

```
        Information = information;
```

```
}
```

```
};
```

```
#define DefaultVertices 20
```

```
#define maxWeight 9000000000
```

```
//邻接矩阵
```

```
template <class T, class E>
```

```
class Graph1
```

```
{
```

```
public:
```

```
    Graph1(int sz = DefaultVertices);
```

```
    ~Graph1();
```

```

//插入点
void addVertices(T v);
//插入边
void addEdge(int i, int j, E e);

//深度优先查找
void DFSTraverse();
void DFS(int i);

//最短路径
void Dijkstra(int v0, Patharc* p, ShortPathTable* D);

friend istream& operator >> (istream& in, Graph1<T, E>& G);    //输入
friend ostream& operator << (ostream& out, Graph1<T, E>& G);    //输出

//顶点表
T* VerticesList;
//邻接矩阵
E** Edge;
//访问标志数组;
int* visited;
//图中最大顶点数
int maxVertices;
//当前顶点数
int numVertices;
//当前边数
int numEdges;
//给出顶点 vertex 在图中位置
int getVertexPos(T vertex)
{
    for (int i = 0; i < numVertices; i++)
    {
        if (VerticesList[i] == vertex)
        {
            return i;
        }
    }
    return -1;
};

};

template <class T, class E>
Graph1<T, E>::Graph1(int sz)

```

```

{
    maxVertices = sz;
    numVertices = 0;
    numEdges = 0;
    visited = new int[maxVertices];
    int i, j;
    VerticesList = new T[maxVertices]; //创建顶点表
    //邻接矩阵
    Edge = (E**) new E * [maxVertices];
    for (i = 0; i < maxVertices; i++)
    {
        Edge[i] = new E[maxVertices];
    }
    //矩阵初始化
    for (i = 0; i < maxVertices; i++)
    {
        for (j = 0; j < maxVertices; j++)
        {
            Edge[i][j] = (i == j) ? 0 : maxWeight;
        }
    }
}

```

```

template <class T, class E>
Graph1<T, E>::~~Graph1()
{
    delete[] VerticesList;
    for (int i = 0; i < numVertices; i++)
    {
        delete[] Edge[i];
    }
    delete[] Edge;
}

```

```

template <class T, class E>
void Graph1<T, E>::addVertices(T v)
{
    VerticesList[numVertices] = v;
    numVertices += 1;
}

```

```

template <class T, class E>
void Graph1<T, E>::addEdge(int i, int j, E e)
{
    Edge[i - 1][j - 1] = e;
    Edge[j - 1][i - 1] = e;
    numEdges++;
}

```

```

template <class T, class E>
void Graph1<T, E>::DFS(int i)
{
    int j;
    visited[i] = 1;
    cout << VerticesList[i].Name << " ";
    for (j = 0; j < numVertices; j++)
    {
        if (Edge[i][j] == 1 && Edge[i][j] != maxWeight && !visited[j])
        {
            DFS(j);
        }
    }
}

```

```

template <class T, class E>
void Graph1<T, E>::DFS Traverse()
{
    int i;
    for (i = 0; i < numVertices; i++)
    {
        visited[i] = 0;
    }
    for (i = 0; i < numVertices; i++)
    {
        if (!visited[i])
        {
            DFS(i);
        }
    }
}

```

```

#define MAXVEX 9
//最短路径
template <class T, class E>
void Graph1<T, E>::Dijkstra(int v0, Patharc *p, ShortPathTable *D)
{
    int v, w, k = v0, min;
    int final[MAXVEX];
    for (v = 0; v < numVertices; v++)
    {
        final[v] = 0;
        (*D)[v] = Edge[v0][v];
        (*p)[v] = 0;
    }
    (*D)[v0] = 0;
    final[v0] = 1;
    //(*p)[v0] = v0;

    for (v = 1; v < numVertices; v++)
    {
        min = INFINITY;
        for (w = 0; w < numVertices; w++)
        {
            if (!final[w] && (*D)[w] < min)
            {
                k = w;
                min = (*D)[w];
            }
        }
        final[k] = 1;
        for (w = 0; w < numVertices; w++)
        {
            if (!final[w] && (min + Edge[k][w] < (*D)[w]))
            {
                (*D)[w] = min + Edge[k][w];
                (*p)[w] = k;
            }
        }
    }
    //(*p)[v0] = v0;
}

```

```

void print(Graph1<InformationNode, int>& G, int n)
{
    cout << "序号" << " " << "名称" << " " << "介绍" << endl;
    if (n == -1)
    {
        for (int i = 0; i < G.numVertices; i++)
        {
            cout << G.VerticesList[i].SerialNumber << " " <<
G.VerticesList[i].Name << " " << G.VerticesList[i].Information << endl;

        }
    }
    else
    {
        cout << G.VerticesList[n - 1].SerialNumber << " " << G.VerticesList[n -
1].Name << " " << G.VerticesList[n - 1].Information << endl;
    }

    //cout << G.VerticesList[i].SerialNumber << " " << G.VerticesList[i].Name << "
" << G.VerticesList[i].Information << endl;
}

void Find(Graph1<InformationNode, int> G, int i)
{
    cout << "序号" << " " << "名称" << " " << "介绍" << endl;
    string name = G.VerticesList[i - 1].Name;
    string information = G.VerticesList[i - 1].Information;
    int num = G.VerticesList[i - 1].SerialNumber;
    cout << num << " " << name << " " << information << endl;
    //cout << G->VerticesList[i].SerialNumber << " " << G->VerticesList[i].Name
<< " " << G->VerticesList[i].Information << endl;

}

template <class T, class E>
void Print(Graph1<T, E>& G)
{
    int i, j;
    cout << "节点个数" << G.numVertices << "边个数是: " << G.numEdges <<
"\n";
}

```

```

    cout << "节点为: " << "\n";
    for (i = 0; i < G.numVertices; i++)
    {
        cout << G.VerticesList[i].Name << " ";
    }
    cout << "\n";
    cout << "邻接矩阵为: " << "\n";
    for (i = 0; i < G.numVertices; i++)
    {
        for (j = 0; j < G.numVertices; j++)
        {
            cout << G.Edge[i][j] << " ";
        }
        cout << "\n";
    }
}

string path(Graph1<InformationNode, int>& G, int m1, int m2, Patharc* p,
ShortPathTable* D)
{
    string strpath;
    int i = m2;
    strpath += G.VerticesList[m2].Name;
    strpath += ">-";
    while (G.VerticesList[m1].Name != G.VerticesList[(p)[i]].Name)
    {
        strpath += G.VerticesList[(p)[i]].Name;
        strpath += ">-";
        i = G.VerticesList[(p)[i]].SerialNumber - 1;
    }
    strpath += G.VerticesList[(p)[i]].Name;
    string path(strpath.rbegin(), strpath.rend());
    return path;
}

int main()
{
    int n;
    cout << "shuahc" << endl;
    Graph1<InformationNode, int> G;
    cout << "请输入图的信息: " << endl;
    cout << "输入的节点数: ";
    cin >> n;
    for (int i = 0; i < n; i++)

```



```

{
    InformationNode node;
    string name, information;
    node.SerialNumber = i + 1;
    cin >> node.Name >> node.Information;
    G.addVertices(node);
}
cout << "输入的边数: ";
cin >> n;
for (int i = 0; i < n; i++)
{
    int a, b, w;
    cin >> a >> b >> w;
    G.addEdge(a, b, w);
}
Print(G);
int d[9];
int m[9];
Patharc* p = &m;
ShortPathTable* D = &d;
string strpath;

while (true)
{
    cout << "*****\n*****\n" << endl;

    print(G, -1);
    cout << "[1]查询信息" << " " << "[2]查询路径" << "[3]退出" << endl;
    cin >> n;
    switch (n)
    {
    case 1:
        int m;
        cout << "请输入查找的序号: ";
        cin >> m;
        print(G, m);
        break;
    case 2:
        int m1, m2;
        cout << "请输入出发的序号和到达的序号: ";
        cin >> m1 >> m2;
        cout << "Dijkstra 最短路径, " << G.VerticesList[m1 - 1].Name << "
出发: " << endl;
    }
}

```

```

        G.Dijkstra(m1 - 1, p, D);
        for (int i = 0; i < G.numVertices; i++)
        {
            cout << i + 1 << " " << G.VerticesList[i].Name << " " <<
G.VerticesList[*p][i].Name << " " << (*D)[i] << endl;
        }
        strpath = path(G, m1 - 1, m2 - 1, p, D);
        cout << "路径为: " << strpath << endl;
        break;
    default:
        exit(1);
        break;
    }
}

return 0;
}

```

/*

*

示例

9

A ThiSisA

B ThiSisB

C ThiSisC

D ThiSisD

E ThiSisE

F ThiSisF

H ThiSisH

I ThiSisI

G ThiSisG

15

1 2 3

1 6 4

2 9 6

6 9 4

6 5 9

5 7 7

5 4 5

4 7 3

4 9 4

9 7 6

2 8 5

8 4 3

8 3 6

3 4 4

2 3 6

9

A ThiSisA

B ThiSisB

C ThiSisC

D ThiSisD

E ThiSisE

F ThiSisF

H ThiSisH

I ThiSisI

G ThiSisG

16

1 2 1

1 3 5

2 3 3

2 4 7

2 5 5

3 5 1

3 6 7

4 5 2

4 7 3

5 6 3

5 7 6

5 8 9

6 7 5

7 8 2

7 9 7

8 9 4

*/