#include<iostream>

#include<stack>

#include<queue>

#include<cstring>

#include<utility>

#include<sstream>

#include<vector>

#include<algorithm>

using namespace std;

const int MAX\_NUM = 2147483647;

const int MIN\_NUM = -2147483648;

template <class TypeOfVer, class TypeOfEdge>

class adjmatrix\_graph {

private:

int vers; //顶点数

int edges; //边数

TypeOfEdge\*\* edge; //存放邻接矩阵（TypeOfEdge表示顶点关系类型。对于无权图，用1或0，表示相邻否；对于带权图，则为权值类型）

TypeOfVer\* ver; //存放结点值，注意结点得位序和结点的值是两回事，结点位序其类型必为int且其合理范围为：0到vers-1，而结点的值则可以是任意类型，并没有什么合理范围

TypeOfEdge noEdge; //邻接矩阵中的∞的表示值（注意邻接矩阵中的∞可以根据不同的情境自由设置）

string graphKind; //图的种类标志

bool DFS(int u, int& num, int visited[])const; //DFS遍历（递归部分）

public:

//一、3个构造函数：

//注意：DG（有向图）, DN（有向网）, UDG（无向图）, UDN（无向网）

//1、构造函数构造一个只有结点没有边的图。4个参数的含义：图的类型、结点数、结点值和邻接矩阵中表示结点间没有边的标记（无权图：0，有权图：输入参数定）

adjmatrix\_graph<TypeOfVer, TypeOfEdge>(const string& kd, const int& vSize, const TypeOfVer d[], const TypeOfEdge noEdgeFlag);

//2、构造函数构造一个无权图。5个参数的含义：图的类型、结点数、边数、结点集和边集

adjmatrix\_graph<TypeOfVer, TypeOfEdge>(const string& kd, const int& vSize, const int& eSize, const TypeOfVer d[], int\*\* e);

//3、构造函数构造一个有权图。7个参数的含义：图的类型、结点数、边数、无边标记、结点集、边集、权集

adjmatrix\_graph<TypeOfVer, TypeOfEdge>(const string& kd, const int& vSize, const int& eSize, const TypeOfEdge noEdgeFlag, const TypeOfVer d[], int\*\* e, const TypeOfEdge w[]);

//4、析构函数

~adjmatrix\_graph();

//两个拷贝赋值操作

//5、拷贝构造函数：

adjmatrix\_graph<TypeOfVer, TypeOfEdge>(const adjmatrix\_graph<TypeOfVer, TypeOfEdge>& G);

//6、拷贝赋值运算符：

adjmatrix\_graph<TypeOfVer, TypeOfEdge>& operator=(const adjmatrix\_graph<TypeOfVer, TypeOfEdge>& G);

//二、判断型操作：

//1、判断图空否（注意只有结点空了一个图才算空了）

bool isEmpty()const { return vers == 0; }

//2、检查两个输入的位序（0到vers-1）所对应的顶点是否为邻接顶点（即两个顶点之间是否至少存在一条边）

bool existEdge(const int& u, const int& v)const;

//三、提取信息型操作：

//1、取得该图的类型

string getGraphKind()const { return graphKind; }

//2、取得当前顶点数

int getVerNum()const { return vers; }

//3、 取得当前边数

int getEdgeNum()const { return edges; }

//4、获取指向结点数组首地址的指针

TypeOfVer\* getVerPointer()const { return ver; }

//5、取得G中指定顶点的值，其中u为序号（从1到vers），将对应结点的值放到data中

bool getVer(int u, TypeOfVer& data)const;

//6、返回G中指定顶点的位置

int locateVer(const TypeOfVer& data)const;

//7、返回G中指定顶点u的第一个邻接顶点的位序（顶点集）。若顶点在G中没有邻接顶点，则返回-1，即若u在允许的范围内

//则检查的是第u行是否有边，若有边，则返回其所找到的第一条边（在邻接矩阵中）

int getFirstAdjVex(const int& u)const;

//8、返回G中指定顶点u的下一个邻接顶点（相对于v）的位序（顶点集）。若顶点在G中没有邻接顶点，则返回-1，本函数要做的就是

//对于在第u行指定的位序v（u、v均在0到vers-1），看是否存在位序在u之后的边（即对应的邻接矩阵的点不为noEdge）

//如果有，则返回遇到的第一个邻接点所对应的位序（0到vers-1）

int getNextAdjVex(const int& u,const int& v)const;

//9、获取有向图的指定顶点位序所对应的顶点的入度，如果图为空或该顶点不存在或不是有向图，则返回-1

//注意：只有对于有向图来说，才有入度，所以不是有向图的一律不考虑入度

int getInDegree(const int& u)const;

//10、获取有向图的指定顶点位序所对应的顶点的出度，如果图为空或该顶点不存在，则返回-1

//注意：这个图统计的出度不分有向图还是无向图，它实际上统计的是邻接矩阵中对应的第u行非noEdge元素的个数

int getOutDegree(const int& u)const;

//11、统计任意一种图的指定顶点位序所对应的顶点的度，如果图为空或该顶点不存，则返回-1

//注意：这个函数统计的是第u行和第u列非noEdge元素的个数，因为我们这里的顶点不会存在一条指向顶点自身的边，所以

//不存在重复统计的情况

int getDegree(const int& u)const;

//四、设置型操作：

//1、对G中指定顶点赋值

bool setVer(const int& u, const TypeOfVer& data);

//五、插入、删除型操作：

//（一）针对结点：

//1、往G中添加一个顶点

//实现策略：重新申请一块所需要的二维数组的空间，然后整体搬家，再将原来的空间释放掉

bool insertVer(const TypeOfVer& data);

//2、往G中删除一个顶点

//实现策略：元素移动，修改vers值和edges值，移动修改完后即删除完成，而不单独删除所多余出来的一部分空间

bool deleteVer(const TypeOfVer& data);

//（二）针对边：

//注意：以下针对边的操作只是针对邻接矩阵的元素的值和边数edges进行变化，而不对矩阵edge的规模有任何影响

//1、无权图插入一条边

bool insertEdge(int u, int v);

//2、有权图插入一条边

bool insertEdge(int u, int v, TypeOfEdge w);

//3、无权图删除一条边

bool deleteEdge(int u, int v);

//4、有权图删除一条边

bool deleteEdgeNet(int u, int v);

//六、展示遍历型操作：

//1、输出邻接矩阵，如果邻接矩阵为空，则会返回一个false，并且没有输出的必要

bool printMatrix()const;

//2、DFS遍历（外壳部分）

void DFS\_Traverse(int u)const;

//3、BFS遍历

void BFS\_Traverse(int u)const;

//4、图的相关信息展示函数（测试时用，含有中文等提示信息，让你能比较清晰地看清楚所建立地整个图的信息，方便检查对应的bug）

void show()const;

//七、一些高级操作：

//1、检查两个结点之间是否有路径存在（外壳部分，公有成员函数）

bool checkRoute(int u, int v)const;

//检查两个结点之间是否有路径存在（递归部分，私有成员函数）

bool checkRoute(int u, int v, int visited[])const;

//2、拓扑排序（仅提供拓扑排序是否成功）

//注意当进入拓扑排序的顶点个数小于实际的总顶点个数，函数要返回 false，其他具体运行规则请参照具体函数

//重载形式1

bool topSort()const;

//拓扑排序（不仅提供拓扑排序是否成功，排序序列存储在引用参数topsort[]中）

//重载形式2

bool topSort(int& num, int topsort[])const;

//3、判断图是否为回路（注意：有回路时返回true，无回路时返回false，千万不要搞混了）

//有向图回路判断

bool isDAG()const;

//无向图回路判断：

bool U\_Judge\_Cir()const;

};

bool cmp2(const pair<pair<int, int>, double >& a, const pair<pair<int, int>, double>& b) {//之前还写过一个排序的规则cmp，但是觉得它没有什么用，就又写了一个，就这么命名吧，懒得改了

if (a.first.first < b.first.first) {

return true;

}

else if (a.first.first == b.first.first) {

if (a.first.second < b.first.second) {

return true;

}

else return false;

}

else {

return false;

}

}

//该函数的目的就是要对weight数组进行重新排列，对带权图的创建做下铺垫

int\* sort\_weight(const string& kd, pair<int, int>edge[], const int& edges, int w[]) {

//参数列表从左到右的含义分别为：边序列数组、边的个数、每个顶点对应的权值

if (kd[0] == 'U') {

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

if (edge[i].first > edge[i].second) {

int tmp = edge[i].first;

edge[i].first = edge[i].second;

edge[i].second = tmp;

}

}

}

pair<pair<int, int>, int>\* p = new pair<pair<int, int>, int>[edges];

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

p[i].first.first = edge[i].first;

p[i].first.second = edge[i].second;

p[i].second = w[i];

}

/\*cout << "sort排序前的情况：" << endl;

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

cout << p[i].first.first << ' ' << p[i].first.second << ' ' << p[i].second << endl;

}\*/

sort(p, p + edges, cmp2);

/\*cout << "sort排完序后的情况：" << endl;

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

cout << p[i].first.first << ' ' << p[i].first.second << ' ' << p[i].second << endl;

}\*/

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

w[i] = p[i].second;

}

return w;

}//暂时还不确定有没有问题

//将输入的边的序列转化为一个整数数组，以便于之后的图的创建

//规则：对于有向图而言，如果两个序列完全相同（大小一样且顺序相同），则表明有两条边，要在对应的值中体现

//对于无向图而言，若两个序列相同（大小一样但方向可以不一致，则表明这之间有两条边，要在对应的值中有所体现）

int\*\* transform\_sequence\_to\_array(const string& kd, pair<int, int>edge[], const int& edges, const int& vers) {

//参数列表从左到右的含义分别为：矩阵类型、边的集合、以及边的个数、顶点个数（用于创建邻接矩阵）

int\*\* e = new int\* [vers];

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

e[i] = new int[vers];

}

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

for (int j = 0; j < vers; ++j) {

e[i][j] = 0;

}

}

for (int i = 0; i < edges; ++i) {//传进这个函数的pair数组一定是一个已经排好顺序的数组，这么做有利于接下来的操作

e[edge[i].first][edge[i].second] += 1;

if (kd == "UDG" || kd == "UDN") {//如果准备构建无向图或无向网

e[edge[i].second][edge[i].first] += 1;

}//构建有向图时则不用进行这一步

}

return e;

}

//以上三个函数都是辅助创建还有边的图的函数，可以应对诸如在输入相应边的 时候不考虑对应结点按顺序输入，但是也能保证创建出正确的图和网（带权）

//一、构造函数：

//1、构造函数构造一个只有结点没有边的图。4个参数的含义：图的类型、结点数、结点值和邻接矩阵中表示结点间没有边的标记（无权图：0，有权图：输入参数定）

template<class TypeOfVer, class TypeOfEdge>

adjmatrix\_graph<TypeOfVer, TypeOfEdge>::adjmatrix\_graph(const string& kd, const int& vSize, const TypeOfVer d[], const TypeOfEdge noEdgeFlag) {

graphKind = kd;

vers = vSize;

ver = new TypeOfVer[vers];

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

ver[i] = d[i];//对结点的值数组进行赋值

}

noEdge = noEdgeFlag;//两个结点之间没有边的标记，可以在该构造函数中自行设定

edges = 0;

edge = new TypeOfEdge \* [vers];

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

edge[i] = new TypeOfEdge[vers];

}//创建一个动态的二维数组

//因为其没有边，所以edge矩阵应该全部赋值为0（在一般情况下，如果遇到特殊要求的情况，应该回来修改这个构造函数）

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

for (int j = 0; j < vers; ++j) {

edge[i][j] = noEdge;//应该将其全部赋值为noEdge，不过这里也提示我们TypeOfEdge应该设为int或double等数字类型的，不过还要看具体情况

}

}

}

//2、构造函数构造一个无权图。5个参数的含义：图的类型、结点数、边数、结点集和边集

template<class TypeOfVer, class TypeOfEdge>

adjmatrix\_graph<TypeOfVer, TypeOfEdge>::adjmatrix\_graph(const string& kd, const int& vSize,

const int& eSize, const TypeOfVer d[], int\*\* e) {

graphKind = kd;//此处虽然是无权图，但要注意无向还是有向

vers = vSize;

ver = new TypeOfVer[vers];

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

ver[i] = d[i];

}

edges = eSize;

edge = new TypeOfEdge \* [vers];

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

edge[i] = new TypeOfEdge[vers];

}

for (int i = 0; i < vers; ++i) {//注意此处输入的边应该是一对顶点序列集，我们要在main函数中将其转化为二维整数数组

for (int j = 0; j < vers; ++j) {//转化之后再进行无权图的构建

edge[i][j] = e[i][j];//传进来的e[i][j]会自动标识好每个位置的值，因此这里不需要我们操心区分情况

}

}

//但是这里有一个疑问，noEdge该设置为什么，应该也是要根据具体的值来看的，还有要结合oj的测试用例

//这里我们暂时把noEdge设置为TypeOfEdge(0)，但是不确定正确。

noEdge = (TypeOfEdge)(0);

}//暂时认为这个构造函数没有什么问题

//3、构造函数构造一个有权图。7个参数的含义：图的类型、结点数、边数、无边标记、结点集、边集、权集

template<class TypeOfVer, class TypeOfEdge>

adjmatrix\_graph<TypeOfVer, TypeOfEdge>::adjmatrix\_graph(const string& kd, const int& vSize,

const int& eSize, const TypeOfEdge noEdgeFlag, const TypeOfVer d[], int\*\* e, const TypeOfEdge w[]) {

graphKind = kd;//图的类型

vers = vSize;//图的结点数（顶点数）

edges = eSize;//图的边数

noEdge = noEdgeFlag;//图的无边标记

ver = new TypeOfVer[vers];

for (int i = 0; i < vers; ++i) {//对结点数组进行赋值

ver[i] = d[i];

}

edge = new TypeOfEdge \* [vers];

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

edge[i] = new TypeOfEdge[vers];

}

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

for (int j = 0; j < vers; ++j) {

if (e[i][j] == 0) {//这里说明两点之间没有边

edge[i][j] = noEdge;

}

else {

edge[i][j] = e[i][j];//注意这里只是表示构造出了表示两点间有没有边的矩阵，但是对于这个边的权，我们这里并没有进行处理

}

}//对权值的处理在下面

}

//我们可以知道只有有边的地方才有权值，因此权数组应该与边数组成一一对应关系

//下面对邻接矩阵网里的各个边进行赋权，但是注意在给定关于边的数对的时候不一定是按照第一个数从小到大排列给的，无疑这是

//一个比较棘手的问题，这提示我在进行关于含权的图创建的时候一定要对其进行排序

if (graphKind == "DN") {//构建的是有向带权图

int k = 0;//这是用于来标记w数组下标的变量

for (int i = 0; i < vers; ++i) {//进行到这一步的权值数组（w数组）一定是已经进行过重新排序的，这样接下来的操作才不会错

for (int j = 0; j < vers; ++j) {

if (edge[i][j] != noEdge) {//注意：目前认为：如果要加权值的话，两个结点之间只能由一条边，如果对于有向图来说

edge[i][j] = w[k];//一个结点到另一个结点最多有一条边

++k;

}

}

}

}

else {//构建的无向带权图

bool\*\* flag = new bool\* [vers];

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

flag[i] = new bool[vers];

}

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

for (int j = 0; j < vers; ++j) {

flag[i][j] = false;

}

}

int k = 0;

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

for (int j = 0; j < vers; ++j) {

if (edge[i][j] != noEdge && flag[i][j] == false) {

edge[i][j] = w[k];

flag[i][j] = true;

edge[j][i] = w[k];

flag[j][i] = true;

++k;

}

}

}

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

delete[] flag[i];

}

delete[] flag;//一定要记得在对应的函数中释放本函数所动态申请的数组空间

}

}//暂时认为这个构造函数没有什么问题

//4、析构函数

template<class TypeOfVer, class TypeOfEdge>

adjmatrix\_graph<TypeOfVer, TypeOfEdge>::~adjmatrix\_graph() {//主要是显式地释放掉动态申请得到的存储空间，即一维数组ver和二维数组edge

delete[]ver;

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

delete[]edge[i];

}

delete[] edge;

//cout << "调用了析构函数";//用于测试的语句

}//暂时经过测试不认为有什么问题

//两个拷贝赋值操作

//5、拷贝构造函数：

template<class TypeOfVer, class TypeOfEdge>

adjmatrix\_graph<TypeOfVer, TypeOfEdge>::adjmatrix\_graph(const adjmatrix\_graph<TypeOfVer, TypeOfEdge>& G) {

graphKind = G.graphKind;

vers = G.vers;

edges = G.edges;

noEdge = G.noEdge;

ver = new TypeOfVer[vers];

if (G.isEmpty()) {//如果G空，则不继续往下拷贝

return;

}

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

ver[i] = G.ver[i];

}

edge = new TypeOfEdge \* [vers];

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

edge[i] = new TypeOfEdge[vers];

}

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

for (int j = 0; j < vers; ++j) {

edge[i][j] = G.edge[i][j];

}

}

}

//6、拷贝赋值运算符：

template<class TypeOfVer, class TypeOfEdge>

adjmatrix\_graph<TypeOfVer, TypeOfEdge>& adjmatrix\_graph<TypeOfVer, TypeOfEdge>::operator=

(const adjmatrix\_graph<TypeOfVer, TypeOfEdge>& G) {

graphKind = G.graphKind;

vers = G.vers;

edges = G.edges;

noEdge = G.noEdge;

ver = new TypeOfVer[vers];

if (G.isEmpty()) {

return \*this;

}

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

ver[i] = G.ver[i];

}

edge = new TypeOfEdge \* [vers];

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

edge[i] = new TypeOfEdge[vers];

}

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

for (int j = 0; j < vers; ++j) {

edge[i][j] = G.edge[i][j];

}

}

return \*this;

}

//二、判断型操作：

//2、检查两个输入的位序（0到vers-1）所对应的顶点是否为邻接顶点（即两个顶点之间是否至少存在一条边）

template<class TypeOfVer,class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::existEdge(const int& u, const int& v)const {

if (this->isEmpty() || vers == 1) {//图为空或仅有一个顶点

return false;

}

else {

if (u < 0 || u >= vers || v < 0 || v >= vers) {//u、v不在允许的范围内

return false;

}

else {

if (edge[u][v] != noEdge || edge[v][u] != noEdge) {

return true;

}

else {

return false;

}

}

}

}

//三、提取信息型操作：

//4、取得G中指定顶点的值

//具体做法为指定结点的标号（合理的标号从1到vers，不合理的标号不在这个范文内）

//如果图为空或传进来的标号u不合理，则给data赋值为0，并返回false；

//如果图不为空且传进来的标号合理，则给data赋值为this->ver[u-1]，并返回true

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::getVer(int u, TypeOfVer& data)const {

if (this->isEmpty() || u<1 || u>vers) {//图为空图或传进来的标号不合理时，就给data赋值为0

data = (TypeOfVer)0;

return false;

}

else {

data = this->ver[u - 1];

return true;

}

}//暂时认为它没有问题

//5、返回G中指定顶点的位置

//注意：返回的这个代表位置的整数的取值范围一定是1到vers（这个函数一定要注意，这是我开始时写的函数，因为没有看要求）

//所以写成了这样，再改的话又不止改一处，所以使用的时候一定要注意

//如果没有找到这个值所对应的结点，那么就返回0

template<class TypeOfVer, class TypeOfEdge>

int adjmatrix\_graph<TypeOfVer, TypeOfEdge>::locateVer(const TypeOfVer& data)const {

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

if (ver[i] == data) {

return i + 1;

}

}

return 0;//表明没有找到对应的结点

}//经测试暂时认为没有问题

//6、返回G中指定顶点u的第一个邻接顶点的位序（顶点集）。若顶点在G中没有邻接顶点，则返回-1

//注意：传进来的u是位序（其合理范围为0到vers-1，其中vers为图的结点的总数）

template<class TypeOfVer, class TypeOfEdge>

int adjmatrix\_graph<TypeOfVer, TypeOfEdge>::getFirstAdjVex(const int& u)const {

if (this->isEmpty() || vers == 1) {//图为空或仅有一个结点

return -1;

}

else {

if (u < 0 || u >= vers) {//传入的u不在合理的范围内

return -1;

}

else {

int i, j;

i = u;

int tmp1 = -1, tmp2 = -1;

for (j = 0; j < vers; ++j) {

if (edge[i][j] != noEdge) {

tmp1 = j;

break;

}

}

return tmp1;

}

}

}//应该没有什么大问题，没有好好详细测试，待测

//7、返回G中指定顶点u的下一个邻接顶点（相对于v）的位序（顶点集）。若顶点在G中没有邻接顶点，则返回-1

template<class TypeOfVer, class TypeOfEdge>

int adjmatrix\_graph<TypeOfVer, TypeOfEdge>::getNextAdjVex(const int& u,const int& v)const {

if (this->isEmpty() || vers == 1) {

return -1;

}

else {

if (u < 0 || u >= vers||v<0||v>=vers-1) {

return -1;

}

else {

for (int i = v+1; i < vers; ++i) {

if (edge[u][i] != noEdge) {

return i;

}

}

return -1;

}

}

}//暂时也没有测试这个函数

//8、获取有向图的指定顶点位序所对应的顶点的入度，如果图为空或该顶点不存在或是不是有向图，则返回-1

template<class TypeOfVer,class TypeOfEdge>

int adjmatrix\_graph<TypeOfVer, TypeOfEdge>::getInDegree(const int& u)const {

if (this->isEmpty()||graphKind[0]=='U') {//图为空或者不是有向图

return -1;

}

else {

if (u < 0 || u >= vers) {//u不在允许的范围内（0到vers-1）

return -1;

}

else {

int count = 0;

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

if (edge[i][u] != noEdge) {

++count;

}

}

return count;

}

}

}

//9、获取图的指定顶点位序所对应的顶点的出度，如果图为空或该顶点不存在，则返回-1

template<class TypeOfVer,class TypeOfEdge>

int adjmatrix\_graph<TypeOfVer, TypeOfEdge>::getOutDegree(const int& u)const {

if (this->isEmpty()) {//图为空或者不是有向图

return -1;

}

else {

if (u < 0 || u >= vers) {//u不在允许的范围内（0到vers-1）

return -1;

}

else {

int count = 0;

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

if (edge[u][i] != noEdge) {

++count;

}

}

return count;

}

}

}

//10、统计任意一种图的指定顶点位序所对应的顶点的度，如果图为空或该顶点不存在或是不是有向图，则返回-1

template<class TypeOfVer,class TypeOfEdge>

int adjmatrix\_graph<TypeOfVer,TypeOfEdge>::getDegree(const int& u)const {

if (this->isEmpty()) {//图为空，则必返回-1

return -1;

}

else {

if(u<0||u>=vers){//u不在合理的范围内（0到vers-1）

return -1;

}

else {

int count = 0;

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

if (edge[u][i] != noEdge) {

++count;

}

}

if (graphKind[0] == 'U') {//无向图不需要区分入度和出度

return 2\*count;

}

else {

for (int i = 0; i < vers; ++i) {//用于统计有向图的指定结点的入度

if (edge[i][u] != noEdge) {

++count;

}

}

return count;

}

}

}

}

//四、设置型操作：

//1、对G中指定顶点赋值（注意改变的是结点的值而非邻接矩阵的值，即改变的是ver数组的某一位置上的值）

//u为结点数组中的位序（从1到vers）

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::setVer(const int& u, const TypeOfVer& data) {

if (this->isEmpty() || u<1 || u>vers) {//该图为空或者位序u不在合理范围内

return false;

}

else {

ver[u - 1] = data;

return true;

}

}//暂时认为这个没有问题

//五、插入删除类操作：

//（一）对于结点：

//1、往G中添加一个顶点

//注意：添加的顶点的位序位于顶点的末端

//要将原有的列表的其他部分不要变

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::insertVer(const TypeOfVer& data) {

++vers;

TypeOfVer\* nv = new TypeOfVer[vers];//重新申请一块暂时保存新的结点的值的一维数组空间

TypeOfEdge\*\* ne = new TypeOfEdge \* [vers];

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

ne[i] = new TypeOfEdge[vers];

}//重新申请一块暂时保存新的边的邻接矩阵的二维数组

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

if (i == vers - 1) {

nv[i] = data;

}

else nv[i] = ver[i];

}

//要释放原来的空间：

if (vers != 1)

delete[] ver;

ver = nv;

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

for (int j = 0; j < vers; ++j) {

if (i < vers - 1 && j < vers - 1) {

ne[i][j] = edge[i][j];

}

else {

ne[i][j] = noEdge;

}

}

}

//释放掉原来的空间：

if (vers != 1) {

for (int i = 0; i < vers - 1; ++i) {

delete[] edge[i];

}

delete[] edge;

}

edge = ne;

return true;

}//目前认为这个函数正确（注意：目前没有想到为什么要把这个函数定义为bool，不是bool类型的也行，我没有找打定义为bool的理由，而且只要调用这个函数）

//一定会返回一个true，不过这么定义也无妨，不想改了

//删除G中的一个顶点

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::deleteVer(const TypeOfVer& data) {

if (this->isEmpty()) {//表明这个图为空

return false;

}

else {

int place = locateVer(data);//如果找到place在1到vers之间，否则place==0

if (place == 0) {//表明没有找到对应的结点

return false;

}

else {//表明找到了对应的结点，下面进行删除操作

//所谓删除一个结点，不光要使ver数组发生变化（vers必减1），还要使得edge数组发生变化，进而使得edges（边的数目）可能发生变化

--vers;//顶点个数-1

int count = edges;//暂时保存删除前的边数

if (place != vers + 1) {//如果place==vers+1，即最理想的情况，根本不用进行覆盖操作，直接--vers即可，就认为其删除完成，

//不过只是表明可以不用再对矩阵里的元素进行移动，但是为了正确地表示删除顶点之后可能引起的边的数目edges的变化，因此需要对

//最外面那一圈进行遍历

//ver数组的变化：

for (int i = place - 1; i < vers; ++i) {

ver[i] = ver[i + 1];//进行后面的值对前面的值覆盖式删除

}

//二维数组edge的变化

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

if (i < place - 1) {

for (int j = place - 1; j < vers; ++j) {

if (j == place - 1 && edge[i][j] != noEdge) {

--edges;//一定要对边的条数及时的修正，并且是先修正再覆盖

}

edge[i][j] = edge[i][j + 1];

}

//int\* tmp = (edge[i] + vers);

//delete tmp ;

}

else {

for (int j = 0; j < vers + 1; ++j) {

if (i == place - 1 && edge[i][j] != noEdge) {

--edges;//一定要对边的条数及时的修正，并且是先修正再覆盖

}

edge[i][j] = edge[i + 1][j];

}

/\*if (i == vers - 1) {

int\*\* tmp2 = edge + vers;

delete[] tmp2;

}\*/

for (int j = place - 1; j < vers; ++j) {

if (j == place - 1 && edge[i][j] != noEdge) {

--edges;//一定要对边的条数及时的修正，并且是先修正再覆盖

}

edge[i][j] = edge[i][j + 1];

}

//int\* tmp = edge[i] + vers;

//delete tmp;

}

}

if (graphKind[0] == 'U') {//表明图是无向图，注意此时很明显edges多减去了实际的减少边数的两倍

edges = count - (count - edges) / 2;

}//如果是有向图，则不会有上述现象

}

else {//如果place==vers+1（实际要删除的点为最后一个结点），虽然不用处理邻接矩阵的情况，但是需要处理edges的变化，因此要遍历一下数组

//查看当行数i=vers+1和列数j=vers+1的情况：

int i, j;

i = vers;//未删除前的最后一行

for (j = 0; j < vers + 1; ++j) {

if (edge[i][j] != noEdge) {

--edges;

}

}

j = vers;//未删除前的最后一列

for (i = 0; i < vers; ++i) {

if (edge[i][j] != noEdge) {

--edges;

}

}

if (graphKind[0] == 'U') {

edges = count - (count - edges) / 2;

}

}

return true;

}

}

}//暂时没有发现什么问题

//（二）针对边的插入和删除：

//1、无权图插入一条边

//注意：传进来的参数u、v指的是顶点的序号，而非顶点的值

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::insertEdge(int u, int v) {

if (this->isEmpty()) {

return false;

}

else {

if (u < 0 || u >= vers || v < 0 || v >= vers) {//u、v是表示边的序列（其合理范围为：0到vers-1）

return false;

}

else {

if (graphKind == "UDG") {//无向无权图

if (edge[u][v] != noEdge) {

return false;

/\*edge[u][v] += 1;

edge[v][u] += 1;\*/

}

else {

++edges;//插入边的时候，对应的边的条数一定要有变化

edge[u][v] = 1;

edge[v][u] = 1;

}

}

else {//有向无权图，注意：在有向图中，u表示起点，v表示终点

if (edge[u][v] != noEdge) {

return false;

//edge[u][v] += 1;

}

else {

++edges;//插入边的时候，对应的边的条数一定要有变化

edge[u][v] = 1;

}

}

return true;

}

}

}//目前认为没有问题

//2、有权图插入一条边

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::insertEdge(int u, int v, TypeOfEdge w) {

if (this->isEmpty()) {

return false;

}

else {

if (u < 0 || u >= vers || v < 0 || v >= vers) {

return false;

}

else {

if (graphKind == "UDN") {//无向有权网

if (edge[u][v] != noEdge) {

return false;

}

else {

++edges;

edge[u][v] = w;

edge[v][u] = w;

}

}

else {//有向有权图，注意：在有权图中，u表示起点，v表示终点

if (edge[u][v] != noEdge) {

return false;

}

else {

++edges;

edge[u][v] = w;

}

}

return true;

}

}

}//暂时认为这个函数没有什么问题

//3、无权图删除一条边

//注意：仍然要区分无向和有向

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::deleteEdge(int u, int v) {

//若对于有向图来说：传进来的参数u、v则为从u到v的边

if (this->isEmpty() || vers == 1) {//图为空或者只有一个结点，没有进行这个操作的必要

return false;

}

else {

if (u < 0 || u >= vers || v < 0 || v >= vers) {//u、v不在合理的范围内

return false;

}

else if (edge[u][v] == noEdge) {//u、v间不存在边（也包括了当为有向边时的情况），没有必要进行删除边的操作

return false;

}

else {//只有这个时候才有必要进行删除操作，注意：因为这是无权图的删除函数，所以邻接矩阵的值不是代表权值，直接减一即可，同时也要注意给边数edges-1

--edge[u][v];

--edges;

if (graphKind[0] == 'U') {//表明是无向图删除边，还应该使邻接矩阵再变一下，有向图就不必进行这个操作

--edge[v][u];

}

}

return true;

}

}//暂时认为没有什么问题

//4、有权图删除一条边

//注意：仍然要区分无向和有向

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::deleteEdgeNet(int u, int v) {

if (this->isEmpty()) {

return false;

}

else {

if (u < 0 || u >= vers || v < 0 || v >= vers) {

return false;

}

else if (edge[u][v] == noEdge) {

return false;

}

else {

edge[u][v] = noEdge;

--edges;

if (graphKind[0] == 'U') {//无向图上必须有这一步操作

edge[v][u] = noEdge;

}

return true;

}

}

}//经过测试暂时没有发现什么问题

//六、展示遍历类操作：

//1、输出邻接矩阵

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::printMatrix()const {

if (this->isEmpty()) {//如果这个图为空，那么就返回一个false

return false;

}

else {//这个图为非空，可以输出相应的邻接矩阵

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

for (int j = 0; j < vers; ++j) {

cout << edge[i][j] << ' ';

}

if (i < vers - 1) cout << endl;

}

return true;

}

}//经测试暂时没有问题

//2、DFS遍历（外壳部分）

//注意：传进来的u是遍历开始的起点的位序（合理范围：0到vers-1）（必须规定一个结点，不让无法下手开始遍历），而且要想成功遍历到每一个元素，这个图必须为连通图

//此处即默认待遍历的都是连通图

template<class TypeOfVer, class TypeOfEdge>

void adjmatrix\_graph<TypeOfVer, TypeOfEdge>::DFS\_Traverse(int u)const {

if (this->isEmpty()) {

return;

}

else {

if (u < 0 || u >= vers) {

return;

}

else {

int\* visited = new int[vers];//用于标记这个结点有没有被访问，为0则是未被访问，为1则是已被访问

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

visited[i] = 0;//初始化

}

int num = 0;//num用于标记已经访问了的结点个数

DFS(u, num, visited);

delete[]visited;//一定要记得显式释放动态申请的内存

}

}

}

//DFS遍历（递归部分）

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::DFS(int u, int& num, int visited[])const {

if (visited[u] == 0) {//表明这个结点没有被访问过

if (num != 0) {

cout << "->";

}

cout << ver[u];

visited[u] = 1;

++num;

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

if (edge[u][i] != noEdge) {

DFS(i, num, visited);

}

}

return true;

}

else {

return false;

}

}//DFS\_Traverse经测试暂时认为没有问题

//3、BFS遍历

template<class TypeOfVer, class TypeOfEdge>

void adjmatrix\_graph<TypeOfVer, TypeOfEdge>::BFS\_Traverse(int u)const {

if (this->isEmpty()) {//图为空，没有遍历的必要

return;

}

else {

if (u < 0 || u >= vers) {//图的下标不在合理的范围内，没有遍历的必要

return;

}

else {

int\* visited = new int[vers];//用于标记这个结点是否已经访问过，为0则还没有被访问，为1则已经被访问

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

visited[i] = 0;

}

int num = 0;

queue<int>q;//存放顶点位序的队列（位序范围：0到vers-1）

q.push(u);

visited[u] = 1;

++num;

while (!q.empty()) {

int ing = q.front();//ing表示当下正在处理的结点的位序

q.pop();

if (num < vers) {

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

if (edge[ing][i] != noEdge && visited[i] == 0) {//有邻接结点且该结点未被访问

q.push(i);

visited[i] = 1;

++num;

}

}

}

cout << ver[ing];

if (!q.empty()) {

cout << "->";

}

}

delete[]visited;//一定要记得显式释放动态申请的内存

}

}

}//暂时认为没有什么问题

//4、图的信息展示函数（内含相关中文辅助加以展示图的相关信息）

template<class TypeOfVer, class TypeOfEdge>

void adjmatrix\_graph<TypeOfVer, TypeOfEdge>::show() const {

cout << "输出这个图的相关信息：" << endl;

cout << "这个矩阵的类型为：" << this->graphKind << endl;

cout << "这个矩阵的两顶点没有边的标志为：" << noEdge << endl;

cout << "顶点个数为：" << vers << endl;

cout << "顶点的值的情况如下：" << endl;

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

cout << ver[i] << ' ';

}

cout << endl;

cout << "边的条数为：" << edges << endl;

cout << "邻接矩阵的情况为：" << endl;

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

for (int j = 0; j < vers; ++j) {

cout << edge[i][j] << ' ';

}

cout << endl;

}

}

//七、一些高级操作：

//检查两个结点之间是否有路径存在（外壳部分，公有成员函数）

//注意：判断的时u到v有没有路径

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::checkRoute(int u, int v)const {

if (this->isEmpty() || edges == 0) {//图为空或一条边都没有时必无路径

return false;

}

else {

if (u < 0 || u >= vers || v < 0 || v >= vers) {//u、v不在合理的范围内

return false;

}

else {

if (u == v) {

return true;

}

int\* visited = new int[vers];

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

visited[i] = 0;//用于表示改结点有没有被访问过，为0则未被访问，为1则已经被访问过

}

bool r=checkRoute(u, v, visited);

delete[]visited;

return r;

}

}

}

//检查两个结点之间是否有路径存在（递归部分，私有成员函数）

//思想：用DFS的方法一条路摸到底，如果找到了目标就立刻返回true，如果没有遇到目标那就返回false

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::checkRoute(int u, int v, int visited[])const {

visited[u] = 1;

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

if (edge[u][i] != noEdge && visited[i] == 0) {

if (i == v) {

return true;

}

if (checkRoute(i, v, visited)) {

return true;

}

}

}

return false;

}

//重载形式1

//拓扑排序（仅提供拓扑排序是否成功）

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::topSort()const {//写这个函数的时候，必须保证图的任何信息不饿能发生任何变化

if (this->isEmpty()) {//所以要定义一个flag标志数组来辅助实现topSort排序算法的中间变化

return false;

}

else {

if (edges == 0) {//边数为0，直接对顶点数组遍历并以此放入topsort数组中

return true;

}

else {

int\* flag = new int[vers];//用于表示第i个顶点是否已经进入队列

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

flag[i] = 0;

}

queue<int>q;

int num = 0;

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

if (this->getInDegree(i) == 0) {

q.push(i);

++num;//表示进入队列的结点的个数

flag[i] = 1;

}

}

int k = 0;

while (!q.empty()) {

int ing = q.front();

q.pop();

++k;

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

if (flag[i] == 0 && edge[ing][i] != noEdge) {//作为当前处理结点的邻接顶点而且又没有被放入队列

//检验一下其入度是否为0

for (int j = 0; j < vers; ++j) {//此时的j代表的是行标

if (edge[j][i] != noEdge && flag[j] == 0) {

break;

}

else if ((j == vers - 1 && edge[j][i] != noEdge && flag[j] == 1) || (j == vers - 1 && edge[j][i] == noEdge)) {

q.push(i);

++num;

flag[i] = 1;

}

}

}

}

}

delete[] flag;

if (num == vers)

return true;//说明进拓扑排序的顶点个数等于总顶点个数，返回true（表明有向图不存在回路）

else return false;//num<vers，说明进拓扑排序的顶点个数小于总顶点个数，返回false（表明有向图存在回路）

}

}

}

//重载形式2

//拓扑排序（不仅提供拓扑排序是否成功，排序序列存储在引用参数topsort[]中）

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::topSort(int& num, int topsort[])const {

if (this->isEmpty()) {

return false;

}

else {

if (edges == 0) {//边数为0，直接对顶点数组遍历并以此放入topsort数组中

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

topsort[i] = i;//topsort数组在传入前已经申请好相应空间

}

return true;

}

else {

int\* flag = new int[vers];//用于表示第i个顶点是否已经进入队列

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

flag[i] = 0;

}

queue<int>q;

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

if (this->getInDegree(i)==0) {

q.push(i);

++num;//表示进入队列的结点的个数

flag[i] = 1;

}

}

int k = 0;

while (!q.empty()) {

int ing = q.front();

q.pop();

topsort[k] = ing;

++k;

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

if (flag[i]==0&&edge[ing][i] != noEdge) {//作为当前处理结点的邻接顶点而且又没有被放入队列

//检验一下其入度是否为0

for (int j = 0; j < vers; ++j) {//此时的j代表的是行标

if (edge[j][i] != noEdge && flag[j]==0) {

break;

}

else if((j==vers-1&&edge[j][i]!=noEdge&&flag[j]==1)||(j==vers-1&&edge[j][i]==noEdge)){

q.push(i);

++num;

flag[i] = 1;

}

}

}

}

}

delete[] flag;

if (num == vers)

return true;

else return false;

}

}

}

//3、判断图是否为回路

template<class TypeOfVer,class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::isDAG()const {

if (this->isEmpty()||edges<2) {//图为空或边数小于2，则该图必无回路

return true;

}

return this->topSort();//如果不满足上述情况，调用拓扑排序函数进行直接判断即可

}

//无向图判断回路

//思想：

/\*（1）对图中的每一个元素染色（实际操作可用字符表示）， 初始状态将全部元素染成白色（'W'），保存在数组visited中；

（2）创建一个队列Q。从图中第一个结点（顶点数组的第一个元素）开始BFS。元素入队染成灰色（'G'），出队后染成

黑色（'B'），保存在数组visited中；

（3）邻近结点有三种可能：黑（'B'）、白（'W'）、灰（'G'）。若为黑色结点，说明是BFS序列之前已访问过的，忽略不管

（避免重复访问）；若是白色结点则入队，颜色染灰（'G'）；若有灰色结点则表明该结点已存在于队列中，即两个BFS方向的序列将在此汇合，说明有回路；

（4）若没有，则继续BFS，直至遍历完全部结点，说明没有回路。\*/

template<class TypeOfVer, class TypeOfEdge>

bool adjmatrix\_graph<TypeOfVer, TypeOfEdge>::U\_Judge\_Cir()const {

if (this->isEmpty()&&edges<2) {//图为空且该图的边数小于2，则必没有回路

return false;

}

else {

int\* visited = new int[vers];//表示每个顶点状态的数组，为-1时表示该顶点还没有进队列，为0时表示该顶点已经

for (int i = 0; i < vers; ++i) {//进入队列但还没有被访问，为1时表示已经被访问

visited[i] = -1;

}

queue<int>q;

q.push(0);

visited[0] = 0;

int num = 0;

while (!q.empty()) {

int ing = q.front();

visited[ing] = 1;

q.pop();

++num;

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

if (edge[ing][i] != noEdge) {

if (visited[i] == -1) {

q.push(i);//未入队，进行入队操作

visited[i] = 0;

}

else if (visited[i] == 0&&i!=ing) {

delete[] visited;

return true;//表明有回路，返回true；

}

else continue;//此时表明已经访问过，不要再度访问

}

}

if (q.empty() && num < vers) {//这是为了应对当无向图的可以分为两个或多个独立的部分时判断回路的情况

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

if (visited[i] == -1) {

q.push(i);

visited[i] = 0;

break;

}

}

}

}

delete[] visited;

return false;

}

}

//注意：main函数是任意输入一个无向图（有权或无权）判断其有没有回路，可在其基础上稍加修改再去做其他的题

int main() {

string kd;//矩阵类型名

cin >> kd;

if (kd == "UDN" || kd == "DN") {

int vers;

int noEdgeFlag;

cin >> vers;

string\* ver = new string[vers];

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

cin >> ver[i];

}

cin >> noEdgeFlag;

int edges;//边数

cin >> edges;

pair<int, int>\* edge = new pair<int, int>[edges];

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

cin >> edge[i].first >> edge[i].second;

}

int\* w = new int[edges];

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

cin >> w[i];

}

w = sort\_weight(kd, edge, edges, w);//权值数组重新排序

int\*\* e = transform\_sequence\_to\_array(kd, edge, edges, vers);//将对应的边转化为二维矩阵e，但是e是一个单纯

//的不含权值的矩阵，每个元素的值只可以反映这两个对应的点之间边的个数，为0表示没有，为1表示1条，以此类推

//它并不是我们所要的最终的邻接矩阵，只是一个中间过渡产物

adjmatrix\_graph<string, int>G(kd, vers, edges, noEdgeFlag, ver, e, w);

string\* s = G.getVerPointer();

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

cout << s[i];

if (i < vers - 1) {

cout << ' ';

}

}

cout << endl << endl;

//cout << G.getEdgeNum() << endl;

G.printMatrix();

cout << endl << endl;

int num = 0;

bool r = G.U\_Judge\_Cir();

if (r) {

cout << "true";

}

else {

cout << "false";

}

delete[] ver;//一定不要忘记显式地释放动态申请到的内存

delete[] edge;

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

delete[] e[i];

}

delete[] e;

delete[] w;

}

else {

int vers;

cin >> vers;

string\* ver = new string[vers];

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

cin >> ver[i];

}

int edges;//边数

cin >> edges;

pair<int, int>\* edge = new pair<int, int>[edges];

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

cin >> edge[i].first >> edge[i].second;

}

int\*\* e = transform\_sequence\_to\_array(kd, edge, edges, vers);//将对应的边转化为二维矩阵e，但是e是一个单纯

//的不含权值的矩阵，每个元素的值只可以反映这两个对应的点之间边的个数，为0表示没有，为1表示1条，以此类推

//它并不是我们所要的最终的邻接矩阵，只是一个中间过渡产物

adjmatrix\_graph<string, int>G(kd, vers, edges, ver, e);

string\* s = G.getVerPointer();

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

cout << s[i];

if (i < vers - 1) {

cout << ' ';

}

}

cout << endl << endl;

//cout << G.getEdgeNum() << endl;

G.printMatrix();

cout << endl << endl;

int num = 0;

bool r = G.U\_Judge\_Cir();

if (r) {

cout << "true";

}

else {

cout << "false";

}

delete[] ver;//一定不要忘记显式地释放动态申请到的内存

delete[] edge;

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

delete[] e[i];

}

delete[] e;

}

return 0;

}