#include<iostream>

#include<cstring>

#include<utility>

#include<algorithm>

#include<queue>

using namespace std;

/\* 边表的结点定义 \*/

template<class TypeOfEdge>

struct edgeNode {

int data;//data值指示的是某个顶点对应的位序（允许的范围为：0到顶点个数-1）

TypeOfEdge weight;//表示权重，但是当建立无权图图时不对其进行赋值也不考虑其值的相关变化

edgeNode<TypeOfEdge>\* next;

edgeNode<TypeOfEdge>() {

next = NULL;

}

edgeNode<TypeOfEdge>(const int& d, edgeNode<TypeOfEdge>\* ptr = NULL) { //构造函数，用于构造其他结点（无权图）

//函数参数表中的形参允许有默认值，但是带默认值的参数需要放后面

next = ptr;

data = d;

}

edgeNode<TypeOfEdge>(const int& d, const TypeOfEdge& w, edgeNode<TypeOfEdge>\* ptr = NULL) { //构造函数，用于构造其他结点（带权图）

//函数参数表中的形参允许有默认值，但是带默认值的参数需要放后面

next = ptr;

data = d;

weight = w;

}

int getData()const { return data; } //取得结点的序号（顶点集）

TypeOfEdge getWeight()const { return weight; } //取得边集中对应边的权值

void setLink(edgeNode<TypeOfEdge>\* link) { next = link; } //修改结点的next域

void setData(int value) { data = value; } //修改结点的序号（顶点集）

void setWeight(TypeOfEdge value) { weight = value; } //修改边集中对应边的权值

};

//图的邻接表类

//存放的是每一个顶点的值和其后所接的按头插法插入的邻接顶点的位序

template<class TypeOfVer, class TypeOfEdge>

struct verNode {

TypeOfVer ver;//表示改邻接表顶点的值，而不是位序（注意：位序必然是一个int型的，而TypeOfVer不一定是int型的）

edgeNode<TypeOfEdge>\* head;//指向其邻接顶点结点的指针域

//构造函数：

//一个默认域传参合为一体的构造函数：

verNode<TypeOfVer, TypeOfEdge>(edgeNode<TypeOfEdge>\* h = NULL) { head = h; }

//带data参数的构造函数：

verNode<TypeOfVer, TypeOfEdge>(const TypeOfVer& d, edgeNode<TypeOfEdge>\* h = NULL) {

ver = d;

head = h;

}

TypeOfVer getVer()const { return ver; } //取得结点值（顶点集）

edgeNode<TypeOfEdge>\* getHead()const { return head; } //取得对应的边表的头指针

void setVer(TypeOfVer value) { ver = value; } //设置结点值（顶点集）

void setHead(edgeNode<TypeOfEdge> value) { head = value; } //设置对应的边表的头指针

};

template <class TypeOfVer, class TypeOfEdge>

class adjlist\_graph {

private:

string graphKind; //图的种类标志

int vers; //顶点数

int edges; //边数

verNode<TypeOfVer, TypeOfEdge>\* verList;

//bool DeleteEdge(int u, int v);暂时好像用不到这个函数

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

public:

//一、构造与析构：

//1、构造函数构造一个只有结点没有边的图：

adjlist\_graph(const string& kd, int vSize, const TypeOfVer d[]);

//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：

adjlist\_graph(const string& kd, int vSize, int eSize, const TypeOfVer d[], int\*\* e);

//3、构造函数构造一个有权图：

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

//4、析构函数

~adjlist\_graph();

//5、拷贝构造函数：

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

//6、拷贝赋值运算符：

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

//二、信息提取型：

//1、获取图的类型：

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、取得G中指定顶点的值，对应的位序为u将值存入data中

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

//5、取得图中的指向存放顶点数组指针的首地址（本质上为一个指针）：

verNode<TypeOfVer, TypeOfEdge>\* getVerListPointer()const { return verList; }

//6、对于有权图，取两端点为v1和v2的边上的权值。获取成功，返回true；否则，返回false

bool getWeight(const int& u,const int& v, int& w)const;

//7、返回G中指定顶点的位置

int locateVer(const TypeOfVer& data)const;

//8、求任意图的指定顶点的出度：

//注意：若图为空或顶点不存在，则返回-1，其他情况均可以返回一个大于等于0的整数

int getOutDegree(const int& u)const;

//9、求有向图中指定顶点的入度：

//若为无向图或图为空或顶点不存在，则返回-1，其他情况均可以返回一个大于等于0的整数

int getInDegree(const int& u)const;

//10、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

int getFirstAdjVex(const int& u)const;

//11、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false

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

//三、设置型函数：

//1、对G中指定顶点赋值

//u为顶点对应的位序（合法范围：0到vers-1），data为要给顶点赋予的值

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

//四、插入删除型：

//（一）对于顶点：

//1、往G中添加一个顶点

bool insertVer(const TypeOfVer& data);

//2、往G中删除一个顶点，传进来的是顶点的值

bool deleteVer(const TypeOfVer& data);

//3、删除一个顶点：传进来的是位序

bool deleteVer(const int& u);

//（二）对于边：

//1、无权图插入一条边

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

//2、有权图插入一条边

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

//3、删除边 (外壳：有向（删除1条边）， 无向（删除2条边））

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

//五、判断型：

//1、判断图空否

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

//2、两点之间是否存在边，即u、v是否为邻接顶点（对于有向图来说，则是是否存在u到v的边）

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

//3、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(const int& u, const 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(const int& u, const int& v, int visited[])const;

//4、（1）重载形式1

//拓扑排序（仅提供拓扑排序是否成功）

bool topSort()const;

//（2）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）

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

//5、判断所给的有向图中有没有回路：

bool isDAG()const;

//6、无向图（网）判断回路（有回路，返回true，无回路，返回false）：

bool U\_Judge\_Cir()const;

//六、遍历型：

//1、输出顶点集

bool printVer()const;

//2、输出邻接矩阵

bool printAdjList()const;

//3、DFS遍历（外壳部分）

void DFS\_Traverse(const int& u)const;

//B4、FS遍历

void BFS\_Traverse(const int& u)const;

//5、图的相关信息展示函数

void show()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、构造与析构：

//1、构造函数构造一个只有结点没有边的图：

template<class TypeOfVer, class TypeOfEdge>

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

graphKind = kd;

vers = vSize;

edges = 0;

verList = new verNode<TypeOfVer, TypeOfEdge>[vers];//动态申请的存放各个顶点值的数组

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

verList[i].ver = d[i];//对每个顶点的值进行一个拷贝

}

}

//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>

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

graphKind = kd;

vers = vSize;

edges = eSize;

verList = new verNode<TypeOfVer, TypeOfEdge>[vers];

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

verList[i].ver = d[i];

}

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

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

if (e[i][j] != 0) {//说明i、j之间有边，则应该往顶点位序为i的head域后挂data值为j的结点

edgeNode<TypeOfEdge>\* tmp = new edgeNode<TypeOfEdge>;

tmp->data = j;

tmp->next = verList[i].head;

verList[i].head = tmp;

}//注意：一般来说，不管对于有向图还是无向图，从一个顶点到另一个顶点之间只有一条边，因此这种构造方法是可行的

}

}

}//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：

template<class TypeOfVer, class TypeOfEdge>

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

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

graphKind = kd;

vers = vSize;

edges = eSize;

verList = new verNode<TypeOfVer, TypeOfEdge>[vers];

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

verList[i].ver = d[i];

}

if (graphKind[0] == 'U') {//表明要创建无向图，这个情况赋予权值比较复杂

int\*\* flag = new int\* [vers];//用flag主要来表示e数组对应的不为0的元素是否已经被处理过，为0表示未被处理，为1表示已经被处理

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

flag[i] = new int[vers];

}

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

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

flag[i][j] = 0;

}

}

int k = 0;

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

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

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

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

flag[i][j] = 1;

if (graphKind == "UDN") {

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

flag[j][i] = 1;

}

++k;

}

}

}

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

delete[] flag[i];

}

delete[] flag;

}

else {

int k = 0;

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

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

if (e[i][j] != 0) {

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

++k;

}

}

}

}

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

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

if (e[i][j] != 0) {

edgeNode<TypeOfEdge>\* tmp = new edgeNode<TypeOfEdge>(j, e[i][j]);

tmp->next = verList[i].head;

verList[i].head = tmp;

}

}

}

}//暂时认为没有什么问题

//4、析构函数

//实现策略：先断链，后释放数组

template<class TypeOfVer, class TypeOfEdge>

adjlist\_graph<TypeOfVer, TypeOfEdge>::~adjlist\_graph() {

//断链：

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

edgeNode<TypeOfEdge>\* p = verList[i].head;

while (p) {

verList[i].head = p->next;

delete p;

p = verList[i].head;

}

}

//释放动态申请的数组空间：

delete[] verList;

//cout << "调用了析构函数" << endl;

}//经过测试没有问题

//5、拷贝构造函数：

template<class TypeOfVer, class TypeOfEdge>

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

graphKind = G.getGraphKind();

vers = G.getVerNum();

edges = G.getEdgeNum();

if (vers == 0) return;

verList = new verNode<TypeOfVer, TypeOfEdge>[vers];

verNode<TypeOfVer, TypeOfEdge>\* v = G.getVerListPointer();

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

verList[i].ver = v[i].ver;

edgeNode<TypeOfEdge>\* p = v[i].head;

edgeNode<TypeOfEdge>\* q = verList[i].head;

while (p) {

edgeNode<TypeOfEdge>\* tmp = new edgeNode<TypeOfEdge>(p->data, p->weight);

if (q == verList[i].head) {

tmp->next = verList[i].head;

verList[i].head = tmp;

q = tmp;

}

else {

tmp->next = q->next;

q->next = tmp;

q = tmp;

}

p = p->next;

}

}

}

//6、拷贝赋值运算符：

template<class TypeOfVer, class TypeOfEdge>

adjlist\_graph<TypeOfVer, TypeOfEdge>& adjlist\_graph<TypeOfVer, TypeOfEdge>::operator=(const adjlist\_graph<TypeOfVer, TypeOfEdge>& G) {

graphKind = G.getGraphKind();

vers = G.getVerNum();

edges = G.getEdgeNum();

if (vers == 0)return \*this;

verList = new verNode<TypeOfVer, TypeOfEdge>[vers];

verNode<TypeOfVer, TypeOfEdge>\* v = G.getVerListPointer();

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

verList[i].ver = v[i].ver;

edgeNode<TypeOfEdge>\* p = v[i].head;

edgeNode<TypeOfEdge>\* q = verList[i].head;

while (p) {

edgeNode<TypeOfEdge>\* tmp = new edgeNode<TypeOfEdge>(p->data, p->weight);

if (q == verList[i].head) {

tmp->next = verList[i].head;

verList[i].head = tmp;

q = tmp;

}

else {

tmp->next = q->next;

q->next = tmp;

q = tmp;

}

p = p->next;

}

}

return \*this;

}

//二、信息提取型：

//4、取得G中指定顶点的值

//传进来的u的合法范围为：0到vers-1

//注意：当返回false的时候，将不对data进行赋值，要当心data的值未定义

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty()) {

//data = (TypeOfVer)(0);

return false;

}

else {

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

//data = (TypeOfVer)(0);

return false;

}

else {//表明此时u的取值合法

data = verList[u].ver;

return true;

}

}

}//暂时认为这个函数没有问题

//6、对于有权图，取两端点为v1和v2的边上的权值。获取成功，返回true；否则，返回false

//注:若返回false，则不对w进行赋值

template<class TypeOfVer,class TypeOfEdge>

bool adjlist\_graph<TypeOfVer, TypeOfEdge>::getWeight(const int& u, const int& v, int& w)const {

if (this->isEmpty() || edges == 0 || vers == 1) {//图为空、边数为0、只有一个顶点，统统返回false

return false;

}

else {

if (graphKind != "UDN" && graphKind != "DN") {//不是有权图，也返回false

return false;

}

else {

if (u < 0 || u >= vers || v < 0 || v >= vers) {//u、v不在合法的范围，返回false

return false;

}

else {//注意：若u、v之间有边，则我们只对verList[u]所在的链进行遍历，因为对于无向图：<u,v>与<v,u>等价，而对于有向图则不是，

//但我们只负责找u到v的边的权值，对于有向图无向图来说均是u到v

edgeNode<TypeOfEdge>\* p = verList[u].head;

while (p) {

if (p->data == v) {

w = p->weight;

return true;

}

else {

p = p->next;

}

}

return false;//表明不存在u到v的边

}

}

}

}

//7、返回G中指定顶点的位置

//注意：如果这里可以找到一个顶点的值为data，则返回的整数的范围为0到vers-1，即为顶点的位序

//如果找不到或是图为空，则返回-1

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty()) {

return -1;

}

else {

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

if (verList[i].ver == data) {

return i;

}

}

return -1;//能运行到这一步表明肯定没有找到data，所以返回-1

}

}//暂时认为没有问题

//8、求任意图的指定顶点的出度：

//注意：若图为空或顶点不存在，则返回-1，其他情况均可以返回一个大于等于0的整数

template<class TypeOfVer,class TypeOfEdge>

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

if (this->isEmpty()) {//图为空

return -1;

}

else {

if (u < 0 || u >= vers) {//u不在合法的范围内

return -1;

}

else {//图不为空且u在合法的范围内

int count = 0;

edgeNode<TypeOfEdge>\* p = verList[u].head;

while (p) {

++count;

p = p->next;

}

return count;

}

}

}

//9、求有向图中指定顶点的入度：

//若为无向图或图为空或顶点不存在，则返回-1，其他情况均可以返回一个大于等于0的整数

template<class TypeOfVer,class TypeOfEdge>

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

if (graphKind[0] == 'U' || this->isEmpty()) {//图为无向图或该图为空

return -1;

}

else {

if (u < 0 || u >= vers) {//u不在合法的范围内

return -1;

}

else {//因为是通过出度关系建立的整个邻接表，这就使得求入度的时候很麻烦，必须遍历整个邻接表

int count = 0;

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

if (i != u) {//注意：我们所处理的图，其中的顶点没有指向自己的边，因此统计入度的时候不需要统计第u条链

edgeNode<TypeOfEdge>\* p = verList[i].head;

while (p) {

if (p->data == u) {

++count;

}

p = p->next;

}

}

}

return count;

}

}

}

//10、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

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty() || vers == 1) {//图为空或顶点个数总共为1

return -1;

}

else {

if (u < 0 || u >= vers) {//u、v不在合法范围内

return -1;

}

else {

edgeNode<TypeOfEdge>\* p = verList[u].head;

if (p) {

return p->data;

}

else {

return -1;

}

}

}

}//暂时认为没有什么问题

//11、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false

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty() || vers == 1) {//图为空或顶点个数总共为1

return -1;

}

else {

if (u < 0 || u >= vers || v < 0 || v >= vers) {//u、v不在合法范围内

return -1;

}

else {

if (u == v) {//要特殊处理u==v的情况

if (verList[u].head) {

return verList[u].head->data;

}

else {

return -1;

}

}

else {

edgeNode<TypeOfEdge>\* p = verList[u].head;

while (p) {

if (p->data == v) {

if (p->next) {

return p->next->data;

}

else {

return -1;

}

}

else p = p->next;

}

return -1;

}

}

}

}//暂时认为没有问题

//三、设置型函数：

//1、对G中指定顶点赋值

//u为顶点对应的位序（合法范围：0到vers-1），data为要给顶点赋予的值

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty()) {//图为空，没有必要进行赋值，返回false

return false;

}

else {

if (u < 0 || u >= vers) {//u的取值不在合法范围内，也没有必要进行赋值，返回false

return false;

}

else {//只有这种情况下才有必要对顶点进行重新设置值并且返回true

verList[u].ver = data;

return true;

}

}

}//暂时认为没有问题

//四、插入删除型：

//1、往G中添加一个顶点

//实现策略：重新申请一块新的空间然后再整体搬家，但一定不要忘了释放掉原来的空间

template<class TypeOfVer, class TypeOfEdge>

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

++vers;

if (vers == 1) {

edges = 0;

verList = new verNode<TypeOfVer, TypeOfEdge>[vers];

verList[0].ver = data;

return true;

}

else {

verNode<TypeOfVer, TypeOfEdge>\* nv = new verNode<TypeOfVer, TypeOfEdge>[vers];

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

nv[i].ver = verList[i].ver;

edgeNode<TypeOfEdge>\* p = verList[i].head;

edgeNode<TypeOfEdge>\* q = nv[i].head;

while (p) {

edgeNode<TypeOfEdge>\* tmp = new edgeNode<TypeOfEdge>(p->data, p->weight);

if (nv[i].head == NULL) {

tmp->next = nv[i].head;

nv[i].head = tmp;

q = tmp;

}

else {

tmp->next = q->next;

q->next = tmp;

q = tmp;

}

edgeNode<TypeOfEdge>\* tq = p;//将来要删除tq

p = p->next;

verList[i].head = p;

delete tq;

}//一边移动、一边插结点、一边删除

}

nv[vers - 1].ver = data;//根据verNode的默认构造函数，其head域的值必为NULL

delete[] verList;//释放原来的空间

verList = nv;//重新赋予新的空间

return true;

}

}//暂时认为没有问题

//2、往G中删除一个顶点，传进来的是顶点的值

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty()) {//空图，没有删除的必要，直接返会=回false

return false;

}

else {

int place = locateVer(data);//先定位

if (place == -1) {//没有找到要删除的顶点

return false;

}

else {//找到了要删除的顶点，下面开始进行删除操作

//先把这个顶点其后所接的邻接顶点的结点全部释放掉

edgeNode<TypeOfEdge>\* p = verList[place].head;

while (p) {

edgeNode<TypeOfEdge>\* dp = p;

p = p->next;

verList[place].head = p;

delete dp;

--edges;//edges也要跟着进行相应的变化

}

//注意还应该删除其他顶点的邻接顶点链中的结点的data值等于place的结点

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

if (i != place) {

edgeNode<TypeOfEdge>\* p = verList[i].head;

edgeNode<TypeOfEdge>\* q = verList[i].head;//q在开始时与p所指的相同，但是当遍历开始后q指向p的前驱

while (p) {

if (p->data == place) {

if (p == verList[i].head) {

verList[i].head = p->next;

delete p;

}

else {

q->next = p->next;

delete p;

}

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

--edges;//删除边edges要有相应的变化

}

break;//从一个顶点到另一个顶点最多只有一条边

}

else {

if (p->data > place) {

p->data -= 1;//注意删除一个顶点后，比这个删除的顶点位序大的其他顶点的位序一定要减1，保持删除后的新位序，

}//不然会引发混乱

q = p;

p = p->next;

}

}

}

}

//进行完上一步之后，开始对verList数组中的元素进行移动使这个顶点在顶点数组中被覆盖掉

--vers;

if (place == vers) {

return true;

}

else {

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

verList[i] = verList[i + 1];//这一步是可以的，因为本质上是交换的不同的不同地址中的一个值对另一个的覆盖

//不会影响值所接的链的关系和其内容

}

}

return true;//千万不要忘记返回值，这个特别容易忘记

}

}

}//暂时经过测试认为没有什么问题

//3、删除一个顶点：传进来的是位序

template<class TypeOfVer, class TypeOfEdge>

bool adjlist\_graph<TypeOfVer, TypeOfEdge>::deleteVer(const int& u) {

if (this->isEmpty()) {//空图，没有删除的必要，直接返会=回false

return false;

}

else {

if (u < 0 || u >= vers) {//没有找到要删除的顶点

return false;

}

else {//找到了要删除的顶点，下面开始进行删除操作

//先把这个顶点其后所接的邻接顶点的结点全部释放掉

edgeNode<TypeOfEdge>\* p = verList[u].head;

while (p) {

edgeNode<TypeOfEdge>\* dp = p;

p = p->next;

verList[u].head = p;

delete dp;

--edges;//edges也要跟着进行相应的变化

}

//注意还应该删除其他顶点的邻接顶点链中的结点的data值等于place的结点

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

if (i != u) {

edgeNode<TypeOfEdge>\* p = verList[i].head;

edgeNode<TypeOfEdge>\* q = verList[i].head;//q在开始时与p所指的相同，但是当遍历开始后q指向p的前驱

while (p) {

if (p->data == u) {

if (p == verList[i].head) {

verList[i].head = p->next;

delete p;

}

else {

q->next = p->next;

delete p;

}

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

--edges;//删除边edges要有相应的变化

}

break;//从一个顶点到另一个顶点最多只有一条边

}

else {

if (p->data > u) {

p->data -= 1;//注意删除一个顶点后，比这个删除的顶点位序大的其他顶点的位序一定要减1，保持删除后的新位序，

}//不然会引发混乱

q = p;

p = p->next;

}

}

}

}

//进行完上一步之后，开始对verList数组中的元素进行移动使这个顶点在顶点数组中被覆盖掉

--vers;

if (u == vers) {

return true;

}

else {

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

verList[i] = verList[i + 1];//这一步是可以的，因为本质上是交换的不同的不同地址中的一个值对另一个的覆盖

//不会影响值所接的链的关系和其内容

}

}

return true;//千万不要忘记返回值，这个特别容易忘记

}

}

}

//（二）对于边：

//1、无权图插入一条边

//我们所构建的图从一个顶点到另一个顶点之间只能有一条边，如果原来u、v之间有边则返回false

//如果可以插入的话则采用的是头插法

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty() || vers == 1) {//图为空，即图没有一个顶点或图只有一个顶点，没有必要插入边，返回false

return false;

}

else {

if (u == v) {//我们不允许插入顶点自己与自己相连的边

return false;

}

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

return false;

}

else {

//先遍历，看看u、v之间有没有边，没有边再进行插入

edgeNode<TypeOfEdge>\* p = verList[u].head;

if (!p) {//p为NULL，直接插入

++edges;//一定要记得edges加1

edgeNode<TypeOfEdge>\* tmp = new edgeNode<TypeOfEdge>(v);

tmp->next = verList[u].head;

verList[u].head = tmp;

if (graphKind[0] == 'U') {//表明是无向图

edgeNode<TypeOfEdge>\* tmp2 = new edgeNode<TypeOfEdge>(u);

tmp2->next = verList[v].head;

verList[v].head = tmp2;

}

return true;

}

else {//p不为NULL，先检查再判断该不该插入

while (p) {

if (p->data == v) {

return false;

}

p = p->next;

}

++edges;//一定要记得edges加1

edgeNode<TypeOfEdge>\* tmp = new edgeNode<TypeOfEdge>(v);//如果能进行到这一步就说明原来的u、v之间一定没有边

tmp->next = verList[u].head;//可以进行插入

verList[u].head = tmp;

if (graphKind[0] == 'U') {//表明是无向图

edgeNode<TypeOfEdge>\* tmp2 = new edgeNode<TypeOfEdge>(u);

tmp2->next = verList[v].head;

verList[v].head = tmp2;

}

return true;

}

}

}

}//经过测试暂时没有问题

//2、有权图插入一条边

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty() || vers == 1) {//图为空，即图没有一个顶点或图只有一个顶点，没有必要插入边，返回false

return false;

}

else {

if (u == v) {//我们不允许插入顶点自己与自己相连的边

return false;

}

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

return false;

}

else {

//先遍历，看看u、v之间有没有边，没有边再进行插入

edgeNode<TypeOfEdge>\* p = verList[u].head;

if (!p) {//p为NULL，直接插入

++edges;//一定要记得edges加1

edgeNode<TypeOfEdge>\* tmp = new edgeNode<TypeOfEdge>(v, w);

tmp->next = verList[u].head;

verList[u].head = tmp;

if (graphKind[0] == 'U') {//表明是无向图

edgeNode<TypeOfEdge>\* tmp2 = new edgeNode<TypeOfEdge>(u, w);

tmp2->next = verList[v].head;

verList[v].head = tmp2;

}

return true;

}

else {//p不为NULL，先检查再判断该不该插入

while (p) {

if (p->data == v) {

return false;

}

p = p->next;

}

++edges;//一定要记得edges加1

edgeNode<TypeOfEdge>\* tmp = new edgeNode<TypeOfEdge>(v, w);//如果能进行到这一步就说明原来的u、v之间一定没有边

tmp->next = verList[u].head;//可以进行插入

verList[u].head = tmp;

if (graphKind[0] == 'U') {//表明是无向图

edgeNode<TypeOfEdge>\* tmp2 = new edgeNode<TypeOfEdge>(u, w);

tmp2->next = verList[v].head;

verList[v].head = tmp2;

}

return true;

}

}

}

}//暂时没有问题

//3、删除边 (外壳：有向（删除1条边）， 无向（删除2条边））

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty() || edges == 0 || vers == 1) {//空图或没有边或只有一个顶点

return false;

}

else {

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

return false;

}

else {

edgeNode<TypeOfEdge>\* p = verList[u].head;

edgeNode<TypeOfEdge>\* q = verList[u].head;//q只在未开始遍历前与p指向同一个地方，开始后指向p的前驱

while (p) {

if (p->data == v) {

--edges;

if (p == verList[u].head) {

verList[u].head = p->next;

delete p;

}

else {

q->next = p->next;

delete p;

}

if (graphKind[0] == 'U') {//无向图要删除两条边，有向图就不用

edgeNode<TypeOfEdge>\* p2 = verList[v].head;

edgeNode<TypeOfEdge>\* q2 = verList[v].head;//q2只在未开始遍历前与p2指向同一个地方，开始后指向p2的前驱

while (p2) {

if (p2->data == u) {

if (p2 == verList[v].head) {

verList[v].head = p2->next;

delete p2;

}

else {

q2->next = p2->next;

delete p2;

}

break;//删除后要跳出去

}

else {

q2 = p2;

p2 = p2->next;

}

}

}

return true;

}

else {

q = p;

p = p->next;

}

}

return false;

}

}

}//暂时认为没有问题

//五、判断型：

//2、两点之间是否存在边，即u、v是否为邻接顶点（对于有向图来说，则是是否存在u到v的边）

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty() || vers == 1) {//图为空或者只有一个顶点，则 一定返回false

return false;

}

else {

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

return false;//u、v不在合法范围，返回false

}

else {

if (u == v) {//我们所考虑的图一般不存在指向自己的边

return false;

}

else {

edgeNode<TypeOfEdge>\* p = verList[u].head;

while (p) {

if (p->data == v) {

return true;

}

else p = p->next;

}

return false;//能进行到这一步表明u、v之间肯定没有边

}

}

}

}//暂时认为没有问题

//3、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）

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty() || vers == 1 || edges == 0) {//图为空、顶点个数为1或边数为0，必没有路径，返回false

return false;

}

else {

if (u < 0 || u >= vers || v < 0 || v >= vers) {//传入的u、v不在合法的范围内

return false;

}

else if (u == v) {

return true;

}

else {

int\* visited = new int[vers];

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

visited[i] = 0;//表示还没有被访问过

}

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

delete[] visited;//一定要记得显式释放动态申请的内存空间

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）

template<class TypeOfVer, class TypeOfEdge>

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

edgeNode<TypeOfEdge>\* p = verList[u].head;

visited[u] = 1;//表明对应的第u个顶点已经被访问过

while (p) {

if (p->data == v) {

return true;

}

else {

if (visited[p->data] == 0) {//没有访问过的顶点才有必要进行递归

bool tmp\_r = checkRoute(p->data, v, visited);

if (tmp\_r) {

return tmp\_r;

}

}

p = p->next;

}

}

return false;//执行到最外层的这一步表明没有找到路径，则返回false

}

//4、（1）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 adjlist\_graph<TypeOfVer, TypeOfEdge>::topSort()const {

if (this->isEmpty()) {//图为空，返回false，将num赋值为0，不为topsort数组赋值

return false;

}

else if (edges == 0) {

return true;

}

else {

int num = 0;

int\* flag = new int[vers];

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

flag[i] = 0;

}

int\* inDegree = new int[vers];//保存对应位序的顶点的入度

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

inDegree[i] = getInDegree(i);

}

queue<int>q;

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

if (inDegree[i] == 0) {

q.push(i);

flag[i] = 1;//表示对应位序的顶点已经入队

++num;

}

}

int k = 0;

while (!q.empty()) {

int ing = q.front();

q.pop();

++k;

edgeNode<TypeOfEdge>\* p = verList[ing].head;

while (p) {

if (flag[p->data] == 0) {

--inDegree[p->data];//已经访问过后才对由这个顶点发出的边做删除，即位序为p->data对应的入度减1

if (inDegree[p->data] == 0) {

q.push(p->data);

flag[p->data] = 1;

++num;

}

}

p = p->next;

}

}

delete[] flag;

delete[] inDegree;

if (num < vers) {

return false;

}

else return true;

}

}

//（2）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 adjlist\_graph<TypeOfVer, TypeOfEdge>::topSort(int& num, int topsort[])const {

if (this->isEmpty()) {//图为空，返回false，将num赋值为0，不为topsort数组赋值

num = 0;

return false;

}

else if (edges == 0) {

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

topsort[i] = 1;

}

return true;

}

else {

int\* flag = new int[vers];

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

flag[i] = 0;

}

int\* inDegree = new int[vers];//保存对应位序的顶点的入度

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

inDegree[i] = getInDegree(i);

}

queue<int>q;

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

if (inDegree[i] == 0) {

q.push(i);

flag[i] = 1;//表示对应位序的顶点已经入队

++num;

}

}

int k = 0;

while (!q.empty()) {

int ing = q.front();

q.pop();

topsort[k] = ing;

++k;

edgeNode<TypeOfEdge>\* p = verList[ing].head;

while (p) {

if (flag[p->data] == 0) {

--inDegree[p->data];//已经访问过后才对由这个顶点发出的边做删除，即位序为p->data对应的入度减1

if (inDegree[p->data] == 0) {

q.push(p->data);

flag[p->data] = 1;

++num;

}

}

p = p->next;

}

}

delete[] flag;

delete[] inDegree;

if (num < vers) {

return false;

}

else return true;

}

}

//5、判断所给的有向图中有没有回路：

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty() || vers == 1 || edges == 0) {

return false;//没有回路

}

else if (graphKind[0] == 'U') {//不判断无向图，返回false

return false;

}

else {

if (this->topSort() == false) {//拓扑排序不成，说明有回路

return true;

}

else {//拓扑排序成功，说明没有回路

return false;

}

}

}

//6、无向图（网）判断回路（有回路，返回true，无回路，返回false）：

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty() || vers == 0 || edges == 0) {

return false;//没有回路

}

else if (graphKind[0] == 'D') {//不判断有向图，返回false

return false;

}

else {

int\* visited = new int[vers];

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

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;

++num;

q.pop();

edgeNode<TypeOfEdge>\* p = verList[ing].head;

while (p) {

if (visited[p->data] == -1) {

q.push(p->data);

visited[p->data] = 0;

p = p->next;

}

else if (visited[p->data] == 0) {

return true;//存在回路

}

else p = p->next;

}

if (q.empty() && num < vers) {

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

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

q.push(i);

visited[i] = 0;

}

}

}

}

delete[] visited;//一定要记得释放动态申请的内存空间

return false;

}

}

//六、遍历型：

//1、输出顶点集

template<class TypeOfVer, class TypeOfEdge>

bool adjlist\_graph<TypeOfVer, TypeOfEdge>::printVer()const {

if (this->isEmpty()) {

return false;

}

else {

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

cout << verList[i].ver;

if (i < vers - 1) {

cout << ' ';

}

}

return true;

}

}//经测试暂时没有问题

//2、输出邻接矩阵

template<class TypeOfVer, class TypeOfEdge>

bool adjlist\_graph<TypeOfVer, TypeOfEdge>::printAdjList()const {

if (this->isEmpty()) {

return false;

}

else {

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

cout << verList[i].ver;

edgeNode<TypeOfEdge>\* p = verList[i].head;

while (p) {

cout << "->";

cout << p->data;

if (graphKind == "DN" || graphKind == "UDN") {//如果是带权图，则需要接着输出对应边的权值

cout << '(' << p->weight << ')';

}

p = p->next;

}

if (i < vers - 1) {

cout << endl;

}

}

return true;

}

}//经测试暂时没有问题

//3、DFS遍历（外壳部分）

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty()) {//图为空，没有遍历的必要

return;

}

else {

if (u < 0 || u >= vers) {//u不在允许的合法范围内，不进行遍历

return;

}

else {

int\* visited = new int[vers];

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

visited[i] = 0;

}

int num = 0;

DFS(u, num, visited);

delete[] visited;//一定要记得对visited所申请的动态空间进行显式释放

}

}

}

//DFS遍历（递归部分）

template<class TypeOfVer, class TypeOfEdge>

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

if (num > 0) cout << "->";

cout << verList[u].ver;

++num;

visited[u] = 1;

edgeNode<TypeOfEdge>\* p = verList[u].head;

if (!p) return false;

while (p) {

if (visited[p->data] == 0) {

DFS(p->data, num, visited);

}

p = p->next;

}

return true;

}//暂时认为DFS遍历没有问题

//4、BFS遍历

template<class TypeOfVer, class TypeOfEdge>

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

if (this->isEmpty()) {//空图，没有遍历的必要

return;

}

else {

if (u < 0 || u >= vers) {//u不在合法的范围内，没有遍历的必要

return;

}

else {

int\* visited = new int[vers];

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

visited[i] = 0;

}

queue<int>q;

q.push(u);

visited[u] = 1;//一放入队列就置为1

int num = 0;

while (!q.empty()) {

int ing = q.front();

q.pop();

if (num > 0) {

cout << "->";

}

cout << verList[ing].ver;

++num;

edgeNode<TypeOfEdge>\* p = verList[ing].head;

while (p) {

if (visited[p->data] == 0) {

q.push(p->data);

visited[p->data] = 1;

}

p = p->next;

}

}

delete[] visited;//一定要记得释放空间

}

}

}

//5、图的相关信息展示函数

template<class TypeOfVer, class TypeOfEdge>

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

cout << "图的相关信息如下：" << endl;

cout << "该图的种类为：" << graphKind << endl;

cout << "该图的顶点个数为：" << vers << endl;

cout << "该图的顶点的具体的值依次为：" << endl;

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

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

}

cout << endl;

cout << "该图的边的个数为：" << endl;

cout << edges << endl;

cout << "该图的邻接表表示的具体情况为：" << endl;

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

cout << verList[i].ver;

edgeNode<TypeOfEdge>\* p = verList[i].head;

while (p) {

cout << "->";

cout << p->data;

if (graphKind == "DN" || graphKind == "UDN") {//如果是带权图，则需要接着输出对应边的权值

cout << '(' << p->weight << ')';

}

p = p->next;

}

if (i < vers - 1) {

cout << endl;

}

}

}

//以下给出的main函数可用于做关于判断一个无向图是否有回路的OJ题，同时也展示了如何运用所写的类创建一个图的过程

int main() {

string kd;

cin >> kd;//图的种类

if (kd == "DN" || kd == "UDN") {//建立并处理带权的图

int vers;

cin >> vers;//顶点个数

string\* d = new string[vers];

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

cin >> d[i];

}

int edges;//边数

cin >> edges;

pair<int, int>\* edge = new pair<int, int>[edges];//用于存储边的序列的动态数组

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

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

}

int\* w = new int[edges];//用于存储每条边权值的数组

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

cin >> w[i];

}

w = sort\_weight(kd, edge, edges, w);

int\*\* e = transform\_sequence\_to\_array(kd, edge, edges, vers);

adjlist\_graph<string, int>G(kd, vers, edges, d, e, w);

G.printVer();

cout << endl;

G.printAdjList();

cout << endl << endl;

if (G.U\_Judge\_Cir()) {

cout << "true";

}

else {

cout << "false";

}

delete[] d;

delete[] edge;

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

delete[] e[i];

}

delete[] e;

delete[] w;

}

else {//建立并处理不带权的图

int vers;

cin >> vers;//顶点个数

string\* d = new string[vers];

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

cin >> d[i];

}

int edges;//边数

cin >> edges;

pair<int, int>\* edge = new pair<int, int>[edges];//用于存储边的序列的动态数组

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

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

}

int\*\* e = transform\_sequence\_to\_array(kd, edge, edges, vers);

adjlist\_graph<string, int>G(kd, vers, edges, d, e);

G.printVer();

cout << endl;

G.printAdjList();

cout << endl << endl;

if (G.U\_Judge\_Cir()) {

cout << "true";

}

else {

cout << "false";

}

delete[] d;

delete[] edge;

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

delete[] e[i];

}

delete[] e;

}

return 0;

}