#include <iostream>

#include <strstream>

#include <queue>

using namespace std;

// LList<T> {

template <class T>

struct elem\_link1

{

T inf;

elem\_link1<T> \*link;

};

template <class T>

class LList

{

public:

LList();

~LList();

LList(LList const &);

LList& operator=(LList const &);

bool empty() const;

void IterStart(elem\_link1<T>\* = NULL);

elem\_link1<T>\* Iter();

void ToEnd(T const &);

void InsertAfter(elem\_link1<T> \*, T const &);

void InsertBefore(elem\_link1<T> \*, T const &);

void DeleteElem(elem\_link1<T> \*, T &);

int DeleteAfter(elem\_link1<T> \*, T &);

int DeleteBefore(elem\_link1<T> \*, T &);

int length();

void concat(LList const &);

void reverse();

void print();

private:

elem\_link1<T> \*Start,

\*End,

\*Current;

void DeleteList();

void CopyList(LList<T> const &);

};

template <class T>

LList<T>::LList()

{

Start = End = Current = NULL;

}

template <class T>

LList<T>::~LList()

{

DeleteList();

}

template <class T>

LList<T>::LList(LList<T> const &r)

{

CopyList(r);

}

template <class T>

LList<T>& LList<T>::operator=(LList<T> const &r)

{

if (this != &r)

{

DeleteList();

CopyList(r);

}

return \*this;

}

template <class T>

void LList<T>::DeleteList()

{

if (Start)

{

elem\_link1<T> \*p;

while (Start)

{

p = Start;

Start = Start->link;

delete p;

}

End = NULL;

}

}

template <class T>

void LList<T>::CopyList(LList<T> const &r)

{

Start = End = NULL;

elem\_link1<T> \*p = r.Start;

while (p)

{

ToEnd(p->inf);

p = p->link;

}

}

template <class T>

bool LList<T>::empty() const

{

return Start == NULL;

}

template <class T>

void LList<T>::IterStart(elem\_link1<T> \*p)

{

if (p)

Current = p;

else

Current = Start;

}

template <class T>

elem\_link1<T>\* LList<T>::Iter()

{

elem\_link1<T> \*p = Current;

if (Current)

Current = Current->link;

return p;

}

template <class T>

int LList<T>::length()

{

int n = 0;

IterStart();

while (Iter())

n++;

return n;

}

template <class T>

void LList<T>::ToEnd(T const &x)

{

Current = End;

End = new elem\_link1<T>;

End->inf = x;

End->link = NULL;

if (Current)

Current->link = End;

else

Start = End;

}

template <class T>

void LList<T>::InsertAfter(elem\_link1<T> \*p, T const &x)

{

elem\_link1<T> \*q = new elem\_link1<T>;

q->inf = x;

q->link = p->link;

if (p == End)

End = q;

p->link = q;

}

template <class T>

void LList<T>::InsertBefore(elem\_link1<T> \*p, T const &x)

{

elem\_link1<T> \*q = new elem\_link1<T>;

\*q = \*p;

p->inf = x;

p->link = q;

if (End == p)

End = q;

}

template <class T>

int LList<T>::DeleteAfter(elem\_link1<T> \*p, T &x)

{

if (p == End)

return 0;

elem\_link1<T> \*q = p->link;

x = q->inf;

p->link = q->link;

if (End == q)

End = p;

delete q;

return 1;

}

template <class T>

void LList<T>::DeleteElem(elem\_link1<T> \*p, T &x)

{

if (p == Start)

{

x = p->inf;

if (Start == End)

{

Start = NULL;

End = NULL;

delete p;

}

else

{

Start = Start->link;

delete p;

}

}

else

{

elem\_link1<T> \*q = Start;

while (q->link != p)

q = q->link;

DeleteAfter(q,x);

}

}

template <class T>

int LList<T>::DeleteBefore(elem\_link1<T> \*p, T &x)

{

if (p == Start)

return 0;

elem\_link1<T> \*q = Start;

while (q->link != p)

q = q->link;

DeleteElem(q,x);

return 1;

}

template <class T>

void LList<T>::concat(LList<T> const &L)

{

elem\_link1<T> \*p = L.Start;

while (p)

{

ToEnd(p->inf);

p = p->link;

}

}

template <class T>

void LList<T>::reverse()

{

LList<T> L;

elem\_link1<T> \*p = Start;

if (p)

{

L.ToEnd(p->inf);

p = p->link;

while (p)

{

L.InsertBefore(L.Start, p->inf);

p = p->link;

}

}

\*this = L;

}

template <class T>

void LList<T>::print()

{

elem\_link1<T> \*p = Start;

while (p)

{

cout<< p->inf <<" ";

p = p->link;

}

cout<<"\n";

}

// LList<T> }

//--------------------------------------------------

// graph<T> {

template <typename T>

class graph

{ public:

void addTop(const T&);

void deleteTop(const T&);

void addRib(const T&, const T&);

void deleteRib(const T&, const T&);

bool top(const T&);

bool rib(const T&, const T&);

bool empty() const;

elem\_link1<T>\* point(const T&);

LList<T> vertexes();

void print();

private:

LList< LList<T> > g;

};

// включва а като връх на неявния граф

template <typename T>

void graph<T>::addTop(const T& a)

{ // създаване на линеен списък, съдържащ елемента a

LList<T> l;

l.ToEnd(a);

// включване на върха a към графа

g.ToEnd(l);

}

// изключва върха а неявния граф

template <typename T>

void graph<T>::deleteTop(const T& a)

{ // обхождане на графа с цел изтриване на всички

// ребра от произволен връх до върха a

g.IterStart();

elem\_link1<LList<T> >\* p = g.Iter(), \*r;

while(p)

{ p->inf.IterStart();

elem\_link1<T>\* q = p->inf.Iter();

if (rib(q->inf, a)) deleteRib(q->inf, a);

p = p->link;

}

// изтриване на линейния списък, представящ

// върха a и неговите наследници

g.IterStart();

elem\_link1<T>\* q;

do

{ r = g.Iter();

r->inf.IterStart();

q = r->inf.Iter();

} while(q->inf != a); // a е връх на графа

LList<T> x;

g.deleteElem(r, x);

}

// включва ребро от върха a до върха b на неявния граф

template <typename T>

void graph<T>::addRib(const T& a, const T& b)

{ // намиране на указател към върха a на графа

elem\_link1<T>\* q = point(a), // намира указател към върха a

\* p;

// включване на върха b в списъка от наследниците на върха a

p = new elem\_link1<T>;

//assert(p != NULL);

p->inf = b;

p->link = q->link;

q->link = p;

}

// изключва реброто от върха a до върха b на неявния граф

template <typename T>

void graph<T>::deleteRib(const T& a, const T& b)

{ g.IterStart();

elem\_link1<LList<T> > \*p;

elem\_link1<T> \*q;

do

{ p = g.Iter();

p->inf.IterStart();

q = p->inf.Iter();

} while (q->inf != a);

q = q->link;

// намиране на указател към наследника b на върха a

while(q->inf != b) q = q->link;

T x;

p->inf.deleteElem(q, x);

}

// проверява дали a е връх на неявния граф

template <typename T>

bool graph<T>::top(const T& a)

{ if(g.empty()) return false;

g.IterStart();

elem\_link1<LList<T> > \*p = g.Iter();

elem\_link1<T> \*q;

do

{ p->inf.IterStart();

q = p->inf.Iter();

p = p->link;

} while(q->inf != a && p);

return q->inf == a;

}

// проверява дали има ребро от върха a до върха b

template <typename T>

bool graph<T>::rib(const T& a, const T& b)

{ elem\_link1<T>\* p = point(a); // намира указател към върха a

p = p->link;

while (p && p->inf != b) p = p->link;

return p != NULL;

}

// проверява дали неявният граф е празен

template <typename T>

bool graph<T>::empty() const

{ return g.empty();

}

// намира указател към върха a на графа

template <typename T>

elem\_link1<T>\* graph<T>::point(const T& a)

{ g.IterStart();

elem\_link1<LList<T> > \*p;

elem\_link1<T> \*q;

do

{ p = g.Iter();

p->inf.IterStart();

q = p->inf.Iter();

} while(q->inf != a);

return q;

}

// връща списък от върховете на неявния граф

template <typename T>

LList<T> graph<T>::vertexes()

{ LList<T> l;

g.IterStart();

elem\_link1<LList<T> > \*p = g.Iter();

while(p)

{ p->inf.IterStart();

elem\_link1<T>\* q = p->inf.Iter();

l.ToEnd(q->inf);

p = p->link;

}

return l;

}

// извежда неявния граф

template <typename T>

void graph<T>::print()

{ g.IterStart();

elem\_link1<LList<T> > \*p = g.Iter();

while (p)

{ p->inf.print();

p = p->link;

}

cout << endl;

}

// graph<T> }

//----------------------------------------------------

//

// Задача 0: да се зареди граф от поток

//

// Избираме следното представяне:

// първо се записва броя N на върховете

// после за всеки връх k следва следната информация:

// - брой на наследниците M[k]

// - спиисък на наследниците a[1] a[2] ... a[M[k]]

//

// в горното представяне се предполага, че стойностите

// на върховете са със стойности 1,2,...,N

template<typename T>

void createGraph(graph<T> & g, istream & is)

{

int ntops; is >> ntops;

for(size\_t k=1; k!=ntops+1; ++k)

g.addTop(k);

for(size\_t k=1; k!=ntops+1; ++k)

{

int top; is >> top;

size\_t nribs; is >> nribs;

for(size\_t l=0; l<nribs; l++)

{

int ribOtherEnd; is >> ribOtherEnd;

g.addRib(top,ribOtherEnd);

cerr << "adding rib (" << top <<"," << ribOtherEnd <<")" << endl;

}

}

}

// Задача 0,5 : да се състави функция,

// която проверява дали една стойности а е

// елемент от списъка l

template<typename T>

bool isMember(T a, LList<T> l)

{

l.IterStart();

while(elem\_link1<T> \* p = l.Iter())

if(p->inf == a)

return true;

return false;

}

//

// 4: Да се състави функция, която

// връща списък на родителите на върха child

// в графа g

//

template<typename T>

LList<T> getParents(T child, graph<T> g)

{

LList<T> rtn;

LList<T> l = g.vertexes();

l.IterStart();

while(elem\_link1<T> \* p = l.Iter())

if(g.rib(p->inf,child))

rtn.ToEnd(p->inf);

return rtn;

}

//

// 5: Път в граф

//

template<typename T>

LList<T> DFS(T start, T end, graph<T> g, LList<T> pathSoFar=LList<T>())

{

pathSoFar.ToEnd(start);

if(start==end)

{

return pathSoFar;

}

// следва неефективна, но проста реализация на намиране

// на всички наследници на връх

LList<T> l = g.vertexes();

l.IterStart();

while(elem\_link1<T> \* p = l.Iter())

if(g.rib(start,p->inf))

if(!isMember(p->inf,pathSoFar))

{

LList<T> path=DFS(p->inf,end,g,pathSoFar);

if(!path.empty())

return path;

}

return LList<T>();

}

// следващите задачи са за домашно

// 5,5: Цикъл в граф, съдържащ връх

template<typename T>

LList<T> Cycle(T top, graph<T> g)

{

// пътища от всички наследници на top до top

}

//

// 6: Всички пътища в граф, които започват от даден връх

//

template<typename T>

LList<T> DFSall(T start, T end, graph<T> g, LList<T> pathSoFar=LList<T>(), LList<LList<T>> & allPaths=LList<LList<T>>())

{

// вместо да връщаме пътя, го прибавяме в allPaths

// забележка : на по-стари реализации на езика

// трябва да се пише LList<LList<T> > вместо LList<LList<T>>

}

// 6,5: Всички цикли в граф, които съдържат връх,

// а елементите им са четни числа и при това

// разликата между максималния и минималния

// елемент не е с повече от 30% по-голяма от

// средноаритметичната стойност на неотрицателните

// стойности на върхове във цикъла ;)

// 7: Въпрос(\*) : в метода deleteTop на graph се вика метод

// deleteElem на LList<T>, а истинското име на метода

// е DeleteElem. Въпреки това, тази програма се компилира

// без грешка. Защо?

int main()

{

char cgraph[] = "9 "

"1 1 2 "

"2 3 3 4 5 "

"3 1 6 "

"4 1 7 "

"5 2 8 9 "

"6 1 7 "

"7 1 8 "

"9 0 ";

graph<int> g;

istrstream iss(cgraph);

createGraph(g,iss);

cout << isMember(1,g.vertexes()) << endl;

cout << isMember(12,g.vertexes()) << endl;

cout << "---------------------" << endl;

getParents(8,g).print();

cout << "---------------------" << endl;

DFS(2,8,g).print();

cout << "---------------------" << endl;

return 0;

}