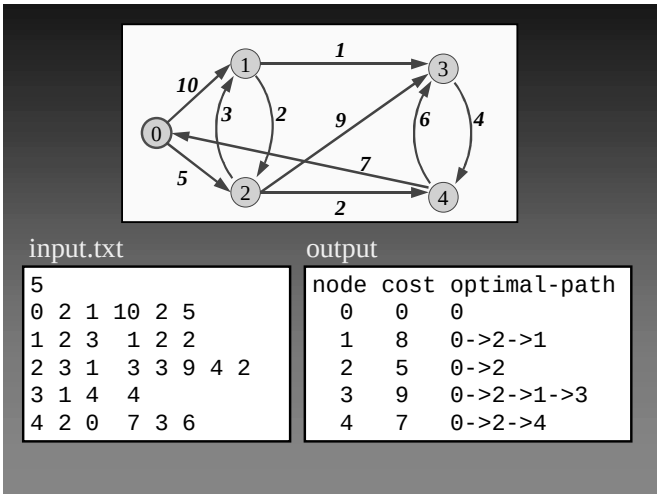
1. Feladat

Határozzuk meg egy nem-negatív értékekkel élsúlyozott irányított gráfban egy adott csúcsból kiinduló legolcsóbb utakat és azoknak költségét a gráf valamennyi csúcsára! A gráf csúcsait 0-val kezdődően sorszámozzuk; a 0-dik az a kitűntetett csúcs, amelyből kivezető optimális utakat keressük.
A gráfot szomszédsági listákkal adjuk meg egy szöveges fájlban. A fájl első sora a gráf csúcsainak számát, a további sorok az azon csúcsok szomszédsági listáját tartalmazzák (egy sor egy szomszédsági lista), amelyekből legalább egy él vezet ki. Minden szomszédsági lista szóközökkel elválasztott számokat tartalmaz: első szám annak a csúcsnak a sorszáma, amelyből kivezető éleket fel akarjuk itt sorolni; ezt követi a kivezető élek száma; majd minden kivezető élre páronként az él végcsúcsának sorszáma és a súlya.



input.txt

```
5
0 2 1 10 2 5
1 2 3  1 2 2
2 3 1  3 3 9 4 2
3 1 4  4
4 2 0  7 3 6
```

output

```
node cost optimal-path
  0   0   0
  1   8   0->2->1
  2   5   0->2
  3   9   0->2->1->3
  4   7   0->2->4
```

Absztrakt megoldás

Specifikáció:

$A = G_g \times vekt(R)_d \times vekt(Z)_p$ $\quad G=rec(V,E)$

$B = G_{g'}$

$Q=(g=g')$

$R=(g=g' \wedge s=0 \wedge d[s]=0 \wedge p[s]=nil \wedge$
$\forall n \in V-\{s\}$: ha van $s \rightarrow n$ út akkor
$d[n]$ = a legolcsóbb $s \rightarrow n$ út költsége $\wedge$
$p[n] = n$ szülője egy legolcsóbb $s \rightarrow n$ úton
különben $d[n]=\infty \wedge p[n]=nil$)

Absztrakt program:

```
for ∀n∈g.V\s-re do
      d[n]:=∞; p[n]:=nil
      Betesz(h,n,d[n])
enddo
d[s]:=0; p[s]:=nil
Betesz(h,s,d[s])

while not ?Üres(h) do
      n:=Kivesz_Min(h)
      for ∀m∈Utódok(n)-re do
            if ?Eleme(h,m) and d[m]>d[n]+c(n,m)then
                  d[m]:=d[n]+c(n,m)
                  p[n]:=m
                  Módosít(h,n,d[n])
            endif
      enddo
enddo
```

Absztrakt típusok

Elsőbbségi sor típus

Specifikáció. Az elsőbbségi sorban egy gráf (*0*-tól *db-1*-ig számozott) csúcsait helyezzük el a megfelelő költségértékkel. Az elsőbbségi sor maximális mérete (`size`) tehát a gráf csúcsainak száma. A *h* elsőbbségi sorra vonatkozó műveletek:

`?Üres()`	~	megvizsgálja, hogy az elsőbbségi sor üres-e;
`?Eleme(m)`	~	megvizsgálja, benne van-e az *m* csúcs a sorban;
`Betesz(n,c)`	~	az *n* csúccsal és a *c* költségértékkel bővíti a sort;
`Módosít(n,d[n])`	~	az *n* csúcs költségértékét *c* –re változtatja a sorban;
`Kivesz_Min()`	~	kiveszi a sor legkisebb költségű csúcsát.

Készítsünk osztálysablont (`Numbered_Priority_Queue`), amelyből az itt specifikált elsőbbségi sor példányosítható! Ebben a sorban olyan elemeket akarunk tárolni, amelyeknek nemcsak értékük, hanem egy azonosítójuk (kulcsuk) is van. A kulcsokról feltehető, hogy azok a `[0..size-1]` intervallumba esnek. A sorban egy adott kulcsú elemből több nem lehet. Vezessük be az adott kulcsú elem értékét megváltoztató műveletet.

Reprezentáció. Induljunk ki a mát ismert (11. fejezet) elsőbbségi sorából, és vegyük sorra milyen változtatásokat kell azon elvégeznünk. Az, hogy kulcsok segítségével meg tudjuk megkülönböztetni egymástól az elsőbbségi sorba bekerült elemeket, kizárólag az `Element` paraméter helyébe írt típuson múlik. Ügyelni kell arra, hogy a `>` operátor az elemeket értékük alapján hasonlítsa össze, az `==` operátor viszont a kulcsok szerint. Egy ilyen felhasználói típust az alábbi osztálysablonból nyerhetünk:

```
template <class KeyType, ValueType>
class Keyed_Element{
public:
    Keyed_Element(){};
    Keyed_Element(const KeyType& a, const ValueType& b)
                 :key(a),value(b){};
    bool operator==(const Keyed_ Element& r) const
                   {return key==r.key;}
    bool operator>(const Keyed_ Element& r) const
                   {return value>r.value;}
    bool operator<(const Keyed_ Element& r) const
                   {return value<r.value;}
    KeyType Key()     const { return key ;}
    ValueType Value() const { return value ;}
private:
    KeyType   key;
    ValueType value;
};
```

Ennek segítségével definiálható az a csúcs-érték párokat tartalmazó típus, amellyel az elsőbbségi sor példányosítható. (Itt a `Node` lényegében egy egész típus.)

```
typedef Keyed_Element<Node, TypeReversed<float> > Item;
```

A többi változtatáshoz azonban már módosítanunk kell az elsőbbségi sor definícióján is. Az új (`Numbered_Priority_Queue`) osztálysablont természetesen származtatással hozzuk létre, hiszen ez egy speciális `Unique_Priority_Queue` osztálysablon lesz. Ezzel máris biztosítottuk azt, hogy egy adott kulcsú elem ne kerülhessen többször a sorba. Természetesen ki kell egészíteni ezt az osztálysablont az adott kulcsú elem értékét megváltoztató `Modify` metódussal, de ezen kívül tehetünk néhány olyan változtatást is, amely hatékonyabb elsőbbségi sort eredményez. Kihasználhatjuk ugyanis, hogy a sorba bekerülő elemek kulcsai a `[0..size-1]` intervallumba esnek. Ha kiegészítjük a rejtett adattagokat egy `size` hosszú `pos` nevű tömbbel, akkor az tulajdonképpen a kulcsokkal lesz indexelve. Ezt a tömböt arra használjuk, hogy megtudjuk, hogy egy `k` kulcsú elem benne van-e a sorban. Ha benne van, a `pos[k]`mutassa meg a `k` kulcsú elem `vect` tömbben elfoglalt pozícióját, ha nincs legyen a `pos[k]=-1`. Ennek következtében `Unique_Priority_Queue`–nál bevezetett `Search` metódust sokkal hatékonyabban, a `pos[e.Key()]` kifejezéssel definiálhatjuk.

```
virtual int Search(const Element& e) const
                  {return pos[e.Key()];}
```

A `pos` tömböt a konstruktorban kell lefoglalni és kezdetben `-1`-ekkel feltölteni, miután meghívtuk az ős osztály konstruktorát. A `pos` tömb felszabadításáról a destruktor (a `vect` tömb felszabadításáról az ősosztály destruktora) gondoskodik:

```
Numbered_Priority_Queue(const int s)
:Unique_Priority_Queue<Element>(s)
{
    pos   = new int[size];
    for(int i = 0; i<size; i++) pos[i] = -1;
}
```

Ügyelni kell arra is, hogy amikor egy művelet során megváltozik egy elemnek a helye az elsőbbségi sort ábrázoló `vect` tömbben, akkor ezt a változást a `pos` tömbben is fel kell jegyezni. Szerencsére `vect` tömb megváltoztatását már a legelső változatban is csak a `Get` és `Put` metódus végezhette: elegendő ezeket újradefiniálni. (lásd. „inline” definíció)

Végül implementáljuk a `Modify` metódust. Ez először megkeresi `Search` metódus segítségével a módosítandó elemet az elsőbbségi sort ábrázoló `vect` tömbben. Ha nem találja, akkor kivételt dob. Ha megtalálja, attól függően, hogy a változtatás csökkenti vagy növeli az elsőbbségi sorban álló elem értékét, azt vagy az `Up`, vagy a `Down` metódus segítségével kell a megfelelő helyre mozgatni:

```
void Modify(const Element& e)
{
    int i = Search(e);
    if(i<0) trhow NOTFOUND;

    if( e>vect[i] ){
        Put(e,i);
        Up(i);
    }else{
        Put(e,i);
        Down(i);
    }
}
```

A `Numbered_Priority_Queue` osztálysablonja:

```
#include "unique_priority_queue.h"

template <class Element> class Numbered_Priority_Queue
                    : public Unique_Priority_Queue<Element> {
  public:
     Numbered_Priority_Queue(const int s);
     void Modify(const Element& e);
     ~Numbered_Priority_Queue(){delete[] pos;}

  protected:
       int* pos;

       virtual int Search(const Element& e) const
                   {return pos[e.Key()];}
       virtual Element Get(int i)
                   {pos[vect[i].Key()] = -1; return vect[i];}
       virtual void Put(const Element& e, int i)
                   {vect[i] = e; pos[e.Key()]=i; }
};
```

Helyezük el mind a `Numbered_Priority_Queue`, mind a `Keyed_Element` osztálysablont a `numbered_priority_queue.h`-ban. Ennek a főprogramba emelése után az alábbi módon példányosíthatjuk az elsőbbségi sorunkat.

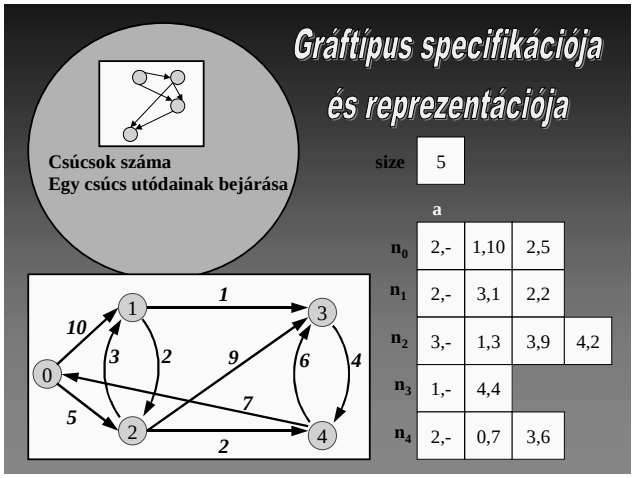
```
Numbered_Priority_Queue<Item> h(db);
```

Irtányított gráf típus

Specifikáció. A *g* gráfra vonatkozó műveletek:

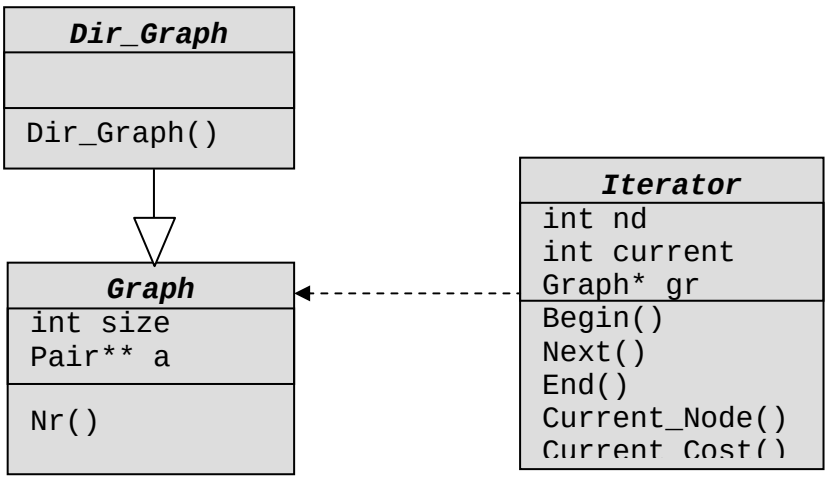
`Nr()`	~	a gráf csúcsainak száma
$\forall$`m`$\in$`Utódok(n)`	~	az *n* csúcs közvetlen utódainak bejárása
`c(n,m)`	~	az *n*-ből *m*-be vezető él súlya

Reprezentáció. Egy dinamikus kétdimenziós tömbben tárolt szomszédsági listákkal egy adott csúcs szomszédainak (`Utódok(n)`) bejárása (a hozzájuk vezető élek súlyával együtt) igen kényelmes. Ezért az alábbi ábrán látható reprezentációt választjuk.



Az utódok bejárására bejáró objektumot fogunk használni (lásd Sorozat típus c. előadás). Egy bejáró objektum létrehozásakor megadjuk azt az *n* csúcsot, amelynek az utódait akarjuk felsorolni, majd a szokásos (`Begin(), Next(), End()`) tagfüggvények segítségével végig mehetünk az *n* csúcs utódain. Az aktuális utód paramétereit a `Current_Node()` és a `Current_Cost()` tagfüggvénnyel kérdezhetjük majd le.

Ez a modell egyaránt alkalmas irányított és irányítatlan gráfok ábrázolására, ugyanis egy irányítatlan gráfbeli élet mindig lehet helyettesíteni két (egy oda- és egy visszafelé irányított) éllel. Ezért az eddig elmondottakat egy absztrakt gráf típusba foglaljuk, a feladat megoldásához szükséges irányított gráf típusához pedig úgy juthatunk, hogy egy olyan konstruktort készítünk, amely az adott formájú szöveges fájlból beolvasott irányított gráf adataival tölti fel a szomszédsági listákat.



graph.h:

```
//Absztrakt gráf típus

#ifndef GRAPH_H
#define GRAPH_H

typedef int Node;

class graph {
public:
    int Nr() const {return size;}
    virtual ~Graph()
    {
        for(Node n=0; n<size; n++){
            delete[] a[n];
        }
        delete[] a;
    }
protected:
    struct Pair{
      Node  n;
      float c;
    };

    int size;
    Pair** a;

    Graph(){};
    Graph(const Graph& g);
    Graph& operator=(const Graph& g);
};

#endif
```

dir_graph.h:

```
// Irányított gráf típus

#ifndef DIR_GRAPH_H
#define DIR_GRAPH_H
#include "graph.h"
#include <string>

class Dir_Graph : public Graph{
  public:
    Dir_Graph(const string name);
};

#endif
```

dir_graph.cpp:

```
#include "dir_graph.h"
#include <fstream>
#include <iostream>

Dir_Graph::Dir_Graph(const string name)
{
    ifstream f(name.c_str());
    if (f.fail()) {
        cout<<"The inputfile does not exsist!"<<endl;
        char ch; cin>>ch;
        exit(2);
    }

    Node n;
    f>>size;
    a = new pair* [size];
    for( n=0; n<size; n++) {
        a[n] = new pair [1];
        a[n][0].n=0;
    }

    int nr_succ;
    while (f>>n){
        f>>nr_succ;
        delete[] a[n];
        a[n] = new pair [nr_succ+1];
        a[n][0].n=nr_succ;
        for(int k=1; k<=nr_succ; k++) {
            f>>a[n][k].n>>a[n][k].c;
        }
    }
}
```

Implementáció

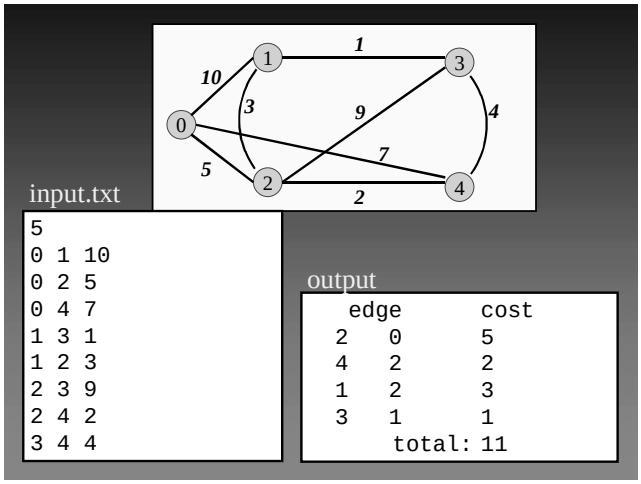
dijkstra.cpp:

```
/*************************************************************/
/* Feladat:Optimális utak keresése egy nem-negatív           */
/* költségekkel élsúlyozott, szomszédsági listákkal          */
/* megadott irányított gráfban                               */
/*************************************************************/

#include "dir_graph.h"
#include "numbered_priority_queue.h"
#include <iostream>
#include <iomanip>
using namespace std;

typedef Keyed_Element<Node, TypeReversed<float> > Item;
const int   nil      = -1;
const float infinite = 1000000.0;

int main()
{
  // Dijkstra's shortest paths algorithm

    Dir_Graph g("input.txt");
    int nr = g.Nr();
    Numbered_Priority_Queue<Item> h(nr);

    float d[nr];
    Node  p[nr];
    Node  n,m;
    float c;

    d[0] = 0; p[0] = nil;
    h.Insert(Item(0,d[0]));
    for(n = 1; n<nr; n++){
        d[n] = infinite; p[n] = nil;
        h.Insert(Item(n,d[n]));
    }

    while( !h.Empty() ) {
        n = h.Extract_Opt().Key();
        Dir_Graph::Iterator it(g,n);
        for(it.Begin(); !it.End(); it.Next()) {
            m = it.Current_Node();
            c = it.Current_Cost();
            if( In(Item(m,c)) && d[m]>d[n]+ c ) {
                d[m] = d[n]+c;
                p[m] = n;
                h.Modify(Item(d[m],m));
            }
        }
    }
```

```
    // Results

        cout << "node" << " cost" << " optimal path" << endl;
        for (n = 0; n<nr; n++) {
            cout << setw(3) << n << setw(5) << d[n];
            Node s[nr];
            int i = 0;
            m = n;
            while ( m!=nil ) {
                s[i++] = m;
                m = p[m];
            }
            cout << setw(4) << 0;
            for (i = i-2;i>=0;i--) {
                cout << "->" << s[i];
            }
            cout << endl;
        }
        char ch;
        cin>>ch;
        return 0;
    }
```

2. Feladat

Határozzunk meg egy nem-negatív értékekkel élsúlyozott irányítatlan gráfban egy minimális élsúlyú feszítőfát! A gráf csúcsait 0-val kezdődően sorszámozzuk. A gráfot éleinek felsorolásával adjuk meg egy szöveges fájlban. A fájl első sora a gráf csúcsainak számát, majd soronként az élek következnek. Minden sor három számot tartalmaz szóközökkel elválasztva: első két szám az él végcsúcsainak sorszáma, a harmadik szám az él súlya.



Absztrakt megoldás

Specifikáció:

$A = G \times E^*$ $\quad G=rec(V,E)$ $\quad B = G$

$g \quad e \qquad\qquad\qquad\qquad\qquad\qquad g'$

$Q=(g=g')$ $\quad R=(g=g' \wedge e=MFF(g))$

Absztrakt program:

```
for ∀n∈g.V-re do
      d[n]:=∞
      p[n]:=nil
      Betesz(h,n,d[n])
enddo
d[s]:=0
while not ?Üres(h) do
      n:=Kivesz_Min(h)
      for ∀m∈Utódok(n)-re do
            if ?Eleme(h,m) and d[m]>c(n,m) then
                  d[m]:=c(n,m)
                  p[n]:=nil
                  Módosít(h,n,d[n])
            endif
      enddo
enddo
```

Absztrakt típusok

Elsőbbségi sor típus

Specifikáció. A *h* elsőbbségi sorra vonatkozó műveletek:

?Üres()	~	megvizsgálja, hogy az elsőbbségi sor üres-e;
?Eleme(*m*)	~	megvizsgálja, benne van-e az *m* csúcs a sorban;
Betesz(*n*,*c*)	~	az *n* csúccsal és a *c* költségértékkel bővíti a sort;
Módosít(*n*,*d*[*n*])	~	az *n* csúcs költségértékét *c* –re változtatja a sorban;
Kivesz_Min()	~	kiveszi a sor legkisebb költségű csúcsát.

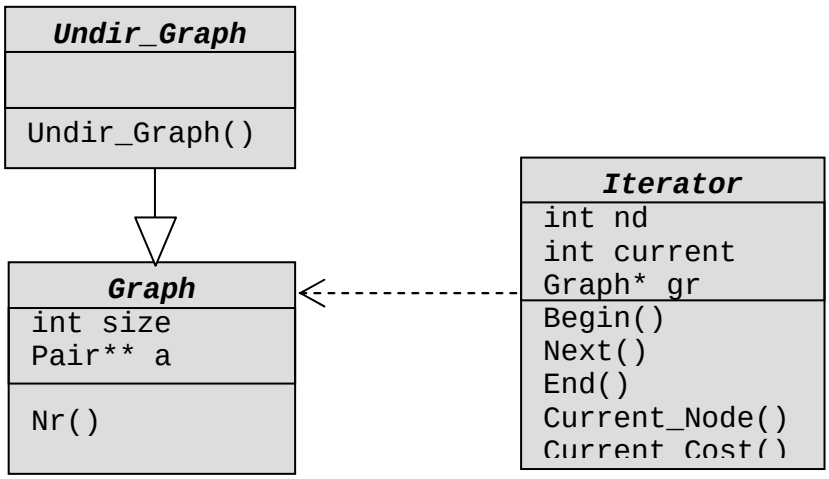
Reprezentáció. Lásd előbb.

Irtányítatlan gráf típus

Specifikáció. A *g* gráfra vonatkozó műveletek:

Nr()	~	a gráf csúcsainak száma
$\forall m \in$ *Utódok*(*n*)	~	az *n* csúcs közvetlen utódainak bejárása
c(*n*,*m*)	~	az *n*-ből *m*-be vezető él súlya

Reprezentáció. A korábban alkalmazott absztrakt gráf típusból származtathatjuk az irányítatlan gráf típust úgy, hogy olyan konstruktort készítünk, amely az adott formájú szöveges fájlból beolvasott irányítatlan gráf adataival tölti fel a szomszédsági listákat.



undir_graph.h:

```
#ifndef UNDIR_GRAPH_H
#define UNDIR_GRAPH_H

#include "graph.h"
#include <string>

class Undir_Graph : public Graph {
  public:
    Undir_Graph(const string name);
    ~Undir_Graph();
};

#endif
```

undir_graph.cpp:

```
#include "undir_graph.h"
#include <fstream>
#include <iostream>
const float infinite =1000000.0;

Undir_Graph::~Undir_Graph()
{
    for(Node n = 0; n<size; n++) {
        delete[] a[n];
    }
    delete[] a;
}

 Undir_Graph::Undir_Graph(const string name)
{
    ifstream f(name.c_str());
    if (f.fail()) {
        cout<<"The inputfile does not exsist!"<<endl;
        char ch; cin>>ch;
        exit(2);
    }

    float **b;
    Node n,m;

    f>>size;
    b = new float* [size];
    for(n = 0; n<size; n++){
        b[n] = new float [size];
        for(m = 0; m<size; m++) {
            b[n][m] = infinite;
        }
    }
    float c;
    while(f>>n>>m>>c){
        b[n][m]=b[m][n]=c;
    }
```

```
        a = new Pair* [size];
        for(n = 0; n<size; n++) {
            int s;
            for(m = 0, s = 0; m<size; m++){
                if(b[n][m]!=infinite) s++;
            }
            a[n] = new Pair [s+1];
            a[n][0].n = s;
            int k = 1;
            for(m = 0; m<=size; m++) {
                while(b[n][m]==infinite) m++;
                a[n][k].n = m;
                a[n][k].c = b[n][m];
                k++;
            }
        }

        for(n = 0; n<size; n++) {
            delete[] b[n];
        }
        delete[] b;
    }
```

Implementáció

prim.cpp:

```
/*************************************************************/
/* Feladat: Optimális súlyú feszítőfa keresése egy           */
/* nem-negatív élköltségekkel súlyozott, szomszédsági        */
/* listákkal megadott irányítatlan gráfban                   */
/*************************************************************/

#include "undir_graph.h"
#include "numbered_priority_queue.h"
#include <iostream>
#include <iomanip>
using namespace std;

typedef Keyed_Element<Node, TypeReversed<float> > Item;

const Node  nil      =-1;
const float infinite =1000000.0;

int main()
{
  // Prim's minimal spanning tree algorithm

    Undir_Graph g("input.txt");
    int nr = g.Nr();
    Numbered_Priority_Queue<Item> h(nr);

    float d[nr];
    Node  p[nr];
    Node  n,m;
    float c;

    d[0] = 0; p[0] = nil;
    h.Insert(Item(0,d[0]));
    for(n = 1; n<nr; n++){
        d[n] = infinite; p[n] = nil;
        h.Insert(Item(n,d[n]));
    }

    while ( !h.Empty() ) {
        n = h.Extract_Opt().Key();
        Undir_Graph::Iterator it(g,n);
        for(it.Begin(); !it.End(); it.Next()) {
            m = it.Current_Node();
            c = it.Current_Cost();
            if ( h.In(Item(m,c)) && c<d[m] ) {
                d[m] = c; p[m] = n;
                h.Modify(Item(m,d[m]));
            }
        }
    }
```

```
    // Results

      float s=0;
      cout<<"node"<<" node"<<" cost"<<endl;
      for (n = 1; n<nr; n++) {
         s+=d[n];
         cout << setw(3) << n << setw(5) << p[n]
              << setw(5) << d[n] << endl;
      }
      cout << setw(11) << "total: " << s << endl;

      char ch;
      cin>>ch;
      return 0;
  }
```