Programozás alapjai II.
(12. ea) **C++**

visszalépéses (backtrack) algoritmusok
táblás játékok

Szeberényi Imre, Somogyi Péter
BME IIT

<szebi@iit.bme.hu>

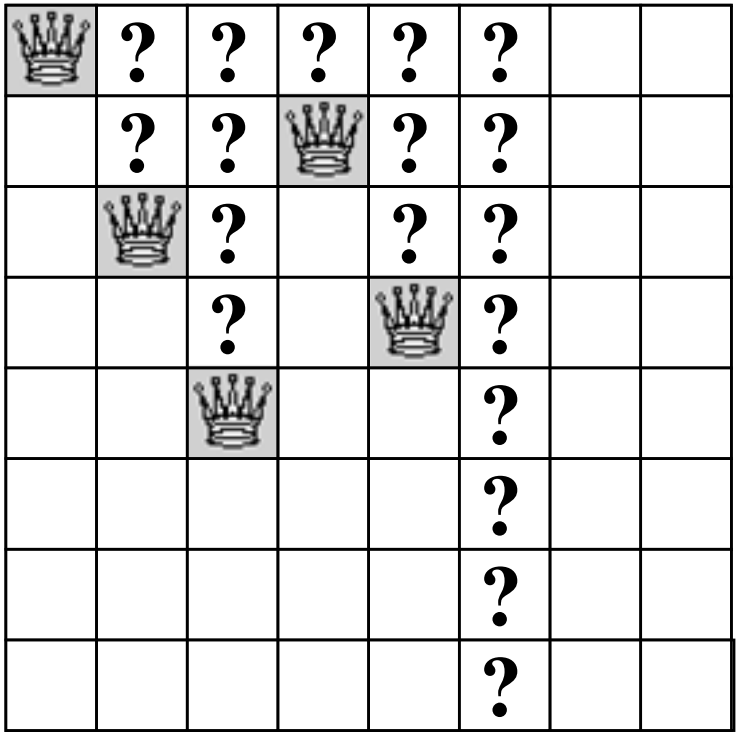
MŰEGYETEM 1782

Nyolc királynő probléma

- Helyezzünk el nyolc királynőt úgy egy sakktáblán, hogy azok ne üssék egymást!
- Egy megoldást keresünk nem keressük az összes lehetséges elhelyezést.

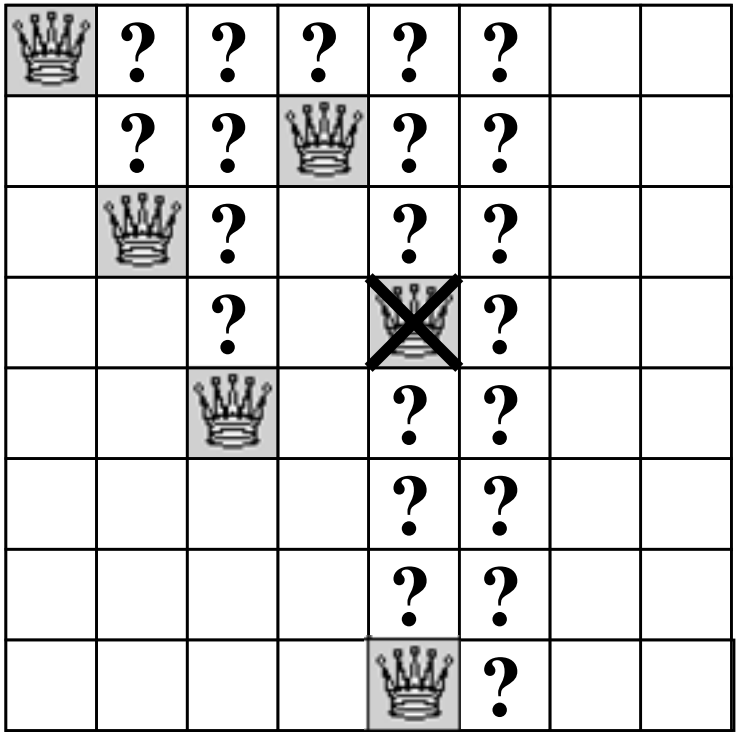
Probálgatás

- Egy oszlopban csak egy királynő lehet, ezért oszloponként próbálgatunk.



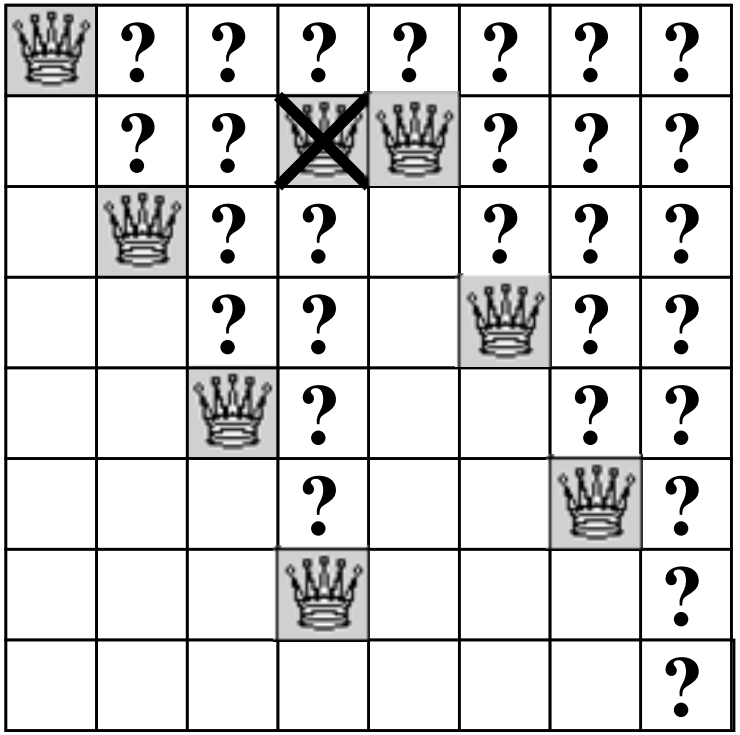
Visszalépés (back track)

- A 24. lépésben visszalépünk, és levesszük a korábban már elhelyezett királynőt.



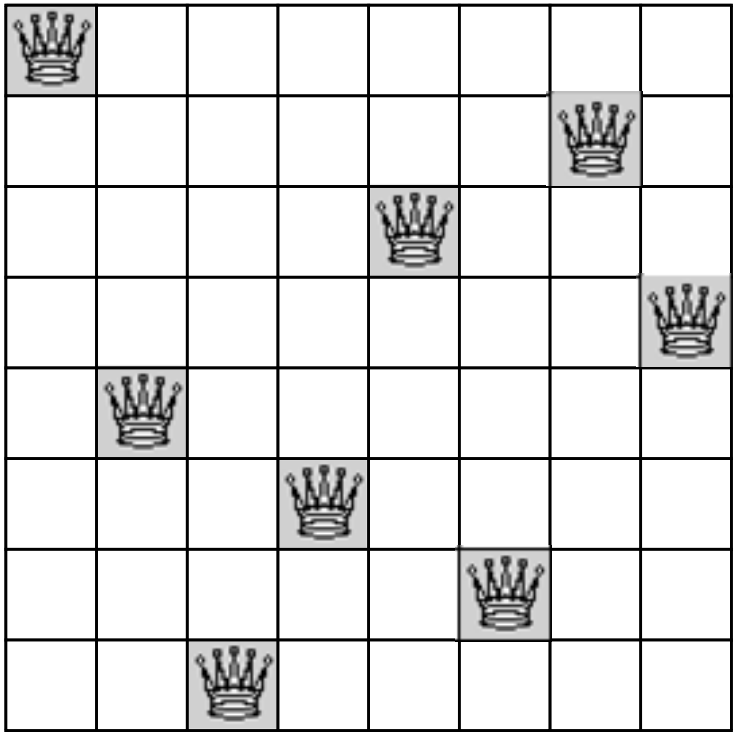
Visszalépés és újból próba

- A 36. lépésben ismét vissza kell lépni.
- A 60. lépés ismét kudarcba fullad.



Visszalépés és újból próba

- Egy lehetséges megoldást a 876. lépésben kapunk.



Hogyan modellezhető ?

- Okos tábla, buta királynő
  - A sakktábla ismeri a szabályokat és nyilvántartja a királynők helyzetét.
  - A királynő lényegében nem csinál semmit.
- Okos királynő, buta tábla
  - A királynő ismeri a szabályokat és nyilvántartja a saját pozícióját.
  - A sakktábla nem tud semmit.
- Okoskodó tábla, okos királynő

Okos tábla metódusai:

- Szabad – adott pozícióra lehet-e lépni
- Lefoglal – adott pozíciót lefoglalja
- Felszabadít – adott pozíciót felszabadítja
- Rajzol – kirajzolja a pillanatnyi állást
- Probal – elhelyezi a királynőket

Metódus spec. - Szabad

bool Szabad(int sor, int oszlop);

- megvizsgálja, hogy az adott helyre lehet-e királynőt tenni.
- bemenet:
  - sor – sor (1..8)
  - oszlop – oszlop (1..8)
- kimenet:
  - true – a pozíció szabad
  - false – ütésben van a királynő

Metódus spec. - Lefoglal

void Lefoglal(int sor, int oszlop);

- Lefoglalja az adott pozíciót.
- bemenet:
  - sor – sor (1..8)
  - oszlop – oszlop (1..8)

Metódus spec. - Felszabadit

void Felszabadit(int sor, int oszlop);

- Felszabadítja az adott pozíciót.
- bemenet:
  - sor – sor (1..8)
  - oszlop – oszlop (1..8)

Metódus spec. - Rajzol

void Rajzol(int sor, int oszlop);

- Kirajzolja a táblát.
  Az aktuális sor, oszlop pozícióba ? jelet tesz
- bemenet:
  - sor – sor (1..8)
  - oszlop – oszlop (1..8)

Metódus spec. - Probal

bool Probal(int oszlop);

- A paraméterként kapott oszlopba és az azt követő oszlopokba megpróbálja elhelyezni a királynőket
- bemenet:
  - oszlop sorszáma (1..8)
- kimenet:
  - true - ha sikerült a 8. oszlopba is.
  - false - ha nem sikerült

Implementáció - Probal

```
bool Probal(int oszlop) {
  int siker = 0, sor = 1;
  do {
    if (Szabad(sor, oszlop)) {
      Lefoglal(sor, oszlop)
      if (oszlop < 8) siker = Probal(oszlop+1);
      else siker = 1;
      if (!siker) Felszabadit(sor, oszlop);
    }
    sor++;
  } while (!siker && sor <= 8);
  return(siker);
}
```

következő oszlopba

nem sikerült, visszalép

Adatszerkezet (mit kell tárolni?)

- Adott sorban van-e királynő
  - 8 sor: vektor 1..8 indexszel
- Adott átlóban van-e királynő
  - főátlóval párhuzamos átlók jellemzője, hogy a sor-oszlop = álladó (-7..7-ig 15 db átló)
  - mellékátlóval párhuzamos átlók jellemzője, hogy a sor+oszlop = álladó (2..16-ig 15 db átló)

Átlók tárolása

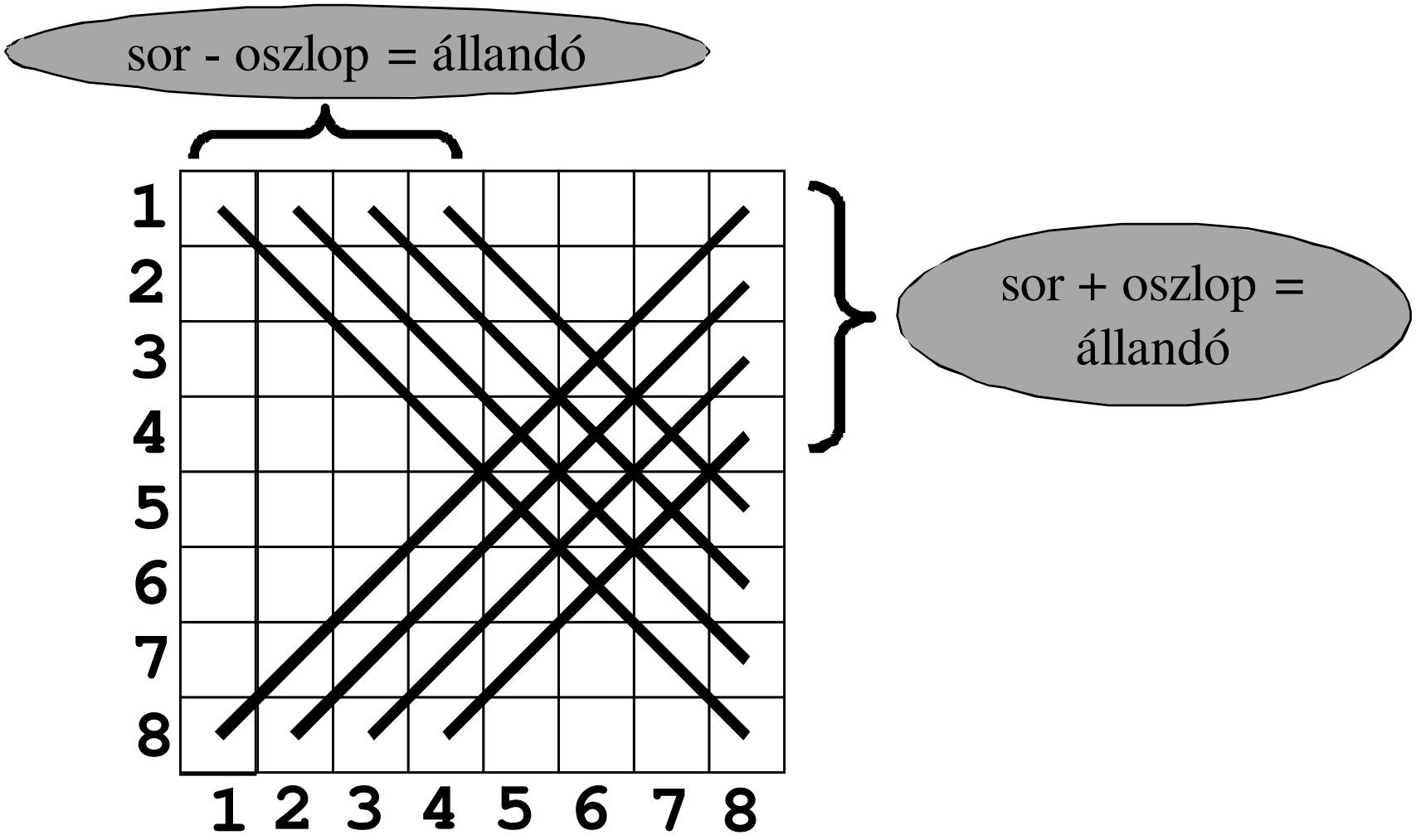


Tabla osztály megvalósítása

```
class Tabla {
   bool sor[8];              // sorok foglaltsága
   bool f_atlo[15];          // főátlók (s-o)
   bool m_atlo[15];          // mellékátlók (s+o)
   int  tab[8];              // kiíráshoz kell
   int probalkozas;          // kiíráshoz kell

   bool& Sor(int i) { return sor[i-1]; }
   bool& Fo(int s, int o) { return f_atlo[s-o+7]; }
   bool& Mellek(int s, int o) { return m_atlo[s+o-2]; }
   int&  Tab(int s) { return tab[s-1]; }
```

Tabla osztály megvalósítása /2

```
bool Szabad(int s, int o) {
    return Sor(s) && Fo(s, o) && Mellek(s, o);
}
void Lefoglal(int s, int o) {
    Sor(s) = Fo(s, o) = Mellek(s, o) = false;
    Tab(s) = o;
}
void Felszabadit(int s, int o) {
    Sor(s) = Fo(s, o) = Mellek(s, o) = true;
    Tab(s) = 0;
}
```

Tabla osztály megvalósítása /3

```
public:
    Tabla();
    bool Probal(int oszlop);
    void Rajzol(int sor, int oszlop);
};
Tabla::Tabla()  { probalkozas = 0;
    for (int i = 0; i < 15; i++) {
        f_atlo[i] = m_atlo[i] = true;
        if (i < 8) {   sor[i] = true; tab[i] = 0;  }
    }
}
```

Tabla osztály megvalósítása /4

```
bool Tabla::Probal(int oszlop) {
   int sor = 1; bool siker = false;
   do { Rajzol(sor, oszlop);
      if (Szabad(sor, oszlop)) {
         Lefoglal(sor, oszlop);
         if (oszlop < 8)      siker = Probal(oszlop+1);
         else                 siker = true;
         if (!siker)          Felszabadit(sor, oszlop);
      }
      sor++;
   } while (!siker && sor <= 8);
   return siker;
}
```

Működés megfigyelése

- A minden próbálkozást számolunk:
  - ➔ kell egy számláló: probalkozas
- Minden próbálkozást kirajzolunk:
  - * a már elhelyezett királynők helyére
  - ? a próba helyére
  - ➔ kell tárolni a táblát: tab változó

Implementáció - kiir

```
void Tabla::Rajzol(int sor, int oszlop) {
   cout.width(3);  cout << ++probalkozas << ".\n";
   for (int i = 1; i <= 8; i++) {
      cout.width(5);  cout << 9-i << '|';
      for (int j = 1; j <= 8; j++) {
         if (i == sor && j == oszlop)  cout << "?|";
         else if (Tab(i) == j)         cout << "*|";
         else                          cout << " |";
      }
      cout << endl;
   }
   cout << "      A B C D E F G H\n\n";
}
```

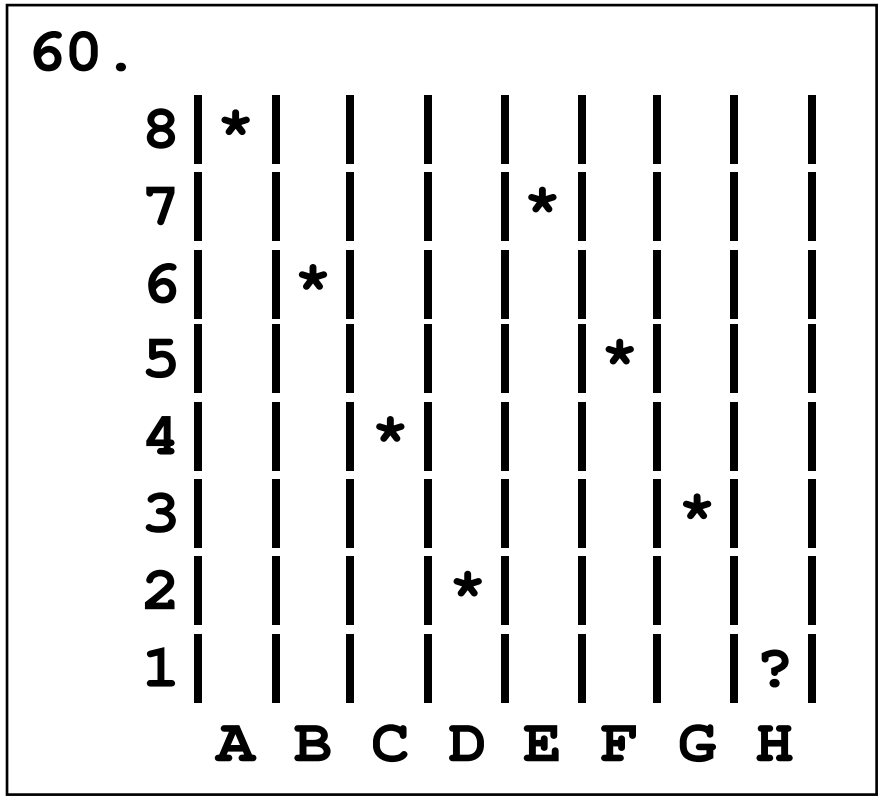
Főprogram

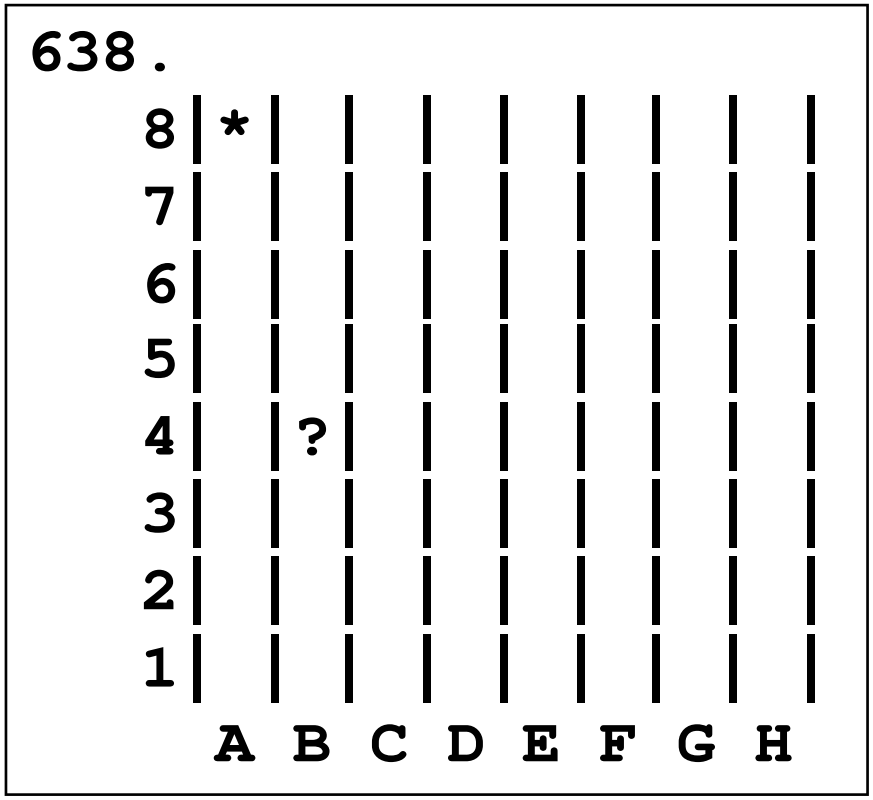
```
int main()
{
   Tabla t;

   if (t.Probal(1)) {
      t.Rajzol(8, 0);          // végleges áll. kirajzolása
      cout << "Siker";
   } else
      cout << "Baj van";
}
```

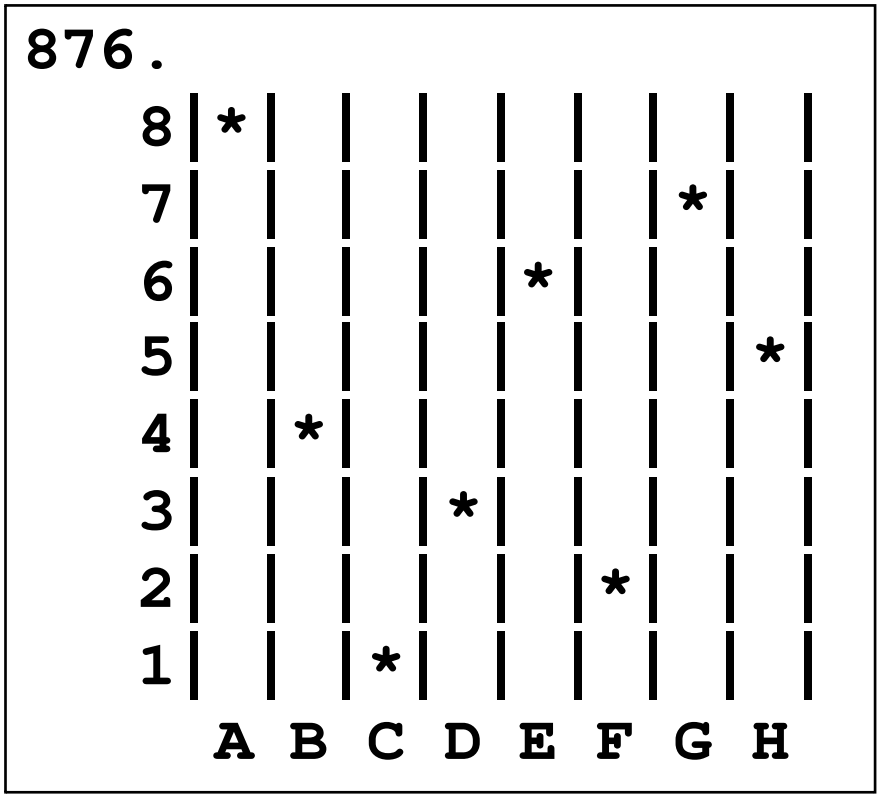
https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_12 → kiralyno

Eredmények





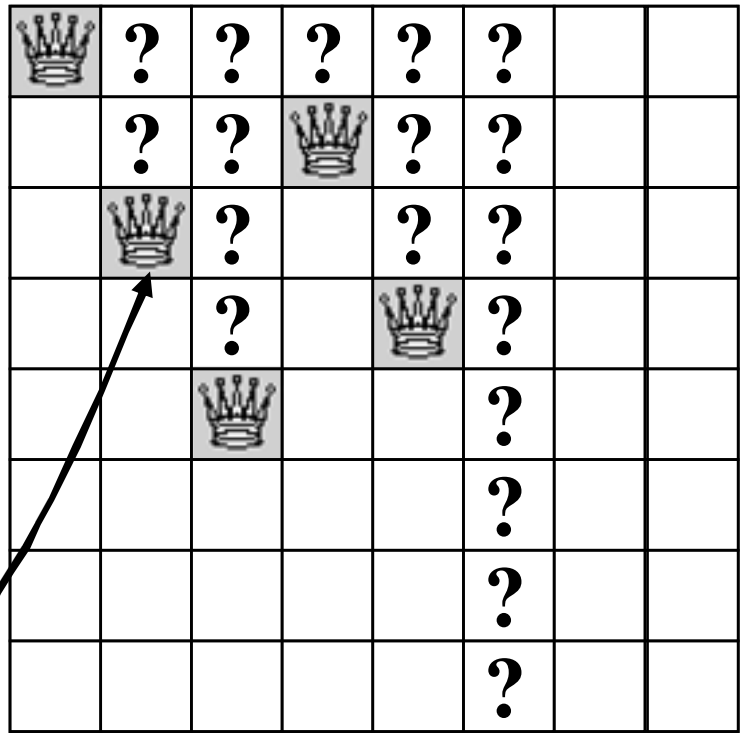
Eredmények /2



Okos királynő

```
Kiralyno *babu = NULL;
  for (int i = 1; i <= 8; i++) {
    babu = new Kiralyno(i, babu);
    babu->Helyezkedj();
}
```

```
bool Kiralyno::Helyezkedj() {
  while (szomszed != NULL &&
    szomszed->Utesben(sor, oszlop))
      if (!Lep()) return false;
  return true;
}
```



Okos királynő metódusai:

- Utesben – Ellenőrzi, hogy az adott pozíció ütésben áll-e.
- Lep – Előre lép az adott oszlopban, ha nem tud, akkor az oszlop elejére áll és a tőle balra levőt lépteti.
- Helyezkedj – Jó helyet keres magának.
- Kiír – kiírja a pozícióját.

Metódus spec. - Utesben

bool Utesben(int sor, int oszlop);

- megvizsgálja, hogy az adott pozíció ütésben áll-e a saját pozíciójával, ha nem, akkor azonos paraméterekkel meghívja a szomszéd Utesben() tagfüggvényét.
- bemenet:
  - sor – sor (1..8)
  - oszlop – oszlop (1..8)
- kimenet:
  - true – a pozíció ütésben áll
  - false – nincs ütési helyzet

Metódus spec. - Lep

void Lep();

- Előre lép az adott oszlopban, ha nem tud, akkor az oszlop elejére áll és meghívja a tőle balra levőt királynő Lep() tagfüggvényét.
- bemenet: -
- kimenet:
  - true – tudott lépni
  - false – nem tudott lépni

Metódus spec. - Helyezkedj

void Helyezkedj();

- Megvizsgála a saját pozícióját, hogy megfelelő-e. Ha ütésben áll, akkor meghívja Lep() tagfüggvényét. Ha tudott lépni, akkor ismét megvizsgálja a saját pozícióját.
- bemenet: -
- kimenet:
  - true – tudott megfelelő helyet találni magának
  - false – nem tudott megfelelő helyet találni

Metódus spec. - Kiir

void Kiir();

– Kiírja a saját és a szomszédok pozícióját.

Kiralyno osztály megvalósítása

```
class Kiralyno {
   int sor;                    // sor 1-8
   int oszlop;                 // oszlop 1-8
   Kiralyno *szomszed;         // szomszéd pointere
public:
   Kiralyno(int o, Kiralyno *sz): szomszed(sz),
                                  oszlop(o) { sor = 1; }
   bool Utesben(int s, int o);
   bool Lep();
   bool Helyezkedj();
   void Kiir();
};
```

Kiralyno osztály megvalósítása/2

```
bool Kiralyno::Utesben(int s, int o) {
    int d = oszlop - o;          // oszlop differencia
    if (sor == s || sor + d == s || sor - d == s)
        return true;
    else if (szomszed != NULL)   // ha van szomszéd
        return szomszed->Utesben(s, o); // akkor azzal is
    else
        return false;
}
```

Kiralyno osztály megvalósítása/3

```
bool Kiralyno::Lep() {
    if (sor < 8) {
        sor++;  return true;        // tudott lépni
    }
    if (szomszed != NULL)   { // nem tud, van szomsz.
        if (!szomszed->Lep())                   return false;
        if (!szomszed->Helyezkedj())        return false;
    } else {
        return false;
    }
    sor = 1;
    return true;
}
```

Kiralyno osztály megvalósítása/4

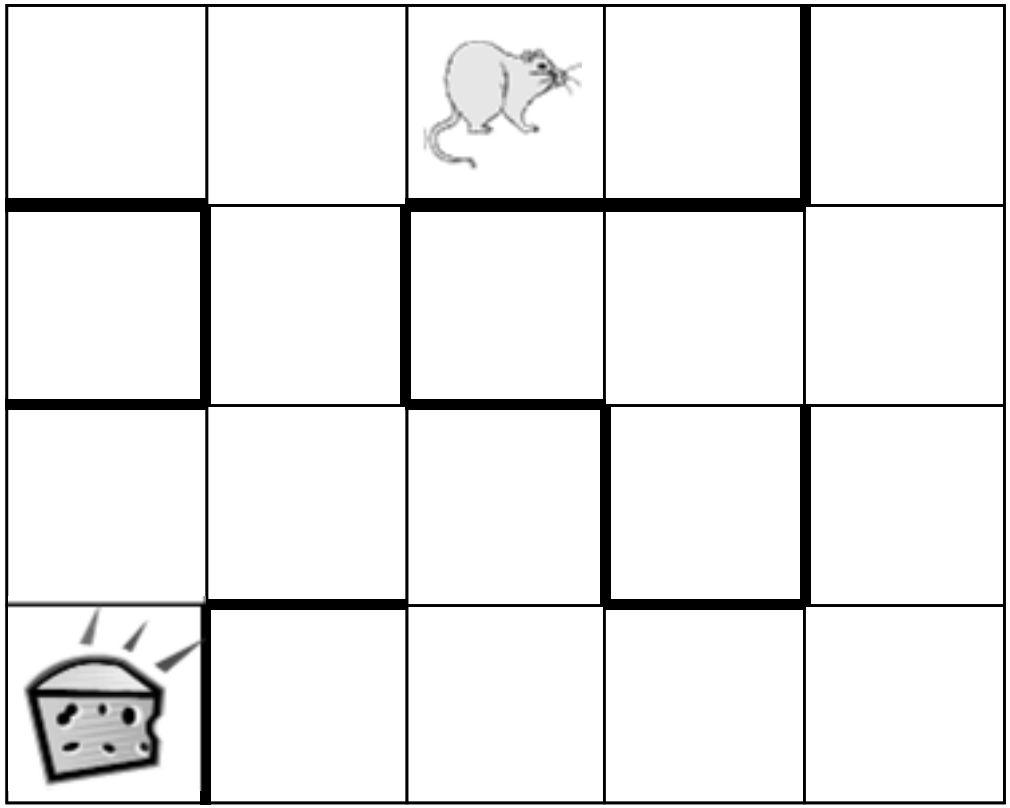
```
bool Kiralyno::Helyezkedj() {
   while (szomszed != NULL &&
          szomszed->Utesben(sor, oszlop))
      if (!Lep())
         return false;
   return true;
}
void Kiralyno::Kiir() {
   if (szomszed != NULL)
      szomszed->Kiir();
   cout << (char)(oszlop-1+'A') << 9-sor << endl;
}
```

Főprogram

```
int main()
{
   Kiralyno *babu = NULL;
   for (int i = 1; i <= 8; i++) {
      babu = new Kiralyno(i, babu);
      babu->Helyezkedj();
   }
   babu->Kiir();
}
```

https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_12 → kiralyno2

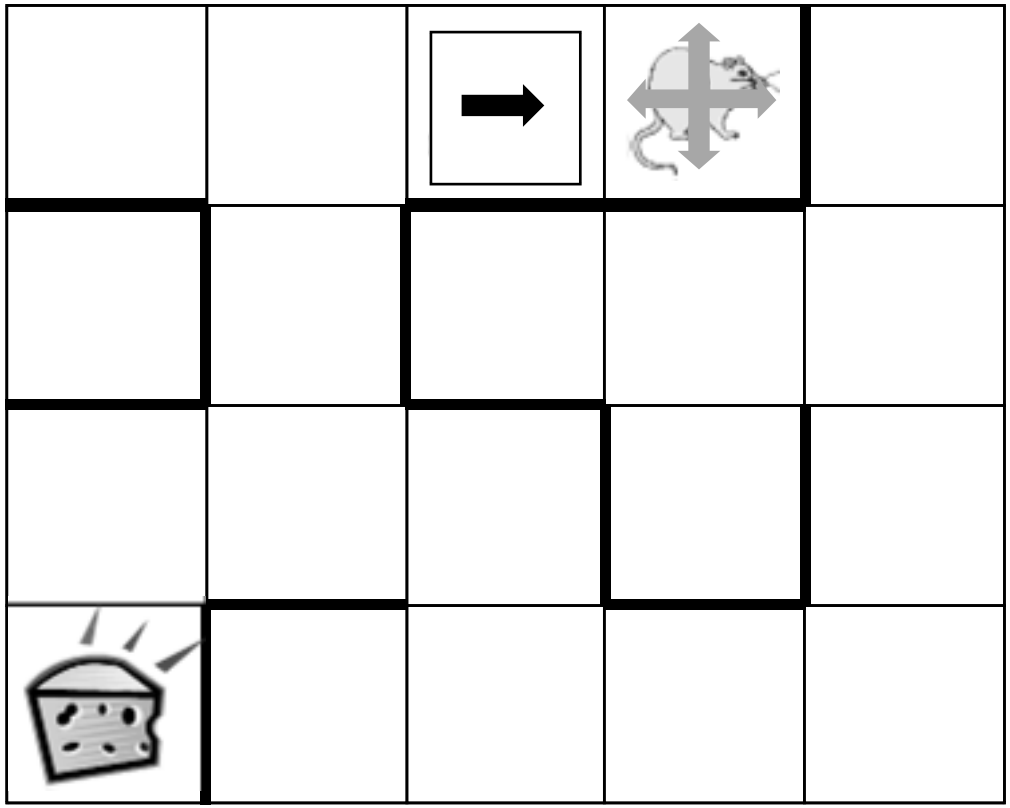
Megtalálja-e az egér a sajtot?



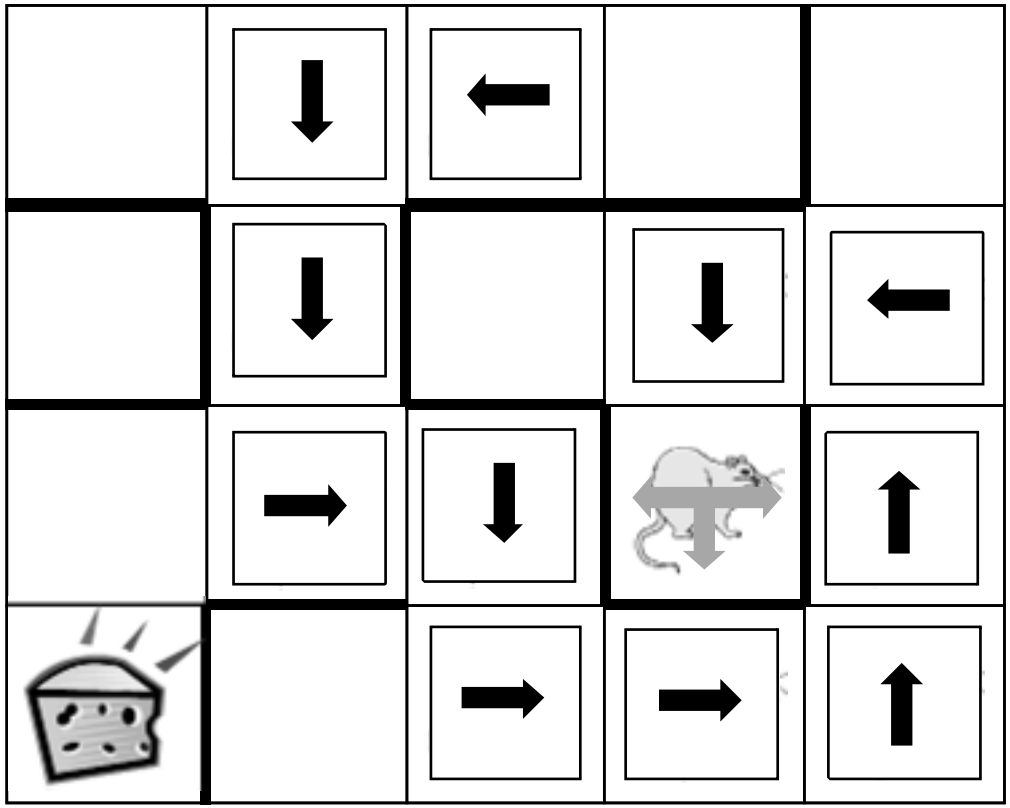
Egér algoritmusa

- Minden cellából először **jobbra**, majd **le**, ezután **balra** és végül **fel** irányokba indul.
- Minden cellában otthagyja a nyomát, azaz azt, hogy onnan merre indult legutoljára.
- Csak olyan cellába lép be, ahol nincs "nyom".
- Ha már nem tud hova lépni, visszalép.

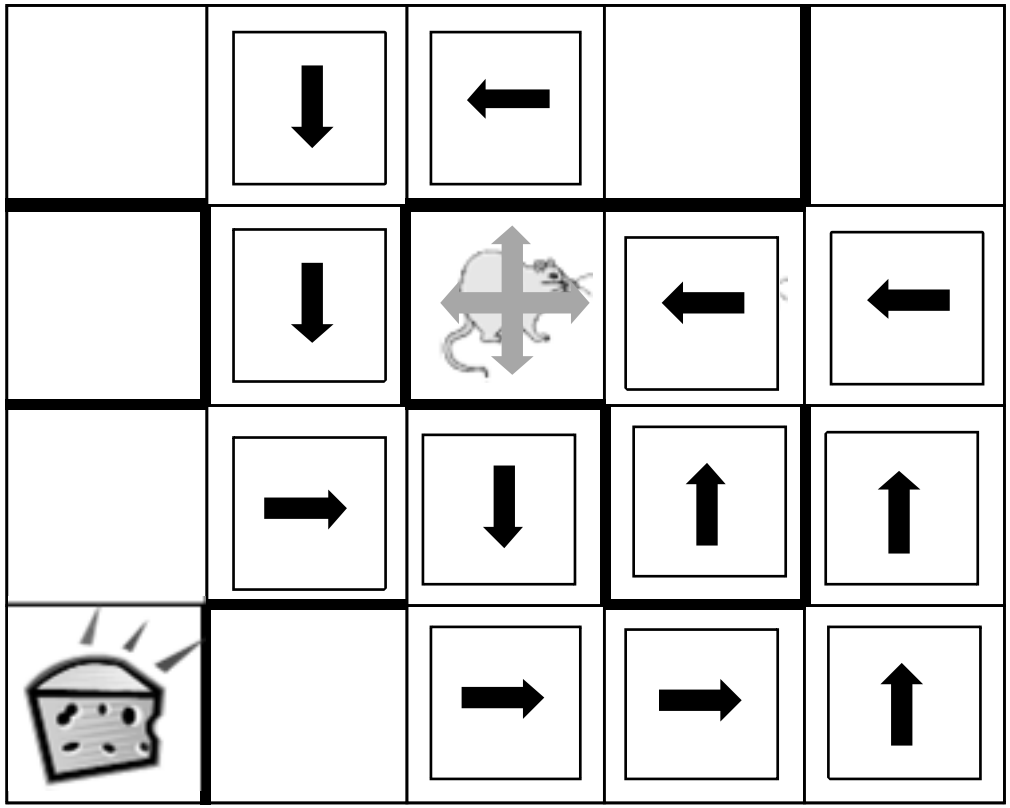
Megtalálja-e az egér a sajtot?



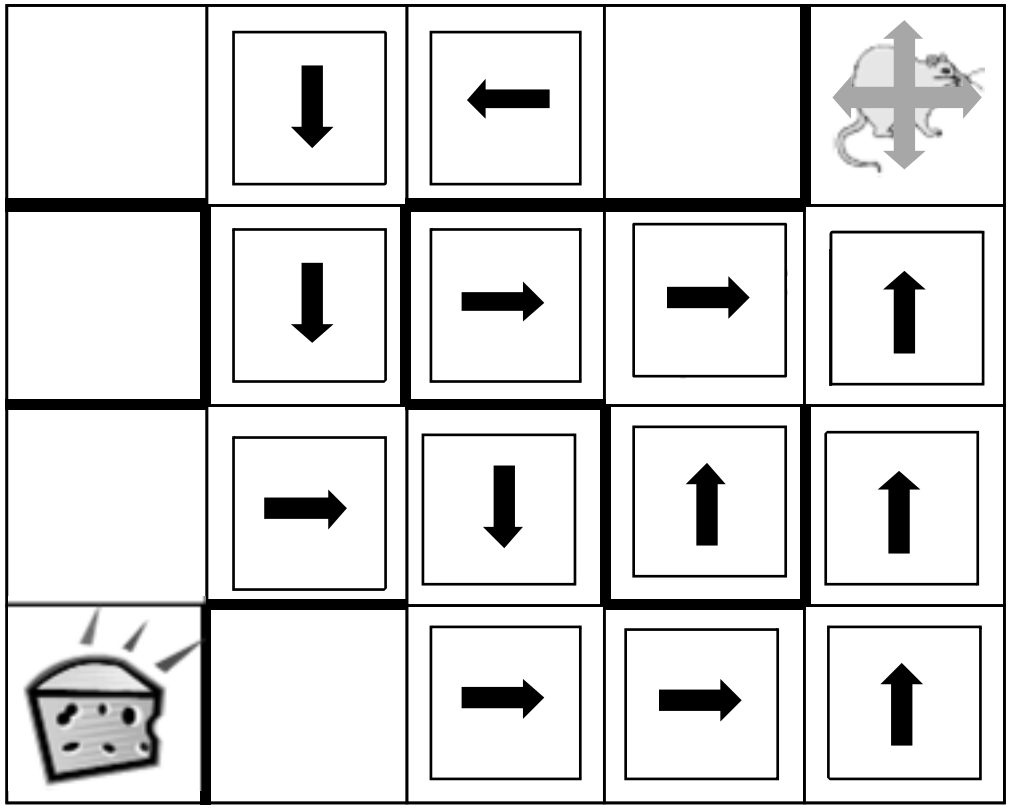
Megtalálja-e az egér a sajtot?



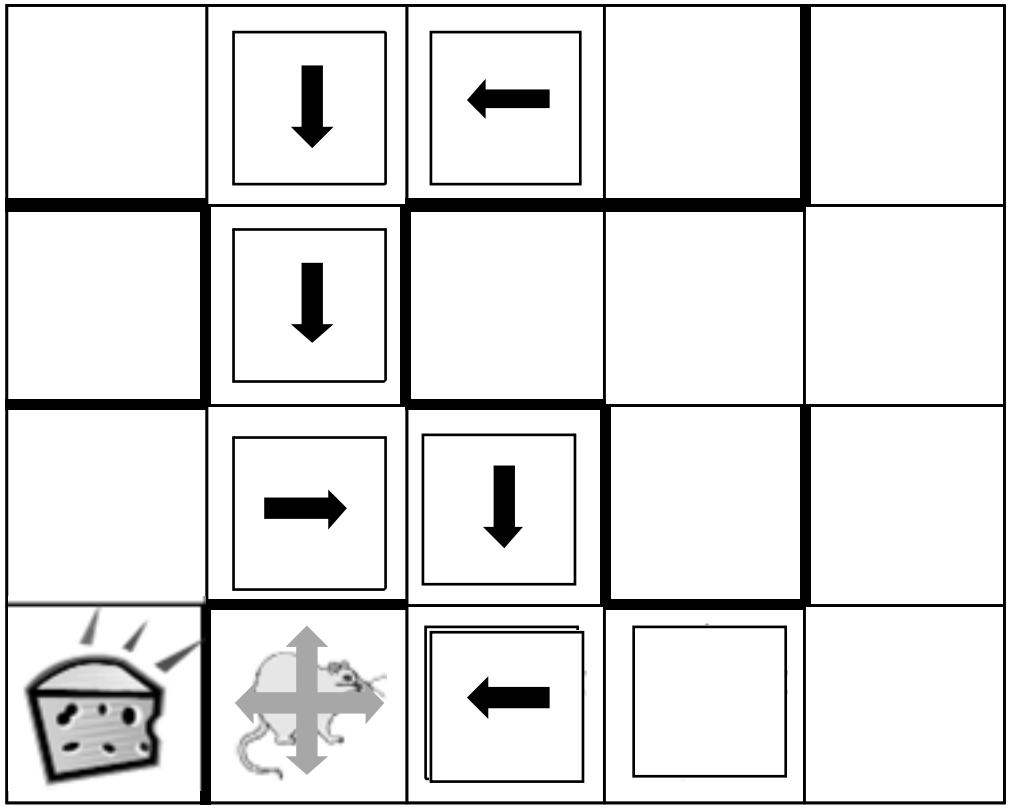
Megtalálja-e az egér a sajtot?



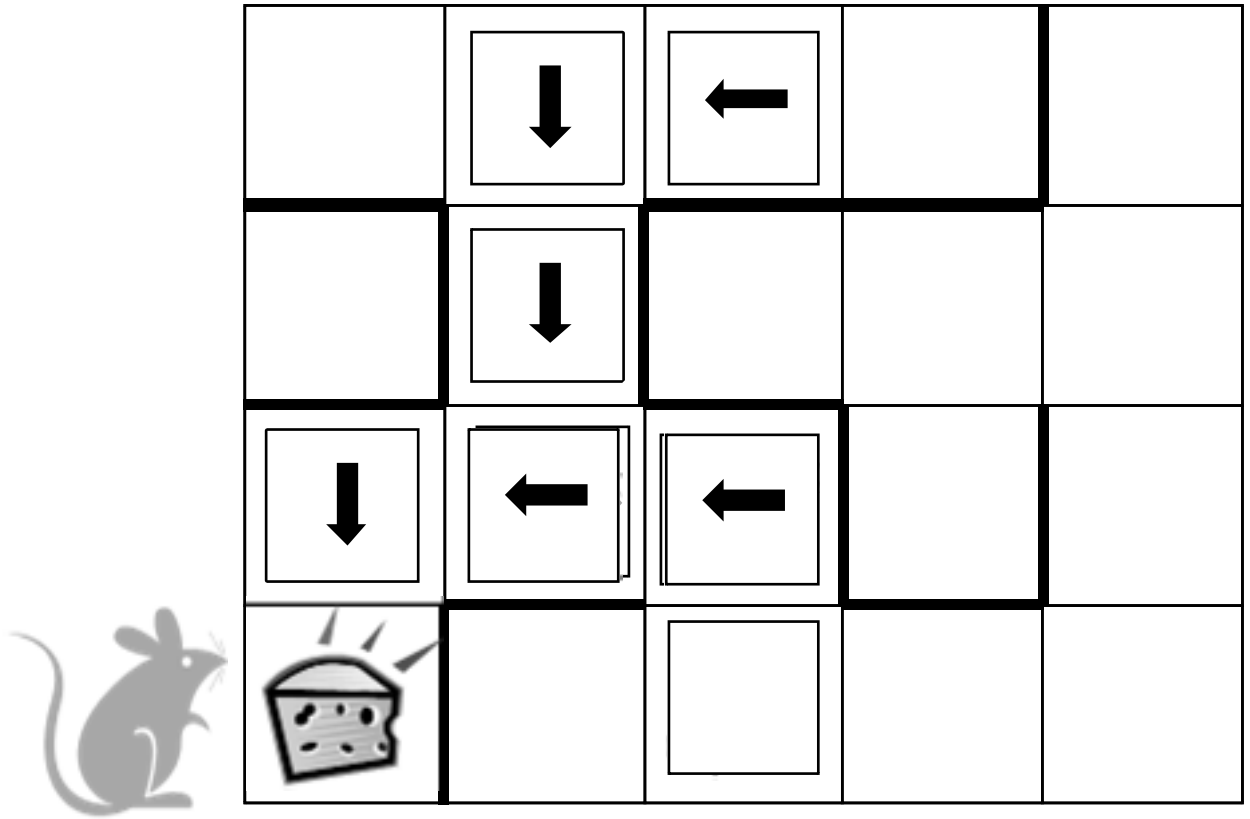
Megtalálja-e az egér a sajtot?



Megtalálja-e az egér a sajtot?



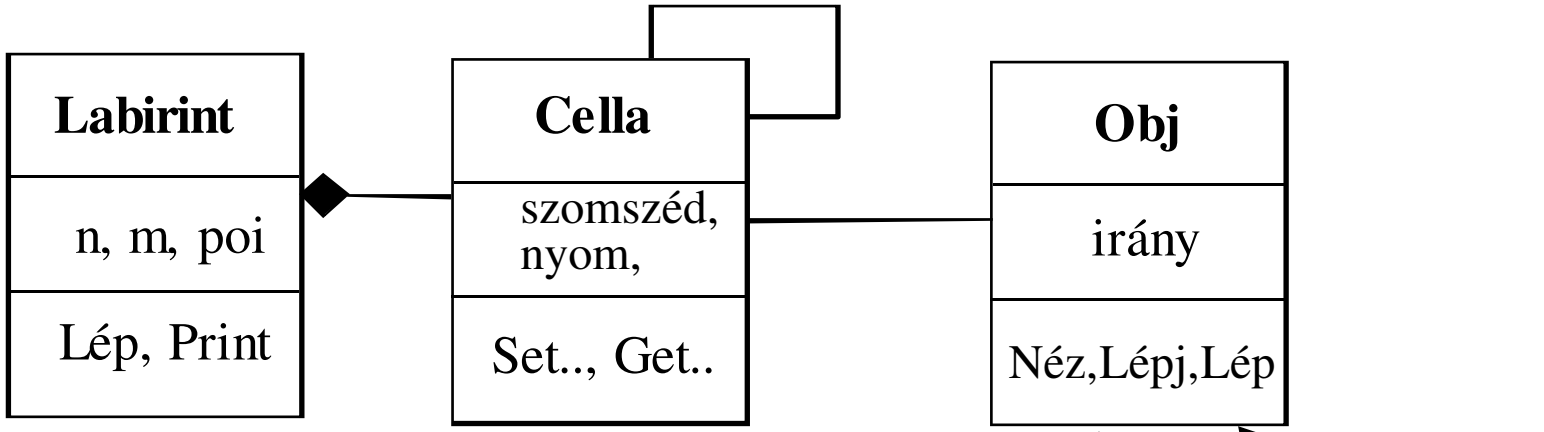
Megtalálja-e az egér a sajtot?



Hogyan modellezhető ?

- Résztvevők, kapcsolatok, tevékenységek
  - labirintus
    - cellákból épül fel
    - a celláknak szomszédai vannak
    - tárolja a rajta álló valamit és annak nyomát
    - megkéri a rajta álló valamit, hogy lépjen
    - kirajzolja az állást
  - egér
    - irány
    - néz
    - lép
  - fal
    - hasonlít az egérhez, de nem lép

Statikus modell



Az attribútumok megadása nem teljes, csak vázlatos! A teljes program ill. dokumentáció letölthető a tárgy honlapjáról

Obj metódusai

```
class Obj {
protected:
   int ir;                    // ebbe az irányba tart most
public:
   Obj() :ir(-1) { };    // kezdő irány
   Obj *Nez(int i, Cella &c, bool b = false);
   void Lepj(int i, Cella &c, bool b = false);
   virtual char Jele() = 0; // jel lekérdzése
   virtual int  Ereje() = 0; // erő lekérdezése
   virtual void Lep(Cella &c); // léptet
   virtual void operator()(Obj*) { }; // ha "megették"
};
```

Eger objektum

```
class Eger :public Obj {
   const int ero;
   const char jel;
public:
   Eger(char j = 'e', int e = 10) :ero(e), jel(j) { }
   char Jele() { return(jel); }          // jele
   int  Ereje() { return(ero); }         //  ereje
   void operator()(Obj *p) {             // meghívódik, ha megették
      cout << "Jaj szegeny " << Jele();
      cout << " megetvett a(z):";
      cout << p->Jele() << " jelu\n";
   }
};
```

Cella objektum /1

```
class Cella {
   int      x, y;        // geometriai koordináták
   Cella   *sz[4];       // szomszédok, ha NULL, akkor nincs
   Obj     *p;           // cella tartalma. URES, ha nincs
   Lista<Labnyom> ny;    // lábnyomok listája
public:
   enum irany {  J, L, B, F }; // nehéz szépen elérni..
   Cella() { SetPos(0, 0); }
   void SetPos(int i, int j);
   void SetSz(int n, Cella *cp) { sz[n] = cp; }// szomszéd
   Cella *GetSz(int n) { return(sz[n]); }
.....
```

Cella objektum /2

```
  ......
    void SetObj(Obj *a) { p = a;}    // objektum beírása
    Obj *GetObj() { return(p); }     // objektum lekérdezése
    void SetNyom(int n);             // lábnyom beírása
    int GetNyom(const Obj *a);       // lábnyom lekérdezése
    void DelNyom() {ny.Torol(Labnyom(GetObj()));}
}
```

Főprogram (részlet)

```
Labirint lab(4,5);                  // 4 * 5 os labirintus
Eger e1, e2;                        // 2 egér; jele: e
lab.SetObj(2, 3, e1);               // egerek elhelyezése
lab.SetObj(1, 4, e2);
try {
   char ch; lab.Print();            // kiírjuk a labirintust
   while (cin >> noskipws >> ch, ch != 'e') {
      lab.Lep(); lab.Print();       // léptetés
   }
} catch (exception& e) {
    cout << e.what() << endl;
}
```

Labirintus léptetés

- Végig kell járni a cellákat
  - meghívni az ott "lakó" objektum léptetését
- Figyelni kell. hogy nehogy duplán léptessünk. (bejárási sorrend elé lépett)
  - Balról jobbra, és lefelé haladunk. Akkor van baj, ha jobbra lép vagy le.
    - Segédváltozókkal az objektumok pointereit ellenőrizzük.
    - Obj::Lep hibát dobhat, kezelni kell (ld. Labirint::Lep())

Obj::lep()/1

```
if (c.GetNyom(this) < 0)        // ha nem volt nyoma, indul
   c.SetNyom(F+1);              // így nem tud visszalépni
  if (++ir <= F) {              // ha van még bejáratlan irány
   if (Obj *p = Nez(ir, c)) {   // ha van szomszéd
    if (Ereje() > p->Ereje()) { // ha Ő az erősebb
      (*p)(this);               // meghívjuk a függv. operátorát
      Lepj(ir, c);              // rálépünk (eltapossuk)
      ir = -1;                  // most léptünk be: kezdő irány
    }
   }
  } else {
```

Obj::Lep() /2

```
 // nincs már bejáratlan irány
    if ((ir = c.GetNyom(this)) > F) {   // kezdő nem lép vissza.
       throw std::logic_error("Kezdo pozicioba jutott");
    } else {                                  // visszalépés
      if (Obj *p = Nez(ir, c, true)) {    // lehet, h. van ott valaki
        if (Ereje() > p->Ereje()) {         // Ő az erősebb
          (*p)(this);                 // meghívjuk a függv. operátorát
          Lepj(ir, c, true);         // visszalépünk és eltapossuk
          ir ^= 2;                   // erre ment be (trükk: a szemben
   levő irányok csak a 2-es bitben különböznek)
}}}}
```

https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_12 → labirintus

Amőba

- Szereplők: tábla, korongok, játékosok
- Kis ismeri a szabályokat?
  - Tábla?
  - Korongok?
- Választott megvalósítás:
  - Tábla ismeri a korongok helyzetét, szomszédokat.
  - A korongok le tudják kérdezni a szomszédokat.
  - A korongok egy általánosított pozíciót és irányt ismernek.
  - Meg tudják állapítani, ha nyert helyzet van. Ekkor színt váltanak.

Table::Position

- A tárolás és a pozíció kapcsolatát elfedi.
- A csak korongra tartozó információkat nyújtja.
- Tárolja az x,y koordinátát, de annak felhasználását a táblára bízza.
- Tábla a „barátja” így az eléri a privát tagfüggvényeket is.
- A korong számára a szomszédokat ill. adott irányú szomszédokat adja, amit a táblától kér el.
- A szomszéd fogalmat így csak a tábla értelmezi.

Table::Position

```
class Position {
  friend class Table; ///< Table osztály hozzáférhet a privát...
  int x, y;           ///< koordináták
  Table *tp;          ///< Erre a táblára hivatkozok.
  int getx() const { return x; } ///< Közvetlenül is elérné, de...
  int gety() const { return y; }
 public:
  Position(int x, int y, Table* tp) :x(x), y(y), tp(tp) { }
  neighbors_type getNeighbors() {
                          return tp->getNeighbors(x, y); }
  Disc* getNext(int dir) { return tp->getNext(x, y, dir); }
 };
```

Table::dirs

```
// Az irányokat egy egész szám jelöli ki, ami nem más,
// mint az irányok táblájának indexe.
// Az irányok fogalom így tetszőlegesen bővíthető és csak a
// tábla értelmezi

struct dir_type { int dx, dy; }
    dirs[8] = { -1, -1 },
              { -1,  0 },
              { -1,  1 },
                  ...
```

Objektum hierachia

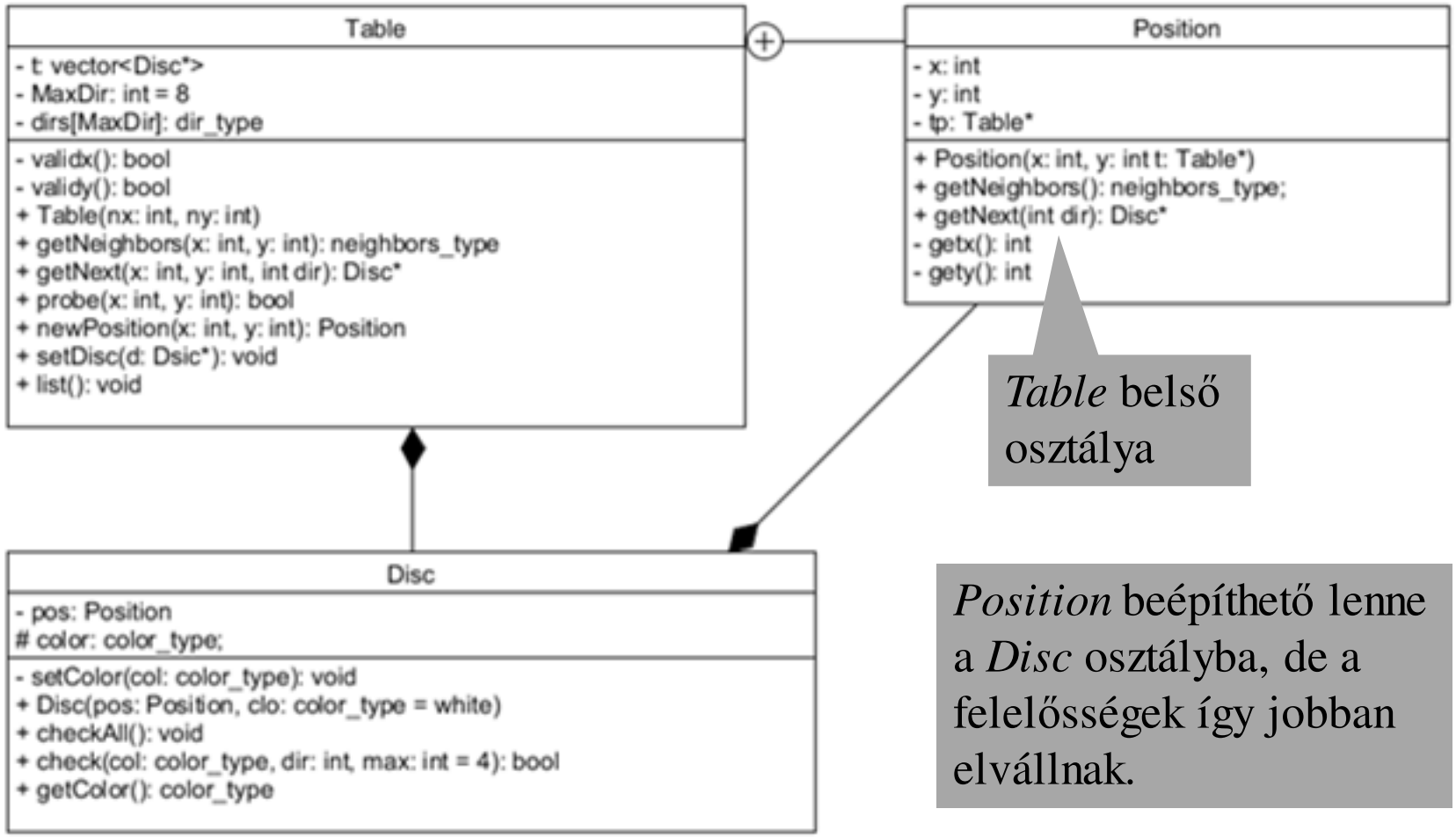


Table belső osztálya

Position beépíthető lenne a *Disc* osztályba, de a felelősségek így jobban elvállnak.

Szomszédok és irányok

```
typedef std::map<int, Disc*> neighbors_type;

class Table {
    std::vector<std::vector<Disc*> > t;
    static const int MaxDir = 8;
    struct dir_type { int dx, dy; } dirs[MaxDir];
...
bool Disc::checkAll() {
   Table::neighbors_type nb = pos.getNeighbors();
   for(Table::neighbors_type::iterator
     it= nb.begin(); it != nb.end(); ++it)
      if ((it->second)->check(color, it->first)) {
         setColor(red);
         return true;
      }
   }
```

Sor ellenőrzése egyik irányban

```
bool Disc::check(color_type col,int dir, int max){
    if (col == color) {
        if (--max == 0) {
            setColor(red); // átváltjuk
            return true;
        } else {
            Disc *dp = pos.getNext(dir);
            if (dp && dp->check(col, dir, max)) {
                setColor(red);
                return true;
            }
        }
    }
    return false;
}
```

Amőba főprogram

```
int main() {
 Table t;
 int x, y;
 t.setDisc(new Disc(t.newPosition(3,3), white));
 t.list(); char c = 'X';
 do {
   cout << c << " lepese: ";
   if (!(cin >> x >> y)) break;
   color_type col = c == 'O' ? white : black;
   if (t.probe(x,y)) {
     t.setDisc(new Disc(t.newPosition(x,y),
                                    color_type(col)));
     t.list();
     c = c == 'X' ? 'O' : 'X';
   } else { cout << "oda nem lephet!"; }
   cout << '\n';
 } while (true);
}
```

https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_12 → amoba