Szoftvertervezés és -fejlesztés II. labor

2. rész

(Adatszerkezetek)

Láncolt lista

Bináris keresőfa

Gráf

Hasító táblázat

Szoftvertervezés és -fejlesztés II.

- Láncolt lista
  - **Generikus típusok**
  - Láncolt lista megvalósítása
- Bináris keresőfa
  - Bejáró tervezési minta
  - Fa megvalósítása
- Gráf
  - Gráf megvalósítása
  - Gráf bejárások
- Hasító táblázat
  - Dictionary osztály
  - Hasító táblázat megvalósítása

Típusok rögzítésének hátrányai

- **Példa: egy Verem osztályt szeretnénk implementálni, milyen típusú adatokat tudjon tárolni?**
  - Feladattól függően rögzítjük a típust, pl. int → nem szerencsés, mert egy hasonló, de pl. stringeket tartalmazó veremnél újra kell írni az egész kódot
  - Object típusokkal dolgozunk, mivel a polimorfizmus hatására így minden típust bele tudunk rakni → működik, de számos hátránnyal jár
    - Sok castolás miatt nehézkes a használata
    - Nincs fordítás idejű típusellenőrzés (csak futás közbeni hibák)
- **Hasonló problémák**
  - Láncolt listát implementálnánk, mi legyen a tartalom mező típusa?
  - Programozási tétel (pl. Keresés) milyen típusú adatokkal dolgozzon?
- **Összefoglalva**
  - Nyelvi támogatást szeretnénk, hogy azonos algoritmusok különböző típusokon is tudjanak működni fordításkori típusellenőrzéssel
  - Hasonló pl. C++ templatek

szenasi.sandor@nik.uni-obuda.hu

Generikus osztályok

- **Előző problémák megoldása: típusparaméterek használata**
- **Minden típusparaméter egy típust képvisel**
  - Osztály tervezésekor ez még nem ismert
  - Objektum példányosításakor már meg kell adni
  - Tehát fordítás idejére már biztosan ismert lesz
- **Típusparaméter használható minden helyen, ahol egyébként típus állhatna**
  - Mezők, tulajdonságok típusa
  - Metódusok paramétereinek típusa
  - Metódusok visszatérési értékének típusa
  - Metódusok lokális változóinak típusa
- **Típusparaméter szintaktika**
  - Típusparaméter jelzése az osztály fejlécben: „<típusnév>”
  - Használata az osztály kódjában: „típusnév”

szenasi.sandor@nik.uni-obuda.hu

Generikus osztály létrehozása

```
class Verem<T>
{
  T[] A;
  int N = 0;

  public Verem(int meret)
  {
    A = new T[meret];
  }

  public void Push(T elem)
  {
    A[N++] = elem;
  }

  public T Pop()
  {
    T vissza = A[--N];
    return vissza;
  }

  public T Top
  {
    get { return A[N-1]; }
  }
}
```

Generikus osztály példányosítása

- **Az előzőleg elkészített verem használata**

```
Verem<int> v1 = new Verem<int>(5);
v1.Push(5);
int a1 = v1.Top;
int b1 = v1.Pop();

Verem<string> v2 = new Verem<string>(5);
v2.Push("Hello");
string a = v2.Top;
string b = v2.Pop();
```

- **A referencia deklarációkor és a példányosításkor már meg kell adni a tényleges típust**
  - Szintaktika deklarációnál és példányosításnál: „osztálynév<tényleges típus>”
- **Előnyei**
  - Azonos kód működik különféle típusokkal
  - Fordításidejű típusellenőrzés
  - Egyszerű, áttekinthető használat

Ismeretlen típusú változó inicializálása

- **A T típusú változóknak nem tudunk kezdőértéket adni, mivel a később megadott típus lehet**
  - Objektum (ilyenkor null lenne az alapérték)
  - Érték típus (ilyenkor bitenkénti nulla az alapérték - 0/0.0/false/stb.)
- **Ezért inicializációra használható a „default(típusparam)” kulcsszó**
  - A tényleges típusnak megfelelően a fenti értékeket egyikét képviseli
- **Használatára példa:**

```
class Minta<T>
{
  T a;

  public Minta( )
  {
    a = default(T);
  }
}
```


szenasi.sandor@nik.uni-obuda.hu

Megszorítások

- **Gyakran nem akarjuk megengedni, hogy T bármely típussal helyettesíthető legyen**
  - Olyan műveleteket akarunk elvégezni, ami csak bizonyos típus helyettesítéseknél lenne végrehajtható (pl. összehasonlítani)
  - Ez elég gyakori, hiszen az eltárolás kivételével minden művelethez már szükségünk van némi információra a tényleges típussal kapcsolatban
- **A megszorításokat az osztály fejléc utáni „where” kulcsszó után sorolhatjuk fel**
- **T alapvető típusára vonatkozó megszorítások**
  - Legyen érték típus

```
class Minta<T> where T : struct
{ ... }
```

  - Legyen referencia típus

```
class Minta<T> where T : class
{ ... }
```


szenasi.sandor@nik.uni-obuda.hu

Interfész/leszármazott megszorítás

- **Gyakori, hogy a T-ként megadható típusra szeretnénk további megszorításokkal élni**
  - Legyen valamely osztály leszármazottja
  - Valósítson meg egy interfészt
- **Ez a megszorítás megadható az alábbi formában:**

```
class RendezettParos<T> where T : IComparable
{
  T kicsi, nagy;

  public RendezettParos(T a, T b)
  {
    kicsi = a.CompareTo(b) < 0 ? a : b;
    nagy = a.CompareTo(b) < 0 ? b : a;
  }
}
```

- **Így használhatjuk a T típus elvárt metódusait (polimorfizmus!)**

szenasi.sandor@nik.uni-obuda.hu

Alapértelmezett konstruktor megszorítás

- **Ha T típusú objektumot akarunk példányosítani, akkor tudnunk, kell, hogy milyen konstruktort hívhatunk meg**
  - Paraméter nélküli konstruktor: ezt megkövetelhetjük egy „new()” megszorítással, utána már meghívható
  - Paraméteres konstruktor: sehogy se hívható meg
- **Ez a megszorítás megadható az alábbi formában:**

```
class ObjektumGyar<T> where T : new()
{
  public T[] Letrehozas(int darab)
  {
    T[] A = new T[darab];
    for (int i = 0; i < darab; i++)
      A[i] = new T();
    return A;
  }
}
```


szenasi.sandor@nik.uni-obuda.hu

Néhány korlát

- **Leszármazott típus megszorítás**
  - Primitív típusok (pl. int) nem használhatók
  - Lezárt típus nem használható
  - Array, Delegate, Enum, ValueType nem szerepelhet
- **Operátorokat nem tudunk használni a T típusú elemeken**
  - Operátorokat (+,-,<,>,stb.) nem tudunk használni T típusú elemeken
  - Néhány kivételtől eltekintve (pl. ==)
  - Ezeket metódusként megvalósítva tudjuk használni (pl. CompareTo)

További lehetőségek

- **Statikus generikus osztályok**
  - Statikus osztály, illetve metódus is lehet generikus
  - Mivel itt nincs példányosítás, ezért a híváskor kell megadni a típust

```
Számoló<int>.Növel();
```

  - Különböző típusokhoz saját statikus osztályok tartoznak (saját mezőkkel)
- **Több típusparaméter használata**
  - Minden helyen, ahol használhatunk típusparamétert, megjelenhet több típusparaméter is
  - Ezek egymástól függetlenül használhatók a kódokban

```
class KulcsAlapjanTarolo<K,T> where K : IComparable
{
  K[] kulcs;
  T[] tartalom;

  public T Keres(K mit) { ... }
}
```


szenasi.sandor@nik.uni-obuda.hu

Generikus osztályok öröklése

- **Generikus típusok is részt vehetnek az öröklésben**
  - Nem generikusnak is lehet generikus leszármazottja
  - Generikusnak lehet nem generikus leszármazottja
  - Generikusnak lehet generikus leszármazottja (és a típusparaméterek száma lehet több/kevesebb/azonos is)
- **Ha a leszármazottban nem jelenik meg az ős valamelyik típusparamétere, akkor az öröklésekor ezt meg kell határozni**

```
class SzamLista : List<int> { }

class GenerikusOs<A,B> { }
class GenerikusLeszarmazott<T> : GenerikusOs<T, int> { }
```

- **Legkésőbb a példányosításkor minden típusparaméternek értéket kell kapnia**
  - Vagy már a leszármazás(ok) során
  - Vagy a példányosításkor
- **A megszorítások nem öröklődnek, azokat meg kell ismételni**

Generikus metódus

- **Nem csak osztály, hanem egy metódus is lehet generikus**
  - Metódus neve után szerepelhet a típusparaméter neve
  - Ez a paraméter használható az előzőekben megismert helyeken
    - Paraméter
    - Visszatérési érték
    - Lokális változó

```
class Minta
{
  T Min<T>(T a, T b) where T : IComparable
  {
    return a.CompareTo(b) < 0 ? a : b;
  }
}
```

- **A T tényleges értéke**
  - Független a osztály példányosítástól (az nem generikus)
  - A híváskor átadott paraméterek típusától függ (ha a hívásnak megfelel generikus és nem generikus változat is, akkor az előbbi fut le)

szenasi.sandor@nik.uni-obuda.hu

Szoftvertervezés és -fejlesztés II.

- Láncolt lista
  - Generikus típusok
  - **Láncolt lista megvalósítása**
- Bináris keresőfa
  - Bejáró tervezési minta
  - Fa megvalósítása
- Gráf
  - Gráf megvalósítása
  - Gráf bejárások
- Hasító táblázat
  - Dictionary osztály
  - Hasító táblázat megvalósítása

.NET környezet beépített gyűjteményei

- **Lista: List<T>**
  - Count – lista elemszáma
  - Add(T item), Insert(int index, T item) – lista végére/adott helyre felvétel
  - [index] – sorszám alapján történő kiolvasás
  - Remove(T item), RemoveAt(index) – elem törlése
  - Clear() – teljes lista törlése
  - bool Contains(T item) – eldöntés, benne van-e a listában az elem
- **Sor: Queue<T>**
  - Count – sorban lévő elemek száma
  - Enqueue(T item) – elemet berakja a sorba
  - T Dequeue() – következő elemet kiveszi a sorból
- **Verem: Stack<T>**
  - Count – veremben lévő elemek száma
  - Push(item) – elemet berakja a verembe
  - T Pop() – utolsó elemet kiveszi a veremből

.NET környezet beépített gyűjteményei (2)

- **Hasító táblázat: Dictionary<TKey, TValue>**
  - Count – elemek száma
  - Add(TKey key, TValue item) – új elem felvétele megadott kulccsal
  - [TKey] – kulcs alapján elem kiolvasás (ha nincs ilyen, akkor kivételt dob)
  - Remove(TKey key) – adott kulcsú elem törlése
  - bool ContainsKey(TKey key) – van adott kulcs?
  - bool ContainsValue(TValue item) – van adott elem?
  - Clear() – teljes törlés
  - Keys – kulcsok listája
  - Values – értékek listája
  - TryGetValue(TKey key) – ha van ilyen kulcs, visszaadja a hozzá tartozó értéket, különben pedig „default(TValue)”-t

szenasi.sandor@nik.uni-obuda.hu

Saját láncolt lista készítése

- **Implementáljuk az előadáson megismert láncolt listát**
- **Két osztályra lesz szükségünk**
  - ListaElem osztály
    - Tartalmazza a tartalmi részt
    - Rendezett lista esetén tartalmaz egy kulcsot is
    - Tartalmazza a következő hivatkozást, ami szintén ListaElem típusú
  - LancoltLista osztály
    - Tartalmazza a fej mezőt, ami a lista első elemére mutat majd
    - Tartalmazza a kívülről elérhető metódusokat: Beszúrás, Törlés, stb.
- **Legyen generikus**
  - A listaelemek tartalma legyen T típusú
  - Rendezett láncolt lista esetén a kulcs típusa legyen K (ami összehasonlítható)
  - Emiatt maga a LancoltLista osztály is generikus lesz
- **Mivel csak a LancoltLista osztály fér hozzá a „fej” mezőhöz, azt nem kell mindig átadni a metódusoknak, közvetlenül a mezőt is használhatják a metódusok (eltérően az ea. diákkal)**

Láthatósági kérdések

- **Célszerű lenne az alábbiaknak megfelelő megoldást találni:**
  - A ListaElem objektumokat csak a LancoltLista érje el
  - A ListaElem objektumok mezői kívülről ne legyenek elérhetőek, csak a LáncoltLista számára
  - Lehetőleg magát a ListaElem típust se lássa más osztályt
- **Megoldás: beágyazott osztályok**
  - A „Belso” nevű osztály a „Kulso” számára látható
  - A „Kulso” osztály metódusai hozzáférnek a „Belso” osztály mezőihez
  - Egyéb osztályok nem látják a „Belso”-t
  - Mivel eleve a típust se látják, így értelemszerűen a mezőihez se tudnak hozzáférni
  - Tulajdonságot a későbbi (fa) címszerinti paraméterátadás miatt nem használunk

```
class Kulso
{
  class Belso
  {
    public int valtozo;
  }

  public void Muvelet()
  {
    Belso x = new Belso();
    x.valtozo = 5;
  }
}
```

Egy lehetséges láncolt lista megvalósítás

```
class LancoltLista<T>
{
  class ListaElem
  {
    public T tartalom;
    public ListaElem kovetkezo;
  }

  ListaElem fej;

  public void ElejereBeszuras(T elem)
  {
    ListaElem uj = new ListaElem();
    uj.tartalom = elem;
    uj.kovetkezo = fej;

    fej = uj;
  }
}
```

További megjegyzések

- **Az előzőhöz hasonlóan elkészíthető az összes lista művelet**
  - Végére beszúrás
  - Adott helyre beszúrás
  - Törlés
  - Keresés
  - Bejárás
- **A bejárás feladata az volt, hogy minden listaelem tartalmával végezzünk el „valamilyen” feladatot. Ezt célszerűen egy delegálttal adhatjuk át, ami T típusú elemet vár paraméterként**
- **Rendezett láncolt lista esetében kell még egy típusparaméter (K), ami összehasonlítható**
- **A lista belseje mindig legyen láthatatlan. A listát használók csak kulcsokkal és tartalmakkal érintkezzenek, ListaElem példányokhoz sose férjenek hozzá.**

Szoftvertervezés és -fejlesztés II. labor

2. rész

(Adatszerkezetek)

Láncolt lista

Bináris keresőfa

Gráf

Hasító táblázat

Szoftvertervezés és -fejlesztés II.

- Láncolt lista
  - Generikus típusok
  - Láncolt lista megvalósítása
- Bináris keresőfa
  - **Bejáró tervezési minta**
  - Fa megvalósítása
- Gráf
  - Gráf megvalósítása
  - Gráf bejárások
- Hasító táblázat
  - Dictionary osztály
  - Hasító táblázat megvalósítása

Bejáró tervezési minta

- **Érdemes észrevenni, hogy az összes beépített adatszerkezet (tömb, List, stb.) használható foreach ciklussal**
- **Ennek háttere a bejáró tervezési minta, ami .NET környezetben két interfész megvalósítását igényli**
- **IEnumerable<T> - a „bejárható szerkezet” valósítja meg**
  - IEnumerator<T> GetEnumerator()
    A bejárást igénylő kód (jelen esetben a foreach ciklus) mindig meghívja a bejárni kívánt adatszerkezet GetEnumerator metódusát, ami létrehoz és visszaad egy IEnumerator interfészt megvalósító másik objektumot. Ezzel végig lehet lépegetni az adatszerkezet elemein, kiolvasva azok tartalmát.
- **IEnumerator<T> - a „bejárást elvégző” objektum valósítja meg**
  - MoveNext() – A következő elemre próbál lépni. Igazat ad vissza, ha van ilyen, különben pedig hamisat. Fontos: a bejáró létrehozásakor az „első elem előtt” áll, tehát egy kezdő MoveNext hívás után fog csak rálépni az első elemre
  - Current – Ezzel lehet kiolvasni az aktuális elem T típusú tartalmát
  - Reset() – Ezzel lehet alaphelyzetbe állítani a bejárót (első elem elé)

Saját bejáró készítése

- **Saját adatszerkezeteinkhez is elkészíthetjük a megfelelő bejáró osztályt, csak meg kell valósítanunk az előző interfészeket**
- **Ha ezt teljesítjük, akkor a saját adatszerkezetünk is bejárható lesz „foreach” ciklus segítségével**
- **A „foreach” a lentihez hasonló kódot futtat le, tehát úgy kell a bejárónkat elkészíteni, hogy megfelelően reagáljon az interfészekben található metódusok hívásaira**

```
List<int> obj = new List<int>();

foreach(int x in obj)
{
  Művelet(x);
}
```

```
List<int> obj = new List<int>();

IEnumerator<T> tmp = obj.GetEnumerator();
while(tmp.MoveNext())
{
  T x = tmp.Current;
  Művelet(x);
}
```


szenasi.sandor@nik.uni-obuda.hu

Láncolt lista bejárása

- **Egészítsük ki az előző órán készített láncolt lista megvalósításunkat úgy, hogy együttműködjön a „foreach”-el**
- **Készítsünk egy „ListaBejaro<T>” osztályt, ami megvalósítja az „IEnumerator<T>” interfészt**
  - Konstruktora kapjon egy lista fejet. Induláskor azonban még ne álljon erre rá, hanem a lista feje „előtt” várakozzon (fizikailag nyilván nem tudunk ide mutatni, egy másik változóval lehet jelölni, hogy még nem indult el a bejárás)
  - Current
    A bejáró egy mezőjével mindig mutasson a láncolt lista aktuális elemre. A Current tulajdonság adja vissza ennek a tartalmi részét.
  - MoveNext()
    - első híváskor rálép a lista első elemére,
    - minden további híváskor átlép a lista következő elemére,
    - mindkét esetben igazat ad vissza ha van ilyen elem, és hamisat ha nincs (lista vége)
  - Reset()
    Visszaáll a lista első eleme „elé”

Szoftvertervezés és -fejlesztés II.

- Láncolt lista
  - Generikus típusok
  - Láncolt lista megvalósítása
- Bináris keresőfa
  - Bejáró tervezési minta
  - **Fa megvalósítása**
- Gráf
  - Gráf megvalósítása
  - Gráf bejárások
- Hasító táblázat
  - Dictionary osztály
  - Hasító táblázat megvalósítása

Saját bináris keresőfa készítése

- **Implementáljuk az előadáson megismert bináris keresőfát**
- **Két osztályra lesz szükségünk**
  - FaElem osztály
    - Tartalmazza a kulcsot (generikus, K)
    - Tartalmazza a tartalmat (generikus, T)
    - Tartalmazza a bal és jobb részfa gyökerét, ami ugyanilyen típusú
  - BinarisKeresofa osztály
    - Tartalmazza a gyökér mezőt, ami a fa első elemére mutat majd
    - Tartalmazza a kívülről elérhető metódusokat: beszúrás, törlés, stb.
- **Bejáráshoz használhatjuk bármelyik ismert módszert, a rekurzió miatt a bejáró implementálása viszont nehézkes. Megoldások:**
  - Készítsünk nem rekurzív bejárást (verem használatával)
  - „yield return” használata (nem tárgyaljuk)
  - Két lépésre bontás:
    - Először a rekurzív bejárás az összes tartalmat összegyűjti egy listába
    - Ezután visszaadjuk ennek a listának a bejáróját

Szoftvertervezés és -fejlesztés II. labor

2. rész

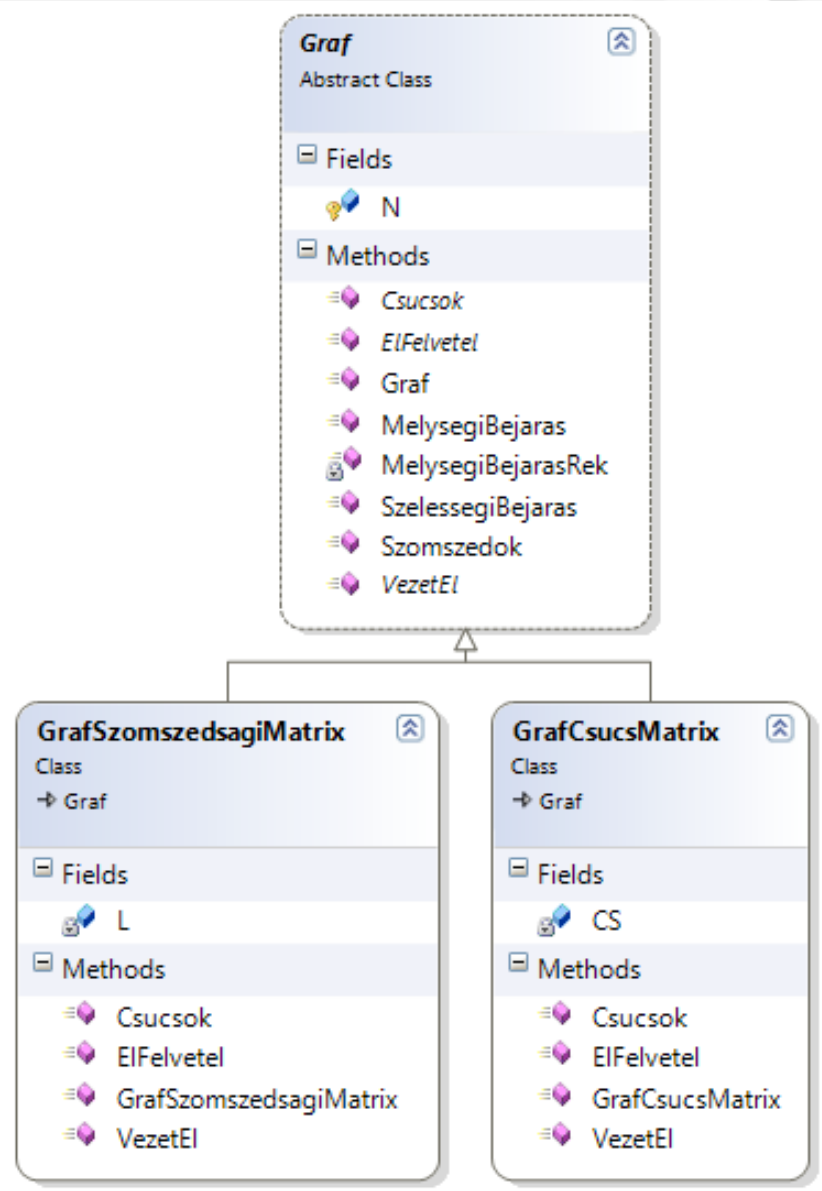
Láncolt lista
Bináris keresőfa
Gráf
Hasító táblázat

Szoftvertervezés és -fejlesztés II.

- Láncolt lista
  - Generikus típusok
  - Láncolt lista megvalósítása
- Bináris keresőfa
  - Bejáró tervezési minta
  - Fa megvalósítása
- Gráf
  - **Gráf megvalósítása**
  - Gráf bejárások
- Hasító táblázat
  - Dictionary osztály
  - Hasító táblázat megvalósítása

Gráf absztrakció

- **Implementáljuk a jegyzetben található Gráf adatszerkezetet**
  - csúcsmátrix segítségével
  - szomszédsági lista segítségével
- **N darab csúcsa legyen (konstruktor)**
- **A gráf csúcsok legyenek számok 0..N-1 között**
- **A gráfok alapvető műveletei már az absztrakt szinten megjelennek**
  - List<int> Csucsok
  - ElFelvetel(int honnan, int hova)
  - bool VezetEl(int honnan, int hova)
- **A fenti három segítségével már implementálható ezen a szinten:**
  - List<int> Szomszedok(int cs)



szenasi.sandor@nik.uni-obuda.hu

Gráf implementáció

- **Az előzőleg elkészített absztrakt gráf leszármazottjai lesznek a tényleges megvalósítások**
- **Ezek tárolják a csúcsok és élek adatait, ezekhez férünk hozzá a felüldefiniált metódusok segítségével**
- **GrafCsucsMatrix**
  - CS : egy NxN méretű tömb
  - Csúcsok : visszaadja a számokat 0..N-1 között
  - ElFelvetel : CS[honnan,hova] mezőbe 1-et rak
  - VezetEl: vezet, ha CS[honnan,hova] nem 0
- **GrafSzomszedsagiLista**
  - L : egy N darab listát tartalmazó tömb
  - Csúcsok : visszaadja a számokat 0..N-1 között
  - ElFelvetel : felveszi a honnan-adik listába a hova számot
  - VezetEl: megnézi, hogy a honnan-adik listában van-e hova szám

Szoftvertervezés és -fejlesztés II.

- Láncolt lista
  - Generikus típusok
  - Láncolt lista megvalósítása
- Bináris keresőfa
  - Bejáró tervezési minta
  - Fa megvalósítása
- Gráf
  - Gráf megvalósítása
  - **Gráf bejárások**
- Hasító táblázat
  - Dictionary osztály
  - Hasító táblázat megvalósítása

Gráf bejárások

- Mind a szélességi, mind pedig a mélységi bejárás megvalósítható az absztrakt osztályban megadott (ott még absztrakt) metódusok segítségével
- Emiatt magukat a bejárásokat is célszerű már ezen a szinten megírni, és ezek működni fognak mindkét tényleges megvalósítás esetében a tényleges tárolási módtól függetlenül
- Az ábra mutatja a szélességi bejárást, hasonlóan a mélységi bejárás is egyszerűen implementálható

```
public void SzelessegiBejaras(int start)
{
  List<int> F = new List<int>();
  Queue<int> S = new Queue<int>();
  S.Enqueue(start);
  F.Add(start);
  while (S.Count != 0)
  {
    int k = S.Dequeue();
    Feldolgoz(k);
    foreach (int x in Szomszedok(k))
    {
      if (!F.Contains(x))
      {
        S.Enqueue(x);
        F.Add(x);
      }
    }
  }
}
```


szenasi.sandor@nik.uni-obuda.hu

Szoftvertervezés és -fejlesztés II. labor

2. rész

(Adatszerkezetek)

Láncolt lista

Bináris keresőfa

Gráf

Hasító táblázat

Szoftvertervezés és -fejlesztés II.

- Láncolt lista
  - Generikus típusok
  - Láncolt lista megvalósítása
- Bináris keresőfa
  - Bejáró tervezési minta
  - Fa megvalósítása
- Gráf
  - Gráf megvalósítása
  - Gráf bejárások
- Hasító táblázat
  - **Dictionary osztály**
  - Hasító táblázat megvalósítása

Beépített lehetőségek

- **A C# *Dictionary* osztálya alkalmas kulcs-érték párok tárolására**
- **Generikus típus, elsőként a kulcs, másodikként pedig a tartalom típusát kell meghatározni**

```
Dictionary<string, Hallgato> tarolo;
tarolo = new Dictionary<string, Hallgato>();
```

- **Alapvető műveletei**
  - *Count* - kulcs-érték párok száma
  - *[kulcs]* – kulcs alapján érték kiolvasás (ha nincs, kivételt dob)
  - *TryGetValue(kulcs,érték)* – kulcs alapján érték kiolvasása (nem dob kivételt)
  - *Keys/Values* – aktuálisan eltárolt kulcsok/értékek gyűjteménye
  - *Add(kulcs, érték)* – új kulcs-érték pár felvétele
  - *Remove(kulcs)/Clear()* – kulcs alapján törlés/teljes törlés
  - *ContainsValue(érték)/ContainsKey(kulcs)* – adott értéket/kulcsot tartalmaz?
  - Bejáró: *KeyValuePair<TKey,TValue>* objektumokat ad vissza

Hasító függvény

- **Egy elem hasítása két lépésben történik**
  - kulcs típusból szám létrehozása
    - a kulcs tetszőleges típus (pl. szöveg, Hallgató objektum, stb.) lehet, így azt a Dictionary nem tudhatja, hogy lehet hatékonyan számmá alakítani
    - ezért lekérdezi az átadott kulcs objektum *.GetHashCode()* értékét
  - a szám alapján a tényleges hasítás
    - a fenti alapján egy belső függvénnyel a Dictionary kiszámolja az objektum tényleges helyét (jelenlegi implementációban egy egyszerű maradékos osztással)
- **GetHashCode**
  - az objektum aktuális tartalma alapján egy egész számot kell visszaadnia
  - az Object szinten megadott virtuális függvény, így minden objektumnak van
  - azonos tartalmú objektumoknál mindig ugyanaz legyen a visszaadott hasító érték is (emiatt az *Equals* metódus felülírásakor ezt is célszerű felülírni)
  - egy Dictionary-ben eltárolt objektumot nem szabad úgy megváltozatni, hogy annak hatására megváltozzon a visszaadott hasító érték!
  - egy Dictionary-ben eltárolt kulcsoknak mindig különbözőnek kell lenniük (ez persze nem zárja ki, hogy két különböző kulcs hasító értéke azonos)

Szoftvertervezés és -fejlesztés II.

- Láncolt lista
  - Generikus típusok
  - Láncolt lista megvalósítása
- Bináris keresőfa
  - Bejáró tervezési minta
  - Fa megvalósítása
- Gráf
  - Gráf megvalósítása
  - Gráf bejárások
- Hasító táblázat
  - Dictionary osztály
  - **Hasító táblázat megvalósítása**

Saját hasító táblázat

- **Valósítsuk meg az előadáson megismert hasító táblázat szerkezetet**
- **A kulcsütközéseket kezeljük**
  - láncolt listával
  - nyílt címzéssel
- **A hasító függvény legyen virtális**
- **A *GetHashCode* metódus értékén alapuljon a számmá alakítás**
- **Hiba esetén dobjon kivételeket**