

ADATSZERKEZETEK (VEREM, SOR)

ADATSZERKEZET FOGALMA

Az adatszerkezet egymással kapcsolatban álló adatok összessége, amelyen meghatározott, az adatszerkezetre jellemző műveletek végezhetők el. Az adatok közötti kapcsolatok alapján megkülönböztethetünk:

- **szekvenciális szerkezeteket, sorozatokat (pl. szekvenciális fájl, verem, sor, lista):** azonos típusú elemeket tartalmaznak, minden elemnek (az utolsót kivéve) pontosan egy rákövetkező eleme van, ez meghatározza a feldolgozás sorrendjét.
- **hierarchikus szerkezeteket:** egy elemek hierarchikusan egymás alá vannak rendelve. (pl. bináris fa, általános fa)
- **hálós szerkezeteket:** bármely elem bármely másikkal kapcsolatban állhat (gráf)

Az adatszerkezetek nagyobb része a programozási nyelvekben nem definiált, azt a programozó hozza létre problémák megoldásához.

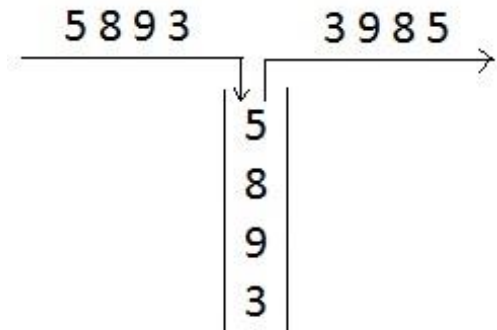
VEREM (STACK)

A verem fogalma

A verem adatszerkezet egy speciális szekvenciális tároló, amelyből mindig a legutolyára betett elemet vehetjük ki legeloszor, azaz a verem egy olyan speciális vektor, melynek csak az egyik végén lehet műveletet végezni, tehát beszúrni (PUSH) vagy törölni (POP).

Emiatt szokás a vermet **LIFO** (**L**ast **I**n – **F**irst **O**ut) szerkezetnek nevezni.

Az adatszerkezet homogén. Az elsőnek beheleyezett elem a verem aljára kerül, majd erre pakoljuk a többi. Csak az utoljára betett elemhez férhetünk hozzá, a többi csak úgy érhetjük el, hogy a felettük levőket eltávolítjuk.



VEREM (STACK)

A verem megengedett műveletei

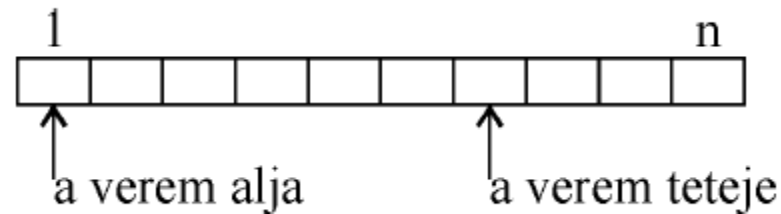
- **Verembe** (*push*): egy elem betétele a verembe (a verem „tetejére”). Csak akkor lehetséges, ha a verem még nem telt meg.
- **Veremből** (*pop*): a verem „tetején” található elem kivétele a veremből. Csak akkor tudunk elemet kivenni, ha a verem nem üres.
- **Inicializálás**:(*init*): alapállapotba helyezés, ürítés. (Dinamikus megvalósítása esetén ez két különböző eljárás.)
- **Üres?**: igaz, ha a veremben egyetlen elemet sem tárolunk.
- **Teli?**: igaz, ha a verembe már nem tehetünk újabb elemet, mert megtelt. (Dinamikus megvalósítás esetén csak akkor van tele a verem, ha a rendelkezésünkre álló memória elfogyott)

VEREM (STACK)

Verem megvalósítása

Folytonos tárolás

A vermet vektorral kezeljük, ahol az első elem mindig a vektor alja. Bővíteni, új elemet felvinni csak a verem tetejéhez lehet. Hozzáférti is csak a verem tetejéhez lehet. Ha egy közbenső elemet akarunk elérni, akkor törölni kell a felette lévő elemeket.



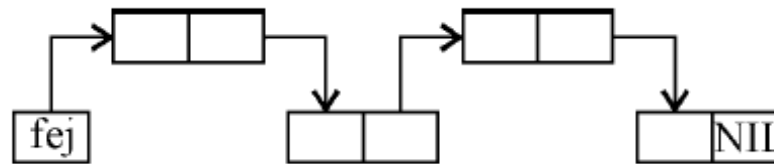
VEREM (STACK)

Verem megvalósítása

Szétszórt ábrázolás

Egy irányban láncolt listával valósítják meg. A listafej mindig az aktuális első elemre mutat.

Ide kell beszúrní az új elemet adatfelvételnél.



Verem műveletek (új):

- **Elérés:** (top) művelet, a verem legfelső elemének lekérdezése.

VEREM (STACK)

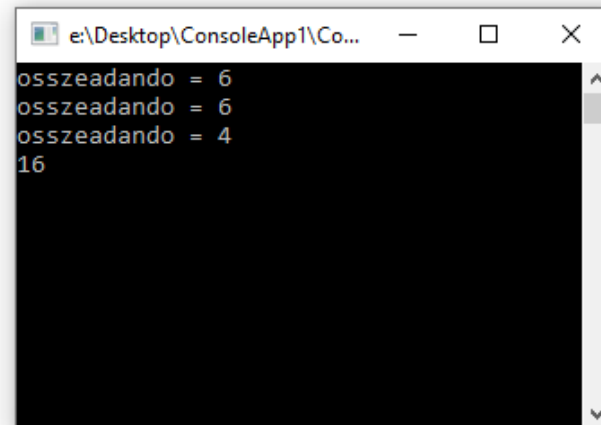
A verem alkalmazása

- Programozási nyelvekben az egymásba ágyazott eljárások, illetve függvények, valamint a rekurzió megvalósítása.
- Zárójelezett kifejezés / (), [], { } / helyességének az ellenőrzése.
- Undo (mégsem) funkció megvalósítása az alkalmazói programokban.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace ConsoleApp1
8  {
9      class Program
10     {
11         static void Main(string[] args)
12         {
13             //Verem létrehozása
14             Stack<int> verem = new Stack<int>();
15
16             //Elemek hozzáadása
17             verem.Push(4);
18             verem.Push(6);
19             verem.Push(6);
20
21             //Összeg kiszámítása
22             int osszeg = 0;
23             int osszeadando = 0;
24
25             while (verem.Count > 0)
26             {
27                 osszeadando = verem.Pop();
28                 Console.WriteLine($"osszeadando = {osszeadando}");
29
30                 osszeg += osszeadando;
31             }
32             Console.WriteLine(osszeg);
33
34             Console.ReadKey();
35         }
36     }
37 }

```



```

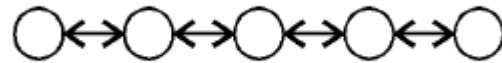
e:\Desktop\ConsoleApp1\Co...
osszeadando = 6
osszeadando = 6
osszeadando = 4
16

```


SOR (QUEUE)

A sor fogalma

- A sor adatszerkezet egy speciális szekvenciális tároló, amelyből mindig a legelsőként betett elemet vehetjük ki legelőször, az elemek mindig két szomszédos elemmel vannak kapcsolatban, kivéve a szélső elemeket.



- Emiatt szokás a sort **FIFO (First In – First Out)** szerkezetnek nevezni.

A sor adatszerkezet alkalmazásai

- processzornál futásra jelölt folyamatok
- billentyűzet, nyomtató puffer
- közlekedés szimulációja

SOR (QUEUE)

Sorműveletek

- **Sorba** (*put*): egy elem betétele a sorba (a sor „végére”). Csak akkor lehetséges, ha a sor még nem telt meg.
- **Sorból** (*get*): a sor „elején” található elem kivétele. Csak akkor tudunk elemet kivenni, ha a sor nem üres.
- **Inicializálás**:(*init*): alapállapotba helyezés, ürítés.
- **Üres-e**: igaz, ha a sorban egyetlen elemet sem tárolunk.
- **Teli-e**: igaz, ha a sorba már nem tehetünk újabb elemet, megtelt.
(Dinamikus megvalósítás esetén csak akkor van tele a sor, ha a rendelkezésünkre álló memória elfogyott.)

SOR (QUEUE)

Sor megvalósítása

A megvalósítás a tárolási módtól függően kétféle lehet:

- **statikus (tömb),**
- **dinamikus (láncolt lista).**

SOR (QUEUE)

Statikus sor

Statikus megvalósítás esetén a sorba tett elemeket egy vektorban tároljuk, az első, ill. az utolsó elem sorszámát (mutatóját), valamint a sorban lévő elemek darabszámát egy-egy egész értékben tároljuk.

Ha új elemet teszünk a sorba, a vége mutató, ha kiveszünk egy elemet, akkor pedig az eleje mutató értéke nő (ciklikusan) eggyel. A ciklikus növelés azt jelenti, hogy a maximális érték elérése után a következő lépésben 1 lesz az értéke. Ez a megoldás lehetővé teszi a vektor teljes kihasználását. (Ha a sorba helyezett elemekkel elértük a vektor végét, és közben vettünk ki az elejéről, akkor a sorba tett újabb elemek – fizikailag - a vektor elejére kerülnek)

A darabszám tárolása egyszerűsíti a megvalósítást.

SOR (QUEUE)

Dinamikus sor

Dinamikus sor esetén az elemeket (a dinamikus veremhez hasonló módon) láncoltan ábrázoljuk, minden elemhez tartozik egy mutató, amely a következő elemre mutat. A legutolsó elemhez tartozó mutató NULL (Java – nil).

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            //Sor létrehozása
            Queue<int> sor = new Queue<int>();

            //Elemek hozzáadása
            sor.Enqueue(1);
            sor.Enqueue(3);
            sor.Enqueue(4);

            //Összeg kiszámítása
            int osszeg = 0;
            int sorElem = 0;

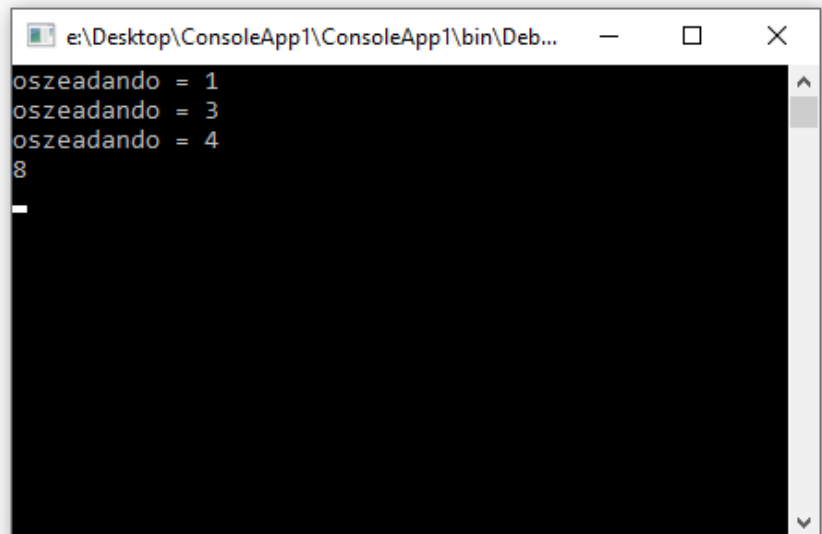
            while (sor.Count > 0)
            {
                sorElem = sor.Dequeue();
                Console.WriteLine($"oszeadando = {sorElem}");

                osszeg += sorElem;
            }

            Console.WriteLine(osszeg);

            Console.ReadKey();
        }
    }
}

```



```

e:\Desktop\ConsoleApp1\ConsoleApp1\bin\Deb...
oszeadando = 1
oszeadando = 3
oszeadando = 4
8

```

LISTA

adatszerkezet

készítette: Vastag Atila

2019

Különböző problémák számítógépes megoldása során gyakran van szükség olyan adatszerkezetre, amely nagyszámú, azonos típusú elem tárolására alkalmas, és nem jelent problémát a közvetlen elérés hiánya.

Azonban gyakran van szükség az adatszerkezet olyan szinten történő megváltoztatására, hogy bizonyos elemeit kitöröljük, illetve a már sorozatbeli elemek közé adott helyre újat vegyünk fel, lehetőleg kevés adatmozgatás elvégzésével. Ez utóbbi feltételezi, hogy nem feltétlenül ismert előre a sorozat elemszáma.

Lista adatszerkezet

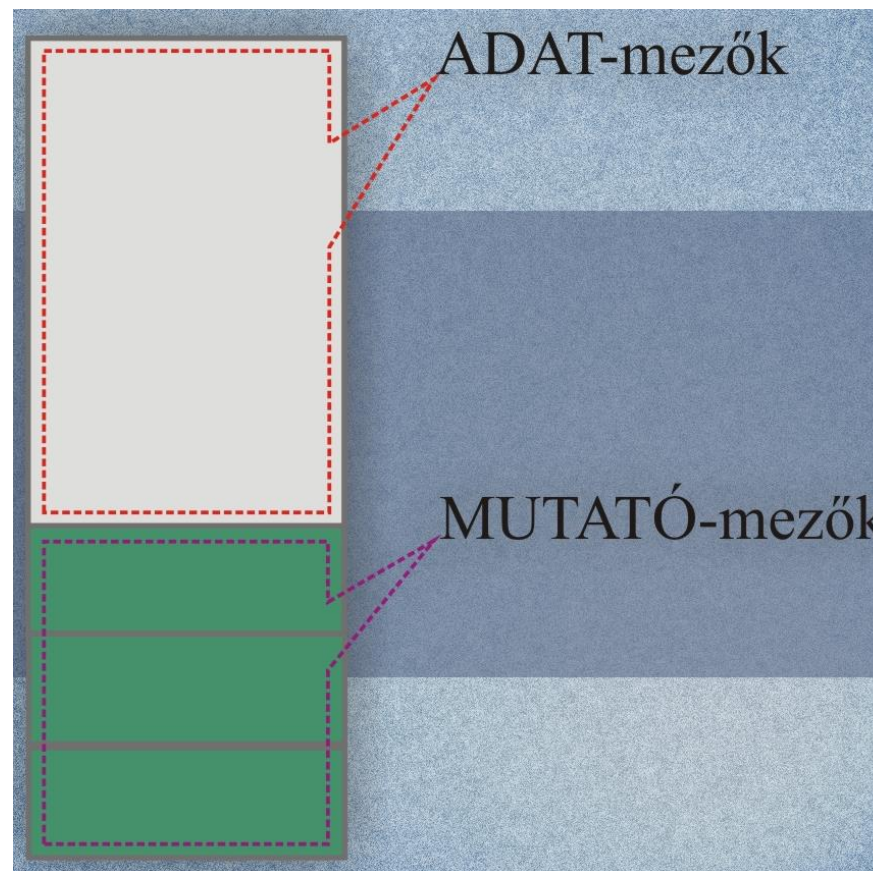
A lista elemek egymásutániságát jelenti. Fajtái:

- **statikus**
- **dinamikus lista:** *mérete az igényeknek megfelelően változik. Megvalósítása láncolt listával történik.*

A lista adatszerkezet jellemzői

A *láncolt lista* egyike a számítógép-programozásban használatos legegyszerűbb adatszerkezeteknek. Olyan csomópontok, cellák sorozatából épül fel, amelyek tetszőleges számú és egy fajtájú adatmezőt, és egy vagy két hivatkozást tárolnak. A hivatkozás(ok) a lista következő (és előző) elemére mutat(nak).

A *listaelem feladata kettős. Egyrészt helyet kell biztosítania a sokaság egy eleme számára, másrészt biztosítania kell a következő elem elérését, hiszen a memória tetszőleges területén lehet. Ennek megfelelően a listaelem funkcionálisan két részből áll: a tárolandó sorozat egy eleme és mutató a következő elem eléréséhez.*

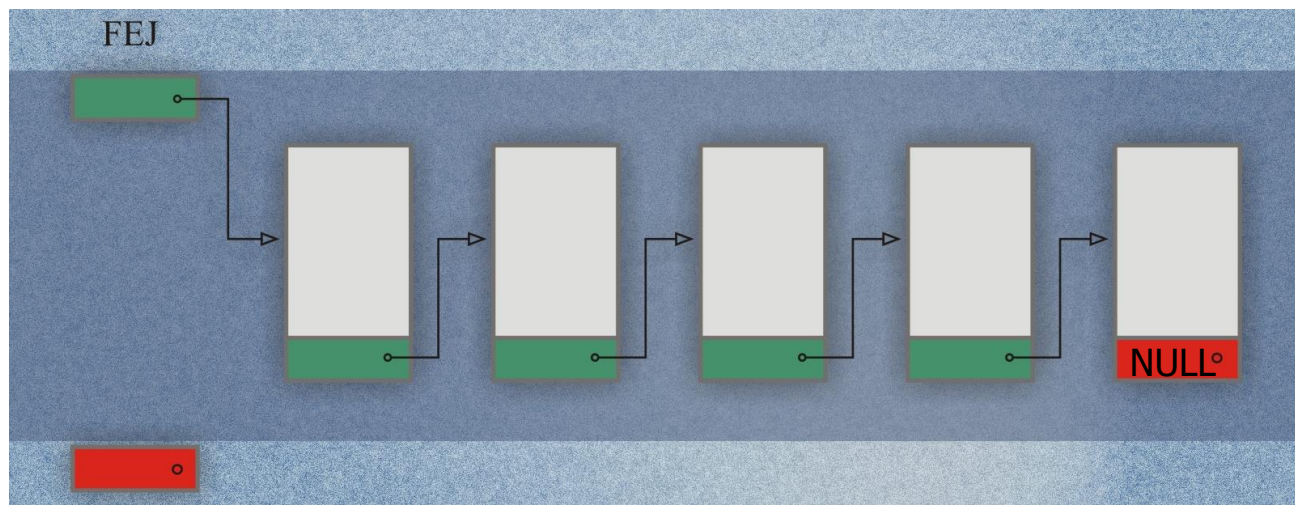


Tárolási módok:

- egyirányú láncolt lista
- cirkuláris láncolt lista
- kétirányú láncolt lista
- multilista

A láncolt lista olyan homogén, dinamikus, szekvenciális elérésű adatszerkezet, amelynek minden eleme - a tárolandó adatokon kívül - azt az információt is tárolja, hogy a következő elemet hol tárolja a rendszer a számítógép memóriájában.

A lánc első elemének a címét a lista feje tartalmazza. A listafej nem tartalmaz információs részt, azaz tényleges listabeli adatot. A lánc végét az jelzi, hogy az utolsó elemben a rákövetkező elem mutatója üres (NULL).



A lista első elemének eléréséhez csupán egy mutató is elegendő. Ennek ismerete a teljes lista ismeretét jelenti, mert így hozzáférhetünk az elemekben tárolt minden adathoz, tehát azt az információt is ki tudjuk olvasni, hogy hol található a következő elem.

A láncolt szerkezet lehetővé teszi listaelemek törlését és beszúrását a lista tetszőleges pontjára, ugyanakkor egy véletlenszerűen kiválasztott elem előkeresése a lista hosszával arányos időt igényel.

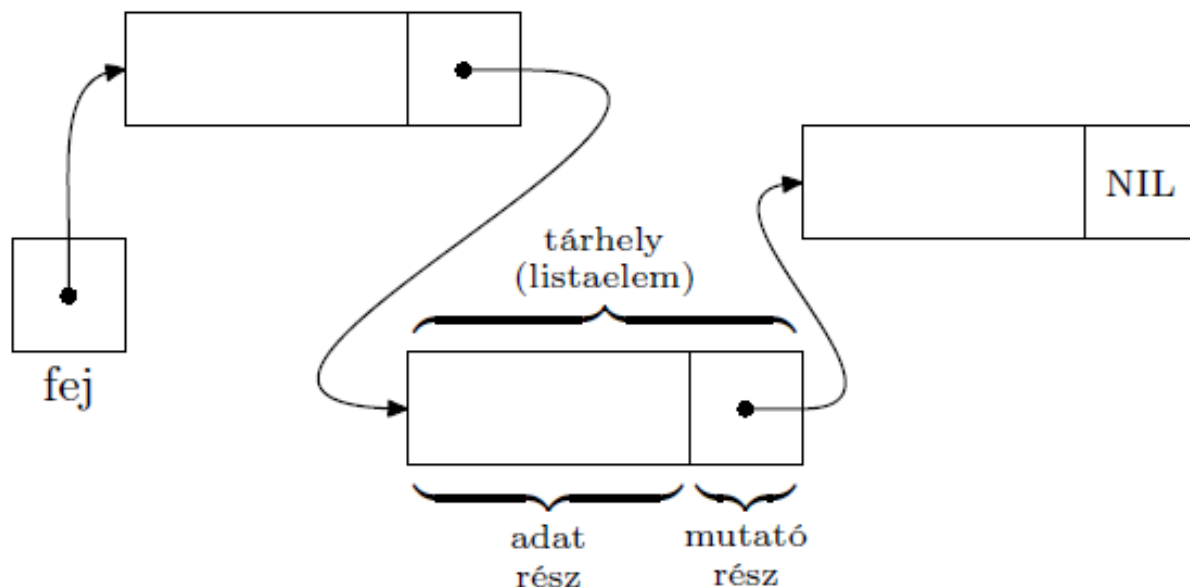
A láncolt listákat gyakran alkalmazzák összetettebb adatstruktúrák, mint például verem vagy sor építéskor. A cellák adatmezői tetszőleges adatot tárolhatnak, így akár hivatkozást is egy másik láncolt listára. Ezzel a trükkel már nem csak lineáris adatszerkezeteket tudunk építeni, hanem tetszőleges elágazó struktúrát, mint például fákat, gráfokat stb.

Új elem felvitele

Új elem felvitelét elvégezhetjük a lista elejére, végére, és adott elem után.

A lista végére történő *elem hozzáadásának lépései*:

- a tárolandó adat bekérése vagy előállítása,
- új listaelem létrehozása; érték beállítása; a következő elem mutatójának „nullázása”
- az új elem hozzáfűzése a lista végére
- az utolsó elem mutatójának ráállítása az új elemre



Lista kiírása, feldolgozása

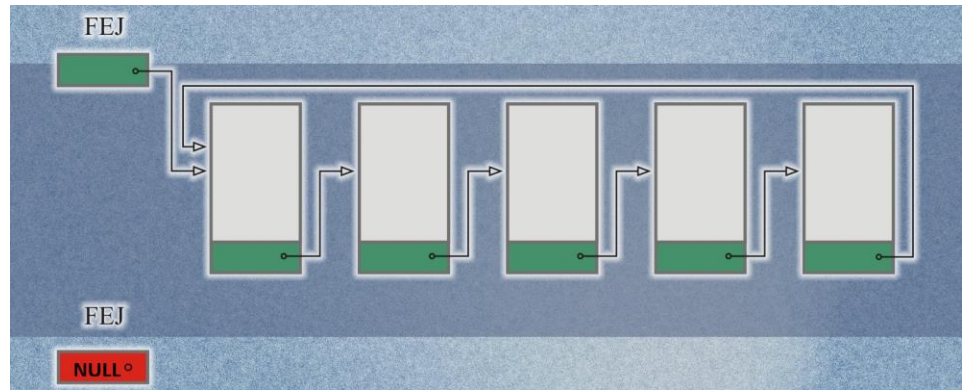
A lista elemein egyesével végig kell menni, és kiírni vagy feldolgozni a tárolt adatot. Az elemek eléréséhez egy segéd mutatót használunk, amelyet kezdetben az első elemre állítunk.

Első, utolsó elem törlése

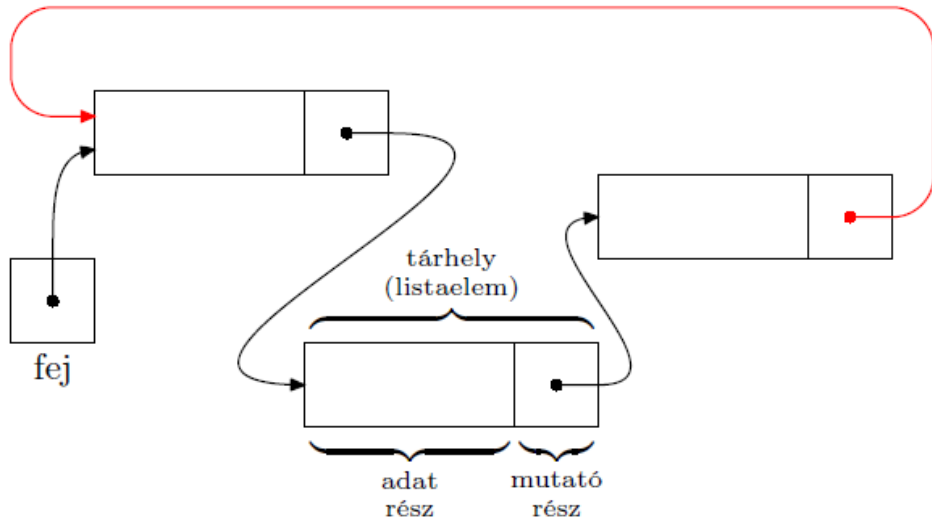
A lista első elemének törlésénél a „fej” mutatót állítjuk a második elemre, majd felszabadítjuk az első elem által lefoglalt memóriát.

A c#, Java nyelv esetén automatikus memória felszabadítás van (c, c++ nyelv esetén nincs). Azokat a memória területeket, amelyekre nem történik hivatkozás a rendszer „Garbage Collection” mechanizmusa felszabadítja. Szokták automatikus szemétdgyűjtésnek is nevezni.

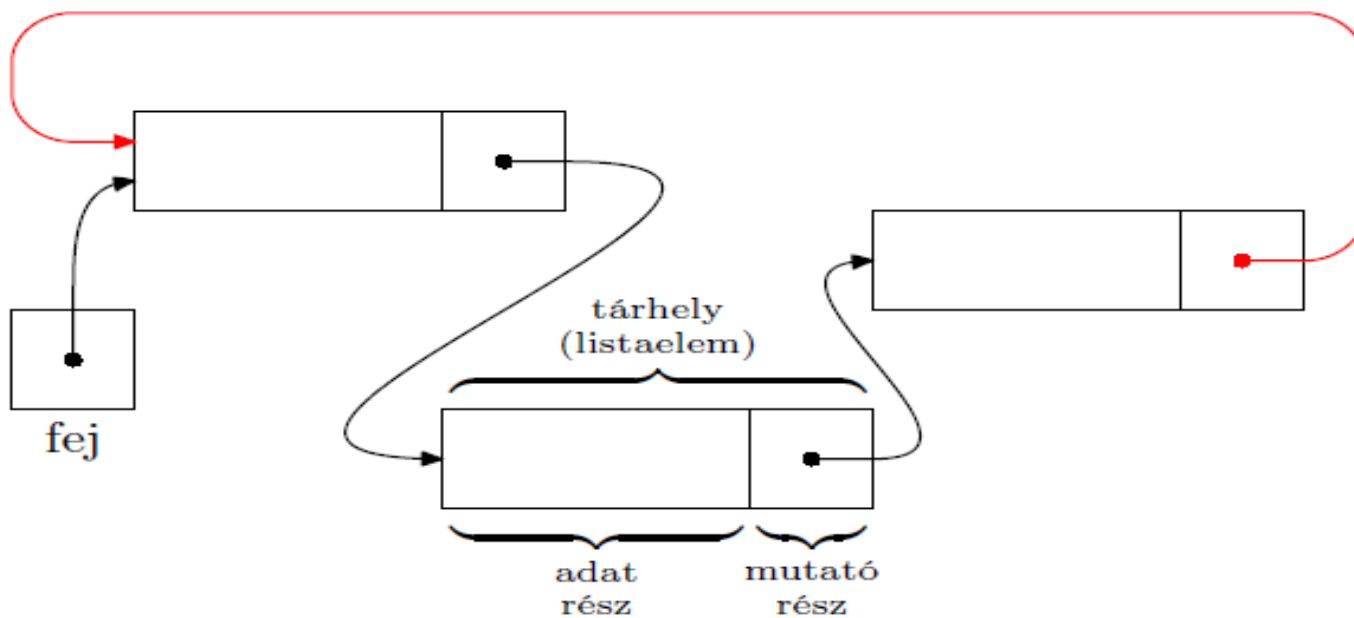
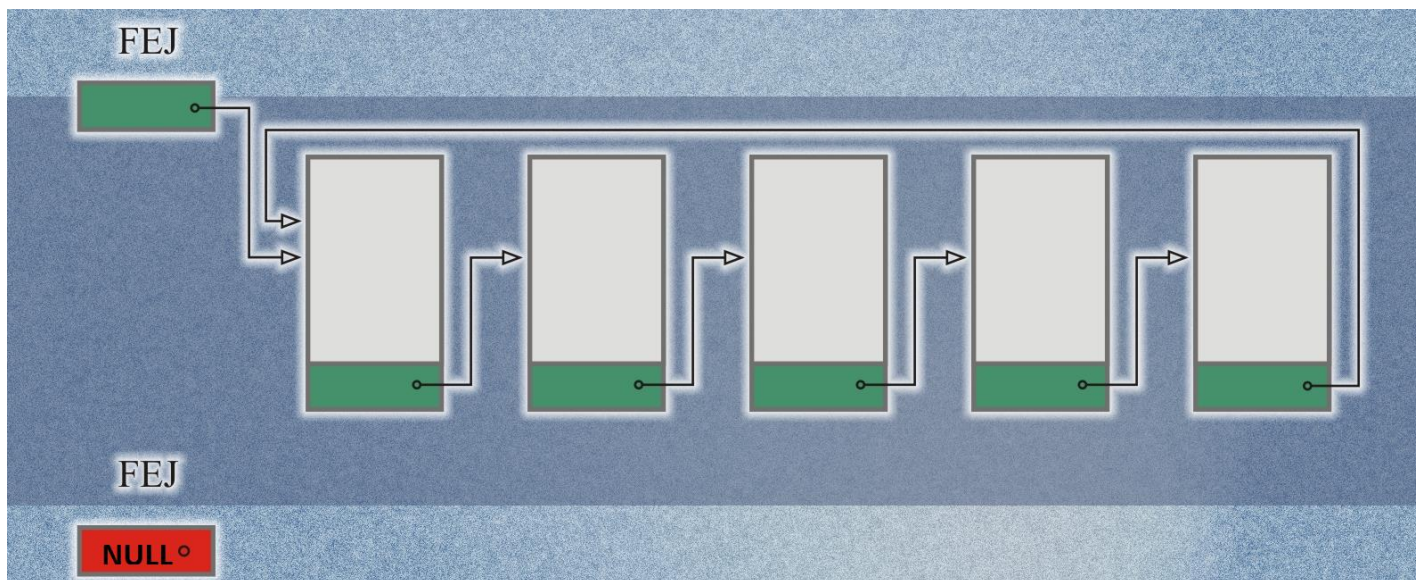
Hasonló az egy irányban láncolt listához, ám itt egyik listaelem mutatórésze sem tartalmazhatja a NULL értéket: az „utolsó” listaelem mutatórészébe az „első” listaelem címe kerül.



A ciklikus lista „első” elemének tárbeli címét most is egy mutató, a fejmutató tárolja. Amennyiben a fejmutató a NULL értéket tartalmazza, akkor a ciklikus lista üres.



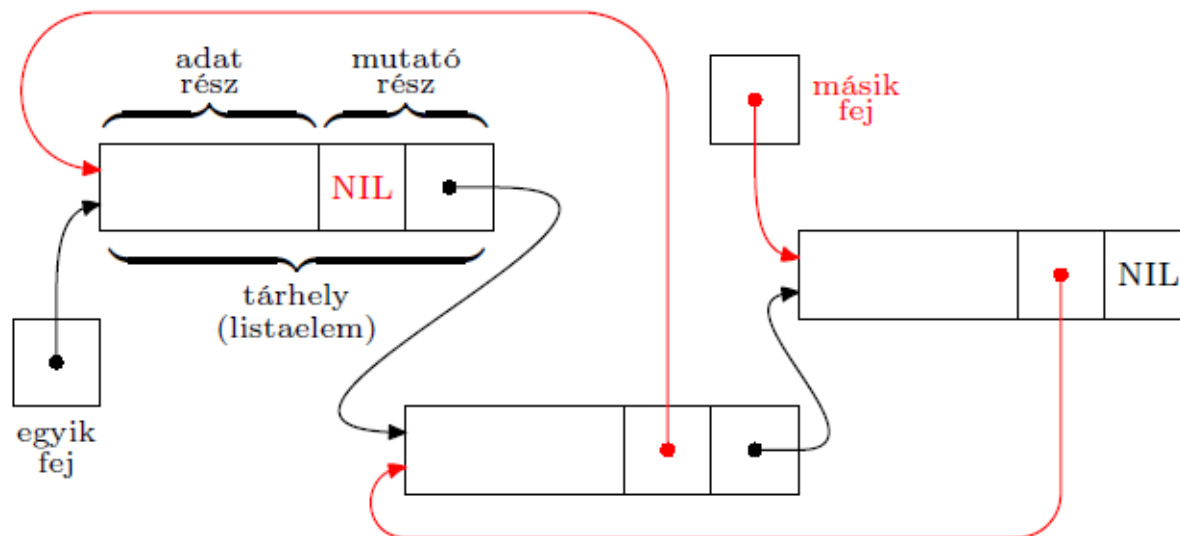
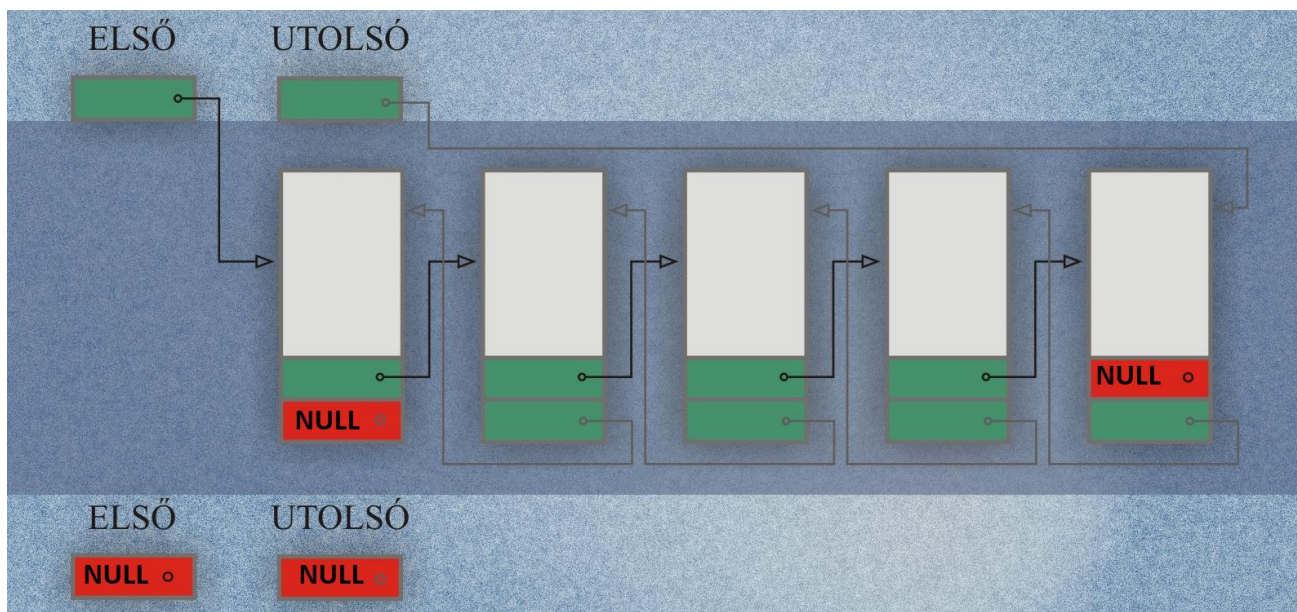
Ciklikusan láncolt lista



Hasonló az egy irányban láncolt listához, ám itt minden listaelem mutatórésze két részből áll: az egyik mutató az adott listaelemet megelőző, a másik az adott listaelemet követő listaelemre mutat. Ha üres a lista, a listát azonosító mutatók értéke végjel.

Két lánc alakul ki, két fejmutatóval. A fejmutatók a két irányban láncolt lista első és utolsó elemére mutatnak. Ha mindkét fejmutató értéke NIL, akkor a két irányban láncolt listának nincs egyetlen eleme sem, azaz üres.

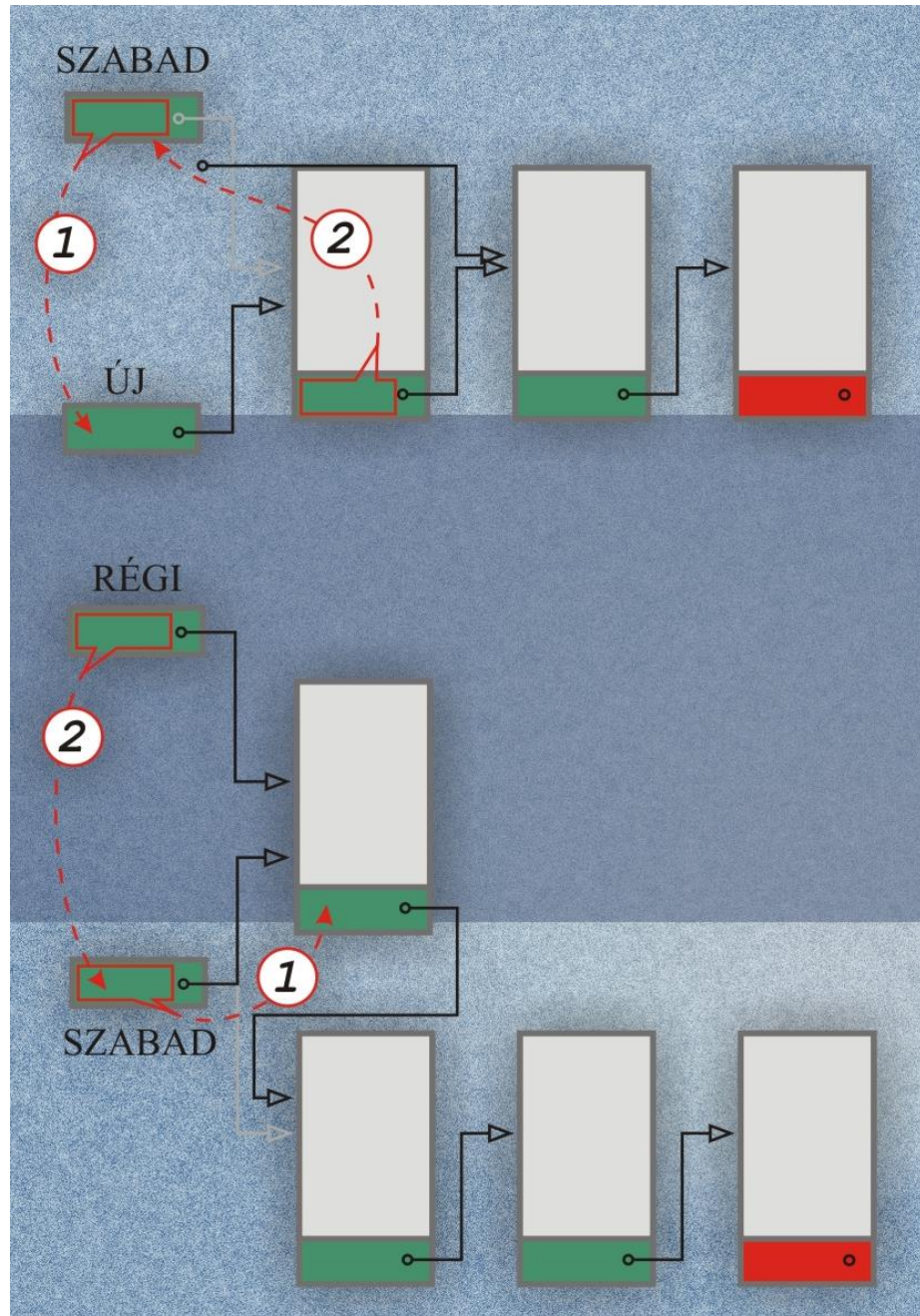
Két irányban láncolt lista



Szabad helyek kezelése

A lista-adatszerkezetek szervezéséből adódóan a listaelemek a memória különböző területein, szétszórtan helyezkedhetnek el (különösen kellő számú új elemmel való bővítés és feleslegessé vált elem listából való elhagyása után). Ez természetesen nem okoz gondot, mert az egyes listaelemek elérését az őket megelőző elemek mutatói biztosítják. A listaelemekkel végezhető műveletek (lista bővítése egy elemmel, feleslegessé vált elem elhagyása a listából) azt feltételezik, hogy a listából törölt elemek is szétszórtan helyezkednek el a memóriában. Az így felszabadult elemekre a későbbiekben egy újabb bővítés alkalmával még szükség lehet, hiszen a lista dinamikus adatszerkezet. Meg kell tehát oldani a szabad helyek kezelését. Ennek legkézenfekvőbb megvalósítása, hogy a szabaddá vált listaelemeket egyszerűen egy listába fűzve tároljuk.

Szabad helyek kezelése



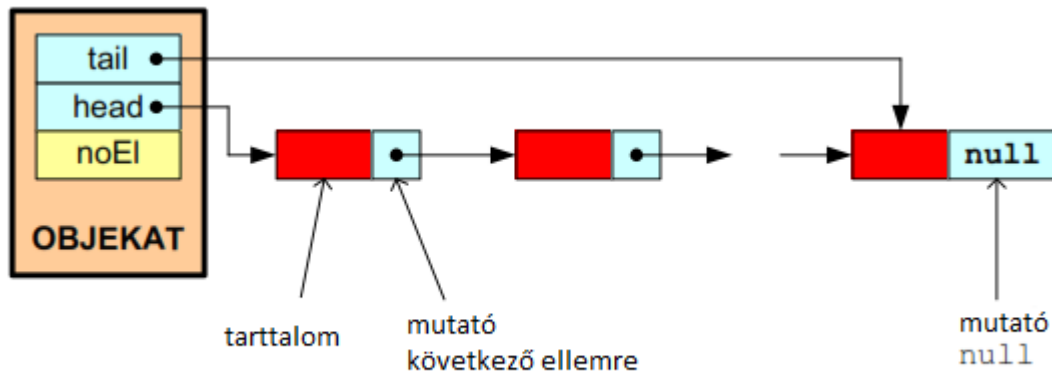
LISTÁK

dinamikus adat tömbök c#-ban

(generikus)

készítette: Vastag Atila

2017



Megvalósítás c++

```
template <class T>
ostream& operator<<(ostream &
out, const List<T> &l){
    out<<endl;
    out<<"-----"<<endl;
    for(int i=1;i<=l.size();i++){
        if(i!=1) out<<" ";
        T res;
        l.read(i,res);
        out<<res;
    }
    out<<endl<<"-----"<<endl;
    return out;
}
```


- A listák olyan összetett, homogén adatszerkezetek, amelyek dinamikus mérettel rendelkeznek, és jellemzően szétszórt reprezentációjúak. Adathozzáférési stratégiájuk pedig szekvenciális elérésű.
- A C#-ban a listák a fentiekkel ellentétben folytonos reprezentációjúak, és véletlen elérésű a hozzáférési stratégiájuk, vagyis minden listaelemhez egyforma sebességgel tudunk hozzáférni.
- A listák a vektorokhoz hasonlóan kezelendők, használhatjuk rajta az indexelő operátort is
- A típusos lista létrehozásához pedig a **List<T>** alakot használjuk, mely a generikus List osztályból készíti el a listát.
- A lista hossza példányosítás után 0, vagyis nem tartalmaznak elemet.

LISTA példányosítása

```
List<T> myListT = new List<T>();
```

T generikus típus, azt jelenti, hogy a létrehozott listánk milyen adat típusú. A listák, ugyanúgy mint a tömbök, csak egy bizonyos típusúak lehetnek, ezt jelöli a **T**.

A **T** lehet:

- primitív adat típus: *int, double, char, string, float, ...*
- struktúra: *struct*
- objektum: valamilyen osztály *típus - class*
- *listába ágyazott lista*

Egy elem hozzáadása a lista végére

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

```
int x = 5;
```

```
intList.Add(x);
```

A példányosított listán (**intList**), az **Add()** metódus segítségével adtunk elemet, (**x**), a lista végére.

Egy elem beszúrása a listába

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

```
int x = 5;
```

```
intList.Insert(0, x);
```



A példányosított listán (**intList**), az **Insert**(pozíció, **x**) metódus segítségével tudunk elemet a beszúrni a listába.

*Mivel a lista első eleme **0-dik** helyen van, ezért ha már egy adatokat tartalmazó lista elejére szeretnénk új elemet elhelyezni, akkor azt a 0-dik pozícióra kell tenni.*

Ugyanezzel a módszerrel tudunk a lista bármelyik helyére elemet beszúrni.

Lista hozzáadása a lista végére

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

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

```
intList.AddRange(x);
```

A példányosított listán (**intList**), az **AddRange()** metódus segítségével tudjuk a lista végére beszúrni az új listát (**x**).

Egy elem beszúrása a listába

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

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

```
intList.InsertRange(0, x);
```



A példányosított listán (**intList**), az **InsertRange**(**pozíció**, **x**) metódus segítségével tudjuk a listát (**x**) beszúrni a másik lista elejére.

Ugyanezzel a módszerrel tudunk a lista bármelyik helyére elemet beszúrni.

```
intList.InsertRange(6, x);
```

az (**x**) listát az **intList**-be szúrja a **6**-dik helytől kezdve

Lista kiírása

```
using System;
using System.Collections.Generic;
class Program
{
    static public void Main()
    {
        List<int> intList = new List<int>();

        for(int i = 0; i < 10; ++i)
        {
            list.Add(i);
        }

        foreach(int item in intList)
        {
            Console.WriteLine(item);
        }
    }
}
```

A **foreach** végiglépked szekvenciálisan a listán, ahol az **item** minden ciklus körben meg fog felelni az **intList** épp soron követő elemének.

Lista kiírása

```
using System;
using System.Collections.Generic;
class Program
{
    static public void Main()
    {
        List<int> intList = new List<int>();

        for(int i = 0; i < 10; ++i)
        {
            list.Add(i);
        }

        for(int i = 0; i < intList.Count; i++)
        {
            Console.WriteLine(intList[i]);
        }
    }
}
```

A **for** ciklust használva szekvenciálisan végiglépkedünk a listán, ahol az **i** index segítségével minden ciklus körben hozzá tudunk az **intList** elemeihez ugyanúgy férni mint egy hagyományos tömbben.

List függvényei

`void Clear()` – törli a lista tartalmát

`int IndexOf(T item)` – A `T` típusú elem (melynek meg kell egyeznie a lista típusával) indexét (sorszámát) adja vissza a halmazból

`void RemoveAt(int index)` – az *index* helyen lévő elem eltávolítása a halmazból

`void Reverse()` – megfordítja a lista tartalmát

`T[] ToArray()` –a listát tömbé alakítja

Ezekon a függvényeken kívül léteznek még más függvények is.

A listák igazi erejét igazán a LAMBDA kifejezések és a LINQ utasítások teszik.

Rólluk később lesz majd szó.

Egy lista csak azonos típusú adatokból állhat c#-ban.

(ezt mondtuk)

IGAZ

HAMIS

A **c#-ban** rendelkezésünkre áll egy típus : **dynamic**.

A dynamic típus lehetővé teszi bármely adatípus eltárolását, akár összetettet is (objektum, struktúra, lista, tömb).

Fordítási időben a fordító nem fog típushibát jelezni, de futási időben igen, ha nincs kezelve, mivel nem tudjuk épp milyen típusú adattal dolgozunk.


```

static void Main(string[] args)
{
    List<dynamic> dynamicList = new List<dynamic>()
    {
        1,
        6.66,
        'a',
        "bla",
        true
    };

    Console.WriteLine("A dinamikus lista elemei: ");

    foreach(dynamic listaElem in dynamicList)
    {
        Console.WriteLine($"{listaElem}\t\t{listaElem.GetType()}");
    }

    Console.ReadKey();
}

```

```

A dinamikus lista elemei:
1                System.Int32
6.66             System.Double
a                System.Char
bla              System.String
True             System.Boolean

```

Dictionary

A *Dictionary* kulcs-érték párokat tárol. Amikor hozzáadunk egy új párt, a kulcs alapján valahogy elhelyezi az adatot a memóriában. Amikor kíváncsiak vagyunk egy értékre, akkor a kulcs alapján kereshetünk rá; vagyis megadunk egy kulcsot, és ha az szerepel a szótárban, akkor valahogy előkeresi, és visszaadja a hozzárendelt értéket. Ha nem szerepel, akkor kapunk egy *KeyNotFoundException*. Ha egy olyan kulcsot próbálunk hozzáadni a szótárhoz, ami már szerepel benne, akkor a szótár ezt valahogy megérzi. Dob egy kivételt, mert egy kulcs csak egyszer szerepelhet.

A szótár elempárok tárolására szolgál, melyek közül egyik a kulcs (**TKey**), amely azonosítja az elempárt, másik az érték.

Minden kulcs egyedi.

Gyakorlatilag a szótár úgy viselkedik, mint egy lista, de az elemek indexe itt tetszőleges típusú lehet (**TKey**).

Konstruktor generikus, paraméter nélküli:

- Dictionary(): létrehoz egy szótárt, ahol **TKey** a kulcs, **TValue** az érték típusa

Fő metódusai:

- *void* **Add**(TKey key, TValue value): a value érték hozzáadása key kulccsal
- *bool* **ContainsKey**(TKey key): megadja, hogy szerepel-e egy kulcs a szótárban
- *bool* **ContainsValue**(TValue value): megadja, hogy szerepel-e egy érték a szótárban
- *bool* **Remove**(TKey key): eltávolít egy elemet a szótárból, a visszatérési érték a művelet sikerességét jelzi
- *int* **Count** { get; }: az elemek (kulcsok) száma
- *void* **Clear**(): szótár kiürítése

```
Random rnd = new Random();
char[] karakterek = new char[23];

//Szótár létrehozása
Dictionary<char, int> karakterekSzama = new Dictionary<char, int>();

for (int i = 0; i < 23; i++)
{
    karakterek[i] = (char)rnd.Next(65, 90);

    Console.Write("Char: {0}\t", karakterek[i]);
}

Console.Write(Environment.NewLine);

//Megszámoljuk, hogy hogy az egyforma karakterek hányszor fordulna elő
for (int i = 0; i < 23; i++)
{
    char key = karakterek[i];
    if (karakterekSzama.ContainsKey(key))
    {
        karakterekSzama[key]++;
    }
    else
    {
        karakterekSzama.Add(key, 1);
    }
}

//Kiíratjuk az értékeket
foreach (KeyValuePair<char, int> elem in karakterekSzama)
{
    Console.WriteLine("char: {0} -> {1}", elem.Key, elem.Value);
}
```

A szótár elemei a [] operátorral érhetőek el.

- A szótárat foreach ciklussal lehet végig olvasni, amellyel a szótárból *KeyValuePair* típusú elemeket kapunk. Ezek *Key* és *Value* mezői adják a megfelelő kulcs és érték párokat.

1 - Egy számokat tartalmazó listát töltünk fel random számokkal, úgy hogy a program kezelője a program elején megadja, hogy hány elemre lehet számítani ($5 < n < 10$). A program a bevitel közben mindig írja ki, hogy hányadik számnál tart a bevitel, és mennyi van még hátra! A program a bevitel végén írja vissza az adatokat a képernyőre, a számokat egymás mellé írva, szóközzel elválasztva, majd adja meg a számok összegét!

2 - Egy törtszámokat tartalmazó listát töltünk fel billentyűzetről oly módon, hogy a program kezelője a program elején nem adja meg, hogy hány adat lesz! Helyette választunk egy speciális karaktert, pl. @. Amennyiben ezt a karaktert íránk be, úgy a program fejezze be a bevitelt! A program a bevitel végén írja vissza az adatokat a képernyőre, a számokat egymás mellé írva, szóközzel elválasztva, majd adja meg a számok átlagát!

3 - Egy számokat tartalmazó listát töltünk fel random számokkal, úgy hogy a program kezelője a program elején megadja, hogy hány elemre lehet számítani ($5 < n < 10$). A program a bevitel végén írja vissza az adatokat a képernyőre, a számokat egymás mellé írva, szóközzel elválasztva, majd adja meg a számok min és max értékét!

4 – Két számokat tartalmazó listát töltünk fel random számokkal, úgy hogy a program kezelője a program elején megadja, hogy hány elemre lehet számítani ($5 < n < 10$). Rakjuk az első számsort növekvő sorrendbe, a másodikat csökkenő sorrendbe majd írjuk ki őket. Végül a növekvő lista végére illesszük be a csökkenőt.

5 – Két számokat tartalmazó listát töltünk fel random számokkal, úgy hogy a program kezelője a program elején megadja, hogy hány elemre lehet számítani ($1 < n < 5$) az első lista még a második ($5 < n < 10$) eleméig rendelkeznek.

- Fűzzük össze őket, rakjuk növekvő sorrendbe és írjuk ki.
- Határozzuk meg az összefűzött tömb átlagát és írjuk ki.

6 - Egy egész számokat tartalmazó listát töltünk fel billentyűzetről oly módon, hogy a program kezelője egy időben egyszerre több számot is beírhat, vesszővel elválasztva! Ekkor minden számot adjunk hozzá a listához! Ha csak egy számot ír be a kezelő, akkor azt az egy számot. A program a bevitel végén írja vissza az adatokat a képernyőre, a számokat egymás mellé írva, kötőjellel elválasztva, növekvő sorrendbe.

7 - Egy egész számokat tartalmazó listát töltünk fel billentyűzetről oly módon, hogy a program kezelője addig írhat be számokat, amíg azok összege el nem éri az előre megadott értéket (például 100)! A program a bevitel közben folyamatosan figyelje az összeghatárt, illetve mindig írja ki, hogy hányadik számnál tart a bevitel, hol tart az összeg, mennyi van még hátra az értékhatárig! A program a bevitel végén írja vissza azokat a bevitt számokat melyek oszthatóak 6-al!

8 - Egy egész *char* listát töltünk fel random (0 – 255 közt) a felhasználó által megadott mennyiséggel! A program határozza meg:

- Hány szám volt a sorozatban, és ez hány százalékot tett ki
- Hány betű volt a sorozatban, és ez hány százalékot tett ki
- Hány karakter volt a sorozatban, és ez hány százalékot tett ki

9 - Egy egész számokat tartalmazó listát töltünk fel billentyűzetről oly módon, hogy a listába nem kerülhet be ugyanazon szám többször is! Ha a program kezelője olyan számot írna be, amely már volt, akkor a program ezt jelezze ki! A bevittet a kezelő akkor fejezheti be, ha sikerült neki egymás után háromszor is olyan értéket beírni, amely még nem szerepelt korábban (amit a program elfogad). A program a bevétel végén írja vissza az adatokat a képernyőre, a számokat egymás mellé írva, vesszővel elválasztva, és hogy hány alkalommal adott meg a felhasználó olyan számot ami már volt!

10 - Egy neveket tartalmazó listát töltünk fel billentyűzetről oly módon, hogy a listába nem kerülhet be ugyanazon név többször is! Ha a program kezelője olyan nevet írna be, amely már volt, akkor a program ezt jelezze ki! A bevittet a kezelő akkor fejezheti be, ha beírja, hogy: „@”. A program a bevétel végén rakja ABC sorba a neveket és írja vissza az adatokat a képernyőre.

11 – Egy egész számokat tartalmazó listát töltünk fel adatokkal, úgy hogy a felhasználó adja meg, hány elemet tartalmazzon a lista (min 10 elem, max 50). Majd határozzuk meg a lista elemeinek:

- Összegét
- Különbségét
- Legkisebb értékét
- Legnagyobb értékét
- Átlagát
- Írjuk ki az elemeket növekvő sorrendben
- Írjuk ki az elemeket csökkenő sorrendben
- Átlagot
- Generáljunk egy random számot és ellenőrizzük szerepel e a listában

12 – A felhasználónak meg kell adnia két különböző nagyságú listát (A és B), de egyiknek sem lehet kevesebb eleme mint öt. Ez a két listát töltjük fel adatokkal (véletlen számok -9 és 9 közt)

a) Határozzuk meg a két lista unióját: $A \cup B$

b) Határozzuk meg a két lista metszetét: $A \cap B$

Írjuk ki az értékeket a képernyőre!

Ügyeljünk arra, hogy az unió és metszet listákban ne szerepeljen egyetlen szám sem kétszer!

13 - Egy szöveges fájl soronként egy vagy több törtszámot tartalmaz. Amikor több szám is szerepel egy sorban, akkor vesszővel vannak elválasztva. A vessző után szóközök is szerepelhetnek a jobb vizuális tördelés érdekében. A text fájl két számlistát tartalmaz, a két számlista között egy üres sor szerepel a fájlban. A program olvassa be mindkét számlistát két különböző listaváltozóba! Adjuk meg, melyik számlistában szereplő számoknak nagyobb az átlaga!

14 – Olvassunk be tetszőleges számú szavat a konzolról. Majd kérjünk be egy új szót a felhasználótól és kérdezzük meg, hanyadik helyre tegye a beolvasott szavak közt. Ezt a szót nagybetűkkel írassuk ki a végén a többi szóval együtt beolvasási sorrendbe.

Feladatok VII.

Program "*" végjelig kérje be egy kézilabda a csapat tagjainak nevét, de max 20-ig, és a bajnokságban szerzett góljainak számát számát!

Írja ki:

1. Hány versenyző teljesített átlag alatt?
2. Listázza ki az átlag felett teljesítők nevét és szerzett góljainak számát!
3. Ki szerezte a legkevesebb gólt és mennyit?

Írjon programot, amely "*" végjelig bekéri n tanuló nevét és megtakarított pénzét!

1. Írja ki a megtakarítások összegét!
2. Írja ki az átlagos megtakarítást!
3. Kinek van a legnagyobb megtakarítása és mennyi?
4. Kinek van a legkisebb megtakarítása és mennyi?
5. Hány főnek van 2000 Ft felletti megtakarítása?
6. Van-e olyan tanuló akinek nincs megtakarítása?
7. Listázza ki azon tanulók nevét, akiknek a megtakarítása átlag alatti!

Feladatok VII.

Egy forgalom ellenőrző ponton "n" számú autó haladt át, a program * végjelig kérje be az autók rendszámát és sebességét!

Írja ki azon autók rendszámát, sebességét és büntetését, amelyek 90 km/h felett közlekedtek.

- Büntetési tételek
- 91-100 km/h 10 ezer FT
- 101-110 km/h 20 ezer FT
- 110 km/h felett 30 ezer FT
- Írja ki a hatóság összesen mekkora büntetést szabott ki?
- Írja ki a leggyorsabb autó rendszámát sebességét!
- Írja ki hány autó közlekedett szabályosan és ez az összes áthaladó autó hány százaléka?
- Írja ki, hogy volt-e 60 km/h-val közlekedő autó?

Feladatok VII.

Egy üzemanyag kútnál "n" számú autó tankolt. A program "*" végjelig kérje be az autók rendszámát és a tankolt mennyiséget! Írja ki, azon autók rendszámát és a tankolt mennyiséget amelyek 40 liter felett tankoltak!

- Írja ki, az autók összesen hány liter üzemanyagot tankoltak?
- Írja ki, melyik autó tankolt a legtöbbet és mennyit!
- Írja ki, melyik autó tankolt a legkevesebbet és mennyit!
- Írja ki, hány autó tankolt 30 liter alatt?
- Írja ki, volt-e olyan autó, amely pontosan 50 litert tankolt?

Feladatok VII.

Program gyümölcsraktár készletének nyilvántartásához.

Kérje be (*) végjelig n számú gyümölcs nevét, mennyiségét és egységárát! (max 10)

- Írja ki összesen hány kg gyümölcs van készleten!
- Írja ki gyümölcs fajtánként a készletértéket (mennyiség egységár szorzata)!
- Írja ki a teljes készletértéket!
- Írja ki a legdrágább gyümölcs nevét és árát!
- Melyik gyümölcsből van a legkevesebb a készleten és mennyi?
- Listázza ki azon gyümölcsök adatait, ahol a mennyiség 100 kg alatti!
- Van-e olyan gyümölcs a nyilvántartásban, amelynek egységára 1500 Ft feletti?

Feladatok VII.

Program végjelig kérje be a kosárlabda csapat tagjainak nevét és a bajnokságban szerzett pontjainak számát, írja ki:

- a csapat a szezonban összesen hány pontot szerzett,
- ki szerezte a legtöbb pontot és mennyit,
- ki szerezte a legkevesebbet és mennyit,
- hány versenyző teljesített átlag alatt,
- listázza ki az átlag felett teljesítők nevét és pontszámát.

<http://redqueen.uw.hu/modules.php?name=Sections&op=viewarticle&artid=57>

http://www.szerencsiszakkepzo.sulinet.hu/jegyzet/info/verem_adatszerkezet_NZS.pdf

<https://chevenix.wordpress.com/2011/04/16/hogyan-mukodik-a-dictionary/>

<http://www.inczedy.hu/~szikszai/szg/esti/lista.pdf>