# Adatszerkezetek

9. hét – Dinamikus adatszerkezetek: a memóriakezelés modellje és az egyszerű láncolt lista

Dr. Koczka Ferenc · Eszterházy Károly Egyetem · 2026

# Ismétlés

- Eddigi adatszerkezeteink **tömbön** alapultak: folytonos tárolás, közvetlen elérés
- Verem, sor: a tömb **végein** dolgozunk → O(1)
- **Mutató / referencia** (1. hét): egy memóriacímet tároló adat

## A mai óra kérdései

- Mik a statikus (tömbös) tárolás korlátai?
- Hogyan foglal és szabadít fel memóriát egy program?
- Hogyan épül fel a **láncolt lista**, és milyen gyorsak a műveletei?

# A statikus adatszerkezetek korlátai

- A méretet **előre** meg kell adni → túl kicsi vagy pazarló
- **Folytonos** szabad memóriaterület kell – nagy tömbnél ez nem mindig van
- Beszúrás és törlés a közepén **O(n)** – elemeket kell mozgatni
- A dinamikus tömb (3. hét) segít, de átméretezéskor **mindent át kell másolni**

> **Ötlet:** tároljuk az elemeket **szétszórtan**, bárhol a memóriában, és mindegyik elem **tudja, hol van a következő**.

# A dinamikus memóriakezelés modellje

# Egy egyszerű modell

- A **memória** egy nagy tömb, amelynek elemei rekordok: **Érték** és **Köv** (következő) mezővel
- A **mutató** egy **index** ebbe a tömbbe – a speciális **Nil** érték: „nem mutat sehova”
- Jelölés: `Elem[P].Érték`, `Elem[P].Köv` – a P mutató által mutatott elem mezői

| index | Érték | Köv |
|---|---|---|
| 1 | C | 5 |
| 2 | | 4 |
| 3 | A | 7 |
| 4 | | 6 |
| 5 | D | Nil |
| 6 | | 8 |
| 7 | B | 1 |
| 8 | | Nil |




**A lista (Fej-től):**

`3 → 7 → 1 → 5 → Nil`

`A → B → C → D`

**Szabad helyek listája:**

`2 → 4 → 6 → 8 → Nil`

*A „mutató” itt egy tömbindex –*
*a valódi memóriában egy memóriacím.*

# Lefoglalás és felszabadítás

A szabad helyeket is egy **listában** tartjuk nyilván (Szabad).

```
Függvény Lefoglal(P)
  Ha Szabad = Nil akkor
    Vissza: hamis       // betelt
  különben
    P ← Szabad
    Szabad ← Elem[P].Köv
    Vissza: igaz
  Elágazás vége
Függvény vége
```

```
Eljárás Felszabadít(P)
  Elem[P].Köv ← Szabad
  Szabad ← P
Eljárás vége
```

- Mindkettő **O(1)**
- A szabad lista **elejéről** veszünk, és oda teszünk vissza – mint egy verem!

A gyakorlatban a szabad memóriát az operációs rendszer és a futtatókörnyezet kezeli – a régi beadandóban is el lehetett tekinteni a megvalósításától.

# A valóságban

| | Lefoglalás | Felszabadítás |
|---|---|---|
| C | `p = malloc(sizeof(Node))` | `free(p)` – a programozó felelőssége |
| **Java** | `new Node(x)` | **automatikus** – szemétgyűjtő (GC) |
| **Go** | `&Node{Value: x}` vagy `new(Node)` | **automatikus** – szemétgyűjtő (GC) |

## Tipikus hibák kézi memóriakezelésnél:

- **Memóriaszivárgás:** a lefoglalt területet nem szabadítjuk fel → a program egyre több memóriát foglal
- **Lógó mutató:** felszabadított területre mutatunk, és használjuk
- A szemétgyűjtő ezeket megelőzi: amire **nem mutat** már semmi, azt felszabadítja

# Dinamikus adatszerkezetek: előnyök és árak

## Előnyök

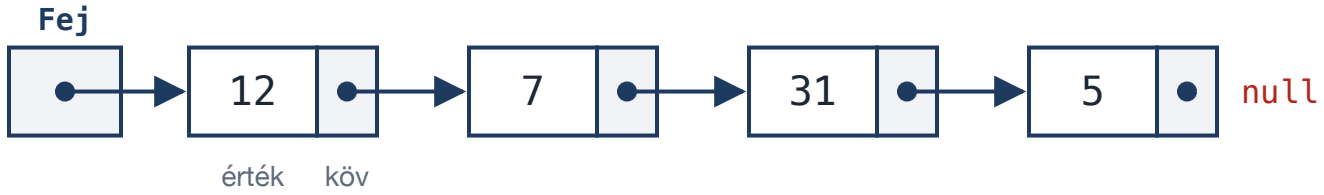
- Annyi memóriát foglal, amennyi **éppen kell** → a memória **megosztható** több adatszerkezet között (erőforrás-megosztás)
- Nem kell folytonos terület
- Beszúrás és törlés **mutatók átállításával**, elemmozgatás nélkül

## Árak

- Minden elem tárol **egy vagy több mutatót** is → nagyobb tárigény
- **Nincs közvetlen elérés** – az **i**-edik elemhez végig kell menni az előzőkön
- A lefoglalás/felszabadítás és a mutatókövetés **időbe kerül**

# Az egyszerű láncolt lista

# A láncolt lista felépítése




- Minden **elem**: érték + mutató a **következő** elemre
- **Fej:** mutató az első elemre · az utolsó elem `Köv` mezője **Nil**
- **Üres lista:** Fej = Nil

| Tulajdonság | Láncolt lista |
|---|---|
| Elemek | homogén |
| Méret | **dinamikus** |
| Tárolás | **szétszórt** |
| Elérés | **szekvenciális** |

# Listaelem Java-ban és Go-ban

**Java**

```java
class Node {
    int value;
    Node next;      // referencia a következőre

    Node(int value, Node next) {
        this.value = value;
        this.next = next;
    }
}

Node head = null;   // üres lista
```

**Go**

```go
type Node struct {
    Value int
    Next  *Node     // mutató a következőre
}

type List struct {
    Head *Node      // nil: üres lista
}
```

| Pszeudokód | Java | Go |
|---|---|---|
| `Elem[P].Érték` · `Elem[P].Köv` | `p.value` · `p.next` | `p.Value` · `p.Next` |
| `Nil` · `Fej` | `null` · `head` | `nil` · `l.Head` |

# Bejárás és keresés

**Bejárás (feldolgozás)**

```
Eljárás Bejár(Fej)
  P ← Fej
  Ciklus amíg P ≠ Nil
    Feldolgoz(Elem[P].Érték)
    P ← Elem[P].Köv
  Ciklus vége
Eljárás vége
```

**Keresés**

```
Függvény Keres(Fej, x)
  P ← Fej
  Ciklus amíg P ≠ Nil és Elem[P].Érték ≠ x
    P ← Elem[P].Köv
  Ciklus vége
  Vissza: P            // Nil, ha nincs benne
Függvény vége
```

**Java**

```java
for (Node p = head; p != null; p = p.next)
    System.out.println(p.value);

Node find(int x) {
    Node p = head;
    while (p != null && p.value != x)
        p = p.next;
    return p;
}
```

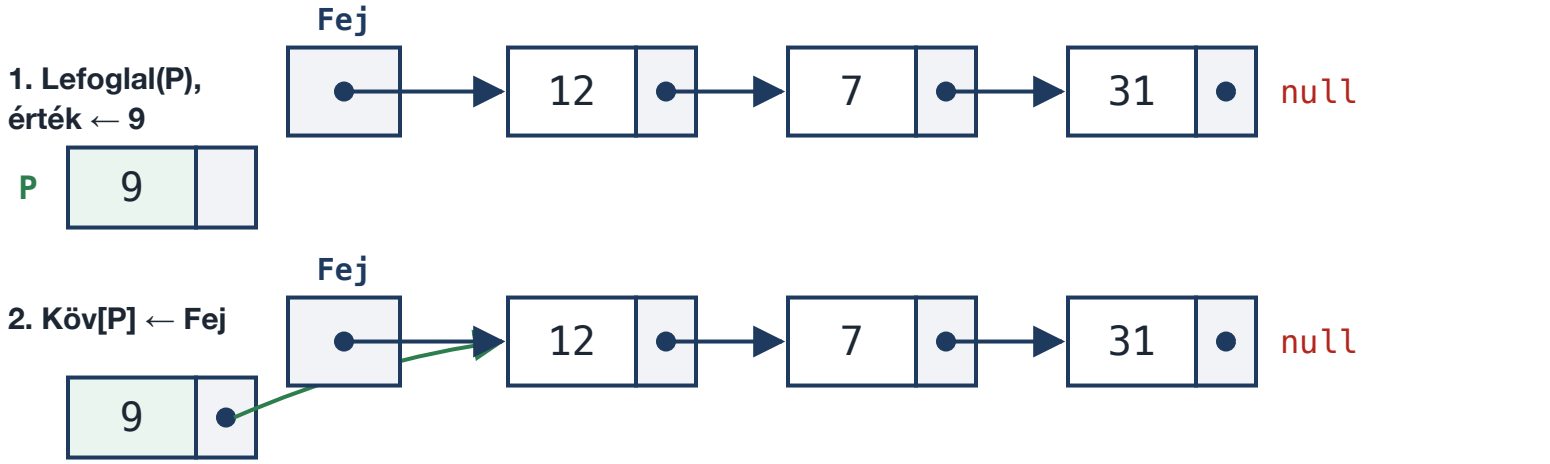
**Go**

```go
for p := l.Head; p != nil; p = p.Next {
    fmt.Println(p.Value)
}
```

Mindkettő **O(n)**. Rendezett listában sincs bináris keresés – nincs közvetlen elérés!

# Beszúrás a lista elejére

**Beszúrás a lista elejére (Adat = 9)**




**3. Fej ← P (a Fej mostantól az új elemre mutat)**

# Beszúrás az elejére és törlés az elejéről

**Beszúrás az elejére – O(1)**

```
Eljárás ElejéreBeszúr(Fej, Adat)
  Ha Lefoglal(P) akkor
    Elem[P].Érték ← Adat
    Elem[P].Köv ← Fej
    Fej ← P
  különben
    HIBA("nincs szabad hely")
  Elágazás vége
Eljárás vége
```

**Törlés az elejéről – O(1)**

```
Eljárás ElejérőlTöröl(Fej)
  Ha Fej ≠ Nil akkor
    P ← Fej
    Fej ← Elem[P].Köv
    Felszabadít(P)
  Elágazás vége
Eljárás vége
```

**Java**

```java
void insertFirst(int x) {
    head = new Node(x, head);
}

int removeFirst() {
    int x = head.value;
    head = head.next;   // a GC felszabadítja
    return x;
}
```

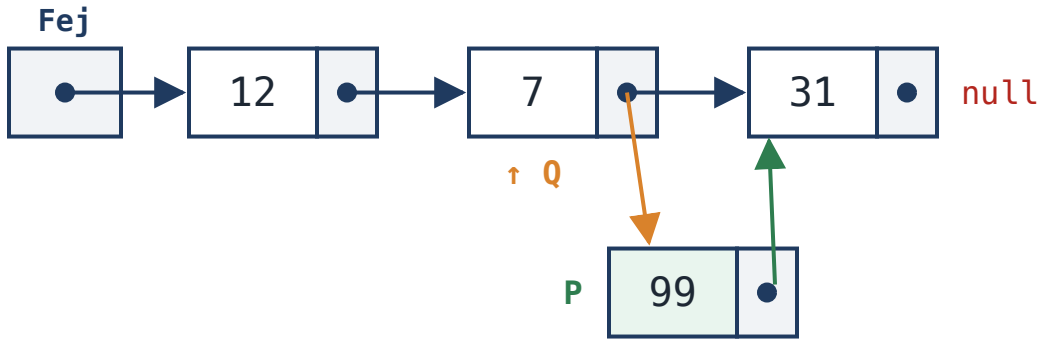
**Go**

```go
func (l *List) InsertFirst(x int) {
    l.Head = &Node{Value: x, Next: l.Head}
}

func (l *List) RemoveFirst() int {
    x := l.Head.Value
    l.Head = l.Head.Next
    return x
}
```

# Beszúrás egy adott elem mögé

**Beszúrás a Q által mutatott elem mögé**



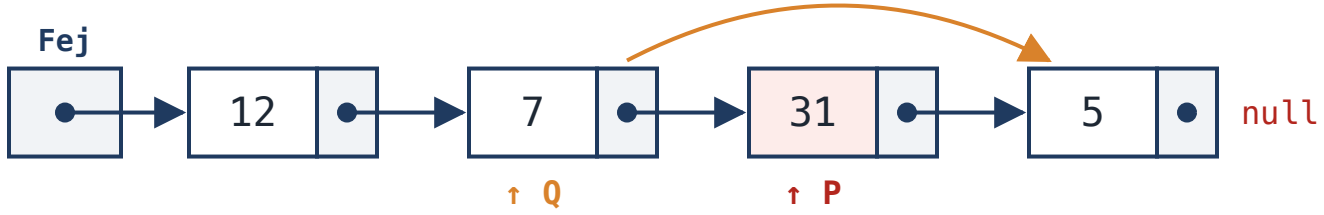

1. `Köv[P] ← Köv[Q]`
2. `Köv[Q] ← P`

**a sorrend fontos!**

Fordított sorrendben ( `Köv[Q] ← P` előbb) **elveszítenénk** a lista hátralévő részét – senki sem mutatna már rá!

# Törlés egy adott elem mögül

**Törlés a Q által mutatott elem mögül**




**P ← Köv[Q] ; Köv[Q] ← Köv[P] ; Felszabadít(P)**

```
Eljárás MögéBeszúr(Q, Adat)
  Ha Lefoglal(P) akkor
    Elem[P].Érték ← Adat
    Elem[P].Köv ← Elem[Q].Köv
    Elem[Q].Köv ← P
  Elágazás vége
Eljárás vége
```

```
Eljárás MögülTöröl(Q)
  P ← Elem[Q].Köv
  Ha P ≠ Nil akkor
    Elem[Q].Köv ← Elem[P].Köv
    Felszabadít(P)
  Elágazás vége
Eljárás vége
```

Mindkettő **O(1)** – ha Q már megvan. (Megkeresni O(n).)

# Műveletek a lista végén

**Beszúrás a végére – O(n)**

```
Eljárás VégéreBeszúr(Fej, Adat)
  Ha Lefoglal(P) akkor
    Elem[P].Érték ← Adat
    Elem[P].Köv ← Nil
    Ha Fej = Nil akkor
      Fej ← P
    különben
      Q ← Fej
      Ciklus amíg Elem[Q].Köv ≠ Nil
        Q ← Elem[Q].Köv
      Ciklus vége
      Elem[Q].Köv ← P
    Elágazás vége
  Elágazás vége
Eljárás vége
```

**Törlés a végéről – O(n)**

```
Eljárás VégérőlTöröl(Fej)
  Ha Fej = Nil akkor kilépés
  Ha Elem[Fej].Köv = Nil akkor
    Felszabadít(Fej) ; Fej ← Nil
  különben
    Q ← Fej       // az utolsó előttit keressük
    Ciklus amíg Elem[Elem[Q].Köv].Köv ≠ Nil
      Q ← Elem[Q].Köv
    Ciklus vége
    Felszabadít(Elem[Q].Köv)
    Elem[Q].Köv ← Nil
  Elágazás vége
Eljárás vége
```

Ha a **Vég** mutatót is nyilvántartjuk, a végére beszúrás O(1) lesz – a végéről törlés viszont nem (az utolsó előttit így is keresni kell). Megoldás: kétirányú lista (10. hét).

# Elem törlése érték szerint

Egyirányú listában csak a **mögötte** lévő elemet tudjuk törölni → a keresés közben az **előző** elemet is meg kell jegyezni.

```
boolean remove(int x) {
    Node prev = null, p = head;
    while (p != null && p.value != x) {
        prev = p;
        p = p.next;
    }
    if (p == null) return false;        // nincs ilyen elem
    if (prev == null) head = p.next;    // az első elemet töröljük
    else prev.next = p.next;
    return true;
}
```

Mindig gondoljunk a **speciális esetekre**: üres lista, az első elem, az utolsó elem, egyelemű lista!

# Láncolt lista vagy tömb?

| Művelet | Tömb | Dinamikus tömb | Láncolt lista |
|---|---|---|---|
| *i*-edik elem elérése | **O(1)** | **O(1)** | O(n) |
| Keresés | O(n) / O(log n) rendezett | O(n) / O(log n) | O(n) |
| Beszúrás / törlés az elején | O(n) | O(n) | **O(1)** |
| Beszúrás / törlés ismert elem mögé | O(n) | O(n) | **O(1)** |
| Beszúrás a végére | O(1) ha van hely | O(1) amortizált | O(n), Vég-mutatóval O(1) |
| Tárigény elemenként | érték | érték (+ tartalék) | érték + **mutató** |

**Mérés** (n = 160 000 elem beszúrása az elejére): `ArrayList` 870 ms · `LinkedList` 2 ms

**Középső elem olvasása** 20 000-szer: `ArrayList` 0,1 ms · `LinkedList` 357 ms

# Összefoglalás

- **Dinamikus adatszerkezet:** az elemek szétszórtan, mutatókkal összekötve; mérete futás közben változik
- **Memóriamodell:** rekordtömb + szabad lista; `Lefoglal(P)`, `Felszabadít(P)` – Java/Go-ban `new` és szemétgyűjtő
- **Láncolt lista:** Fej → elem → elem → … → Nil; **szekvenciális** elérés
- Elején és ismert elem mögött: beszúrás, törlés **O(1)** · keresés, végén: **O(n)**
- A mutatók átállításának **sorrendje** fontos · speciális esetek: üres lista, első elem

**Következő hét:** strázsás lista, kétirányú lista, ciklikus lista.

# Ellenőrző kérdések

1. Igaz vagy hamis: a dinamikus adatszerkezetek használata erőforrás-megosztást tesz lehetővé.
2. Igaz vagy hamis: a láncolt lista elemei közvetlenül (index alapján) érhetők el.
3. Igaz vagy hamis: a dinamikus adatszerkezetek helytelen használata memóriaszivárgást okozhat.
4. Mi történik, ha a mögé beszúrásnál előbb `Elem[Q].Köv ← P`, majd `Elem[P].Köv ← Elem[Q].Köv` hajtódik végre?
5. Írd meg pszeudokódban: a Fej mutatóval adott lista **elejére** szúrd be az `Adat` értéket! (`Lefoglal(P)` igazat ad, ha sikerült.)
6. Miért O(n) a törlés a lista végéről, még akkor is, ha ismerjük a Vég mutatót?

# Gyakorlat

# ✎ Bemutató programok

| Program | Java | Go | Mit mutat? |
|---|---|---|---|
| Memóriamodell | `MemoryModel.java` | `h09/memorymodel` | rekordtömb, szabad lista, Lefoglal/Felszabadít |
| Láncolt lista | `SinglyLinkedList.java` | `h09/singlylist` | minden alapművelet |
| Lista vs. tömb | `ListVsArray.java` | – | beszúrás az elejére, középső elem olvasása |

A `MemoryModel` program pontosan az előadás pszeudokódját valósítja meg – kövesd végig a kiírt táblázatokat!

# ✎ Feladatok

1. **Összeg és maximum:** számítsd ki egyetlen bejárással!
2. **i-edik elem:** add vissza a lista **i**-edik elemét! Mi a lépésszám?
3. **Megszámlálás:** hányszor fordul elő **x** a listában?
4. **Megfordítás helyben:** fordítsd meg a listát a mutatók átállításával, új elemek létrehozása nélkül!
5. **Összes x törlése:** töröld az összes **x** értékű elemet! (Figyelj a lista elején lévőkre!)
6. **Középső elem:** egyetlen bejárással – az egyik mutató egyet, a másik kettőt lép!
7. **Rendezett beszúrás:** szúrj be úgy, hogy a lista növekvő sorrendű maradjon!

Megoldások: `Exercises09.java`, `h09/exercises`

# ✎ Házi feladat

1. Egészítsd ki a listát egy **Vég** mutatóval! Melyik műveletek lesznek gyorsabbak, és melyikeket kell módosítani?
2. Fésülj össze két **rendezett** láncolt listát egy rendezett listává új elemek létrehozása nélkül, O(n + m) lépésben!
3. Döntsd el, hogy van-e **kör** egy listában (valamelyik elem `Köv` mezője egy korábbi elemre mutat)! (Tipp: nyúl és teknős.)
4. Valósítsd meg a vermet láncolt listával (push = beszúrás az elejére, pop = törlés az elejéről)! Miért jobb ez, mint a végén dolgozni?