# Adatszerkezetek

## 11. hét – Listára épülő verem és sor; a hash tábla

Dr. Koczka Ferenc · Eszterházy Károly Egyetem · 2026

# Ismétlés

- **Verem** (7. hét) és **sor** (8. hét) tömbbel: fix méret, „tele” állapot
- **Láncolt listák** (9–10. hét): az elején O(1), Vég mutatóval a végére is O(1)
- **Absztrakt adattípus:** a műveletek számítanak, nem a megvalósítás

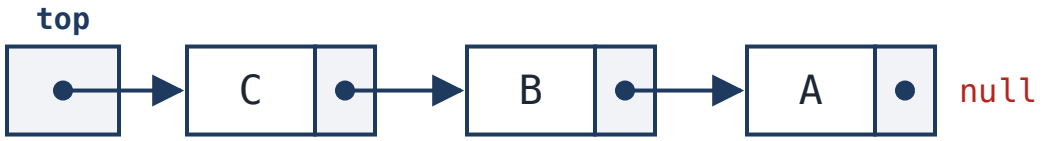
## A mai óra kérdései

- Hogyan valósítható meg a verem és a sor láncolt listával?
- Lehet-e egy elemet **O(1)** idő alatt megtalálni – rendezés és bináris keresés nélkül?

# Verem és sor láncolt listával

# Verem láncolt listával

**Verem listával: push és pop a lista ELEJÉN – mindkettő O(1)**




`push(x)` = beszúrás az elejére

`pop()` = törlés az elejéről

**nincs „tele” állapot**

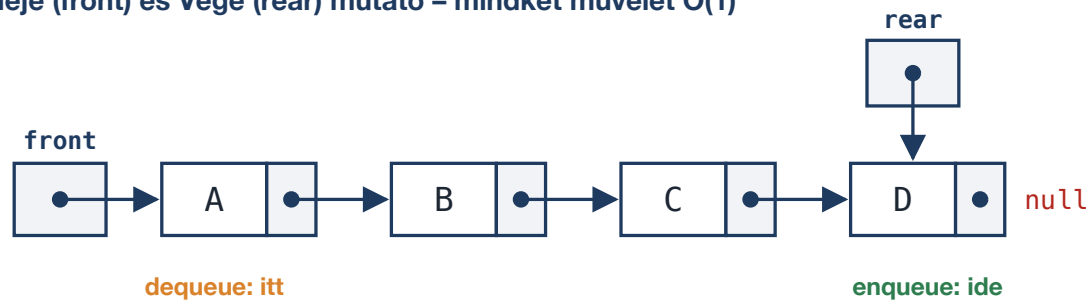
```
Eljárás Push(Tető, Adat)
  Ha Lefoglal(P) akkor
    Elem[P].Érték ← Adat
    Elem[P].Köv ← Tető
    Tető ← P
  különben HIBA
Eljárás vége
```

```
Függvény Pop(Tető)
  Ha Tető = Nil akkor HIBA
  x ← Elem[Tető].Érték
  P ← Tető
  Tető ← Elem[Tető].Köv
  Felszabadít(P)
  Vissza: x
Függvény vége
```

Ez pontosan a 9. heti „beszúrás az elejére” és „törlés az elejéről”.

# Sor láncolt listával

**Sor listával: Eleje (front) és Vége (rear) mutató – mindkét művelet O(1)**




```
Eljárás Sorba(Eleje, Vége, Adat)
  Ha Lefoglal(P) akkor
    Elem[P].Érték ← Adat
    Elem[P].Köv ← Nil
    Ha Vége = Nil akkor Eleje ← P
    különben Elem[Vége].Köv ← P
    Vége ← P
  különben HIBA
Eljárás vége
```

```
Függvény Sorból(Eleje, Vége)
  Ha Eleje = Nil akkor HIBA
  x ← Elem[Eleje].Érték
  P ← Eleje
  Eleje ← Elem[Eleje].Köv
  Ha Eleje = Nil akkor Vége ← Nil
  Felszabadít(P)
  Vissza: x
Függvény vége
```

Ha a sor kiürül, a **Vége** mutatót is Nil-re kell állítani!

# Egy interfész, két megvalósítás

**Java**

```java
interface Stack<T> {
    void push(T x);
    T pop();
    boolean isEmpty();
}

class LinkedStack<T> implements Stack<T> {
    private Node<T> top;
    public void push(T x) { top = new Node<>(x, top); }
    public T pop() {
        T x = top.value;
        top = top.next;
        return x;
    }
    public boolean isEmpty() { return top == null; }
}

// a felhasználó csak az interfészt ismeri:
String reverse(String s, Stack<Character> st)
```

**Go**

```go
type Stack[T any] interface {
    Push(x T)
    Pop() (T, error)
    IsEmpty() bool
}

type LinkedStack[T any] struct{ top *node[T] }

func (s *LinkedStack[T]) Push(x T) {
    s.top = &node[T]{x, s.top}
}

// bármelyik megvalósítás átadható:
reverse("数据结构", &LinkedStack[rune]{})
reverse("数据结构", &SliceStack[rune]{})
```

# Tömbös vagy láncolt megvalósítás?

| | Tömbbel | Láncolt listával |
|---|---|---|
| Push / Pop, Enqueue / Dequeue | O(1) | O(1) |
| Méret | fix – „tele” állapot lehetséges | **korlátlan** (amíg van memória) |
| Tárigény elemenként | csak az érték | érték + **mutató** |
| Kihasználatlan hely | a tömb üres része | nincs |
| Gyorsítótár-barát | **igen** (folytonos) | kevésbé (szétszórt) |
| Elemenkénti foglalás | nincs | minden push/enqueue foglal |

A gyakorlatban a **dinamikus (ciklikus) tömbös** megvalósítás a leggyakoribb (Java `ArrayDeque`, Go szelet) – gyors és memóriatakarékos.

# Hash tábla

# A probléma: gyors keresés

Eddigi lehetőségeink egy elem megtalálására:

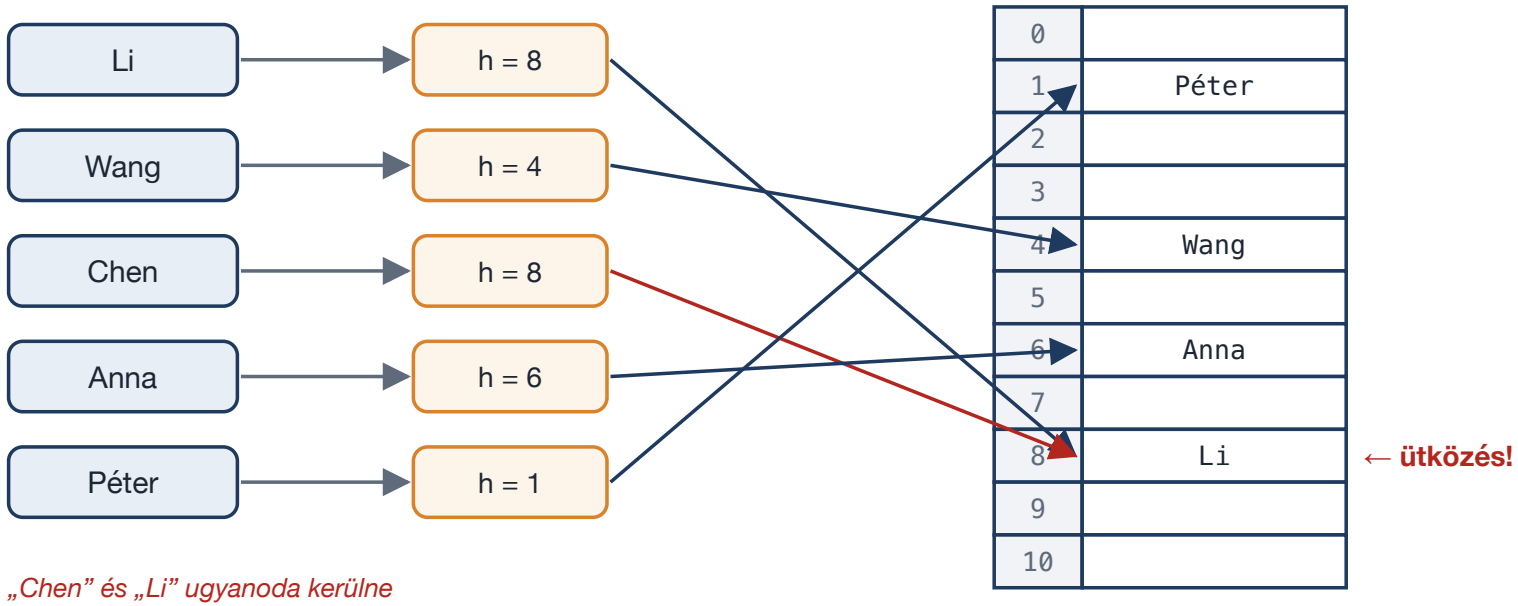
| Módszer | Keresés | Beszúrás |
|---|---|---|
| rendezetlen tömb / lista | O(n) | O(1) |
| rendezett tömb + bináris keresés | O(log n) | O(n) |
| **logikai vektor** (6. hét) | **O(1)** | **O(1)** |

A logikai vektor gyors, de csak **kicsi, ismert** alaphalmazra működik.

> **Ötlet:** alakítsuk át a kulcsot (bármilyen nagy szám vagy string) egy **kis indexszé** egy függvénnyel, és tároljuk az elemet egy tömb ezen indexén.

# Hash függvény

**Hash függvény: kulcs → index (0 … m – 1), itt m = 11**




*„Chen” és „Li” ugyanoda kerülne*

# Jó hash függvény

**Hash függvény:** h: kulcsok → {0, 1, …, m – 1}

Elvárások:

- **Gyorsan** számolható – O(kulcs hossza)
- **Egyenletesen** szórja a kulcsokat a rekeszek között
- **Determinisztikus:** ugyanarra a kulcsra mindig ugyanazt adja

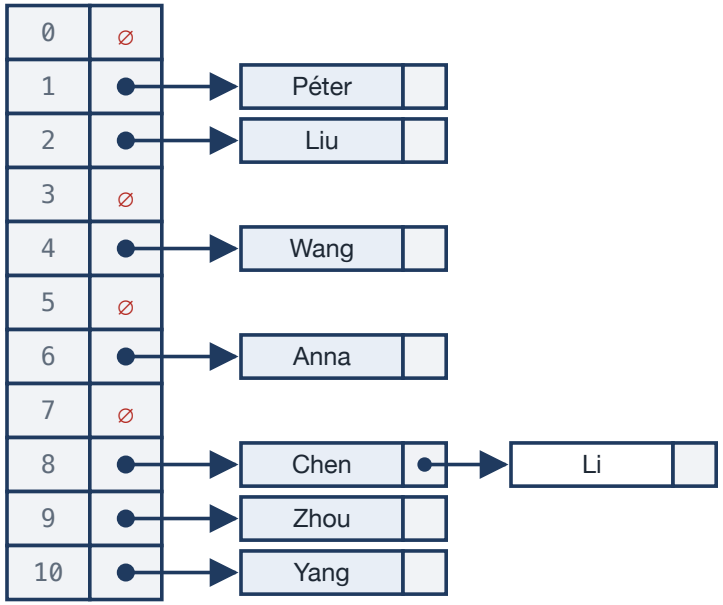
**Példák:**

- Egész kulcs: **h(k) = k mod m** (m legyen prímszám)
- String: **polinomiális hash**: h = (h · 31 + c) mod m minden c karakterre
  Java: `"Li".hashCode()` = 'L' · 31 + 'i' = 76 · 31 + 105 = **2461**

> **Ütközés:** két különböző kulcsnak ugyanaz a hash értéke. Elkerülhetetlen (több kulcs van, mint rekesz) – **kezelni kell**!

# Ütközéskezelés 1.: láncolás

**Láncolás: minden rekesz egy láncolt lista**




beszúrás: a h(k) rekesz listájának elejére

keresés: csak a h(k) listában

törlés: törlés a h(k) listából

**telítettség: α = n / m = 8 / 11 ≈ 0,73**

átlagos listahossz = α

# Láncolás – műveletek

**Pszeudokód**

```
Eljárás Beszúr(T, kulcs, érték)
  i ← h(kulcs)
  P ← Keres(T[i], kulcs)       // a listában
  Ha P ≠ Nil akkor
    Elem[P].Érték ← érték      // módosítás
  különben
    ElejéreBeszúr(T[i], kulcs, érték)
  Elágazás vége
Eljárás vége

Függvény Keres(T, kulcs)
  Vissza: ListábanKeres(T[h(kulcs)], kulcs)
Függvény vége
```

**Java**

```java
void put(String key, int value) {
    int i = hash(key);
    for (Entry e = buckets[i]; e != null; e = e.next)
        if (e.key.equals(key)) { e.value = value; return; }
    buckets[i] = new Entry(key, value, buckets[i]);
    size++;
    if (size > 0.75 * buckets.length) resize();
}

Integer get(String key) {
    for (Entry e = buckets[hash(key)];
         e != null; e = e.next)
        if (e.key.equals(key)) return e.value;
    return null;
}
```

# Telítettség és újrahashelés

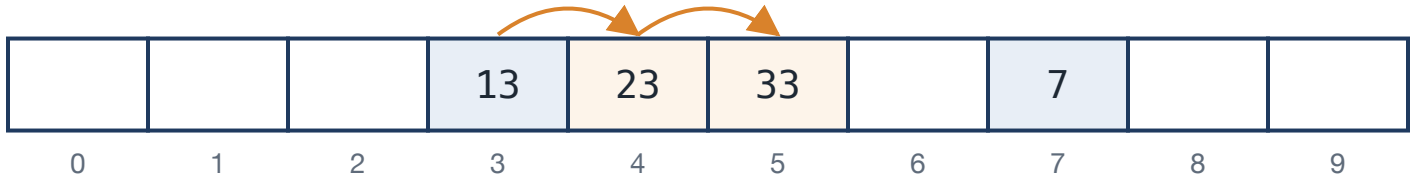
**Telítettség (load factor):** α = n / m (elemek száma / rekeszek száma)

- Láncolásnál az átlagos listahossz **α** → keresés átlagosan **O(1 + α)**
- Ha α túl nagy (pl. > 0,75): **újrahashelés** – kétszer akkora tábla, és **minden** elemet újra elhelyezünk (a hash függ **m**-től!)
- Mint a dinamikus tömbnél: egy átméretezés O(n), de **amortizáltan O(1)**

**Példa** (`HashTable` program): m = 11, 8 elem (α = 0,73) – egy ütközés (Li, Chen)
A 9. elem után α = 0,82 > 0,75 → m = 22, α = 0,41 – nincs ütközés

# Ütközéskezelés 2.: nyílt címzés

**Nyílt címzés, lineáris próbálás: h(k) = k mod 10; ütközéskor a következő szabad hely**




`13, 23, 33 mind a 3-as helyre képződik → 3, 4, 5`

keresés: h(k)-tól lépkedünk, amíg meg nem találjuk vagy üres helyre nem érünk

- Minden elem **magában a tömbben** van – nincs lista
- Ütközéskor a **következő** szabad helyet keressük (lineáris próbálás)
- **Törlés:** nem írhatunk egyszerűen üreset – megszakítaná a keresési láncot → „törölt” jelölés kell

# Nyílt címzés: a telítettség hatása

Átlagos próbálásszám sikeres keresésnél (mért értékek, `LinearProbing` program):

| α | 0,25 | 0,50 | 0,75 | 0,90 | 0,99 |
|---|---|---|---|---|---|
| próbálás | 1,17 | 1,50 | 2,48 | 5,52 | **36,16** |

- Kis telítettségnél szinte mindig **elsőre** megtaláljuk
- Telítődéskor a foglalt helyek **csomókba** tömörülnek → drasztikusan lassul
- Nyílt címzésnél $\alpha < 1$ kötelező, a gyakorlatban $\alpha \leq 0{,}5$–$0{,}75$

# A hash tábla hatékonysága

| Művelet | Átlagos eset | Legrosszabb eset |
|---|---|---|
| Keresés | **O(1)** | O(n) |
| Beszúrás | **O(1)** (amortizált) | O(n) |
| Törlés | **O(1)** | O(n) |

- A legrosszabb eset: **minden** kulcs ugyanabba a rekeszbe kerül (rossz hash függvény, vagy rosszindulatú bemenet)
- **Nincs sorrend:** a minimum, a maximum, a rendezett bejárás O(n) / O(n log n)
- Rendezett kulcsokhoz **keresőfa** kell (15. hét, Java `TreeMap`)

# Beépített hash táblák

**Java – HashMap, HashSet**

```java
Map<String, Integer> freq = new HashMap<>();
for (String w : text.split(" "))
    freq.merge(w, 1, Integer::sum); // freq[w]++

freq.get("fox");                // 2
freq.getOrDefault("cat", 0);    // 0
freq.containsKey("dog");        // true
freq.remove("the");

Set<String> seen = new HashSet<>();
```

A kulcs osztályának `hashCode()` és `equals()` metódusa **összhangban** kell legyen!

**Go – map**

```go
freq := map[string]int{}
for _, w := range strings.Fields(text) {
    freq[w]++          // hiányzó kulcs: 0
}

v, ok := freq["cat"]   // "comma ok"
delete(freq, "the")
len(freq)

seen := map[string]bool{}
```

A map bejárási sorrendje **véletlenszerű** – szándékosan!

**Alkalmazások:** szótárak, gyakoriság, gyorsítótár, adatbázis-indexek, ismétlődések kiszűrése, fordítóprogramok szimbólumtáblája

# Összefoglalás

- **Verem listával:** push/pop a lista elején · **sor listával:** Eleje + Vége mutató – mind O(1), nincs „tele”
- Ugyanaz az **interfész** többféle megvalósítással
- **Hash tábla:** a kulcsból hash függvénnyel index → átlagosan **O(1)** keresés, beszúrás, törlés
- **Ütközéskezelés:** láncolás (rekeszenként lista) vagy nyílt címzés (következő szabad hely)
- **Telítettség** α = n/m; túl nagy α → újrahashelés
- Java `HashMap` / `HashSet` · Go `map`

**Következő hét:** rekurzió.

# Ellenőrző kérdések

1. Miért a lista **elején** dolgozik a láncolt verem, és nem a végén?
2. Mi történik a láncolt sorban, ha az utolsó elemet vesszük ki, és elfelejtjük a Vége mutatót Nil-re állítani?
3. h(k) = k mod 7, láncolással. Szúrd be: 10, 3, 17, 24, 5! Rajzold le a táblát!
4. Ugyanezek a kulcsok lineáris próbálással, m = 7 esetén – hová kerül a 24?
5. Igaz vagy hamis: a hash tábla keresési ideje a legrosszabb esetben is O(1).
6. Miért nem lehet hash táblában hatékonyan megkeresni a legkisebb kulcsot?

# Gyakorlat

# ✎ Bemutató programok

| Program | Java | Go | Mit mutat? |
|---|---|---|---|
| Listás verem és sor | `LinkedStackQueue.java` | `h11/linkedstackqueue` | interfész, két megvalósítás |
| Hash tábla | `HashTable.java` | `h11/hashtable` | láncolás, ütközés, újrahashelés |
| Lineáris próbálás | `LinearProbing.java` | `h11/linearprobing` | nyílt címzés, telítettség hatása |
| Szógyakoriság | `WordCount.java` | `h11/wordcount` | `HashMap` / `map` használata |

# ✎ Feladatok

1. **Két szám összege:** van-e a tömbben két elem, amelynek összege adott? $O(n^2)$ helyett $O(n)$ hash táblával!
2. **Első egyedi karakter:** melyik az első karakter, amely csak egyszer fordul elő? (`"swiss"` → `'w'`)
3. **Anagramma-csoportok:** csoportosítsd a szavakat úgy, hogy az egymás anagrammái egy csoportba kerüljenek!
4. **Ismétlődés:** van-e ismétlődő elem a tömbben? (O(n) átlagosan)
5. **Zárójelek:** a 7. heti feladat, a zárójelpárokat map-ben tárolva.
6. **Telefonkönyv:** név → telefonszám, keresés név szerint.

Megoldások: `Exercises11.java`, `h11/exercises`

# ✎ Házi feladat

1. Egészítsd ki a saját `HashTable` osztályt egy `keys()` metódussal, amely visszaadja az összes kulcsot! Mi a lépésszáma?
2. Valósítsd meg a lineáris próbálásos táblában a **törlést** „törölt” jelöléssel! Hogyan módosul a keresés és a beszúrás?
3. Mérd meg a saját hash táblád ütközéseinek számát 10 000 véletlen szóra két hash függvénnyel: (a) a karakterkódok összege, (b) polinomiális hash! Mit tapasztalsz?
4. Valósíts meg egy **LRU gyorsítótárat** hash táblával és kétirányú listával (lásd a 10. heti házi feladatot), ahol a `get` és a `put` is O(1)!