Adatszerkezetek

7. hét – A verem (stack): LIFO elv, megvalósítás, alkalmazások

Dr. Koczka Ferenc · Eszterházy Károly Egyetem · 2026

Ismétlés

- **Tömb:** közvetlen elérés, a végére beszúrás és onnan törlés **O(1)**
- **Absztrakt adattípus (ADT):** **mit** tud, nem **hogyan** (1. hét)
- **Halmaz:** különböző ábrázolások, különböző hatékonyság

A mai óra kérdései

- Mi az a verem, és miért olyan fontos?
- Hogyan valósítható meg tömbbel?
- Hogyan számol a számítógép egy `(3 + 4) * 2` kifejezéssel?

A verem fogalma

A **verem** (stack) olyan adatszerkezet, amelyben csak a **legutoljára betett** elemhez férünk hozzá.

LIFO – **Last In, First Out**: ami utoljára ment be, az jön ki először.

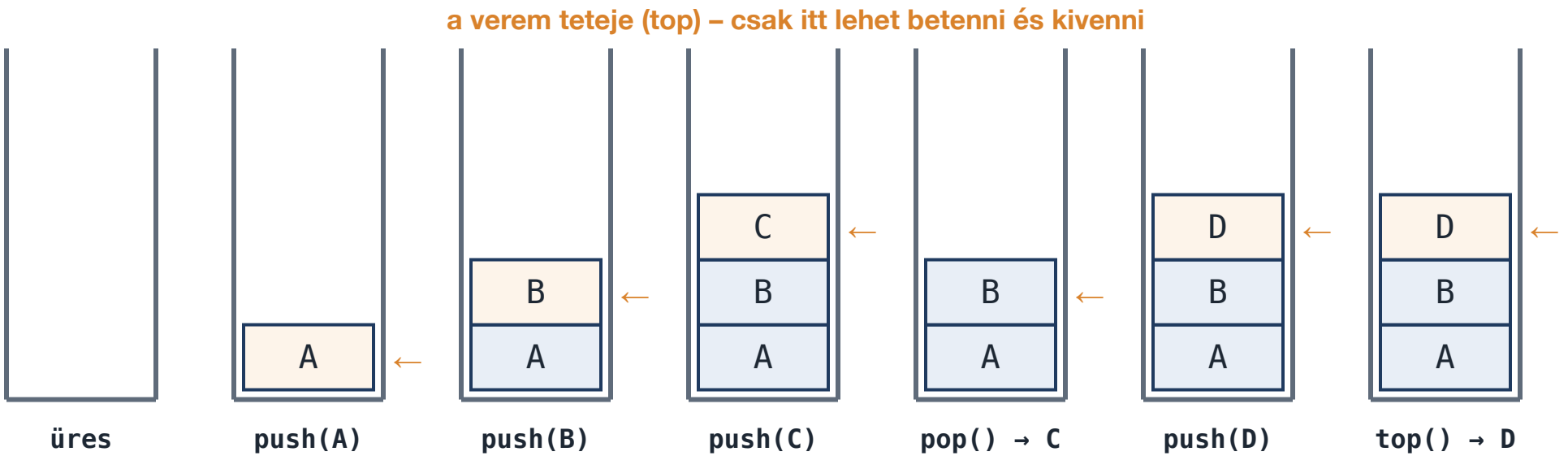
Hétköznapi példák

- Egymásra rakott tányérok
- Pringles chipses doboz
- Böngésző „Vissza” gombja
- Szövegszerkesztő „Visszavonás” (Ctrl+Z)

Tulajdonságai

Elemek	homogén
Kapcsolat	lineáris
Elérés	**korlátozott** – csak a tető
Méret	statikus vagy dinamikus

A verem műveletei



A verem műveletei – összefoglaló

Művelet	Jelentés	Lépésszám
Létrehoz	üres verem	O(1)
Push (verembe)	elem a tetőre	O(1)
Pop (veremből)	a tető kivétele és visszaadása	O(1)
Top / Peek (tető)	a tető lekérdezése kivétel nélkül	O(1)
Üres? · **Tele?**	állapot lekérdezése	O(1)

Hibás műveletek

- **Pop vagy Top üres veremből** → **alulcsordulás** (stack underflow)
- **Push teli verembe** → **túlcsordulás** (stack overflow) – csak fix méretű veremnél

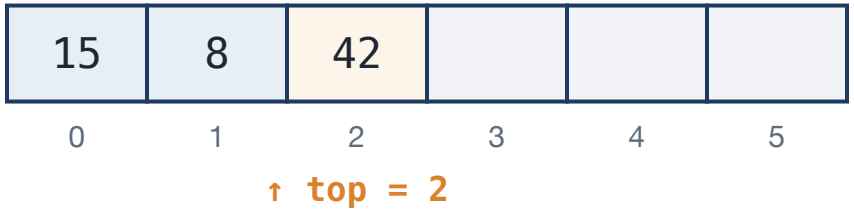
A műveletek előtt mindig ellenőrizni kell az **Üres?** / **Tele?** állapotot – vagy a megvalósítás jelezze a hibát (Java: kivétel, Go: `error` visszatérési érték).

A **Stack Overflow** név innen ered: túl mély rekurzió esetén a program **hívási verme** megtelik (lásd később és a 12. héten).

Megvalósítás tömbbel

Verem tömbbel

Verem tömbbel: a tömb eleje a verem alja, a teteje a top indexnél




```
push(x): top ← top + 1; T[top] ← x
pop(): x ← T[top]; top ← top − 1
üres: top = −1 tele: top = n − 1
```

mindkettő O(1)

- A verem **alja** a tömb eleje, a **teteje** a `top` index
- Push és Pop a tömb **végén** dolgozik → nem kell elemeket mozgatni → **O(1)**

Verem tömbbel – pszeudokód

```
// T[1..n] tömb, tető: az utolsó elem indexe

Eljárás Létrehoz
  tető ← 0
Eljárás vége

Függvény Üres
  Vissza: tető = 0
Függvény vége

Függvény Tele
  Vissza: tető = n
Függvény vége
```

```
Eljárás Push(x)
  Ha Tele akkor HIBA("túlcsordulás")
  különben
    tető ← tető + 1
    T[tető] ← x
  Elágazás vége
Eljárás vége

Függvény Pop
  Ha Üres akkor HIBA("alulcsordulás")
  különben
    x ← T[tető]
    tető ← tető − 1
    Vissza: x
  Elágazás vége
Függvény vége
```

A pszeudokódban 1-től indexelünk, ezért üres veremnél tető = 0; Java/Go-ban 0-tól, ott üresnél top = −1.

Verem tömbbel – Java és Go

Java

```java
class ArrayStack {
    private final int[] data;
    private int top = -1;

    ArrayStack(int capacity) {
        data = new int[capacity];
    }
    boolean isEmpty() { return top == -1; }
    boolean isFull()  { return top == data.length - 1; }

    void push(int x) {
        if (isFull())
            throw new IllegalStateException("tele");
        data[++top] = x;
    }
    int pop() {
        if (isEmpty())
            throw new IllegalStateException("üres");
        return data[top--];
    }
}
```

Go – generikus, szelettel

```go
type Stack[T any] struct {
    items []T
}

func (s *Stack[T]) Push(x T) {
    s.items = append(s.items, x)
}

func (s *Stack[T]) Pop() T {
    n := len(s.items) - 1
    x := s.items[n]
    s.items = s.items[:n]
    return x
}

func (s *Stack[T]) IsEmpty() bool {
    return len(s.items) == 0
}
```

A szelet dinamikus tömb → nincs „tele” állapot.

Beépített verem

Java

```java
Deque<Integer> st = new ArrayDeque<>();
st.push(1);
st.push(2);
st.peek();     // 2
st.pop();      // 2
st.isEmpty();  // false
```

A régi `java.util.Stack` osztály helyett az **`ArrayDeque`** ajánlott (gyorsabb).

Go

```go
st := []int{}
st = append(st, 1)        // push
st = append(st, 2)
top := st[len(st)-1]      // peek
st = st[:len(st)-1]       // pop
empty := len(st) == 0
```

Go-ban nincs külön verem típus – a **szelet** tökéletesen megfelel.

Alkalmazás: kifejezések kiértékelése

Infix, prefix, postfix

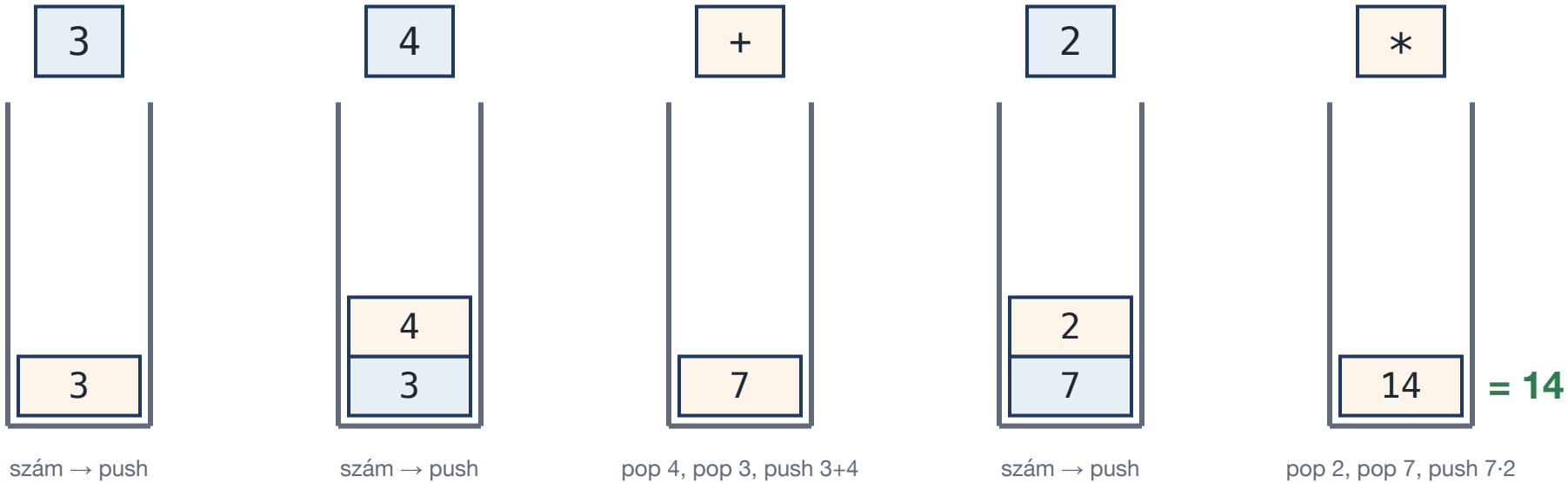
Jelölés	Példa	Operátor helye
Infix	`(3 + 4) * 2`	az operandusok között – zárójel kell
Prefix (lengyel)	`* + 3 4 2`	az operandusok előtt
Postfix (fordított lengyel)	`3 4 + 2 *`	az operandusok után

A postfix előnyei:

- **Nincs szükség zárójelekre** és precedenciaszabályokra
- Balról jobbra, **egyetlen bejárással**, egy veremmel kiértékelhető
- Fordítóprogramok, a Java virtuális gép és HP számológépek is így dolgoznak

Postfix kiértékelése veremmel

Postfix kifejezés: 3 4 + 2 ∗ (= (3 + 4) · 2)



Postfix kiértékelése – algoritmus

Pszeudokód

```
Függvény PostfixÉrték(kifejezés)
  Létrehoz
  Ciklus minden t tokenre balról jobbra
    Ha t szám akkor
      Push(t)
    különben                  // t operátor
      b ← Pop                 // jobb operandus!
      a ← Pop                 // bal operandus
      Push(a t b)
    Elágazás vége
  Ciklus vége
  Vissza: Pop
Függvény vége
```

Előbb a **jobb** oldali operandus jön ki! `5 2 -` → b = 2, a = 5 → 5 – 2 = 3

Java

```java
ArrayDeque<Double> st = new ArrayDeque<>();
for (String t : expr.split("\\s+")) {
    if (isOperator(t)) {
        double b = st.pop();
        double a = st.pop();
        st.push(apply(a, b, t));
    } else {
        st.push(Double.parseDouble(t));
    }
}
return st.pop();
```

Lépésszám: O(n) – minden tokent egyszer dolgozunk fel

Hibás kifejezés: Pop üres veremből, vagy a végén nem pontosan egy elem marad

Példa: `1 2 + 3 4 * - 9 /`

Token	Művelet	Verem (alulról)
1	push	1
2	push	1 2
+	1 + 2	3
3	push	3 3
4	push	3 3 4
*	3 · 4	3 12
–	3 – 12	–9
9	push	–9 9
/	–9 / 9	**–1**

Infix alakban: ((1 + 2) – 3 · 4) / 9 = –1

Infixből postfix: tolatópályaudvar-algoritmus

E. W. Dijkstra (1961), egyszerűsített változat – az **operátorokat** tesszük verembe:

1. **Operandus** → azonnal a kimenetre
2. `(` → a verembe
3. `)` → a veremből a kimenetre mindent az első `(` -ig; a `(` -t eldobjuk
4. **Operátor** → amíg a verem tetején **legalább ilyen erős** operátor van, azt a kimenetre tesszük; utána az új operátort a verembe
5. A végén a verem maradékát a kimenetre

Infix	Postfix
`3 + 4 * 2`	`3 4 2 * +`
`(3 + 4) * 2`	`3 4 + 2 *`
`(a + b) * (c - d)`	`a b + c d - *`

Alkalmazás: zárójelek ellenőrzése

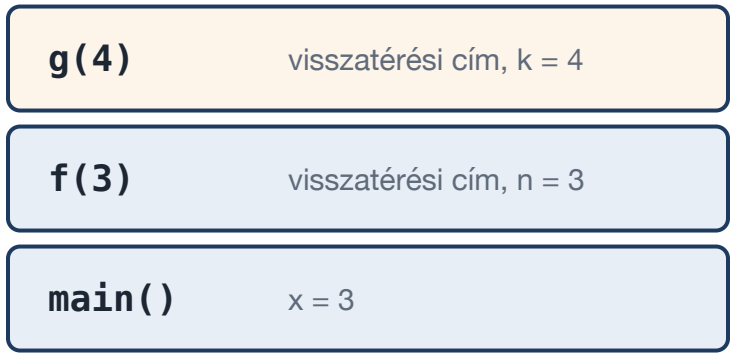
Helyes-e a zárójelezés? `{[()()]}` ✓ · `([)]` ✗ · `((a + b)` ✗

```
Függvény Helyes(S)
  Létrehoz
  Ciklus minden c karakterre
    Ha c nyitó zárójel akkor Push(c)
    különben ha c csukó zárójel akkor
      Ha Üres akkor Vissza: hamis          // nincs nyitó párja
      Ha Pop nem c párja akkor Vissza: hamis
    Elágazás vége
  Ciklus vége
  Vissza: Üres                             // maradt bezáratlan?
Függvény vége
```

Alkalmazás: fordítóprogramok, szövegszerkesztők, HTML/XML ellenőrzése · **O(n)**

Alkalmazás: a hívási verem

Hívási verem (call stack)



← a futó függvény kerete (top)

függvényhívás → push (új keret)

return → pop (vissza a hívóhoz)

túl mély rekurzió → StackOverflowError

Minden függvényhívás **keretet** (helyi változók, paraméterek, visszatérési cím) tesz a verembe – ezért működik a **rekurzió** (12. hét).

További alkalmazások

- **Visszavonás (undo):** minden módosítás előtti állapot a verembe kerül
- **Böngésző előzmények:** „Vissza” = pop, új oldal = push
- **Visszalépéses keresés (backtracking):** útvesztő, sudoku, 8 királynő – az eddigi döntések a veremben
- **Mélységi bejárás** fákban, gráfokban
- **Számrendszer-átváltás:** a maradékok fordított sorrendben → verem
- **Szövegmegfordítás:** minden karakter be, aztán mind ki

Összefoglalás

- **Verem:** LIFO – csak a tetőhöz férünk hozzá
- Műveletek: **Push, Pop, Top, Üres?, Tele?** – mind **O(1)**
- **Tömbös megvalósítás:** `top` index a tömb végén; fix méretnél túlcsordulás lehetséges
- Java: `ArrayDeque` · Go: szelet (`append` , `s[:len(s)-1]`)
- **Postfix kiértékelés:** egy bejárás, egy verem, O(n) – a jobb operandus jön ki előbb!
- Alkalmazások: zárójelek, undo, hívási verem, visszalépéses keresés

Következő hét: a sor (queue) – FIFO elv, ciklikus sor, kétvégű sor, prioritási sor.

Ellenőrző kérdések

1. Mit ír ki? `push(1); push(2); pop(); push(3); push(4); pop(); pop(); pop();` (a pop kiírja az elemet)
2. Értékeld ki veremmel, és írd fel a verem állapotát minden lépés után: `6 2 / 3 1 - *`
3. Alakítsd postfix alakra: `a * (b + c) - d / e`
4. Mit ír ki a következő kód, ha `n = 50`? Általában mit csinál?

   ```
   while (n > 0) { s.push(n % 2); n = n / 2; } while (!s.isEmpty())
   print(s.pop());
   ```
5. Igaz vagy hamis: tömbös megvalósításnál a Pop lépésszáma O(n), mert az elemeket el kell tolni.

Gyakorlat

✎ Bemutató programok

Program	Java	Go	Mit mutat?
Tömbös verem	`ArrayStack.java`	`h07/arraystack`	push, pop, túl-/alulcsordulás, generikus verem
Postfix	`Postfix.java`	`h07/postfix`	kiértékelés a verem állapotának kiírásával
Infix → postfix	`InfixToPostfix.java`	`h07/infixtopostfix`	tolatópályaudvar-algoritmus
Zárójelek	`Brackets.java`	`h07/brackets`	`()`, `[]`, `{}` ellenőrzése

✎ Feladatok

1. **Kettes számrendszer:** írd át egy szám tízes alakját kettesre veremmel!
2. **Megfordítás:** fordíts meg egy szöveget veremmel! (Go-ban kínai szövegre is!)
3. **Minimum-verem:** valósíts meg egy vermet, amelynek `min()` művelete is O(1)!
4. **Undo:** szimuláld egy szövegszerkesztő visszavonás funkcióját!
5. **Kódkövetés:** betű = push, `*` = pop és kiírás. Mit ír ki? `D A T * A * S * T R * U C * T`
6. **Szomszédos párok:** töröld ismételten a szomszédos azonos betűpárokat! (`"abbaca"` → `"ca"`)

Megoldások: `Exercises07.java`, `h07/exercises`

✎ Házi feladat

1. Egészítsd ki a postfix kiértékelőt a `^` (hatványozás) és a `%` (maradék) operátorral!
2. Bővítsd az infix → postfix átalakítót **többjegyű számokra** (pl. `12 + 345 * 6`)!
3. Írj programot, amely két vermet használva **sort** valósít meg (FIFO)! Mennyi az egyes műveletek lépésszáma?
4. Írj programot, amely egy HTML-szövegben ellenőrzi, hogy a címkék (`<b>…</b>`, `<i>…</i>`) helyesen vannak-e egymásba ágyazva!