Adatszerkezetek

8. hét – A sor (queue): FIFO elv, ciklikus sor, kétvégű sor, prioritási sor

Dr. Koczka Ferenc · Eszterházy Károly Egyetem · 2026

Ismétlés

- **Verem:** LIFO – csak a tetőhöz férünk hozzá
- Push, Pop, Top – mind **O(1)**, tömbbel a tömb végén
- Alkalmazások: postfix kiértékelés, zárójelek, undo, hívási verem

A mai óra kérdései

- Mi a különbség a verem és a sor között?
- Miért nem jó a „naiv” tömbös sor, és hogyan javítja a **ciklikus** sor?
- Mi a kétvégű sor és a prioritási sor?

A sor fogalma

A **sor** (queue) olyan adatszerkezet, amelybe a **végén** teszünk be elemet, és az **elején** veszünk ki.

FIFO – **First In, First Out**: ami először ment be, az jön ki először.

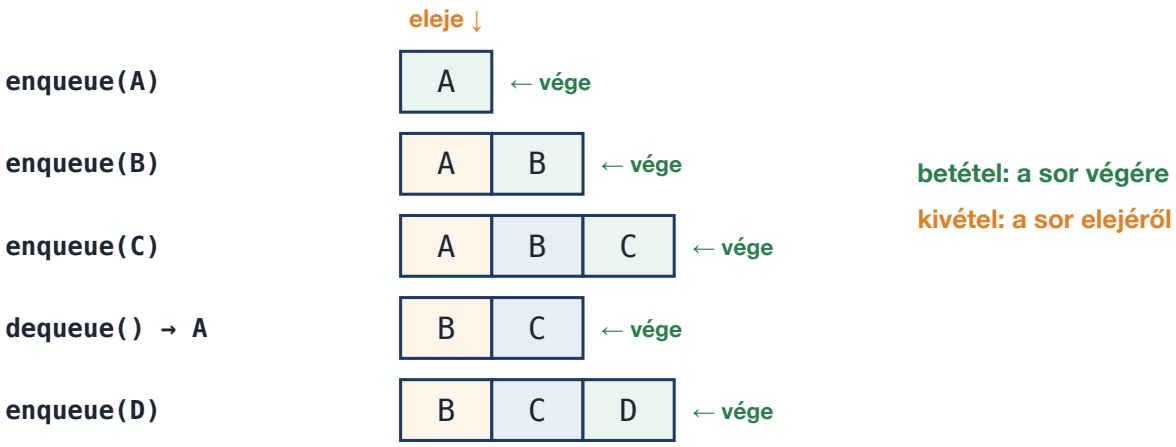
Hétköznapi példák

- Sorban állás a menzán, a pénztárnál
- Nyomtatási sor
- Ügyfélszolgálati telefonvonal
- Billentyűzetpuffer

Tulajdonságai

Elemek	homogén
Kapcsolat	lineáris
Elérés	**korlátozott** – eleje és vége
Méret	statikus vagy dinamikus

A sor műveletei



betétel: a sor végére

kivétel: a sor elejéről

Művelet	Jelentés	Lépésszám
Enqueue (sorba)	elem a sor **végére**	O(1)
Dequeue (sorból)	az **első** elem kivétele	O(1)
Peek / Front	az első elem lekérdezése	O(1)
Üres? · Tele?	állapot	O(1)

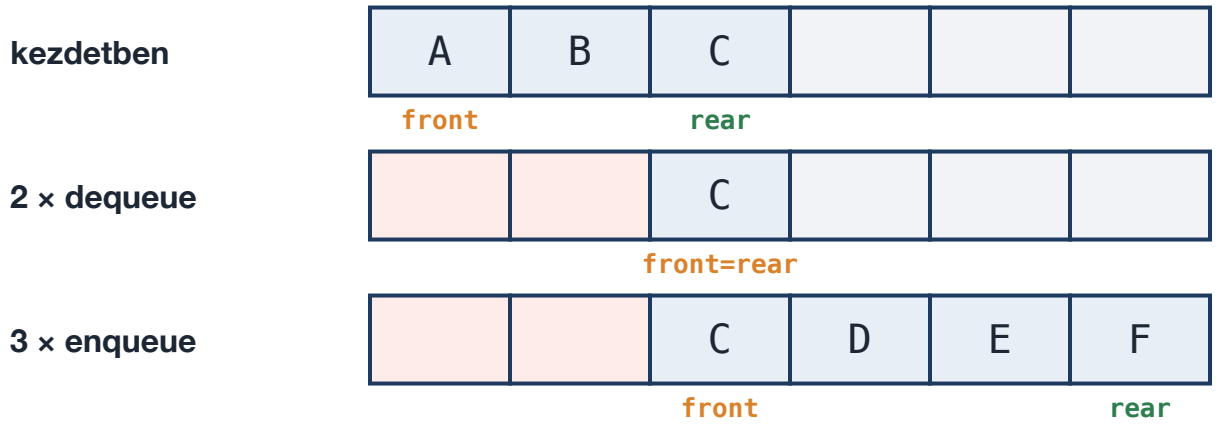
Verem és sor összehasonlítása

	Verem	Sor
Elv	**LIFO**	**FIFO**
Betétel	a tetőre (push)	a végére (enqueue)
Kivétel	a tetőről (pop)	az elejéről (dequeue)
Hozzáférés	egy vég	két vég
Mutatók tömbös megvalósításban	`top`	`front`, `rear`
Tipikus alkalmazás	undo, rekurzió, kifejezések	ütemezés, pufferelés, szélességi bejárás

Sor tömbbel

Első próbálkozás: naiv tömbös sor

`front` az első, `rear` az utolsó elem indexe; enqueue: `rear ← rear + 1`, dequeue: `front ← front + 1`



A sor „vándorol” jobbra:

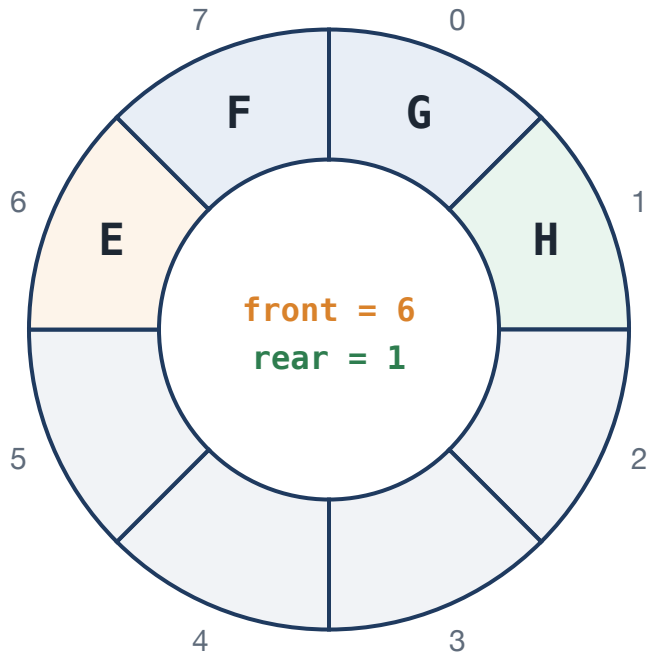
rear a tömb végén → „tele”,

pedig elöl 2 hely üres!

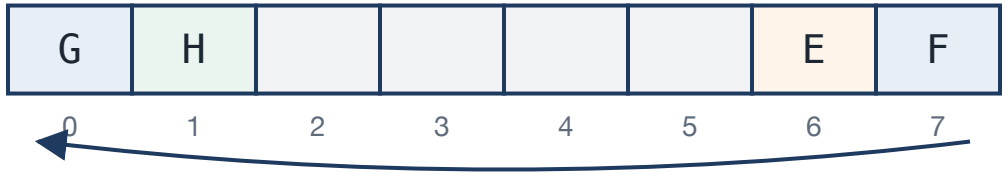
A naiv sor javítási lehetőségei

1. **Eltolás:** minden dequeue után az elemeket egy hellyel előre toljuk
   - A front mindig 0 marad, nem vész el hely
   - De a dequeue **O(n)** lesz ✗
2. **Ciklikus sor:** a tömböt **körként** kezeljük – ha a `rear` a tömb végére ér, az **elején** folytatja
   - Nincs elemmozgatás, minden művelet **O(1)** ✓
   - Az index „körbefordítása”: **(i + 1) mod n**

Ciklikus sor



Ugyanez tömbként:




```
enqueue: rear ← (rear + 1) mod n
dequeue: front ← (front + 1) mod n
üres: darab = 0
tele: darab = n
```

minden művelet O(1), nincs elemmozgatás

Üres vagy tele? A ciklikus sor csapdája

Ha csak `front` és `rear` van, **üres** és **tele** sor esetén ugyanaz a viszonyuk: `front = (rear + 1) mod n`!

Megoldások:

1. **Darabszám** (`db`) nyilvántartása – üres: `db = 0`, tele: `db = n` ← **ezt használjuk**
2. **Egy hely mindig üresen marad** – tele: `(rear + 1) mod n = front`, de csak **n** – 1 elem fér el
3. Egy logikai jelző (`üres`), amit a műveletek állítanak

> A darabszámos megoldás a legegyszerűbb és legkevésbé hibaérzékeny.

Ciklikus sor – pszeudokód

```
// T[0..n-1] tömb (itt kivételesen 0-tól,
// hogy a mod n egyszerű legyen)

Eljárás Létrehoz
  eleje ← 0 ; vége ← n − 1 ; db ← 0
Eljárás vége

Függvény Üres
  Vissza: db = 0
Függvény vége

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

```
Eljárás Sorba(x)
  Ha Tele akkor HIBA("a sor tele")
  vége ← (vége + 1) mod n
  T[vége] ← x
  db ← db + 1
Eljárás vége

Függvény Sorból
  Ha Üres akkor HIBA("a sor üres")
  x ← T[eleje]
  eleje ← (eleje + 1) mod n
  db ← db − 1
  Vissza: x
Függvény vége
```

Ciklikus sor – Java és Go

Java

```java
class CircularQueue {
    private final int[] data;
    private int front = 0, rear = -1, count = 0;

    void enqueue(int x) {
        if (count == data.length)
            throw new IllegalStateException("tele");
        rear = (rear + 1) % data.length;
        data[rear] = x;
        count++;
    }

    int dequeue() {
        if (count == 0)
            throw new IllegalStateException("üres");
        int x = data[front];
        front = (front + 1) % data.length;
        count--;
        return x;
    }
}
```

Go

```go
type CircularQueue struct {
    data               []int
    front, rear, count int
}

func (q *CircularQueue) Enqueue(x int) error {
    if q.count == len(q.data) {
        return errors.New("tele")
    }
    q.rear = (q.rear + 1) % len(q.data)
    q.data[q.rear] = x
    q.count++
    return nil
}

func (q *CircularQueue) Dequeue() (int, error) {
    if q.count == 0 {
        return 0, errors.New("üres")
    }
    x := q.data[q.front]
    q.front = (q.front + 1) % len(q.data)
    q.count--
    return x, nil
}
```

Beépített sorok

Java

```
Queue<String> q = new ArrayDeque<>();
q.offer("Li");      // enqueue
q.offer("Wang");
q.peek();           // "Li"
q.poll();           // "Li" – dequeue
q.isEmpty();
```

Az `ArrayDeque` belül **ciklikus, dinamikusan bővülő** tömb.

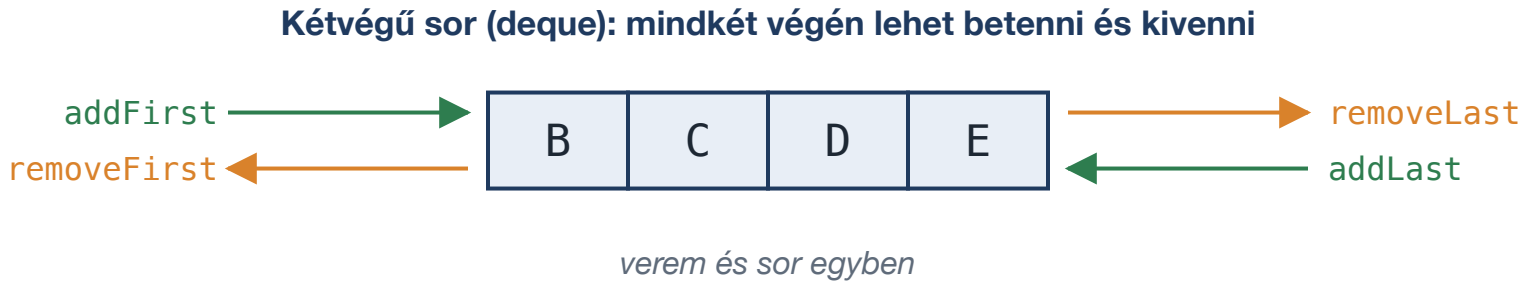
Go

```
q := []string{}
q = append(q, "Li")   // enqueue
q = append(q, "Wang")
first := q[0]         // peek
q = q[1:]             // dequeue
```

Egyszerű, de a `q[1:]` nem szabadítja fel az elejét – sok művelethez saját ciklikus sor vagy `container/list` jobb.

A sor változatai

Kétvégű sor (deque)



- **Deque** (**double-ended queue**, „dek”): mindkét végén **betehetünk** és **kivehetünk**
- Csak az egyik végét használva **verem**, ellentétes végeket használva **sor**
- Ciklikus tömbbel: `addFirst` esetén `front ← (front − 1 + n) mod n`

Java-ban és Go-ban a `−1 % n` értéke **−1**, nem **n** − 1! Ezért kell a `+ n`.

Prioritási sor

Minden elemhez **prioritás** tartozik; mindig a **legfontosabb** (pl. legkisebb prioritási számú) elemet vesszük ki – nem a legrégebbit.

Példák: kórházi sürgősségi ellátás, operációs rendszer folyamatütemezése, útvonaltervezés (Dijkstra)

Megvalósítás	Betétel	Kivétel (min)
rendezetlen tömb	**O(1)** – a végére	O(n) – minimumkeresés
rendezett tömb	O(n) – a helyére tolni	**O(1)** – a végéről
kupac (15. hét)	**O(log n)**	**O(log n)**

Beépített: Java `PriorityQueue` · Go `container/heap`

Alkalmazások

- **Pufferelés:** billentyűzet, hálózati csomagok, videó streamelés – a termelő és a fogyasztó eltérő sebességének kiegyenlítése
- **Ütemezés:** nyomtatási sor, operációs rendszer **round-robin** ütemezője
- **Szimuláció:** pénztárak, forgalom, kiszolgálási rendszerek
- **Szélességi bejárás** (BFS): fák szintenkénti bejárása, legrövidebb út gráfokban
- **Üzenetsorok** elosztott rendszerekben (pl. a webáruház rendeléseinek feldolgozása)

Round-robin: minden folyamat legfeljebb egy időszeletig fut, utána a sor végére áll – így mindenki sorra kerül.

Összefoglalás

- **Sor:** FIFO – a végére teszünk be, az elejéről veszünk ki
- Műveletek: **Enqueue, Dequeue, Peek, Üres?, Tele?** – mind **O(1)**
- **Naiv tömbös sor:** a sor „vándorol”, elöl elvész a hely
- **Ciklikus sor:** `(i + 1) mod n` – körbefordul, nincs mozgatás; üres/tele: **darabszám**
- **Deque:** mindkét végén lehet betenni és kivenni
- **Prioritási sor:** a legfontosabb elem jön ki először – kupaccal O(log n)

Következő hét: dinamikus adatszerkezetek – a dinamikus memóriakezelés modellje és az egyszerű láncolt lista.

Ellenőrző kérdések

1. Mit ír ki? `enqueue(1); enqueue(2); dequeue(); enqueue(3); enqueue(4); dequeue(); dequeue(); dequeue();`
2. Ciklikus sor, n = 5, kezdetben üres (eleje = 0, vége = 4). Végrehajtjuk: 4 × sorba, 3 × sorból, 3 × sorba. Mennyi most `eleje`, `vége` és `db`?
3. Mit ír ki a következő kód n = 10 esetén? Általában mit csinál?
   `q.enqueue(0); q.enqueue(1); for (i = 0; i < n; i++) { a = q.dequeue(); print(a); b = q.peek(); q.enqueue(a + b); }`
4. Igaz vagy hamis: a ciklikus sorban a dequeue O(n), mert az elemeket előre kell tolni.
5. Melyik adatszerkezetet használnád: a) böngésző „Vissza” gombja, b) nyomtatási feladatok, c) kórházi sürgősségi felvétel?

Gyakorlat

✎ Bemutató programok

Program	Java	Go	Mit mutat?
Naiv sor	`NaiveQueue.java`	–	a „vándorló” sor problémája, eltolásos javítás
Ciklikus sor	`CircularQueue.java`	`h08/circularqueue`	körbefordulás, üres/tele darabszámmal
Deque	`CircularDeque.java`	`h08/deque`	mindkét végén, palindrom-ellenőrzés
Prioritási sor	`PriorityQueueDemo.java`	`h08/priorityqueue`	rendezetlen tömbbel és kupaccal

✎ Feladatok

1. **Kódkövetés:** futtasd a 3. ellenőrző kérdés kódját, és magyarázd el a kimenetet!
2. **Josephus-probléma:** **n** ember áll körben, minden **k**-adik kiesik. Ki marad utoljára? (Sorral: k – 1 embert hátraküldünk, a **k**-adik kiesik.)
3. **Round-robin:** szimuláld az ütemezést! Folyamatok: P1 = 5, P2 = 2, P3 = 4 időegység, időszelet = 2. Mikor végez az egyes folyamat?
4. **Sor megfordítása** egy verem segítségével.
5. **Kódkövetés:** betű = enqueue, `*` = dequeue és kiírás: `D A T * A * S * T R * U C * T` (hasonlítsd össze a veremmel!)
6. **Mozgó átlag:** írd ki minden új adat után az utolsó **k** adat átlagát O(1) lépésben!

Megoldások: `Exercises08.java`, `h08/exercises`

✎ Házi feladat

1. Valósítsd meg a ciklikus sort **darabszám nélkül**, egy üresen hagyott hellyel! Hány elem fér el egy **n** méretű tömbben?
2. Bővítsd a ciklikus sort úgy, hogy tele állapotban **kétszer akkora** tömbbe másolja az elemeket (dinamikus ciklikus sor)! Figyelj a körbefordult elemek sorrendjére!
3. Valósíts meg **sort két verem** segítségével (lásd a 7. heti házi feladatot)! Hasonlítsd össze a ciklikus sorral!
4. **Pénztárszimuláció:** véletlenszerűen érkeznek vásárlók, egy pénztáros percenként egy vásárlót szolgál ki. Mennyi az átlagos várakozási idő különböző érkezési gyakoriságok esetén?