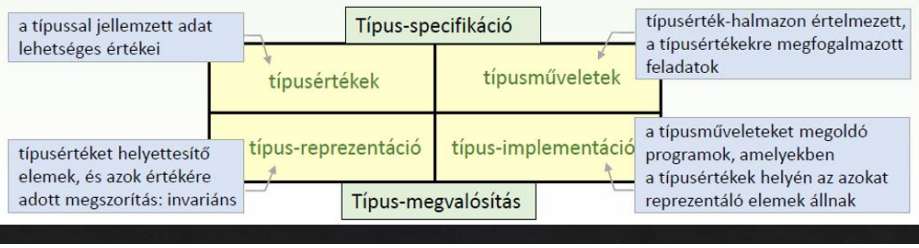
OEP 3. gyakorlat

## Adattípus fogalma (1. előadás)

- Típus-specifikáció
  - Típusértékek
  - Típusműveletek
- Típus megvalósítás
  - Típus-reprezentáció
  - Típus-implementáció

### Adattípus fogalma

- Egy adat (változó) típusának definiálásához szükség van a típus specifikációjára és annak megvalósítására.
- A típus-specifikáció megadja:
  - az adat által felvehető értékek halmazát
  - a típusértékekkel végezhető műveleteket
- A típus-megvalósítás megmutatja:
  - hogyan ábrázoljuk (reprezentáljuk) a típus értékeit
  - milyen programok helyettesítsék (implementálják) a műveleteket




**Asszociatív tömb**

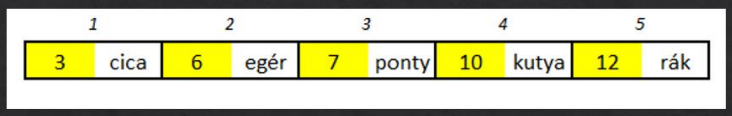
- Ez egy olyan kulcs-adat párokat tároló gyűjtemény, amelyben kulcs alapján lehet visszakeresni az értékeket.
- A kulcs típusa egész lesz, az adat típusa pedig szöveg.
- A tárolóban az elemeket kulcsuk alapján lehet megkeresni, elérni, így fontos, hogy a kulcs egyedi legyen.
- A típust Map-nek fogjuk nevezni.
- A tervezésnél fontos szempont, hogy a visszakeresés gyors legyen, akár a beszúrás, törlés rovására!

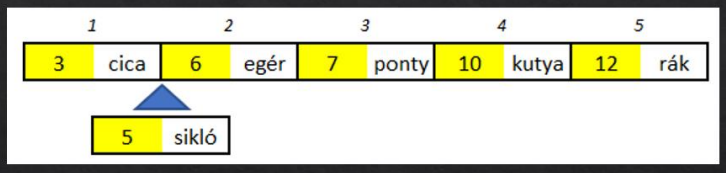
*Típusdefiníció:*

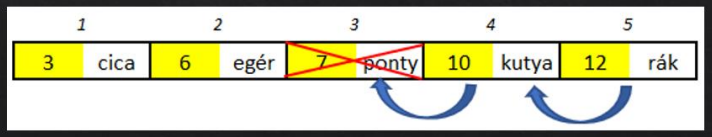
| **Map**<br><br>Ez asszociatív tömbök (azaz speciális tárolók) halmaza. Egy tároló elemei $\mathbb{Z} \times \mathbb{S}$ (kulcs-adat) típusú párok, ahol egy adat a kulcsa alapján egyértelműen beazonosítható | **map := SetEmpty(map)** map : Map<br>*//kiüríti az asszociatív tömböt*<br>**c := Count(map)** map : Map, c : $\mathbb{N}$<br>*//megadja az elemek számát*<br>**map := Insert(map,e)** map : Map, e : $\mathbb{Z} \times \mathbb{S}$<br>*//új elemet tesz be, ha a kulcsa még nem létezik*<br>**map := Erase(map, key)** map : Map, key : $\mathbb{Z}$<br>*// törli az adott kulcsú elemet, ha a kulcs létezik, különben hiba*<br>**l := In(map, key)** map : Map, key : $\mathbb{Z}$, l : $\mathbb{L}$<br>*//lekérdezi, van-e adott kulcsú elem*<br>**data := map[key]** map : Map, key : $\mathbb{Z}$, data : $\mathbb{S}$<br>*// lekérdezi az adott kulcsú elem adatát, ha a kulcs létezik, különben hiba* |
|---|---|

## Kulcs szerint rendezett tárolás

- Kulcs szerint szigorúan monoton növekedően tároljuk az asszociatív tömb elmeit. A sorozatban kulcs szerint szigorúan monoton növekvő sorrendben tároljuk az elemeket.
- Ekkor a kulcsok kereséséhez használhatjuk a logaritmikus keresést.
- Ha tömbben tárolnánk az elemeket, akkor beszúrásnál „helyet kell csinálni” az új elemnek: hátrébb kell csúsztatni az új elemnél nagyobb kulcsúakat.
- törlésnél pedig a keletkezett „lyukat” el kell tüntetni: előrébb kell csúsztatni a törlés pozíciója utáni elemeket.



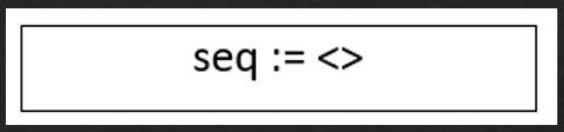




*(Megjegyzés: található hatékonyabb tárolás (bináris keresőfák, hash tábla), lásd majd Algoritmusok tárgyban.)*

# Map reprezentáció

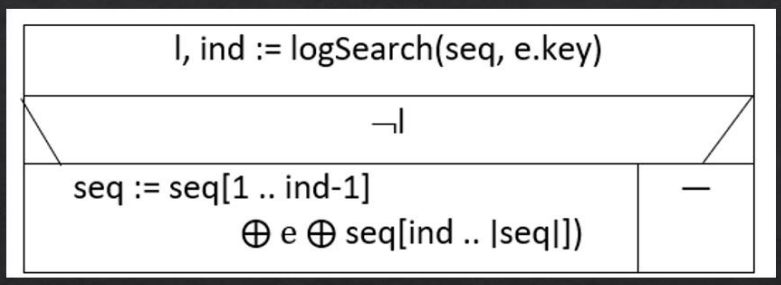
- Típus reprezentációja:
  - **Item** = rec(key: $\mathbb{Z}$, data: $\mathbb{S}$)
  - **seq: Item*** – az elemeket kulcsuk szerint rendezetten tároló sorozat
- Típus műveletek implementációja:
  - **map:=setEmpty(map)** **map:Map** *//kiüríti az asszociatív tömböt*



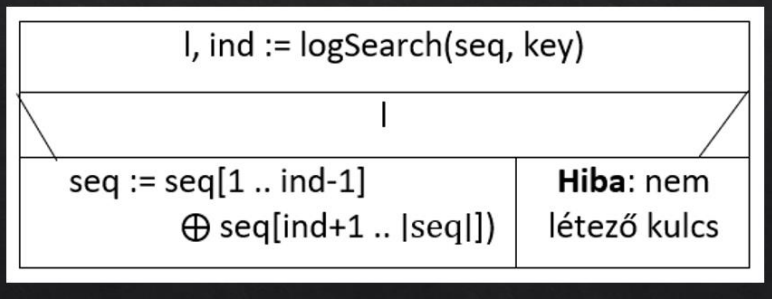

  - **c := count(map)** **map:Map, c:** $\mathbb{N}$ *//megadja az elemek számát*


# Map reprezentáció

- **map := insert(map,e)** map:Map, e: Item *//új elemet tesz be, ha a kulcsa még nem létezik*




- **map := erase(map, key)** map:Map, key: $\mathbb{Z}$
  *//törli az adott kulcsú elemet, ha a kulcs létezik, különben hiba*



## Map reprezentáció

- **l := in(map, key)** map:Map, key: $\mathbb{Z}$, l : $\mathbb{L}$ *//lekérdezi, van-e adott kulcsú elem*

| l, ind := logSearch(seq, key) | ind : $\mathbb{N}$ |
|---|---|

- **data := map[key]** map:Map, key: $\mathbb{Z}$ , data: $\mathbb{S}$
  *// lekérdezi az adott kulcsú elem adatát, ha a kulcs létezik, különben hiba*

| l, ind := logSearch(seq, key) | |
|---|---|
| l | |
| data := seq[ind].data | **Hiba**: nem létező kulcs |

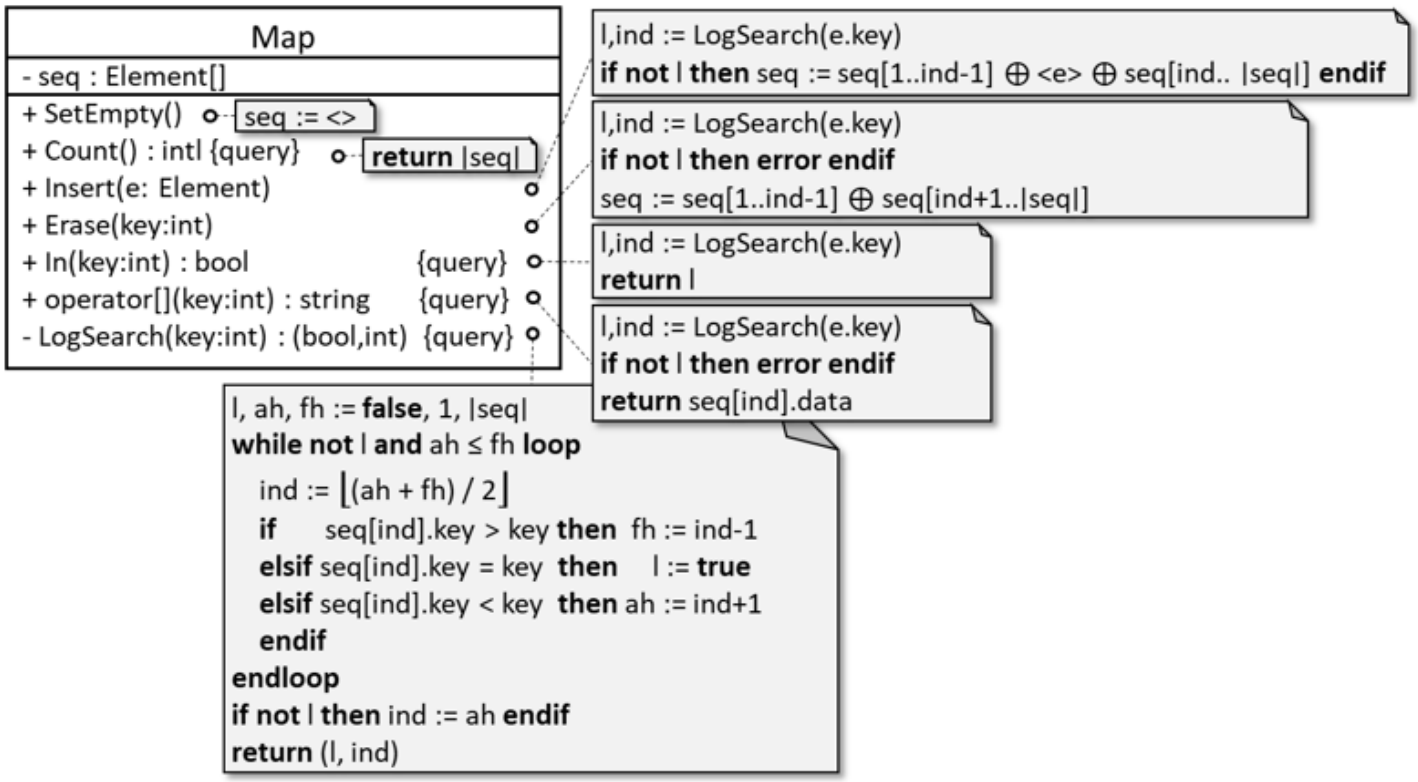
Kiegészítjük a szokásos algoritmust: ha nincs a sorozatban keresett kulcsú elem, akkor az első olyan elem indexét adjuk vissza, amelynek kulcsa nagyobb a keresett kulcsnál; ha nem lenne ilyen, akkor a sorozat hossza plusz egyet.

Ezzel ügyesebb lesz az insert művelet: nem kulcs-összehasonlításra támaszkodik az eltolás ciklusa, hiszen tudjuk, hogy mettől meddig kell a tömb elemeit jobbra tolni. Ha nincs a keresett kulcsnál nagyobb a tömbben, akkor az utolsó elem utáni sorszámot kapjuk vissza.

**l, ind := logSearch (seq, key)**

| l, ah, fh := hamis, 1, \|seq\| | | | ah, fh : $\mathbb{N}$ |
|---|---|---|---|
| $\neg l \wedge ah \leq fh$ | | | |
| ind := ⌊(ah + fh) / 2⌋ | | | |
| seq[ind].key > key | seq[ind].key = key | seq[ind].key < key | |
| fh := ind-1 | l := igaz | ah := ind+1 | |
| $\neg l$ | | | |
| ind := ah | — | | |

*Osztály diagram:*




## Prioritásos sor

◈Készítsünk maximum prioritásos sort.

◈Az elemek két mezőből állnak (prioritás (egész), adat (szöveg)).

◈A sorból mindig a legnagyobb prioritású elemet vesszük ki (több legnagyobb esetén nem meghatározott, hogy melyiket).

| **PrQueue**<br><br>a maximum prioritásos sorok halmaza, amely soroknak az elemei $\mathbb{Z} \times \mathbb{S}$ típusú párok. | **SetEmpty(pq)** pq : PrQueue<br>*// Kiüríti a pr sort*<br>**l := IsEmpty(pq)** pq : PrQueue, l : $\mathbb{L}$<br>*//Igazat ad, ha üres a pr sor, hamisat ha nem.*<br>**pq := Add(pq, e)** pq : PrQueue, e : $\mathbb{Z} \times \mathbb{S}$<br>*//Betesz egy új elemet a pr sorba.*<br>**e := GetMax(pq)** pq : PrQueue, e : $\mathbb{Z} \times \mathbb{S}$<br>*//Visszadja az egyik legnagyobb prioritású elemet, nem veszi ki. Fontos, hogy a sor nem lehet üres.*<br>**pq, e := RemMax(pq)** pq : PrQueue, e : $\mathbb{Z} \times \mathbb{S}$<br>*//Kiveszi az egyik legnagyobb prioritású elemet. Fontos, hogy a sor nem lehet üres.* |
|---|---|

A reprezentációnál két lehetőség között választhatunk. Mindkettőhöz szükségünk van egy sorozatra, de a műveletek futási ideje eltérő lehet.

(1) **Rendezetlen** (n hosszú) **sorozat**:

a. **SetEmpty** : üres sorozatot készít. $\Theta(1)$ (Habár nem tudjuk, hogy a vector clear() metódusa mit is csinál pontosan.)
b. **IsEmpty:** hosszból azonnal eldönthető. $\Theta(1)$
c. **Add:** az új elemet a sorozat végéhez fűzzük. $\Theta(1)$
d. **GetMax:** ha nem üres s sor, a tanult maximum kiválasztás algoritmussal megkeressük az egyik legnagyobb prioritású elemet, és visszaadjuk. A sorozat nem változik. $\Theta(n)$
e. **RemMax:** ha nem üres a sorozat, a tanult maximum kiválasztás algoritmussal megkeressük az egyik legnagyobb prioritású elemet, és kivesszük. $\Theta(n)$

(2) **Rendezett** (n hosszú) **sorozat.** Elemek prioritás szerint növekvő a sorrendben vannak.

a. **SetEmpty** : üres sorozatot készít. $\Theta(1)$ (Habár nem tudjuk, hogy a vector clear() metódusa mit is csinál pontosan.)
b. **IsEmpty:** hosszból azonnal eldönthető $\Theta(1)$
c. **Add:** az új elemet betesszük a rendezettség szerinti helyére, amelyet lineáris kereséssel vagy kiválasztással kell megkeresnünk $\Theta$ (n)
d. **GetMax:** ha nem üres a sorozat, akkor a rendezettség miatt a sorozat utolsó eleme az egyik legnagyobb prioritású elem. A sorozat nem változik. $\Theta(1)$
e. **RemMax:** ha nem üres a sorozat, akkor a rendezettség miatt a sorozat utolsó eleme az egyik legnagyobb prioritású elem. Azt ki is kell vennünk a sorozatból. $\Theta(1)$

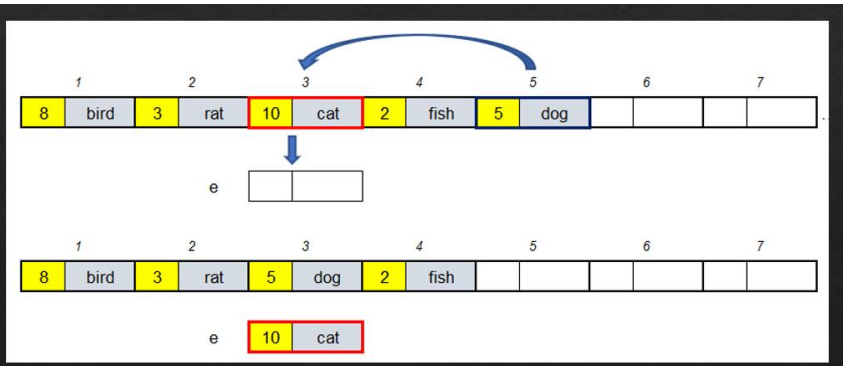
| n hosszú sorozat | **rendezetlen** | **rendezett** |
|---|---|---|
| **SetEmpty()** | Θ(1) | Θ(1) |
| **IsEmpty()** | Θ(1) | Θ(1) |
| **Add()** | Θ(1) * | Θ(n) * |
| **GetMax()** | Θ(n) | Θ(1) |
| **RemMax()** | Θ(n) ** | Θ(1) ** |

(*) + sorozat végéhez új elem hozzáírása (*) log n + sorozat közepére betesszünk elemet

(**) nem kell léptetni a sorozat elemeit (**) + sorozat végéről elveszünk elemet

- Azt, hogy melyik a jobb, annak függvényében lehet eldönteni, hogy melyik művelet lesz gyakoribb, és azt megvalósítani hatékonyan.
- (Megjegyzés: Tovább növelhető a hatékonyság kupac adatszerkezettel, erről majd az Algoritmusok és adatszerkezetek tárgyban lesz szó.)

- Pr. sorból legnagyobb prioritású elem kiszedése Terv:
  - Maximum kiválasztással meghatározzuk a legnagyobb prioritású elemet, kivesszük egy változóba,
  - A helyén keletkező „lyukba” a vector utolsó elemét bemásoljuk,
  - A vector utolsó elemét eldobjuk.
  - Hibát kell jelezni, ha üres a prioritásos sor!



seq: Element*

- a prioritásos sor elemeit rendezetlenül tároló sorozat, ahol Element = rec(pr : $\mathbb{Z}$, data : $\mathbb{S}$)

**SetEmpty(pq)**)

| seq := <> |
|---|

(C++ vector: clear())

**l := IsEmpty(pq)**)

| l := \|seq\| =0 |
|---|

(C++ vector: size())

**pq := Add(pq, e)**

| seq := seq ⊕ <e> |
|---|

(C++ vector: push_back())

**e := GetMax(pq)**

| \|seq\|>0 | |
|---|---|
| max, ind := $MAX_{i=1..[seq]}$(seq[i].pr) | **Hiba:** üres prioritásos sor |
| e := seq[ind] | |

**pq, e := RemMax(pq)**

| \|seq\|>0 | |
|---|---|
| max, ind := $MAX_{i=1..[seq]}$(seq[i].pr) | **Hiba** üres prioritásos sor |
| e := seq[ind] | |
| seq[ind] := seq[\|seq\|] | |
| pop_back(seq) | |

*Osztály diagram:*