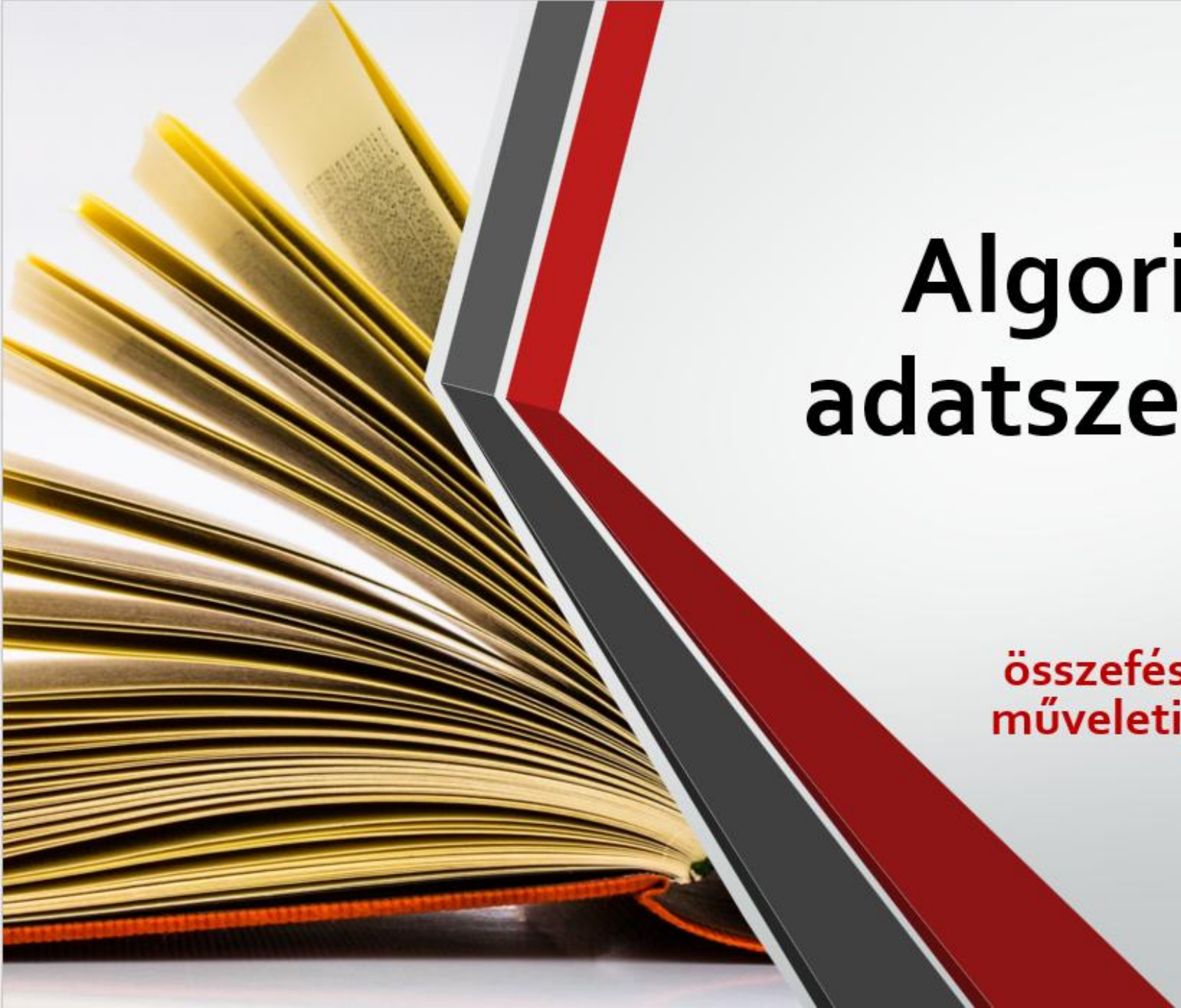
Algoritmusok és adatszerkezetek I.

Előadás

Bemutató

Tartalom



Algoritmusok és adatszerkezetek I.
1. Előadás

Beszúró rendezés, összefésülő rendezés (tömbös), műveletigény, tárigény fogalma

Tartalom

Ajánlott irodalom

- A tárgy hivatalos honlapja:
  - http://aszt.inf.elte.hu/~asvanyi/ad/
- Saját honlapom:
  - https://people.inf.elte.hu/kinga/algoritmusok1/seged_ea.htm
  - https://people.inf.elte.hu/kinga/algoritmusok1/algasz2.htm
- Fekete István jegyzetei:
  - http://people.inf.elte.hu/fekete/algoritmusok_jegyzet/

Néhány alapvető jelölés

- $\mathbb{B}$ = {false=0; true=1} a logikai (boolean) értékek halmaza.
- $\mathbb{N}$ = {0; 1; 2; 3; . . . } a természetes számok halmaza.
- $\mathbb{N}_+$ = {1; 2; 3; . . . } a pozitív természetes számok halmaza.
- $\mathbb{Z}$ = {· · · − 3; −2, −1; 0; 1; 2; 3; . . . } az egész számok halmaza.
  - i, j, k, l, m, n, I, J, K, M, N : $\mathbb{Z}$
- $\mathbb{R}$ a valós számok halmaza.
- $\mathbb{P}$ a pozitív valós számok halmaza.
- $\mathbb{P}_0$ a nemnegatív valós számok halmaza.

Néhány alapvető jelölés

- $log\ n\ = \begin{cases} log_2 n\ ha\ n\ > 0 \\ 0 \qquad ha\ n\ =\ 0 \end{cases}$
- *fele*(*n*)= $\left\lfloor \frac{n}{2} \right\rfloor$, ahol $n \in \mathbb{N}$
- *Fele*(*n*)=$\left\lceil \frac{n}{2} \right\rceil$, ahol $n \in \mathbb{N}$
- $\left.\begin{array}{l} [u..v]\ =\ \{i\ \in\ \mathbb{Z} : u\ \leq\ i\ \leq\ v\} \\ [u..v)\ =\ \{i\ \in\ \mathbb{Z} : u\ \leq\ i\ <\ v\ \} \end{array}\right\}\ ahol\ u, v \in \mathbb{Z}$
- *p*, *q*, *r*, *s*, *t*, *H*, *L*: pointerek
- $\mathcal{T}$: ismert, de meg nem nevezett típust
  - értékadás (pl. x := y)
  - teljes rendezés (az =,≠ , ≤, ≥ összehasonlításokkal)

Változók

- Láthatósága és hatásköre: őket tartalmazó struktogram
- Élettartamuk:
  - az első, őket tartalmazó utasítás végrehajtásától a struktogram végéig
- Kivétel: globális változók:
  - program egészében élnek és láthatók
  - Deklarálni kell a **global** prefix segítségével
  - Lokálisan nem definiálhatók felül

[utasítások]

- Bizonyos programozási környezetekben szükséges lehet
  - Elhagyhatók

Tömb (Array)

- Tömbhivatkozás (array reference)
  - tömb hossza (length, elemszáma)
  - mutató (pointer):
    tömbtárolóra hivatkozik

+

- Tömbtároló (array storage)
  - a tömb elemeit tartalmazó memóriacelláknak a memóriában folytonos sorozata

- $A : \mathcal{T}[n]$ deklaráció kiértékelése
  - Létrehoz:
    - $\mathcal{T}$ elemtípusú, n elemű tömbtárolót ($n \in \mathbb{N}$)
    - $A : \mathcal{T}[]$ típusú tömbhivatkozást
  - Élettartam: a deklarációt tartalmazó programblokk
    - befejeződésekor a tömbhivatkozással együtt a tömbtároló is automatikusan törlődik

Tömbök indexelése

- A tömb ismeri a saját méretét
  - $A.length = n$
  - nem változtatható meg
  - Az *A.pointer* mutató a tömb első elemére hivatkozik
- A tömböket alapértelmezésben o-tól indexeljük.
  - $*A.pointer = A[o]$, és $*(A.pointer + i) = A[i]$, ahol $i \in [o..n)$.
  - $B/k : \mathcal{T}[n]$ deklaráció jelentése:
    - *B* tömb hossza: *n*
    - Kezdőindex: k ($k \in \mathbb{Z}$)
    - $B[k] = *B.pointer$, és $B[i] = *(B.pointer + i-k)$, ahol $i \in [k..k+n)$.

Tömbök

- $A[u..v]$ résztömb: $<A[u], A[u+1], \ldots, A[v]>$
  - Ha $u>v$ => a résztömb és a reprezentált sorozat üres
- $A[u..v)$ résztömb: $<A[u], A[u+1], \ldots, A[v-1]>$
  - Ha $u \geq v$ => a résztömb és a reprezentált sorozat üres
- A $\mathcal{T}$ elemtípusú tömbökre hivatkozó P pointer deklarálása:
  - $P: \mathcal{T}[]$ (vagy $Q/k : \mathcal{T}[]$)
  - Ezután a P pointer inicializálható pl. a $P := A$ értékadó utasítással
  - a $P[1] := 5$ => $A[1] = 5$
- Új tömb-objektum dinamikus létrehozása, törlése:
  - $P :=$ new $\mathcal{T}[n]$ <=> delete P

Résztömb és tömbhivatkozás

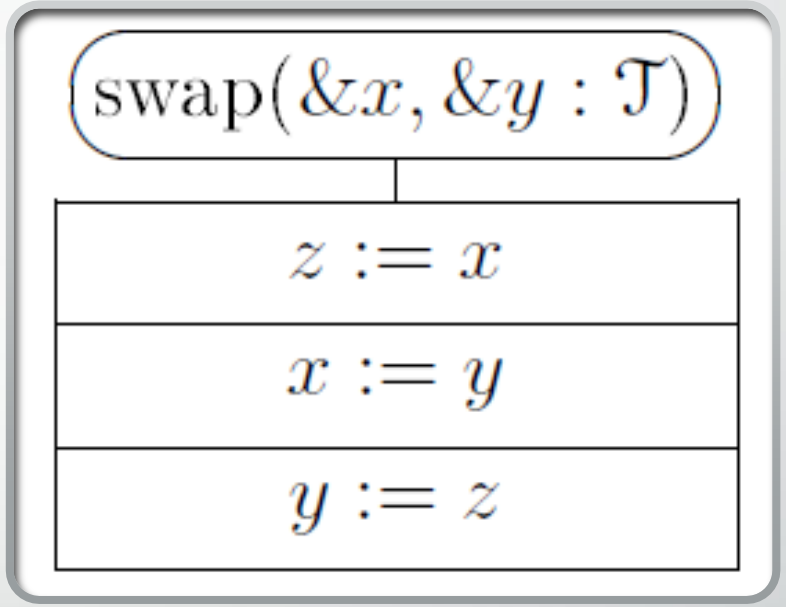
- A tömbhivatkozások résztömböket is azonosíthatnak
  - $A : \mathbb{Z}[5] = \langle 2, 3, 5, 7, 11 \rangle$
  - $P : \mathbb{Z}[]$
  - $P := A[1 \,..\, 4)$
    - $P = A[1 \,..\, 3] = \langle 3, 5, 7 \rangle$
  - $P[1] := 19$
    - $P[1]$ és A[2] ugyanazt a memóriacellát jelölik
    - $A[2] = 19$
      - $A = \langle 2, 3, 19, 7, 11 \rangle$, $P = \langle 3, 19, 7 \rangle$
  - $P := P[1 \,..\, P.\text{length})$
    - Levágjuk a P által hivatkozott résztömb 1. elemét
      - $P = \langle 19, 7 \rangle$
      - az A tömböt nem módosítja

A struktogramok paraméterlistái

- Alprogramok:
  - Eljárások
  - Függvények
  - Osztályok metódusai
- Stuktogram fejléce:
  - Alprogram_Név ([form. par. lista]):[Vissz.érték típusa]
  - Ha egy struktogramhoz csak név tartozik: a benne található kód a hívás helyén lenne
  - Globális / lokális változó

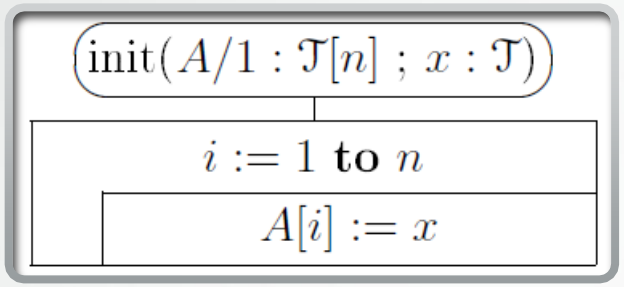
Cím szerinti /érték szerinti paraméterátadás

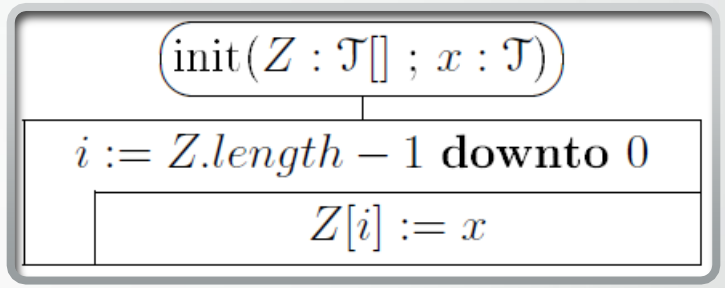
- A skalár típusok (szám, pointer, felsorolás)
  - Alapértelmezésben: érték szerinti paraméterátadás
  - cím szerinti: &
- Aktuális paraméterlista
  - Nem jelöljük külön a cím szerinti paraméterátadást
  - Pl. swap(x, y)
- A nem-skalár típusok
  - csak cím szerint adhatók át
    - ➢ nem jelöljük a paraméterátadás módját
  - összetett típusok: tömb, sztring, rekord, fájl, halmaz, zsák, sorozat, fa, gráf típusok
  - struct, illetve class kulcsszavakkal definiált osztályok
  - linearSearch($A : \mathcal{T}[n]$; $x : \mathcal{T}$) : $\mathbb{N}$
  - sort($B/1 : \mathcal{T}[]$)

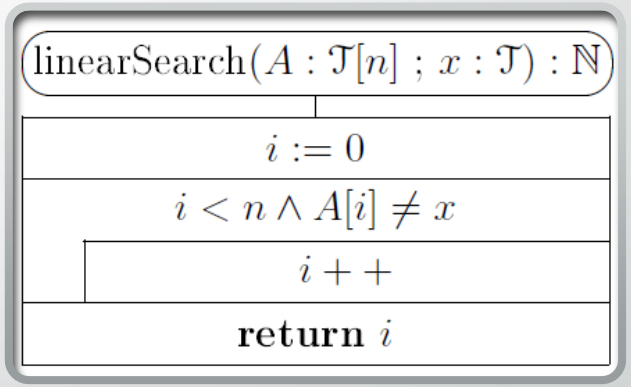


Eljárások, függvények, ciklusok, rekurzió

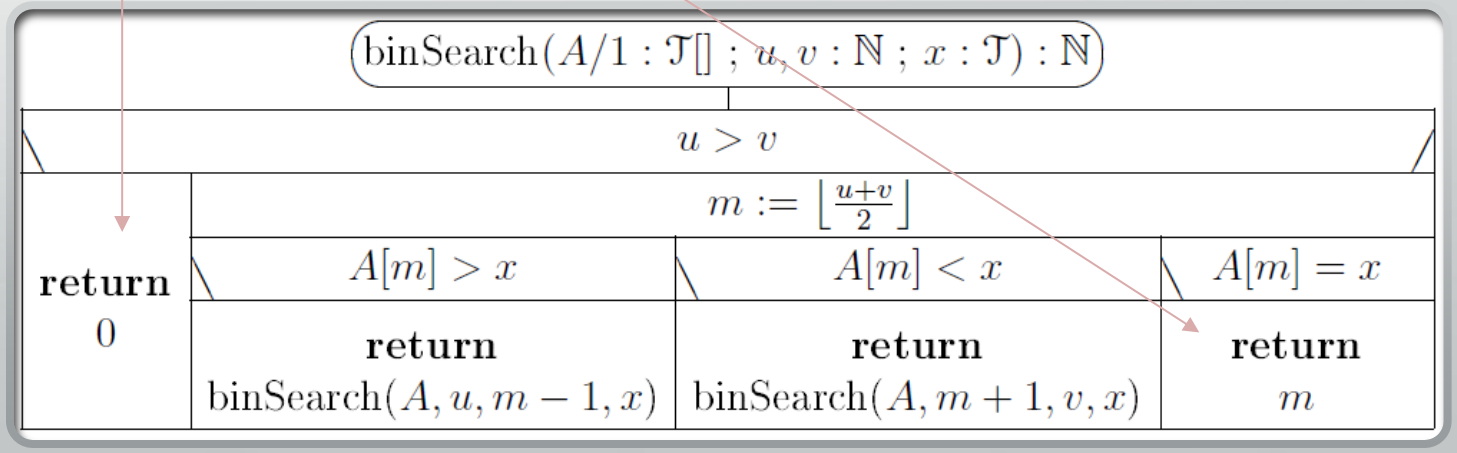
- Eljárás
- Függvény
  - *i*:=linearSearch(*A* , *x*)
- Rekurzió
  - Önmagát meghívó függvény
  - binarySearch(*A*, *x*)
  - binSearch(*A*, *u*, *v*, *x*)

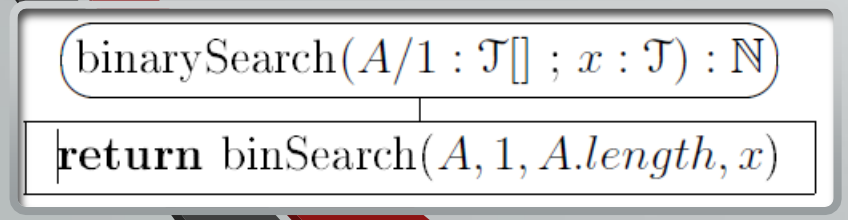






Leálló ágak: alapesetek





Programok, alprogramok és hatékonyságuk

- A műveletigényeket *n* függvényében adjuk meg
  - *n*: a formális paraméterben megadott tömb mérete
- init($A/1 : \mathcal{T}[n]$; $x : \mathcal{T}$) eljárás:
  - pontosan *n* iteráció
    - $T(n) = n+1$, $T(n) \in \Theta(n)$
- linearSearch($A : \mathcal{T}[n]$; $x : \mathcal{T}$) : $\mathbb{N}$ függvény
  - legjobb eset:
    - $mT(n)=1$, $mT(n) \in \Theta(1)$ (A tömb első eleme)
  - legrosszabb eset:
    - $MT(n) = n+1$, $MT(n) \in \Theta(n)$ (A tömb nem tartalmazza *x*-et)

Programok, alprogramok és hatékonyságuk

- binarySearch(A : $\mathcal{T}[n]$; $x : \mathcal{T}$) : $\mathbb{N}$ függvény
  - legjobb eset:
    - $mT(n)=2$, $mT(n) \in \Theta(1)$ (x a tömb $\left\lfloor \frac{n+1}{2} \right\rfloor$. eleme)
  - legrosszabb eset:
    - $MT(n) \in \Theta(\log n)$ (x nincs a tömbben)
    - az előnye a linearSearch-höz képest az n növelésével tovább nő : $\lim_{n\to\infty} \frac{\log n}{n} = 0$
- $AT(n)$ – átlagos műveletidő
  - $mT(n) \leq AT(n) \leq MT(n)$

Algoritmus, elő- és utófeltétel

- *Algoritmus*:
  - egy jól definiált kiszámítási eljárás
  - valamely adatok (*bemenet* vagy *input*) felhasználásával
  - újabbakat (*kimenet*, *eredmény* vagy *output*) állít elő
  - Pl. két egész szám legnagyobb közös osztójára: [lnko(x, y : $\mathbb{Z}$) : $\mathbb{Z}$]
- Az algoritmus bemenete
  - egy adott *előfeltételnek* kell eleget tegyen.
  - Pl. lnko(x, y) függvénynél: x és y egész számok, és nem mindkettő nulla

Algoritmus, elő- és utófeltétel

- Ha az előfeltétel teljesül, a kimenet adott *utófeltételnek* kell eleget tegyen
  - Az algoritmus bemenete és a kimenete közt elvárt kapcsolat
- Az algoritmus
  - számítási lépésekből áll
  - szekvenciák, elágazások, ciklusok, eljárás- és függvényhívások segítségével
  - pszeudo-kódot (pl. struktogramokat) felhasználva formálunk algoritmussá.

Rendezési feladat

- Szinte minden komolyabb számítógépes alkalmazásban szükséges
  - a tárolt adatok hatékony visszakeresése miatt
- *kulcs*: olyan adat, aminek típusán teljes rendezés definiált
  - pl. egy szám vagy egy sztring
- **Bemenet:** *n* darab kulcs $\langle a_1, a_2, \ldots, a_n \rangle$ sorozata
- **Kimenet:** A bemenet egy olyan $\langle a_{p_1}, a_{p_2}, \ldots, a_{p_n} \rangle$ permutációja, amelyre $a_{p_1} \leq a_{p_2} \leq \ldots \leq a_{p_n}$
- Rendezés:
  - alapértelmezésben mindig monoton növekvő (nem-csökkenő) rendezést fogunk érteni
  - a rendezés megfeleljen a fenti specifikációnak

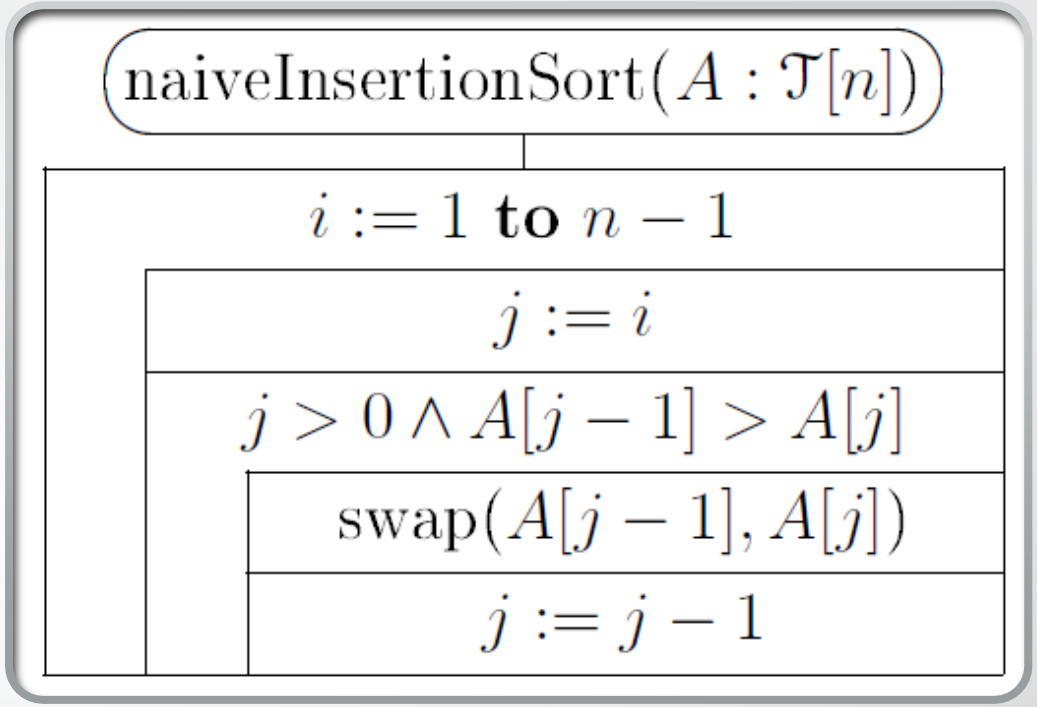
Rendezési feladat sorozatra

- Egy sorozatot legegyszerűbben egy tömbben tárolhatunk
- A tömb rendezéseknél feltesszük:
  - a tömb $\mathcal{T}$ elemtípusára teljes rendezés definiált
  - az értékadó utasítás is értelmezve van
- $A : \mathcal{T}\ [n]$ vagy $B/1 : \mathcal{T}\ [n]$
  - A és a B fizikailag tömbhivatkozások
    - (*length*: tömb hossza, *pointer*: tömbtároló)
    - ➢ alkalmasak a tömbök azonosítására

Rendezési feladat sorozatra

- a tömb elemei: (A[0], . . . ,A[n – 1] / B[1], . . . ,B[n])
  - n = 0 -> a tömbnek nincs eleme
- B[k..u]
  - k/u: első/utolsó elem indexe
  - k>u =>B[k..u] résztömb üres
- A[k..u)
  - k/u-1: első/utolsó elem indexe
  - k≥u =>A[k..u) résztömb üres
- B[1..n] / A[0..n) résztömb a B / A tömb minden elemét tartalmazza

Beszúró rendezés (Insertion sort)



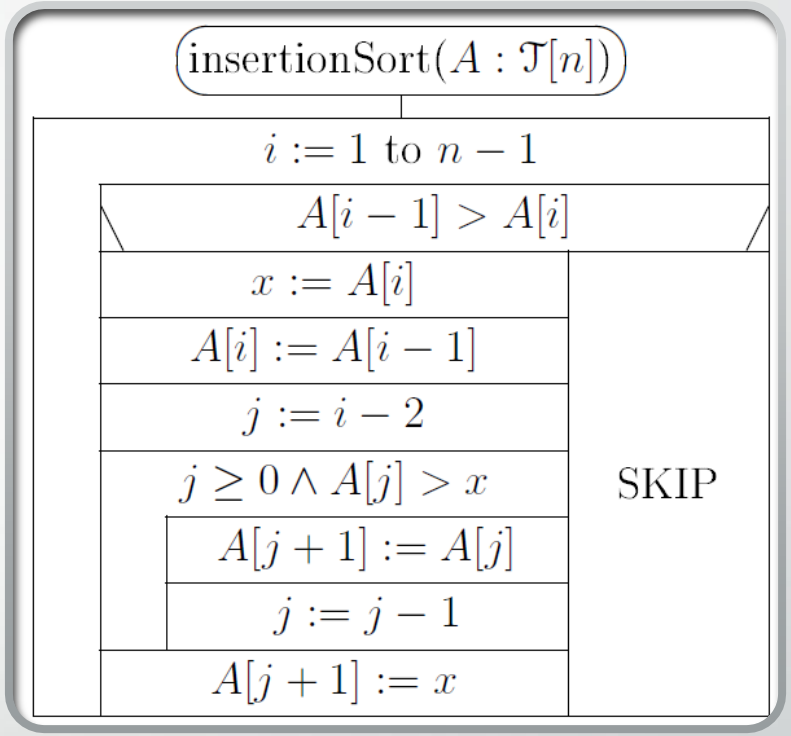
- A rendezés lépései: (pl. az ⟨5, 4, 2, 8, 3⟩)
  - Felosztjuk a sorozatot egy rendezett és egy ezt követő rendezetlen szakaszra
    - ⟨5 | 4, 2, 8, 3⟩
  - Beszúrjuk a rendezetlen szakasz első elemét a rendezett részbe a megfelelő helyre
    - → ⟨4, 5 | 2, 8, 3⟩
  - Ezt ismételgetjük, amíg a sorozat rendezetlen vége el nem fogy
    - → ⟨2, 4, 5 | 8, 3⟩ → ⟨2, 4, 5, 8 | 3⟩ → ⟨2, 3, 4, 5, 8 | ⟩ = ⟨2, 3, 4, 5, 8⟩

Beszúró rendezés optimalizálása

- A rendezett beszúrás technikája:
  - A sorozat tárolásától függ
- Naiv megoldás:
  - A beszúrandó elemet addig cserélgetjük a bal szomszédjával, amíg a helyére nem ér
  - ➢ Sok felesleges adatmozgatás
- Hatékonyabb:
  - Ha a helyén van -> nem mozdítjuk
  - Különben:
    - Elején kivesszük
    - Megkeressük a helyét
    - Visszatesszük a tömbbe

Naiv beszúró rendezés optimalizálása

- Beszúrás a beszúró rendezés alapváltozatában:
  - a rendezetlen szakasz első elemét (x) összehasonlítjuk a rendezett szakasz utolsó elemével (u)
    - $u \leq x$
      - x a helyén van
      - a rendezett szakasz felső határát eggyel növeljük
    - $u > x$
      - x-et elmentjük egy temporális változóba
      - u-t az x helyére csúsztatjuk -> u régi helyén: „lyuk”
      - mozgatás, amíg a lyuknak van bal szomszédja, és ez > x
      - az x -et a „lyuk”-ba tesszük



⟨2, 4, 5, 8|, 3⟩

⟨2, 4, 5, 8,_⟩, x = 3

⟨2, 4, 5,_, 8⟩, x = 3

⟨2, 4,_, 5, 8⟩, x = 3

⟨2,_, 4, 5, 8⟩, x = 3

⟨2, 3, 4, 5, 8⟩

Beszúró rendezés C# kódja*

```
// Teszt
public static void Main()
{
    int[] array = { 9, 5, 1, 4, 3 };
    InsertionSort(array);
    Console.WriteLine(string.Join(", ", array));
}
```

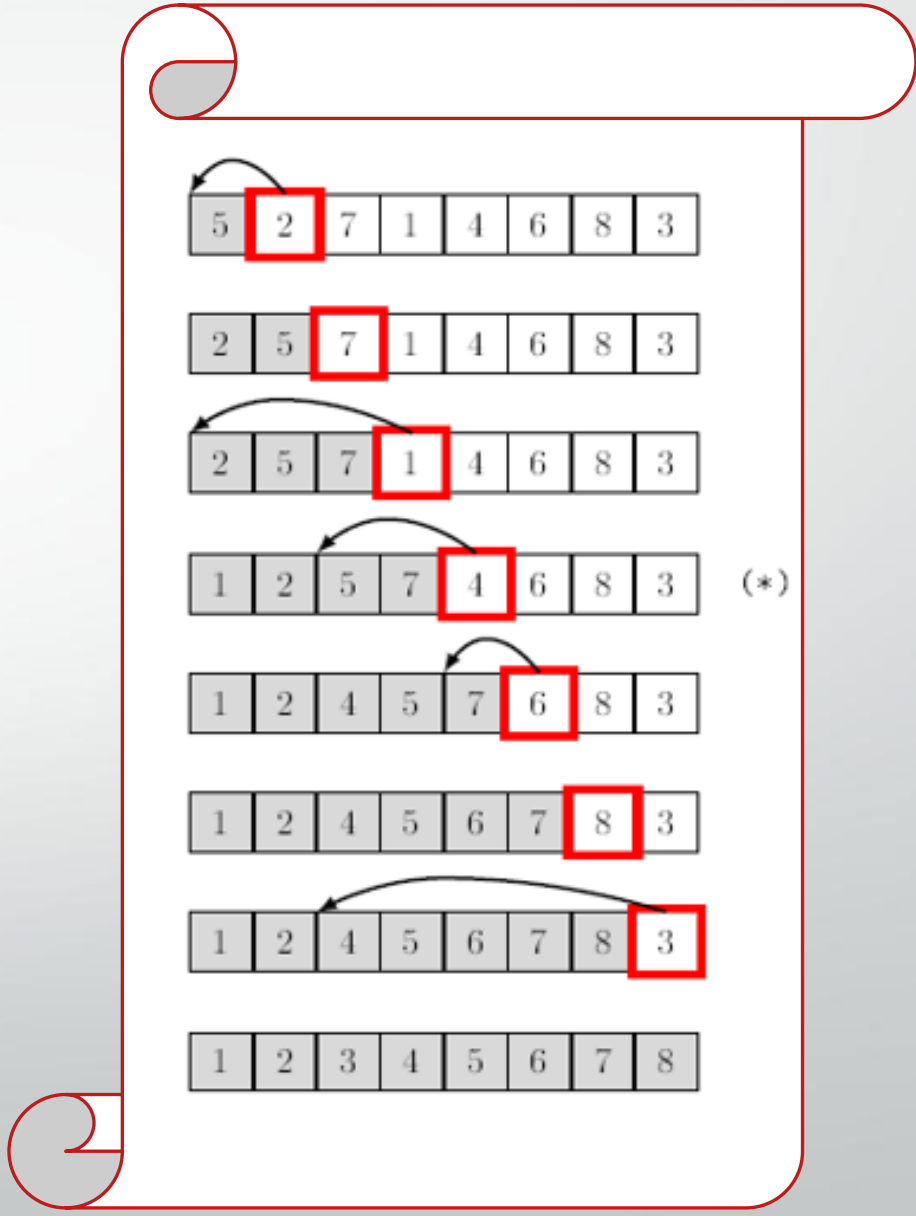
```
class InsertionSortAlgorithm
{
    public static void InsertionSort(int[] A)
    {
        int n = A.Length;
        for (int i = 1; i < n; i++)
        {
            if (A[i - 1] > A[i])
            {
                int x = A[i];
                A[i] = A[i - 1];
                int j = i - 2;

                while (j >= 0 && A[j] > x)
                {
                    A[j + 1] = A[j];
                    j--;
                }

                A[j + 1] = x;
            }
        }
    }
```

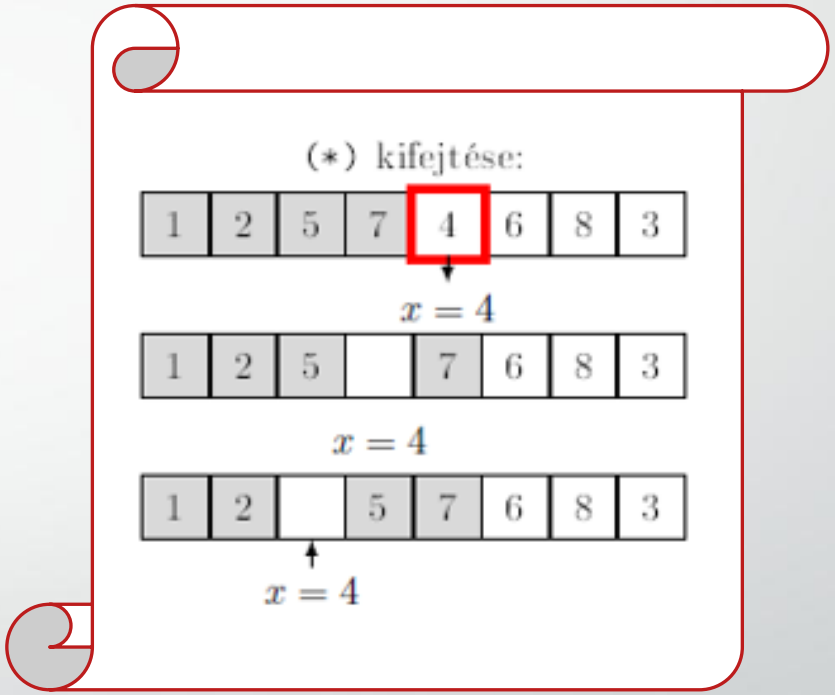
Beszúró rendezés

- A fő ciklus invariánsa:
  - $1 \leq i \leq n \wedge$
  - $A[0..n)$ az input tömb egy permutáltja,
    - Ahol $A[0..i)$ prefixe monoton növekvően rendezett
- Összefoglalva:
  - Ha n < 2
    - az *A* tömb üres, vagy egyelemű -> rendezett
    - a program fő ciklusa egyszer sem fut le



Beszúró rendezés

- Összefoglalás folytatása:
  - Ha $n \geq 2$
    - $A[0..1)$ rendezett és $i := 1$-re fennáll az invariáns
    - A fő ciklus magja:
      - $A[i]$-t beszúrja a tömb rendezett szakaszába
      - i-t eggyel növeli
    - tartja az invariánst
  - Mikor i eléri az n értéket:
    - már a teljes A tömb rendezett
    - az eljárás befejeződik



S Program hatékonysága

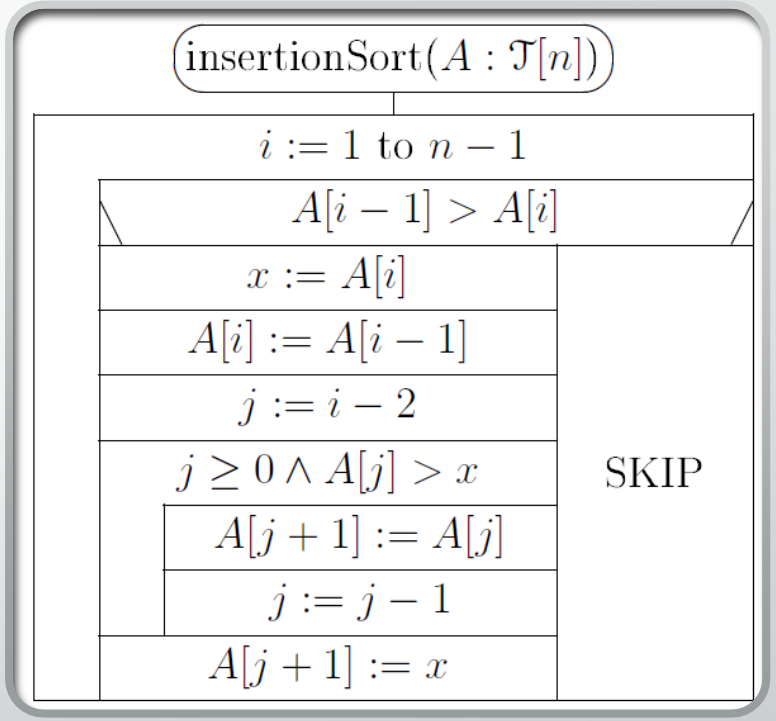
- ***Hatékonyság***: az eljárás erőforrás igénye
  - Futási ideje
  - Tárigénye
- Az algoritmusok erőforrásigényét a bemenet mérete függvényében szokás megadni
  - Rendező algoritmusoknál a rendezendő adatok száma (*n*)
- Futási idő (~ a műveletigény~ futási idő ~költség)
  - Sok ismeretlen körülmény
  - ➢ futási idő nem meghatározható

S Program hatékonysága

- Becslések:
  - Legrosszabb vagy maximális: $MT_S(n)$
  - Várható vagy átlagos: $AT_S(n)$
  - Legjobb vagy minimális $mT_S(n)$ esetek
  - Ha $MT_S(n) = mT_S(n)$ -> (def. szerint) $T_S(n)$ a minden esetre vonatkozó műveletigény
  - adott *n* méretű input esetén:
    - adott *S* algoritmus: alprogramhívások végrehajtásának száma
    - +kódban szereplő különböző ciklusok iterációinak száma

Beszúró rendezés hatékonysága

- Tárigény (space complexity)
  - a rendezendő tömbön kívül csak néhány segédváltozót igényel
  - ➢ Extra tárigénye minimális, *n*-től független konstans
  - ➢ $S_{IS}(n) \in \Theta(1)$ [*M* ~ memóriaigény, *IS* ~Insertion Sort]
- Futási idő
  - egyetlen eljáráshívás:
    - insertionSort($A : \mathcal{T}$ [])
  - Az eljárás fő ciklusa:
    - minden esetben pontosan (*n* – 1) -szer fut le

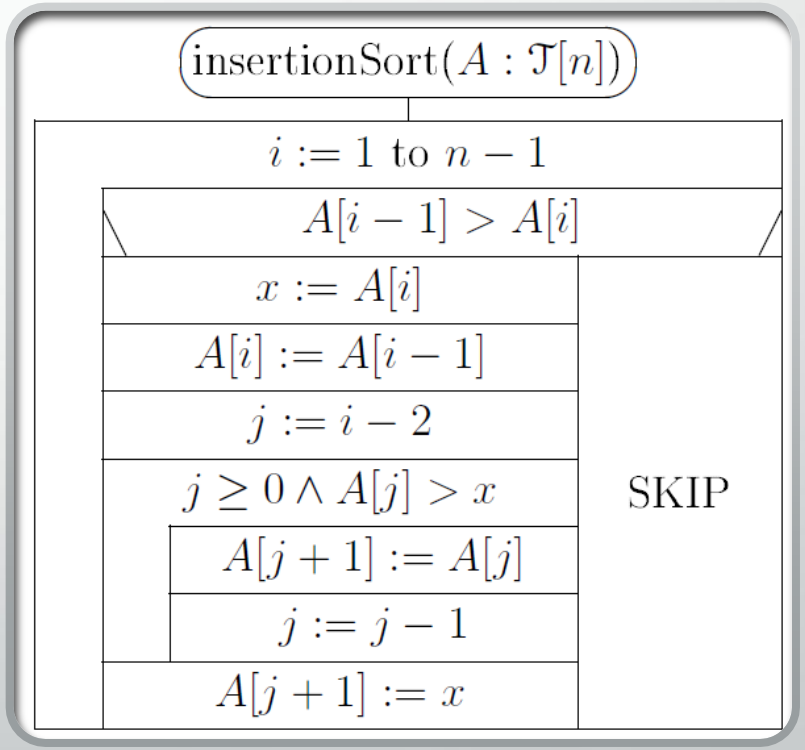


Becslés a beszúró rendezés minimális futási idejére $mT_{IS}(n)$ (n: a rendezendő tömb mérete)

- $A[1..n]$ eleve monoton növekvően rendezett
  - Külső ciklus: mindig jobb ág
  - Belső ciklus: egyet sem iterál

$$mT_{IS}(n) = 1 + (n - 1) = n \ (\in \Theta(n))$$

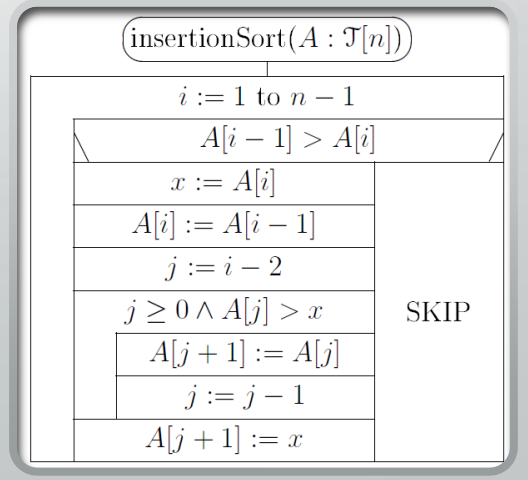
- 1 eljáráshívás + a külső ciklus ($n - 1$) iterációja.



Beszúró rendezés maximális futási ideje $MT_{IS}(n)$ (ahol n a rendezendő tömb mérete)

- $A[1..n]$ eleve monoton csökkenően rendezett
  - Külső ciklus: mindig bal ág
  - Belső ciklus: mindig j=-1-ig fut
- Egy eljáráshívás + a külső ciklus ($n - 1$) iterációja +
  a külső ciklus adott i értékkel való iterációjakor a belső ciklus maximum (i–1)-szer iterál
  - i: 1-től n-1-ig fut
  - a belső ciklus összesen legfeljebb $\sum_{i=1}^{n-1}(i-1)$
  - $MT_{IS}(n) = 1 + (n-1) + \sum_{i=1}^{n-1}(i-1) = n + \sum_{j=0}^{n-2} j = n + \frac{(n-1)*(n-2)}{2}$

$$MT_{IS}(n) = \frac{1}{2}n^2 - \frac{1}{2}n + 1 \ (\in \Theta(n^2))$$

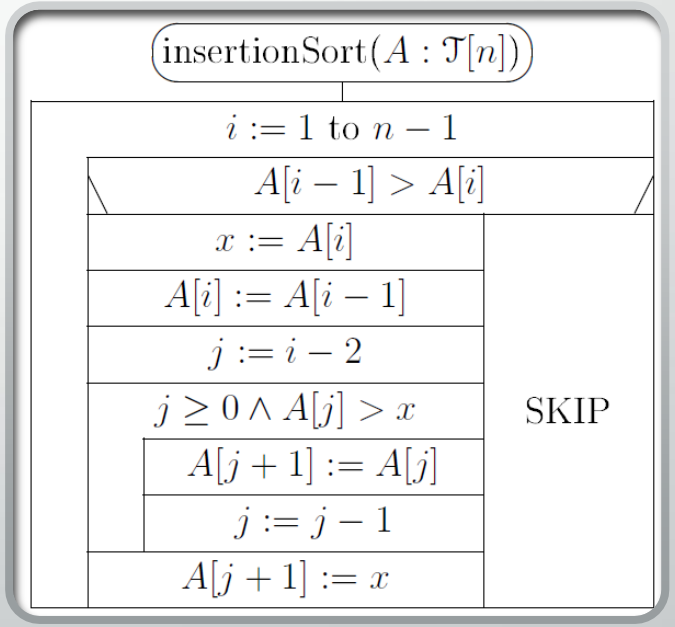


Beszúró rendezés átlagos futási ideje $AT_{IS}(n)$ (ahol n a rendezendő tömb mérete)

- Probléma:
  - nem ismerjük az input sorozatok eloszlását
- Véletlenített input sorozat esetén
  - Beszúrásnál átlagosan az elemek fele > a beszúrandó elemnél:
  - $AT_{IS}(n) \approx 1 + (n-1) + \sum_{i=1}^{n-1}(\frac{i-1}{2}) = n + \frac{1}{2} * \sum_{j=0}^{n-2} j =$

$$= n + \frac{1}{2} * \frac{(n-1)*(n-2)}{2} = \frac{1}{4}n^2 + \frac{1}{4}n + \frac{1}{2}$$

$$\boldsymbol{AT_{IS}(n) \approx \frac{1}{4}n^2 \in \Theta(n^2)}$$

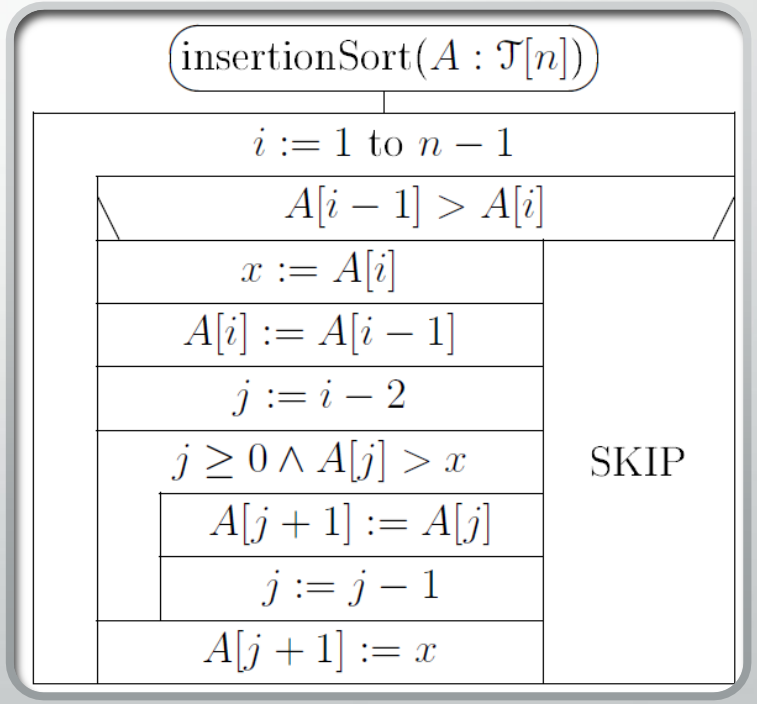


Beszúró rendezés átlagos futási ideje $AT_{IS}(n)$ (ahol n a rendezendő tömb mérete)

- Előrendezett inputok esetén: lineáris időben tudunk rendezni ~ optimális
  - Monoton növekvően előrendezett inputok:
    - az input sorozat elemeinek a rendezés utáni helyüktől való távolsága:
      - n -től független k konstanssal felülről becsülhető
    - a végső pozíciójuktól távolabb lévő elemek száma:
      - n -től független s konstanssal felülről becsülhető
    - Belső ciklus futási ideje max. $(k+s)*n$
    - Műveletigény: Lineáris ($\Theta(n)$)
  - Monoton csökkenően előrendezett inputok:
    - Műveletigény: közelít a legrosszabb esethez
    - n milliós (v. >) nagyságrendű –> nagyon hosszú futási idő –> használhatatlan
    - Ha ezt tudjuk: Fordítsuk meg a sorozatot ($\Theta(n)$) => monoton növekvően előrendezett sorozat lesz.

A futási időkre vonatkozó becslések*

- *MrT*(*n*) és *mrT*(*n*): insertionSort(*A* : $\mathcal{T}[n]$) tényleges maximális és minimális futási ideje
- Becslés a minimális futási időre (*mrT*(*n*))
  - Ha a fő ciklus minden elemet a végső helyén talál
    - mindig a jobb oldali ágon fut le
    - ha a tömb már eleve monoton növekvően rendezett
  - L! ***a***: a fő ciklus jobb oldali ága egyszeri végrehajtásának futási ideje
  - L! ***b***: T(eljárás meghívása) + T(a fő ciklus előkészítése és befejezése) + T(az eljárásból való visszatérés)
    - *a* és *b* pozitív konstansok
    - $mrT(n) = a * (n - 1) + b$



A futási időkre vonatkozó becslések*

- L! $p = \min(a, b)$ és $P = \max(a, b)$ $(\Rightarrow 0 < p \leq P)$
  - $p * (n - 1) + p \leq mrT(n) = a * (n - 1) + b \leq P * (n - 1) + P$
  - $p * n \leq mrT(n) \leq P * n$
  - $p * mT_{IS}(n) \leq mrT(n) \leq mT_{IS}(n)$
- Becslés a maximális futási időre ($MrT(n)$)
  - ha mindig a külső ciklus bal ágát hajtja végre és a belső ciklus j = -1-ig fut
    - ha a tömb már eleve szigorúan monoton csökkenően rendezett

A futási időkre vonatkozó becslések*

- L! ***d***: a belső ciklus egy lefutásának a műveletigénye
- L! ***c***: T(külső ciklus bal ága egy lefutása)- T(a belső ciklus lefutásai)+T(a belső ciklusból való kilépés)
  (amibe beleértjük a belső ciklus feltétele utolsó kiértékelését azaz a j = -1 esetet)
  - c és d pozitív konstansok
- $MrT(n) = b + c * (n-1) + \sum_{i=1}^{n-1} d * (i-1) =$

$$= b + c * (n-1) + d * \sum_{j=0}^{n-2} j =$$

$$= b + c * (n-1) + d * \frac{(n-1) * (n-2)}{2}$$

A futási időkre vonatkozó becslések*

- L! q= min(b,c,d) és Q = max(b,c,d) (=> o < $q \leq Q$)
- $q + q * (n-1) + q * \frac{(n-1)*(n-2)}{2} \leq MrT(n) \leq Q + Q * (n-1) + Q * \frac{(n-1)*(n-2)}{2}$
- $q * \left(n + \frac{(n-1)*(n-2)}{2}\right) \leq MrT(n) \leq Q * (n + \frac{(n-1)*(n-2)}{2})$
- $q * MT_{IS}(n) \leq MrT(n) \leq Q * MT_{IS}(n)$

➢ Mindkét esetben *a valódi futási idő* alulról és felülről becsülhető (*T*(*az eljáráshívások*)+*T*(*a ciklusiterációk számának*))*pozitív konstans

➢ ugyanolyan nagyságrendű

Rendezések stabilitása

- Egy rendezés stabil (stable):
  - ha megtartja az egyenlő kulcsú elemek eredeti sorrendjét
- A stabilitás (stability) előnyös tulajdonság:
  - Pl. rekordoknál, ha vannak azonos kulcsú rekordok
- A stabilitás nélkülözhetetlen tulajdonság:
  - pl. (lineáris műveletigényű) radix rendezésnél

Rendezések stabilitása

Stabil rendezések

Beszúró rendezés
(insertion sort)

- az előadás szerinti változatban
- A tömb rendezett szakaszába az újabb elemeket jobbról balra szúrjuk be
- A beszúrandóval egyenlő kulcsú elemeket már nem lépjük át

Összefésülő rendezés
(merge sort)

Nem stabil rendezések

Kupacrendezés
(heap sort)

Gyorsrendezés
(quicksort)

Kiválasztó rendezések (selection sorts)

- Minimumkiválasztásos rendezés:
  - Megkeressük az A[0..n) tömb minimális elemét -> csere(min, A[0])
  - Megkeressük a A[1..n) résztömb legkisebb elemét -> csere(min, A[1])
  - Folytassuk ezen a módon az A[0..n) első (n – 1) elemére!
- Tömb
  - rendezett szakasz: a tömb elején kezdetben üres
  - rendezetlen szakasz: tömb vége
- Műveletidő
  - $\ddot{O}(n) = \sum_{i=1}^{n-1} i = \frac{n*(n-1)}{2} \in \Theta(n^2)$
  - $Cs(n) = n - 1$

⟨3; 9; 7; 1; 6; 2⟩

⟨1 | 9; 7; 3; 6; 2⟩

⟨1; 2 | 7; 3; 6; 9⟩

⟨1; 2; 3 | 7; 6; 9⟩

⟨1; 2; 3; 6 | 7; 9⟩

⟨1; 2; 3; 6; 7 | 9⟩

⟨1; 2; 3; 6; 7; 9⟩

„oszd meg és uralkodj” elv

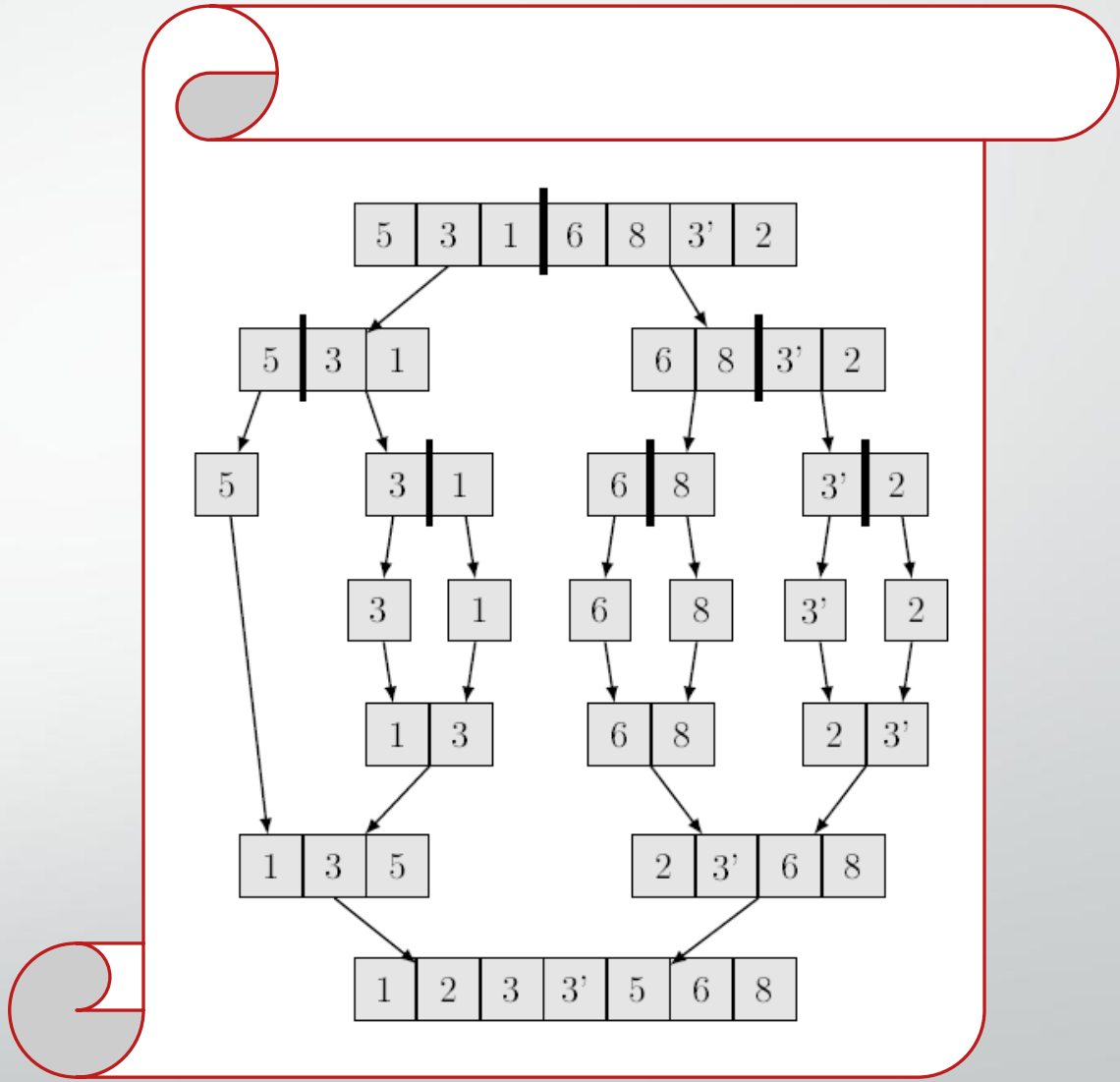
1. Az adott problémát két (vagy több) részfeladatra bontjuk
   - Eredetihez hasonló
   - Egyszerűbb (kisebb)

2. Ezeket megoldjuk
   - Ha a megoldandó (rész)probléma elég egyszerű -> közvetlenül
   - Egyébként: ugyanazzal az algoritmussal

3. A részeredményekből összerakjuk a felbontott probléma megoldását

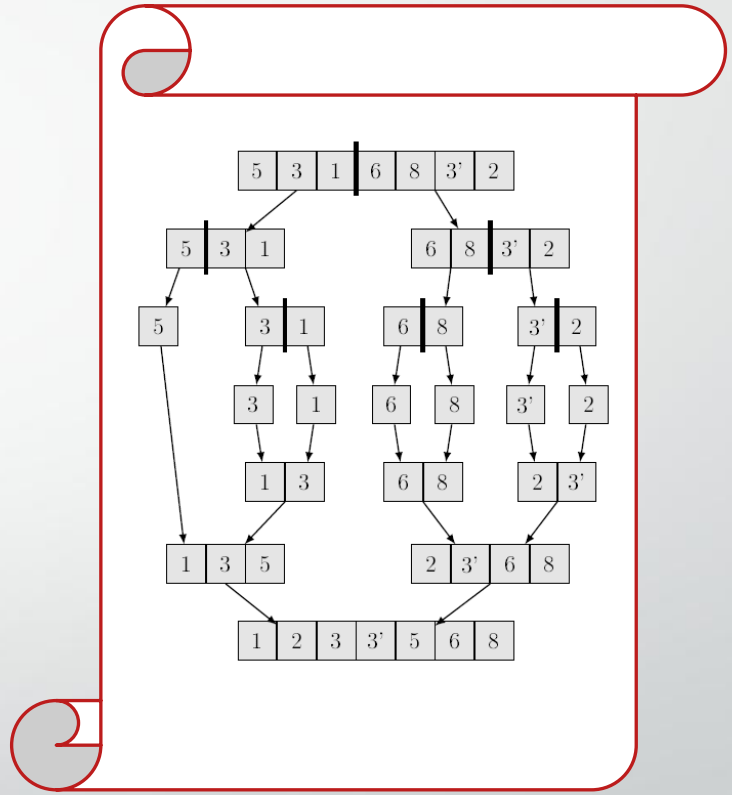
Összefésülő rendezés (Merge Sort)

- „oszd meg és uralkodj” elv
- Adott: egy rendezendő kulcssorozat
- 2 eset:
  - Üres és az egyelemű sorozatok:
    - eleve rendezettek
  - Hosszabb sorozatok:
    - A két fél-sorozatot ugyanezzel a módszerrel rendezzük
    - A rendezett fél-sorozatokat rendezetten összefésüljük

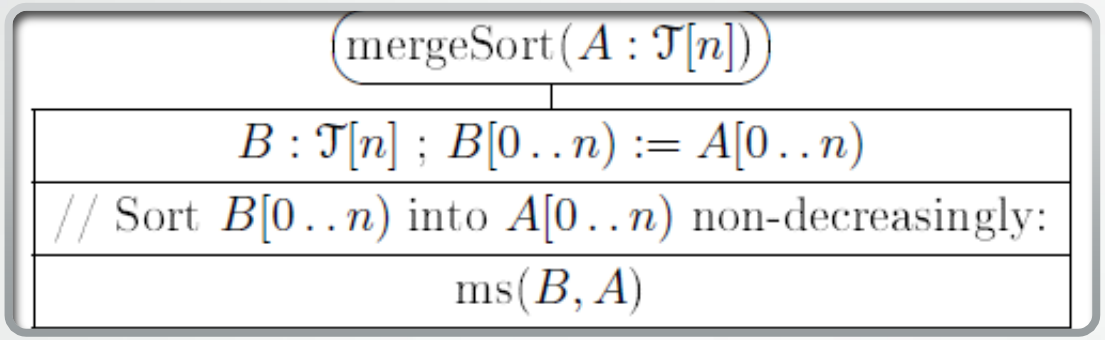


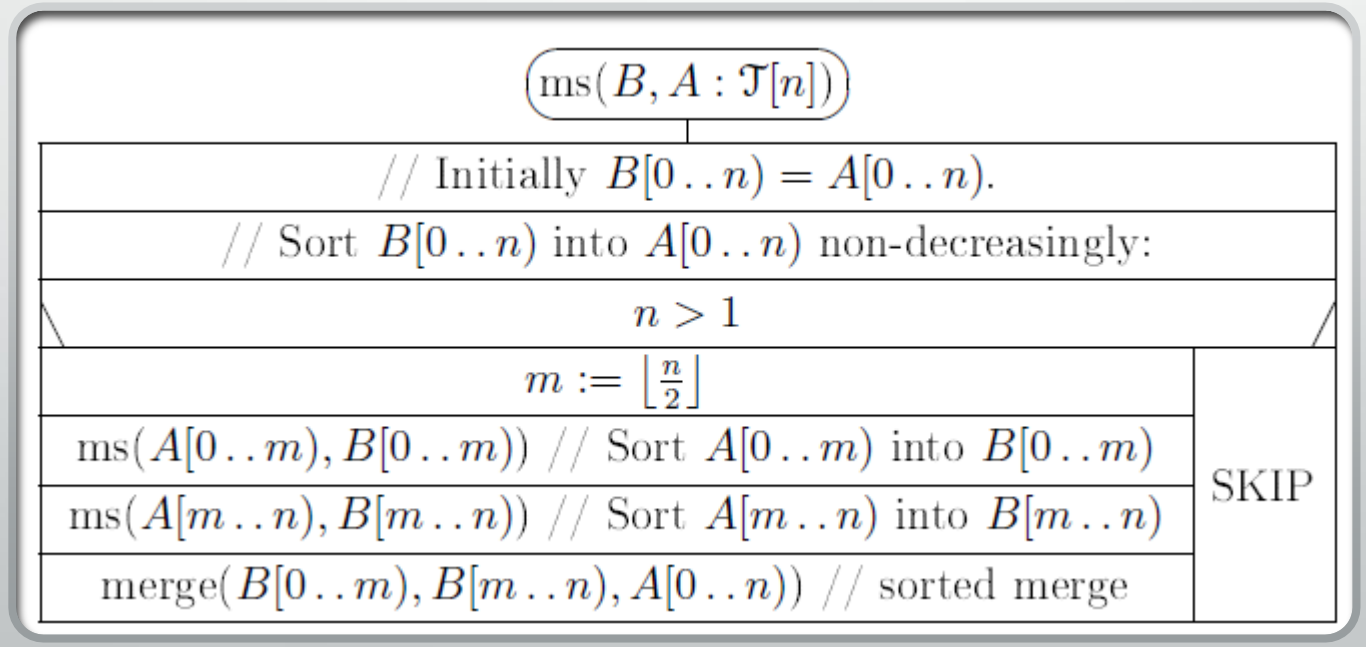
Összefésülő rendezés tulajdonságai, szemléltetése

- Stabil rendezés
  - megőrzi az egyenlő kulcsú elemek bemeneti sorrendjét
  - rendezett résztömbök összefésülésekor, egyenlő kulcsok esetén a bal oldali résztömbből származó kulcsot tesszük először a helyére
- Műveletigénye
  - $MT_{MS}(n), mT_{MS}(n) \in \Theta(n \log n)$
  - $MT_{MS}(n)$ aszimptotikusan optimális az ún. összehasonító rendezések között
  - nagy elemszámú sorozatokat is viszonylag gyorsan rendez

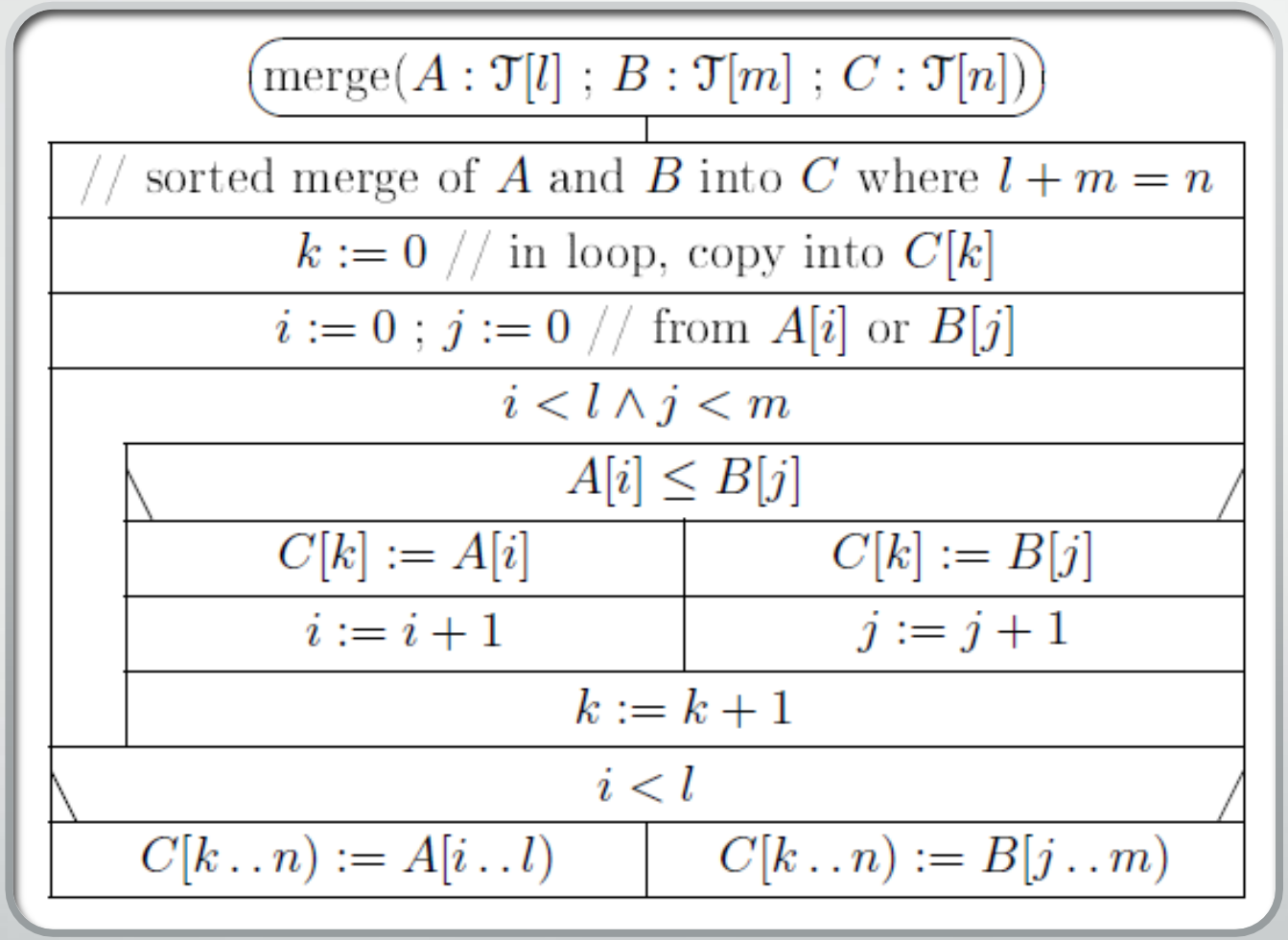


Összefésülő rendezés algoritmusa





Összefésülő rendezés algoritmusa



Összefésülő rendezés C# kódja*

```
class MergeSortAlgorithm
{
    public static void MergeSort(int[] A)
    {
        int n = A.Length;
        int[] B = new int[n];
        Array.Copy(A, B, n);
        Ms(B, A, 0, n);
    }

    private static void Ms(int[] B, int[] A, int left, int right)
    {
        // Sort B[left..right) into A[left..right)
        int n = right - left;
        if (n <= 1)
            return;

        int m = left + n / 2;

        // Recursively sort A[left..m) into B[left..m)
        Ms(A, B, left, m);

        // Recursively sort A[m..right) into B[m..right)
        Ms(A, B, m, right);
        // Merge sorted halves B[left..m) and
        B[m..right) into A[left..right)
        Merge(B, left, m, B, m, right, A, left);
    }
```

```
    public static void Main()
    {
        int[] array = { 5, 2, 9, 1, 5, 6 };
        MergeSort(array);
        Console.WriteLine(string.Join(", ", array));
    }
}
```

```
private static void Merge(int[] A, int aStart, int aEnd, int[] B, int bStart, int bEnd, int[]
{
    int i = aStart, j = bStart, k = cStart;

    while (i < aEnd && j < bEnd)
    {
        if (A[i] <= B[j])
        {
            C[k++] = A[i++];
        }
        else
        {
            C[k++] = B[j++];
        }
    }

    while (i < aEnd)
    {
        C[k++] = A[i++];
    }

    while (j < bEnd)
    {
        C[k++] = B[j++];
    }
}
```

Merge vizsgálata

- A merge sort stabilitását a merge() eljárás explicit ciklusának elágazása biztosítja:
  - $A[i] = B[j] \Rightarrow A[i]$-t másolja $C[k]$-ba
    - Az A résztömb megelőzi a B résztömböt a mindkettőt tartalmazó (rész)tömbben
- A merge() eljárás műveletigénye: $T_{mb}(n) = n \in \Theta(n)$
  - $C.length=n$
  - Az eljárás során ennek mindegyik eleme egy-egy iterációval áll elő A-ból vagy B-ből
    - Az explicit ciklus:
      - $C[0..k)$-t tölti fel, k iterációval.
    - Az implicit ciklusok:
      - $C[k..n)$:= . . . alakú utasításokba rejtve jelennek meg
      - csak az egyik fog végrehajtódni $n-k$ iterációval

➢ Összesen: $k+(n-k) = n$ iteráció

A merge sort műveletigénye: szemléletes megközelítés

- $MT_{MS}(n), mT_{MS}(n) \in \Theta(n \log n)$
- Bizonyítása szemléletesen: (matematikai biz. később)
  - A műveletek túlnyomó részét a merge(A, B, C) eljárás végzi el
    - $mT_{merge}(l), MT_{merge}(l) \in \Theta(l)$ (ahol l az aktuális résztömb hossza ($l = C.length$)
  - A rekurzív ms(B, A) eljárás minden hívásban felezi a rendezendő résztömb hosszát
  - ➢ A rekurziónak kb. $\log n + 1$ szintje van
  - Minden rekurziós szinten: a merge hívások résztömbjei együtt lefedik az egész A tömböt
    - Kivétel az alsó egy vagy két szint: kevesebb
  - ➢ Egy tetszőleges szint összes merge hívásának műveletigényét összeadva: $\Theta(n)$
  - A szintenkénti műveletigényt a szintek számával szorozva nagyságrendben $\Theta(n \log n)$

Tárigény

- Tárigény tömbökre:
  - *n* méretű segédtömb
  - néhány segédváltozó
  - a rekurzív hívások adminisztrálásának tárigénye: ~ log n

  ➢ $S_{MS}(n) \in \Theta(n + \log n) = \Theta(n)$

- Röviden:

$S_{MS}(n) \in \Theta(n)$

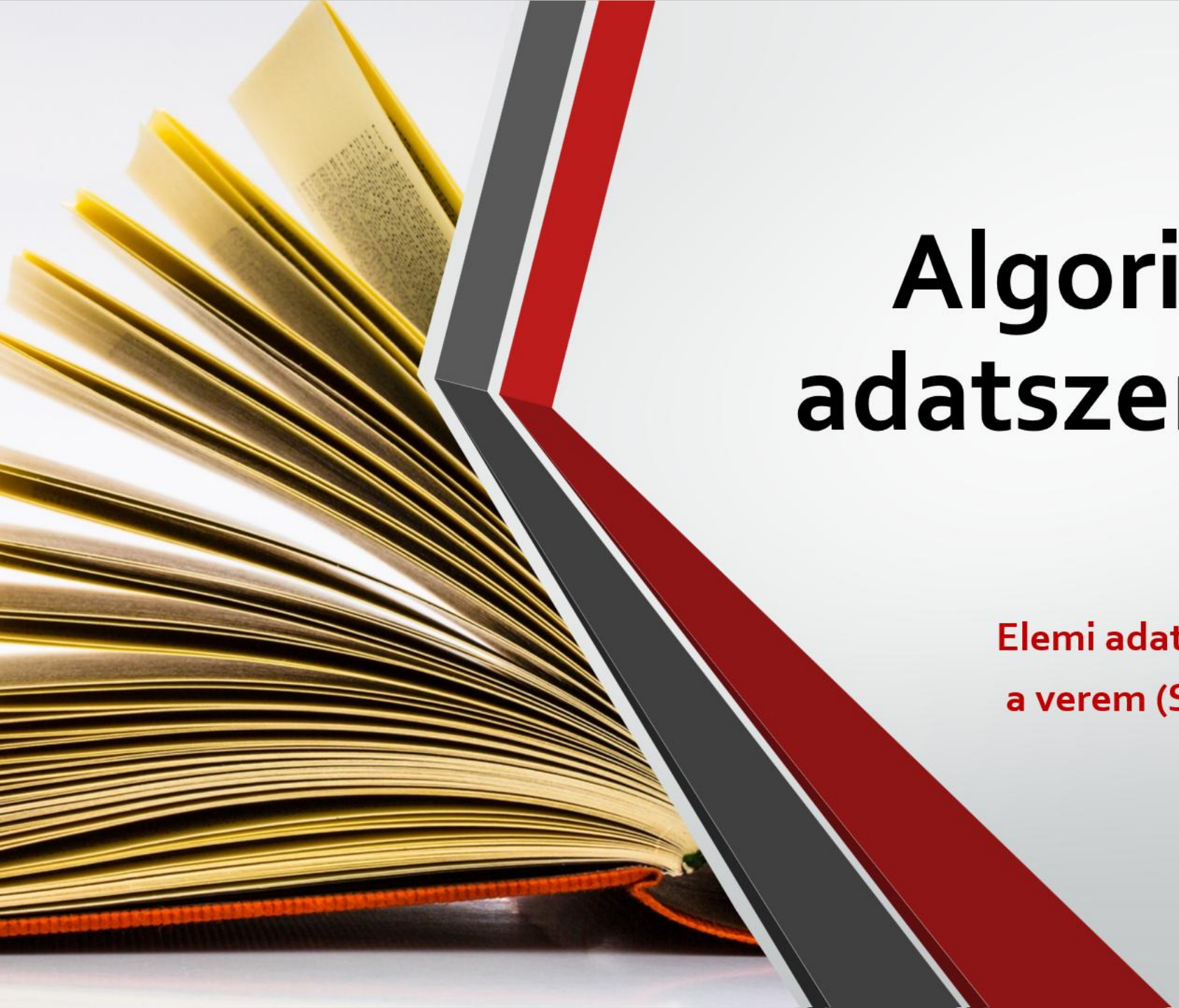
Ellenőrző kérdések

- Adja meg a rendezési feladat specifikációját!
- Mit nevezünk stabil rendezésnek?
- A tanult rendezések közül melyik stabil?
- Mennyi a beszúrásos rendezés műveletigénye és tárigénye? Miért?
- Mennyi az összefésüléses rendezés műveletigénye és tárigénye? Miért?
- Adja meg a tanult rendezőalgoritmusok struktogramját!
- Mit jelent az „oszd meg és uralkodj” elv? Melyik rendezés alapul ezen az elven?

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek I. előadásjegyzete alapján készült.



Algoritmusok és adatszerkezetek I.
2. Előadás

Elemi adatszerkezetek és adattípusok: a verem (Stack) és a sor (Queue) típus. Láncolás, egyirányú listák

Tartalom

Elemi adatszerkezetek és adattípusok

Adatszerkezet (data structure)

- Adatok tárolásának és elrendezésének egy lehetséges módja
  - adatok elérése, módosítása, beszúrása, törlése
- A programozási feladat megoldásának alapvető része:
  - A megfelelő adatszerkezetek kiválasztása vagy megalkotása
    - A programok hatékonysága nagymértékben függ az alkalmazott adatszerkezetektől

Elemi adatszerkezetek és adattípusok

Adattípus (data type)

- Adatszerkezet
- \+ a rajta értelmezett műveletek

Absztrakt adattípus (ADT)
(Abstract Data Type)

- Nem definiáljuk pontosan az adatszerkezetet
  - csak informálisan a műveleteket
- Az ADT megvalósításának részei: (UML)
  - reprezentálása:
    - adatszerkezet megadása (adattagok)
  - implementálása:
    - műveleti kód (metódusok)

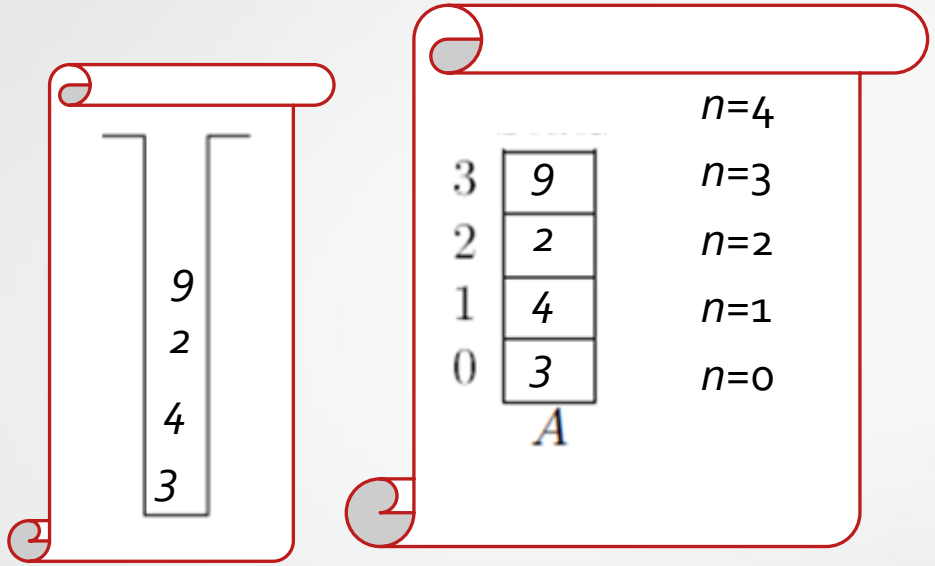
Verem (Stack) adattípus: LIFO (Last-In First-Out)

- Az utoljára benne eltárolt, és még benne lévő adat érhető el, illetve eltávolítható

Stack
$-\ A : \mathcal{T}[]$ // $\mathcal{T}$ is some known type ; $A.length$ is the physical $-$ constant $m0 : \mathbb{N}_+ := 16$ // size of the stack, its default is m0. $-\ n : \mathbb{N}$ // $n \in 0..A.length$ is the actual size of the stack
$+$ Stack$(m : \mathbb{N}_+ := m0)$ $\{A := \textbf{new}\ \mathcal{T}[m]\ ;\ n := 0\}$ // create empty stack $+ \sim$ Stack() { **delete** A } $+$ push$(x : \mathcal{T})$ // push x onto the top of the stack $+$ pop() : $\mathcal{T}$ // remove and return the top element of the stack $+$ top() : $\mathcal{T}$ // return the top element of the stack $+$ isEmpty() : $\mathbb{B}$ $\{\textbf{return}\ n = 0\}$ $+$ setEmpty() $\{n := 0\}$ // reinitialize the stack

Verem példa

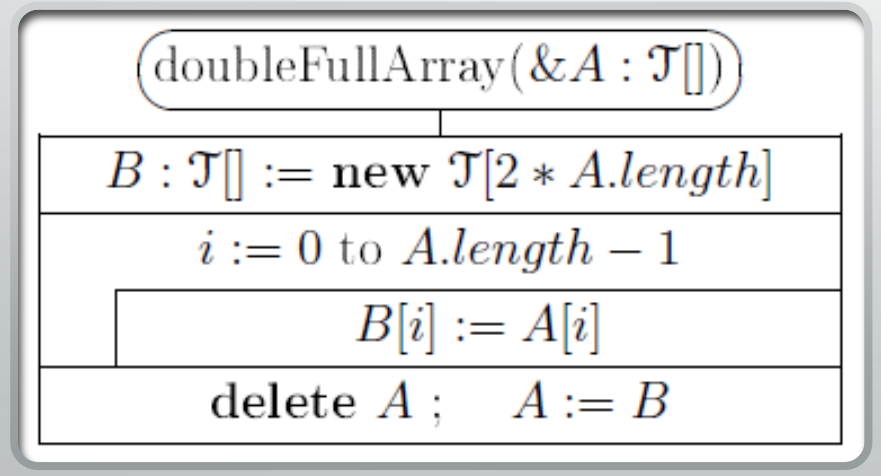
- Stack (4)
- push(3)
- push(4)
- push(2)
- push(9)
- pop() :9

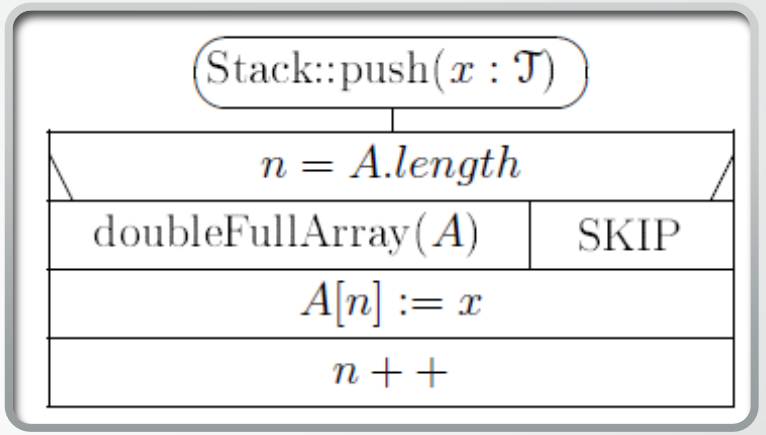


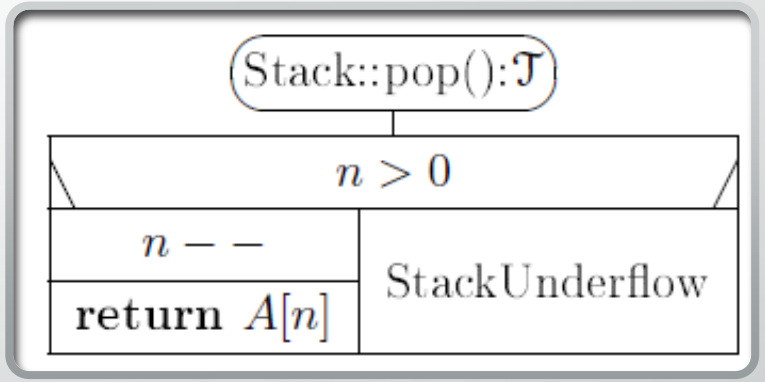
- Gyakorlati alkalmazása
  - CALL STACK
  - Kifejezések postfix formára hozása
  - Postfix forma kiértékelés

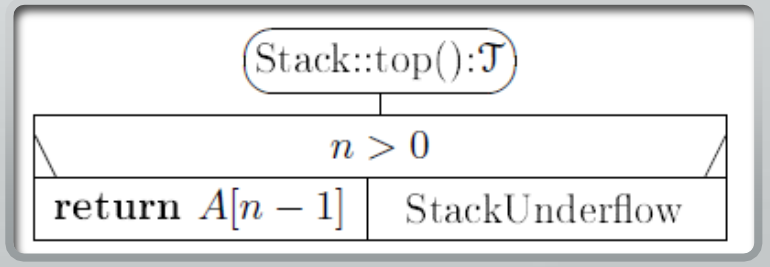
Verem műveletek

- Műveletidő: Θ(1)
  - Nem tartalmaz ciklust / rekurziót
- Push művelet:
  - *mT(n)* ∈ Θ(1)
  - *MT(n)* ∈ Θ(n)
  - *AT(n)* ∈ Θ(1)





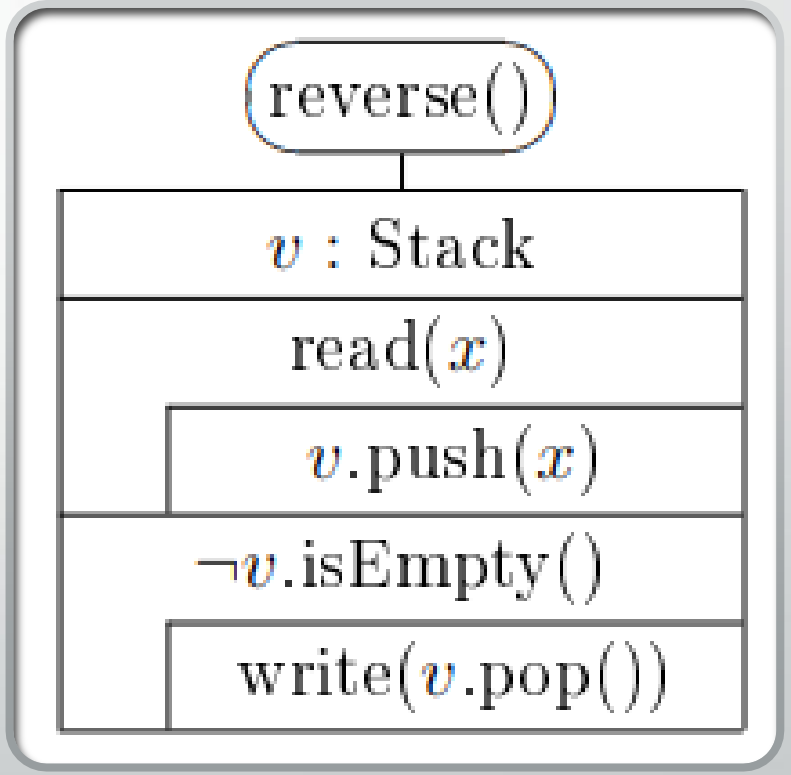




Példa a verem használatára:

Az input adatok kiírása fordított sorrendben

- read(&x:$\mathcal{T}$): $\mathbb{B}$ függvény
  - a kurrens inputról olvas
  - sikeres (visszatérés: igaz)
    - nincs még vége az inputnak
    - beolvassa x-be a következő input adatot
  - Sikertelen (visszatérés: hamis):
    - vége van az inputnak
    - x értéke definiálatlan
- write(x): a kurrens outputra írja x értékét



Példa a verem használatára: **Lengyel forma***

- Aritmetikai kifejezés
  - Infix alak: 4+5
  - Prefix alak: +4 5
  - Postfix alak: 4 5 +
- Lengyel forma:
  - Egy aritmetikai kifejezés postfix alakja
  - Jan Łukasiweicz 1920
  - Reverse Polish Notation
  - Nem tartalmaz zárójeleket

- Operandusok:
  - változó, konstans, postfix kifejezés
  - sorrendje nem változik, az infix kifejezéshez képest
  - Minden operátort közvetlen megelőznek az operandusai
- Operátorok
  - Műveleti jelek
  - Az elvégzésük sorrendjében szerepelnek

Operátorok precedenciája és asszociativitása*

Megnevezés	Operátor	Asszociativitás
Elsődleges operátorok	() [] . ->	balról jobbra
Egyoperandusú operátorok	**! ~ - ++ -- & * sizeof**	**jobbról balra**
Aritmetikai operátorok	**/ %** **+ -**	**balról jobbra**
Biteltoló műveletek	<< >>	balról jobbra
Relációs operátorok	**< <= > >=** **== !=**	**balról jobbra**
Bitműveletek		balról jobbra
Logikai műveletek	**&&** **||**	**balról jobbra**
Feltételes operátor	?:	jobbról balra
Értékadó operátorok	**= += -= ...**	**jobbról balra**
Vessző operátor	,	balról jobbra

- Asszocativitás:
  - Az azonos precedenciájú operátorokat tartalmazó kifejezésekben a kiértékelés iránya
- Kép forrása:
  - https://slideplayer.hu/slide/2140789/

Lengyel forma algoritmusváz*

Lengyel forma előállítása: Shunting-yard algoritmus

- Ha a soron következő karakter :
  - Operandus: → outputra
  - Operátor:
    - BJ típusú:
      1. → üríti a vermet az outputra
         - az első nálánál < precedenciájú operátorig (ezt már nem beleértve)
         - ha ilyen nincs => a verem aljáig
         - ha (-t lát => megáll
      2. → verembe

Lengyel forma algoritmusváz*

Lengyel forma előállítása: Shunting-yard algoritmus

- JB típusú:
  1. → üríti a vermet az outputra
     - az első nálánál ≤ precedenciájú operátorig (ezt már nem beleértve)
     - ha ilyen nincs => a verem aljáig
     - ha (-t lát => megáll
  2. → verembe
- (:→ verembe
-):
  1. → üríti a vermet az outputra a következő (-ig
  2. Kiveszi a (-et, de azt nem írja az outputra

- Ha a kifejezés valamennyi karakterét feldolgoztuk
  => a vermet kiürítjük az outputra

Lengyel formára hozás lejátszása*

- Infix alak:

a = b + c / (d - e) + f ^ g ^ (h * i) / j + (k - l) * m

- Verem:

- Lengyel forma:

a b c d e - / + f g h i * ^ ^ j / + k l - m * + =

Lengyel forma kiértékelése algoritmusváz*

Lengyel forma kiértékelése: evalRPN algoritmus

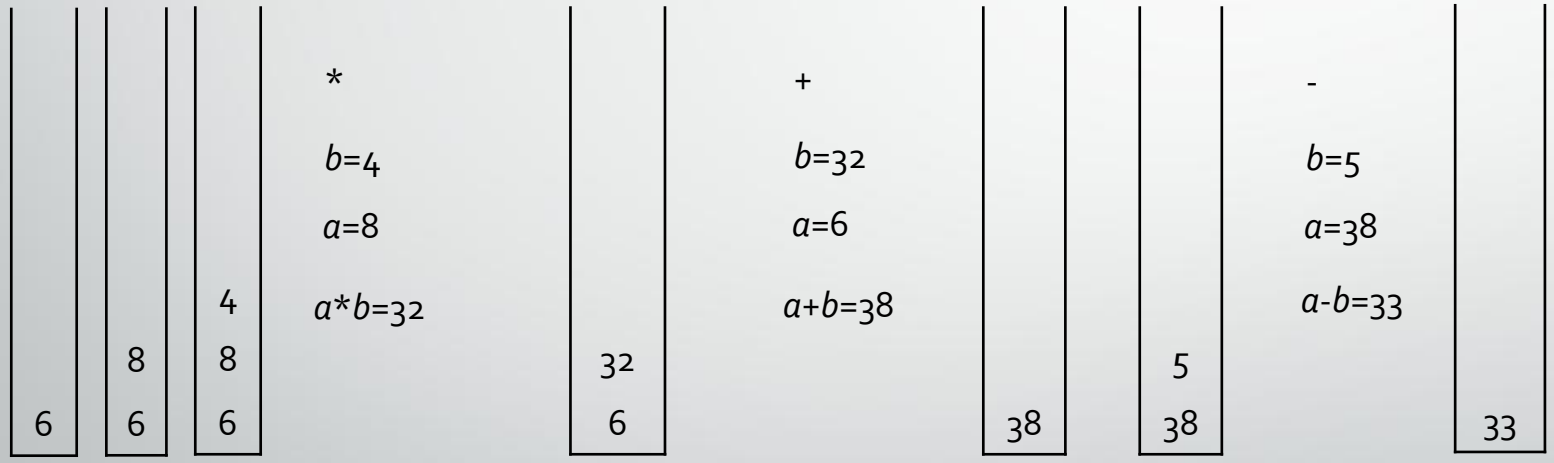
- Ha a soron következő karakter :
  - Operandus értékét: → verembe
  - *n* aritású operátor (op):
    1. kiveszünk *n* elemet
       - $x_n, \ldots, x_1$
    2. Visszatesszük: op($x_n, \ldots, x_1$) értékét
- Végül: a veremben csak a kifejezés értéke lesz

Lengyel forma kiértékelésének lejátszása*

- Infix alak:

6 8 4 * + 5 -

- Verem:



- Végeredmény:

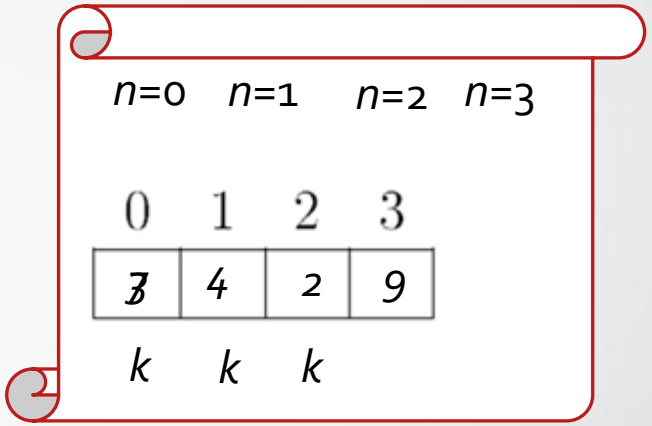
33

Sor (Queue) adattípus: FIFO (First-In First-Out)

Queue
$-\ Z : \mathcal{T}[]$ // $\mathcal{T}$ is some known type ; $Z.length$ is the physical $-$ constant $m0 : \mathbb{N}_+ := 16$ // length of the queue, its default is m0. $-\ n : \mathbb{N}$ // $n \in 0..Z.length$ is the actual length of the queue $-\ k : \mathbb{N}$ // $k \in 0..(Z.length - 1)$: the starting position of the queue in Z
$+$ Queue$(m : \mathbb{N}_+ := m0)\{\ Z := \textbf{new}\ \mathcal{T}[m]\ ;\ n := 0\ ;\ k := 0\ \}$ // create an empty queue $+$ add$(x : \mathcal{T})$ // join x to the end of the queue $+$ rem$() : \mathcal{T}$ // remove and return the first element of the queue $+$ first$() : \mathcal{T}$ // return the first element of the queue $+$ length$() : \mathbb{N}$ $\{\textbf{return}\ n\}$ $+$ isEmpty$() : \mathbb{B}$ $\{\textbf{return}\ n = 0\}$ $+ \sim$ Queue() { **delete** Z } $+$ setEmpty() $\{n := 0\}$ // reinitialize the queue

Sor példa

- Queue (4)
- add(3)
- add(4)
- rem(): 3
- rem(): 4
- add(2)
- add(9)
- add(7)



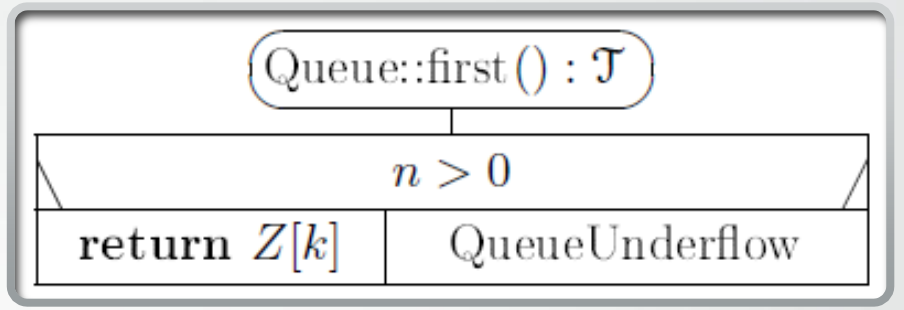
Gyakorlati alkalmazása

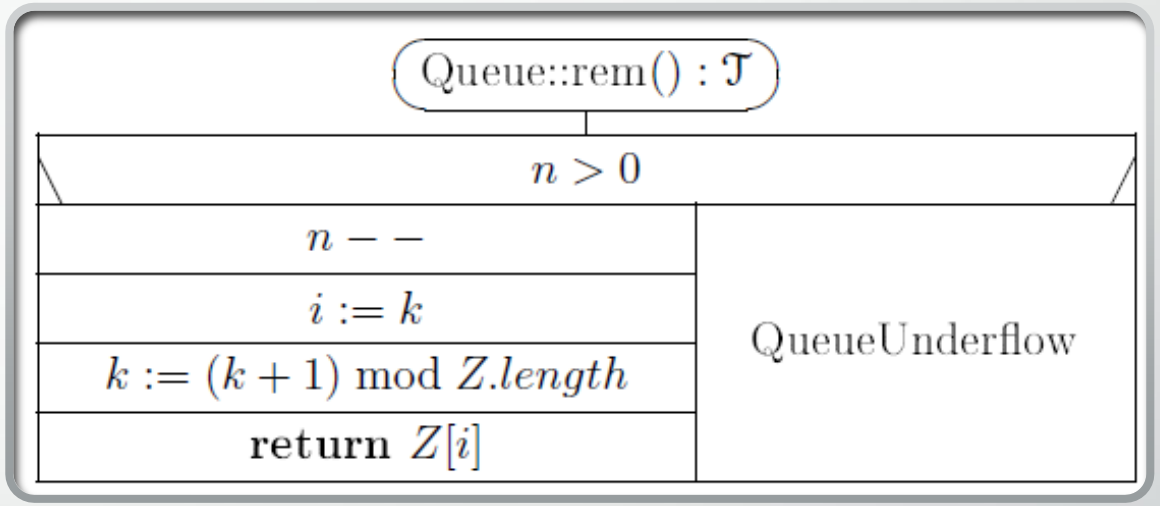
- IO pufferek
- Ügyfélszolgálati programok (üzletek, bankok)
- Folyamatütemezés
- Fák szintenkénti bejárása
- Szélességi keresés gráfokon
- Legrövidebb út keresése a legáltalánosabb esetben

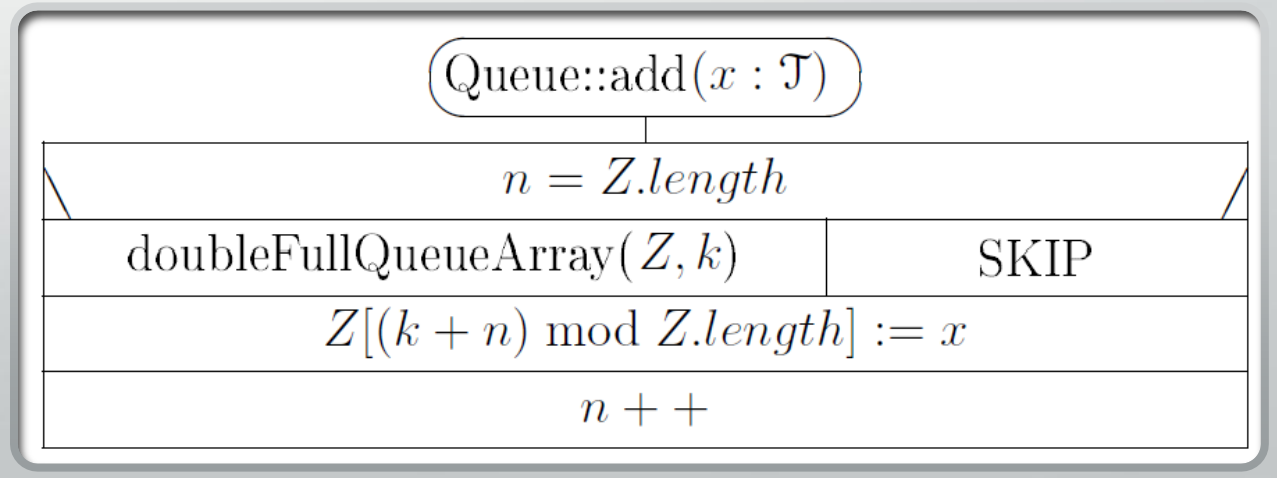
Sor műveletek

- Műveletidő: Θ(1)
- Add művelet
  - *mT(n)* ∈ Θ(1)
  - *MT(n)* ∈ Θ(n)
  - *AT(n)* ∈ Θ(1)

- doubleFullQueueArray(Z, k)
  - ha már tele van a tömb
    - cserélje le nagyobbra (2x)







Lineáris adatszerkezetek

Tömbök vs. Listák

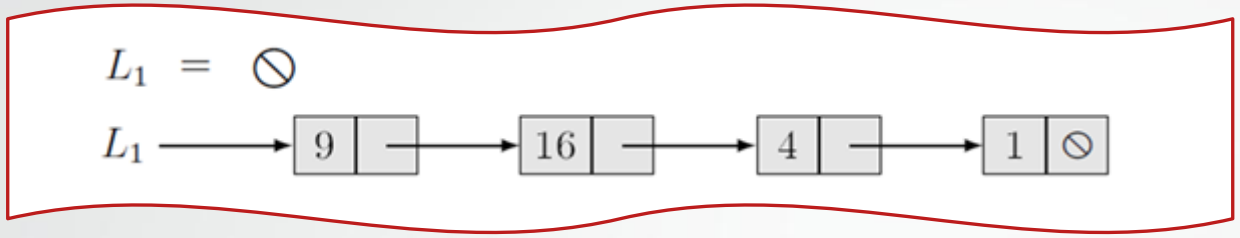
MT(beszúrás / törlés adott pozícióra):	$\Theta(n)$	$\Theta(1)$
MT(egy elemének elérése):	$\Theta(1)$	$\Theta(n)$

Láncolt listák (Linked Lists) fajtái

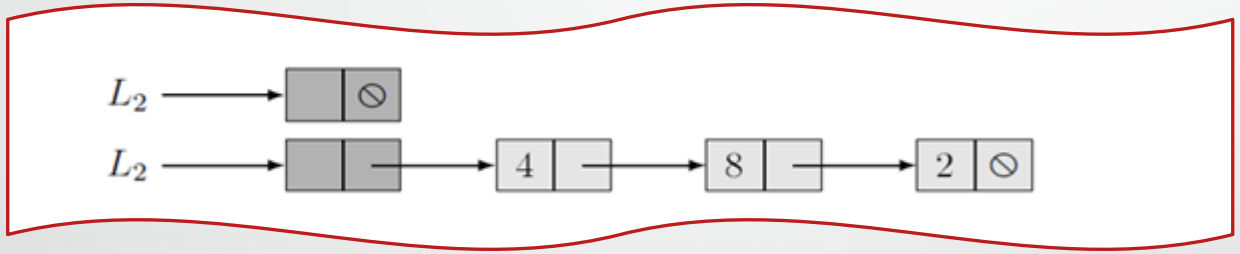
- Egyirányú – kétirányú
  - Mozgás: csak a lista elejétől vs vissza felé is (több memória, utasítás)
- Fejelemes- fejelem nélküli

Egyirányú listák (one-way or singly linked lists)

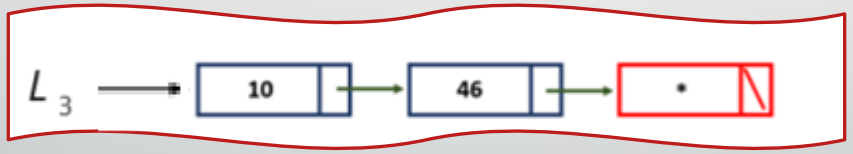
- Egyszerű egyirányú listák (*S1L* = Simple One-way List)



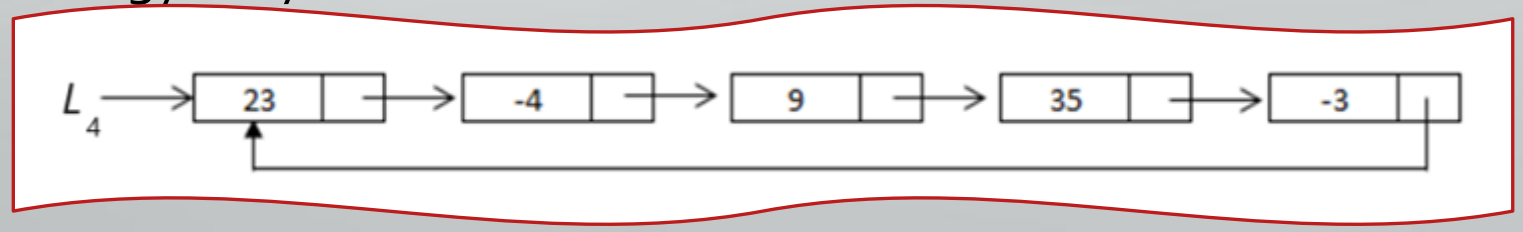
- Fejelemes egyirányú listák (*H1L* = One-way List with Header node)



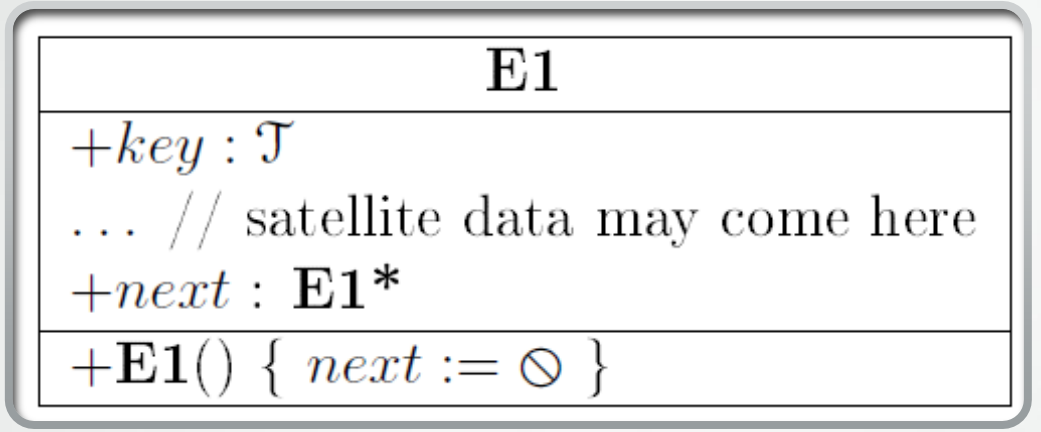
- Végelemes egyirányú listák



- Ciklikus egyirányú listák



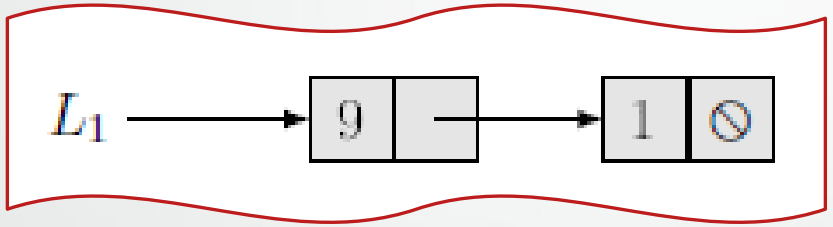
Egyirányú listák elemeinek osztálya



- *p* : E1* - pointer:
  - E1 típusú objektum címét tartalmazhatja (vagy ⊘)
  - A mutatott objektum: **p*
    - Mezői (adattagjai):
      - (**p*).*key* / *p->key*
      - (**p*).*next* / *p->next*

Egyszerű egyirányú listák (S1L = Simple 1-way List)

- L_1->*key*=9; L_1->*next*->*key*=1; L_1->*next*->*next*= ⦸

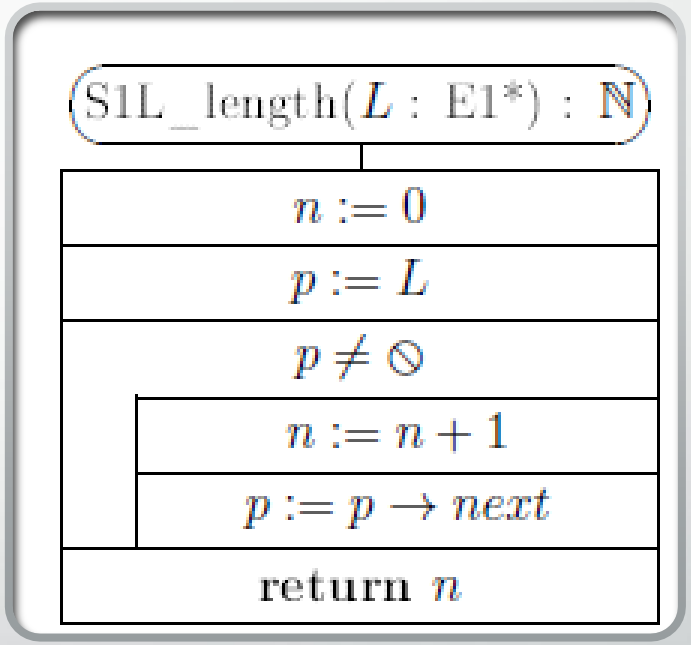


- $p = ⦸$
  - **p* hibás
  - *p*->*next*; *p*->*key* hibásak
  - futási hiba
- Ha *p* pointernek nem adunk értéket
  - **p*; *p*->*next*; *p*->*key* definiálatlanok

Példa: egyszerű egyirányú lista (S1L) hossza

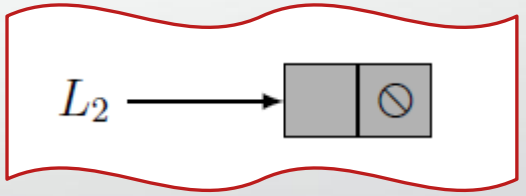
- S1L_length(*L*:E1*):ℕ
  - *L* paraméter
    - absztrakt (logikai) szinten: lista
    - *L*:E1*: a függvény által elvégzett számítás helyességének *előfeltétele*
    - konkrét (fizikai) szinten: memóriacím, ami a listát azonosítja
- Műveletigény:
  - *n:* az *L* lista hossza
  - a ciklus *n* iterációt végez

➢ $T_{S1L\_length}(n) \in \Theta(n)$



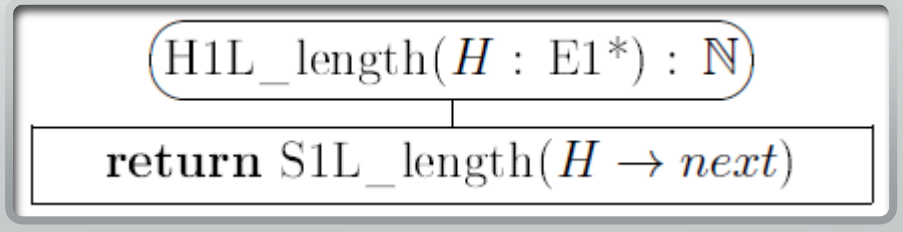
Fejelemes listák (H1L = Header node + 1-way List)

- Mindig tartalmaznak egy fejelemet
  - A fejelemére mutató pointer azonosítja
  - Key mező: definiálatlan
  - Next pointere: a H1L-nek megfelelő S1L-t azonosítja
- Üres H1L:
  - van fejeleme
    - next pointere ⦸
  - (L_2->*key*: definiálatlan; L_2->*next*= ⦸)



- H1L_length(*H*:E1*):ℕ

$T_{H1L\_length}(n) \in \Theta(n)$



Egyirányú listák alapműveletei

- Egyirányú listák kezelése
  - Listaelemek megfelelő átláncolása
    - Kerüljük a felesleges adatmozgatást
    - Járulékos adatok
- Listába szúrás
  - ∗p eleme után fűzi a ∗q objektumot
  - végrehajtása előtt ∗q nincs listába fűzve
  - $T \in \Theta(1)$
- Kifűzés listából
  - EF: ∗p és ∗q valamely egyirányú lista egymás utáni elemei
    - $p \rightarrow next = q \neq \oslash$
  - $T \in \Theta(1)$.
  - átfűzésnél a $q \rightarrow next := \oslash$ elhagyható

// Let $*p$ be followed by $*q$.
$q \rightarrow next := p \rightarrow next$
$p \rightarrow next := q$

// Provided that $*p$ is followed
// by $*q$, unlink $*q$.
$p \rightarrow next := q \rightarrow next$
$[q \rightarrow next := \oslash]$

Egyirányú listák további műveletei

- Fejelemes egyirányú listák (H1L)
  - Az alapműveletek segítségével minden összetett listamódosító művelet megadható
- Egyszerű egyirányú listák (S1L)
  1. Elem (*q) beszúrása a lista legelejére
  2. Első elem kifűzésére

1.

$// \text{ Insert } *q \text{ at the}$
$// \text{ front of list } L.$
$q \to next := L$
$L := q$

2.

$// \text{ Unlink the first element of list } L.$
$q := L$
$L := q \to next$
$[q \to next := \oslash]$

Műveletek egyirányú listákon: Keresés*

- Sikeres keresés:
  - *p*=Az első *k* kulcsú elem címe
- Sikertelen keresés:
  - *p*= ⦸

Műveletigény: **O(*n*)**

Keres_S1L(L:E1*; k:ℑ):E1*

```
p:=L
p≠⦸ és p->key ≠k
    p:=p->next
return p
```

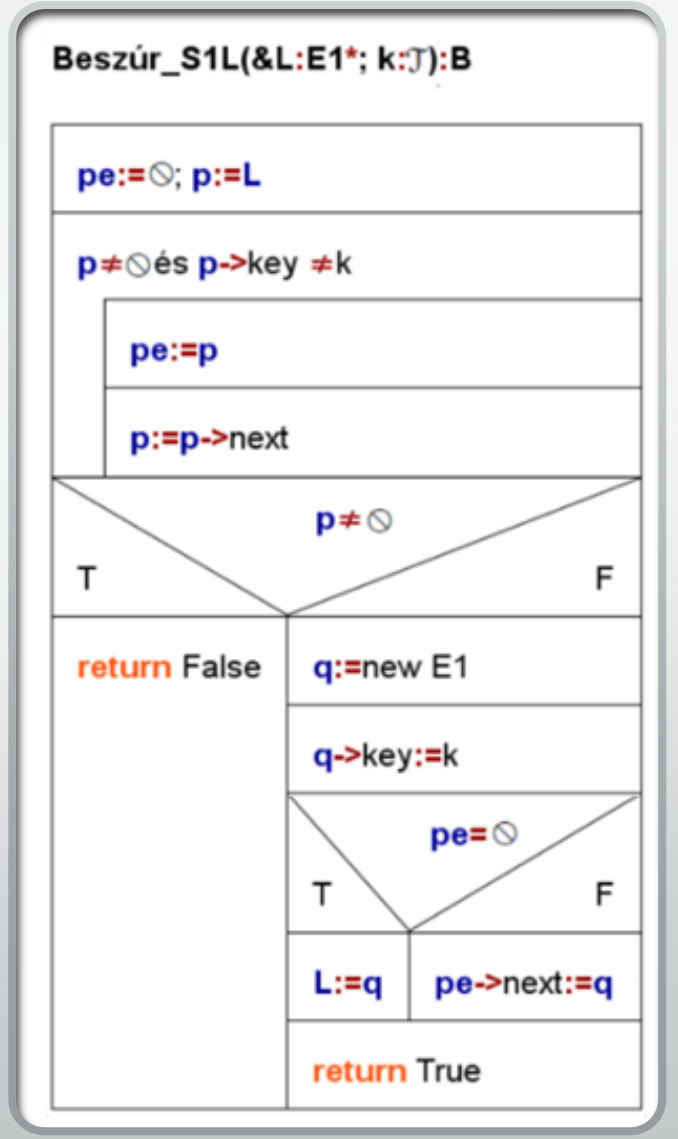
Keres_H1L(L:E1*; k:ℑ):E1*

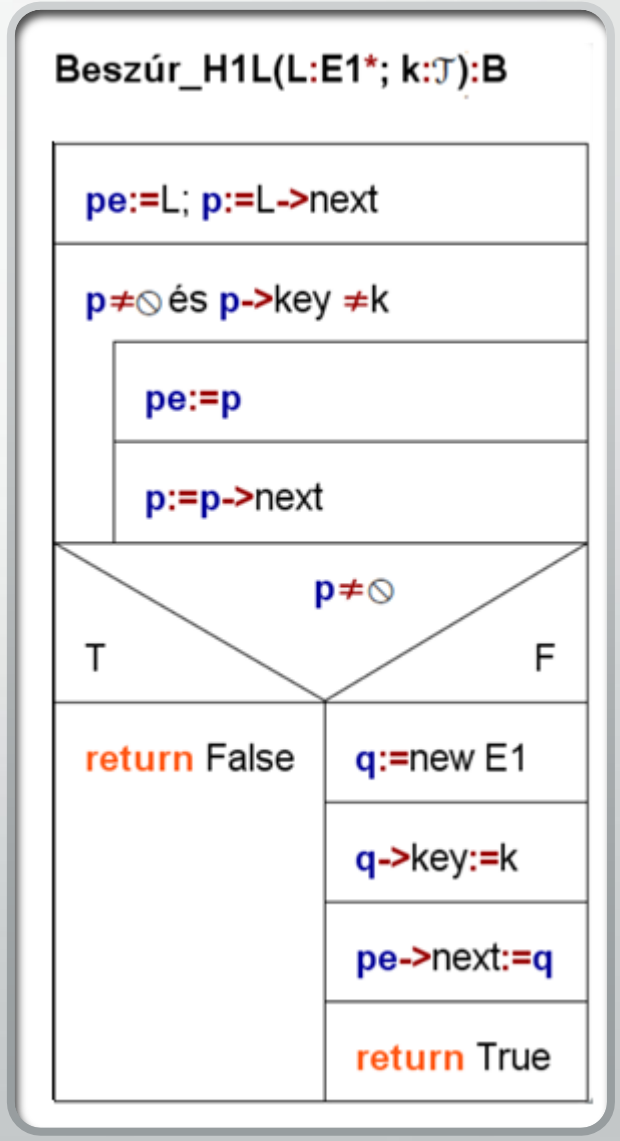
```
p:=L->next
p≠⦸ és p->key ≠k
    p:=p->next
return p
```

Műveletek egyirányú listákon: Beszúrás*

- Egyedi kulcsok
  - Ismétlődés nem megengedett
- Két futó pointer
  - *pe*: *p* előtti elem
- Sikeres keresés:
  - Nincs beszúrás ->False
- Sikertelen keresés:
  - Beszúrás utolsó elemnek ->True

Műveletigény: **O(*n*)**

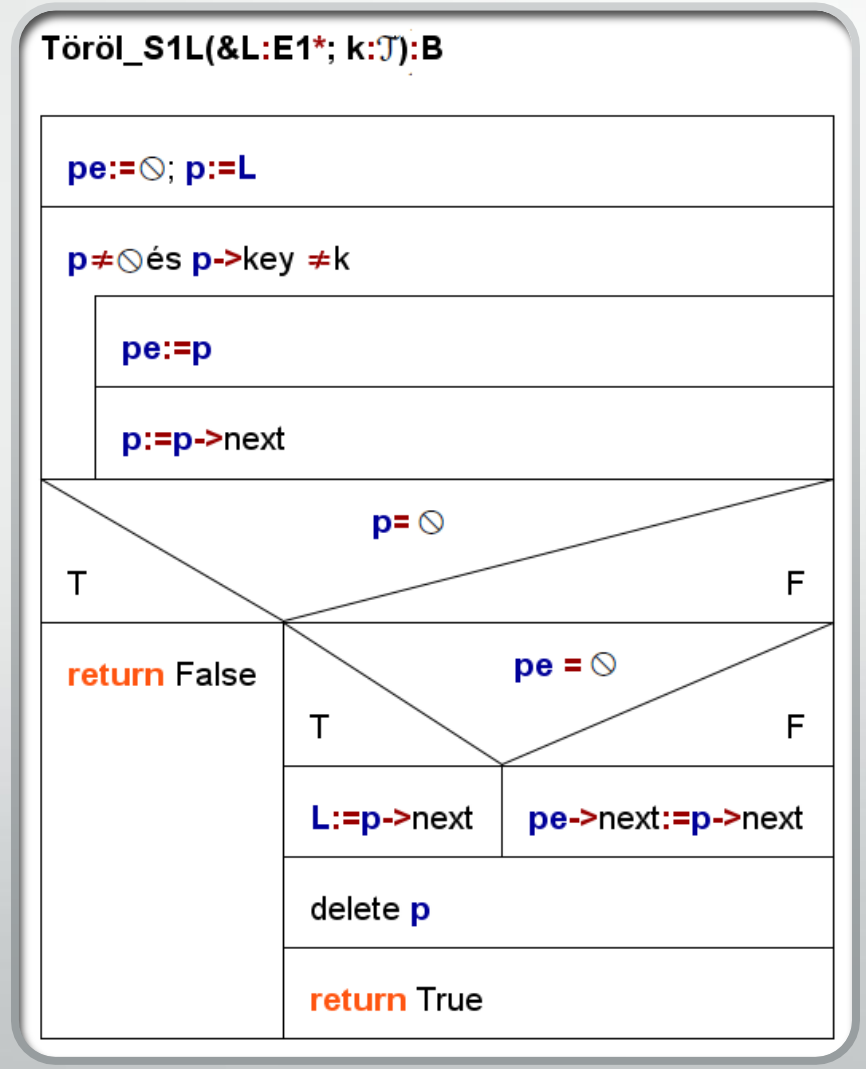


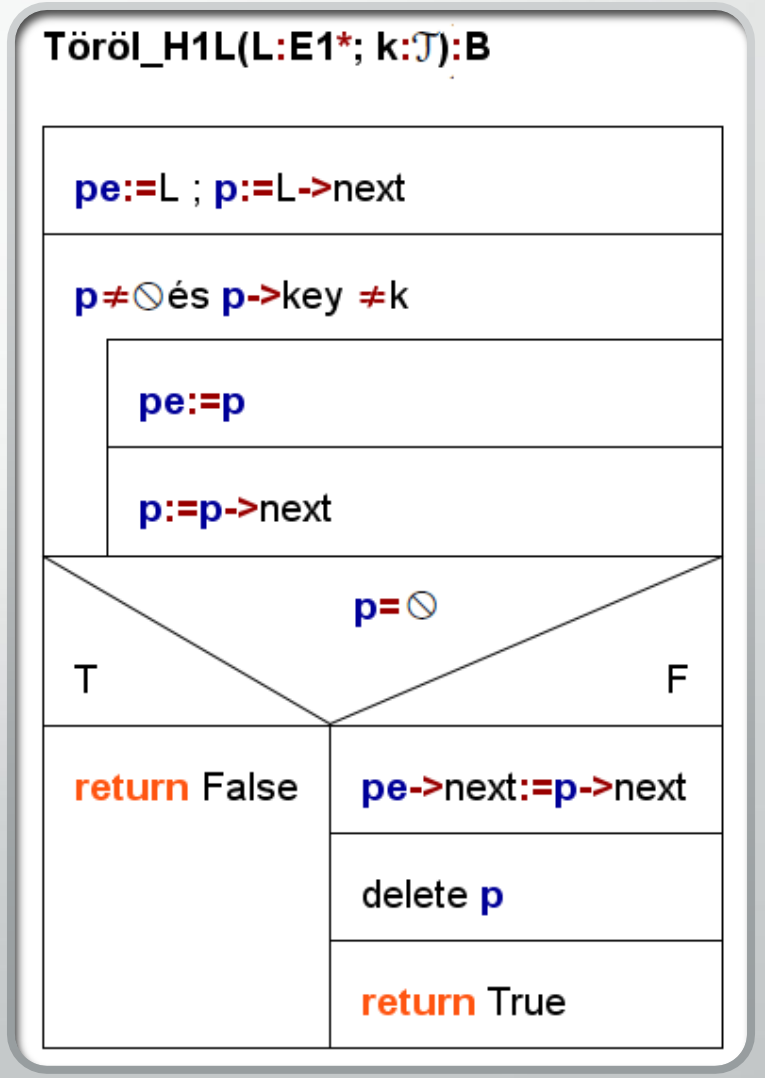


Műveletek egyirányú listákon: Törlés*

- Két futó pointer
  - *pe*: *p* előtti elem
- Sikertelen keresés:
  - Nincs törlés ->False
- Sikeres keresés:
  - Kifűzés és törlés ->True

Műveletigény: **O(*n*)**





S1L és H1L összehasonlítása

S1L

- Sok rövid lista
  - Szignifikáns a tárigények különbsége
  - Fejelemek allokálása és deallokálása a futási időt jelentősen megnövelheti
  - (pl. hasító táblák, összefésüléses rendezés)
- Egy (rész)feladatban a listát mindig csak a legelején (~ veremszerűen) kell módosítani
- A lista első eleme biztosan a helyén marad

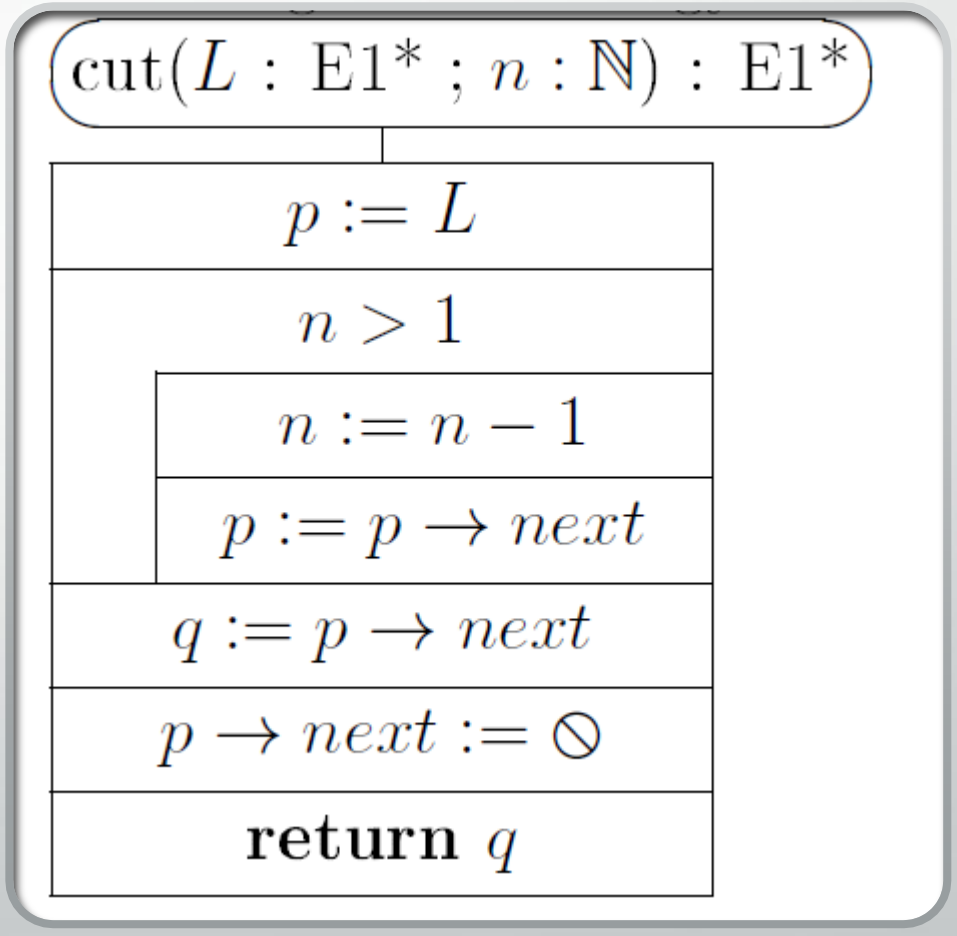
H1L

- Kevesebb esetszétválasztás
  - Mindig valami után kell beszúrni
  - Mindig valami mögül kell kifűzni.
- Eggyel több objektumot tartalmaz
  - Megnöveli a program tárigényét
- Őrszem (sentinel):
  - fejelem, végelem
- Végelem: pl. sorok

cut(L : E1* ; n : $\mathbb{N}$) : E1*

- *L* kettévágása (L: S1L)
  - Az első *n* elemét hagyja *L*-ben
  - Visszaadja a lista levágott maradékát azonosító pointert
- A listaelemek sorrendjét megtartja

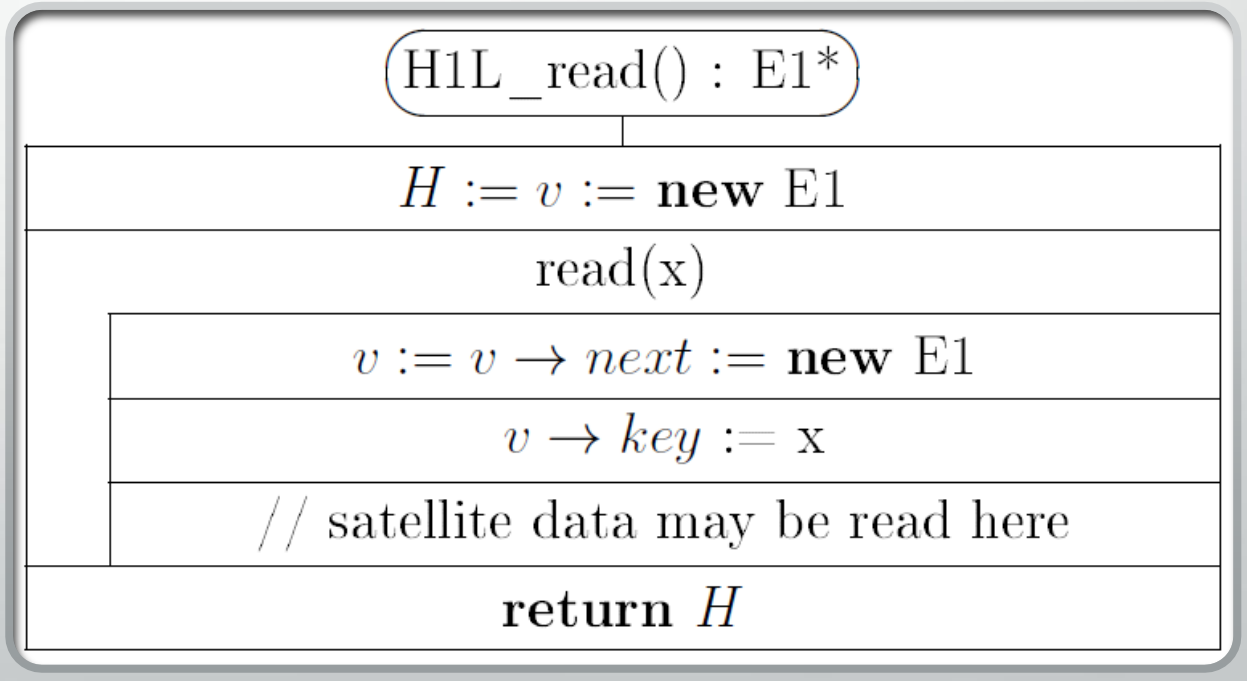
$T_{cut}(n) \in \Theta(n)$



H1L_read() : E1*

$T_{H1L\_read}(n) \in \Theta(n)$

- Feladata:
  - beolvas egy adatsort a kurrens inputról
  - bemenet sorrendje szerint egy H1L-t épít belőlük
  - visszaadja a fejeleme címét
- read(&x: $\mathcal{T}$): $\mathbb{B}$
  - következő adat beolvasása x-be
  - igaz: ha a beolvasás előtt még volt adat a bemeneten
  - hamis: különben (x definiálatlan marad)



- Feladat: Valósítsuk meg a Queue osztályt egyirányú, végelemes listával! //mo: 4. óra

Dinamikus memóriagazdálkodás

- objektumok dinamikus létrehozása: new $\mathcal{T}$
  - $\mathcal{T}$ típusú objektumot hoz létre, és visszaadja a címét
  - helyfoglalás a memóriában
- *p* mutató
  - a *p* := new T utasítás végrehajtása előtt is létezik
    - ➢ a mutató deklarációjának kiértékelése hozza létre

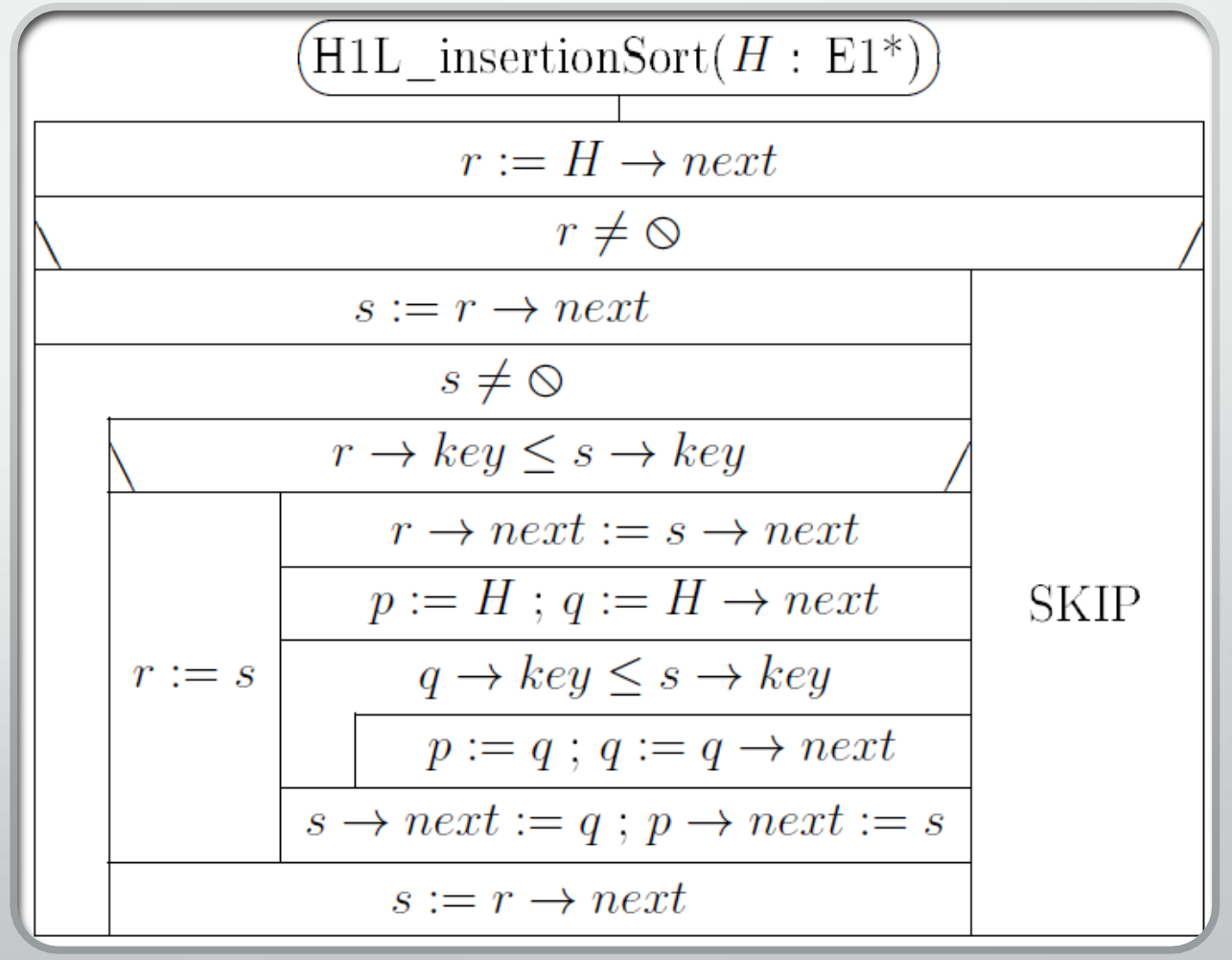
- objektumok dinamikus törlése: delete *p*
  - *p* mutató által hivatkozott objektumot törli
  - memória felszabadítása
- delete *p* végrehajtása után is létezik
  - egészen az őt (automatikusan) deklaráló eljárás vagy függvény végrehajtásának befejezéséig

- Az absztrakt programokban: new / delete utasítások műveletigénye:
  - Konstans értékeknek (Θ(1)-nek) vesszük
  - Valójában nem tudjuk, mennyi.
  - ➢ A lehető legkevesebbet használjuk a new és a delete utasításokat.

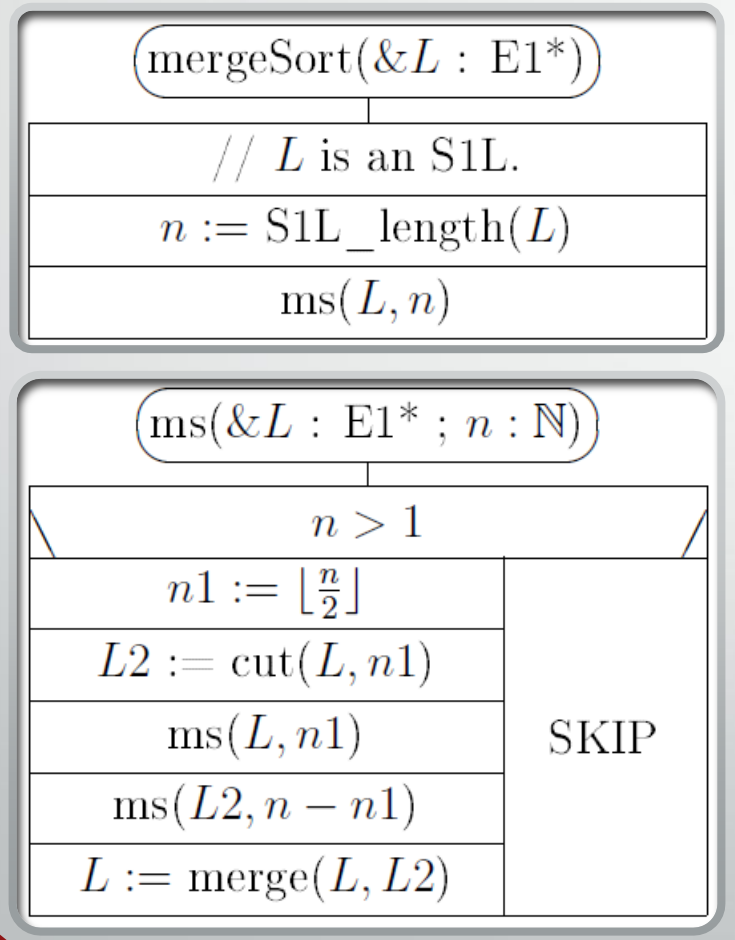
Beszúró rendezés H1L listákra

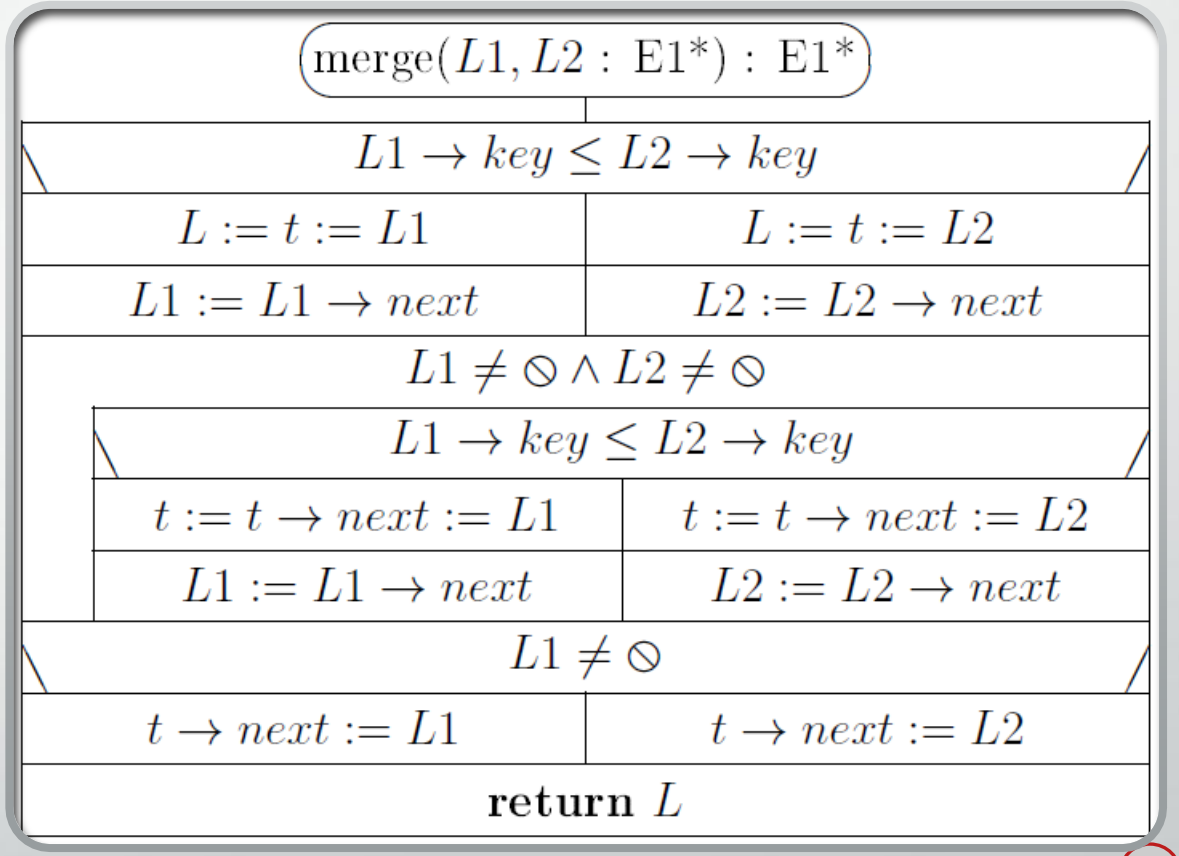
$$mT_{IS}(n) \in \Theta(n)$$

$$AT_{IS}(n), MT_{IS}(n) \in \Theta(n^2)$$



Összefésüléses rendezés S1L-re





$$mT_{MS}(n), MT_{MS}(n) \in \Theta(n \log n)$$

Ellenőrző kérdések

- Adja meg a Stack/Queue osztályok – tömbös reprezentációra alapozott – leírását, a metódusok struktogramjaival együtt! Mennyi a verem egyes műveleteinek a futási ideje? Miért?
- Adja meg a Stack/Queue osztályok – egyirányú, nem ciklikus, láncolt listás reprezentációra alapozott - leírását, a metódusok struktogramjaival együtt! Mennyi a verem egyes műveleteinek a futási ideje? Miért?
- Adja meg a Queue osztályok – egyirányú, ciklikus, fejelemes láncolt listás reprezentációra alapozott - leírását, a metódusok struktogramjaival együtt! Mennyi a verem egyes műveleteinek a futási ideje? Miért?
- Melyek a tanult lineáris adatszerkezetek? Mennyi ezeknek a beszúrás, a törlés, illetve az *i*. elem elérésének a műveletigénye?
- Milyen típusú láncolt listát ismer? Mik az egyes típusok előnyei és hátrányai?

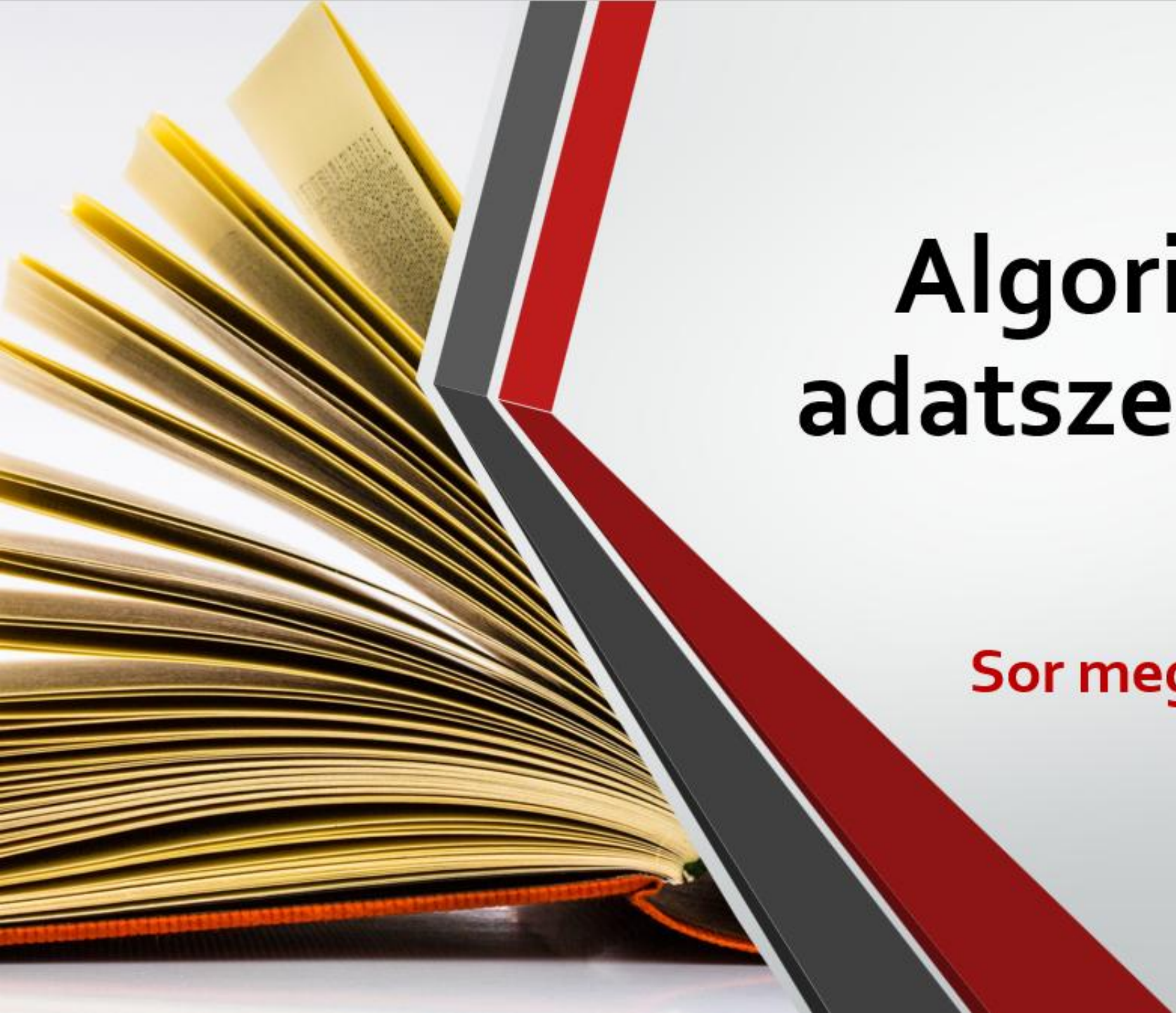
Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek I. előadásjegyzete alapján készült.

További felhasznált irodalom:

- *dr Póder Margit f. docens Rendszer- és Szoftvertechnológia Tanszék*. (n.d.). https://slideplayer.hu/slide/2140789/ (2025. 01. 30.)
- *oktatas:programozas:algoritmusok:lengyelforma [szit]*. (n.d.). https://szit.hu/doku.php?id=oktatas:programozas:algoritmusok:lengyelforma (2025. 01. 30.)
- Prezi, M. W. O. (n.d.). *Lengyel-Forma*. prezi.com. https://prezi.com/p/150uth61cj1u/lengyel-forma/ (2025. 01. 30.)
- *Lengyel forma*. (n.d.). https://people.inf.elte.hu/veanna/alg1/segedanyagok/LengyelForma/index.htm (2025. 01. 30.)
- https://ntibi.web.elte.hu/programozas/elmelet/Lengyel-forma.pdf (2025. 01. 30.)



Algoritmusok és adatszerkezetek I.

3. Előadás

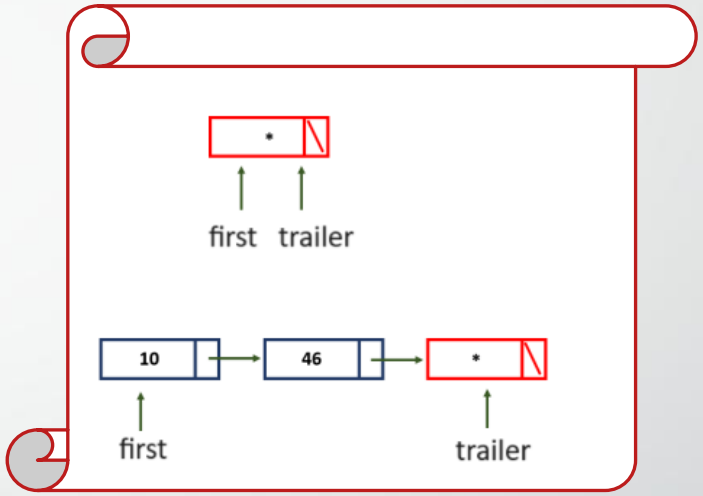
Sor megvalósítása listával

Kétirányú listák

Tartalom

Sor megvalósítása végelemes listával

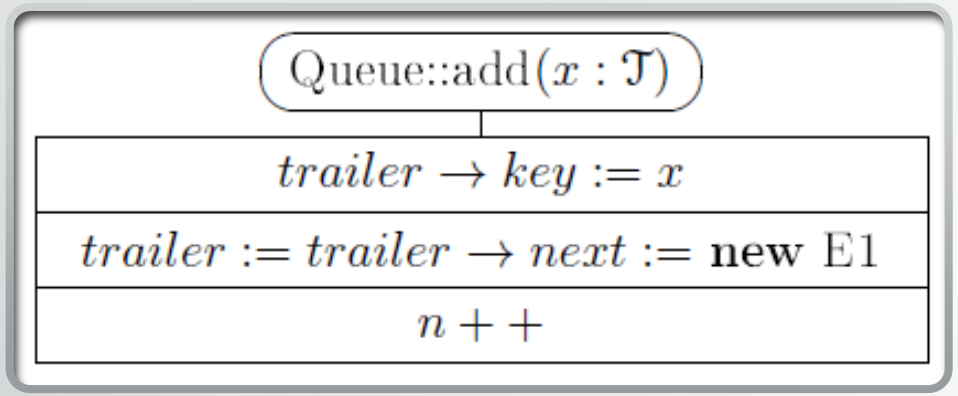
- Valósítsuk meg a Queue osztályt egyirányú, végelemes listával!
- A listára két külső pointer mutat
  - *first:* az első elemre
  - *trailer:* a (nemdefiniált kulcsú) végelemére
- A sor műveleteinek műveletigénye
  - setEmpty() és a destruktor :Θ(*n*)
  - többi: Θ(1)
    - *n* a kiürítendő sor hossza
    - *new* és a *delete* utasítások: Θ(1)
- Tfh. a new utasításokhoz
  - elegendő a memória

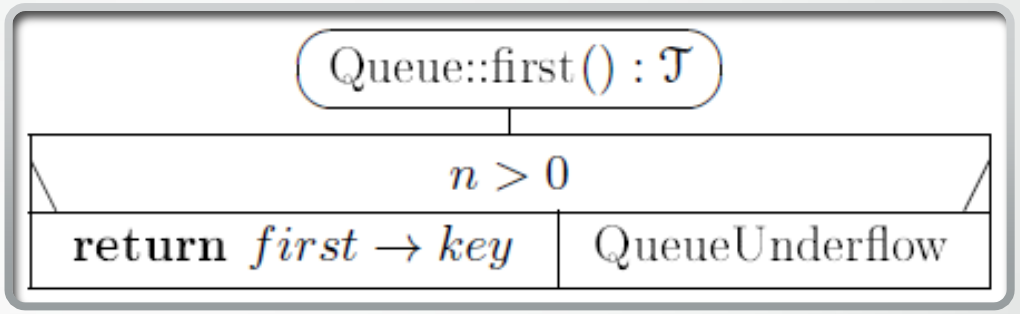


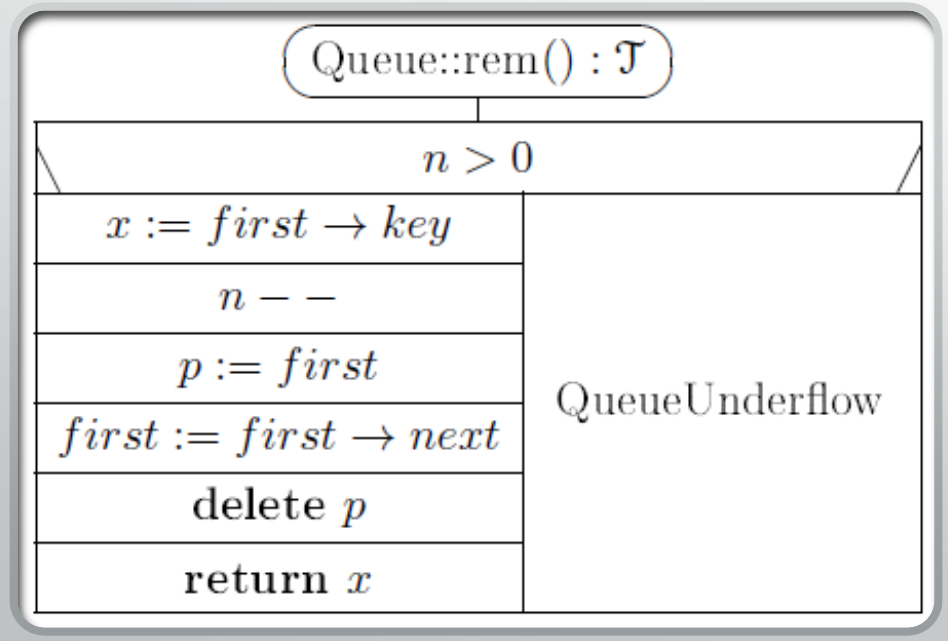
Queue
$-$ $first, trailer$: E1* // a one-way list with trailer represents the queue $-$ $n : \mathbb{N}$ // n is the actual length of the queue

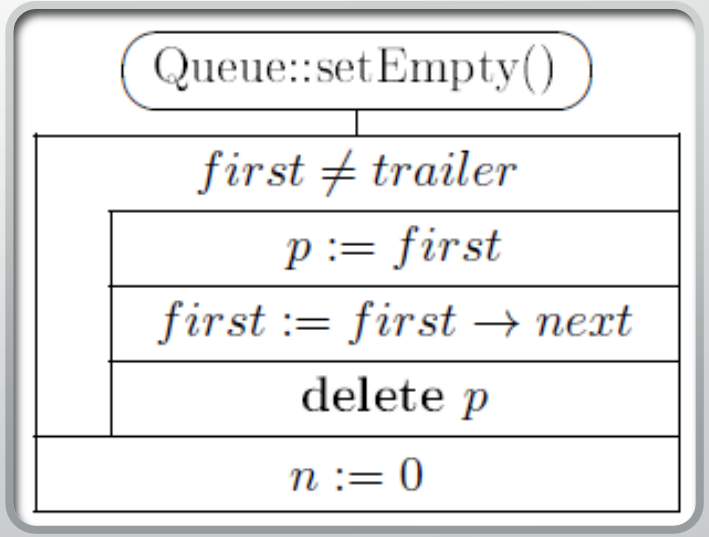
+ Queue(){ $first := trailer :=$ **new** E1 ; $n := 0$ } // create an empty queue
+ add($x : \mathcal{T}$) // join x to the end of the queue
+ rem() : $\mathcal{T}$ // remove and return the first element of the queue
+ first() : $\mathcal{T}$ // return the first element of the queue
+ length() : $\mathbb{N}$ {**return** n}
+ isEmpty() : $\mathbb{B}$ {**return** $n = 0$}
+ setEmpty() // reinitialize the queue
+ $\sim$ Queue() { setEmpty() ; **delete** $trailer$ }

Queue műveletek



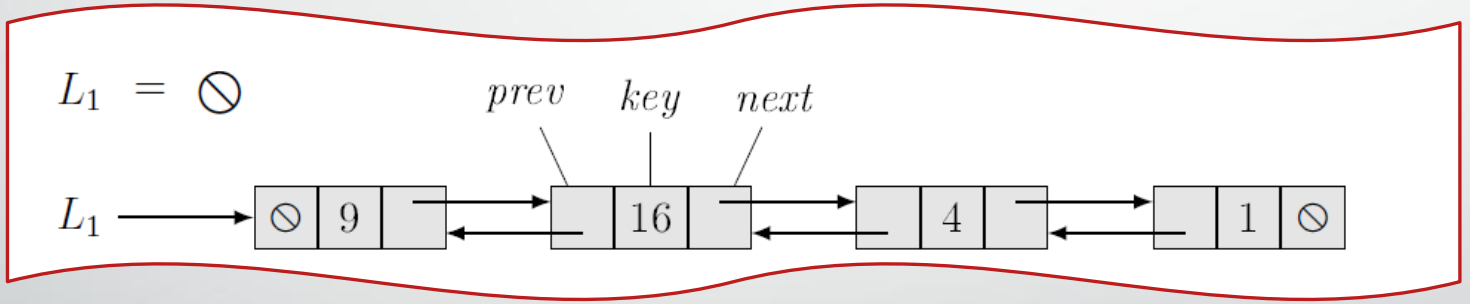




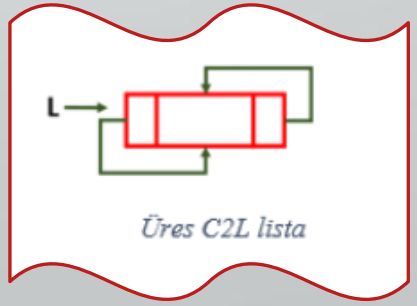


Kétirányú listák (two-way or doubly linked lists)

- Egyszerű kétirányú listák (S2L = Simple Two-way List)
  - a lista módosításakor különbözőképpen kell kezelni:
    - a lista első / utolsó / közbülső elemeit
  - Hasító táblánál használjuk



- Fejelemes ciklikus kétirányú listák (C2L = Cyclic Two-way List with header)
  - Gyakoribb
  - listamódosító műveletek:
    - egyszerűbbek és hatékonyabbak



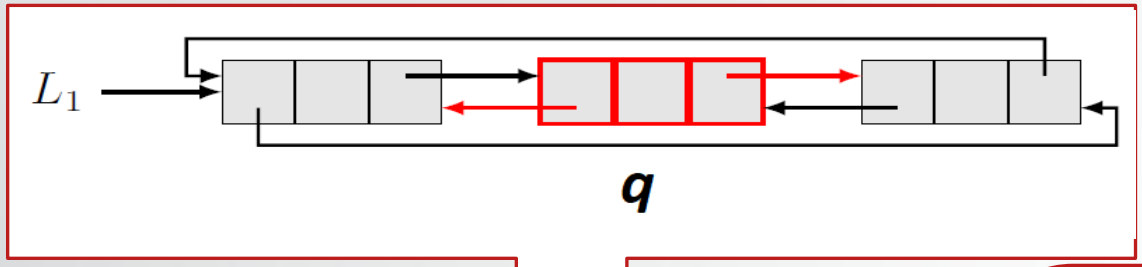
Ciklikus kétirányú listák (C2L) (Cyclic Two-way List)

- Fejelemes vs. fejelem nélküli C2L
  - Megegyeznek:
    - a listák elemeinek osztálya (E2)
    - alapvető listakezelő műveleteik (Θ(1))
  - Fejelem használatának előnye:
    - Nem kell külön kezelni:
      - az üres listába való beszúrást
      - az utolsó listaelem törlését
    - ➢ Listakezelés tovább egyszerűsödik

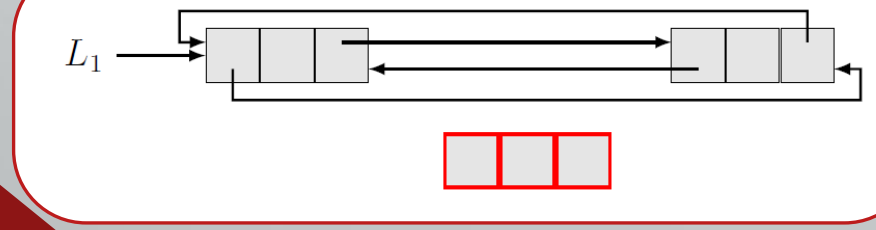
E2
$+prev, next : \mathbf{E2*}$ $+key : \mathcal{T}$
$+ \text{E2}() \{ prev := next := \mathbf{this} \}$

➢ C2L: Fejelemes

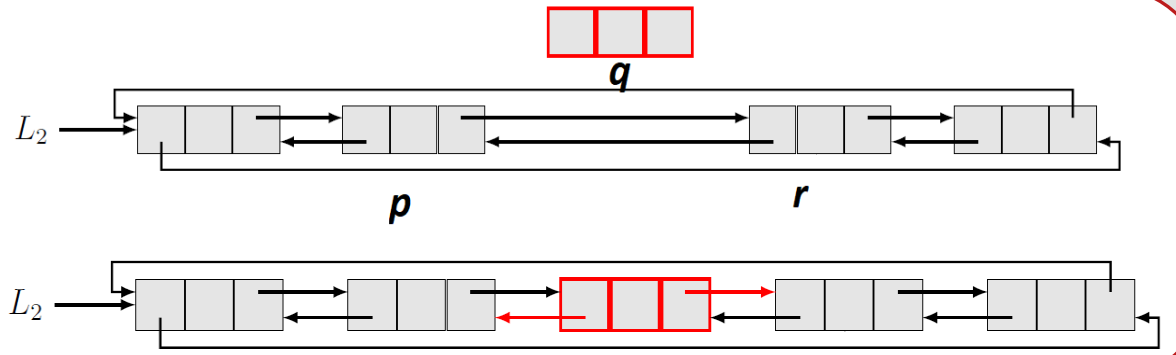
Listakezelő műveletek



unlink(q : E2*)
// remove $(*q)$
$p := q \to prev$; $r := q \to next$
$p \to next := r$; $r \to prev := p$
$q \to prev := q \to next := q$



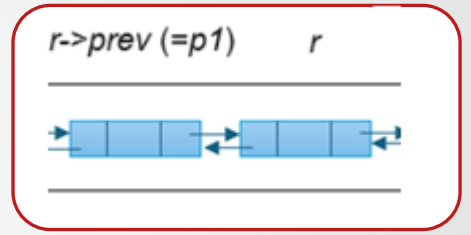
precede(q, r : E2*)
// $(*q)$ will precede $(*r)$
$p := r \to prev$
$q \to prev := p$; $q \to next := r$
$p \to next := r \to prev := q$



follow(p, q : E2*)
// $(*q)$ will follow $(*p)$
$r := p \to next$
$q \to prev := p$; $q \to next := r$
$p \to next := r \to prev := q$

Példák C2L listákra: splice listaművelet

- Előfeltétel:
  - $*p$ és $*q$ ugyanannak a C2L-nek az elemei
  - $*p$ után jön valahol $*q$ ($p = q$ is lehet)
  - a C2L $[p, \ldots, q]$ szakasza nem tartalmaz:
    - sem fejelemet
    - sem a $*r$ elemet
  - $*r$ elem: ennek, vagy egy másik C2L-nek eleme
- **splice(p, q, r)** elemi listaművelet
  - A fenti előfeltétellel
  - **C2L egy adott $[p, \ldots, q]$ szakaszát eltávolítja**
  - **majd** az eltávolított szakaszt egy **$*r$ eleme elé fűzi**



$$\text{splice}(p, q, r : \text{E2}^*)$$

$p1 := p \to prev \ ; \ q2 := q \to next$
$p1 \to next := q2 \ ; \ q2 \to prev := p1$
$p1 := r \to prev$
$p \to prev := p1 \ ; \ q \to next := r$
$p1 \to next := p \ ; \ r \to prev := q$

Műveletigény: Θ(1)

Példák C2L listákra: append eljárás

- Előfeltétel:
  - $*p$ és $*q$ ugyanannak a C2L-nek az elemei
  - $*p$ után jön valahol $*q$ ($p = q$ is lehet)
  - $[p, \ldots, q]$ nem tartalmazza sem a fejelemet, sem a $*r$ elemet
  - $*r$ elem: ennek, vagy egy másik C2L-nek is lehet eleme
- **append(*L*,*H*)** eljárás
  - **átfűzi az *L* C2L végére a *H* C2L elemeit**
  - eredeti sorrendben
  - használja a slice(*p*,*q*,*r*)

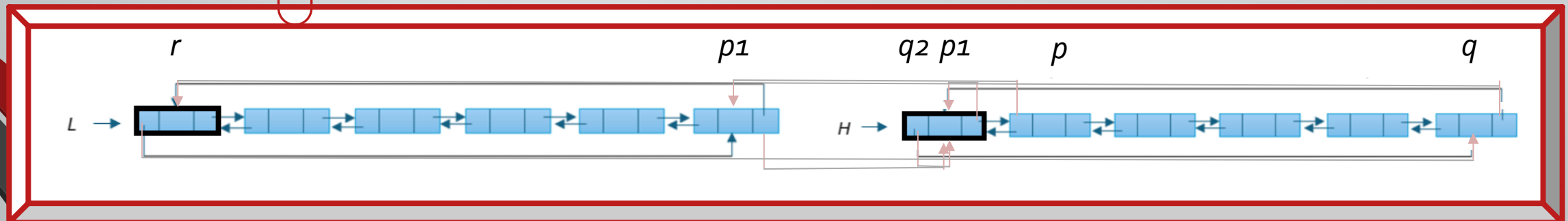
Műveletigény: Θ(1)

$$\text{splice}(p, q, r : \text{E2}^*)$$

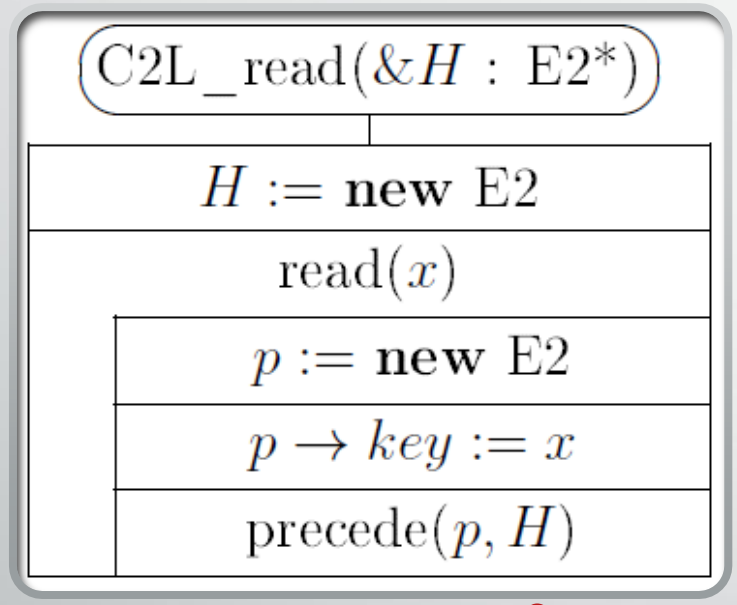
$p1 := p \to prev \ ; \ q2 := q \to next$
$p1 \to next := q2 \ ; \ q2 \to prev := p1$
$p1 := r \to prev$
$p \to prev := p1 \ ; \ q \to next := r$
$p1 \to next := p \ ; \ r \to prev := q$

$$\text{append}(\ L, H : \text{E2}^*\ )$$

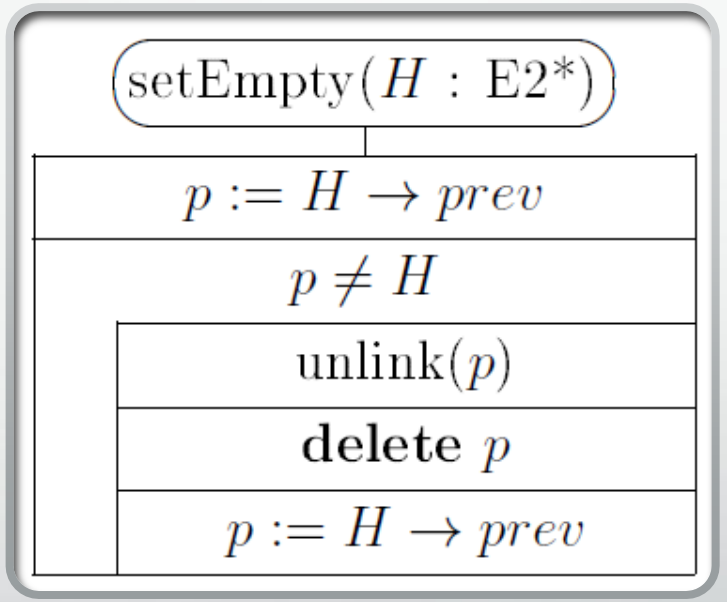
$H \to next \neq H$	
$\text{splice}(H \to next, H \to prev, L)$	SKIP



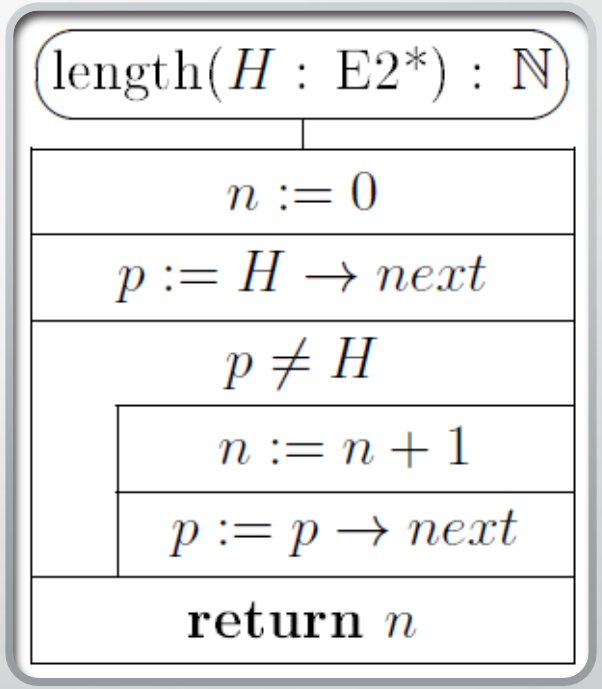
Példaprogramok C2L listákra



$T_{C2L\_read}(n)$



$T_{setEmpty}(n) \in \Theta(n)$



$T_{lenght}(n) \in \Theta(n)$

Példaprogramok C2L listákra

insertionSort(H : E2*)

$r := H \to \text{next}\ ;\ s := r \to \text{next}$

$s \neq H$

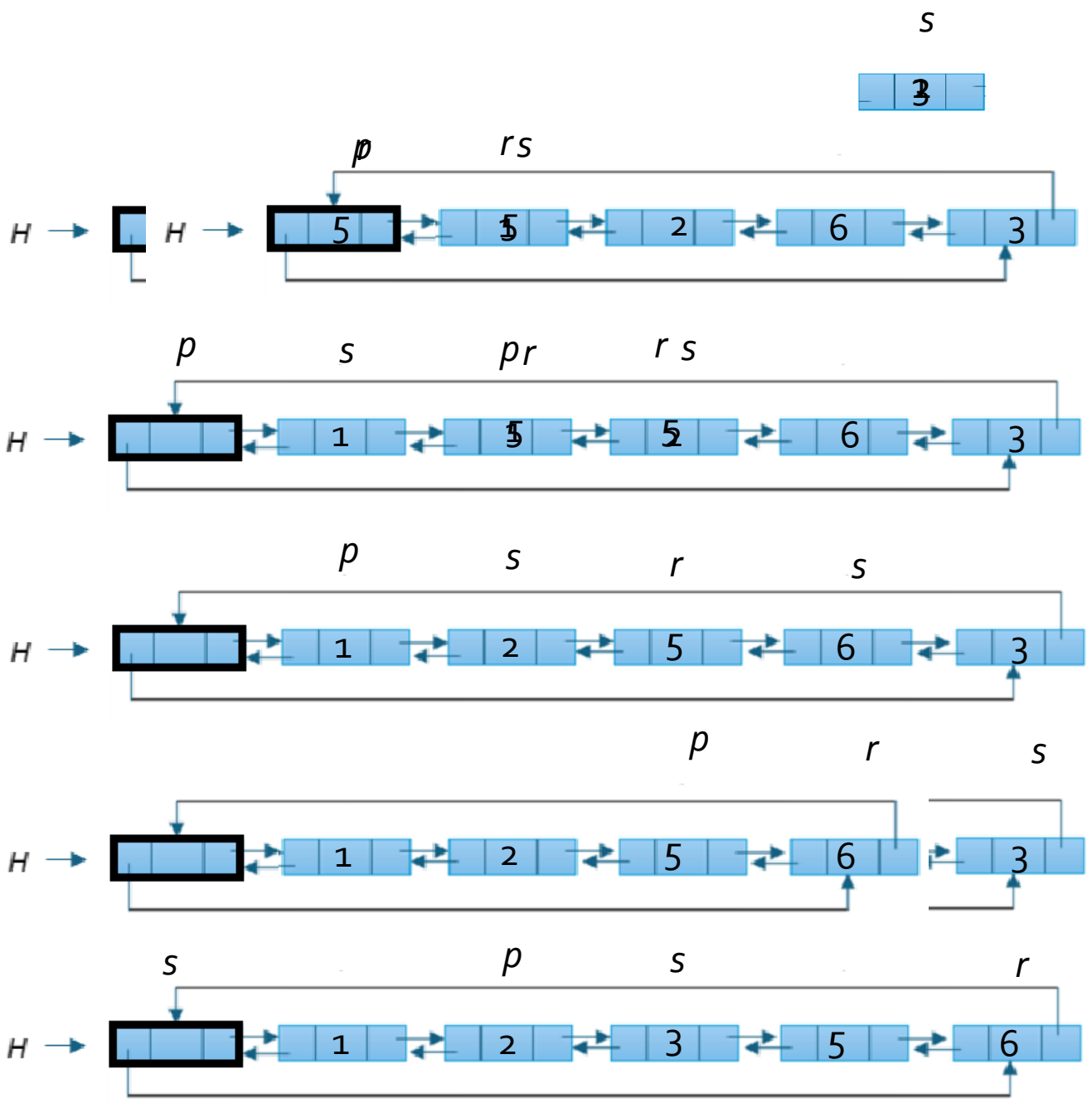
$r \to key \leq s \to key$

$r := s$	
	$\text{unlink}(s)$
	$p := r \to \text{prev}$
	$p \neq H \wedge p \to key > s \to key$
	$p := p \to \text{prev}$
	$\text{follow}(p, s)$

$s := r \to \text{next}$

$$mT_{IS}(n) \in \Theta(n)$$

$$AT_{IS}(n), MT_{IS}(n) \in \Theta(n^2)$$

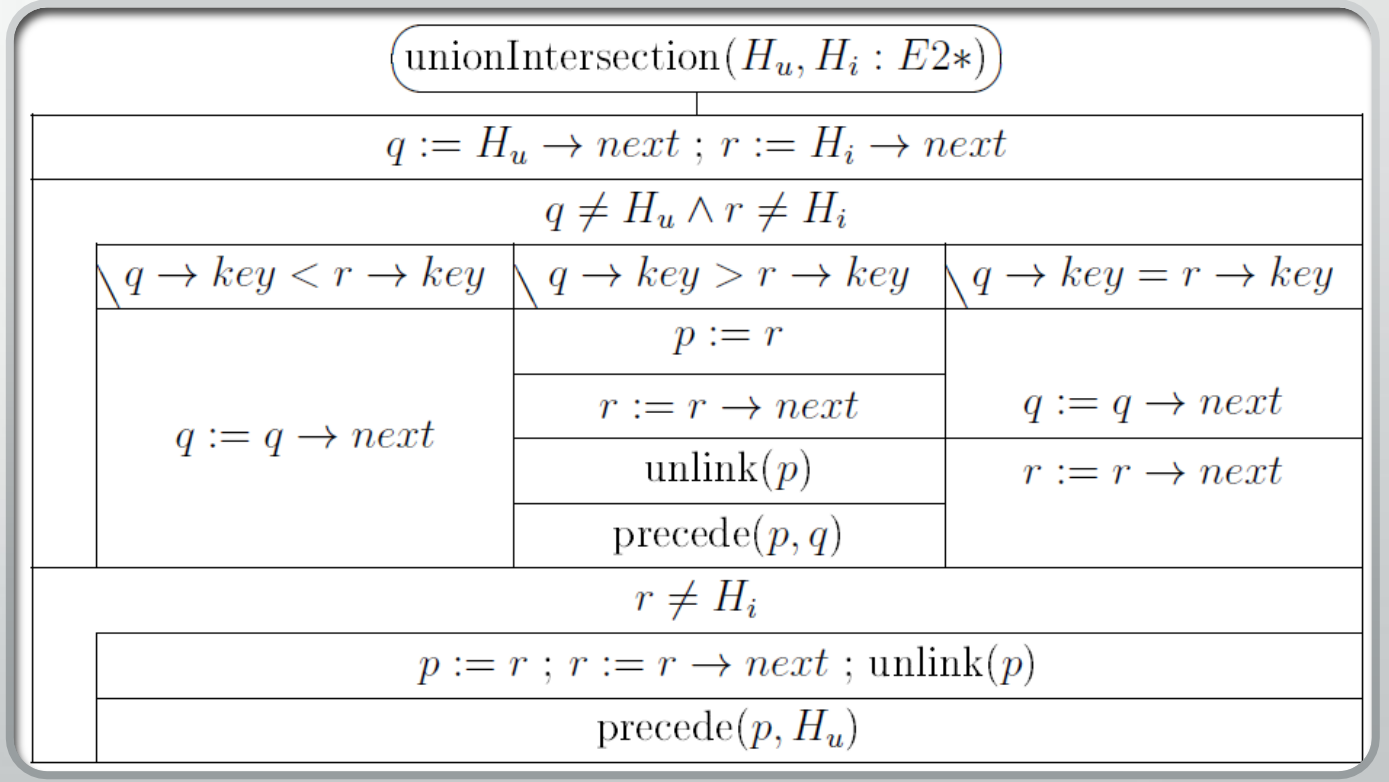


Halmazműveletek szig. mon. növő C2L listákra

- **unionIntersection (H_u, H_i : E2*)**
  - H_u és H_i : szig. mon. növő rendezett C2L-ek
  - H_i megfelelő elemeit átfűzi a H_u listába

- Eredmény:
  - **H_u listában:** az eredeti listák ~ halmazok **uniója**
  - **H_i listában: a metszetük** marad

- Feltételek:
  - Csak listaelemek átfűzése
    - Nincs allokálás/ deallokálás
  - **$MT(n_u, n_i) \in \Theta(n_u + n_i)$**
    - H_u C2L hossza: n_u
    - H_i C2L hossza: n_i
  - Minkét lista marad
    - Szigorúan monoton növekvően rendezett C2L



Ellenőrző kérdések

- Adja meg a Kétirányú ciklikus listák műveleteit!
- Írja meg a tanult rendező algoritmusokat C2L listára!
- Adott két szigorúan monoton növekedően rendezett C2L. Írjon eljárást, mely a megfelelő elemek átfűzésével előállítja a két lista mint halmazok unióját az első listába! A másik lista lebontható!
  - Írjon eljárást, mely az előző feladathoz hasonlóan a következő halmazműveleteket oldja meg:
    - Két halmaz metszetét
    - Két halmaz különbségét
    - Két halmaz szimmetrikus differenciáját

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek I. előadásjegyzete alapján készült.

Algoritmusok és adatszerkezetek I.
4. Előadás

Gyorsrendezés

Tartalom

Gyorsrendezés - Quicksort

- Jelenleg nem ismerünk elfogadható hatékonyságú stabil Quicksort tömbrendezőt

Ha nagy hatékonyságú stabil rendezés
- Merge sort és továbbfejlesztései

Ha a stabilitás nem szükséges
- Quicksort és továbbfejlesztései

nagyméretű tömbökre ennek a legjobb az átlagos műveletigénye

A futtatási tapasztalatok:

- láncolt listákra általában jobb választás a merge sort

Gyorsrendezés lépései

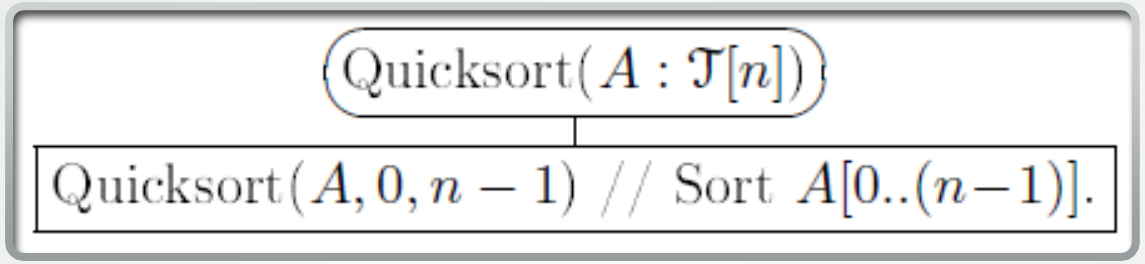
„Oszd meg és uralkodj” elv

1. **Tengely (pivot) kiválasztása:** rendezendő (rész)tömb egy tetszőleges eleme
2. **Részekre bontás: particionálás (partitioning)**: Tömb átrendezése
   - minden, a tengelynél kisebb elem -> a tengely elé
   - a nagyobbak -> utána
   - A tengellyel egyenlők -> bármelyik részbe

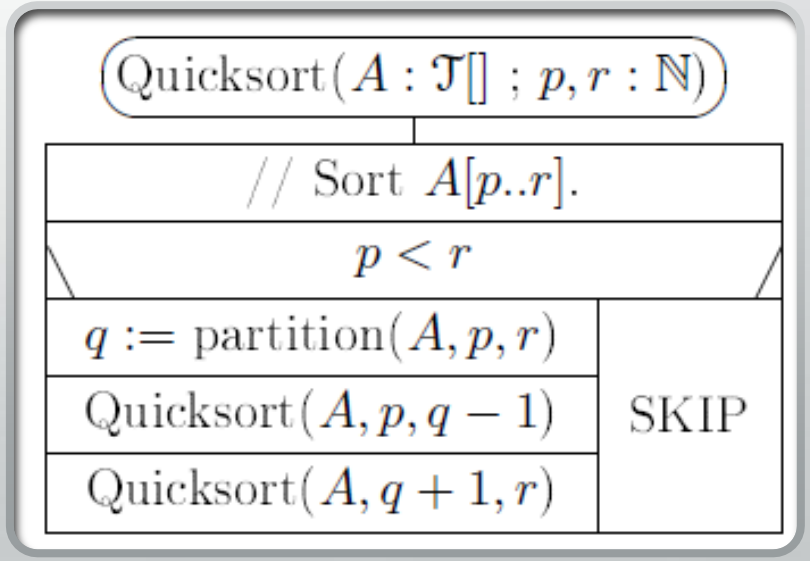
   ∑ Tengely a végleges helyére kerül
3. A fenti lépések **rekurzív alkalmazása:**
   - **tengelynél kisebb** elemek résztömbjére
   - **tengelynél nagyobb** elemek résztömbjére
4. **A rekurzió alapesetei**
   - **üres és az egyelemű résztömbök**

   ➢ Eleve készen vannak -> nem kell őket rendezni.

Gyorsrendezés algoritmusa



- A tengely kiválasztása és a részekre bontás lépései
  - Többféleképpen
    - A módszerek konkrét megválasztása erősen befolyásolja a rendezés hatékonyságát.
  - Fontos követelmény:
    - együtt lineáris időben befejeződjenek



Partition függvény:

- Jelölések:
  - $A[k..m] \le x \Leftrightarrow$ ha $\forall\, l\,(k \le l \le m) : A[l] \le x$
  - $A[k..m] \ge x \Leftrightarrow$ ha $\forall\, l\,(k \le l \le m) : A[l] \ge x$
- Bemenet: $A[p..r]$ résztömb, pivot: 5 (p+3. indexű elem)

	p			i				r
$A:$	5	3	8	**5**	6	4	7	**1**

	p							r	
$A:$	5	3	8	1	6	4	7	⑤	$pivot = A[r] = 5$

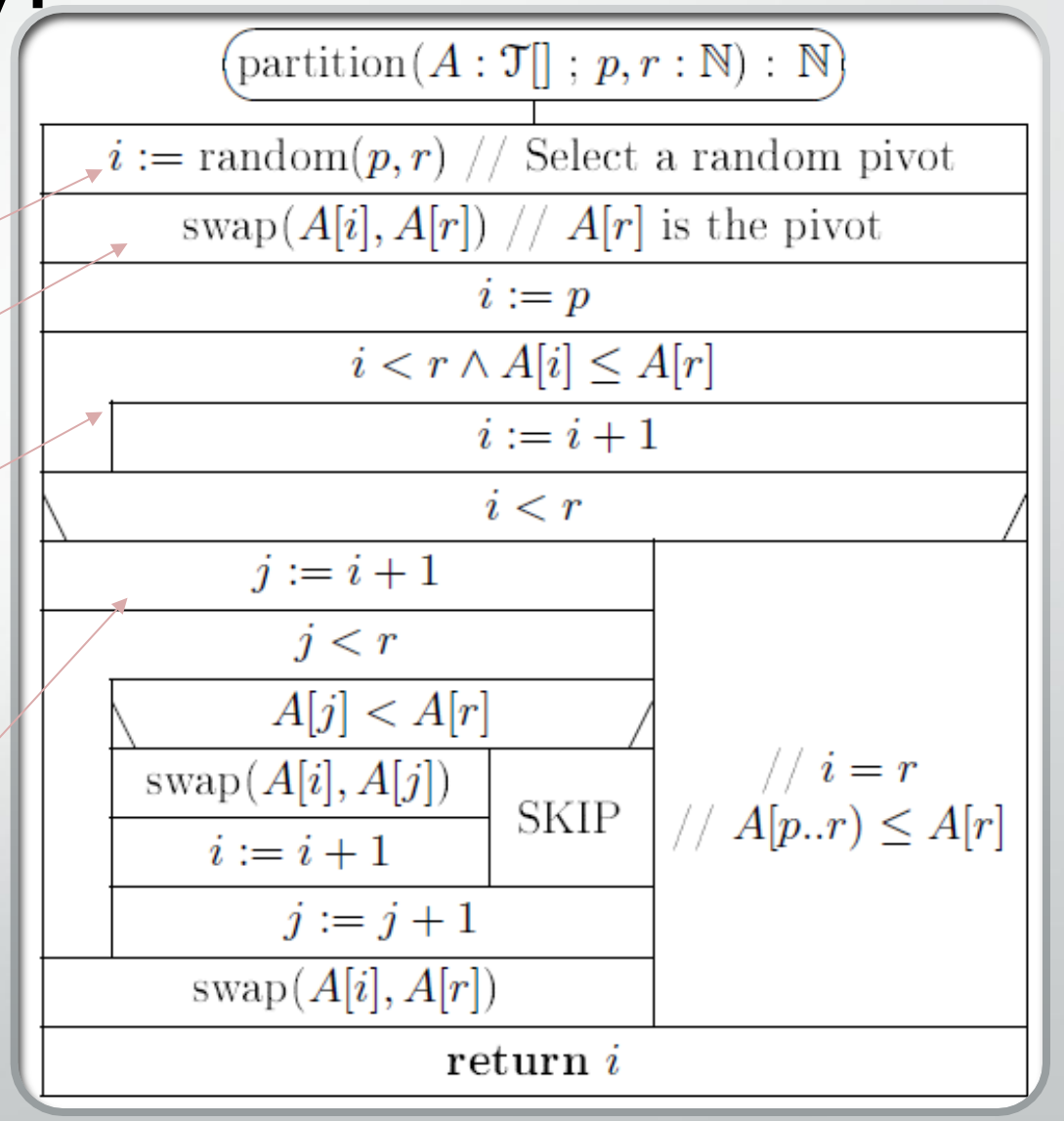
- 1. ciklus: megkeresi az 1. tengelynél > elemet (ha van)

	i=p	i	i					r
$A:$	5	3	**8**	1	6	4	7	⑤

- A j a következő elemre áll:
  - $p \le i < j \le r$

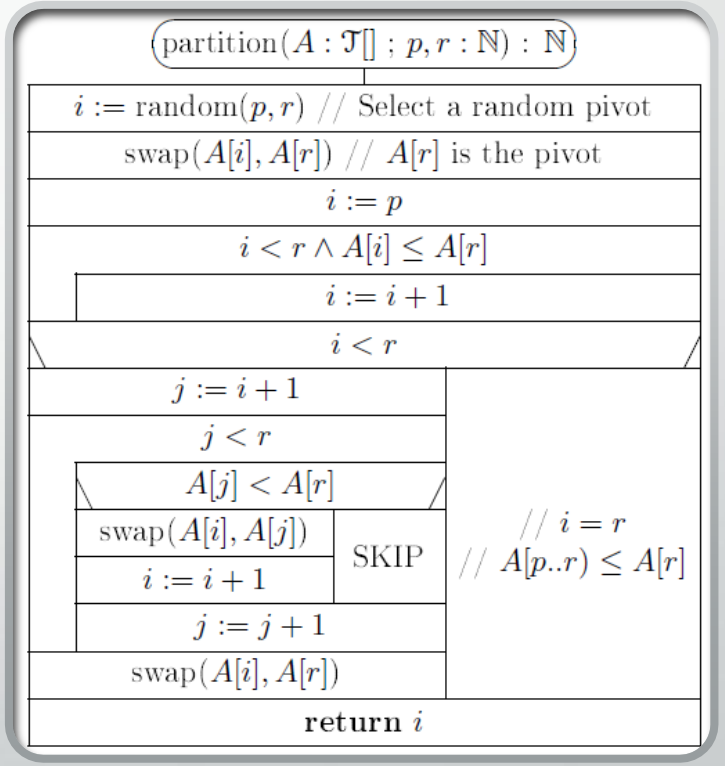
	p		i	j				r
$A:$	5	3	**8**	1	6	4	7	⑤

  - Az $A[p..r]$ résztömb szakaszai: (2. inv. tul.)
    - $A[p..i) \le pivot$ $p \le$
    - $A[i..j) \ge pivot$ $i <$
    - $A[j..r)$ (ismeretlen) $j \le$
    - $A[r]$ (*pivot*) r



Partition függvény

- A továbbiakban: az ismeretlen szakasz elemeit
  - sorban a tengelynél ≤ vagy ≥ elemek szakaszához kapcsoljuk
  - amíg az ismeretlen szakasz el nem fogy.
- Végül: a tengely -> az első két szakasz közé kerül
- A második szakasz eltolása: nagyon rossz hatékonyság
  - elkerülése: az aktuális az elemet $A[j] < A[r]$ esetben megcseréljük a második szakasz első elemével
  - következmény: a Quicksort nem stabil



$$\underset{A[p..i) \le pivot,}{p} \quad \underset{\le}{} \quad \underset{A[i..j) \ge pivot,}{i} \quad \underset{<}{} \quad \underset{A[j..r) \text{ (ismeretlen)},}{j} \quad \underset{\le}{} \quad \underset{A[r] \text{ (a } pivot)}{r}$$

Partition függvény 2. ciklus végrehajtásai

- $A[j] = 1 < pivot = 5$
  - csere($A[j]$, $A[i]$)
    - A[j] -> az A[$p..r$] 1. szakaszához (a tengelynél ≤ elemei)
    - A[i] (az inv. alapján) ≥ x -> a 2. szakasz (a tengelynél ≥ elemek szakasza)
    - Szakaszhatárok változása -> i ++; j ++ (cikl. inv. igaz maradjon)
- $A[j] = 6 \geq pivot = 5$
  - A[j]-t hozzávesszük a 2. szakaszhoz
  - j++

- A 2. ciklus változásai:

	p		i	j			r
$A:$	5	3	**8**	**1**	6	4	7	⑤

	p			i	j			r
$A:$	5	3	1	**8**	**6**	4	7	⑤

Partition függvény 2. ciklus végrehajtásai

- $A[j] = 4 < pivot = 5$
  - csere($A[j]$, $A[i]$)
  - i++
  - j++ (~1. eset)
- $A[j] = 7 \geq pivot = 5$
  - A[j]-t hozzávesszük a 2. szakaszhoz
  - j++
- $A[p..r]$ résztömb részekre bontását befejeztük

A 2. ciklus változásai:

	p			i		j		r
$A:$	5	3	1	**8**	6	**4**	7	⑤

	p				i		j	r
$A:$	5	3	1	4	**6**	8	**7**	⑤

	p				i			j=r
$A:$	5	3	1	4	**+5**	8	7	6

Partition függvény

- A partition fv helyességének ellenőrzéséhez vezessük még be a következő jelöléseket:
  - $A_0[p..r]$: az $A[p..r]$ tömb kezdeti állapota a partition függvény meghívásakor.
- A partition fv előfeltétele:
  - $0 \leq p < r < A.length$ (p, r a fv-en belül konstansok.)

Partition függvény

A partition fv. 2. ciklusának invariánsa:

- $A[p..r]$ egy permutációja az $A_0[p..r]$ résztömbnek $\wedge$
- $p \le i < j \le r \wedge$
- $A[p..(i-1)] \le A[r] \wedge$
- $A[i..(j-1)] \ge A[r]$

A partition fv utófeltétele:

- $A[p..r]$ az $A_0[p..r]$ permutációja $\wedge$
- $p \le i \le r \wedge A[p..(i-1)] \le A[i] \wedge$
- $A[(i+1)..r] \ge A[i]$, ahol i a visszatérési érték.

A gyors-rendezés C# kódja*

```
using System;

class QuickSortAlgorithm
{
    public static void QuickSort(int[] A, int p, int r)
    {
        if (p < r)
        {
            int q = Partition(A, p, r);
            QuickSort(A, p, q - 1);
            QuickSort(A, q + 1, r);
        }
    }
```

```
    private static void Swap(int[] A, int i, int j)
    {
        int temp = A[i];
        A[i] = A[j];
        A[j] = temp;
    }

    public static void Main()
    {
        int[] array = { 3, 8, 2, 5, 1, 4, 7, 6 };
        Console.WriteLine("Eredeti tömb: " + string.Join(", ", array));

        QuickSort(array, 0, array.Length - 1);

        Console.WriteLine("Rendezett tömb: " + string.Join(", ", array));
    }
}
```

```
static int Partition(int[] A, int p, int r)
{
    Random rand = new Random();
    int pivotIndex = rand.Next(p, r + 1); // Véletlenszerű pivot
    Swap(A, pivotIndex, r);               // Pivotot a végére tesszük

    int i = p;
    while (i < r && A[i] <= A[r])
    {
        i++;
    }

    if (i < r)
    {
        int j = i + 1;
        while (j < r)
        {
            if (A[j] < A[r])
            {
                Swap(A, i, j);
                i++;
            }
            j++;
        }
        Swap(A, i, r);
    }

    return i;
}
```

A gyorsrendezés (quicksort) műveletigénye

- A fenti szétvágás (partition) műveletigénye lineáris
  - a két ciklus együtt $r - p - 1$ vagy $r - p$ iterációt végez
- Műveletigény:

$$mT(n), AT(n) \in \Theta(n \log n)$$

$$MT(n) \in \Theta(n^2)$$

  - A várható vagy átlagos műveletigény aszimptotikusan a legjobb esethez esik közel
  - A legrosszabb eset valószínűsége nagyon kicsi

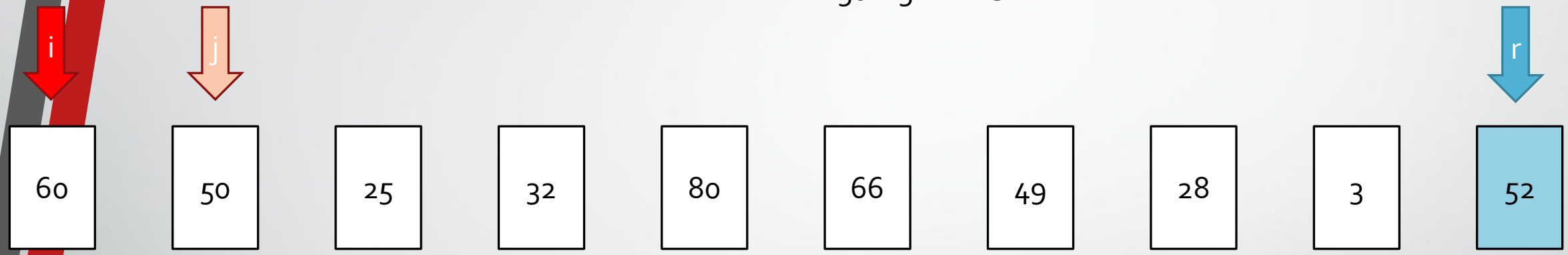
A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

i < r and A[i] <= A[r]

50 < 52 ? YES

j < r ? Igen



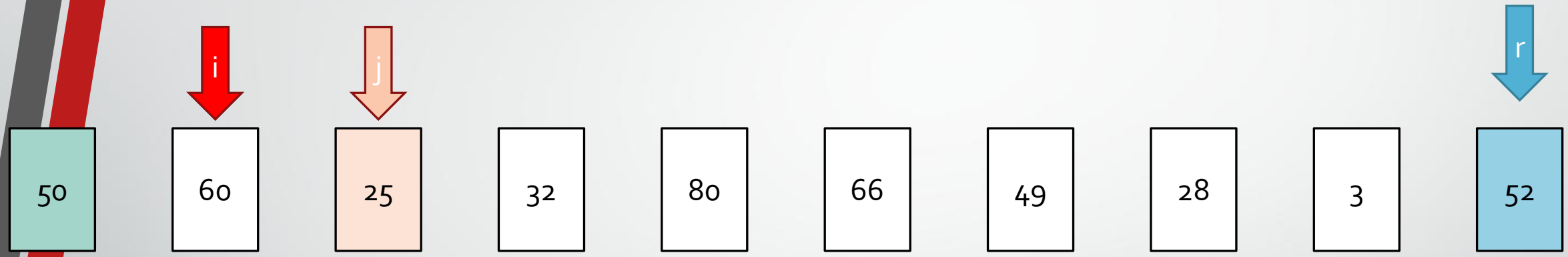
i := 0

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

j < r ? Igen

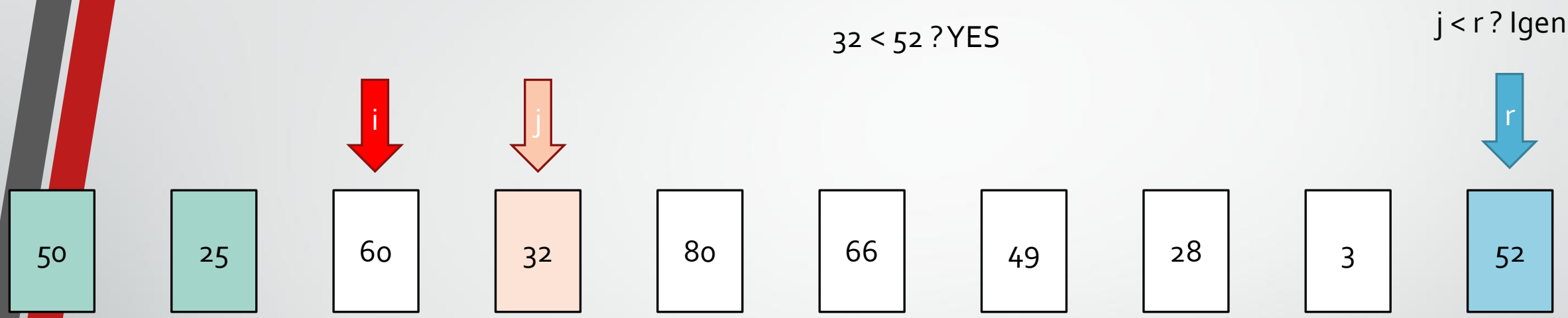
25 < 52 ? YES



i := 1

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

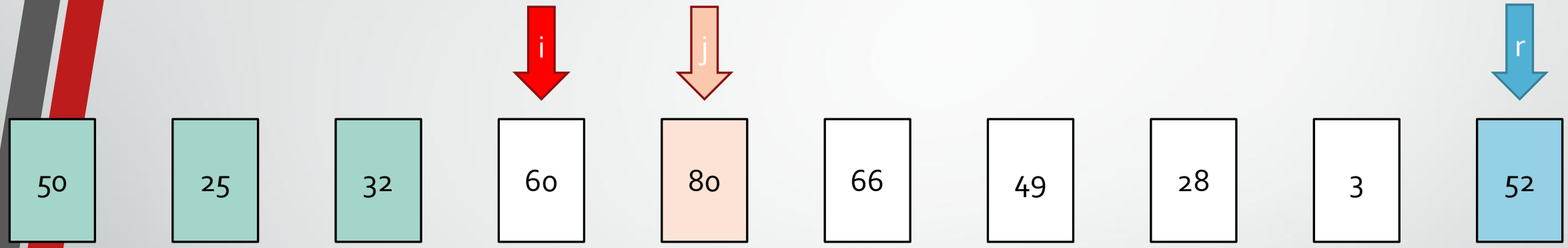


A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

80 < 52 ? NO

j < r ? Igen



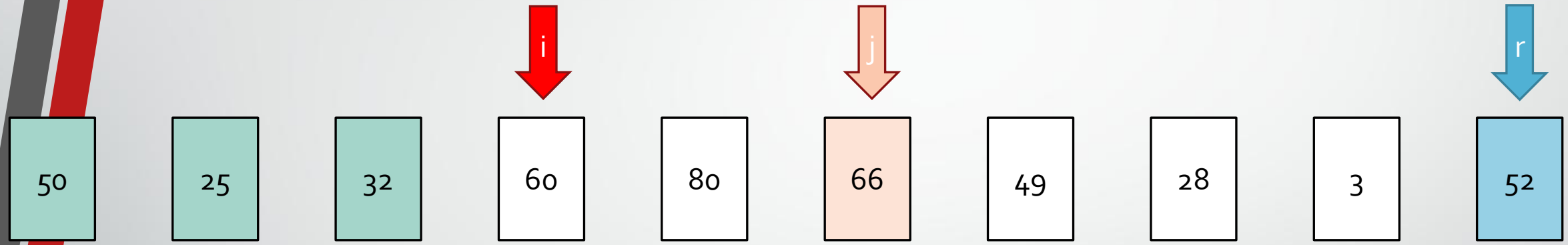
i := 3

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

66 < 52 ? NO

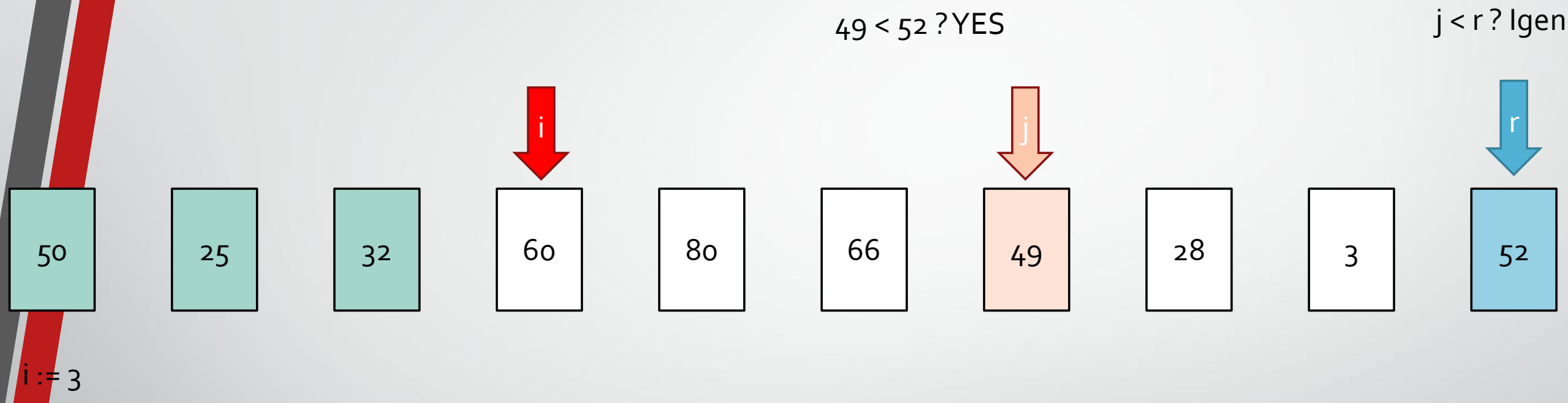
j < r ? Igen



i := 3

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

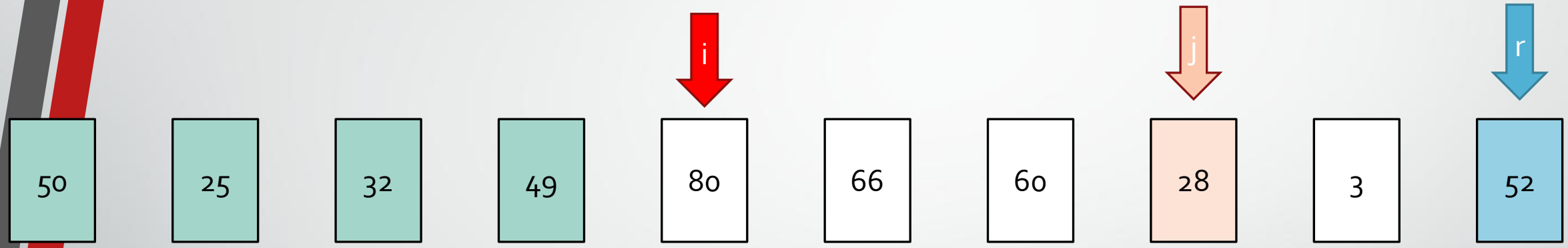


A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

j < r ? Igen

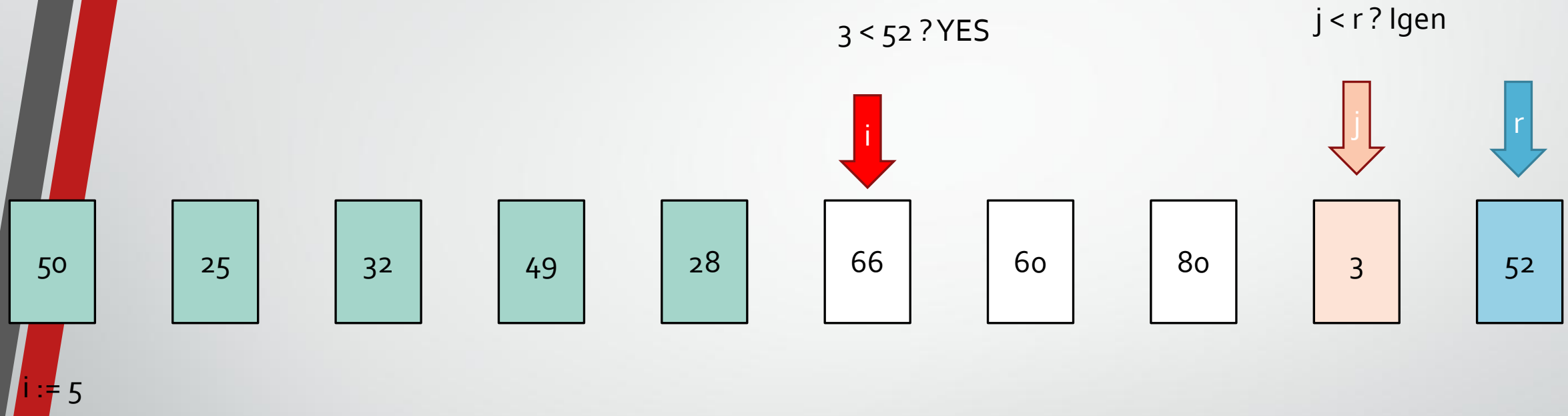
28 < 52 ? YES



i := 4

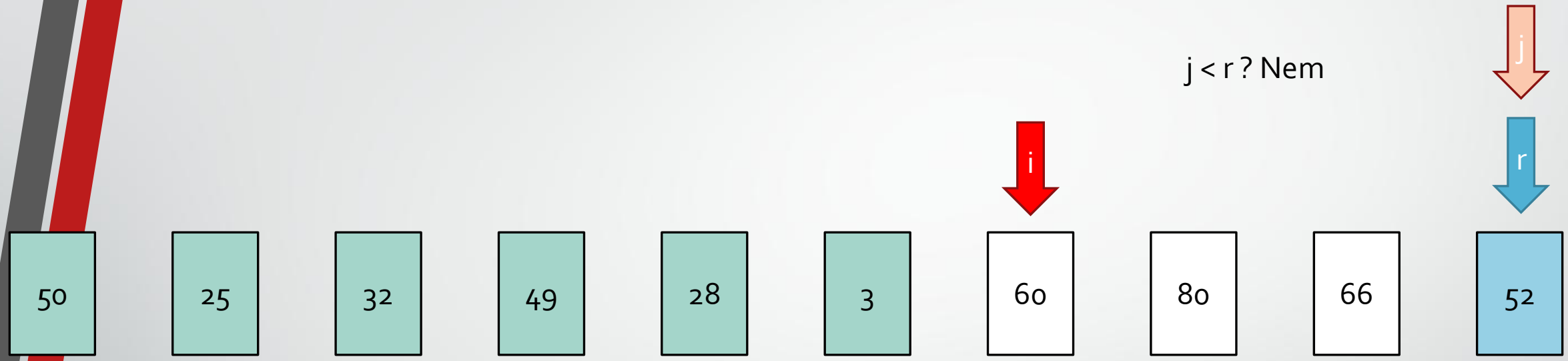
A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek



A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek



i := 6

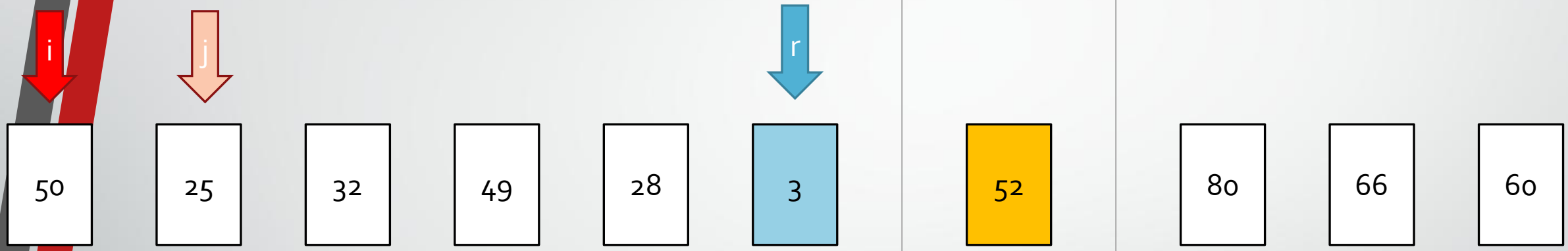
A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

i < r and A[i] <= A[r]

25 < 3 ? NO

j < r ? Igen



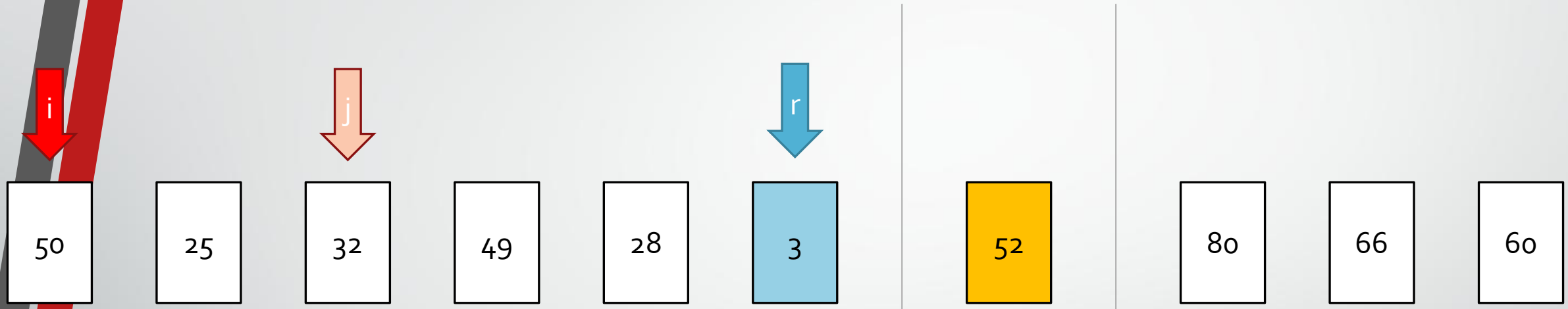
i := 0

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

32 < 3 ? NO

j < r ? Igen



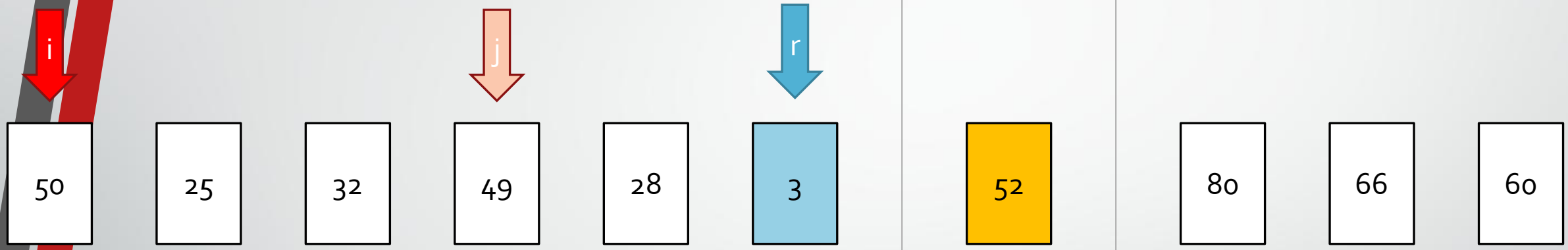
i := 0

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

49 < 3 ? NO

j < r ? Igen



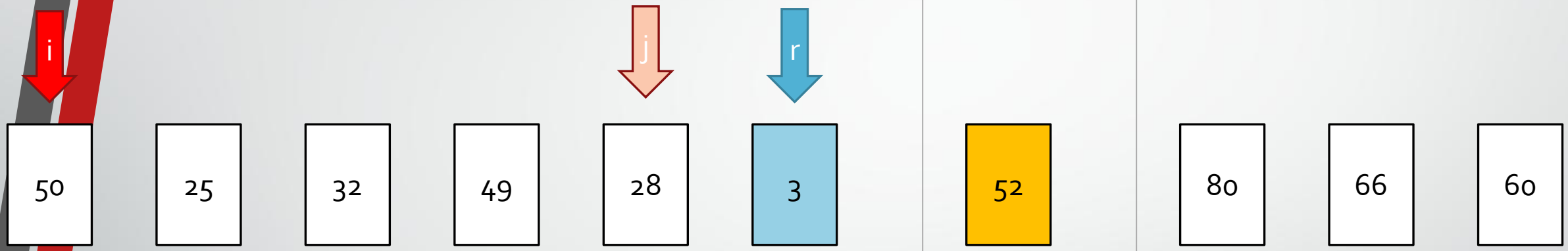
i := 0

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

28 < 3 ? NO

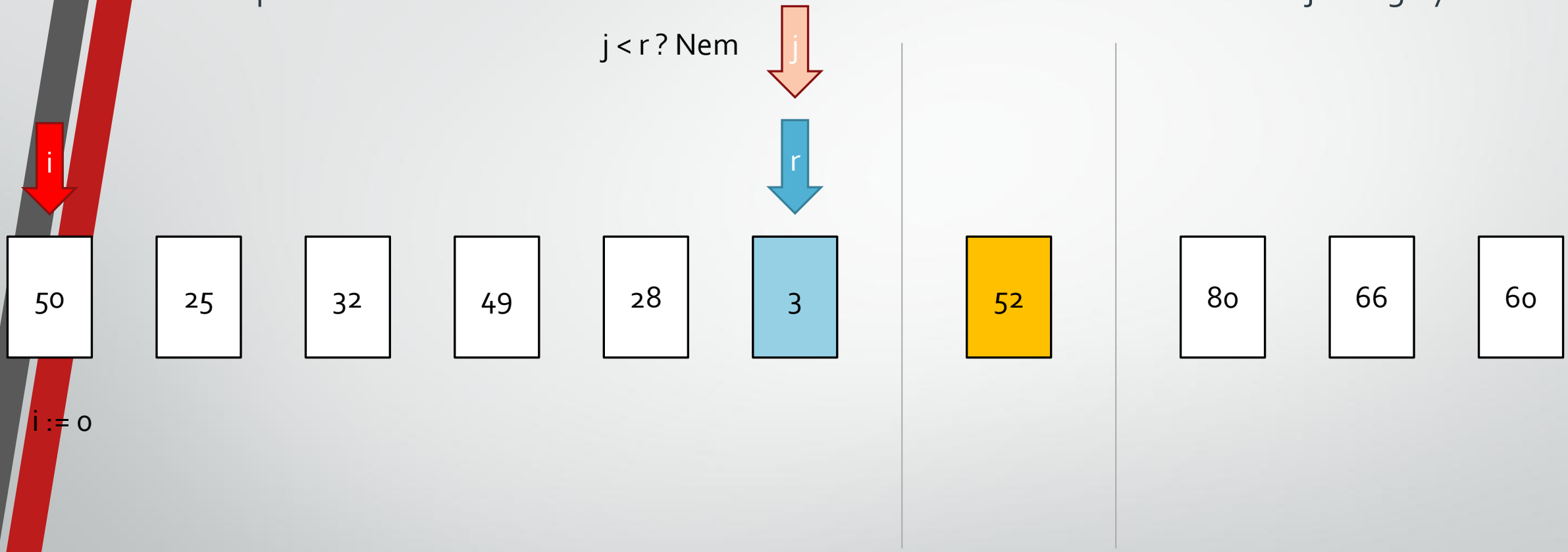
j < r ? Igen



i := 0

A gyorsrendezés lejátszása:*

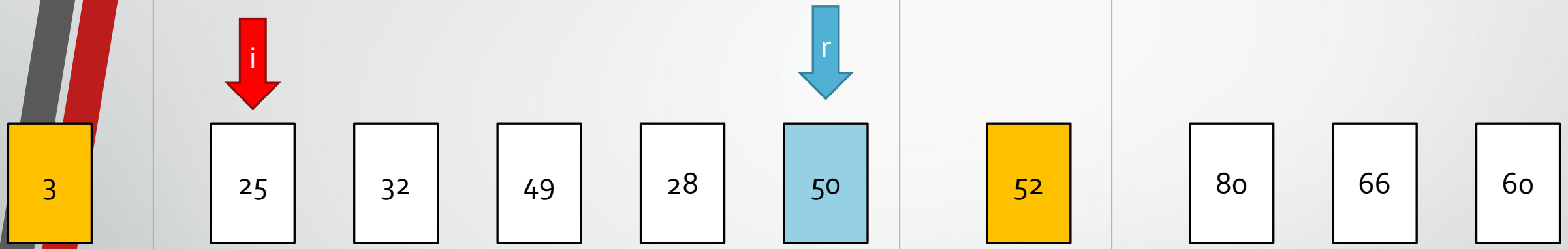
most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek



A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

i < r and A[i] <= x

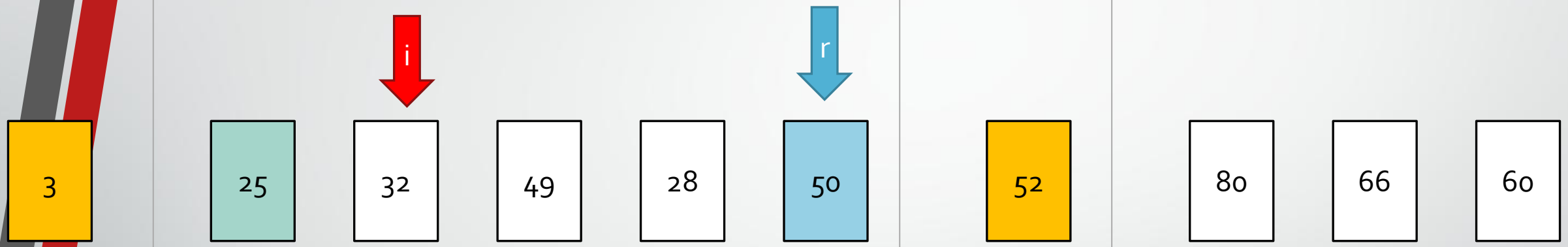


i := 1

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

i < r and A[i] <= x

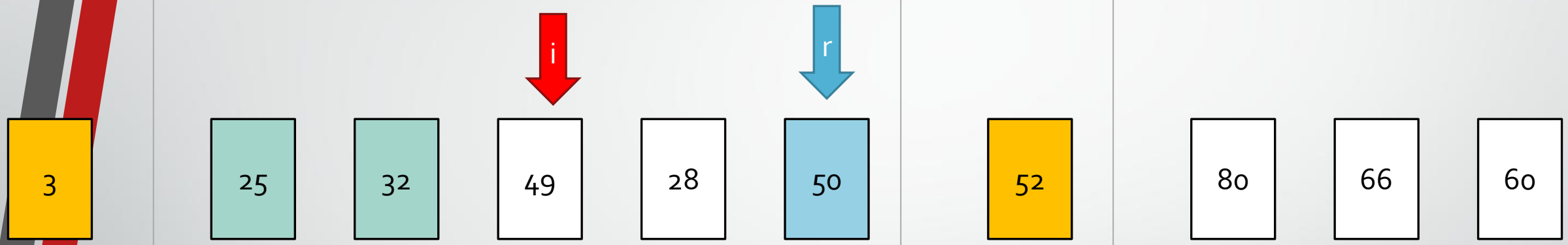


i := 2

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

i < r and A[i] <= x

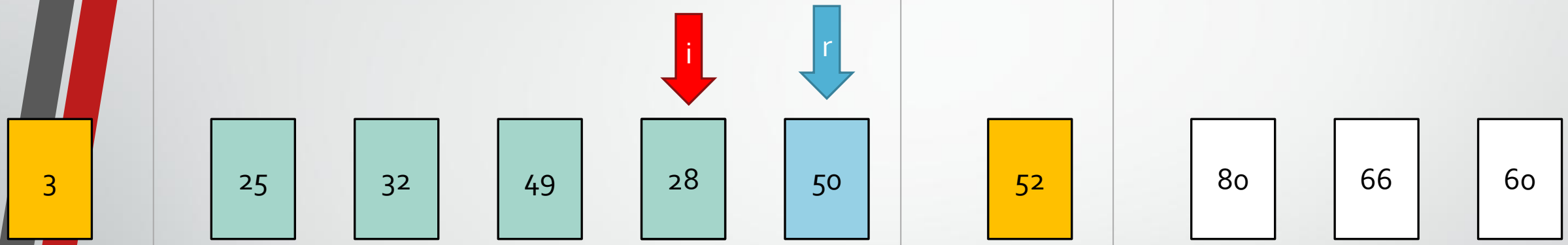


i := 3

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

i < r and A[i] <= x



i := 4

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

i < r and A[i] <= A[r]

i := 1

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

i < r and A[i] <= A[r]

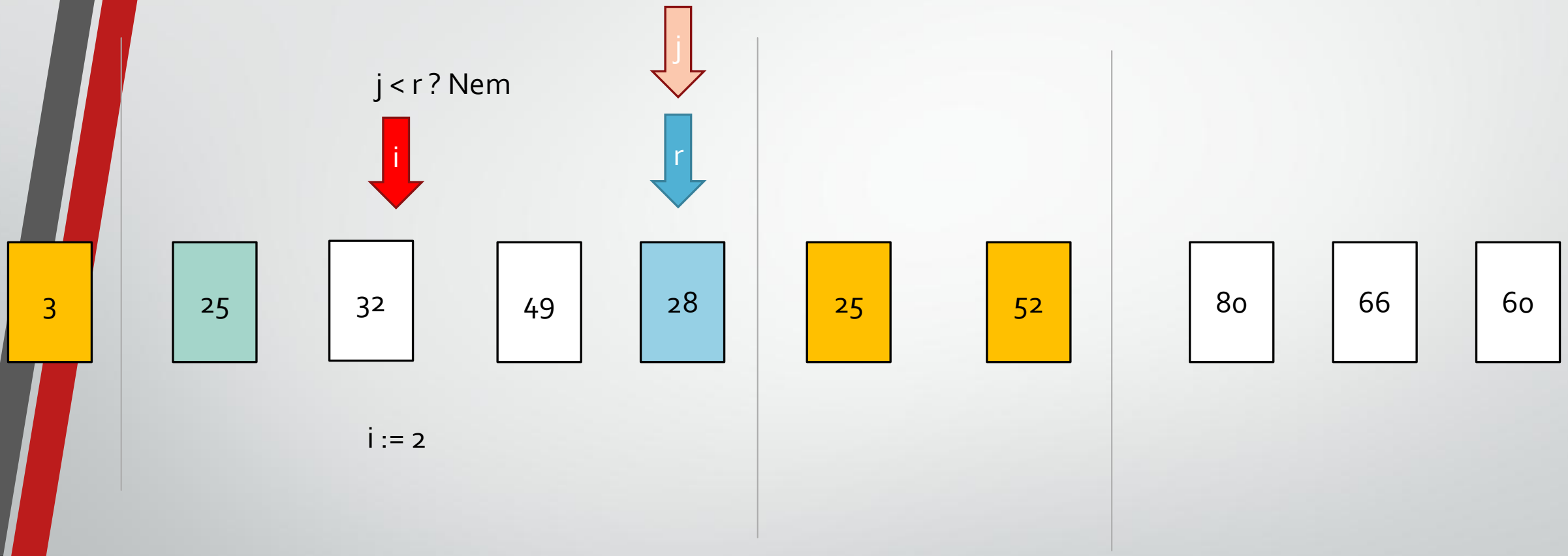
49 < 28 ? NO

j < r ? Igen

i := 2

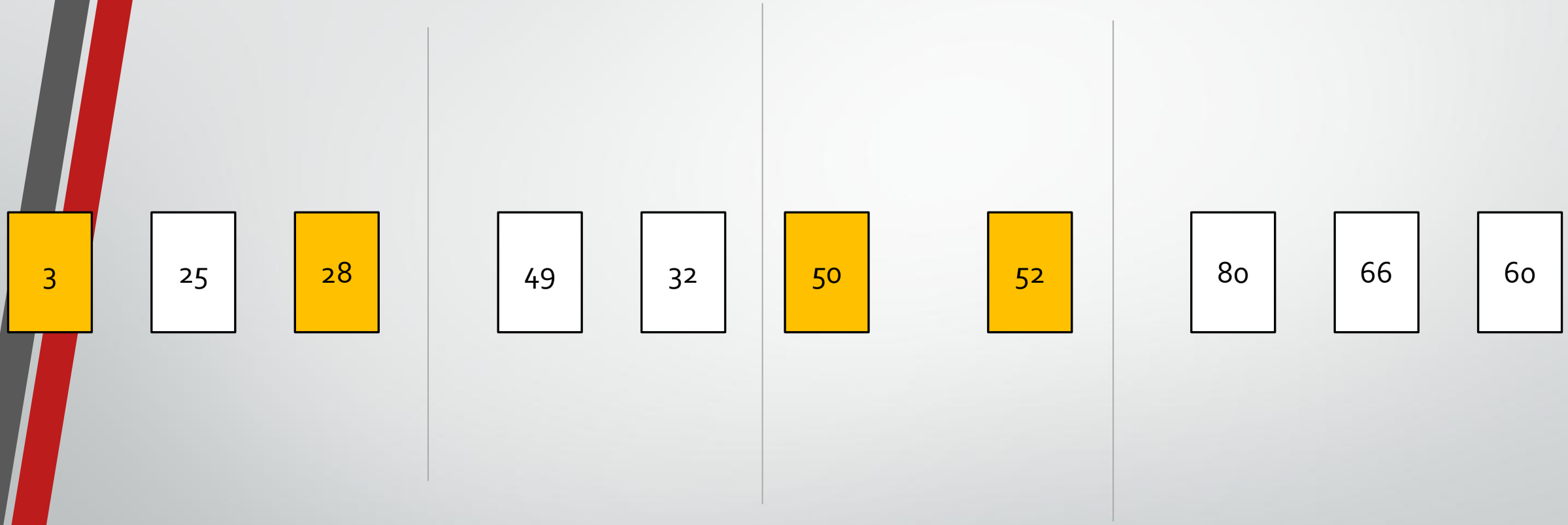
A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek



A gyorsrendezés lejátszása:*

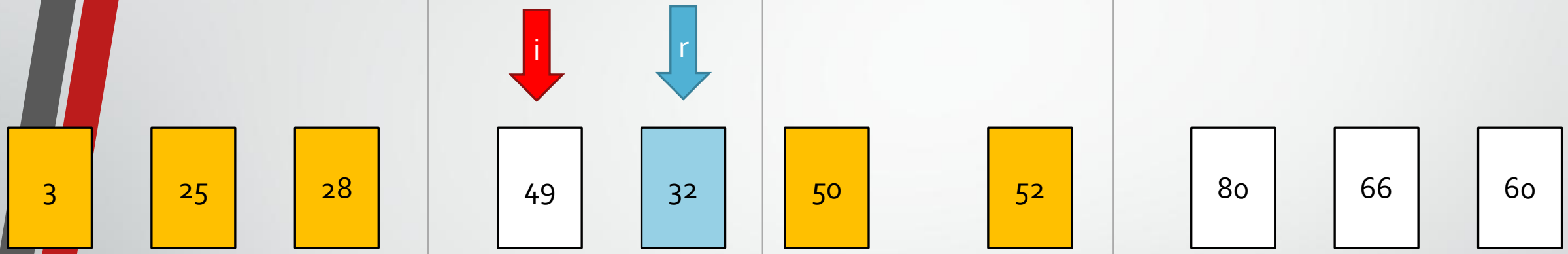
most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek



A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

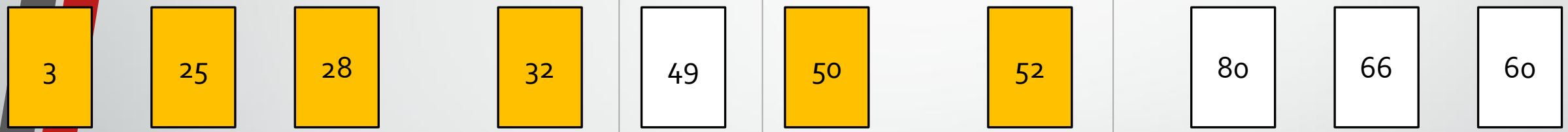
i < r and A[i] <= A[r]



i := 3

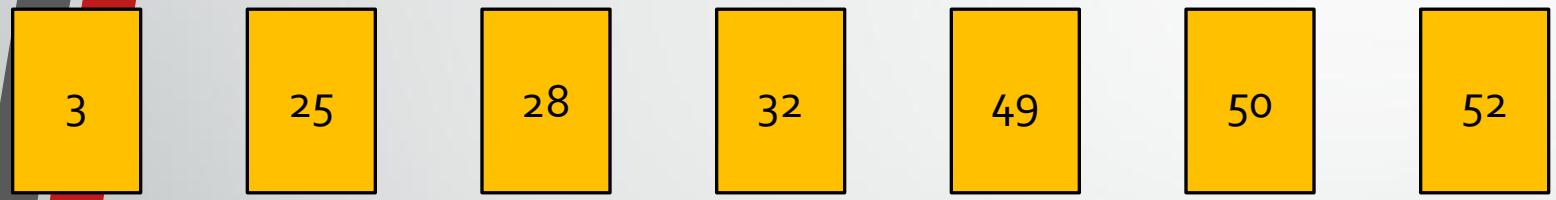
A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek



A gyorsrendezés lejátszása:*

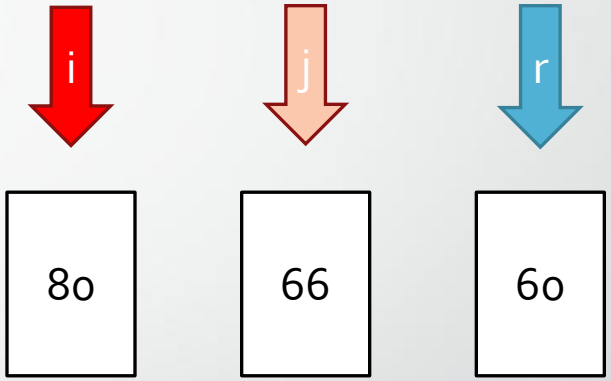
most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek



66 < 60 ? NO

j < r ? Igen

i < r and A[i] <= A[r]

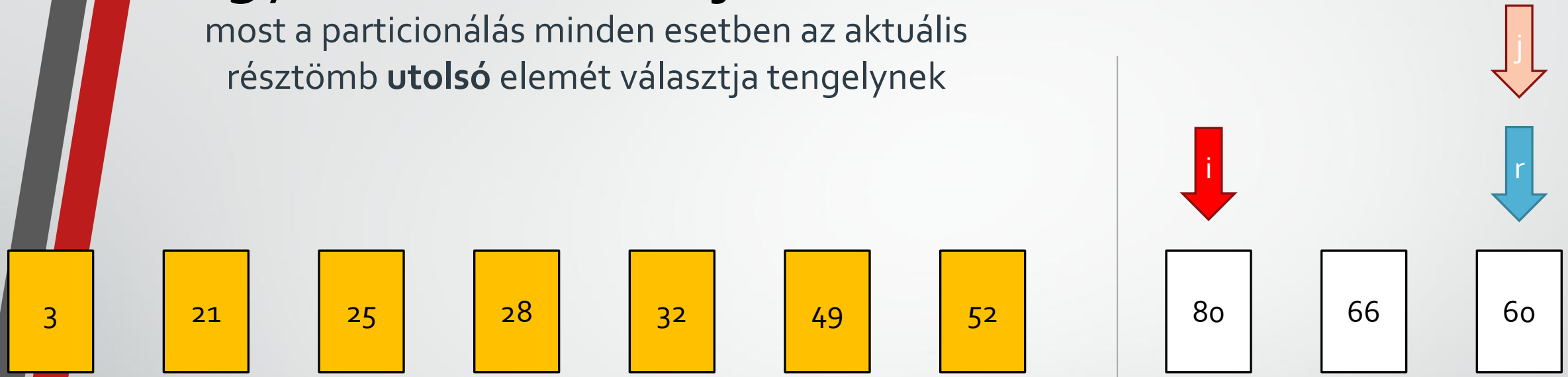


i := 7

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

j < r ? Nem

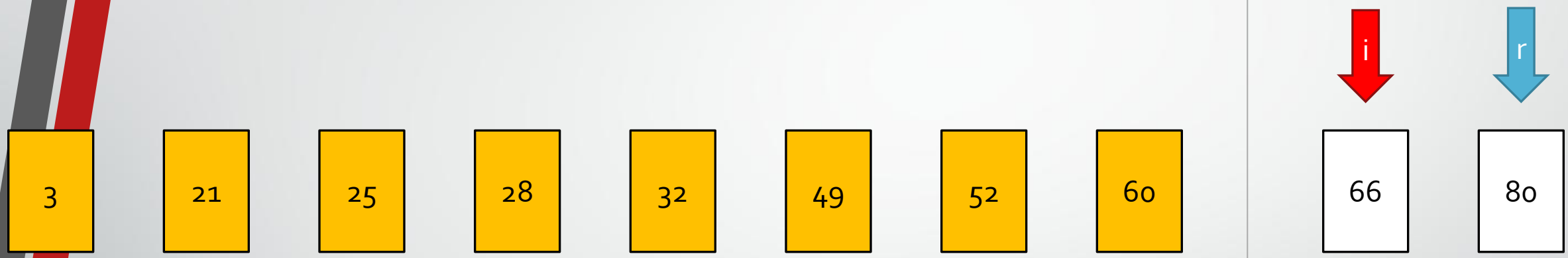


i := 7

A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

i < r and A[i] <= A[r]

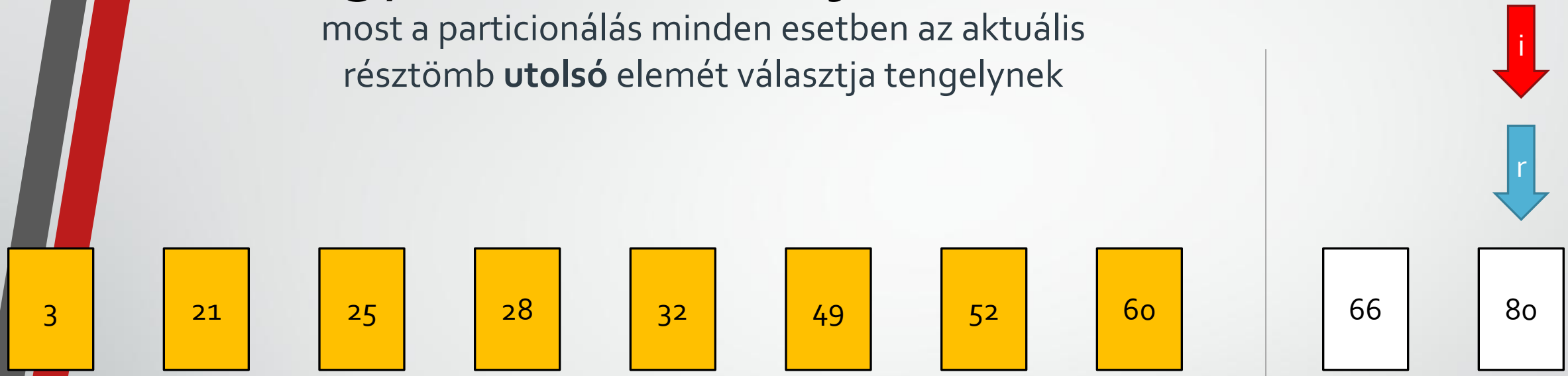


i := 8

i < r Nem

A gyorsrendezés lejátszása:*

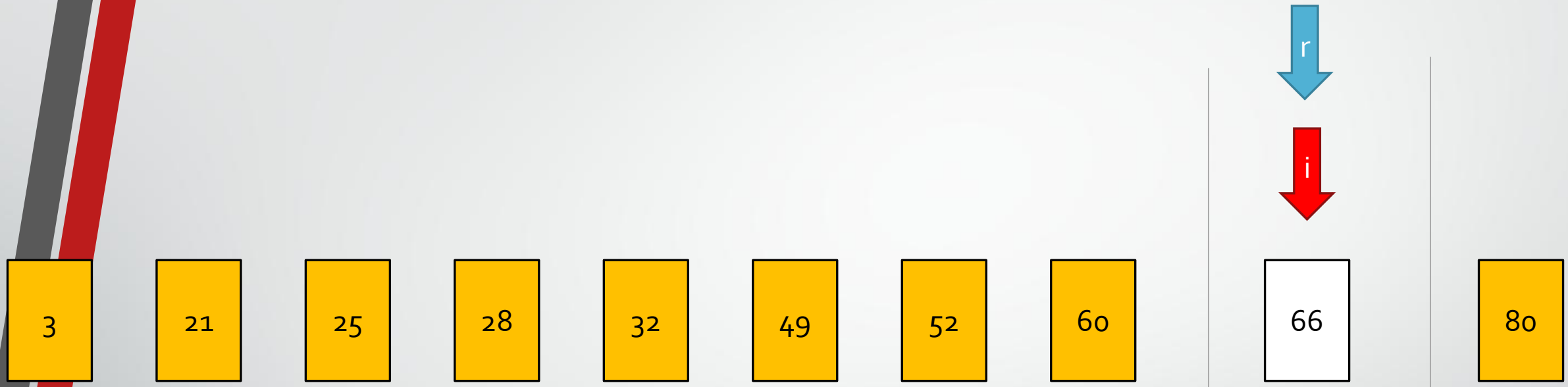
most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek



i := 9

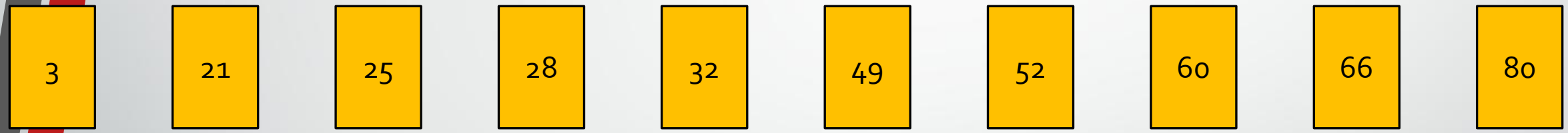
A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek



A gyorsrendezés lejátszása:*

most a particionálás minden esetben az aktuális résztömb **utolsó** elemét választja tengelynek

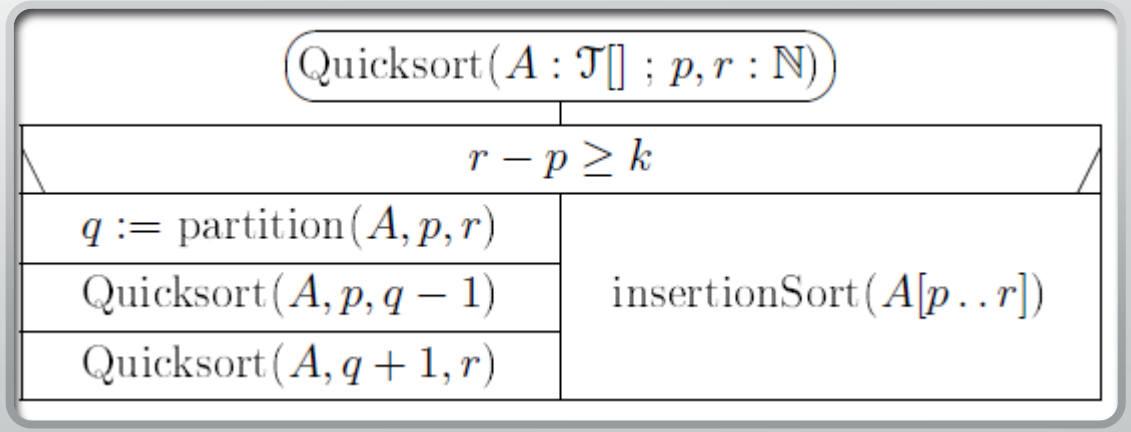


Vegyes gyorsrendezés

- Kis elemszámnál (max. néhányszor tíz rendezendő elem):
  - a beszúró rendezés hatékonyabb a gyors rendezéseknél (merge sort, heap sort, quicksort)

➢ Quicksort(*A*, *p*, *r*) eljárás jelentős gyorsítása:

- kis méretű résztömböknél áttérünk a beszúró rendezésre
- $k \in \mathbb{N}$ (20 és 40 között)

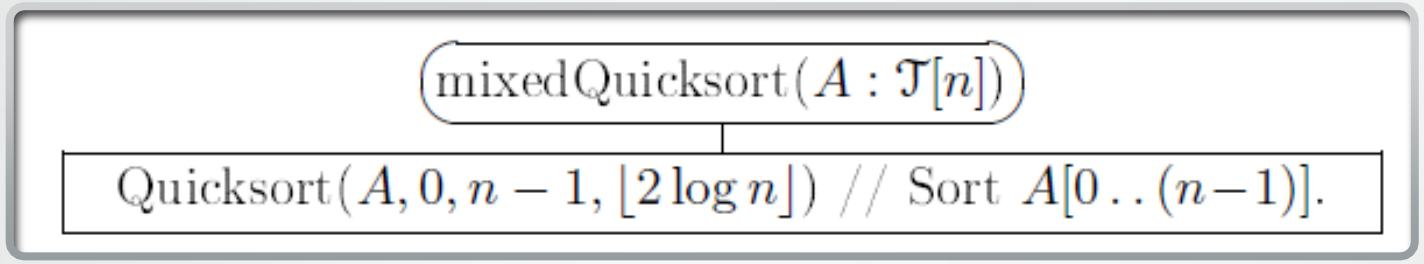


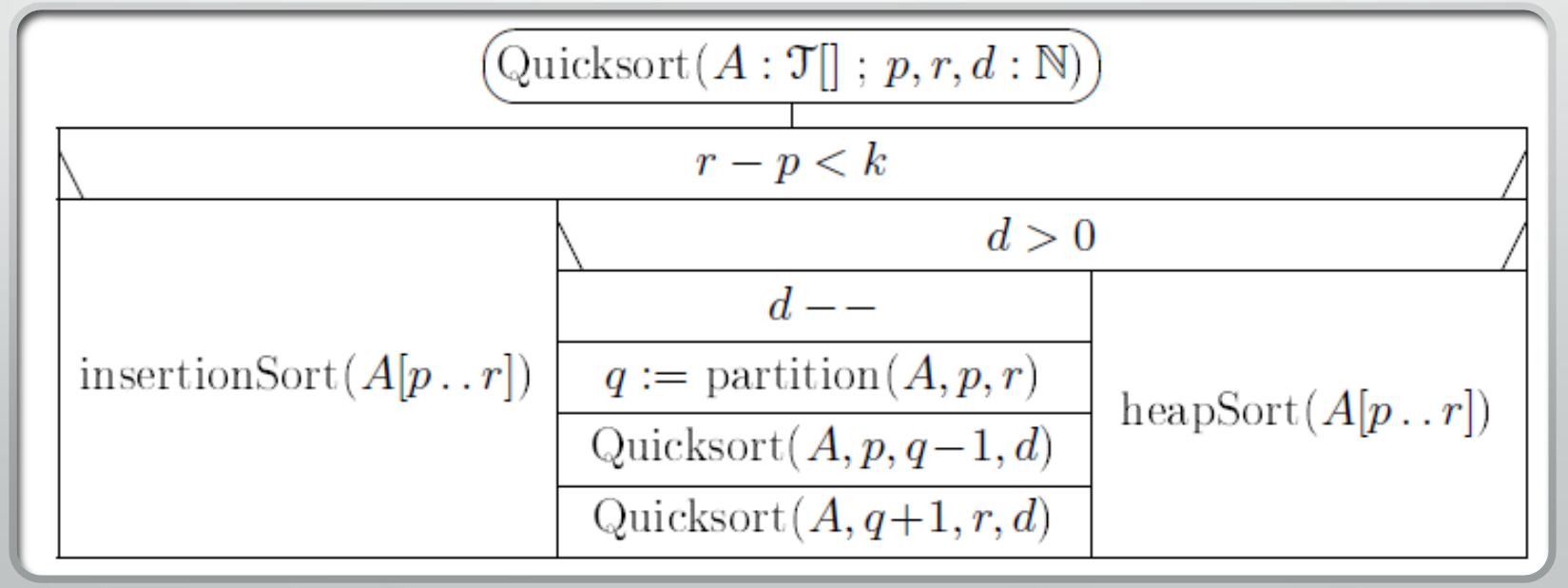
Vegyes gyorsrendezés

- Legrosszabb esetének javítása ($\Theta(n^2)$ -> $\Theta(n \log n)$)
  - rekurzív eljárásában: a rekurziós mélységet figyelése
  - Bizonyos (pl. 2 log *n*) mélység meghaladásánál
  - áttérünk valamelyik $MT(n) \in \Theta(n \log n)$ gyors rendezésre (pl. heap, merge).

Quicksort
+ insertionSort
+ heapSort

Vegyes gyorsrendezés algoritmusa





Vegyes gyorsrendezés műveletigénye

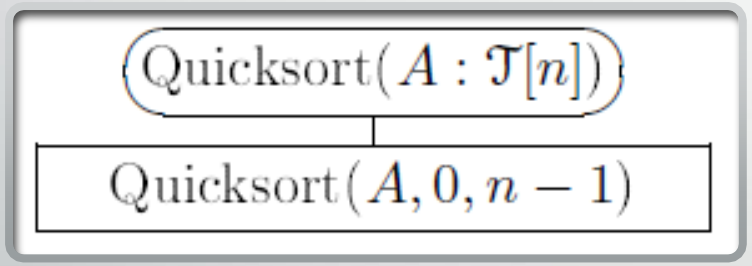
- Ha a mélységi korlát elérésekor **kupacrendezésre** váltunk
  - előny: helyben rendez (Θ (1) tárigény) ~ beszúró rendezés
- Tárigény
  - = a rekurziós mélység (pozitív együtthatós) lineáris függvényével
    - a legjobb és a legrosszabb esetben is) Θ (log n)

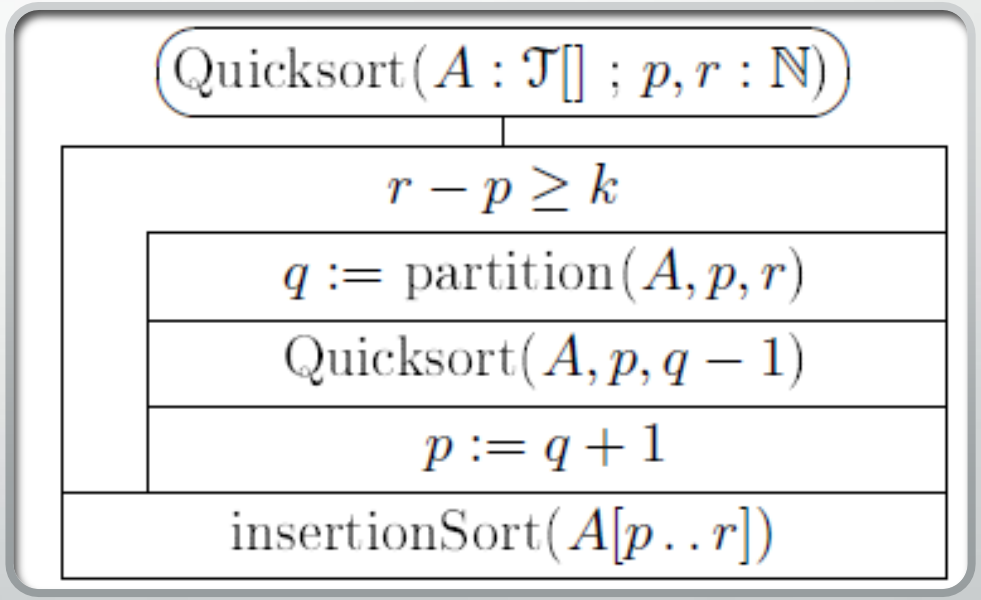
➢ minimális és maximális tárigény: Θ (log n)

$$mS_{\text{mixedQuicksort}}(n),\ MS_{\text{mixedQuicksort}}(n) \in \Theta(\log n)$$

$$mT_{\text{mixedQuicksort}}(n),\ MT_{\text{mixedQuicksort}}(n) \in \Theta(n \log n)$$

A gyorsrendezés végrekurzió-optimalizált változata *





Ellenőrző kérdések

- A gyorsrendezés stabil rendezés-e?
- Mennyi a gyorsrendezés műveletigénye és tárigénye? Miért?
- Adja meg a gyorsrendezés lépéseit!
- Adja meg a gyorsrendezés struktogramjait!
  - Tömbre
  - Egyszerű egyirányú listára
  - Ciklikus kétirányú listára
- Hogyan lehet a gyorsrendezést javítani?

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek I. előadásjegyzete alapján készült.

Algoritmusok és adatszerkezetek I.
5. Előadás

Függvények aszimptotikus viselkedése

(a Θ,O,Ω,$\prec$,$\succ$, o, ω matematikája)

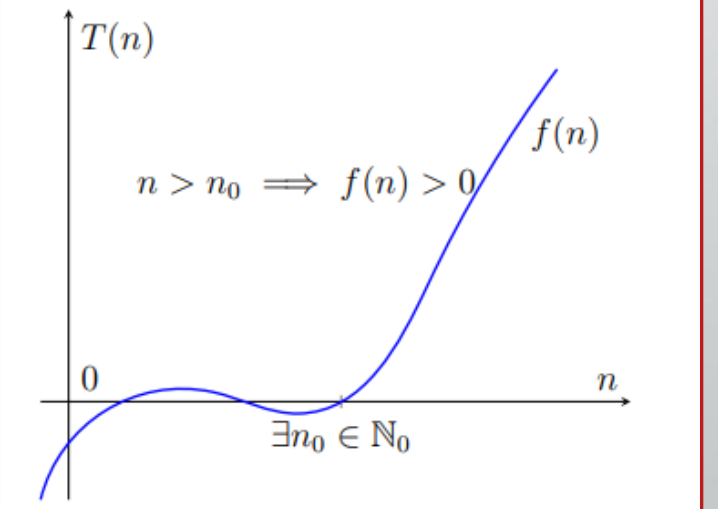
Tartalom

Függvények aszimptotikus viselkedése

1. **Definíció.** Valamely *P(n)* tulajdonság elég nagy *n* -ekre pontosan akkor teljesül, ha $\exists N \in \mathbb{N}$, hogy $\forall n \in \mathbb{N}$ -re $n \geq N$ esetén igaz *P(n)*.

2. **Definíció.** Az *f* **AP (aszimptotikusan pozitív) függvény**, ha elég nagy *n*-ekre *f(n)* > 0.

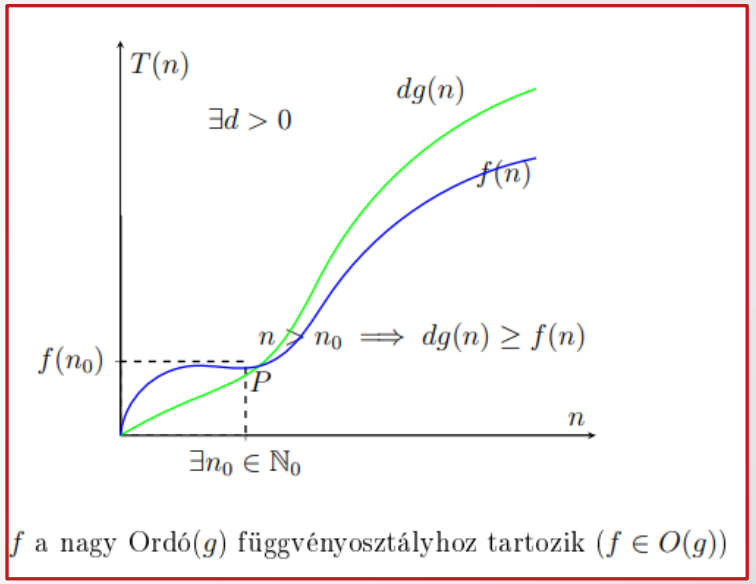
- Az (alsó és/vagy felső) becslések végzése a futási időre vagy a tárigényre
  - Az input adatszerkezetek méretének függvényében végezzük a becsléseket
  - a becsléseket leíró függvények $\mathbb{N} \to \mathbb{R}$ típusúak
  - $\mathbb{N} \to \mathbb{P}$, de az egyszerűség miatt megelégszünk fv-ek AP-k legyenek
- **Jelölések.** Az *f, g, h* (esetleg indexelt) latin betűkről feltesszük, hogy $\mathbb{N} \to \mathbb{R}$ típusú, aszimptotikusan pozitív függvényeket jelölnek, míg a ϕ, ψ görög betűkről csak azt tesszük fel, hogy $\mathbb{N} \to \mathbb{R}$ típusú függvényeket jelölnek.



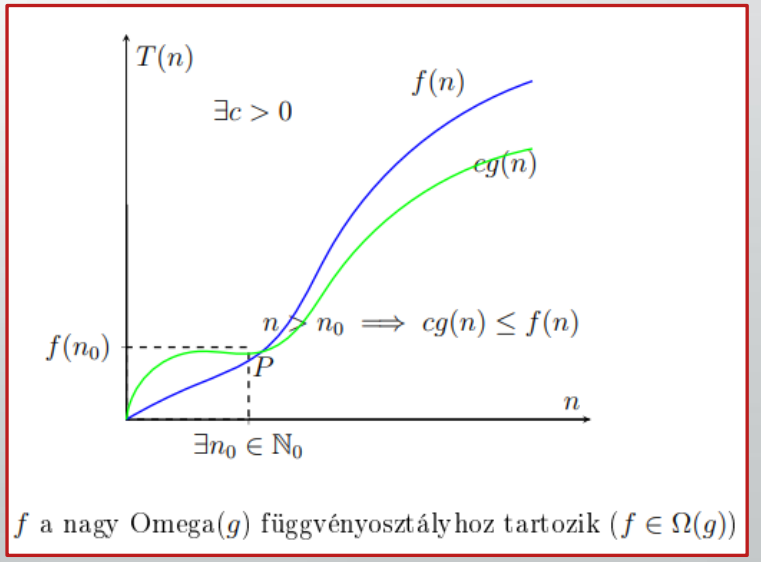
Aszimptotikus korlátok

3. Definíció. Az $O(g)$ függvényhalmaz olyan f függvényekből áll, amiket elég nagy n helyettesítési értékekre, megfelelő pozitív konstans szorzóval felülről becsül a g függvény:
$O(g) = \{f : \exists d \in \mathbb{P}$, hogy elég nagy n-ekre $d * g(n) \geq f(n).\}$
$f \in O(g)$ esetén azt mondjuk, hogy ***g* aszimptotikus felső korlátja *f*-nek**.

4. Definíció. Az $\Omega(g)$ függvényhalmaz olyan f függvényekből áll, amiket elég nagy n helyettesítési értékekre, megfelelő pozitív konstans szorzóval alulról becsül a g függvény:
$\Omega(g) = \{f : \exists c \in \mathbb{P}$, hogy elég nagy n-ekre $c * g(n) \leq f(n).\}$
$f \in \Omega(g)$ esetén azt mondjuk, hogy ***g* aszimptotikus alsó korlátja *f*-nek**.



f a nagy Ordó(g) függvényosztályhoz tartozik ($f \in O(g)$)



f a nagy Omega(g) függvényosztályhoz tartozik ($f \in \Omega(g)$)

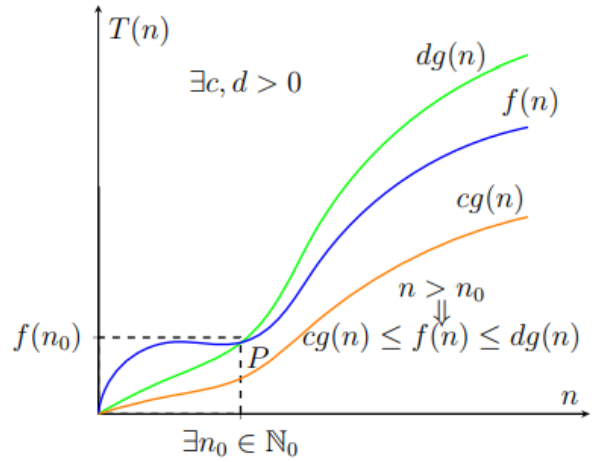
Aszimptotikus korlátok

5. Definíció. $\Theta(g) = O(g) \cap \Omega(g)$

6. Következmény. A $\Theta(g)$ függvényhalmaz olyan f függvényekből áll, amiket elég nagy n helyettesítési értékekre, megfelelő pozitív konstans szorzókkal alulról és felülről is becsül a g függvény:

$\Theta(g) = \{f : \exists c, d \in \mathbb{P}$, hogy elég nagy n –ekre $c * g(n) \leq f(n) \leq d * g(n).\}$

$f \in \Theta(g)$ esetén tehát azt mondhatjuk, hogy ***g* aszimptotikus** alsó és felső, azaz **éles korlátja *f*-nek.** (Ezt a 9. tulajdonságnál is láthatjuk.)



f a Theta(g) függvényosztályhoz tartozik ($f \in \Theta(g)$)

Függvények aszimptotikus összehasonlítása

7. Definíció.

$$\varphi \prec g \iff \lim_{n\to\infty} \frac{\varphi(n)}{g(n)} = 0$$

Ilyenkor azt mondjuk, hogy ***φ* aszimptotikusan kisebb, mint *g*.** (Vegyük észre, hogy φ nem okvetlenül AP!) AP függvényekre $f \prec g \iff f \in o(g)$
azaz definíció szerint:

$$o(g) = \{f : f \prec g\}$$

8. Definíció.

$$f \prec \psi \iff \psi \succ f$$

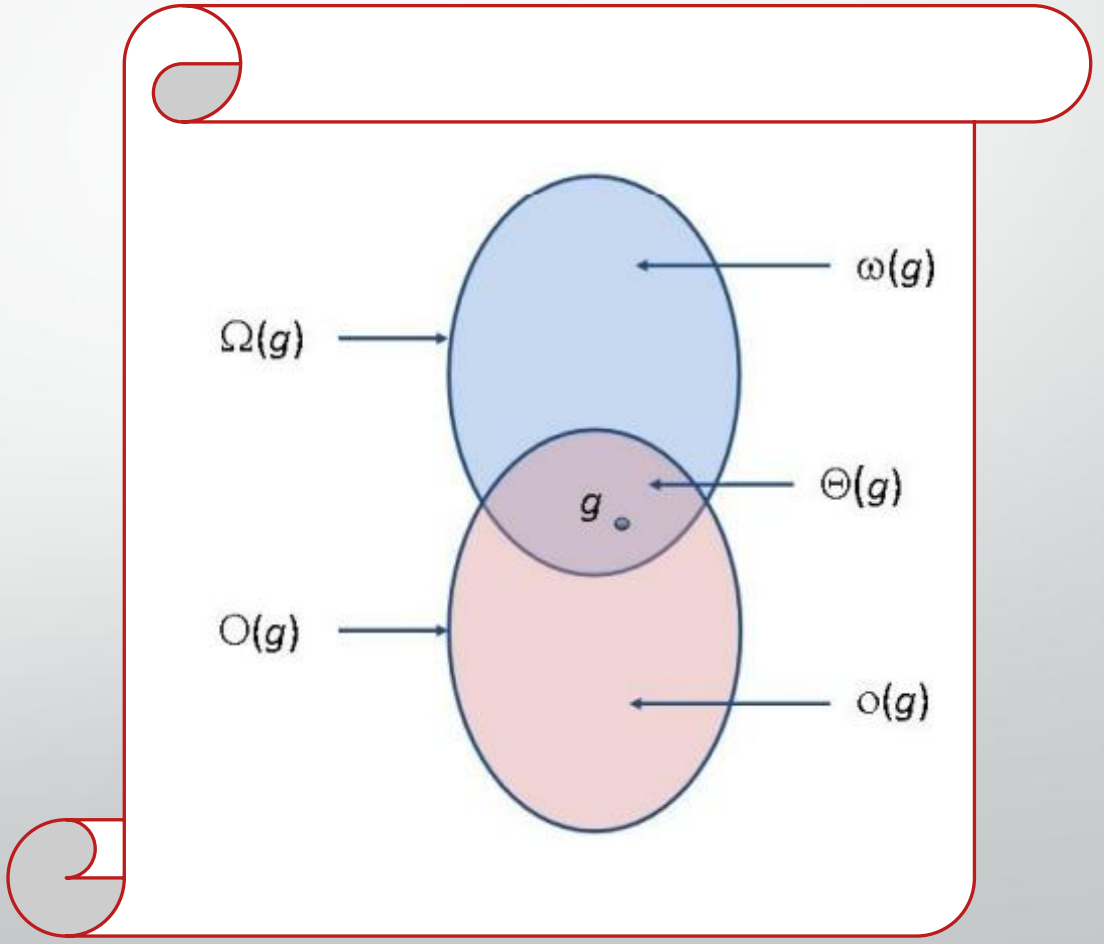
Ilyenkor azt mondjuk, hogy ***f* aszimptotikusan nagyobb, mint *ψ*.** (Vegyük észre, hogy ψ nem okvetlenül AP!) AP függvényekre $f \succ g \iff f \in \omega(g)$
azaz definíció szerint:

$$\{f : \omega(g) = \{f : f \succ g\}$$

A függvényosztályok kapcsolata

9. Tulajdonság.

- $\Theta(g) = O(g) \cap \Omega(g)$
- $o(g) \subsetneq O(g) \setminus \Omega(g)$
- $\omega(g) \subsetneq \Omega(g) \setminus O(g)$



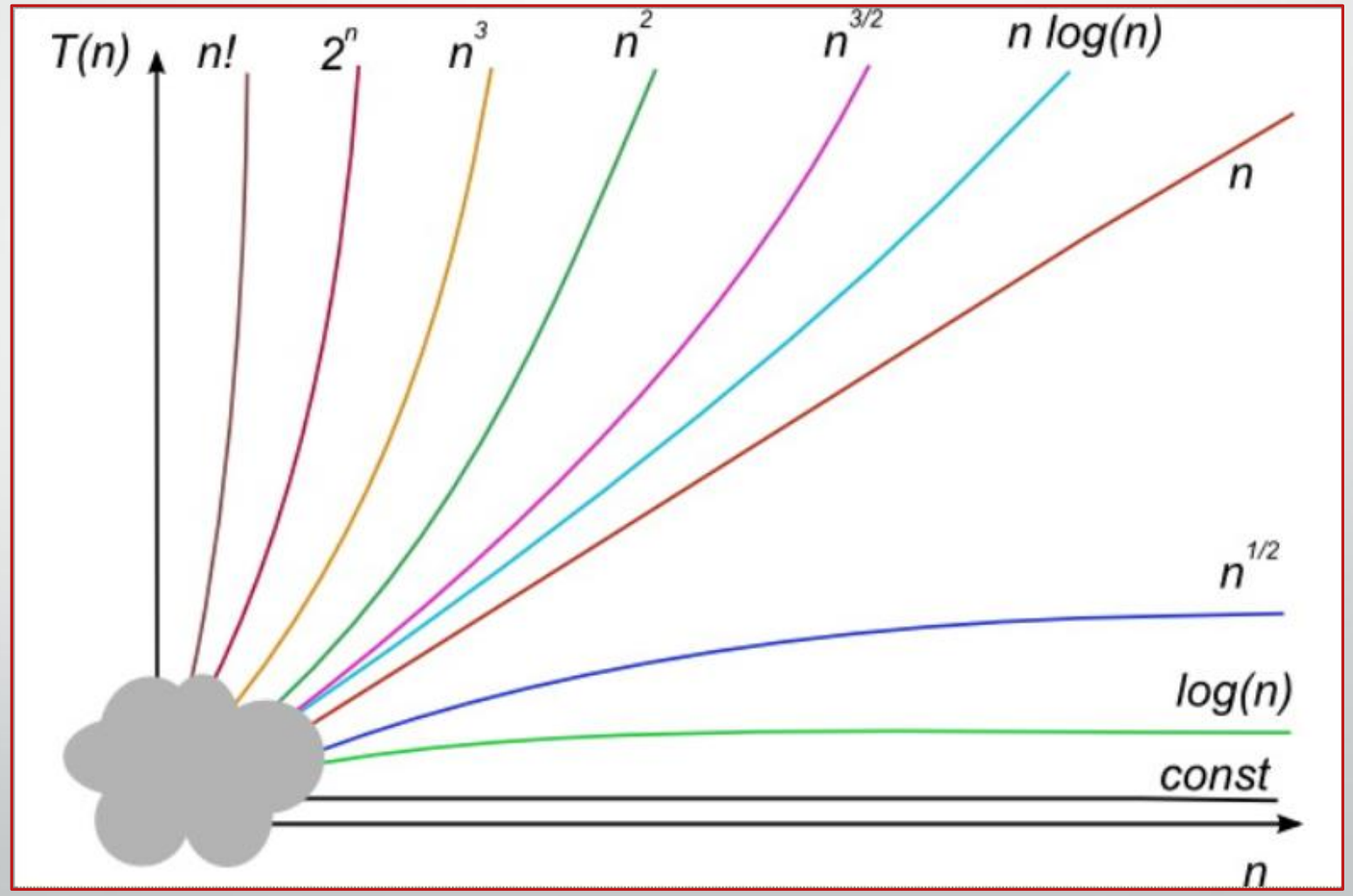
A függvényosztályok tulajdonságai

10. Tranzitivitás
    - $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$
    - $f \in \Omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \Omega(h)$
    - $f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h)$
    - $\varphi \prec g \wedge g \prec h \Rightarrow \varphi \prec h$
    - $f \succ g \wedge g \succ \psi \Rightarrow f \succ \psi$
11. Szimmetria
    - $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$
12. Reflexivitás
    - $f \in O(f) \wedge f \in \Omega(f) \wedge f \in \Theta(f)$

13. Következmény
    - **Θ ekvivalenciareláció**
    - AP függvények halmazának osztályozását adja
    - $f \in \Theta(g) \Leftrightarrow \Theta(f) = \Theta(g)$ (f függvény aszimptotikusan ekvivalens a g függvénnyel.)
    - Az egyes osztályok reprezentálhatók a legegyszerűbb függvényükkel, pl.: $\Theta(1)$, $\Theta(n)$, $\Theta(n^2)$, $\Theta(\sqrt{n})$

Példa AP függvények nagyságrendjére:

- $\log n \prec$
- $\sqrt{n}\ (n^{1/2}) \prec$
- $n \prec$
- $n \log n \prec$
- $n^2 \prec$
- $n^2 \log n \prec$
- $n^3 \prec$
- $2^n \prec$
- $n!$



További tulajdonságok

14. $f_1, g_1 \in \Theta(h_1) \land f_2, g_2 \in \Theta(h_2) \land f_1 \prec f_2 \Rightarrow g_2 \prec g_2$

15. **Definíció.**
    - $\Theta(f) \prec \Theta(g) \Leftrightarrow f \prec g$

16. Zártság
    - $f \in O(g) \land c \in \mathbb{P} \Rightarrow c * f \in O(g)$
    - $f \in O(h_1) \land g \in O(h_2) \Rightarrow f + g \in O(h_1 + h_2)$
    - $f \in O(h_1) \land g \in O(h_2) \Rightarrow f * g \in O(h_1 * h_2)$
    - $f \in O(g) \land \varphi \prec f \Rightarrow f + \varphi \in O(g)$

17. Felcserélt szimmetria
    - $f \in O(g) \Leftrightarrow g \in \Omega(f)$
    - $f \prec g \Leftrightarrow g \succ f$

18. Aszimmetria
    - $f \prec g \Rightarrow \neg(g \prec f)$
    - $f \succ g \Rightarrow \neg(g \succ f)$

19. A $\prec$ és a $\succ$ relációk irreflexívek.
    - $\neg(f \prec f)$
    - $\neg(f \succ f)$

A függvények aszimptotikus viszonya

20. Tétel.

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \prec g$
2. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in \mathbb{P} \Rightarrow f \in \Theta(g)$
3. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f \succ g$

- **Bizonyítás.**

1. a $\prec$ reláció definíciójából adódik
2. A $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$

   elég nagy *n*-re: $\left|\frac{f(n)}{g(n)} - c\right| < \frac{c}{2}$

   $\Rightarrow \frac{c}{2} < \frac{f(n)}{g(n)} < 2c$

   Mivel *g* AP, elég nagy *n*-ekre *g(n)* > o

   $\Rightarrow$ átszorozhatunk vele

   $\Rightarrow \frac{c}{2} * g(n) < f(n) < 2c * g(n)$

   $\Rightarrow f \in \Theta(g)$
3. a $\succ$ reláció definíciójából adódik

A függvények aszimptotikus viszonya

21. Következmény.

- $k \in \mathbb{N} \wedge$
- $a_0, a_1, \ldots, a_k \in \mathbb{R} \wedge$
- $a_k > 0 \Rightarrow$
- $\boldsymbol{a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \in \Theta(n^k)}$

- **Bizonyítás.**

$$\lim_{n\to\infty} \frac{a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0}{n^k} =$$

$$\lim_{n\to\infty} \left(\frac{a_k n^k}{n^k} + \frac{a_{k-1} n^{k-1}}{n^k} + \ldots + \frac{a_1 n}{n^k} + \frac{a_0}{n^k}\right) =$$

$$\lim_{n\to\infty} \left(a_k + \frac{a_{k-1}}{n} + \ldots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k}\right) =$$

$$\lim_{n\to\infty} a_k + \lim_{n\to\infty} \frac{a_{k-1}}{n} + \ldots + \lim_{n\to\infty} \frac{a_1}{n^{k-1}} + \lim_{n\to\infty} \frac{a_0}{n^k}) =$$

$$a_k + 0 + \cdots + 0 + 0 = a_k \in \mathbb{P} \Longrightarrow$$

$$a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \in \theta(n^k)$$

L'Hospital szabály és következménye

22. L'Hospital szabály.

- Ha elég nagy helyettesítési értékekre az f és g függvények valós kiterjesztése differenciálható, valamint
- $\lim_{n\to\infty} f(n) = \infty \ \wedge \ \lim_{n\to\infty} g(n) = \infty \ \wedge$
- $\exists \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \Longrightarrow$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$

23. Következmény. (20. és 22. alapján)

- $c, d \in \mathbb{R} \wedge c < d \Rightarrow n^c \prec n^d$
- $c, d \in \mathbb{P}_0 \wedge c < d \Rightarrow c^n \prec d^n$
- $c, d \in \mathbb{R} \wedge d > 1 \Rightarrow n^c \prec d^n$
- $d \in \mathbb{P}_0 \Rightarrow d^n \prec n! \prec n^n$
- $c, d \in \mathbb{P} \wedge c, d > 1 \Rightarrow \log_c n \in \Theta(\log_d n)$
- $\varepsilon \in \mathbb{P} \Rightarrow \log n \prec n^\varepsilon$
- $c \in \mathbb{R} \wedge \varepsilon \in \mathbb{P} \Rightarrow n^c \log n \prec n^{c+\varepsilon}$

Bizonyítás és következmény

- **Bizonyítás.** $\varepsilon \in \mathbb{P} \Rightarrow \log n \prec n^{\varepsilon}$ (L'Hospital szabály alkalmazásával)
  - $\lim_{n\to\infty} \frac{\log n}{n^{\varepsilon}} = \log e \lim_{n\to\infty} \frac{\ln n}{n^{\varepsilon}} = \log e \lim_{n\to\infty} \frac{\ln' n}{(n^{\varepsilon})'} =$

    $\frac{\log e}{\varepsilon} \lim_{n\to\infty} \frac{\frac{1}{n}}{n^{\varepsilon-1}} = \frac{\log e}{\varepsilon} \lim_{n\to\infty} \frac{\frac{1}{n}}{n^{\varepsilon}} = \frac{\log e}{\varepsilon} 0 = 0$

24. Következmény.

- $\Theta(\log n) \prec \Theta(n) \prec \Theta(n * \log n) \prec \Theta(n^2) \prec \Theta(n^2 * \log n) \prec \Theta(n^3)$

További tételek

25. Tétel. $f \in O(g) \Leftrightarrow \exists d \in \mathbb{P}$ és $\exists \psi \prec g$, hogy elég nagy n-ekre

$$d * g(n) + \psi(n) \geq f(n)$$

26. Tétel. $f \in \Omega(g) \Leftrightarrow \exists c \in \mathbb{P}$ és $\exists \varphi \prec g$, hogy elég nagy n-ekre

$$c * g(n) + \varphi(n) \leq f(n)$$

27. Tétel. $f \in \Theta(g) \Leftrightarrow \exists c, d \in \mathbb{P}$ és $\exists \varphi, \psi \prec g$, hogy elég nagy n-ekre

$$c * g(n) + \varphi(n) \leq f(n) \leq d * g(n) + \psi(n)$$

$\mathbb{N} \times \mathbb{N}$ értelmezési tartományú függvények

28. Definíció. $g : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ függvény *AP*, ha elég nagy n és elég nagy m értékekre $g(n,m) > 0$.

29. Megjegyzés. A továbbiakban az egyszerűség kedvéért feltesszük, hogy f, g, $h : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ *AP* függvényeket jelölnek.

30. Definíció. $O(g) = \{f \mid \exists d \in \mathbb{P}$, hogy $f(n,m) \leq d*g(n,m)$, tetszőleges elég nagy n és elég nagy m értékekre$\}$.

$\mathbb{N} \times \mathbb{N}$ értelmezési tartományú függvények

31. Definíció. $\Omega(g) = \{f \mid \exists c \in \mathbb{P}$, hogy $f(n,m) \geq c * g(n,m)$, tetszőleges elég nagy n és elég nagy m értékekre$\}$.

32. Definíció. $\Theta(g) = \{f \mid \exists c, d \in \mathbb{P}$, hogy $c * g(n,m) \leq f(n,m) \leq d * g(n,m)$, tetszőleges elég nagy n és elég nagy m értékekre$\}$.

33. Megjegyzés. A korábban a természetes számokon értelmezett függvényekre vonatkozó tételek az itt tárgyaltakra természetes módon általánosíthatók.

Ellenőrző kérdések

1. Tegyük fel, hogy $f : \mathbb{N} \to \mathbb{R}$, aszimptotikusan nemnegatív függvény! Adja meg az $O(f)$, az $\Omega(f)$ és a $\Theta(f)$ függvényhalmazok definícióját!
2. Milyen alapvető összefüggést ismer az $O(f)$, az $\Omega(g)$ és a $\Theta(f)$ függvényhalmazok között?
3. Igaz-e, hogy $(5n + 1)(2n-3) \in \Theta(n^2)$? Miért?
4. Igaz-e, hogy $2^n \in O(n!)$? Miért?
5. Igaz-e, hogy $2^n \in \Omega(n!)$? Miért?

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek I. előadásjegyzete alapján készült.

A képek forrása:

http://tamop412.elte.hu/tananyagok/algoritmusok/lecke2_lap1.html#hiv7

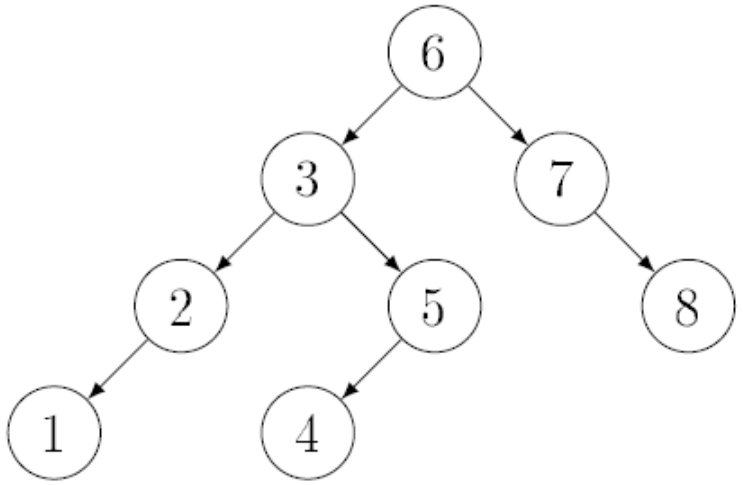
Algoritmusok és adatszerkezetek I.
6. Előadás

Bináris fák

Tartalom

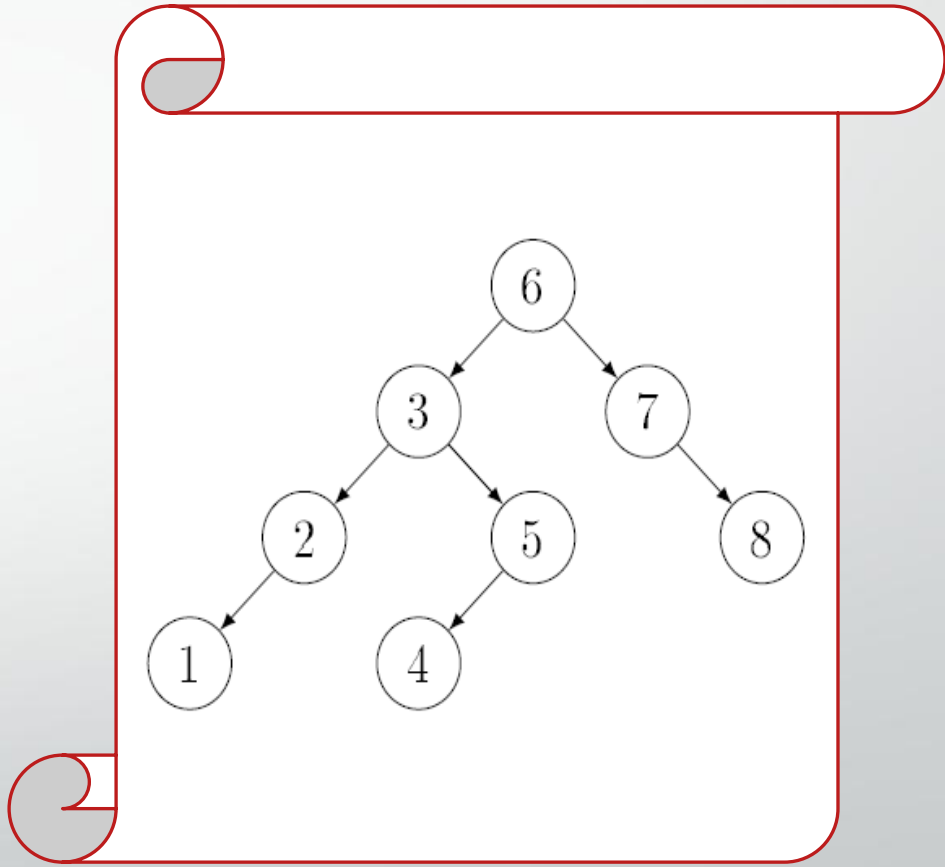
Bináris fa

- Felhasználása
  - Nagy méretű adathalmazok és multihalmazok (zsákok) ábrázolására
  - egyéb adatreprezentációs célokra
- Fogalmak:
  - *Csúcs* (node)
  - (max 2.) rákövetkező: *bal* (left) / *jobb* (right)
  - *gyerek* (child) <-> *szülő* (parent)
  - *testvér* (sibling)
  - *levél* (leaf) : (1,4,8)
    - nincs gyereke

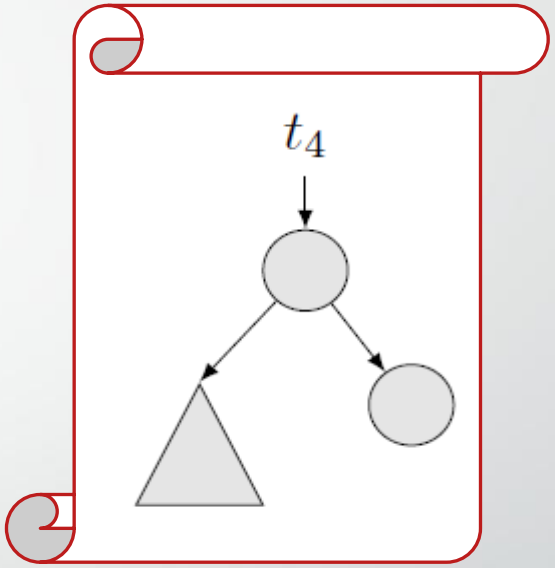


Bináris fa

- Fogalmak:
  - *Gyökér* (root): (6)
    - nincs szülője
  - *Belső csúcs* (internal node):
    - nem-levél csúcs
  - csúcs *leszármazottai* (descendants):
    - a gyerekei és a gyerekei leszármazottai.
  - csúcs *ősei* (ancestors):
    - a szülője és a szülője ősei.
- Fák ábrázolása:
  - felülről lefelé:



Általános megjegyzések



- Jelölés
  - Konkrét elem: kör
  - Ha nem fontos az adott részfa szerkezete: háromszög
- ⦸ *üres fa* (empty tree):
  - nincs csúcsa
- Egy tetszőleges nemüres *t* fát a gyökércsúcsa ($*t$) határoz meg
  - A fa többi csúcsa: ennek a leszármazottja

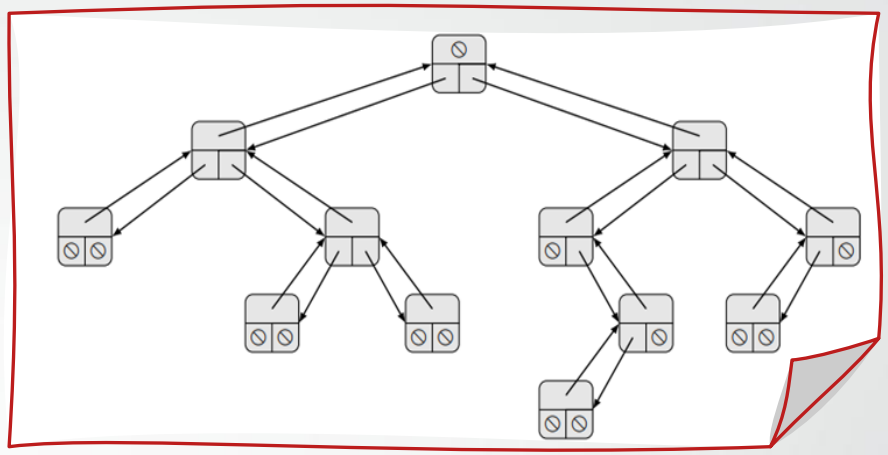
Általános megjegyzések

- A $*t$ bal/jobb gyerekéhez tartozó fák:
  - A t bal/jobb részfájának nevezzük
  - Jelölése: $t \longrightarrow left$ / $t \longrightarrow right$ ($left(t)$ és $right(t)$)
- Ha a gyerek létezik:
  - $*t \longrightarrow left$ / $*t \longrightarrow right$
  - (Itt a „$\longrightarrow$” erősebben köt, mint a „$*$”. Pl. $*t \longrightarrow left = *(t \longrightarrow left)$)
  - Ha $*t$-nek nincs bal/jobb gyereke:
    - $t \longrightarrow left = \oslash$ / $t \longrightarrow right = \oslash$

Általános megjegyzések

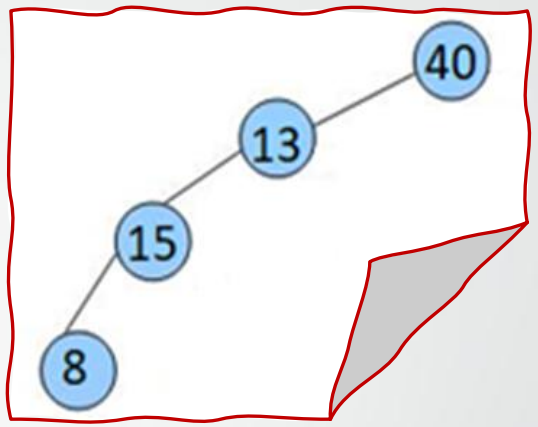
- A *t* bináris fának (*t* = ⦸ esetén is) *részfái* (subtree)
  - Önmaga
  - Ha *t* ≠ ⦸
    - $t \rightarrow left$ és $t \rightarrow right$
    - ezek részfái
- A *t valódi részfája* (proper subtree) *f*
  - *t* részfája *f*
  - *t*≠ *f*≠ ⦸

Általános megjegyzések

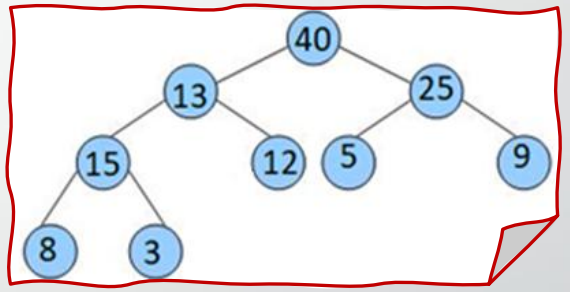


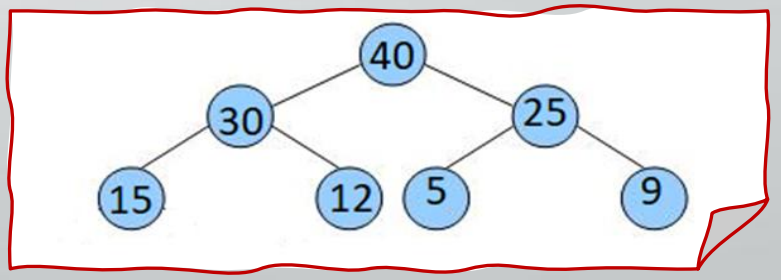
- A $*t$-ben tárolt kulcs
  - Jelölése: $t \rightarrow key$ (illetve $key(t)$).
  - Általában csak a csúcsban tárolt adat egy kis része, vagy abból egy függvény segítségével számítható ki
- Ha $*g$ egy fa egy csúcsa
  - a szülője a $*g \rightarrow parent$
  - A szülőjéhez, mint gyökércsúcshoz tartozó fa a $g \rightarrow parent$ (illetve $parent(g)$).
- Ha $*g$ a teljes fa gyökere
  - $g \rightarrow parent = \oslash$

Speciális bináris fák



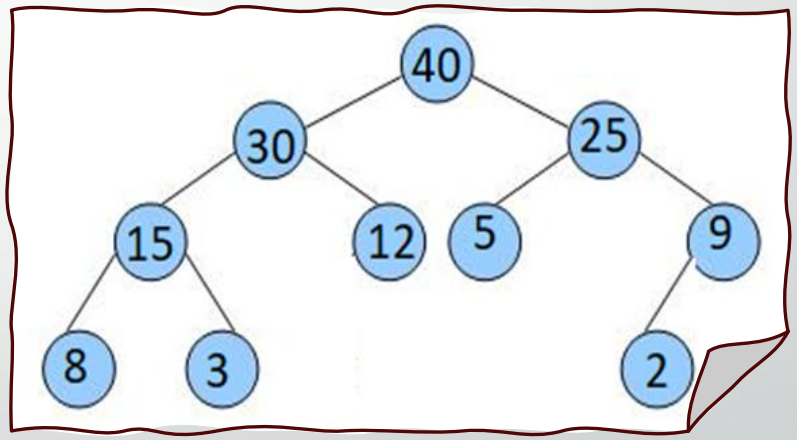
- **Listává torzult fa**: Azok a fákat, amelyekben minden belső (azaz nem-levél) csúcsnak egy gyereke van
- **Szigorúan bináris fa** (strictly binary tree vagy full binary tree): Azok a bináris fák, amelyekben minden belső (azaz nem-levél) csúcsnak két gyereke van
- **Tökéletes bináris fa** (perfect binary tree): Ha a szigorúan bináris fának minden levele azonos szinten van
  - Tetszőleges *h* magasságú teljes bináris fa csúcsainak száma: $1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$.
  - Az üres fa is tökéletes





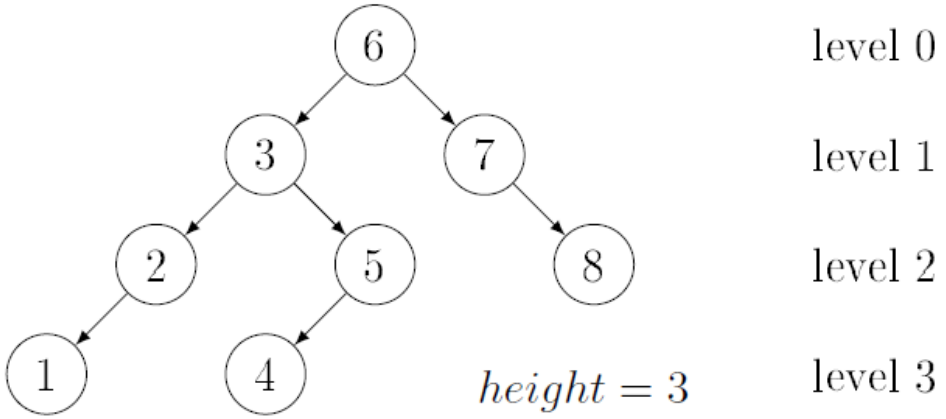
Speciális bináris fák

- **Majdnem teljes bináris fa** (nearly complete binary tree): Ha egy tökéletes bináris fa levélszintjéről nulla, egy vagy több levelet elveszünk, de nem az összeset
  - Az üres fa is
  - Minden tökéletes bináris fa egyben majdnem teljes is (fordítva viszont nem igaz).
  - Tetszőleges ***h* mélységű**, nemüres, majdnem teljes bináris fa **csúcsainak száma**:
    - $n \in [2^h..2^{h+1})$ (a fenti definíció szerint)
    - ➢ $h = [\log n]$.
  - Az alsó szinten levő leveleket elvéve
    - $h-1$ mélységű tökéletes bináris fát kapunk.



Bináris fa mérete, szintjei, magassága

- Bináris fa *mérete* (size): csúcsainak száma
  - A *t* bináris fa méretének jelölése:
    - $|t|$
    - $n(t)$
    - n (ha egyértelmű)
- Fa *szintjei* (levels)
  - A gyökér: nulladik szint
  - Az *i*-edik szintű csúcsok gyerekeit az $(i + 1)$-edik szinten találjuk.
- A fa *magassága* (height) (~mélysége)
  - A legmélyebben fekvő leveleinek szintszáma.
  - A *t* bináris fa magasságának jelölése: $h(t)$, vagy h (ha egyértelmű)
  - Az üres fa magassága $h(\varnothing) = -1$.
  - Tetszőleges nemüres *t* bináris fára: $h(t) = 1 + \max(h(t \rightarrow left), h(t \rightarrow right))$



Bináris fa magasságáról szóló tétel

1. Tétel. Tetszőleges $n > 0$ méretű és $h \geq 0$ magasságú (azaz nemüres) bináris fára:

$$\lfloor \log n \rfloor \leq h \leq n - 1$$

- **Bizonyítás.**
- $\lfloor \log n \rfloor \leq h$
  - A h mélységű bináris fák között a legtöbb csúcs: tökéletes bináris fának van: $n = 2^{h+1} - 1$
  - tetszőleges bináris fára: $n < 2^{h+1}$
  - $n > 0 \Rightarrow \log n < \log 2^{h+1} = h + 1$
  - $\lfloor \log n \rfloor \leq h$

Bináris fa magasságáról szóló tétel

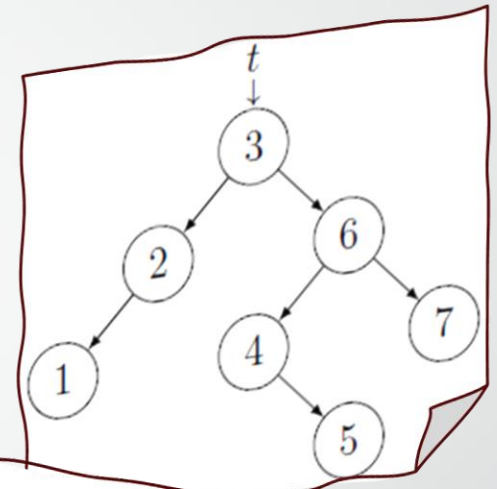
- $h \leq n-1$
  - tetszőleges h magasságú fa szintjeit 0-tól h-ig sorszámoztuk: $h + 1$ szint
  - Mivel a fa minden szintjén van legalább egy csúcs
  - tetszőleges fára $n \geq h + 1$
  - $h \leq n - 1$
  - $h = n - 1$ pontosan akkor teljesül, ha a fa listává torzult.

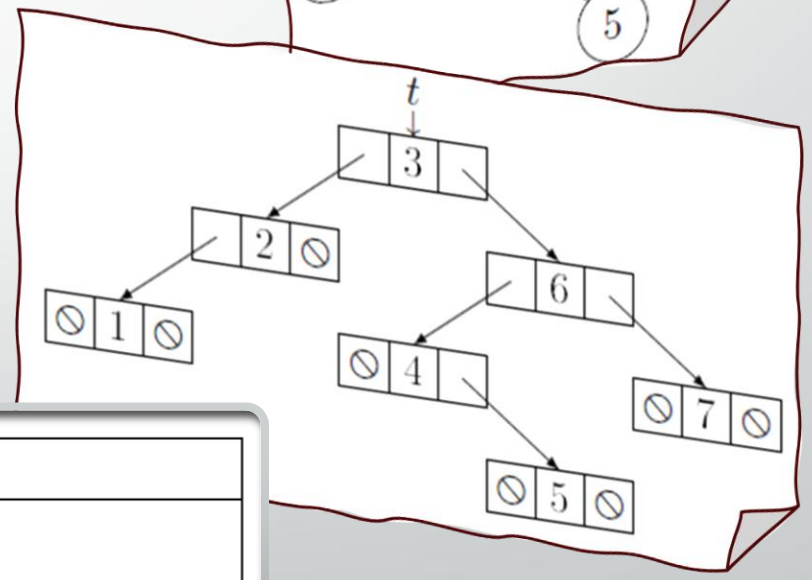
Bináris fák reprezentációi

- Bináris fák láncolt (linked) ábrázolása
- Bináris fák zárójelezett, szöveges formája
- Bináris fák aritmetikai ábrázolása

Bináris fák láncolt ábrázolásai

- Az üres fa reprezentációja: $\oslash$ pointer, jelölése ~ absztrakt fáknál
- BinTree absztrakt típus reprezentációja: Node*



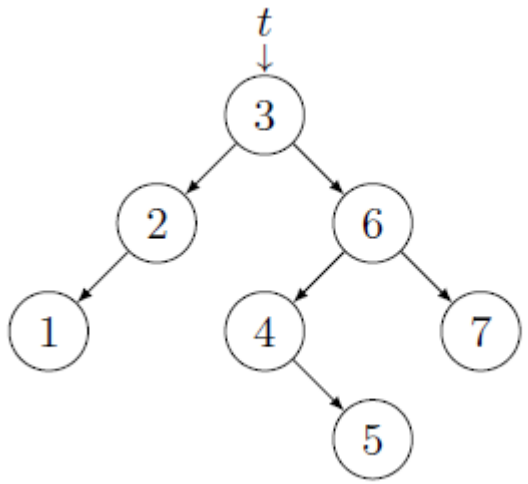


Node
+ $key : \mathcal{T}$ // $\mathcal{T}$ valamilyen ismert típus + $left, right$: Node*
+ Node() { $left := right := \oslash$ } // egycsúcsú fát képez belőle + Node($x : \mathcal{T}$) { $left := right := \oslash$; $key := x$ }

Node3
+ $key : \mathcal{T}$ // $\mathcal{T}$ valamilyen ismert típus + $left, right, parent$: Node3*
+ Node3(p:Node3*) { $left := right := \oslash$; $parent := p$ } + Node3($x : \mathcal{T}$, p:Node3*) { $left := right := \oslash$; $parent := p$; $key := x$ }

Bináris fák zárójelezett, szöveges formája

- Tetszőleges nemüres bináris fa zárójeles, azaz szöveges alakja:
  - (balRészFa Gyökér jobbRészFa)

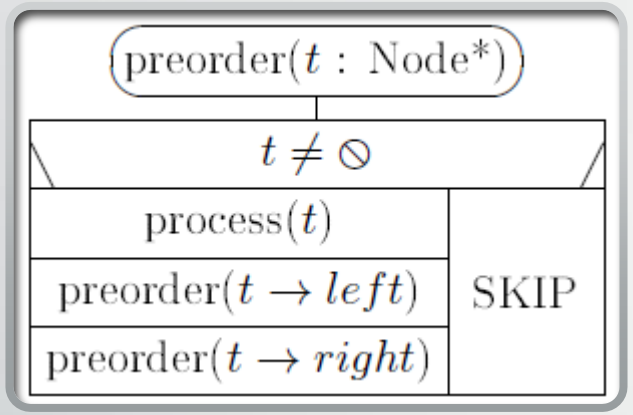


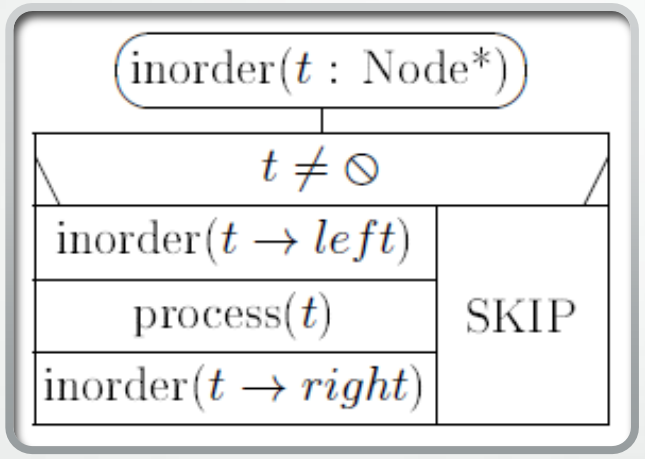
A balra látható t bináris fa

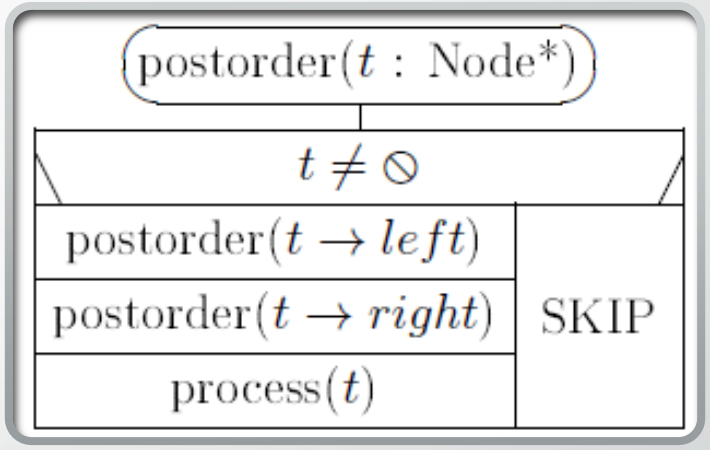
- egyszerű zárójeles alakja:
  (((1) 2 ()) 3 ((() 4 (5)) 6 (7)))
- elegáns zárójeles alakja:
  { [(1) 2 ()] 3 [(⟨⟩ 4 ⟨5⟩) 6 (7)] }

- ha az aktuális *l* szintre
  - *l* mod 4 = 0 => { }
  - *l* mod 4 = 1 => [ ]
  - *l* mod 4 = 2 => ( )
  - *l* mod 4 = 3 => ⟨ ⟩
    zárójeleket használhatunk.

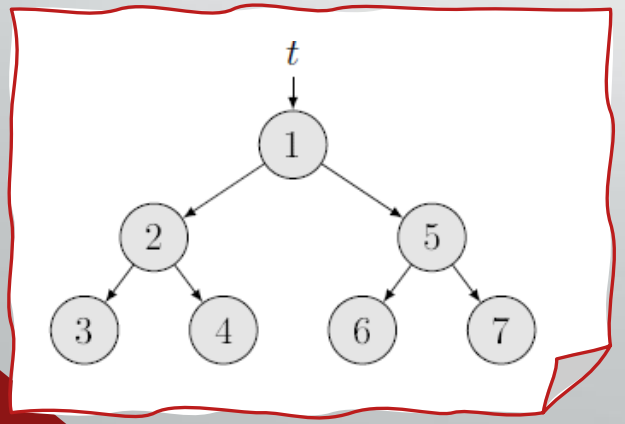
Bináris fák rekurzív bejárásai
(Binary tree recursive traversals)

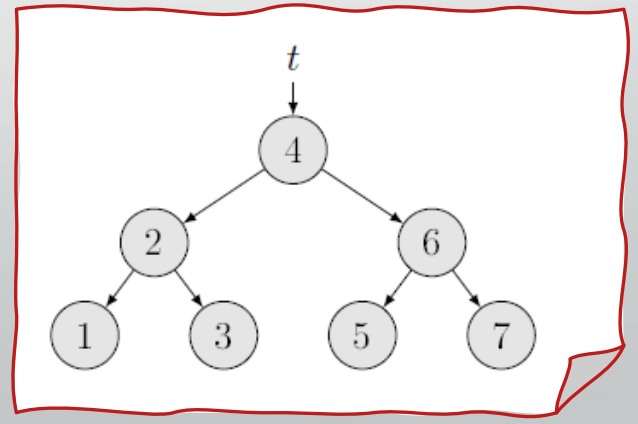


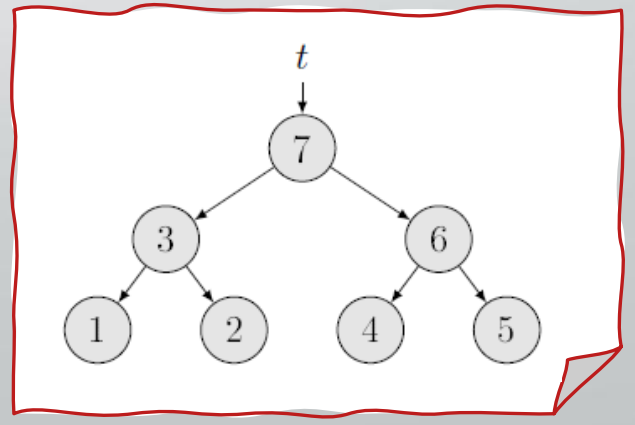




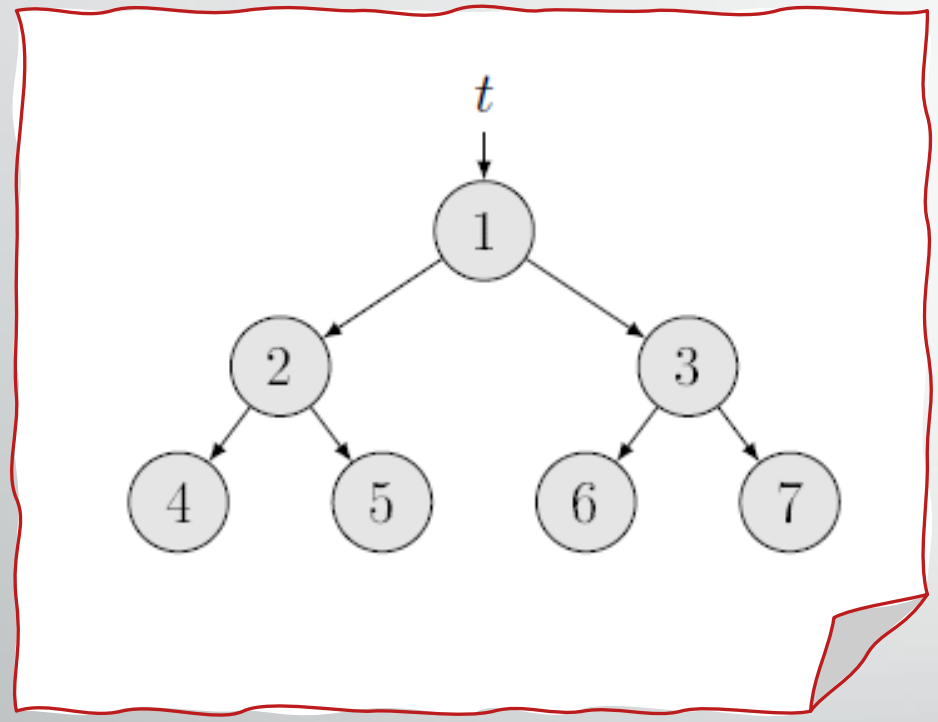
- Nevek: gyökércsúcsot a részfákhoz képest mikor dolgozzák fel.

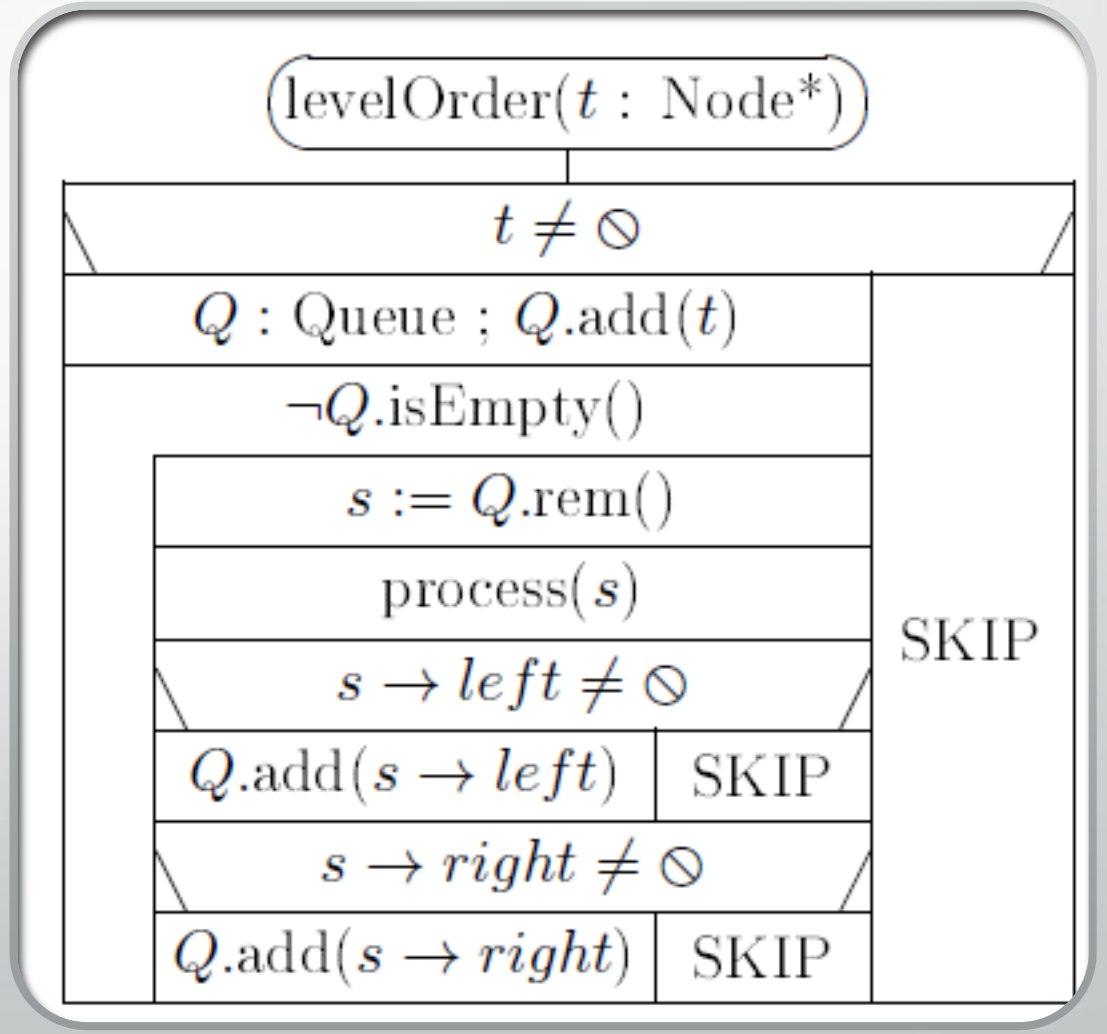






Bináris fa szintenkénti bejárása
(Binary tree Breadth First or Level Order traversal)





Bináris fa C# kód*

- Node osztály megvalósítása

```
public class Node<T>
{
    public T Key { get; set; }
    public Node<T>? Left { get; set; }
    public Node<T>? Right { get; set; }

    public Node()
    {
        Left = Right = null;
    }

    public Node(T key)
    {
        Key = key;
        Left = Right = null;
    }
}
```

- Bináris fa iteratív bejárása

```
public static void LevelOrder<T>(Node<T>? node, Action<T> process)
{
    if (node == null) return;

    Queue<Node<T>> queue = new Queue<Node<T>>();
    queue.Enqueue(node);

    while (queue.Count > 0)
    {
        Node<T> current = queue.Dequeue();
        process(current.Key);

        if (current.Left != null) queue.Enqueue(current.Left);
        if (current.Right != null) queue.Enqueue(current.Right);
    }
}
```

Bináris fa C# kód*

- Bináris fa rekurzív bejárások

```
public static void Preorder<T>(Node<T>? node, Action<T> process)
{
    if (node != null)
    {
        process(node.Key);
        Preorder(node.Left, process);
        Preorder(node.Right, process);
    }
}
```

```
public static void Inorder<T>(Node<T>? node, Action<T> process)
{
    if (node != null)
    {
        Inorder(node.Left, process);
        process(node.Key);
        Inorder(node.Right, process);
    }
}
```

```
public static void Postorder<T>(Node<T>? node, Action<T> process)
{
    if (node != null)
    {
        Postorder(node.Left, process);
        Postorder(node.Right, process);
        process(node.Key);
    }
}
```

Bináris fák bejárásai

- Üres fára mindegyik bejárása
  - üres program
- BinTree
  - a bináris fák absztrakt típusa
- A struktogramokban a $*t$ csúcs feldolgozása:
  - process(t)
  - $T(\text{process}(t)) \in \Theta(1)$
- **Állítás**:

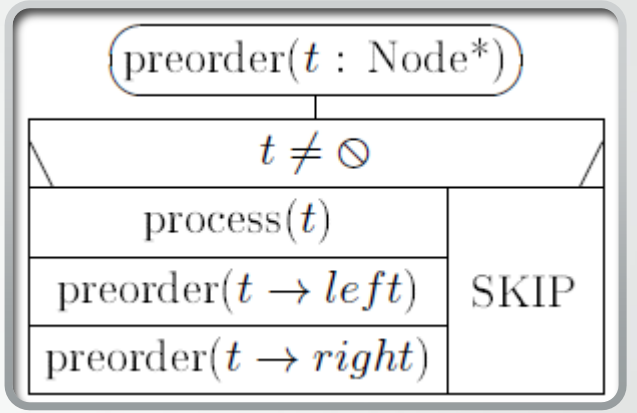
$$T_{\text{preorder}}(n),\ T_{\text{inorder}}(n),\ T_{\text{postorder}}(n),\ T_{\text{levelOrder}}(n) \in \Theta(n)$$

ahol n a fa mérete (csúcsainak száma)

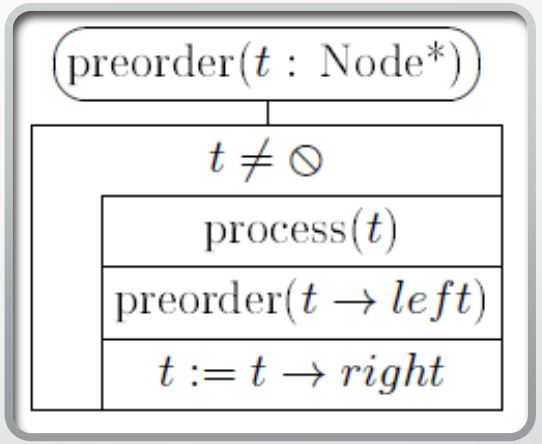
Bináris fák bejárásai

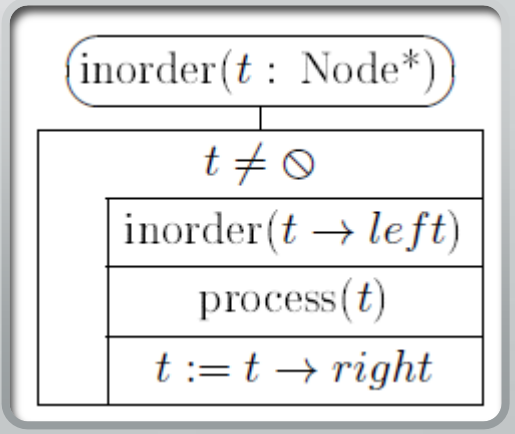
- **Igazolás**:
- Rekurzív bejárások
  - Hívások száma= Bináris fa részfáinak száma (az üres részfák is)
    - *n* szerinti teljes indukcióval belátható:
      - tetszőleges *n* csúcsú bináris fának $2n + 1$ részfája van.
  - egy-egy hívás végrehajtása: $\Theta(1)$
- A szintenkénti bejárás:
  - minden csúcsnál: a ciklus egy-egy végrehajtása
  - egy fa csúcsainak szintenkénti felsorolásában a korábbi csúcs gyerekei megelőzik a későbbi csúcs gyerekeit
    - ✓ akár egy szinten, akár különböző szinten van a két csúcs a fában

Bejárások hatékonyságának javítása



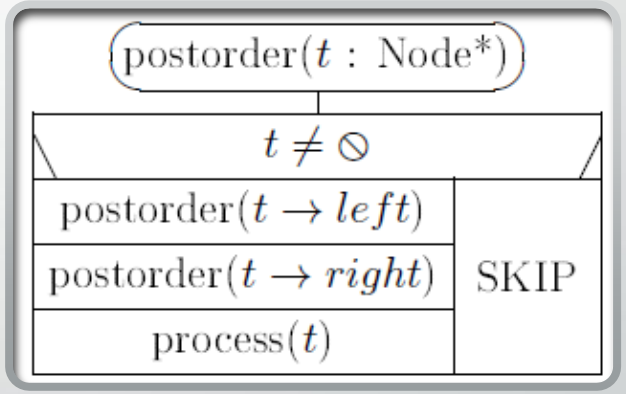
- A preorder és az inorder bejárások hatékonysága konstans szorzóval javítható
  - A végrekurziókat ciklussá alakítjuk
- A *t* paramétert érték szerint adjuk át
  - ➢ az aktuális paraméter nem változik meg
- [Általában nem célszerű cím szerint átvett formális paramétereket ciklusváltozóként vagy segédváltozóként használni]

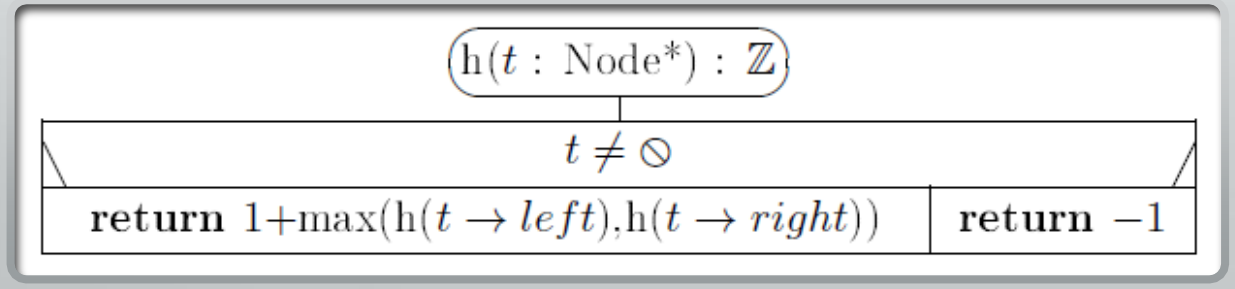




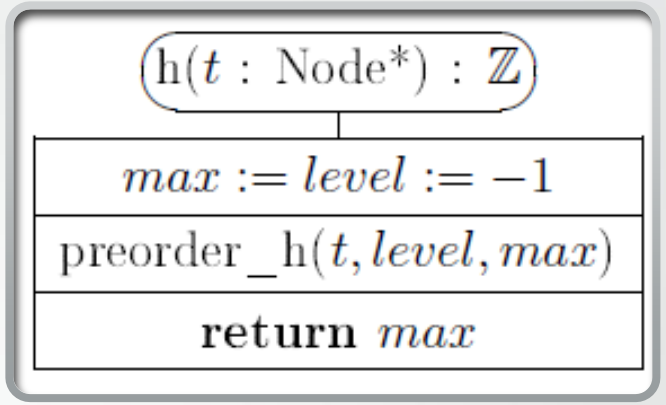
Fabejárások alkalmazása: Bináris fa magassága

- Legegyszerűbb
  - definíció lekódolása
  - ~postorder bejárás

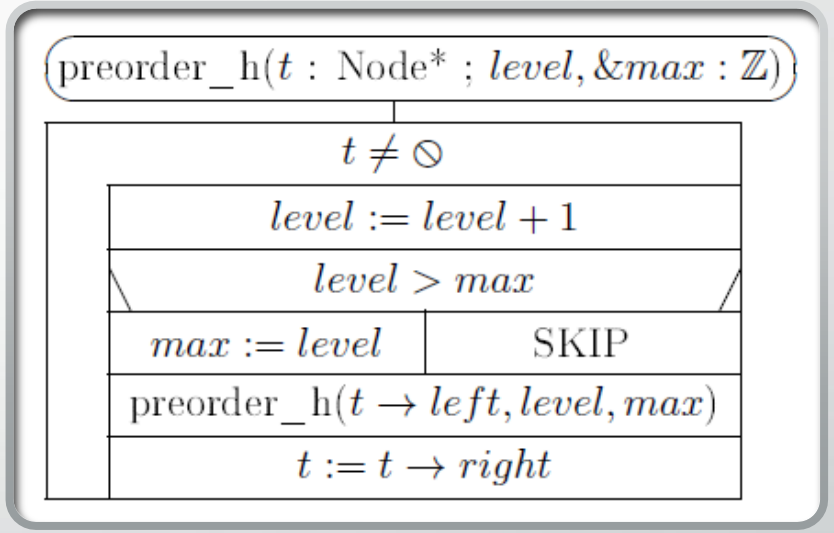


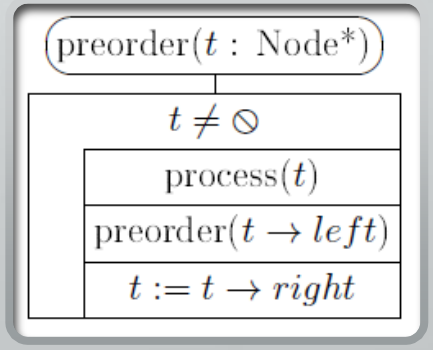


Fabejárások alkalmazása: bináris fa magassága



- Preorder bejárással
  - \+ egy nemrekurzív keret
    - a legtöbb rekurzív programnál
    - előkészíti a legkülső rekurzív hívást
    - a végén is biztosítja a megfelelő interfészt
  - \+ két extra paraméter
    - *level*: tárolja, milyen mélyen (hányadik szinten) járunk a fában
    - *max*: tárolja, mi a legmélyebb szint, ahol eddig jártunk
  - bejárás és a maximumkeresés összefésülése



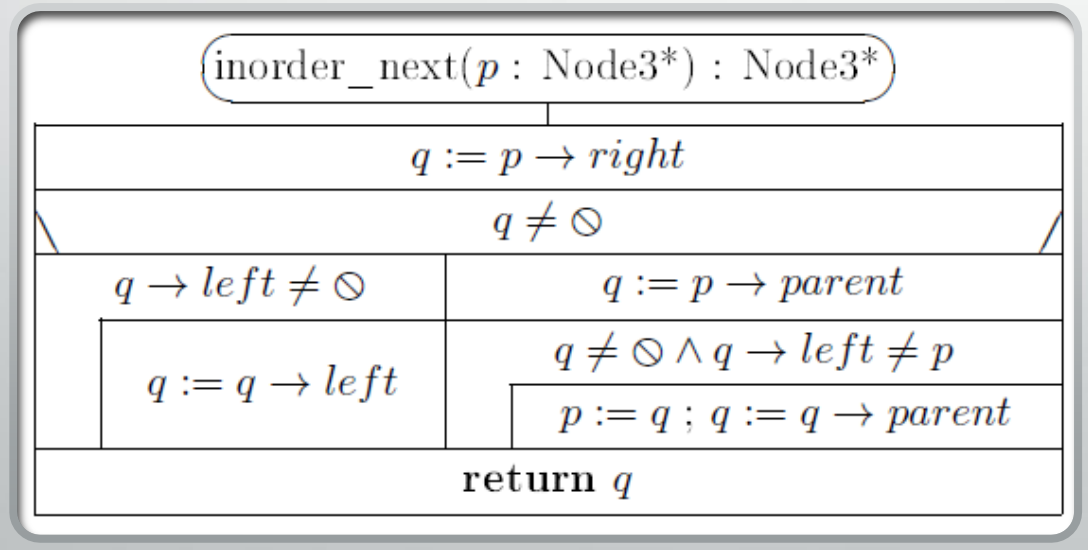


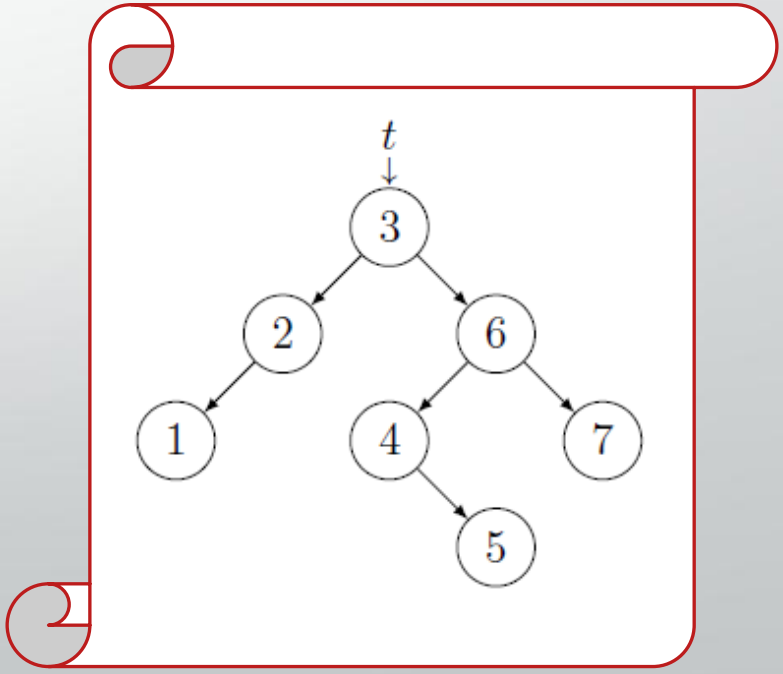
Példa a *parent* pointer szükségességére

Feladat. Írjuk meg az inorder_next(p) függvényt

- a $*p$ csúcs inorder bejárás szerinti rákövetkezőjének címét adja vissza
- ha nincs ilyen, $\oslash$-t!

- **Műveletidő**: $MT(h(t)) \in O(h(t))$, ahol t a $*p$ csúcsot tartalmazó bináris fa
- ~ inorder_megel(p): inorder bejárás szerinti megelőzője





Bináris fa C# kód*

- Node3 osztály megvalósítása

```
class Node3<T>
{
    public T Key;
    public Node3<T> Left, Right, Parent;

    public Node3(Node3<T> p)
    {
        Left = Right = null;
        Parent = p;
    }

    public Node3(T x, Node3<T> p)
    {
        Key = x;
        Left = Right = null;
        Parent = p;
    }
}
```

- InorderNext

```
class BinaryTree
{
    public static Node3<T> InorderNext<T>(Node3<T> p)
    {
        if (p == null) return null;

        Node3<T> q = p.Right;
        if (q != null)
        {
            while (q.Left != null)
            {
                q = q.Left;
            }
            return q;
        }

        q = p.Parent;
        while (q != null && q.Left != p)
        {
            p = q;
            q = q.Parent;
        }
        return q;
    }
```

```
public static void Main(string[] args)
{
    Node3<int> root = new Node3<int>(10, null);
    root.Left = new Node3<int>(5, root);
    root.Right = new Node3<int>(15, root);
    root.Left.Left = new Node3<int>(3, root.Left);
    root.Left.Right = new Node3<int>(7, root.Left);
    root.Right.Right = new Node3<int>(18, root.Right);

    Node3<int> next = InorderNext(root.Left);
    Console.WriteLine(next != null ? next.Key.ToString() : "null"
}
```

Bináris fa általánosítása: *r*-áris fa

- A fában egy tetszőleges csúcsnak legfeljebb *r* rákövetkezője van
- A hozzájuk tartozó részfákat:
  - [0..r)-beli *szelektorokkal* sorszámozzuk
- Ha egy csúcsnak nincs *i*-edik gyereke ($i \in [0..r)$)
  - az *i*-edik részfa üres.
- Bináris fa ~ 2-áris fa
  - (*left* ~ 0 és a *right* ~ 1)

Nemüres *r-áris* fák bejárása

- Preorder bejárás:
  1. a fa gyökerét dolgozza fel
  2. sorban bejárja a 0..*r* – 1-edik részfákat
- Postorder bejárás:
  1. sorban bejárja a 0..*r* – 1-edik részfákat
  2. a fa gyökerét csak a részfák után dolgozza fel

- Inorder bejárás:
  1. bejárja a nulladik részfát
  2. a fa gyökerét dolgozza fel
  3. sorban bejárja az 1..*r* – 1-edik részfákat
- Szintenkénti bejárás:
  - a csúcsokat a gyökértől kezdve szintenként
  - minden szintet balról jobbra bejárva dolgozza fel

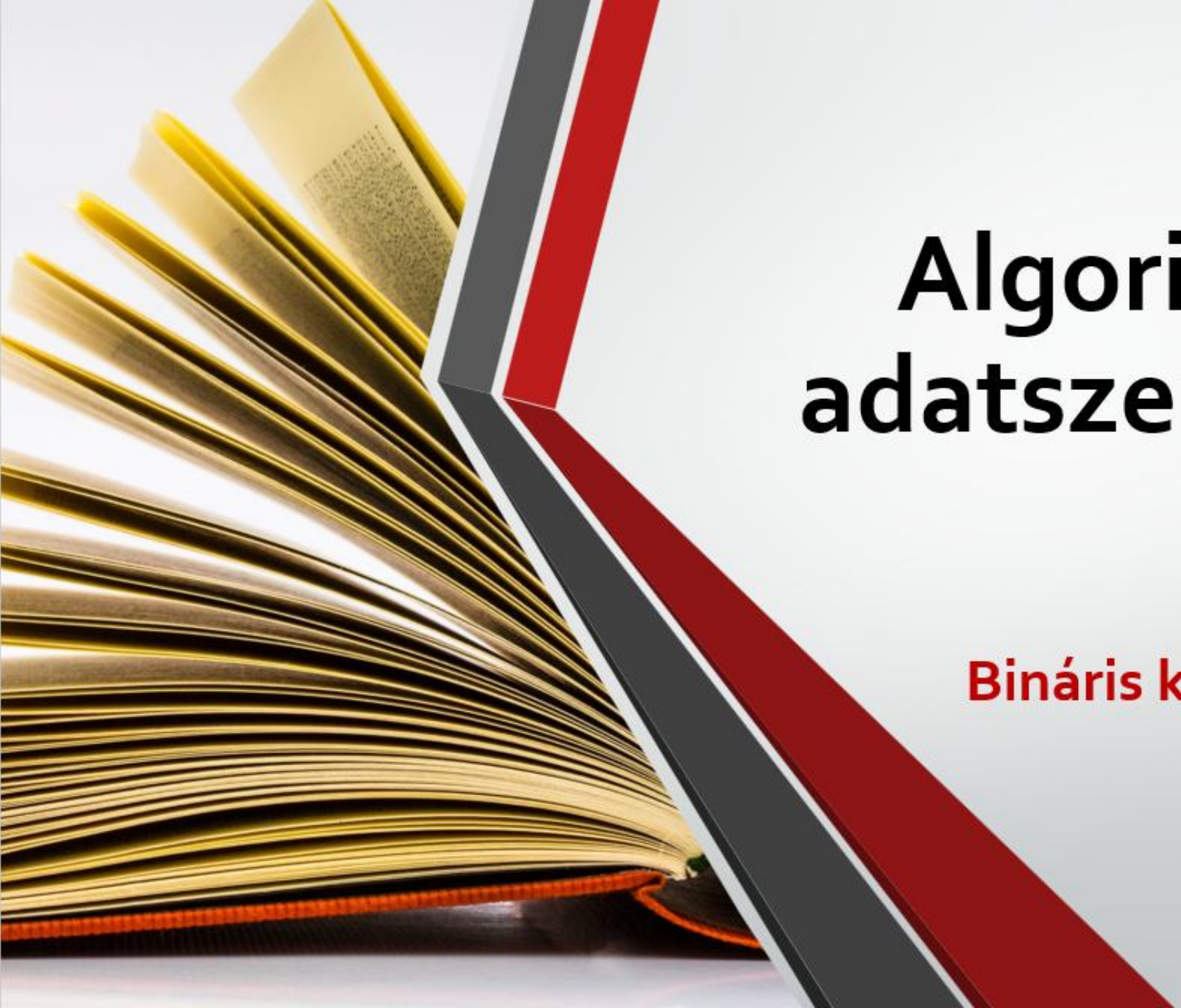
Ellenőrző kérdések

- Milyen speciális bináris fákat ismer?
  - Adja meg a definícióikat!
- Mennyi a bináris fa mérete, illetve magassága?
- Mondja ki és bizonyítsa be a bináris fa magasságáról szóló tételt!
- Milyen bináris fa reprezentációkat ismer?
- Milyen bináris fa bejárásokat ismer?
  - Adja meg a struktogramjaikat!
  - Számolja ki a struktogramokhoz tartozó műveletigényeket!
- Mit jelent az *r*-áris fa?

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek I. előadásjegyzete alapján készült.



Algoritmusok és adatszerkezetek I.
7. Előadás

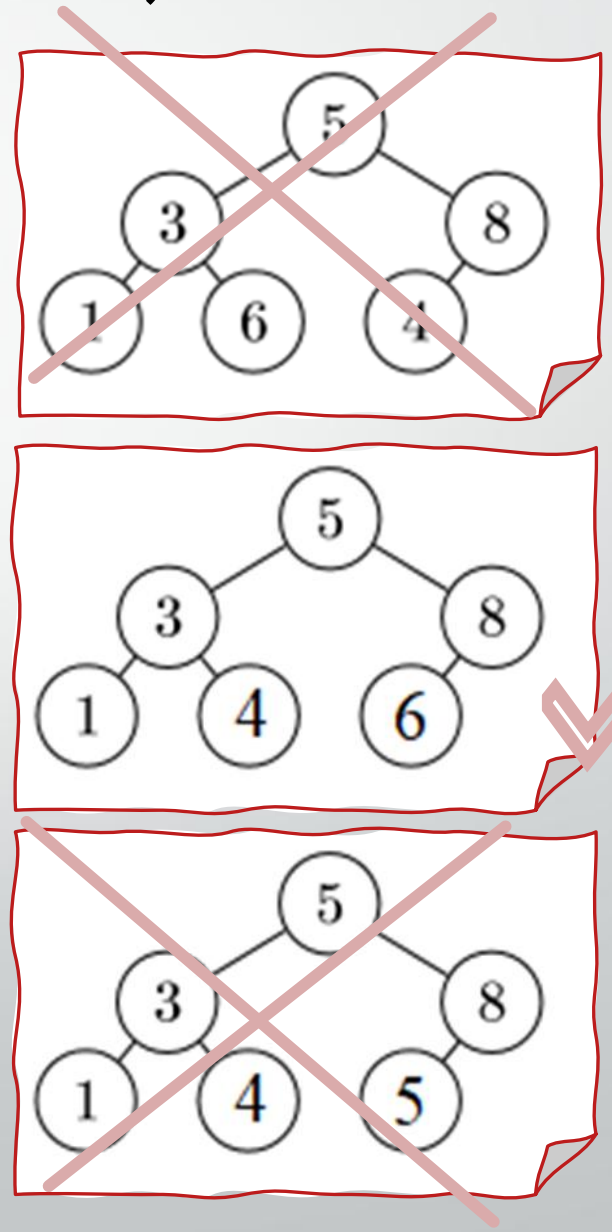
Bináris keresőfa, prioritásos sor, kupac

Tartalom

Bináris keresőfa (Binary search tree)

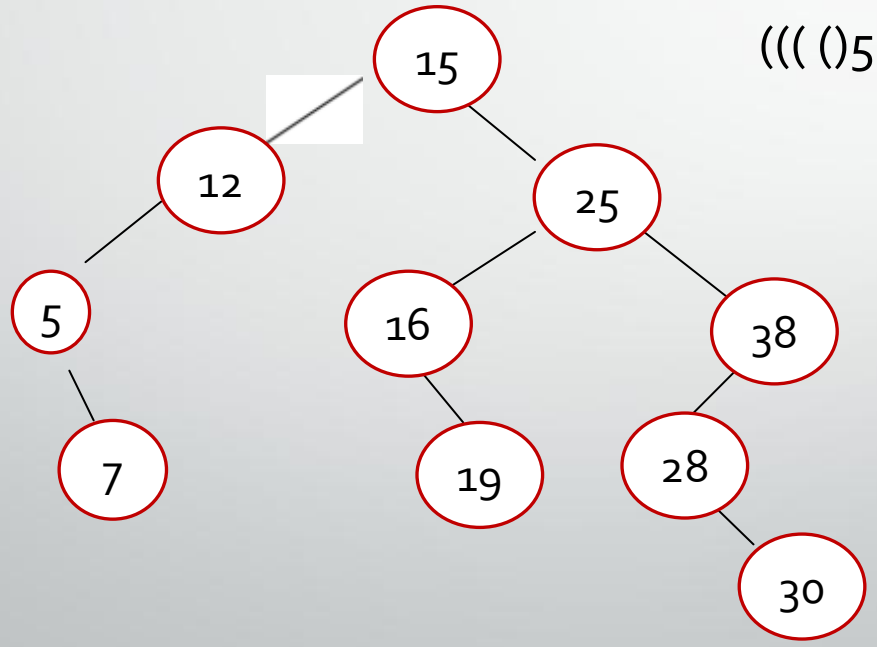
- Egy bináris fát **keresőfának** nevezünk,
  ha minden belső csúcsára és annak *y* kulcsára igaz:
  - A csúcs bal részfájában minden csúcs *x* kulcsára x<*y*,
  - A csúcs jobb részfájában minden csúcs z kulcsára *z*>*y*
- Rendezőfa (binary sort tree):
  ha megengedjük az egyenlőséget
  - Egy bináris fát **rendezőfának** nevezünk, ha minden belső csúcsára és annak *y* kulcsára igaz:
    - A csúcs bal részfájában minden csúcs *x* kulcsára x≤*y*,
    - A csúcs jobb részfájában minden csúcs z kulcsára *z*≥*y*

Bináris fa zárójelezett alakja:
(((1) 3 (4)) 5 ((6) 8 ()))



Bináris keresőfa építése

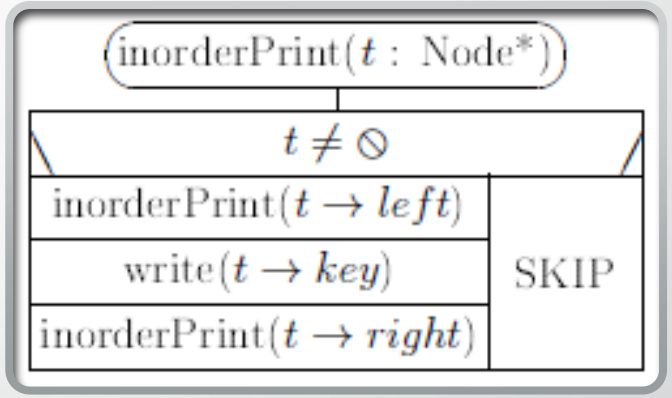
- Építsünk keresőfát a következő számokból:
- 15, 12, 25, 5, 16, 7, 38, 19, 28, 30



Bináris fa zárójelezett alakja:
((( )5 (7)) 12 ()) 15 ((() 16 (19)) 25 ((() 28 (30)) 38 ())))

Keresőfák~ halmazok, sorozatok reprezentációi

A bináris keresőfák tekinthetők véges halmazok, vagy szigorúan monoton növekvő sorozatok reprezentációinak.

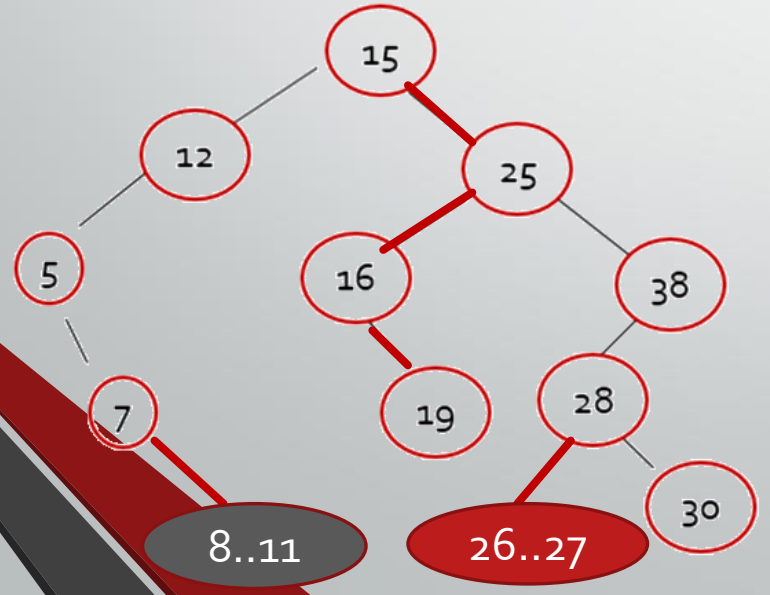


- Jelölje $H(t)$ a t bináris keresőfa által reprezentált halmazt! Ekkor definíció szerint:
  - $H(⦸) = \{\}$
  - $H(t) = H(t \rightarrow left) \cup \{t \rightarrow key\} \cup H(t \rightarrow right)$, amennyiben $t \neq ⦸$
- A *t* bináris keresőfa által reprezentált sorozatot megkaphatjuk, ha Inorder bejárással kiíratjuk a fa csúcsainak tartalmát:
  - A *t* bináris keresőfa ⟺ ha a fa kulcsait szigorúan monoton növekvő sorrendben írja ki!
  - Következmény: amennyiben egy bináris fa transzformáció a fa inorder bejárását nem változtatja meg, és adott egy bináris keresőfa, akkor a fa a transzformáció végrehajtása után is bináris keresőfa marad (Keresőfák kiegyensúlyozásánál (pl. AVL fa) használjuk fel)

Bináris keresőfa alapműveletei:

- *search(t, k):* A t fában megkeresi a k kulcsú elemet, ∅ -t ad vissza, ha nincs benne.
- *insert(t, k)*: Beszúrja a k kulcsú elemet, ha még nincs a fában.
- *del(t, k)*: Törli a k csúcs elemet, amennyiben az megtalálható a fában.
- *min(t)*: A minimális kulcsú csúcs címét adja vissza.
- *remMin(t, minp)*: A fából kiveszi a minimális kulcsú csúcsot, és a címét a minp pointerben ada vissza.

19 megkeresése
7 jobb gyerek?
28 bal gyerek?



Bináris keresőfa alapműveletei:

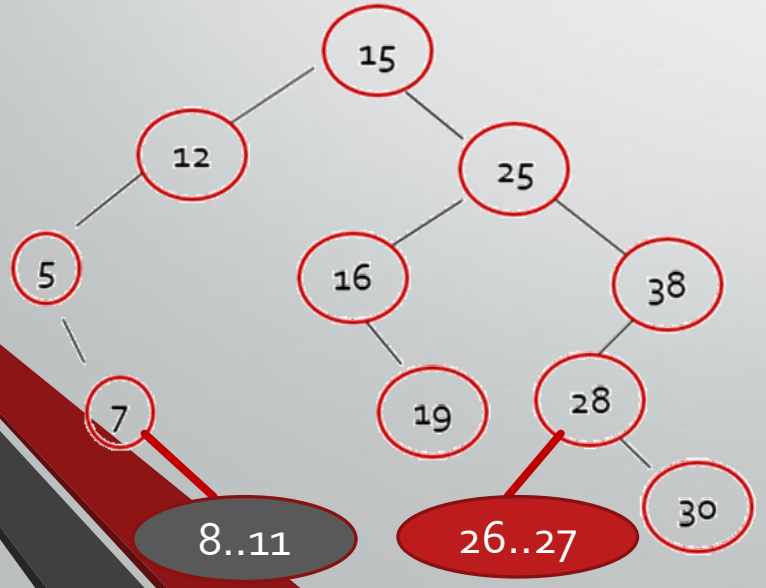
- *search(t, k):* A t fában megkeresi a k kulcsú elemet, ∅ -t ad vissza, ha nincs benne.
- *insert(t, k)*: Beszúrja a k kulcsú elemet, ha még nincs a fában.
- *del(t, k)*: Törli a k csúcs elemet, amennyiben az megtalálható a fában.
- *min(t)*: A minimális kulcsú csúcs címét adja vissza.
- *remMin(t, minp)*: A fából kiveszi a minimális kulcsú csúcsot, és a címét a minp pointerben ada vissza.

19 megkeresése

7 jobb gyerek?

28 bal gyerek?

7 törlése

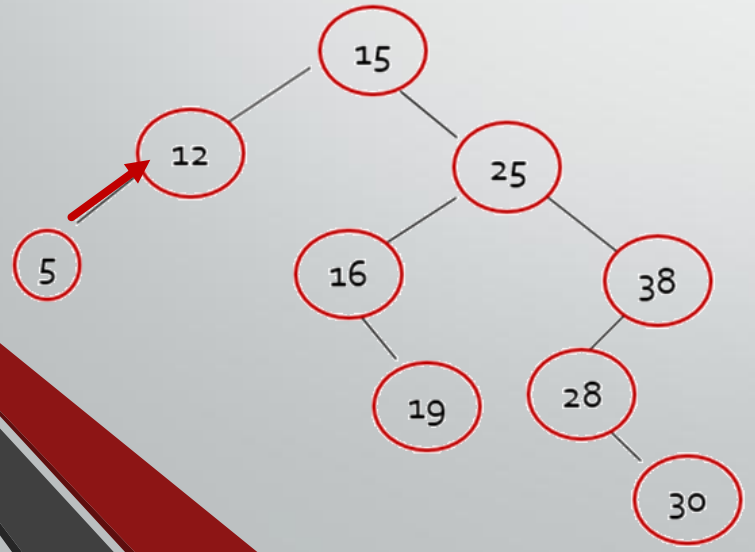


Bináris keresőfa alapműveletei:

- *search(t, k):* A t fában megkeresi a k kulcsú elemet, ∅ -t ad vissza, ha nincs benne.
- *insert(t, k)*: Beszúrja a k kulcsú elemet, ha még nincs a fában.
- *del(t, k)*: Törli a k csúcs elemet, amennyiben az megtalálható a fában.
- *min(t)*: A minimális kulcsú csúcs címét adja vissza.
- *remMin(t, minp)*: A fából kiveszi a minimális kulcsú csúcsot, és a címét a minp pointerben ada vissza.

19 megkeresése
7 jobb gyerek?
28 bal gyerek?

7 törlése
12 törlése

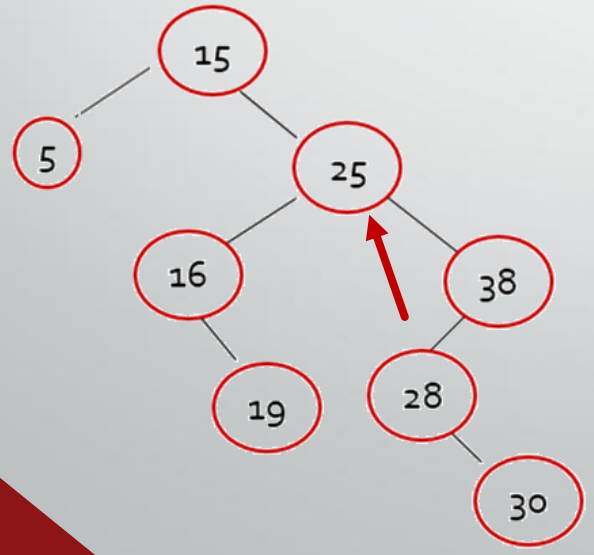


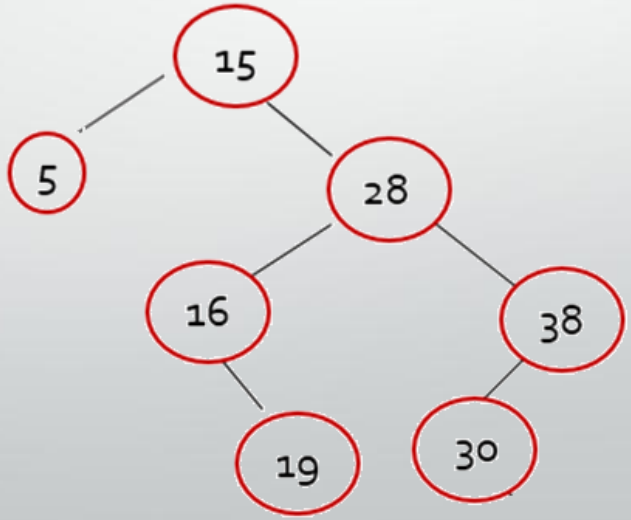
Bináris keresőfa alapműveletei:

- *search(t, k):* A t fában megkeresi a k kulcsú elemet, ∅ -t ad vissza, ha nincs benne.
- *insert(t, k)*: Beszúrja a k kulcsú elemet, ha még nincs a fában.
- *del(t, k)*: Törli a k csúcs elemet, amennyiben az megtalálható a fában.
- *min(t)*: A minimális kulcsú csúcs címét adja vissza.
- *remMin(t, minp)*: A fából kiveszi a minimális kulcsú csúcsot, és a címét a minp pointerben ada vissza.

19 megkeresése
7 jobb gyerek?
28 bal gyerek?

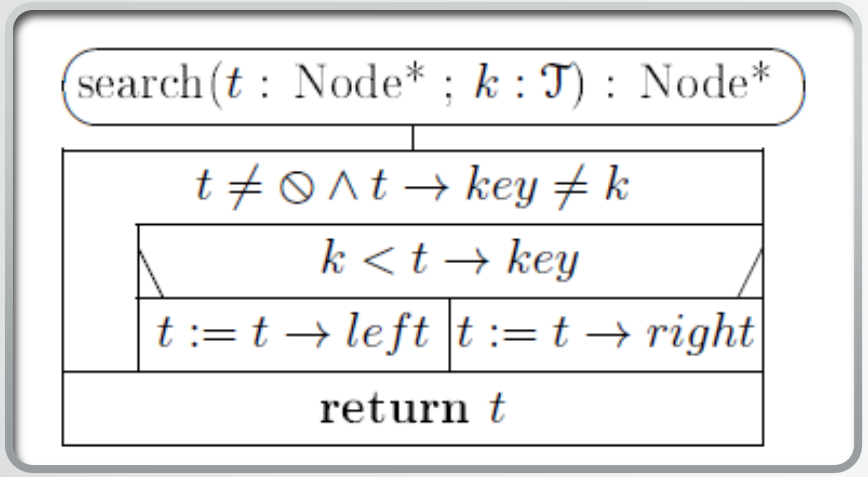
7 törlése
12 törlése
25 törlése

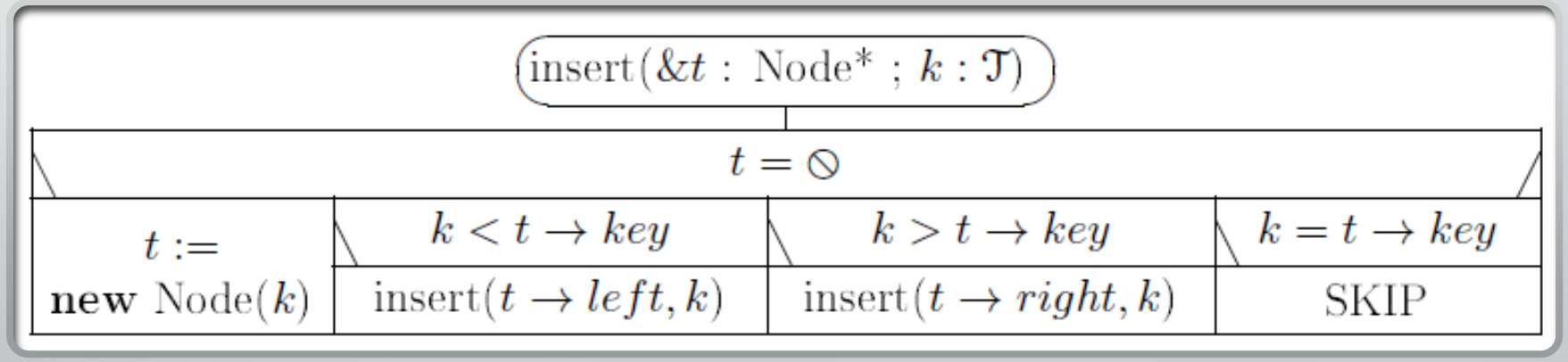




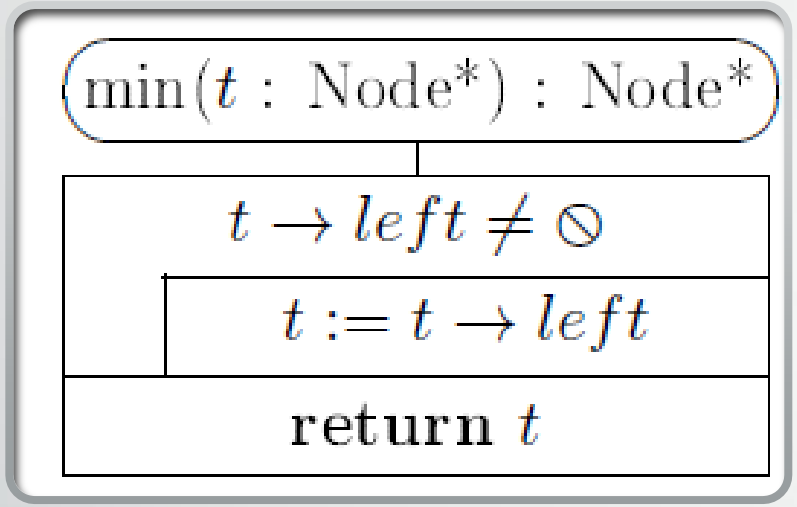
Mindegyik műveletidő $\Theta(h)$, ahol h a fa magassága
$AT(n) \in O(\log(n)) \sim 1{,}4\log(n)$,
$MT(n) \in O(n)$

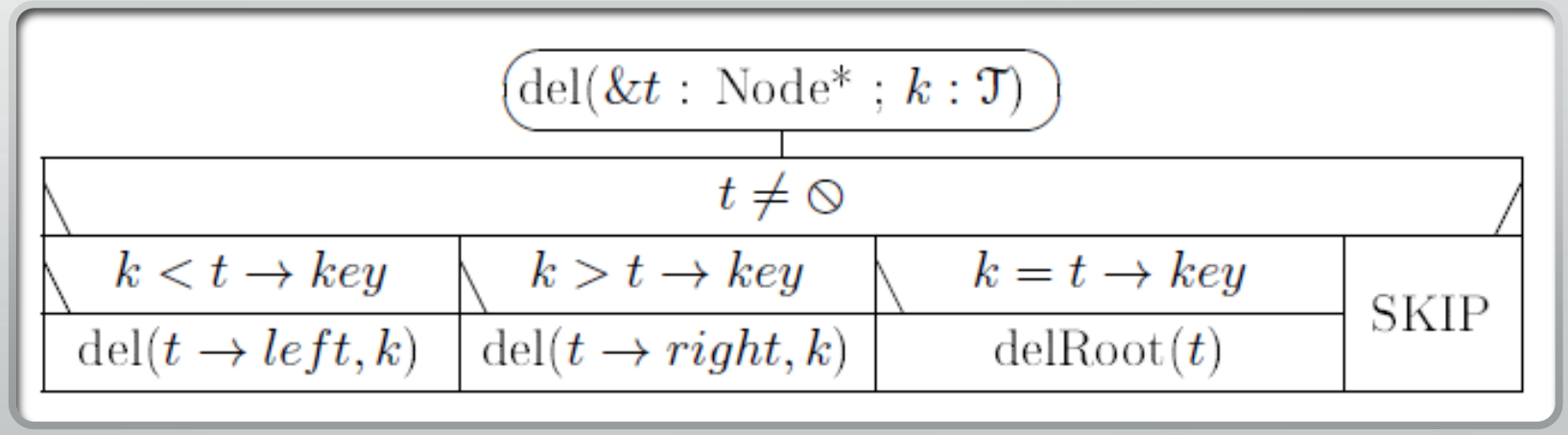
Bináris keresőfák műveletei



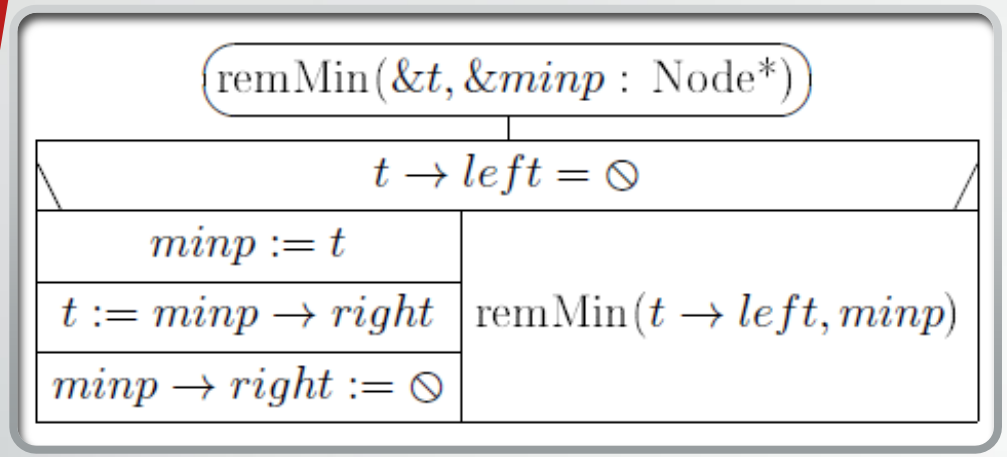


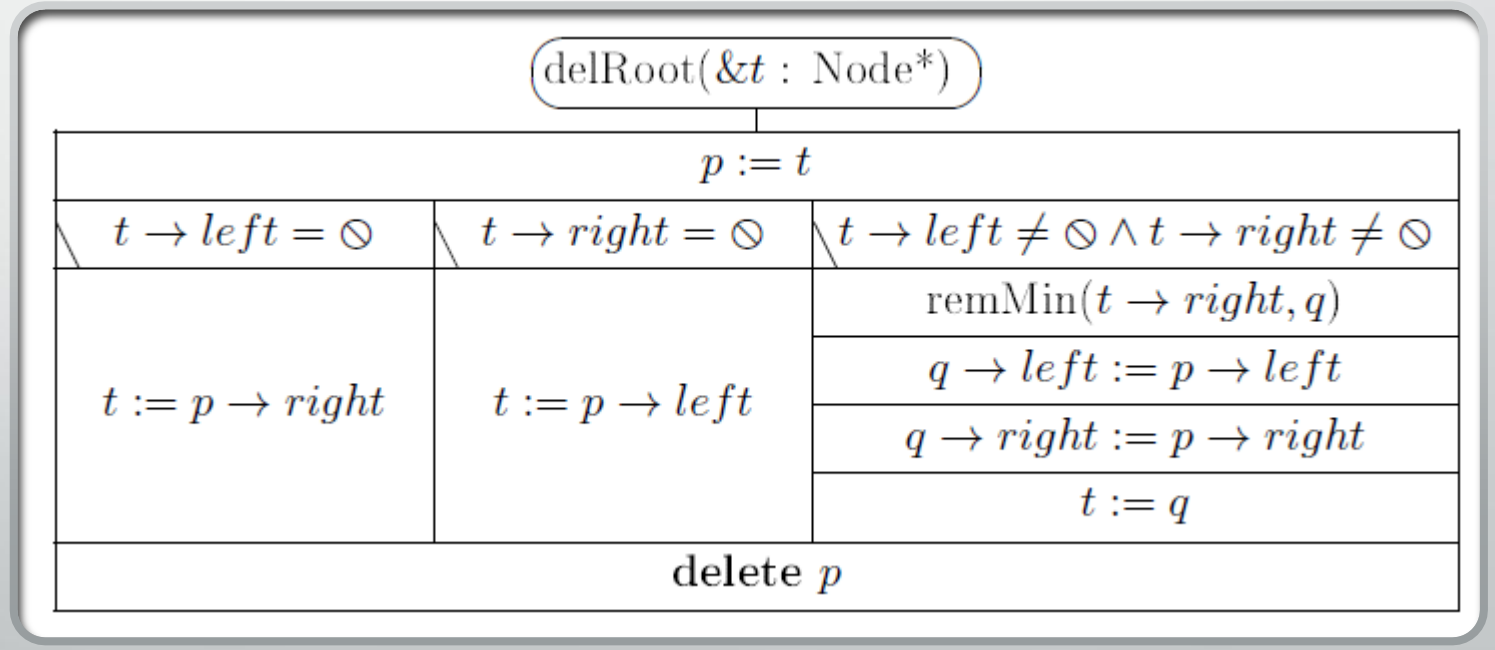
Bináris keresőfák műveletei





Bináris keresőfák műveletei





Bináris keresőfák műveleteinek C# kódja*

```
class Node
{
    public int Key;
    public Node Left, Right;

    public Node(int key)
    {
        Key = key;
        Left = Right = null;
    }
}
```

```
class BinarySearchTree
{
    public Node Root;

    public BinarySearchTree()
    {
        Root = null;
    }

    // Keresés egy adott kulcs alapján
    public Node Search(Node root, int key)
    {
        while (root != null && root.Key != key)
        {
            if (key < root.Key)
                root = root.Left;
            else
                root = root.Right;
        }
        return root;
    }
```

```
// Beszúrás
public Node Insert(Node root, int key)
{
    if (root == null)
        return new Node(key);

    if (key < root.Key)
        root.Left = Insert(root.Left, key);
    else if (key > root.Key)
        root.Right = Insert(root.Right, key);

    return root;
}

// Minimum érték keresése
public Node FindMin(Node root)
{
    while (root.Left != null)
        root = root.Left;
    return root;
}
```

A BinarySearchTree: egy osztállyal implementált modul (nem típus)

Bináris keresőfák műveleteinek C# kódja*

```
public void del(ref Node t, T key)
{
    if (t == null) return;

    if (key.CompareTo(t.key) < 0)
    {
        del(ref t.left, key);
    }
    else if (key.CompareTo(t.key) > 0)
    {
        del(ref t.right, key);
    }
    else
    {
        delRoot(ref t);
    }
}
```

```
private void remMin(ref Node t, ref Node minp)
{
    if (t.left == null)
    {
        minp = t;
        t = minp.right;
        minp.right = null;
    }
    else
    {
        remMin(ref t.left, ref minp);
    }
}
```

```
private void delRoot(ref Node t)
{
    Node p = t;

    if (t.left == null)
    {
        t = p.right;
    }
    else if (t.right == null)
    {
        t = p.left;
    }
    else
    {
        Node q = null;
        remMin(ref t.right, ref q);
        q.left = p.left;
        q.right = p.right;
        t = q;
    }

    // Itt felszabadítás vagy további kezelés szükség esetén
}
```

Teljes bináris fák, Kupacok
(Complete binary trees, heaps)

- **Szigorúan bináris fa** (full binary trees)**:**
  - a fa minden belső pontjának két gyereke van.
- **Tökéletes** (perfect) **bináris fa:**
  - olyan szigorúan bináris fa, ahol minden levél azonos szinten helyezkedik el.
  - *h* mélységű tökéletes bináris fa csúcsainak száma: $1+2+4+\ldots+2^h = 2^{h+1}-1$

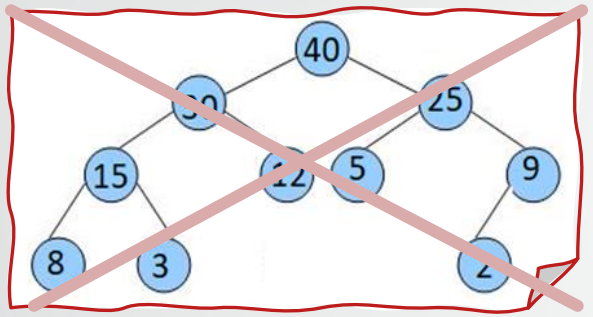
Teljes bináris fák, Kupacok

- **Majdnem teljes** (nearly complete) **bináris fa:**
  - olyan tökéletes bináris fa,
    melynek legalsó (levél) szintjéről elhagytunk néhány levelet (de nem az összeset).
  - h mélységű majdnem teljes bináris fa csúcsainak száma: $n \in 2^h..2^{h+1}-1$, és így $h = \lfloor \log n \rfloor$
  - Az alsó szinten levő leveleket elvéve egy $h - 1$ mélységű tökéletes bináris fát kapunk
- **Majdnem teljes balra tömörített bináris fa**:
  - Majdnem teljes bináris fa, de levelek csak a legalsó szintről, a jobb oldalról hiányozhatnak.
  - Ezeket ***teljes* (complete) fáknak** is nevezzük.

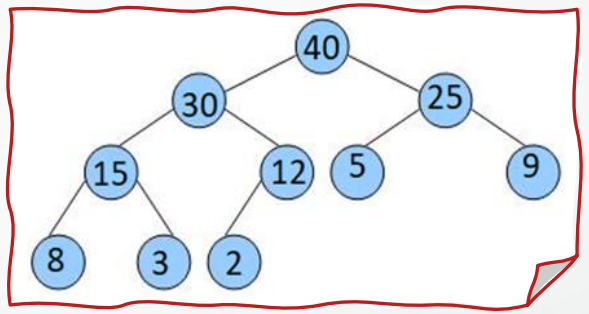
Teljes bináris fák, Kupacok

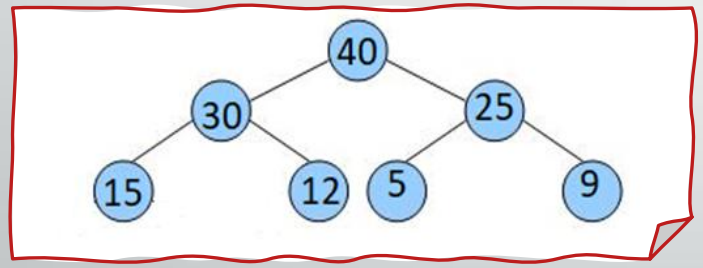
- **Kupac:** *Maximum kupac (heap)*: olyan teljes bináris fa, melynek minden belső csúcsára teljesül, hogy a belső csúcs kulcsa nagyobb vagy egyenlő a gyerekei kulcsánál.
  Így kupac tetején (a fa gyökerében) mindig az egyik legnagyobb elem található.
  - *Minimum kupac* hasonlóan: a szülő kulcsa ≤ a gyerekei kulcsánál.
- **Csonka kupac:** Olyan teljes bináris fa
  - minden szülő-gyerek párosban: a szülő kulcsa≥ a gyereke kulcsa
  - kivéve, ha a szülő a gyökércsúcs.
  - A gyökércsúcs kulcsa is definiált, de lehet, hogy kisebb, mint a gyereke kulcsa.
  - Átmeneti adatszerkezetek: egy tömb kupaccá alakítása során jönnek majd létre

Kupac-e?

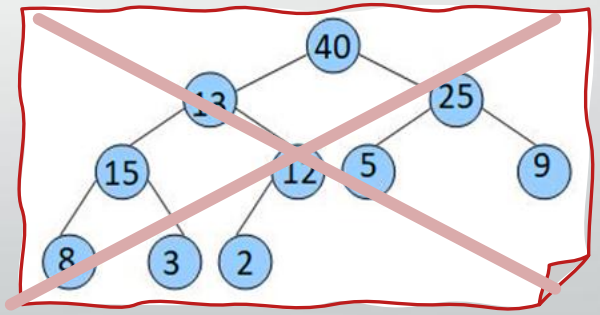


Nem balra tömörített





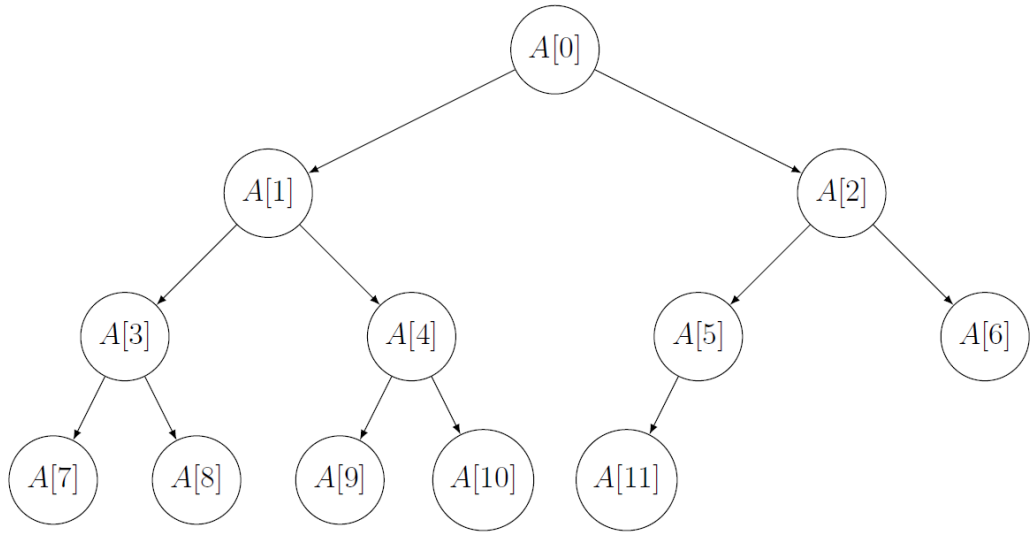
Tökéletes fa is



a belső csúcs kulcsa (13) nem nagyobb vagy egyenlő a gyerekei (15) kulcsánál

Teljes bináris fák aritmetikai ábrázolása:

- $A : \mathcal{T}[m]$ tömbben szintfolytonosan
  - Ha a fának n csúcsa van: $n \leq m$
  - Csúcsokat szintfolytonosan az A[0..n) résztömbben tároljuk, ahol:
  - A[0]: fa gyökércsúcsa (ha n>0)
- Indexelés:
  - Legyen a csúcs indexe: i
  - Bal gyerekének indexe: $2*i+1$
  - Jobb gyerekének indexe: $2*i+2$
  - Szülő indexe (ha i>0): $\left\lfloor \frac{i-1}{2} \right\rfloor$



Kupacok és elsőbbségi (prioritásos) sorok

- Maximum prioritásos sor (alapértelmezett):
  - mindig a legnagyobb prioritású elemet tudjuk kivenni
- Minimum prioritásos sor: a legkisebb prioritású elem kivétele
- Mindkettő fajta ábrázolható kupaccal
  - Alapértelmezett kupac: maximum kupac
- Műveletidő: (az *A* tömb *n* hosszúságú kezdőszelete egy kupac aritmetikai ábrázolása)

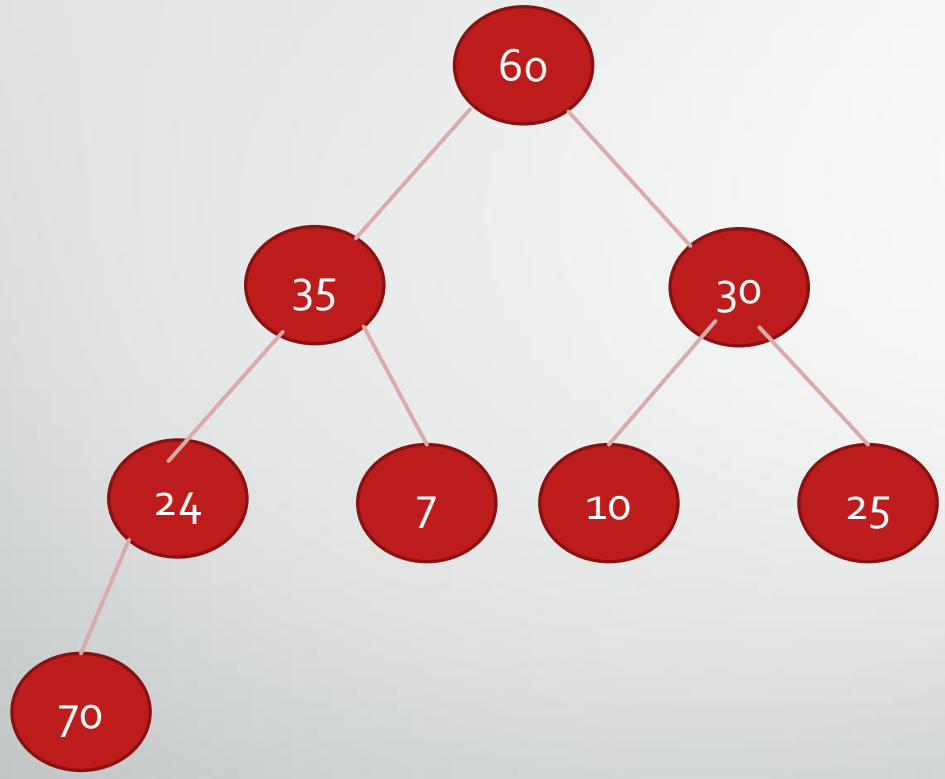
- $MT_{\text{add}:\, n<A.length}(n) \in \Theta(\log n)$
  - $MT_{\text{add}:\, n=A.length}(n) \in \Theta(n)$
  - $AT_{\text{add}}(n) \in O(\log n)$
  - $mT_{\text{add}}(n) \in \Theta(1)$
  - $MT_{\text{remMax}}(n) \in \Theta(\log n)$
  - $mT_{\text{remMax}}(n) \in \Theta(1)$
  - $T_{\max}(n) \in \Theta(1)$

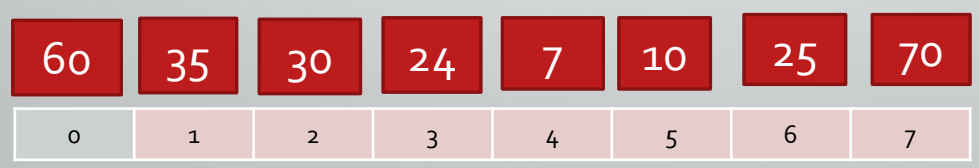
PrQueue
$-\ A : \mathcal{T}[]$ // $\mathcal{T}$ is some known type ; $A.length$ is the physical
$-$ constant $m0 : \mathbb{N}_+ := 16$ // size of the PrQ, its default is m0.
$-\ n : \mathbb{N}$ // $n \in [0..A.length]$ is the actual length of the PrQ
$+$ PrQueue$(m : \mathbb{N}_+ := m0)\{\ A := \textbf{new}\ \mathcal{T}[m];\ n := 0\ \}$ // create an empty PrQ
$+$ add$(x : \mathcal{T})$ // insert x into the priority queue
$+$ remMax$():\mathcal{T}$ // remove and return the maximal element of the priority queue
$+$ max$():\mathcal{T}$ // return the maximal element of the priority queue
$+$ isEmpty$() : \mathbb{B}\ \{\textbf{return}\ n = 0\}$
$+ \sim$ PrQueue$()\ \{\ \textbf{delete}\ A\ \}$
$+$ setEmpty$()\ \{n := 0\}$ // reinitialize the priority queue

Kupac műveletei: (animációval)

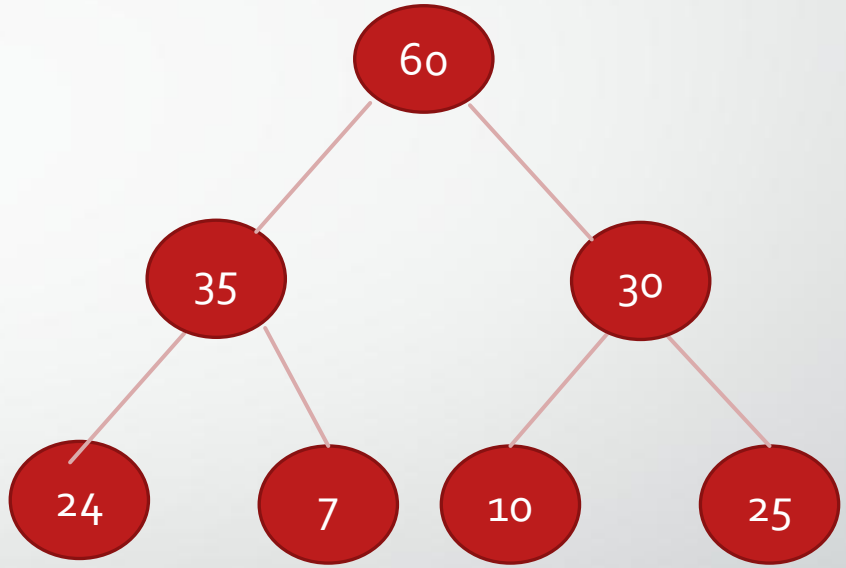
Műveletidő: $\lceil \log_2 n \rceil$

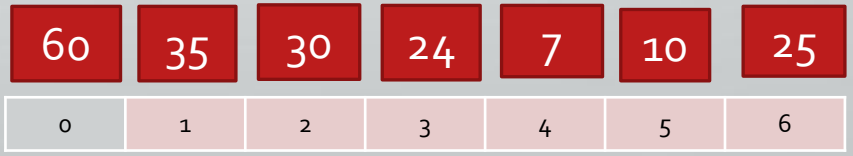
Beszúrás





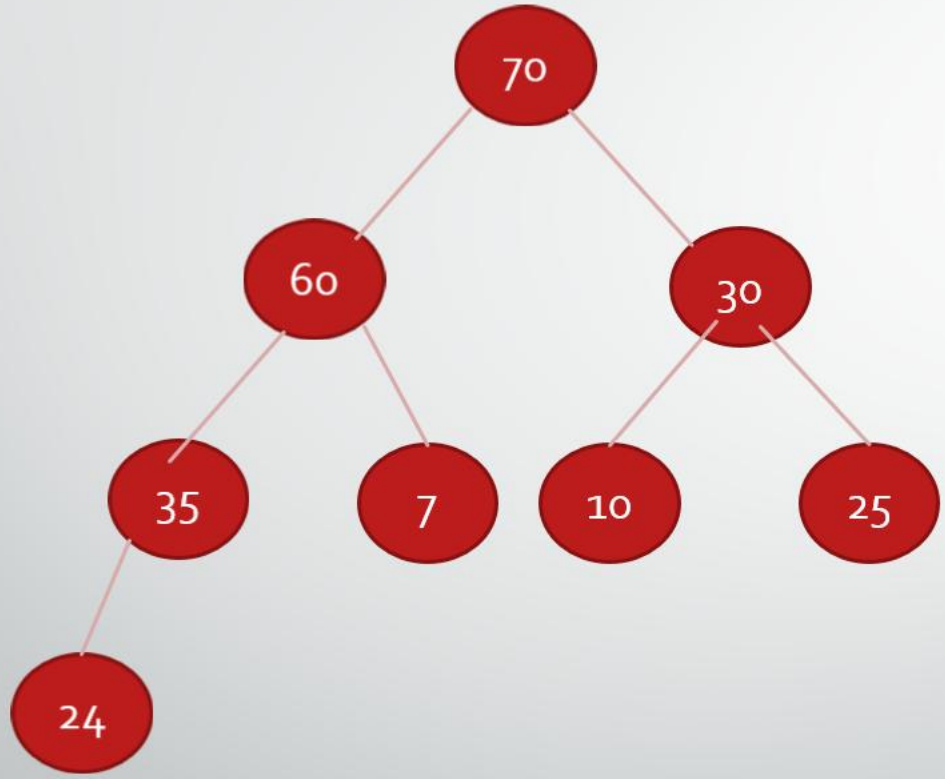
Maximum Törlés

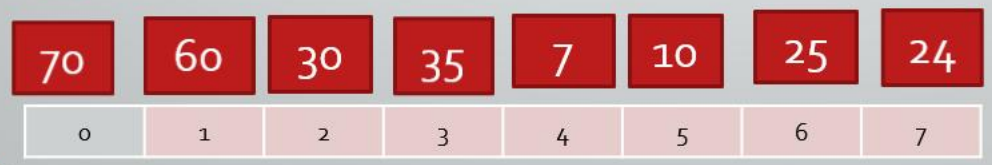




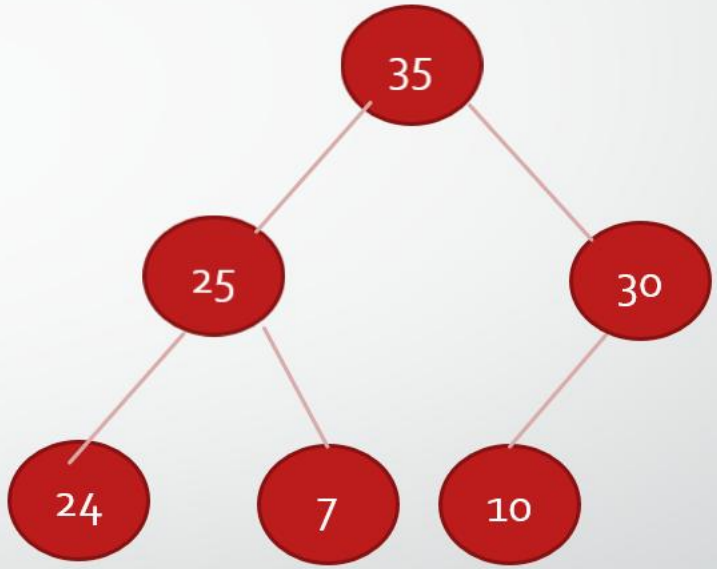
Kupac műveletei: (animáció nélkül)

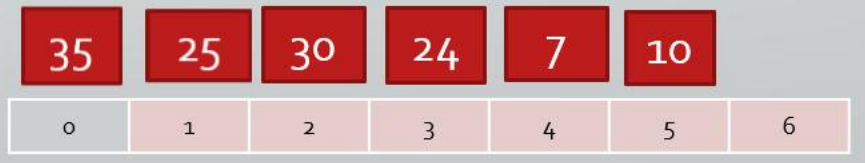
Beszúrás



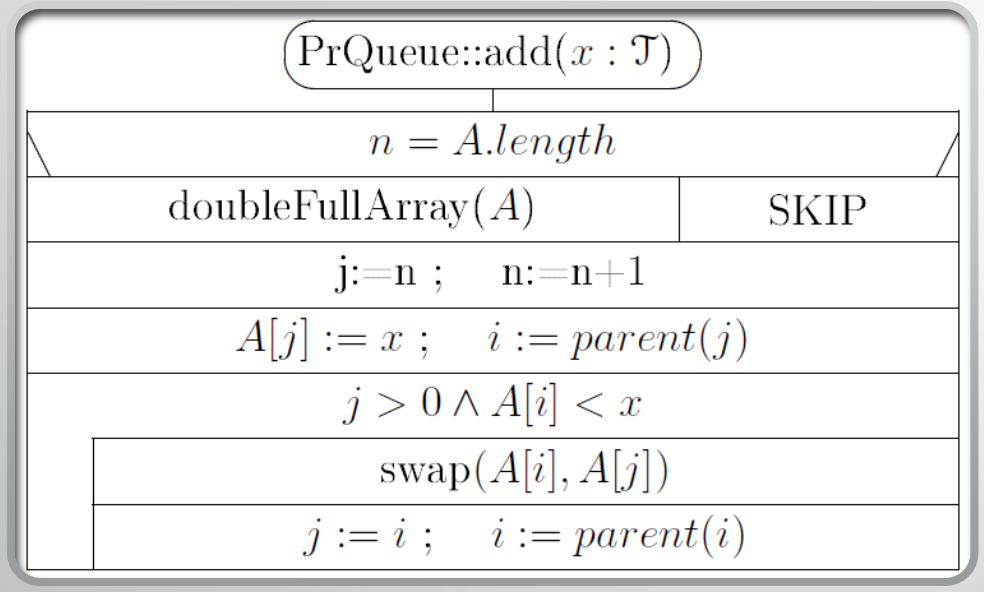


Maximum Törlés



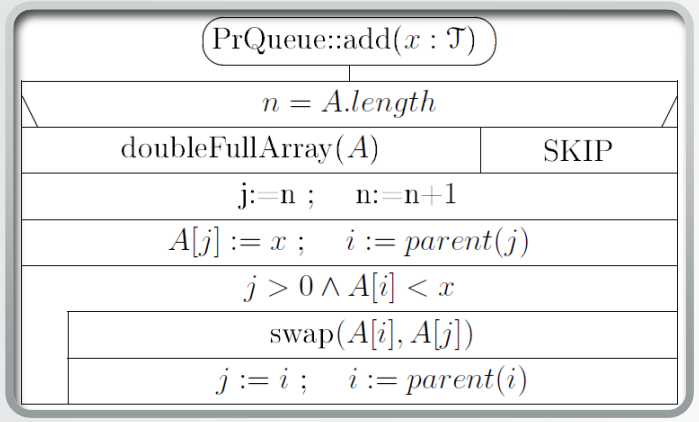


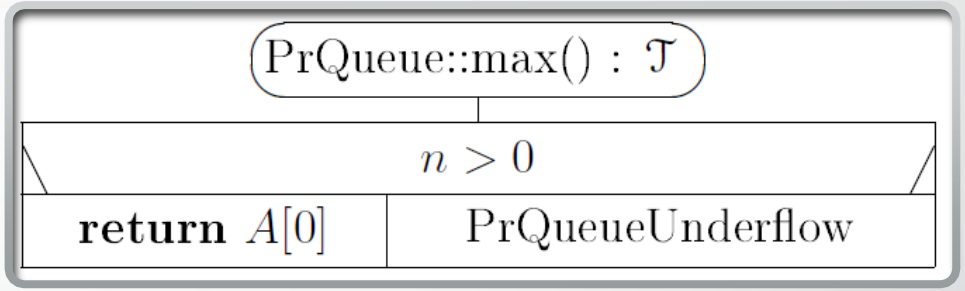
Prioritásos sor műveletei kupac esetén



- $MT_{\text{add} : n<A.length}(n) \in \Theta(\log n)$
  - a ciklus legfeljebb annyiszor iterál, amennyi a fa magassága: $\lfloor \log(n+1) \rfloor$
  - $\log n \leq MT_{\text{add} : n<A.length}(n) = \lfloor \log(n+1) \rfloor + 1 \leq \log n + 2$
- $MT_{\text{add} : n=A.length}(n) \in \Theta(n)$
  - a doubleFullArray(A) ciklusa: n iterációt hajt végre
  - ez dominál a többi tevékenységek műveletigénye fölött
- $mT_{\text{add}}(n) \in \Theta(1)$
  - X elég kicsi-> ciklus egyet sem iterál

Prioritásos sor műveletei kupac esetén

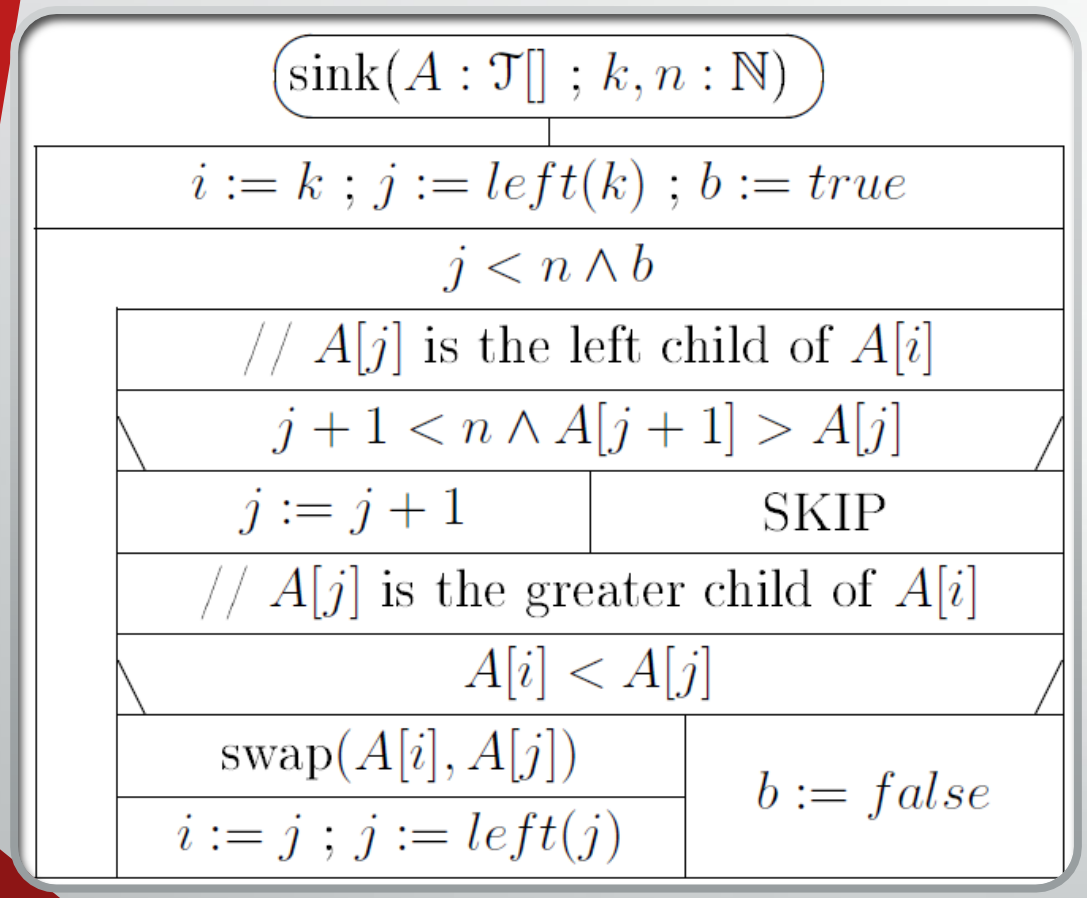




- $AT_{add}(n) \in O(\log n)$
  - (amortizált műveletigény számítás)
  - egyrészt $MT_{add\ :\ n<A.length}(n) \in \Theta(\log n)$
  - másrészt $n = A$.length -> doubleFullArray(A) n iteráció
  - ez kettesével képzeletben szétoszthető az előző $n/2$ add() hívás között:
    - biztosan nem hívtuk meg a doubleFullArray(A) eljárást
    - Így az $n/2$ add() hívás $O(\log n)$ műveletigénye 2-vel nő meg
    - azaz $O(\log n)$ marad

- $T_{max}(n) \in \Theta(1)$

Prioritásos sor műveletei kupac esetén

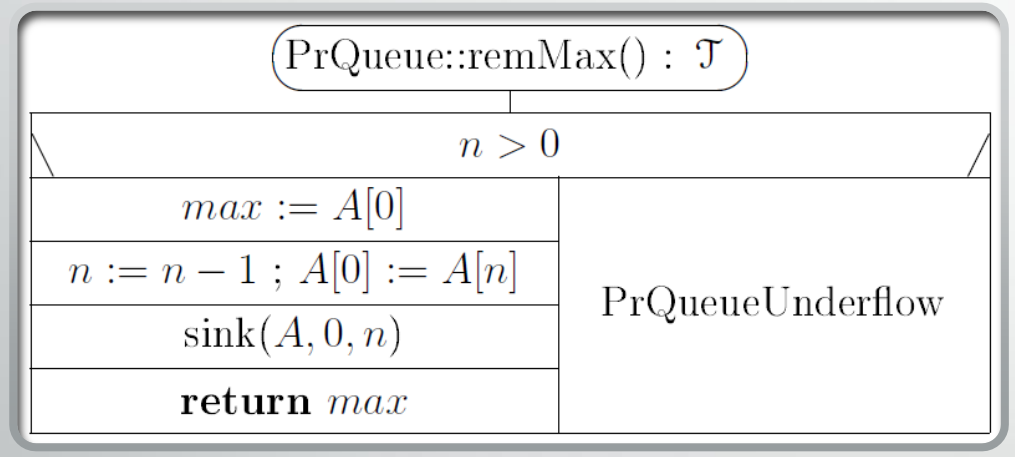


- $MT_{sink}(h) = h + 1$
  - (h: k gyökerű csonka kupac magassága)
  - a ciklus legfeljebb h iterációt végez
  - $1 \leq mT_{sink}(h) \leq 2$
  - Ha $k = 0$ ->

    $\log n \leq MT_{sink}(n) = h + 1 =$

    $= \lfloor \log n \rfloor + 1 \leq \log n + 1$

Prioritásos sor műveletei kupac esetén



- $MT_{\text{remMax}}(n) \in \Theta(\log n)$
  - $\log n \leq \log(n-1) + 1 \leq MT_{\text{remMax}}(n) \leq \log(n-1) + 2 \leq \log n + 2$
- $mT_{\text{remMax}}(n) \in \Theta(1)$
  - $2 = 1 + 1 \leq mT_{\text{remMax}}(n) = 1 + mT_{\text{sink}}(n\text{-}1) \leq 1 + 2 = 3$

Prioritásos sor C# kód*

```
using System;
using System.Collections.Generic;

class PriorityQueue<T> where T : IComparable<T>
{
    private List<T> heap = new List<T>();

    public int Count => heap.Count;

    // Elem hozzáadása a prioritásos sorhoz
    public void Add(T item)
    {
        heap.Add(item);
        HeapifyUp(heap.Count - 1);
    }
```

```
// Legnagyobb prioritású elem eltávolítása és visszaadása
public T RemMax()
{
    if (heap.Count == 0)
        throw new InvalidOperationException("A prioritásos sor üres!");

    T root = heap[0];
    heap[0] = heap[heap.Count - 1];
    heap.RemoveAt(heap.Count - 1);

    Sink(0);
    return root;
}
```

```
// Csúcsérték visszaadása eltávolítás nélkül
public T Max()
{
    if (heap.Count == 0)
        throw new InvalidOperationException("A prioritásos sor üres!");
    return heap[0];
}
```

Prioritásos sor C# kód*

```
// Heapify felfelé az újonnan beszúrt elem rendezéséhez
private void HeapifyUp(int index)
{
    while (index > 0)
    {
        int parentIndex = (index - 1) / 2;
        if (heap[index].CompareTo(heap[parentIndex]) <= 0)
            break;

        Swap(index, parentIndex);
        index = parentIndex;
    }
}
```

```
// Segédfüggvény két elem felcserélésére
private void Swap(int i, int j)
{
    T temp = heap[i];
    heap[i] = heap[j];
    heap[j] = temp;
}
```

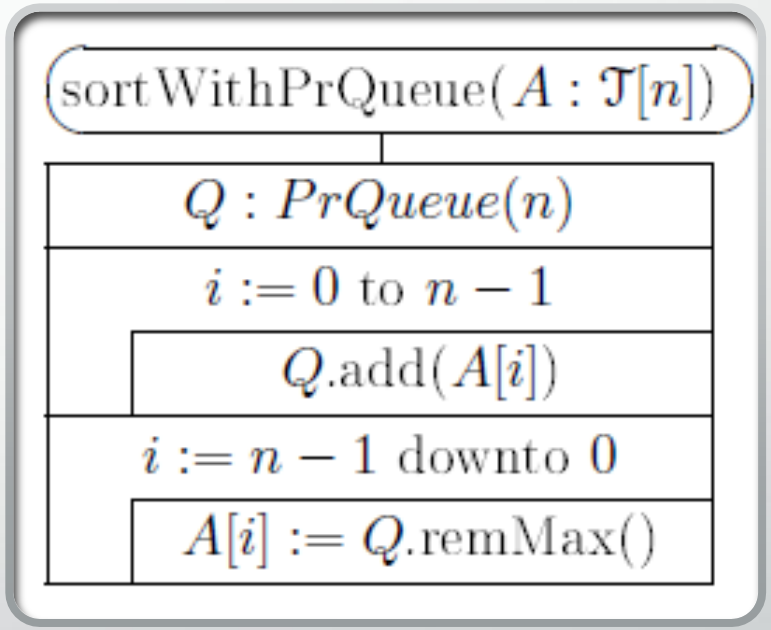
```
private void Sink(int index)
{
    int n = heap.Count - 1;
    int j = 2 * index + 1;
    bool b = true;
    while (j < n && b)
    {
        // Ha van jobb gyerek, és az nagyobb, mint a bal gyerek
        if (j + 1 < n && heap[ j + 1] > heap[ j])
        {
            j = j + 1;
        }
        // Ha a gyerek nagyobb, mint a szülő
        if ( heap[ i] < heap[ j])
        {
            // Cseréljük a szülőt és a nagyobb gyereket
            Swap( i , j );
            i = j;
            j = 2 * i + 1;
        }
        else
        {
            b = false;
        }
    }
}
```

Rendezés prioritásos sorral

- *Q*.add(*A*[*i*]), *A*[*i*]:= *Q*.remMax() utasítások műveletideje: $O(\log n)$
  - PrQueue-t sort reprezentáló kupac magassága $\leq \log n$

➢ a rendezés műveletigénye $O(n \log n)$

- Maximális műveletigénye $\Theta(n \log n)$
  - az input tömb szigorúan monoton növekvő
  - kupacépítés minden új csúcsot a fa gyökeréig mozgat fel
  - ➢ A végső kupac leendő levelei beszúrásánál:
    - A fa mélysége: $\lfloor \log n \rfloor - 1$
    - A leendő levelek száma: $\lceil n/2 \rceil$
    - ➢ A levelek beszúrásának futási ideje $\Omega(n \log n)$
- SortWithPrQueue hátránya
  - munkamemória: $S(n) \in \Theta(n)$
  - (Beszúró rendezésnél: $S(n) \in \Theta(1)$)



➢ Kupacrendezés

Rendezés prioritásos sorral C# kód*

```
static void Main()
{
    int[] array = { 5, 3, 8, 1, 2, 7, 4, 6 };

    Console.WriteLine("Eredeti tömb:");
    Console.WriteLine(string.Join(", ", array));

    HeapSort(array);

    Console.WriteLine("Rendezett tömb (növekvő sorrendben):");
    Console.WriteLine(string.Join(", ", array));
}
}
```

```
Eredeti tömb:
5, 3, 8, 1, 2, 7, 4, 6
Rendezett tömb (növekvő sorrendben):
1, 2, 3, 4, 5, 6, 7, 8
```

```
class Program
{
    static void HeapSort(int[] array)
    {
        PriorityQueue<int> pq = new PriorityQueue<int>();

        // Összes elem hozzáadása a prioritásos sorhoz
        foreach (int num in array)
        {
            pq.Add(num);
        }

        // Elemek kivétele sorrendben (nagyobb elöl)
        for (int i = array.Length - 1; i >= 0; i--)
        {
            array[i] = pq.RemMax();
        }
    }
```

Ellenőrző kérdések

1. A bináris fa fogalmát ismertnek feltételezve, mondja ki a bináris keresőfa definícióját!
2. A *t* egy láncoltan ábrázolt bináris keresőfa gyökérpointere. A csúcsokban nincsenek „parent” pointerek.
   - Írja meg az alapműveletek (search(*t* , *k*), insert(*t* , *k)*, del(*t* , *k*), min(*t*), remMin(*t* , *minp*)) eljárások struktogramját!
     - Rekuzívan / iteratívan
   - Mennyi az egyes eljárások (maximális / minimális) műveletigénye?
3. Szemléltesse a keresőfa alapműveleteinek hatását az alábbi bináris keresőfán!
   { [({} 1 {2}) 3 (4)] 5 [({} 6 {7}) 8 (9)] }

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek I. előadásjegyzete alapján készült.

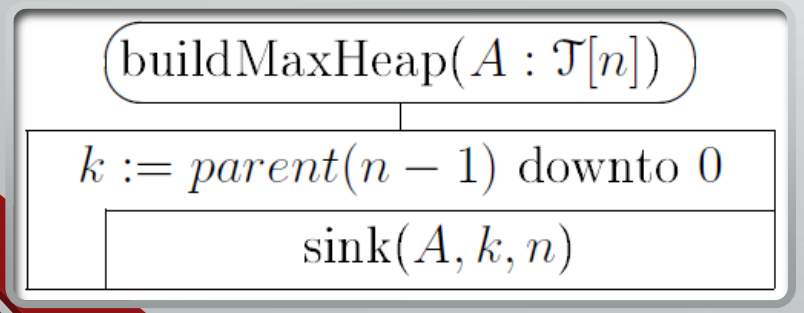
Algoritmusok és adatszerkezetek I.
8. Előadás

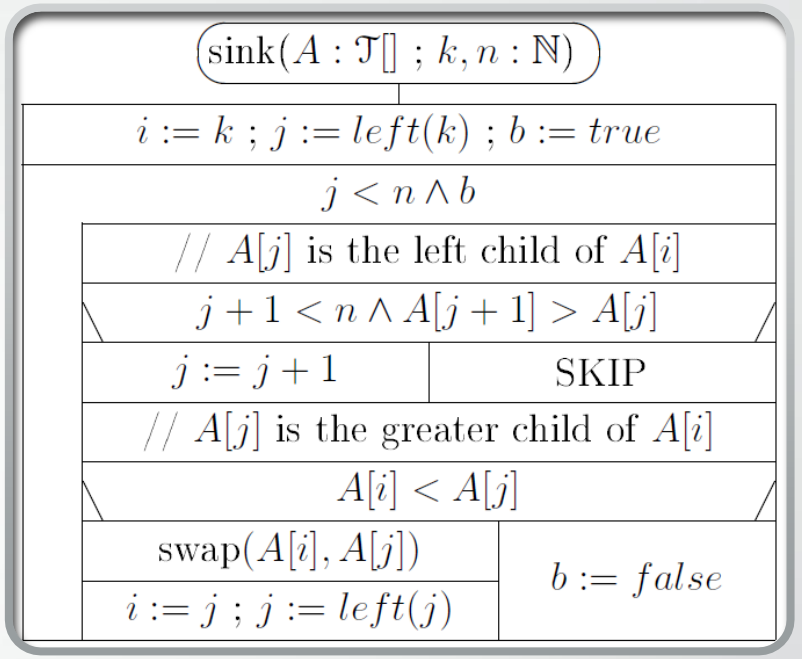
Kupacrendezés

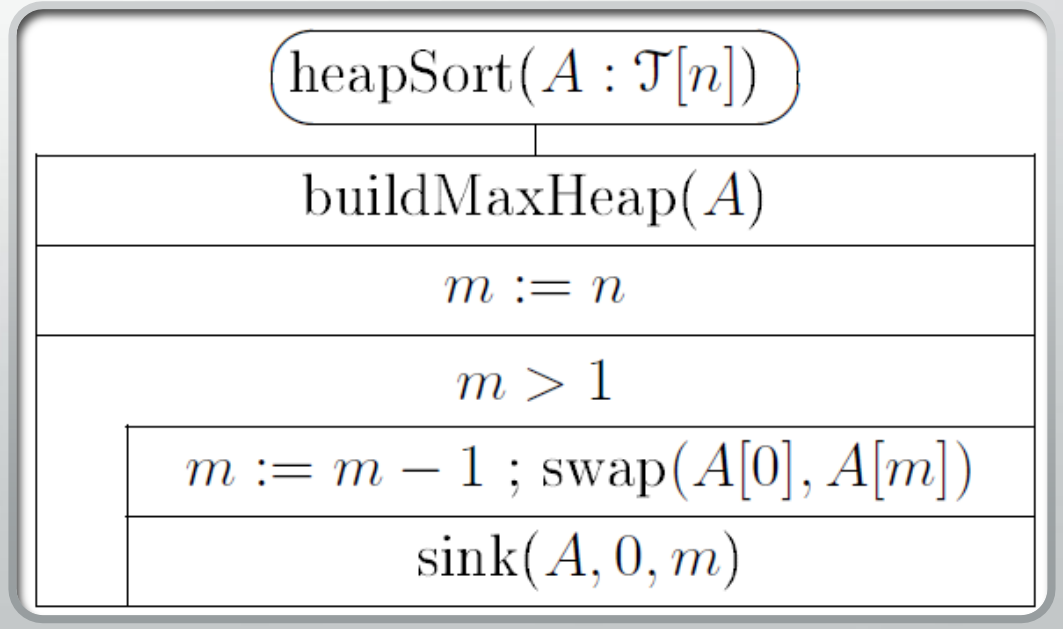
Tartalom

Kupacrendezés (heap sort)

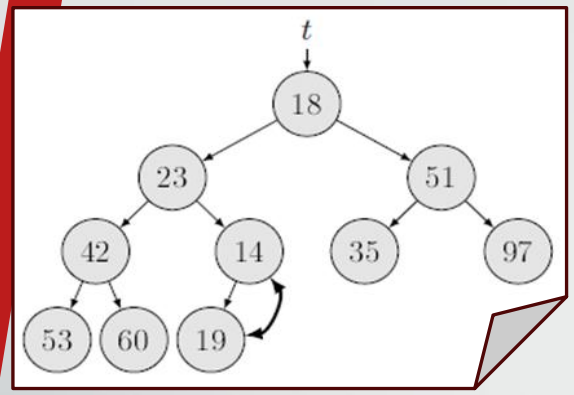
- sortWithPrQueue($A : \mathcal{T}[n]$) rendezés optimalizálása
  - helyben rendezővé alakítjuk: Θ (*n*) -> Θ (1) segédmemória
  - optimalizáljuk a kupac felépítését (O(*n* log *n*))
- $MT(n) \in \Theta(n \log n)$
- Lépések:
  - tömb kupaccá alakítása
  - rendezés



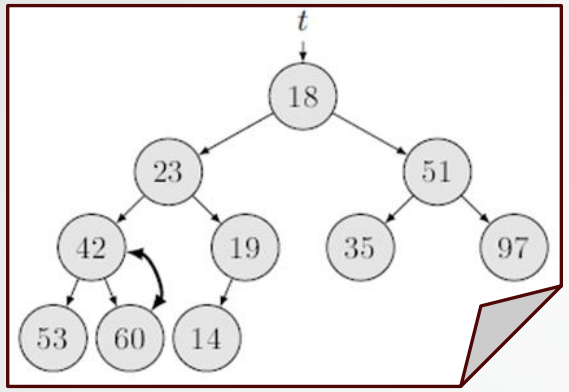




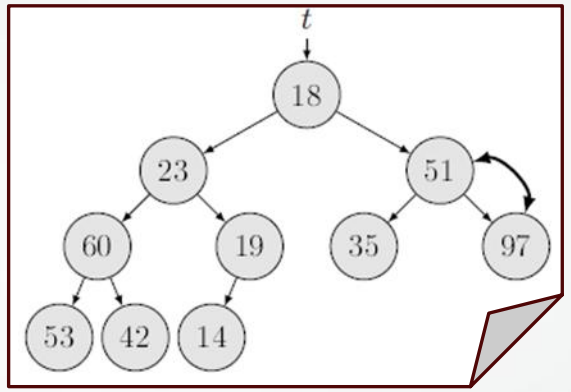
Kupaccá alakítás szemléltetése



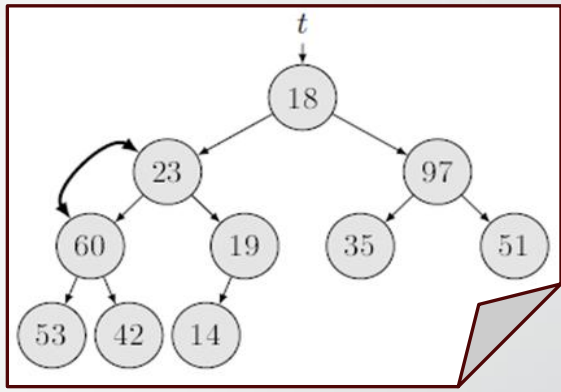
18, 23, 51, 42, ***14***, 35, 97, 53, 60, ***19***



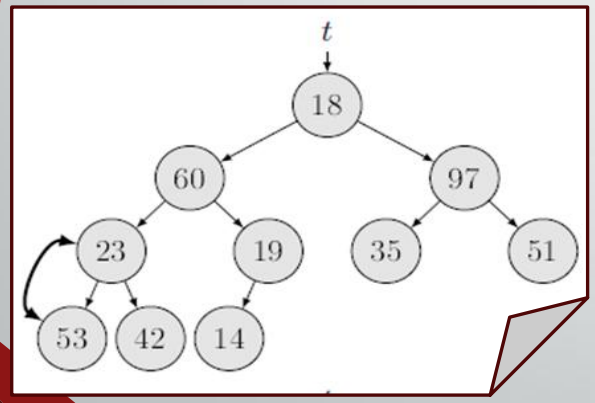
18, 23, 51, ***42***, 19, 35, 97, 53, ***60***,14



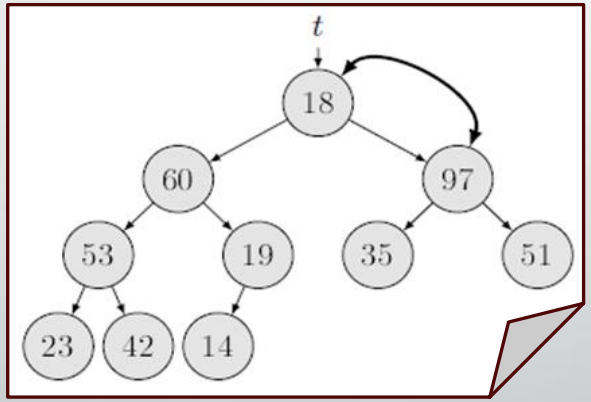
18, 23, ***51***, 60, 19, 35, ***97***, 53,42, 14



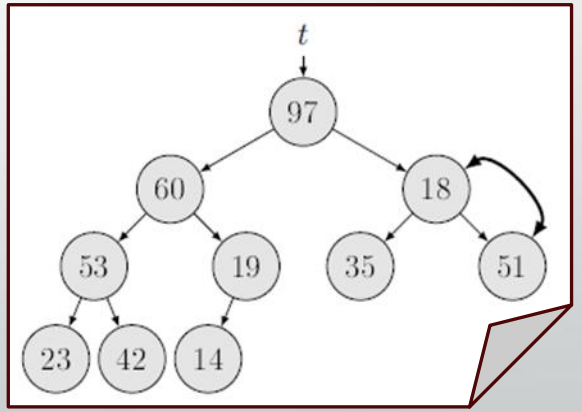
18, ***23***,97, ***60***, 19, 35,51, 53,42, 14



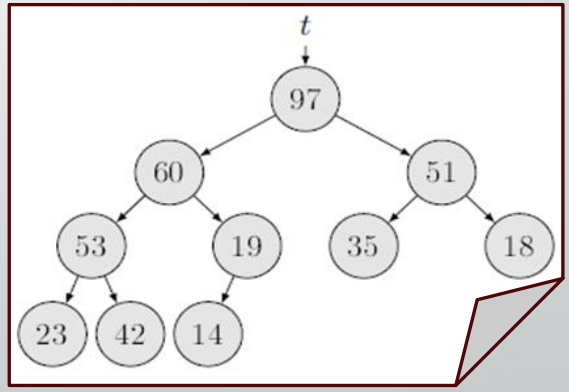
18,60,97,***23***, 19, 35,51, ***53***,42, 14



18,60,***97***, 53, 19, 35,51,23,42, 14



97, 60,***18***, 53, 19, 35,***51***,23,42, 14

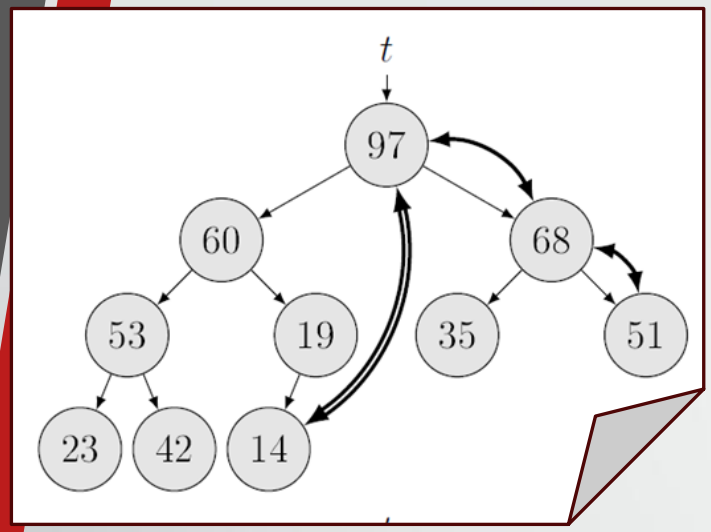


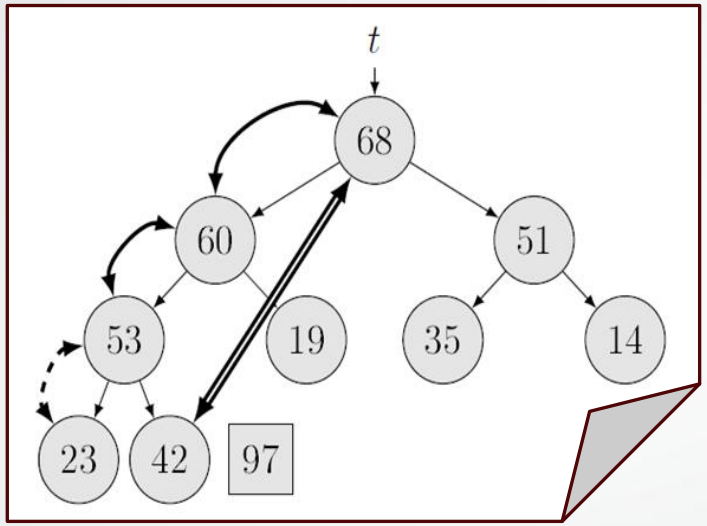
97, 60, 51, 53, 19, 35, 18,23,42, 14

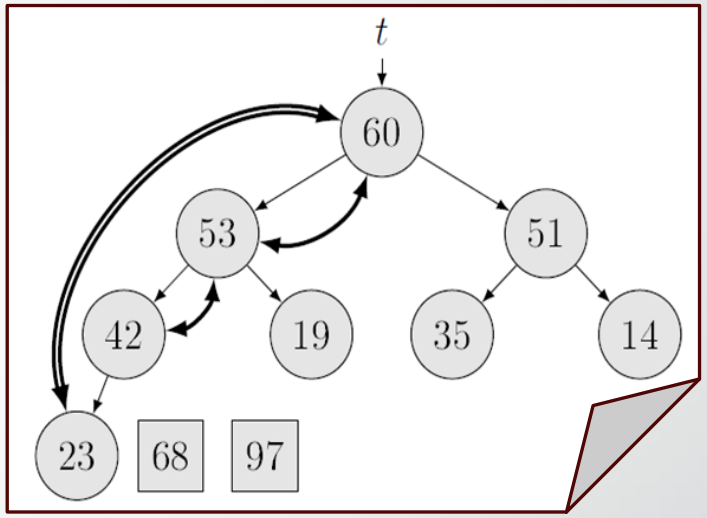
Csonka kupacok: csak a gyökérelem lehet „rossz helyen”

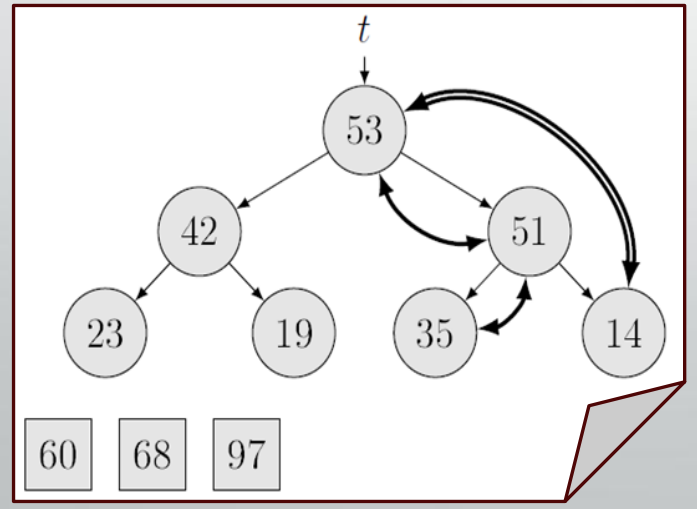
Ha ∀ szülő-gyerek párosban a szülő kulcsa ≥ a gyereke kulcsa, kivéve, ha a szülő a gyökércsúcs.

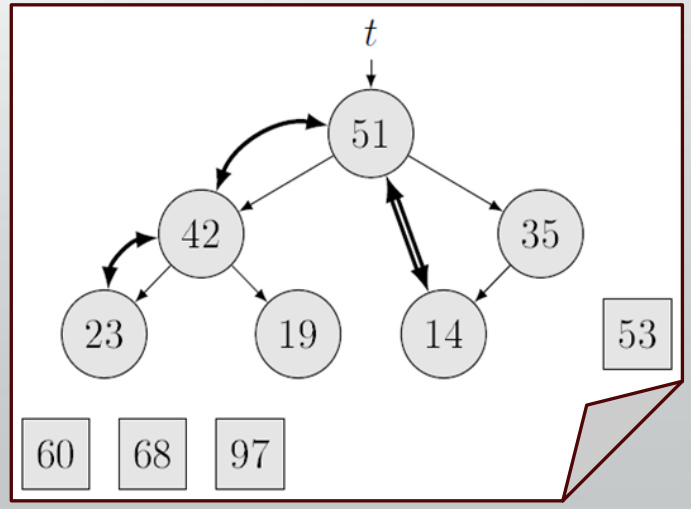
Kupacrendezés szemléltetése



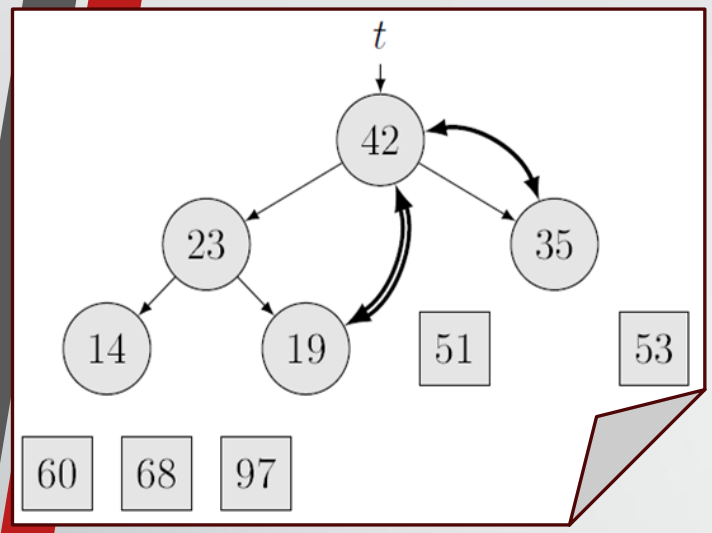


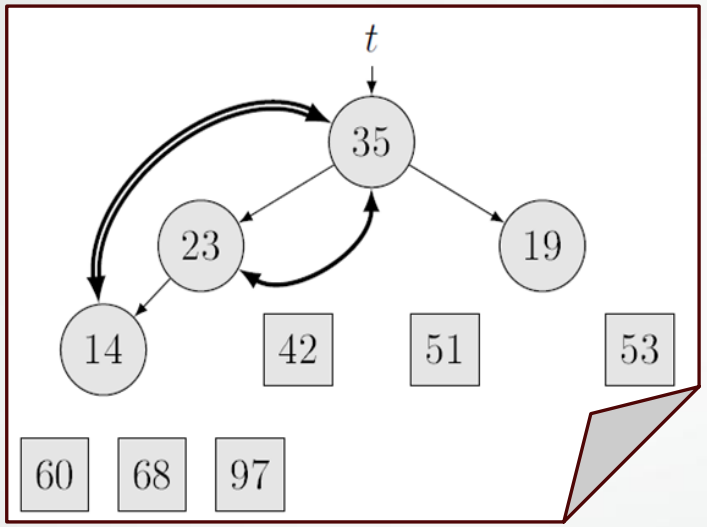


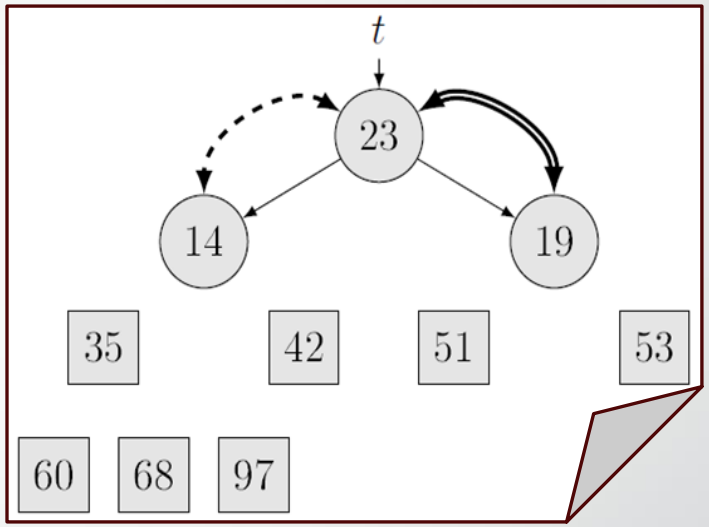


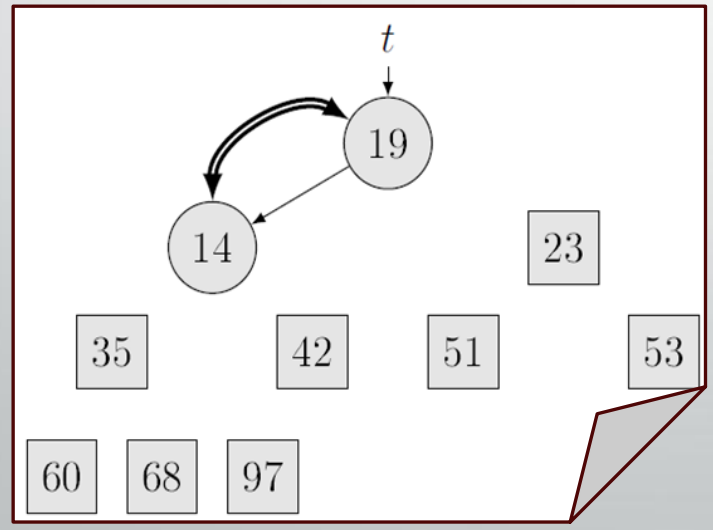


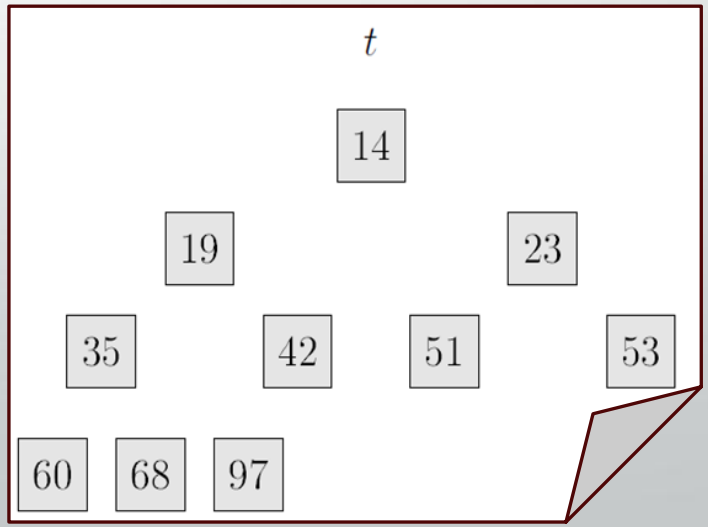
Kupacrendezés szemléltetése











A heap sort szemléltetése az $A : \mathbb{N}[10]$ tömbön

Jelölések:

- lesüllyesztéseknél
  - *: szülő
  - #: nagyobbik gyerek
- Swap műveleteknél
  - ~: megcserélendő elemek
  - +: végleges helyükre érkezett kulcsok

op	0	1	2	3	4	5	6	7	8	9
sink	68	23	51	42	*19	35	97	53	60	#14
sink	68	23	51	*42	19	35	97	53	#60	14
sink	68	23	*51	60	19	35	#97	53	42	14
sink	68	*23	97	#60	19	35	51	53	42	14
...	68	60	97	*23	19	35	51	#53	42	14
sink	*68	60	#97	53	19	35	51	23	42	14
...	97	60	*68	53	19	35	#51	23	42	14
.	97	60	68	53	19	35	51	23	42	14
swap	~97	60	68	53	19	35	51	23	42	~14
sink	*14	60	#68	53	19	35	51	23	42	+97
...	68	60	*14	53	19	35	#51	23	42	+97
swap	~68	60	51	53	19	35	14	23	~42	+97
sink	*42	#60	51	53	19	35	14	23	+68	+97
...	60	*42	51	#53	19	35	14	23	+68	+97
.	60	53	51	*42	19	35	14	#23	+68	+97
swap	~60	53	51	42	19	35	14	~23	+68	+97
sink	*23	#53	51	42	19	35	14	+60	+68	+97
...	53	*23	51	#42	19	35	14	+60	+68	+97
swap	~53	42	51	23	19	35	~14	+60	+68	+97
sink	*14	42	#51	23	19	35	+53	+60	+68	+97
...	51	42	*14	23	19	#35	+53	+60	+68	+97
swap	~51	42	35	23	19	~14	+53	+60	+68	+97
sink	*14	#42	35	23	19	+51	+53	+60	+68	+97
...	42	*14	35	#23	19	+51	+53	+60	+68	+97
swap	*42	23	35	14	#19	+51	+53	+60	+68	+97
sink	*19	23	#35	14	+42	+51	+53	+60	+68	+97
swap	~35	23	19	~14	+42	+51	+53	+60	+68	+97
sink	*14	#23	19	+35	+42	+51	+53	+60	+68	+97
swap	~23	14	~19	+35	+42	+51	+53	+60	+68	+97
sink	*19	#14	+23	+35	+42	+51	+53	+60	+68	+97
swap	*19	#14	+23	+35	+42	+51	+53	+60	+68	+97
sink	*14	+19	+23	+35	+42	+51	+53	+60	+68	+97
.	+14	+19	+23	+35	+42	+51	+53	+60	+68	+97

Kupacrendezés műveletigénye

A kupaccá alakítás: $O(n \log n)$

- Az iteráció és a sink(A, k, n) eljáráshívás végrehajtásainak száma: $\left\lfloor \frac{n}{2} \right\rfloor$
- k gyökerű részfa magassága ≤ log n
  - durva felső becsléssel: $O(\log n)$

➢ $\left\lfloor \frac{n}{2} \right\rfloor$ hívás össz futási ideje: $O(n \log n)$

- Ciklus $\left\lfloor \frac{n}{2} \right\rfloor$ iterációja nem befolyásolja
  - $(O(n) \prec O(n \log n))$

Kupacrendezés műveletigénye

Kupacrendezés: $O(n \log n)$

- sink(A, 0, m) hívások határozzák meg:
  - $O(\log m)$ ($m \in \{n-1, n-2, \ldots, 1\}$)
  - durva felső becsléssel: $O(\log n)$

➢ Az $(n-1)$ hívás: $O(n \log n)$

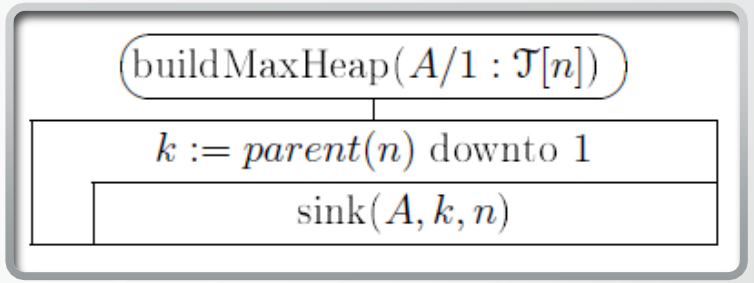
- A ciklus $(n-1)$ iterációja nem befolyásolja
  - $O(n)$ műveletigényt ad hozzá
  - $O(n) \prec O(n \log n)$.

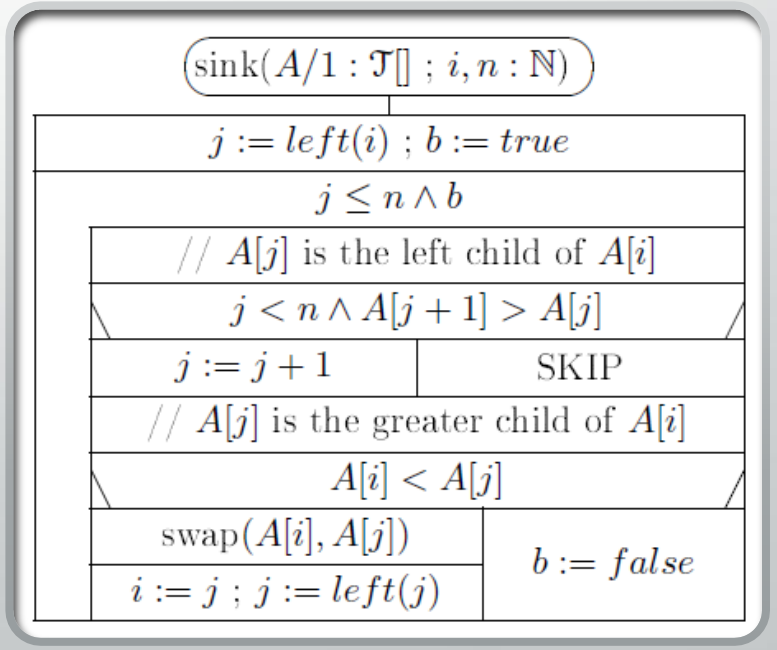
A kupaccá alakítás műveletigénye lineáris*

- *n* csúcsú fa szintfolytonosan az *A*/1 : T[*n*] tömbben
  - Az alábbi gondolatmenet
    1-től indexelt tömbökre vonatkozik.
    - nem befolyásolja a műveletigényt
    - egyszerűsíti a gonolatmenetet és a képleteket.
  - last(*n*) = *n*
  - *i* indexű csúcs gyerekei:
    - 2*i*, ill. 2*i* + 1, (ha 2*i* ≤ *n*, ill. 2*i* + 1 ≤ *n*)
  - *j* >1 indexű csúcs szülője: $\left\lfloor \frac{j}{2} \right\rfloor$

➢ 1..*k* sorszámú csúcsok szülei az 1.. $\left\lfloor \frac{k}{2} \right\rfloor$

→ 1 ≤ *j* ≤ *k*, és van szülője (*j* ≥ 2) -> a szülő index: $1 \leq \left\lfloor \frac{j}{2} \right\rfloor \leq \left\lfloor \frac{k}{2} \right\rfloor$





A kupaccá alakítás műveletigénye lineáris*

- ← Ha *i* indexű csúcsra $1 \le i \le \left\lfloor \frac{k}{2} \right\rfloor$
  - bal gyerekére: $2 \le 2i \le 2\left\lfloor \frac{k}{2} \right\rfloor \le k$
  - van gyereke az 1..*k* csúcsok között
- Egy *n* csúcsú, teljes bináris fának:

  fele(*n*) = $\left\lfloor \frac{n}{2} \right\rfloor$ belső csúcsa van

  - Ha a csúcsokat sorfolytonosan az 1..*n* sorszámokkal indexeljük
    - a szüleik sorszámai 1 .. $\left\lfloor \frac{n}{2} \right\rfloor$
- Az előbb belátott állítás szerint:

  egy *n* csúcsú teljes bináris fának $\left\lceil \frac{n}{2} \right\rceil$ levele van

A kupaccá alakítás műveletigénye lineáris*

- n_d :n méretű teljes bináris fa ***d* magasságú részfáinak száma**
- $n_{\geq d}$:a fa **legalább *d* magasságú részfáinak száma**
  - $1 \leq d \leq h = \lfloor \log n \rfloor$ (h az eredeti fa magassága)
- $n_{\geq 1}$ = fele(n) (=belső csúcsok)
- $n_{\geq 2}$ = fele(fele(n)) = fele^2(n)
  - a min. 2 magasságú részfák gyökércsúcsai: a min. 1 magasságú részfák gyökércsúcsainak szülei
  - utóbbiak szintfolytonosan 1..fele(n) sorszámúak -> az előbbiek fele^2(n) sorszámúak
- Teljes indukcióval:

$$\sum_{m=d}^{h} n_m = n_{\geq d} = fele^d(n) \leq \frac{n}{2^d}$$

A kupaccá alakítás műveletigénye lineáris*

- Hatékonyság mértéke:
  - Elemek beszúrása mekkora kupacba történik
- sortWithPrQueue(*A*)
  - Elemek beszúrása:
    - a végsőhöz közeli magasságú kupacba
- buildMaxHeap(*A*)
  - az elemek felét
    - nem kell lesüllyeszteni
  - ezek szüleit (¼) - egy magasságú fában süllyesztjük le
  - nagyszüleit ($^1/_8$) - kettő magasságú fában stb.

A kupaccá alakítás műveletigénye lineáris*

1. buildMaxHeap(*A*) maximális műveletigénye: $MT_b(n) \leq 2n + 1$

   = az eljárás meghívása

   + a ciklus $\left\lfloor \frac{n}{2} \right\rfloor$ iterációja

   + a lesüllyesztő eljárás $\left\lfloor \frac{n}{2} \right\rfloor$ -szeri meghívása

   +benne a lesüllyesztő ciklus végrehajtásai

   - *d* magasságú részfára a lesüllyesztő ciklus műveletigénye legfeljebb *d*
   - Az összes: n_d darab *d* magasságú részfákra a lesüllyesztő ciklusok műveletigénye:
     - legfeljebb $d * n_d$
   - A lesüllyesztő eljárás hívásai során $d \in 1..h$

     ➢ a kupaccá alakítás alatt a lesüllyesztő ciklusok műveletigényének összege legfeljebb $\sum_{d=1}^{h} d * n_d$

$$MT_b(n) \leq 1 + 2\left\lfloor \frac{n}{2} \right\rfloor + \sum_{d=1}^{h} d * n_d$$

A kupaccá alakítás műveletigénye lineáris*

- $\sum_{d=1}^{h} d * n_d$ kifejtése:
  - n_1 +
  - $n_2 + n_2$ +
  - $n_3 + n_3 + n_3$ +
  - . . .
  - $n_h + n_h + n_h + \cdot \cdot \cdot + n_h$ (h tagú összeg)
- Ezt oszloponként összeadva:

$$\sum_{d=1}^{h} d * n_d = \sum_{d=1}^{h} \sum_{m=d}^{h} n_m = \sum_{d=1}^{h} n_{\geq d} = \sum_{d=1}^{h} fele^d(n) \leq \sum_{d=1}^{h} \frac{n}{2^d} \leq n \sum_{d=1}^{\infty} \frac{1}{2^d} = n$$

- Az eredményeket összesítve:

$$MT_b(n) \leq 1 + (\left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2} \right\rfloor) + \sum_{d=1}^{h} d * n_d \leq 1 + n + n = 2n + 1$$

A kupaccá alakítás műveletigénye lineáris*

2. buildMaxHeap(A) minimális műveletigénye: $mT_b(n) \geq n$

- a kupaccá alakításból a lesüllyesztések belső műveletigényét elhanyagolva
  - csak a külső eljáráshívás + a ciklusiterációk +a lesüllyeszt-hívások:
  - $mT_b(n) \geq 1 + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2} \right\rfloor \geq n$
- $n \leq mT_b(n) \leq MT_b(n) \leq 2n + 1$

$mT_b(n), MT_b(n) \in \Theta(n)$

A merge sort műveletigényének kiszámítása*

A merge sort műveletigénye a merge sort egy másik változatának az algoritmusára vonatkozik!

- Az algoritmus:

mergeSort($A : \mathcal{T}[n]$)

// sort $A[0..n)$
$\mathrm{ms}(A, 0, n)$

ms($A : \mathcal{T}[n]$; $u, v : [0..n]$)

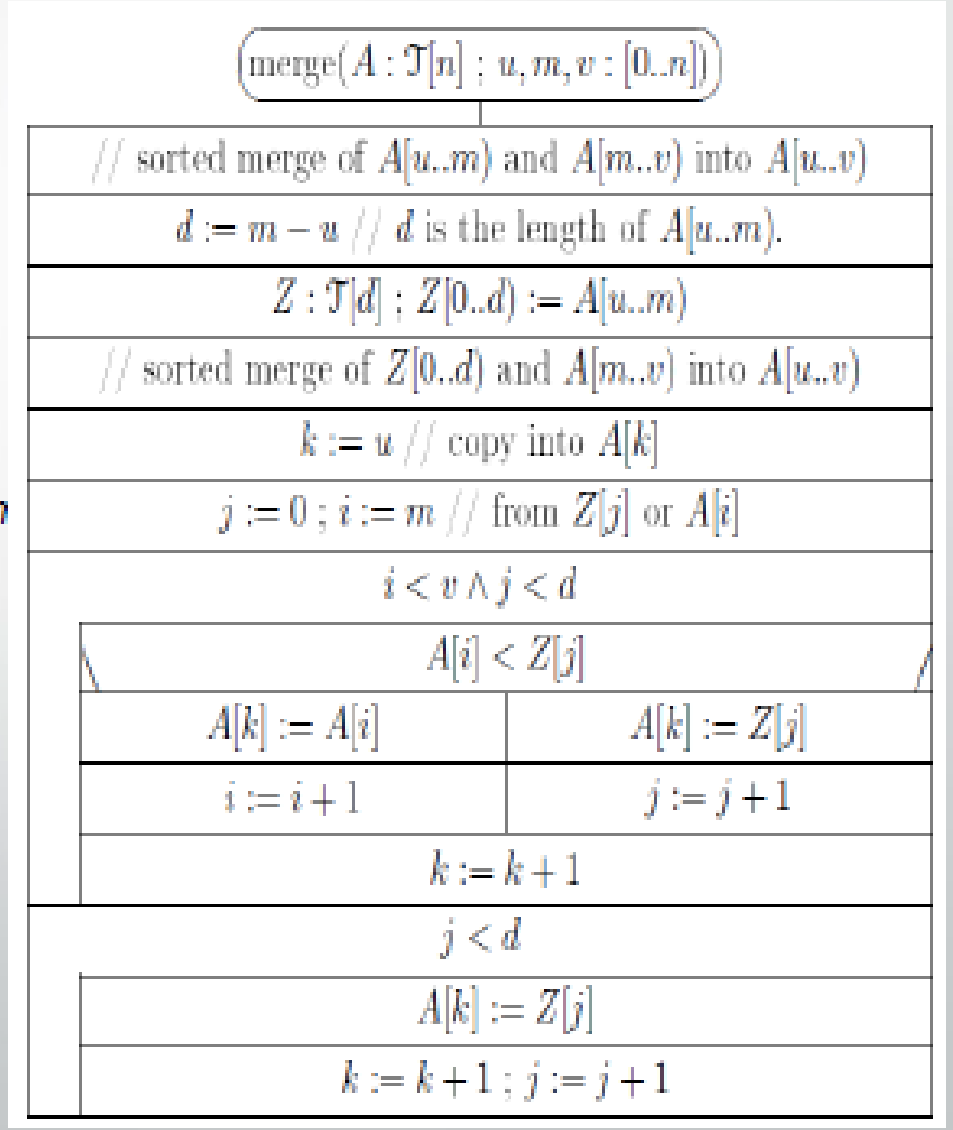
// sort $A[u..v)$	
$u < v - 1$	
$m := \lfloor \frac{u+v}{2} \rfloor$	SKIP
$\mathrm{ms}(A, u, m)$	
$\mathrm{ms}(A, m, v)$	
$\mathrm{merge}(A, u, m, v)$	

merge($A : \mathcal{T}[n]$; $u, m, v : [0..n]$)

// sorted merge of $A[u..m)$ and $A[m..v)$ into $A[u..v)$	
$d := m - u$ // d is the length of $A[u..m)$.	
$Z : \mathcal{T}[d]$; $Z[0..d) := A[u..m)$	
// sorted merge of $Z[0..d)$ and $A[m..v)$ into $A[u..v)$	
$k := u$ // copy into $A[k]$	
$j := 0$; $i := m$ // from $Z[j]$ or $A[i]$	
$i < v \wedge j < d$	
$A[i] < Z[j]$	
$A[k] := A[i]$	$A[k] := Z[j]$
$i := i + 1$	$j := j + 1$
$k := k + 1$	
$j < d$	
$A[k] := Z[j]$	
$k := k + 1$; $j := j + 1$	

Összefésülő rendezés algoritmusa*

- Összefésülés:
  - A merge(A, u, m, v) eljárás 2 ciklusa
- $Z[0..d)$ segédtömb:
  - az összefésülés során az output ne írja felül az inputot
  - elég, mert az összefésülés első explicit ciklusa során végig igaz lesz, hogy $k < i$
    - az első explicit összefésülő ciklus magjában $j < d = n$
    - $A[u..v)$ résztömbbe másolt elemek száma: $k - u$
      - $A[m..v)$ résztömbből: $i - m$ elemet
      - $Z[0..d)$ segédtömbből: j elemet
    - ➢ $k - u = i - m + j < i - m + m - u$
    - $k - u < i - u$
    - $k < i$



Merge eljárás műveletigénye *

- $l := v - u$ (az aktuális résztömb hossza)
- A merge eljárást csak akkor hívjuk meg, ha $u < v - 1$ -> $l \geq 2$
- 1 eljáráshívás + az 1. ciklus $\left\lfloor \frac{l}{2} \right\rfloor$ iterációja + a 2. és 3. ciklus iterációi
- a 2. és 3. ciklus iterációi
  - Min: Z tömbben lévő $\left\lfloor \frac{l}{2} \right\rfloor$ elemet biztosan visszamásoljuk az A-ba $\geq \left\lfloor \frac{l}{2} \right\rfloor$ iteráció
  - Max: a 2. és a 3. ciklus mindegyik elemet átmásolja: l iteráció

➢ $mT_{merge}(l) \geq 1 + \left\lfloor \frac{l}{2} \right\rfloor + \left\lfloor \frac{l}{2} \right\rfloor \geq \left\lfloor \frac{l}{2} \right\rfloor + \left\lfloor \frac{l}{2} \right\rfloor = l$ és $MT_{merge}(l) \leq 1 + \left\lfloor \frac{l}{2} \right\rfloor + l \leq 2l$

➢ $l \leq mT_{merge}(l) \leq MT_{merge}(l) \leq 2l$

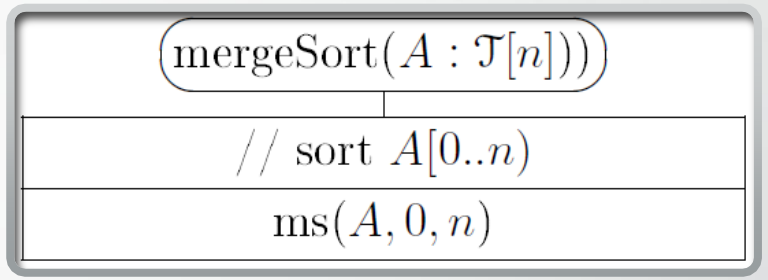
➢ $mT_{merge}(l), MT_{merge}(l) \in \Theta(l)$

A merge sort műveletigénye: szemléletes megközelítés*

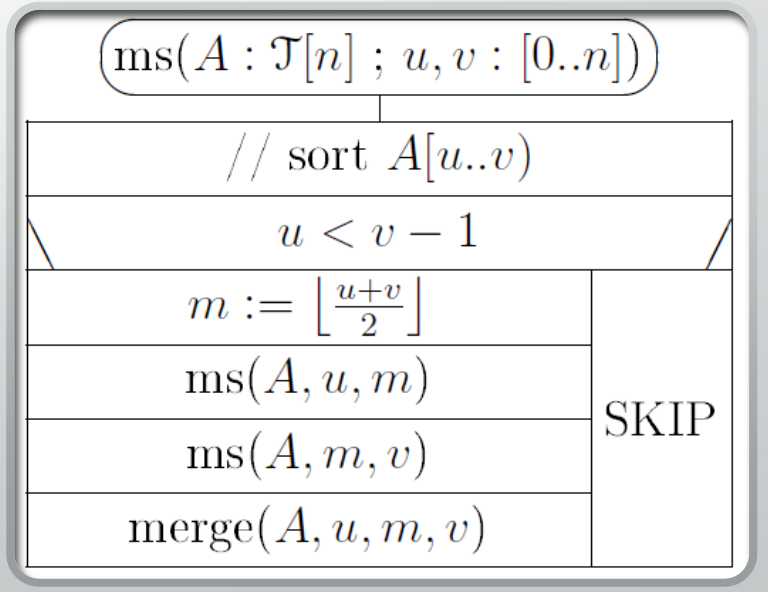
- $MT_{MS}(n), mT_{MS}(n) \in \Theta(n \log n)$
- Bizonyítása szemléletesen: (matematikai biz. később)
  - A műveletek túlnyomó részét a merge eljárás végzi el
    - $mT_{merge}(l), MT_{merge}(l) \in \Theta(l)$ (ahol l az aktuális résztömb hossza (l = v − u))
  - A rekurzív ms(A, u, v) eljárás minden hívásban felezi a rendezendő résztömb hosszát
  - ➢ A rekurziónak kb. log n + 1 szintje van
  - Minden rekurziós szinten: a merge hívások résztömbjei együtt lefedik az egész A tömböt
    - Kivétel az alsó egy vagy két szint: kevesebb
  - ➢ Egy tetszőleges szint összes merge hívásának műveletigényét összeadva: $\Theta(n)$
  - A szintenkénti műveletigényt a szintek számával szorozva nagyságrendben $\Theta(n \log n)$

A merge sort műveletigényének kiszámítása*

$MT_{MS}(n), mT_{MS}(n) \in \Theta(n \log n)$
ahol n a rendezendő tömb mérete



- ms(. . .) eljáráshívásai szigorúan bináris fát alkotnak
  - merge sort algoritmus tetszőleges $n > 0$ méretű input tömb esetén pontosan $2n - 1$ -szer hívja meg az ms(. . .) rekurzív eljárást
  - ahol az ms(. . .) hívások fájának $n - 1$ belső csúcsa és n levele van



A merge sort műveletigényének kiszámítása*

- Egy bináris fa a levelek száma szerint kiegyensúlyozott:
  - ha tetszőleges nemüres részfája bal és jobb részfája leveleinek száma legfeljebb eggyel tér el
- A levelek száma szerint kiegyensúlyozott szigorúan bináris fák halmaza a majdnem teljes bináris fák halmazának valódi részhalmaza
  - Biz: pl. a fa magassága szerinti teljes indukciót!

A merge sort műveletigényének kiszámítása*

- A fenti rekurzív program ms(. . .) eljáráshívásai a levelek száma szerint kiegyensúlyozott szigorúan bináris fát alkotnak
  - az ms(. . .) hívások fája majdnem teljes
    - $n > 0$ méretű input tömb esetén pontosan $2n - 1$ csúcsa van
  - így a magasságára:

$$\lfloor \log n \rfloor \leq h = \lfloor \log(2n - 1) \rfloor \leq \lfloor \log(2n) \rfloor = \lfloor \log(n) + 1 \rfloor = \lfloor \log n \rfloor + 1,$$

$$\text{tehát } \lfloor \log n \rfloor \leq h \leq \lfloor \log n \rfloor + 1.$$

Becslés a merge(A, u, m, v) eljárás összes végrehajtásának teljes műveletigényére*

∑ *Eddig:*

- $n > 0$ méretű input tömb esetén, az ms(. . .) hívások fájának $2n - 1$ csúcsa van, így a teljes merge sort műveletigénye a merge(. . .) eljárás végrehajtásai nélkül $2n$
- a merge(A, u, m, v) végrehajtásának műveletigényére: $l \leq mT_{\text{merge}}(l) \leq MT_{\text{merge}}(l) \leq 2l$, ahol $l = v-u+1$

- ms(A, u, v) rekurzív eljárás hívásai majdnem teljes fájának
  - leveleiben $u = v$ -> nem hívódik meg a merge(A, u, m, v)
  - belső csúcsaiban $u < v$ -> meghívódik a merge(A, u, m, v)
    - Utolsó előtti szinten valahányszor meghívódik
    - a magasabb szinteken mindig meghívódik

Becslés a merge(A, u, m, v) eljárás összes végrehajtásának teljes műveletigényére*

- A legalsó szinttől eltekintve bármelyik szinten: az adott szint ms(. . .) hívásai együtt lefedik a teljes A tömböt
  - Az egyes hívások által lefedett résztömbök összhossza pontosan n
- Az alsó két szinttől eltekintve minden szinten, az ms(A, u, v) rekurzív eljárás minden hívásában meghívódik a merge(A, u, m, v) eljárás, így az adott szinten, ez utóbbi hívások is lefedik az A tömböt.
- A rekurzió alsó két szintjétől eltekintve ∀ szinten, a merge(. . .) hívások összes műveletigénye ≥ n
  - $l \leq mT_{\text{merge}}(l) \leq MT_{\text{merge}}(l) \leq 2l$
  - a szorzás és összeadás diszributivitásának szabályai miatt
- A rekurzió legalsó szintjétől eltekintve ∀ szinten, a merge(. . .) hívások összes műveletigénye ≤ $2n$
  - (A legalsó szinten ez nulla.)

Becslés a merge(A, u, m, v) eljárás összes végrehajtásának teljes műveletigényére*

- A rekurzió h mélységére $h \geq \lfloor \log n \rfloor$ (ahol a rekurziós fa gyökere o mélségben van)

➢ A merge(A, u, m, v) eljárás összes végrehajtására együtt:

$$\boldsymbol{mT(n) \geq n(\lfloor \log n \rfloor - 1) \geq n(\log n - 2)}$$

- A rekurzió h mélységére a $h \leq \lfloor \log n \rfloor + 1$

➢ A merge(A, u, m, v) eljárás összes végrehajtására együtt:

$$\boldsymbol{MT(n) \leq 2n(\lfloor \log n \rfloor + 1) \leq 2n(\log n + 1)}$$

A merge sort műveletigényének kiszámítása*

∑ A merge(A, u, m, v) eljárás összes végrehajtására együtt:

$$n \log n - 2n \leq mT(n) \leq MT(n) \leq 2n(\log n) + 2n$$

\+ A merge(. . .) eljárás végrehajtásai nélküli $2n$ műveletigényt

➢ A teljes mergeSort(A) eljárásra:

$$n \log n \leq mT_{MS}(n) \leq MT_{MS}(n) \leq 2n(\log n) + 4n$$

- Ebből a 8.29. tétellel kapjuk a bizonyítandó összefüggést:

$$\boldsymbol{MT_{MS}(n),\ mT_{MS}(n) \in \Theta(n \log n)}$$

Versenyrendezés (Tournament sort)*

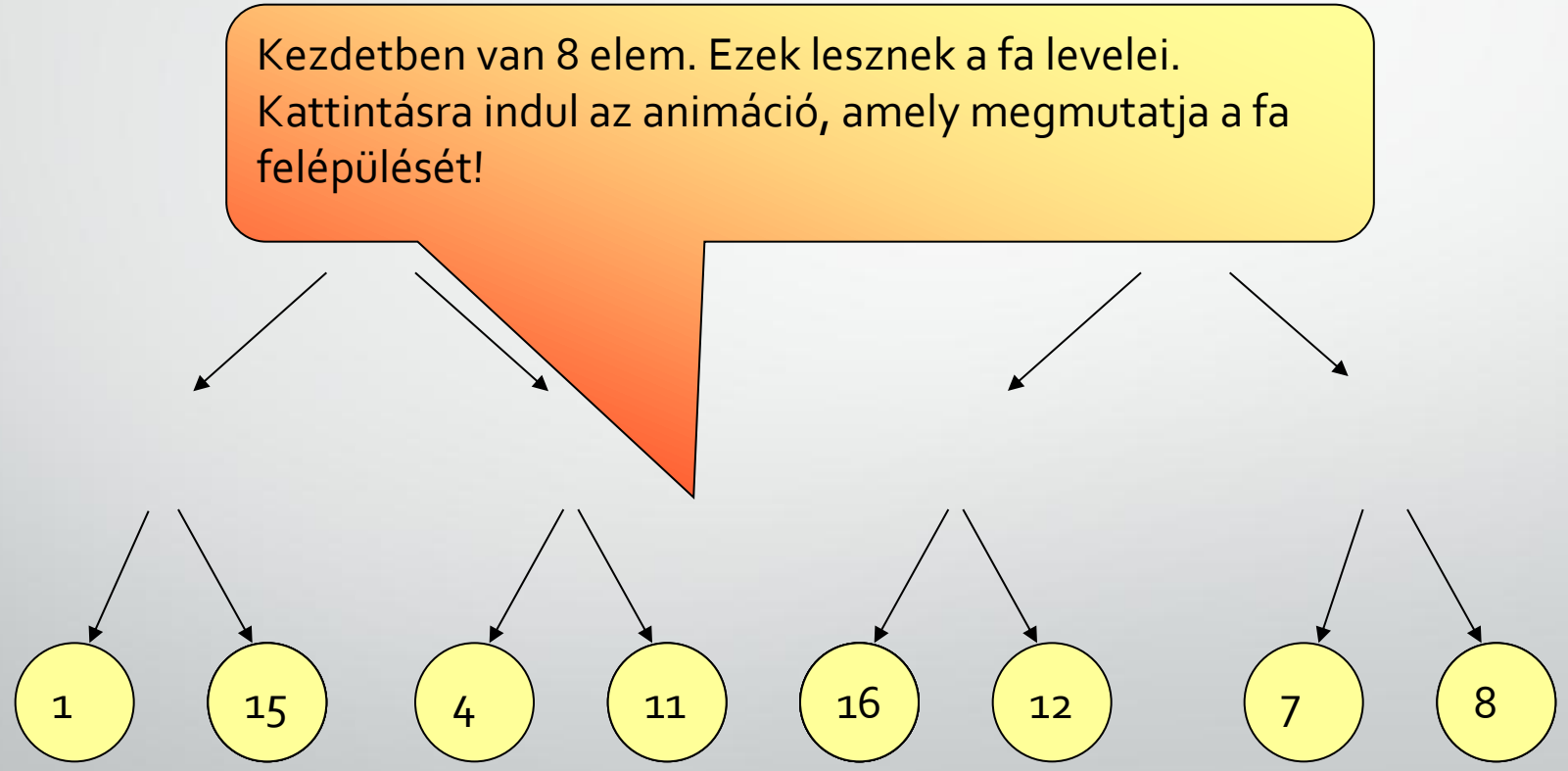
- Maximumkiválasztó rendezések közé tartozik
- ~Kieséses sportversenyek
- Tetszőleges elemszámra megvalósítható ($n=2^k$-ra nézzük)
- Adatszerkezet: versenyfa (tournament)
  - Fa levelei: elemek
  - Belső csúcsok: „nyertes elemek”

Versenyrendezés (Tournament sort)*

- Algoritmus:
  - 1. speciális menet
    - Versenyfa felépítése
  - (n-1) menet
    - Az akt. Max. elem helyett $-\infty$
    - Az ág újra játszása
- Dinamikus programozás elve
  - feljegyzi egy adott menet maximum kiválasztásának az összehasonlításait

Kezdő tournament létrehozása*

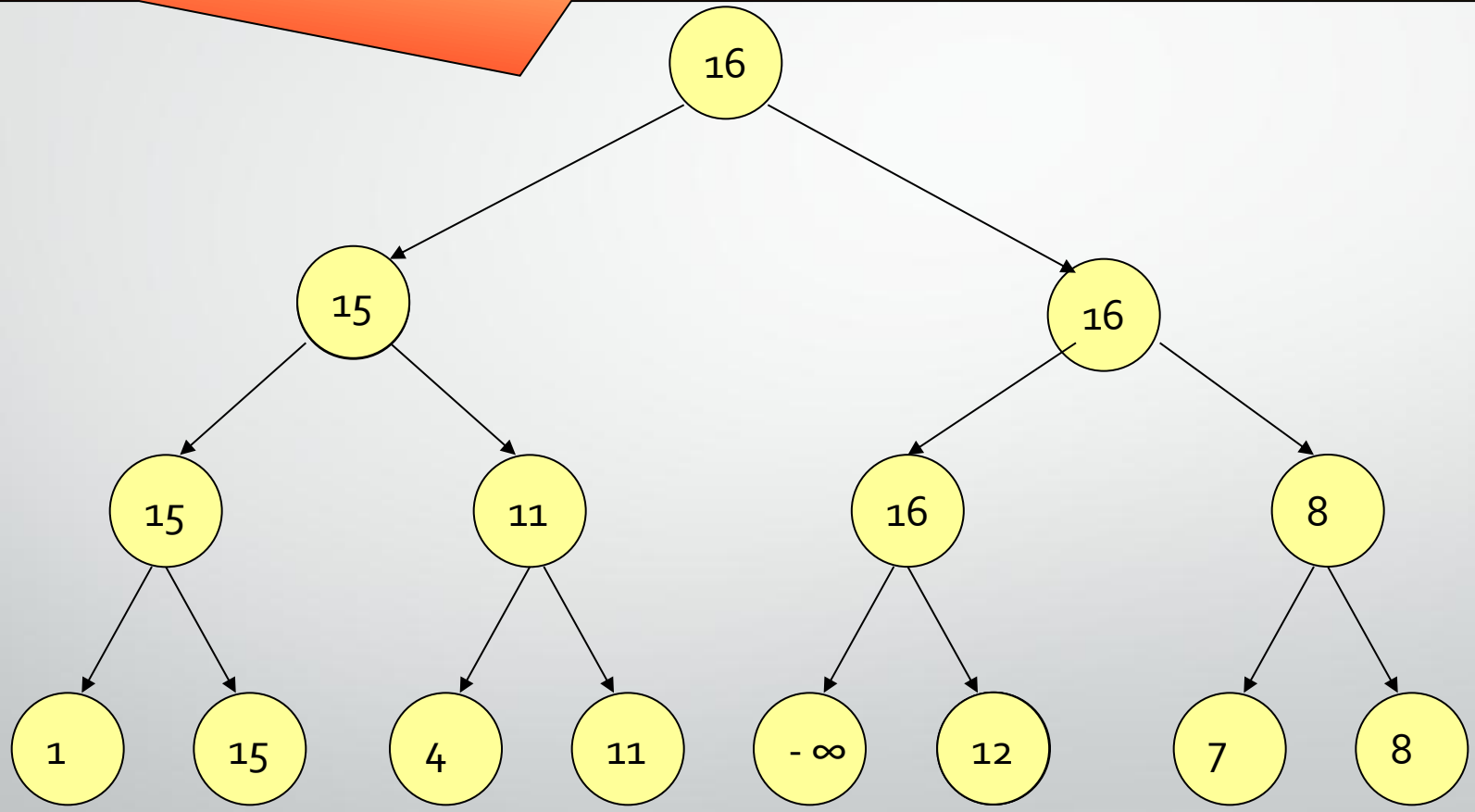
Def.: A tournament olyan teljes bináris fa, amelynek minden belső pontjában a két gyermek nagyobbika foglal helyett.



A mérkőzések újrajátszása a tournament egy ágán*

A rajzon látható, hogy a maximális 16 érdekli helyére (-∞)-t írtunk, majd lejátszunk ezen az ágon 3 mérkőzést, azaz végrehajtunk 3 összehasonlítást:
Kattintásra indul az animáció, amely bemutatja ezt!

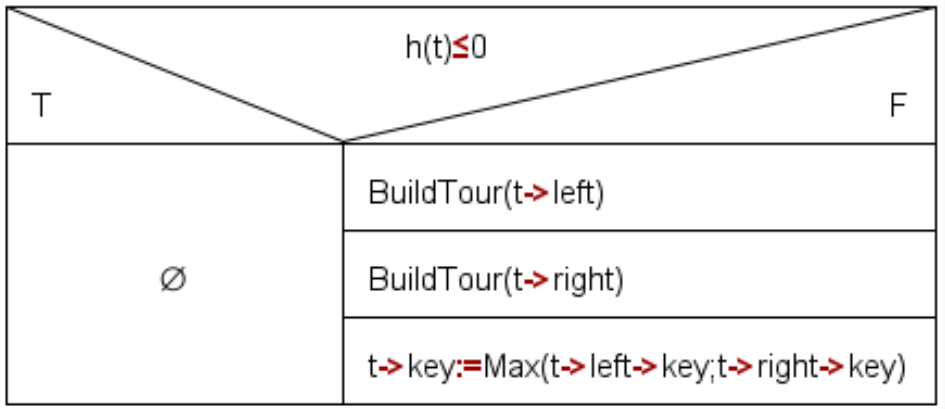
Az újrajátszott ág győztese a második helyére (-∞)-t írtunk.
Ezt (n-1)-szer ismételjük.



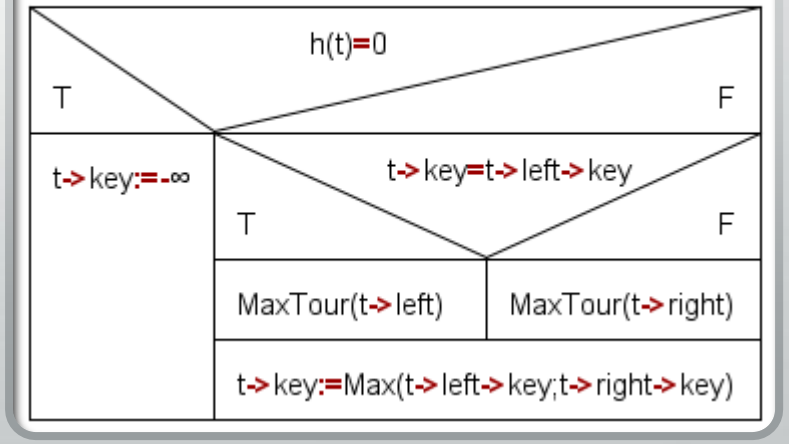
A versenyrendezés rekurzív algoritmusa*

- Műveletigény
  - BuildTour
    - a versenyzők száma: $n=2^k$
    - $\ddot{O}_{BT}(n) = 1 + 2 + \ldots + 2^{k-1} + 2^k = 2^k - 1 = n - 1$
  - MaxTour
    - $\ddot{O}_{MT}(n) = 2k = 2\log_2 n$

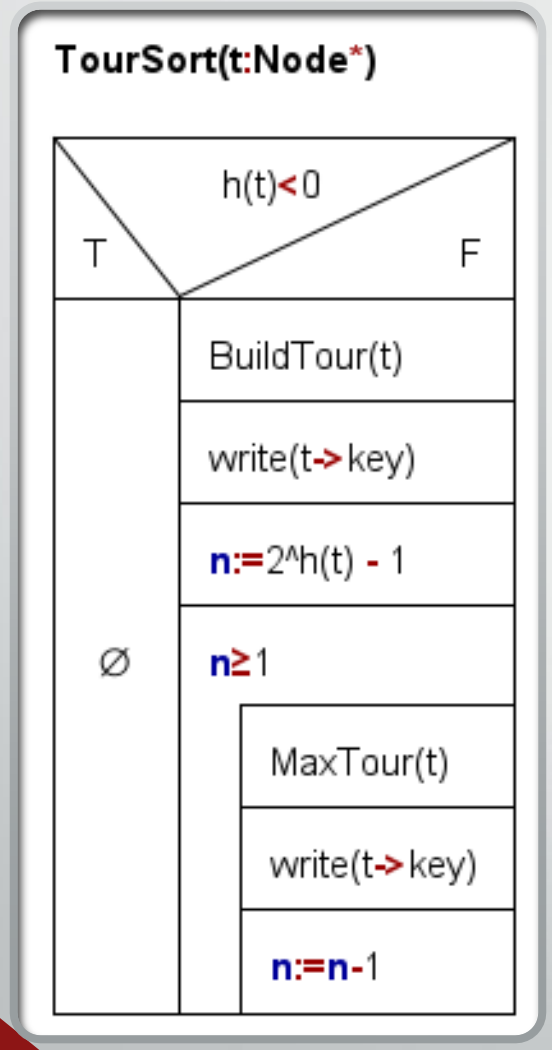
BuildTour(t:Node*)
//Felt: A t fa levelei már tartalmazzák a rendező adatsorozatot



MaxTour(t:Node*)



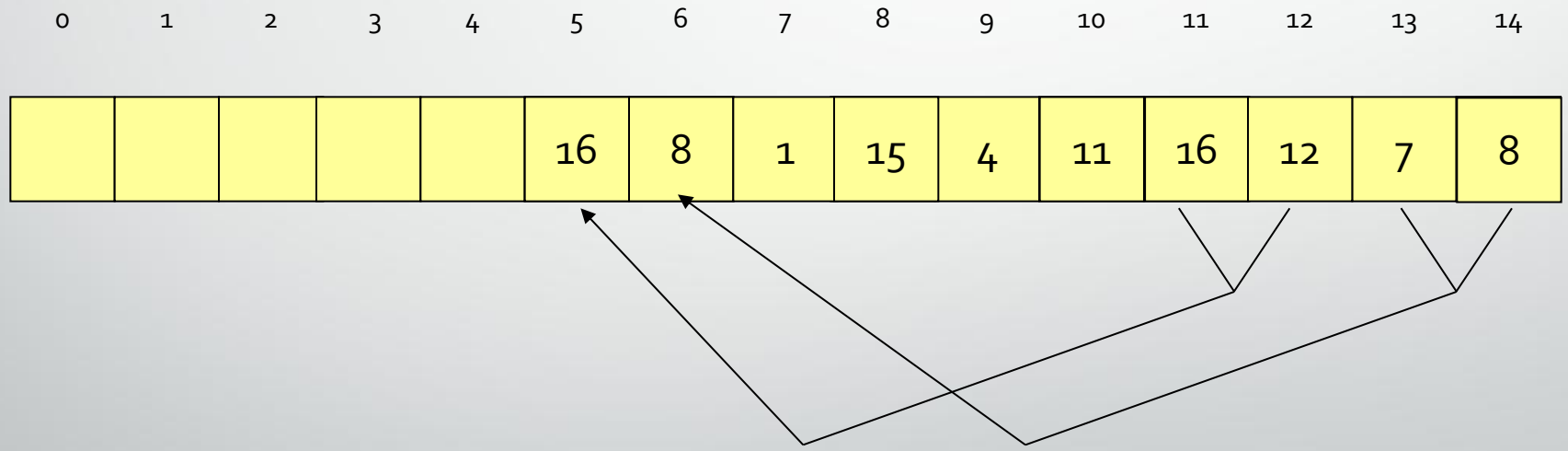
A versenyrendezés rekurzív algoritmusa*



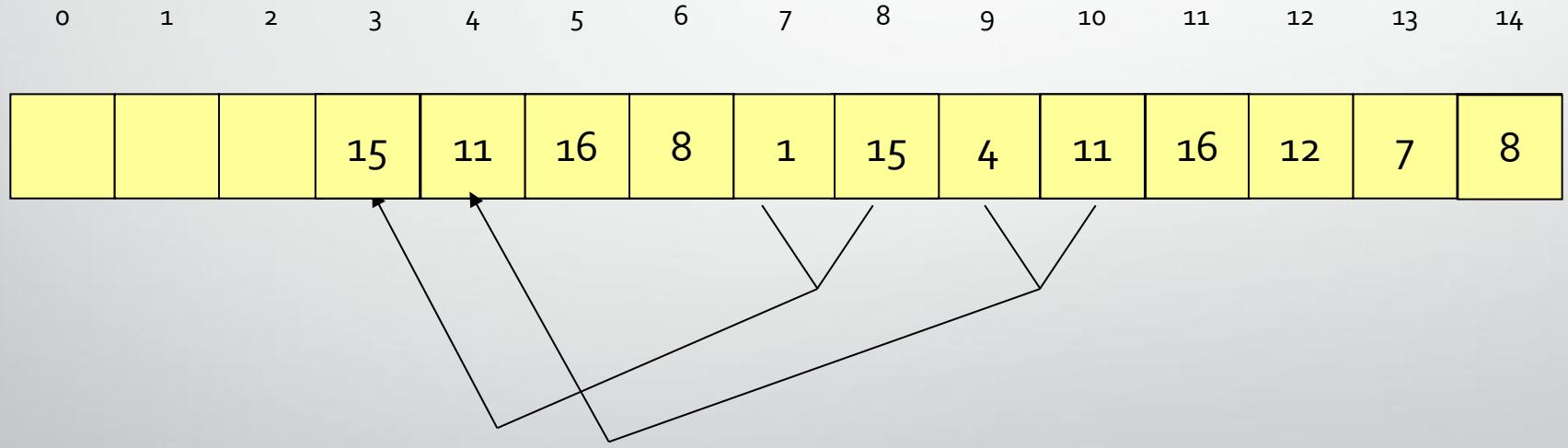
- Műveletigény
  - A teljes eljárás összehasonlítás száma:
    - $\ddot{O}_{TS}(n) = (n\text{-}1) + (n\text{-}2)\ 2\log_2 n = (n\text{-}1)(2\log_2 n + 1)$
    - $\ddot{O}_{TS}(n) \in \Theta(n \log_2 n)$
  - Memória igény: $2n - 1$
- Nem helyben rendez!

TourSort iteratív változata*

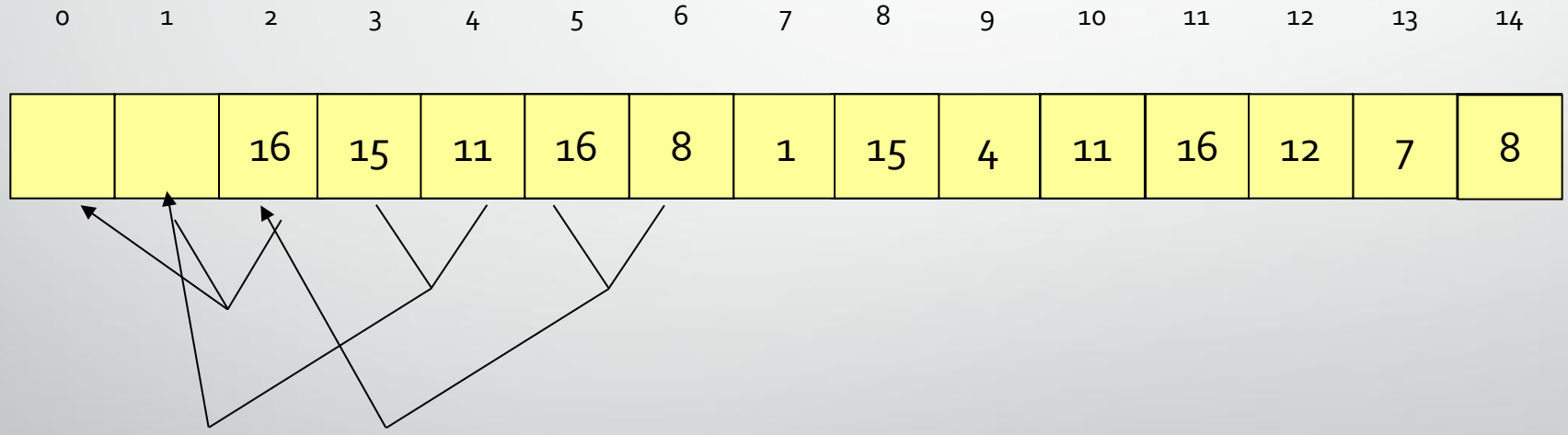
A versenyfát szintfolytonosan elhelyezzük egy tömbben. A rendezendő elemek a fa levelén foglalnak helyet, ezért a *2n-1*elemszámú tömb utolsó *n* eleme van kitöltve. A megelőző elemet éppen a Ktour eljárás tölti ki, méghozzá jobbról balra haladva. Kattintásra indul az animáció, amely bemutatja ezt!



TourSort iteratív változata*



TourSort iteratív változata*



A versenyrendezés iteratív algoritmusa*

BuildTour(A:T[2n-1])

```
i:=n-2 downto 0
    A[i]:=max(A[2i+1], A[2i+2])
```

MaxTour(A:T[2n-1])

```
j:=0
j<n-1
    A[j]=A[2j+1]
    T                F
    j:=2j+1          j:=2j+2
A[j]:=-∞
j:=floor((j-1)/2)
j≥0
    A[j]:=Max(A[2j+1],A[2j+2])
    j:=floor((j-1)/2)
```

TourSort(B:T[n])

```
A:T[2n-1]; A[n-2..2n-2]:=B[0..n)
BuildTour(A)
write(A[0])
i:=n-1 downto 1
    MaxTour(A)
    write(A[0])
```

Ellenőrző kérdések

1. Adott az A = ⟨8; 6; 8; 5; 4; 5; 2; 3; 4; 2⟩ tömb, ami egy bináris maximum-kupacot reprezentál.
   - Szemléltessük a 9, a 8, a 7 és a 3 beszúrását a fenti kupacba! Mindegyik esetben a fent adott kupacra alkalmazzuk a műveletet!
   - Szemléltessünk sorban három remMax() műveletet erre a kupacra!
2. Egy bináris fa mikor szigorúan bináris? Mikor tökéletes? Mikor majdnem teljes? Ez utóbbi mikor balra tömörített, és mikor kupac?
   - Szemléltesse a kupacrendezést a következő tömbre! <3; 9; 8; 2; 4; 6; 7; 5>
   - Minden lesüllyesztés előtt jelölje a csúcs mellett egy kis körbe tett sorszámmal, hogy ez a rendezés során a hányadik lesüllyesztés; akkor is, ha az aktuális lesüllyesztés nem mozdítja el a csúcsban lévő kulcsot! Minden valódi lesüllyesztés előtt jelölje a lesüllyesztés irányát és útvonalát! Minden olyan lesüllyesztés előtt rajzolja újra a fát, ami az aktuális ábrán már módosított csúcsokat érinti! Újra rajzoláskor adja meg a fát tartalmazó tömb pillanatnyi állapotát is!

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek I. előadásjegyzete és Fekete István: Algoritmusok és adatszerkezetek Előadásjegyzete alapján készült.

Algoritmusok és adatszerkezetek I.
9. Előadás

Rendezés lineáris időben

(Sorting in linear time)

Tartalom

Rendező algoritmus

- **Tétel:** Tetszőleges rendező algoritmusra $mT(n) \in \Omega(n)$.
- **Tétel:** Bármely összehasonlító rendezés végrehajtásához a legrosszabb esetben $MC(n) \in \Omega(n \lg n)$ kulcsösszehasonlítás szükséges.
- **Tétel:** Tetszőleges összehasonlító rendezésre $MT(n) \in \Omega(n \lg n)$.
- Lehet-e más elven működő algoritmussal ezen javítani?
  - A kulcsok összehasonlítása helyett **kulcsok osztályozása** (aszimptotikusan optimális)
    - radix sort (számjegypozíciós rendezés)
    - szétválogató rendezés (distributing sort)
    - counting sort (leszámláló rendezés)
  - + kulcsösszehasonlító segédrendezés
    - Bucket sort (egyszerű edényrendezés) ($AT(n)$, $mT(n) \in \Theta(n)$)
- *S* rendezés **aszimptotikusan optimális,** ha $MT_S(n) \in O(n)$
  - $MT_S(n)$, $mT_S(n) \in \Theta(n)$

Radix rendezés
(Radix Sort = Számjegypozíciós rendezés)

- A kulcsok
  - ***r*** alapú számrendszerben
  - *d* számjegyű,
  - **dDigitNumber** típusú: előjel nélküli egész szám
  - Számjegyek sorszámozása: jobbról balra
    - 1. szj: legkevésbé szignifikáns, …
    - a *d*-edik a legszignifikánsabb

Helyiértékek sorszámai (d=5, r=4, kulcs:10320):

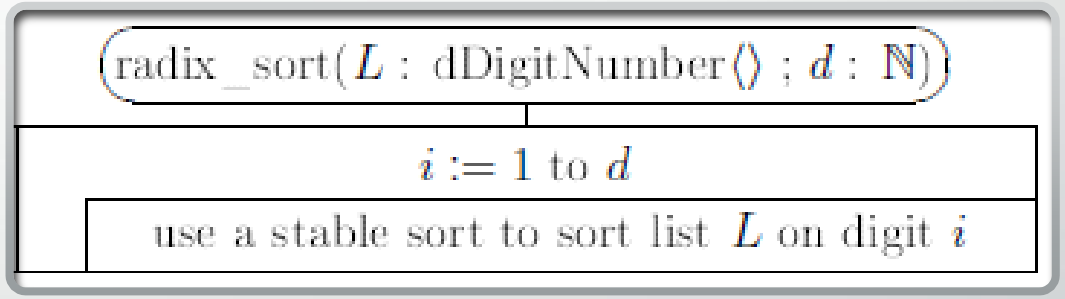
5	*4*	*3*	*2*	*1*
1	**0**	**3**	**2**	**0**

- A rendezés
  - *d* menet
    - az *i*-edik menetben a jobbról *i*-edik számjegy szerint rendezünk
  - **stabil rendezés**
  - *L*: absztrakt lista (véges sorozat)
    - dDigitNumber⟨⟩: olyan véges sorozatok, amiknek az elemtípusa: dDigitNumber

```
radix_sort(L : dDigitNumber⟨⟩ ; d : ℕ)
    i := 1 to d
        use a stable sort to sort list L on digit i
```

Radix rendezés lineáris műveletigény

- Műveletidő: $\Theta\big(d * rendezés(n)\big)$
  - a $rendezés(n)$: a belső rendezés költsége
- Elvárások a belső rendezéstől
  - Stabil
  - Lineáris műveletigény
- Belső rendezés
  - Szétválogató rendezés (distributing sort)
    - láncolt listáknál
  - Leszámláló rendezés (counting sort)
    - tömböknél



Szétválogató rendezés (distributing sort)

- Stabil
- Láncolt listákra optimális
- Hatékony a Radix rendezéssel
- **A rendezési probléma:**
  - Adott:
    - az n hosszúságú
    - T elemtípusú
    - L absztrakt lista,
    - $r \in O(n)$ $(\in \mathbb{P})$
    - $\varphi : T \rightarrow [0..r)$ kulcskiválasztó függvény.
  - Rendezzük L-et stabil rendezéssel, lineáris műveletigénnyel!
- **Műveletigény**:

$\Theta(n + r) = \Theta(n)$ ($r \in O(n)$ miatt)

```
distributing_sort(L : T⟨⟩ ; r : ℕ ; φ : T → [0..r))
B : T⟨⟩[r] // array of lists, i.e. bins                                  Θ(r)
k := 0 to r−1
    Let B[k] be empty list // init the array of bins
L is not empty                                                           Θ(n)
    Remove the first element x of list L                                 Θ(1)
    Insert x at the end of B[φ(x)] // stable distribution                Θ(1)
k := r−1 downto 0                                                        Θ(r)
    L := B[k] + L // append B[k] before L
```

Radix rendezés listákra

- A radix rendezés által felhasznált szétválogató rendezés
  - r alapú számrendszer
  1. számok szétválogatása
     - i-edik számjegyük értéke szerint
     - $B_0, B_1, B_2, \ldots B_{r-1}$ polcokra (bins)
       - Polcok: kezdetben üresek
     - fontos:
       - az egyes polcokon **megmaradjon** a számoknak **a szétválogatás előtti sorrendje**
  2. a polcok tartalmát összefűzi L-be.

$$T_{radix\_sort}(n) \in \Theta(n)$$

- Műveletigény:
  - Belső (szétválogató rendezés): $\Theta(n + r)$
    - r darab polc üresre inicializálása
    - n elemet szétválogatása
    - r db polc összefűzése
  - A radix rendezés: $\Theta(d * (n + r))$
    - a belső (stabil) rendezést d-szer hívja meg
    - d konstans és $r \in O(n)$

Radix rendezés listákra - lejátszás

L = <103, 232, 111, 013, 211, 002, 012> ($r = 4$; $d = 3$)

- 1. mentet (utolsó szj. szerint)
  - $B_0 = \langle\rangle$
  - $B_1 = \langle 111, 211 \rangle$
  - $B_2 = \langle 232, 002, 012 \rangle$
  - $B_3 = \langle 103, 013 \rangle$
- Összefűzve: $L = \langle 111, 211, 232, 002, 012, 103, 013 \rangle$
- 2. Menet: (utolsó előtti szj. szerint)
  - $B_0 = \langle 002, 103 \rangle$
  - $B_1 = \langle 111, 211, 012, 013 \rangle$
  - $B_2 = \langle\rangle$
  - $B_3 = \langle 232 \rangle$
- Összefűzve: $L = \langle 002, 103, 111, 211, 012, 013, 232 \rangle$
- 3. Menet: (első szj. szerint)
  - $B_0 = \langle 002, 012, 013 \rangle$
  - $B_1 = \langle 103, 111 \rangle$
  - $B_2 = \langle 211, 232 \rangle$
  - $B_3 = \langle\rangle$
- Összefűzve: $L = \langle 002, 012, 013, 103, 111, 211, 232 \rangle$

Az absztrakt listák ha nem láncolt listák

- Az egyes polcok tételszáma
  - nem ismert: [0..n] között (n :az input mérete)
    - r darab n méretű segédtömb
      - túl sok munkamemória
- Ha az interfész egy tömb, de a polcok láncolt listák
  - összesen $n * d$ memória allokálás, és ugyanennyi deallokálás
    - Nagyon megnöveli a $\Theta(n)$ műveletigényben rejtett „konstanst”
      - a rendezés gyakorlatilag használhatatlan

A gyakorlatban az absztrakt listák: láncolt listák

- Ha a polcok láncolt listák
  - A bemenet és a kimenet is a polcokkal azonos típusú láncolt lista
    - a memória allokálások (pl. new) és deallokálások (pl. delete) elkerülése miatt
  - Az egyes menetek bemeneti listája elemeinek szétpakolása hatékony legyen
    - minden elemre: $\Theta(1)$ idő
  - A listák végéhez is direkt hozzáférés szükséges
- Megvalósítása
  - S1L-ekkel és a nemüres polcok végére mutató pointerekkel
  - C2L-ek alkalmazásával
    - konstans szorzók árán

Radix rendezés összetett kulcsú rekordok rendezésére: pl. Dátum

- három komponenst (év, hó, nap) tartalmaz
- Radix rendezése:
  - Legkevésbé szignifikánstól a legszignifikánsabb felé (3 menet):
    - nap -> hó -> év mező szerint
    - Stabil rendezéssel
  - Már néhány ezer tétel esetén is gyorsabb rendezést kapunk
    - Feltétel: nincs túl sok lehetséges évszám
- Összehasonlító rendezésnél
    - a sorrend: év -> hó -> nap

Radix rendezés listára - algoritmus

- L : fejelemes, C2L típusú lista
  - a listaelemek kulcsai
    - d számjegyű ($d \in \mathbb{P}$ konstans)
    - r alapú számrendszer ($r \in O(n)$)
  - digit(i, r: $\mathbb{N}$, x:$\mathcal{T}$): [0..r)
    - x kulcs jobbról i-dik szj. r alapú szrdsz-ben

```
radix_sort( L : E2* ; d, r : ℕ )
    BinHead : E2[r] // the headers of the lists representing the bins
    B : E2*[r] // pointers to the headers
    i := 0 to r − 1
        B[i] := &BinHead[i] // Initialize the ith pointer.
    i := 1 to d
        distribute(L, i, B) // Distribute L on the ith digits of keys.
        gather(B, L) // Gather form the bins back into L
```

```
distribute( L : E2* ; i : ℕ ; B : E2*[r] )
    L → next ≠ L
        p := L → next ; unlink(p)
        precede( p, B[digit(i, r, p → key)]
```

Radix rendezés listára - algoritmus

gather(B : E2*[r] ; L : E2*)
$i := 0$ to $r - 1$
append($L, B[i]$) // add to the end of L the elements form $B[i]$

Műveletigény: $\Theta(n)$ (n: lista hossza)

- $T_{append} \in \Theta(1)$-> $T_{gather}(r) \in \Theta(r)$
- $T_{distribute}(n) \in \Theta(n)$, ahol $n = |L|$
- $T_{radix\_sort}(n, d, r) \in \Theta(r + d(n + r))$
- d konstans és $r \in O(n)$
- $T_{radix\_sort}(n) \in \Theta(n)$

append(L : E2*, Bi : E2*)	
Bi->next ≠ Bi	
p, q, r := L->prev, Bi->next, Bi->prev	
p->next, q->prev := q, p	
r->next, L->prev := L, r	
Bi->next := Bi->prev := Bi	

Leszámláló rendezés (counting sort)

- Stabil
- Radix sort ideális segédrendezése
  - tömbben $\Theta(r)$
- **A rendezési probléma:** $\Theta(n)$
  - Adott: $\Theta(r)$
    - A: $\mathcal{T}[n]$ tömb $\Theta(n)$
    - $r \in O(n)$ $(\in \mathbb{P})$
    - $\varphi : T \rightarrow [0..r)$ kulcsfüggvény
  - Rendezzük *A* tömböt stabil rendezéssel, lineáris műveletigénnyel -> az eredmény: *B*: $\mathcal{T}[n]$ tömbben keletkezzék
- **Műveletigény**:

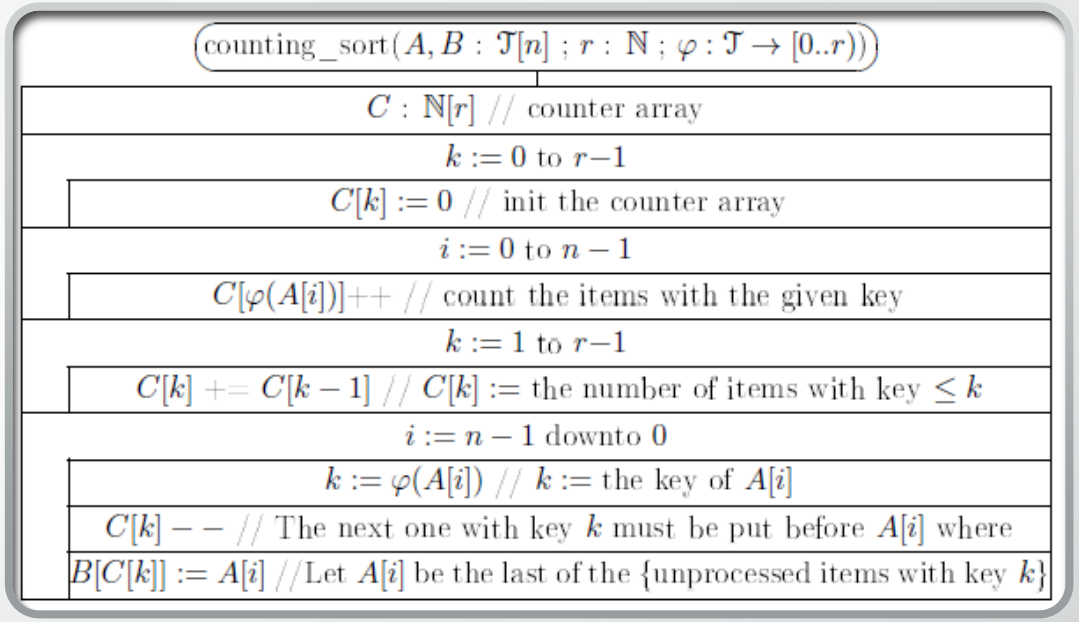
$\Theta(n + r) = \Theta(n)$ ($r \in O(n)$ miatt)

```
counting_sort(A, B : 𝒯[n] ; r : ℕ ; φ : 𝒯 → [0..r))
    C : ℕ[r] // counter array
    k := 0 to r−1
        C[k] := 0 // init the counter array
    i := 0 to n − 1
        C[φ(A[i])]++ // count the items with the given key
    k := 1 to r−1
        C[k] += C[k − 1] // C[k] := the number of items with key ≤ k
    i := n − 1 downto 0
        k := φ(A[i]) // k := the key of A[i]
        C[k] − − // The next one with key k must be put before A[i] where
        B[C[k]] := A[i] //Let A[i] be the last of the {unprocessed items with key k}
```

Leszámláló rendezés magyarázata

1. ciklus:
   - *C* számláló tömb kinullázása
2. ciklus:
   - ∀ *k* kulcsra *a C*[*k*]–ban:
     - *k* kulcsú elemek megszámolása
3. ciklus:
   - ∀ *k* kulcsra *C*[*k*]-ban:
     - ≤ *k* kulcsú elemek számának összegzése
       - ≤ 0 kulcsú elem = 0 kulcsú -> *C*[0] értéke nem változik
       - > *k* kulcsokra: ≤ *k* kulcsú elem = *k* kulcsú + *k*-nál kisebb kulcsú

   ➢ *C*[*k*] új értéke = *C*[*k*] régi értéke + *C*[*k*–1] új értéke
4. ciklus:
   - a bemeneti tömbön visszafelé haladva
   - ∀ elem elhelyezése az eredmény tömbbe



4. ciklus részletezése: *k* kulcsú elem feldolgozása

- Fordított sorrendben: (először az utolsó *k* kulcsú elemet)
  - Utolsó helyre, ahova *k* kulcsú elem kerülhet
- *C*[*k*] :hány darab legfeljebb *k* kulcsú elem van a bemeneten

➢ pontosan az eredmény tömb *C*[*k*]-adik (*C*[*k*]–1 indexű) elemébe tesszük

1. *C*[*k*]-t eggyel csökkentjük
2. az aktuális elemet *B*[*C*[*k*]]-ba másoljuk

4. ciklus részletezése: *k* kulcsú elem feldolgozása

- A fordított bejárás szerint következő *k* kulcsú elem közvetlenül a mostani elé fog kerülni stb.
  - *k* kulcsra a *k* kulcsú elemek eredeti sorrendje megmarad az eredmény tömbbe is
  - kisebb kulcsú elemek megelőzik a nagyobb kulcsúakat
    - stabil rendezést kapunk

Leszámláló rendezés C# kódja*

```csharp
static void Main()
{
    int[] A = { 4, 2, 2, 8, 3, 3, 1 };
    int maxValue = 9; // r: the range of values is [0..r)
    int[] B = new int[A.Length];

    // Identity key function (used when values are keys themselves)
    CountingSort(A, B, maxValue, x => x);

    Console.WriteLine("Sorted array:");
    Console.WriteLine(string.Join(", ", B));
}
```

```csharp
class CountingSortAlgorithm
{
    public static void CountingSort(int[] A, int[] B, int r, Func<int, int> keyFunc)
    {
        int n = A.Length;
        int[] C = new int[r]; // Counter array

        // Step 1: Initialize the counter array
        for (int k = 0; k < r; k++)
        {
            C[k] = 0;
        }

        // Step 2: Count the items with the given key
        for (int i = 0; i < n; i++)
        {
            C[keyFunc(A[i])]++;
        }

        // Step 3: Compute prefix sums (number of items with key ≤ k)
        for (int k = 1; k < r; k++)
        {
            C[k] += C[k - 1];
        }

        // Step 4: Place items into the output array in sorted order
        for (int i = n - 1; i >= 0; i--)
        {
            int k = keyFunc(A[i]);
            C[k]--;
            B[C[k]] = A[i];
        }
    }
```

Leszámláló rendezés (animációval):

A=<21, 30, 20, 33, 31, 10>

- Leszámlálás a második számjegy szerint:

C		21	30	20	33	31	10	GY	KGY	10	31	33	20	30	21
0	0														
1	0														
2	0														
3	0														

B=	0	1	2	3	4	5

- B= <30, 20, 10, 21, 31, 33>

Leszámláló rendezés animáció nélkül:

A=<21, 30, 20, 33, 31, 10>

- Leszámlálás a második számjegy szerint:

C		21	30	20	33	31	10	GY	KGY	10	31	33	20	30	21
0	0		1	2			3	3	3	2			1	0	
1	0	1				2		2	5		1				0
2	0							0	5						
3	0				1			1	6			0			

Helyre rakáskor a C tömbből olvassuk ki, hova kell tenni az elemet miután csökkentettük eggyel C megfelelő értékét.

B=

0	1	2	3	4	5
30	20	10	21	31	33

- B= <30, 20, 10, 21, 31, 33>

Leszámláló rendezés folytatása (animációval):

B=< *30, 20, 10, 21, 31, 33* >

- Leszámlálás az első számjegy szerint:

C		30	20	10	21	31	33	GY	KGY	33	31	21	10	20	30
0	0														
1	0														
2	0														
3	0														

A =	0	1	2	3	4	5

- A=<*10,20,21,30,31,33*>
- Páros darabszámú számjegy esetén a rendezendő számok épp abban a tömbben vannak, amelyikben eredetileg voltak.

Leszámláló rendezés folytatása animáció nélkül

B=< *30, 20, 10, 21, 31, 33* >

- Leszámlálás az első számjegy szerint:

C		30	20	10	21	31	33	GY	KGY	33	31	21	10	20	30
0	0							0	0						
1	0			1				1	1				0		
2	0		1		2			2	3			2		1	
3	0	1				2	3	3	6	5	4				3

A =	0	1	2	3	4	5
	10	20	21	30	31	33

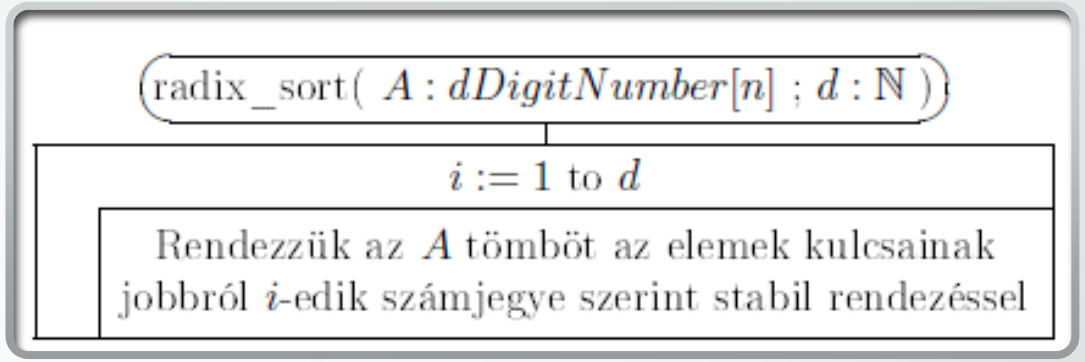
- A=<*10,20,21,30,31,33*>
- Páros darabszámú számjegy esetén a rendezendő számok épp abban a tömbben vannak, amelyikben eredetileg voltak.

Radix rendezés (Radix-Sort) tömbökre

- A rendezendő *A* tömb kulcsai
  - *r* alapú számrendszer
  - *d*-jegyű $\mathbb{P}_0$
    - A jobbról 1. szj helyiértéke: min
    - A jobbról a *d.* szj helyiértéke: max
- Műveletigény: $\Theta(d * (n + r))$
  - a rendezés *d* leszámláló rendezésből áll
  - *d* konstans és $r \in O(n)$

➢ $\Theta(d*(n + r)) = \Theta(n)$

➢ $T_{radix\_sort}(n) \in \Theta(n)$



- pl. a kulcsok négy bájtos nemnegatív egészek
  - számjegyek: a számok bájtjai
  - $d = 4$ és $r = 256$
  - mindkettő *n*-től független konstans

➢ a feltételek teljesülnek

➢ a rendezés lineáris időben lefut

- az i-edik számjegy: (C++):(key >> (8 * (*i* – 1)))&255

Egyszerű edényrendezés (bucket sort)

- A rendezendő elemek kulcsai [0;1) valós iv. elemei
  - Kulcsok normálása
    - $k \in \mathbb{Z}$ || $k \in \mathbb{R}$ || azzá konvertálható
    - k kulcsra $k \in [a; b)$
    - ➢ $\frac{k-a}{b-a} \in [0;1)$
- Egyenletes eloszlás
  - (input kulcsai a [0; 1) iv-on)
- Műveletigény
  - $mT(n) \in \Theta(n)$
  - $AT(n) \in \Theta(n)$ (egyenletes eloszlást felt.)
  - $MT(n)$ ~ milyen rendezést választunk az edények (buckets) rendezésére
    - Beszúró rendezés: $MT(n) \in \Theta(n^2)$,
    - összefésülő rendezés: $MT(n) \in \Theta(n \log n)$

```
bucket_sort( L : list )
    n := the length of L
    B : list[n] // Create the buckets B[0..n)]
    j := 0 to (n−1)
        Let B[j] be empty list
    L ≠ ⦰
        Remove the first element of list L
        Insert this element according to its key k into list B[⌊n ∗ k⌋]
    j := 0 to (n−1)
        Sort list B[j] nondecreasingly
    Append lists B[0], B[1], . . . , B[n − 1] in order into list L
```

Edényrendezés C# kódja*

```
    static void Main()
    {
        List<double> list = new List<double> { 0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68 };
        Console.WriteLine("Original list:");
        Console.WriteLine(string.Join(", ", list));

        BucketSort(list);

        Console.WriteLine("Sorted list:");
        Console.WriteLine(string.Join(", ", list));
    }
}
```

```
class BucketSortAlgorithm
{
    public static void BucketSort(List<double> L)
    {
        int n = L.Count;
        if (n <= 0) return;

        // Create n empty buckets
        List<List<double>> B = new List<List<double>>(new List<double>[n]);
        for (int j = 0; j < n; j++)
        {
            B[j] = new List<double>();
        }

        // Put elements into buckets
        foreach (double x in L)
        {
            int index = (int)(n * x); // Assumes x is in range [0, 1)
            B[index].Add(x);
        }
        L.Clear();
        // Sort individual buckets and Concatenate all buckets into L
        for (int j = 0; j < n; j++)
        {
            B[j].Sort();
            L.AddRange(B[j]);
        }
    }
```

Edényrendezés lejátszása

A < 0,45; 0,19; 0,63; 0,36; 0,20; 0,81; 0,12; 0,77; 0,54; 0,78 > tíz elemű listán mutassa be a bucket sort algoritmusát [0; 1)-beli kulcsokra!

- A listák típusa:
  - egyszerű egyirányú lista (S1L)
- Kezdeti lista:
  - <0,45; 0,19; 0,63; 0,36; 0,20; 0,81; 0,12; 0,77; 0,54; 0,78>
- Végeredmény:
  - <0,12; 0,19; 0,20; 0,36; 0,45; 0,54; 0,63; 0,77; 0,78; 0,81>

	Beszúrás után	Rendezés után
0		
1	0,12 0,19	0,12 ; 0,19
2	0,20	
3	0,36	
4	0,45	
5	0,54	
6	0,63	
7	0,78 0,77	0,77 ; 0,78
8	0,81	
9		

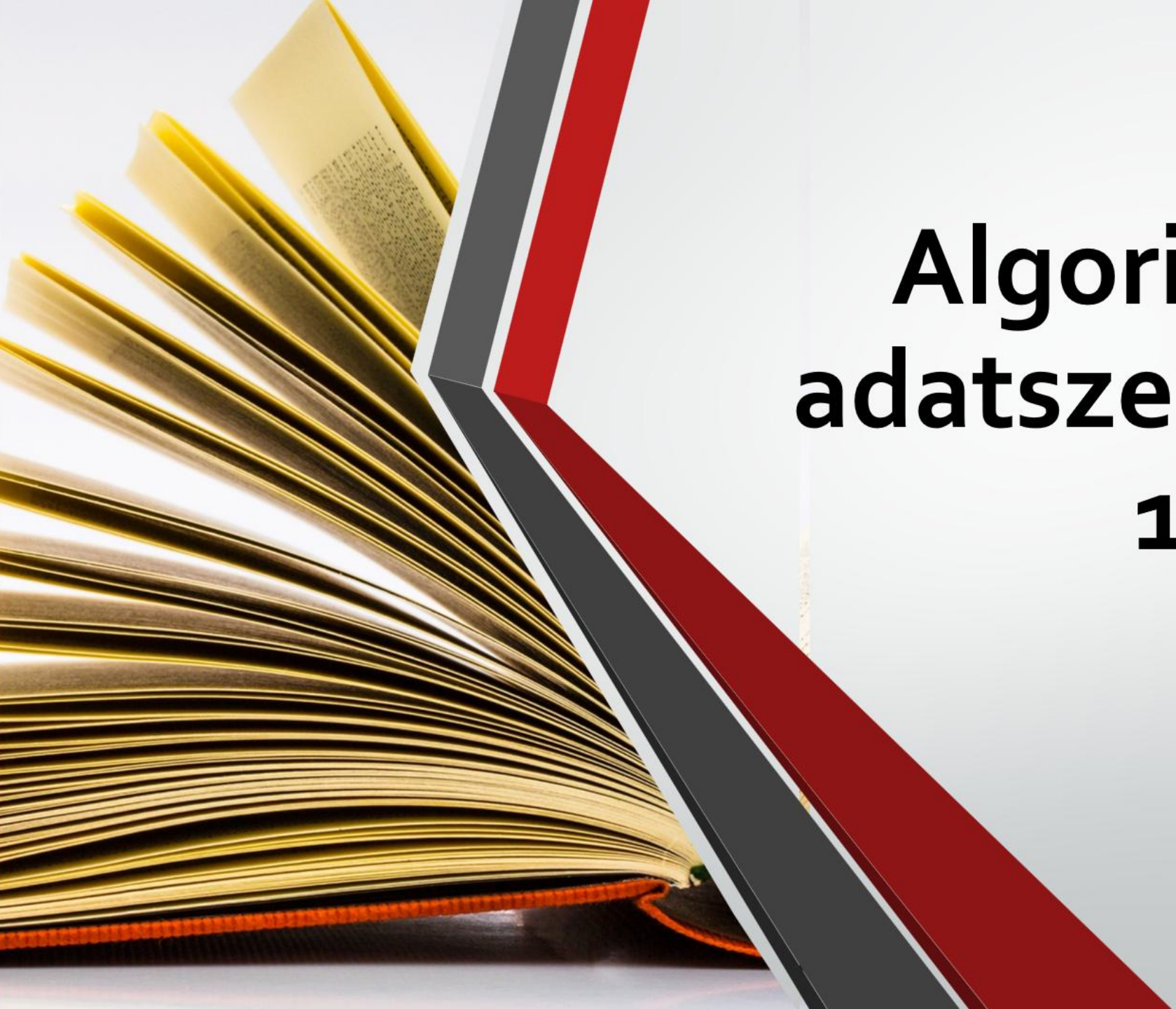
Ellenőrző kérdések

1. Mutassa be a számjegypozíciós (Radix) rendezés működését a <20; 02; 21; 01; 31; 20> négyes számrendszerbeli számok tömbjén! Az egyes menetekben leszámláló rendezést alkalmazzon!
2. Mutassa be a számjegypozíciós (Radix) rendezés működését a <h31; 20; 11; 23; 21; 10> négyes számrendszerbeli számok listáján! Az egyes menetekben a megfelelő számjegy szerinti szétválogató rendezést alkalmazzon!
3. A <0,4; 0,82; 0,0; 0,53; 0,73; 0,023; 0,64; 0,7; 0,39; 0,203> tíz elemű listán mutassa be a bucket sort algoritmusát [0; 1)-beli kulcsokra!
4. Mekkora a fenti rendezések aszimptotikus műveletigénye, és miért?
5. A leszámláló rendezés/szétválogató rendezés – mint segédprogram- mely tulajdonságára épül a Radix rendezés? Mit jelent ez a tulajdonság az adott esetben?
6. Adja meg a fenti rendezések struktogramjait!

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek I. előadásjegyzete alapján készült.



Algoritmusok és adatszerkezetek I.
10. Előadás

Hasitó táblák

(hash tables)

Tartalom

Szótárak (dictionaries)

- Szótárak műveletei:
  - adat **beszúrása** a szótárba
  - kulcs alapján a szótárban a hozzá tartozó adat meg**keresés**e
  - a szótárból adott kulcsú, vagy egy korábbi keresés által lokalizált adat **eltávolítás**a
- Szótárak megvalósítása:
  - AVL fák
  - B+ fák (ld. a következő félévben)
  - egyéb kiegyensúlyozott keresőfák
  - **hasító táblák**
    - Műveleteknek az **átlagos (**$\Theta(1)$**)** (nem a maximális ($\Theta(n)$)) futási idejét minimalizáljuk

Jelölések:

- m : a hasító tábla mérete
- $T[0..m)$: a hasító tábla
- $T[0], T[1],\ldots,T[m-1]$: a hasító tábla rései (slot-jai)
- $\oslash$: üres rés a hasító táblában (direkt címzésnél és a kulcsütközések láncolással való feloldása esetén)
- E: üres rés (empty slot) kulcsa a hasító táblában (nyílt címzésnél)
- D: törölt rés (deleted slot) kulcsa a hasító táblában (nyílt címzésnél)

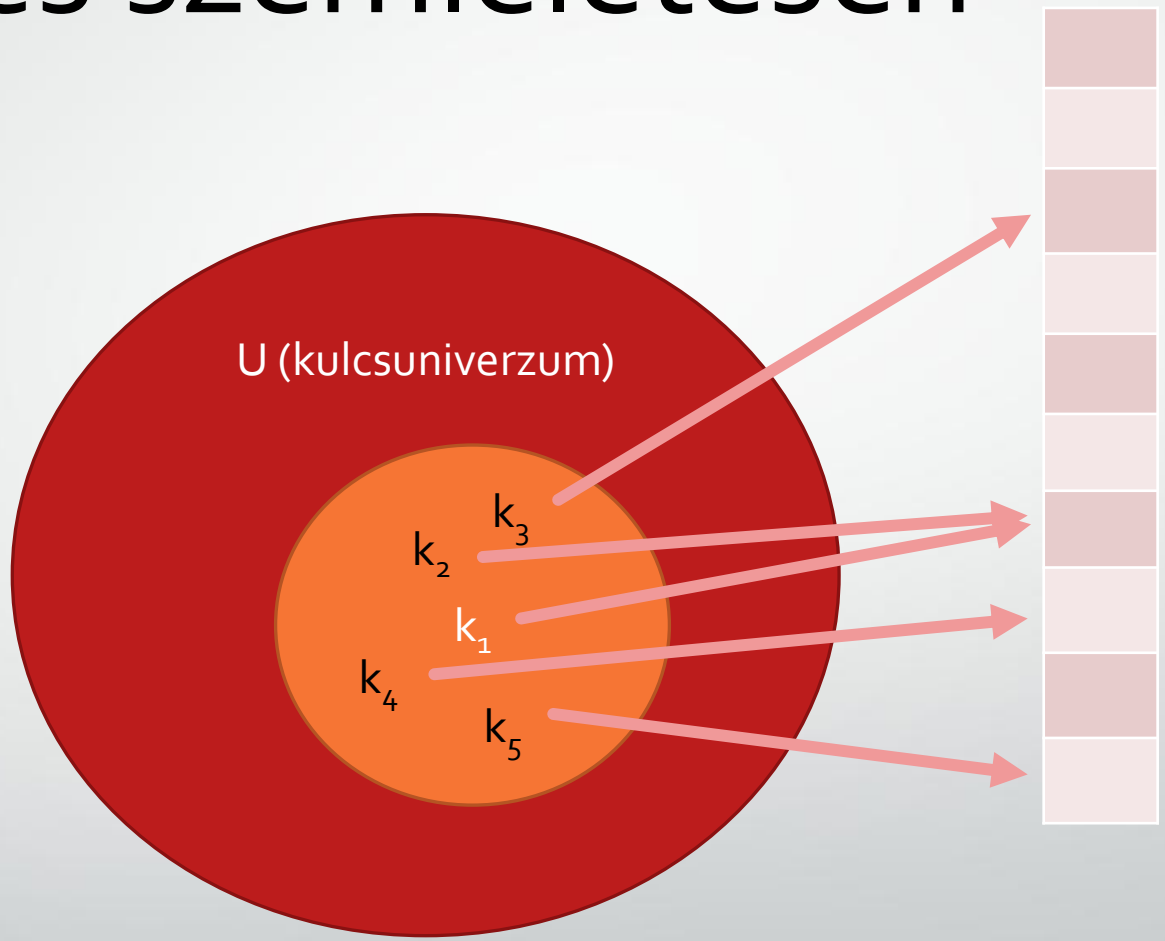
Jelölések:

- n: a hasító táblában tárolt adatok száma
- $\alpha = n/m$: a hasító tábla kitöltöttségi aránya (load factor)
- U: a kulcsok univerzuma: k, k', $k_i \in U$
- h: U $\rightarrow$ 0.. (m-1): hasító függvény (hash function)

Feltevések:

- a hasító tábla nem tartalmazhat két vagy több azonos kulcsú elemet
- h(k) $\theta(1)$ időben számolható

Hashelés szemléletesen

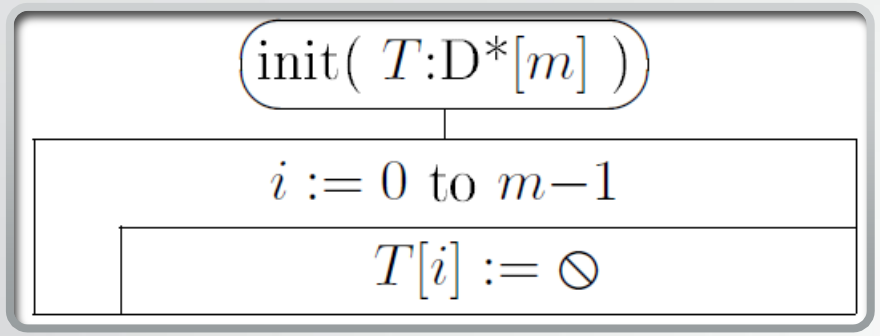


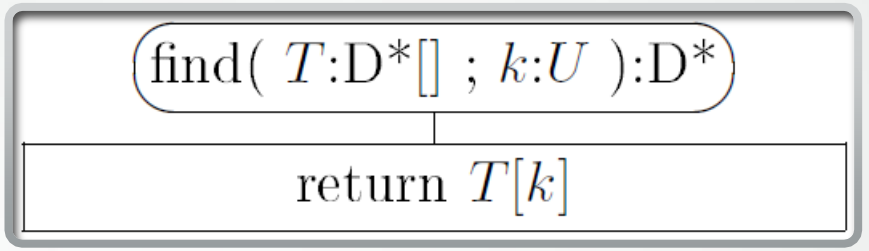
Direkt címzés (direct-address tables)

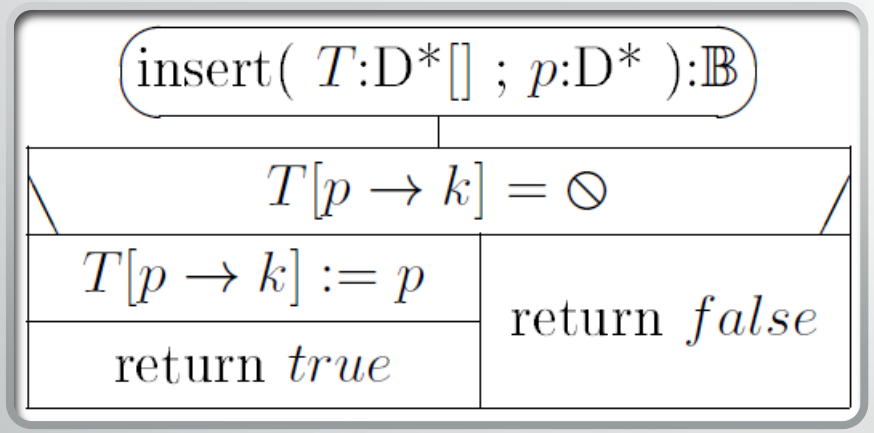
- $U = [0..m)$, ahol $m \geq n$, de m nem túl nagy
- $T : D*[m]$ hasító tábla rései pointerek
  - D típusú adatrekordokra mutatnak
- A rekordok:
  - $k : U$ kulcsmező
  - járulékos mezők
- A hasító tábla inicializálása: ⦸ pointerekkel

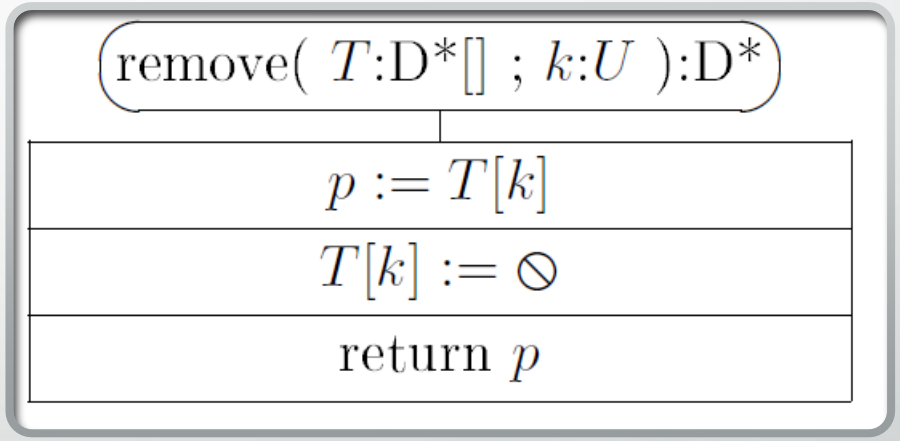
D
+ $k : U$ // k is the key
+ ... // satellite data

Direkt címzés műveletek









- Műveletigény:

$T_{init}(m) \in \Theta(m)$

$T_{műveletek}(m) \in \Theta(1)$

Hasító táblák (hash tables)

- Hasító függvény (hash function):
  - $|U| >> n$ -> a direkt címzés nem alkalmazható, vagy nem gazdaságos
  - ➢ $h : U \rightarrow 0..(m-1)$ hasító függvényt alkalmazunk
  - $|U| >> m$
  - A k kulcsú adatot a $T[0..m)$ hasító tábla $T[h(k)]$ résében tároljuk (próbáljuk tárolni).
    - a hasító táblában minden rekord kulcsa egyedi

!! **Hasító függvény egyszerű egyenletes hasítás legyen**

Hasító táblák (hash tables)

- A $h : U \rightarrow [0..m)$ függvény **egyszerű egyenletes hasítás**
  - simple uniform hashing
  - a kulcsokat a rések között egyenletesen szórja szét
    - hozzávetőleg ugyanannyi kulcsot képez le az m rés mindegyikére.
- Kulcsütközések (key collisions):
  - Ha két adat $k_1 \neq k_2$ kulcsára $h(k_1) = h(k_2)$
  - $|U| >> m$ -> kulcsütközés szinte biztosan előfordul -> kezelni kell

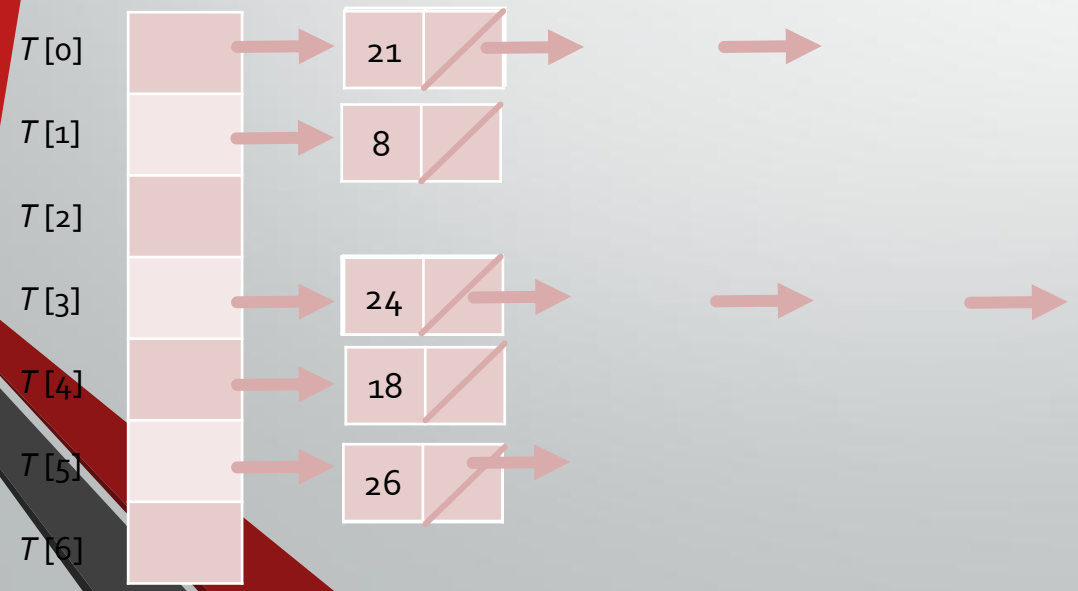
Kulcsütközés feloldása láncolással (collision resolution by chaining)
Láncolt/vödrös hashelés (animációval)

- Feltesszük, hogy a hasító tábla rései egyszerű láncolt listákat (S1L) azonosítanak
- Hashtábla *T*[*i*] eleme: *i.* láncolt lista első elemére mutató pointer
  - Elemei: *h(k)=i*
- Beszúrás: megfelelő lista elejére

- *T*:E1*[*m*]
- listaelemek:
  - *key*
  - *next*
  - járulékos mezők (satellite data)

E1
$+key : U$ $\ldots$ // satellite data may come here $+next :$ E1*
$+\mathbf{E1}()\ \{\ next := \oslash\ \}$

- Példa:
  - *m* =7
  - *h:* N ⟶ 0..6, h(k) = k mod 7
  - A beszúrt elemek : 3, 21, 73, 8, 24, 35, 26, 14, 5, 10, 18

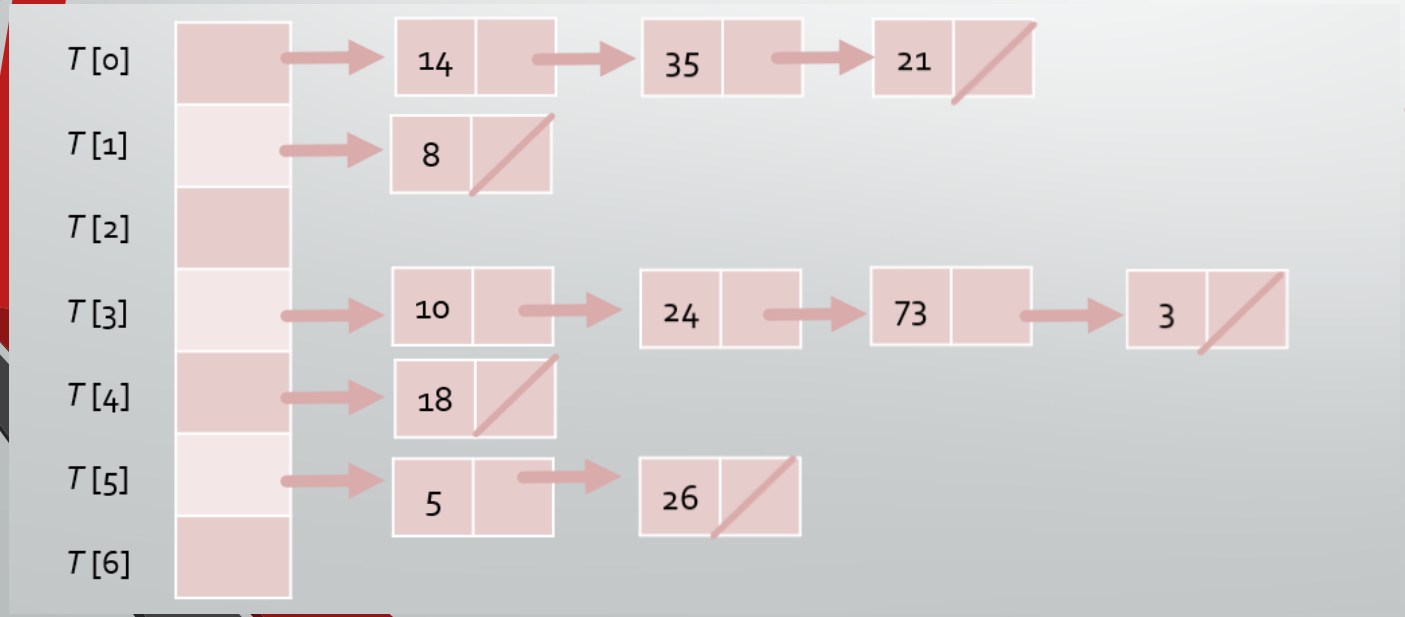


Kulcsütközés feloldása láncolással (collision resolution by chaining)
Láncolt/vödrös hashelés nem animált változata

- Feltesszük, hogy a hasító tábla rései egyszerű láncolt listákat (S1L) azonosítanak
- Hashtábla *T*[*i*] eleme: *i.* láncolt lista első elemére mutató pointer
  - Elemei: *h(k)=i*
- Beszúrás: megfelelő lista elejére

- *T*:E1*[*m*]
- listaelemek:
  - *key*
  - *next*
  - járulékos mezők (satellite data)

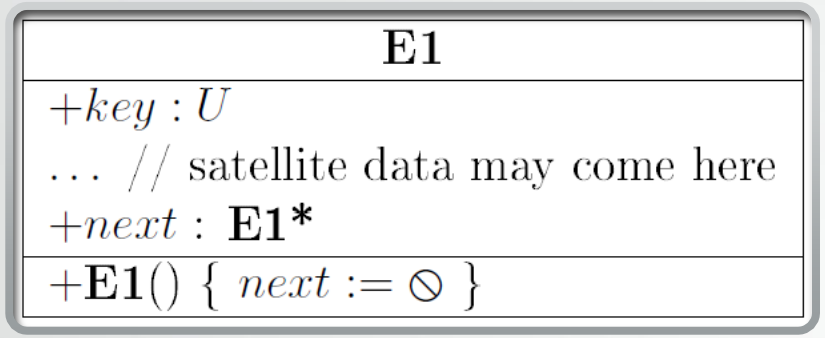
E1
$+key : U$ $\ldots$ // satellite data may come here $+next$: **E1***
+**E1**() { $next := \oslash$ }

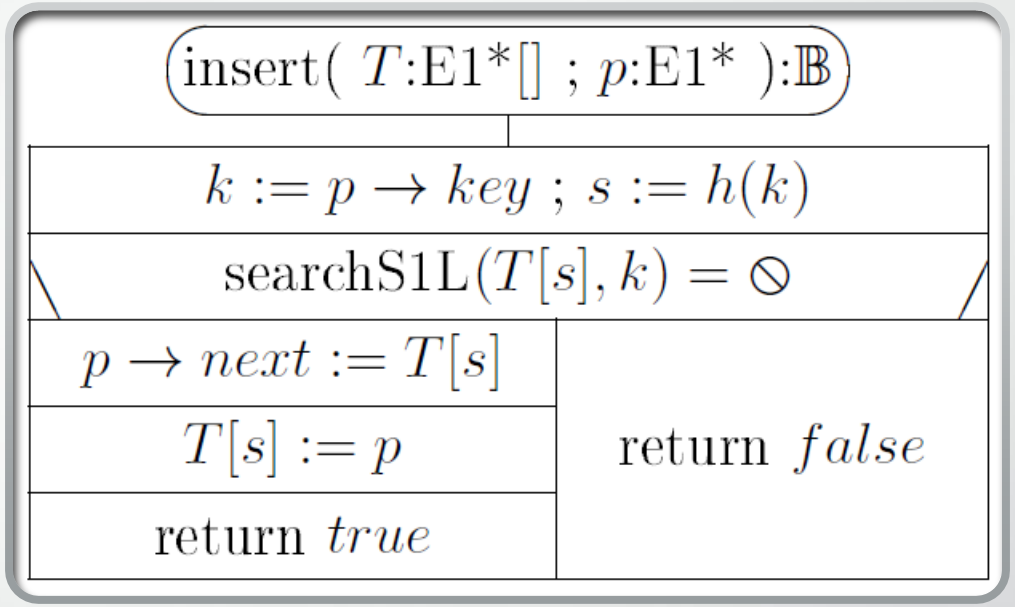


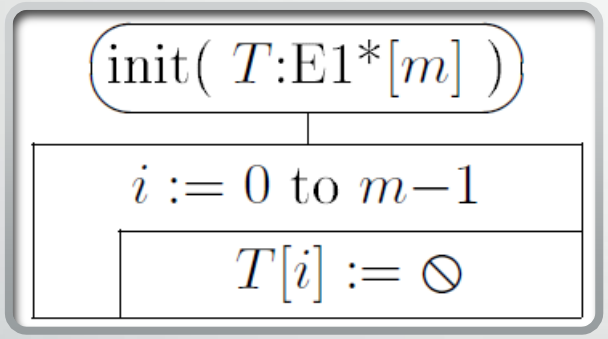
- Példa:
  - *m* =7
  - *h:* N ⟶ 0..6, h(k) = k mod 7
  - A beszúrt elemek : 3, 21, 73, 8, 24, 35, 26, 14, 5, 10, 18

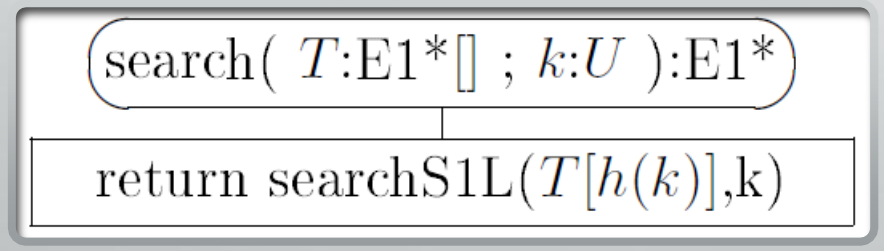
Kulcsütközés feloldása láncolással: műveletek

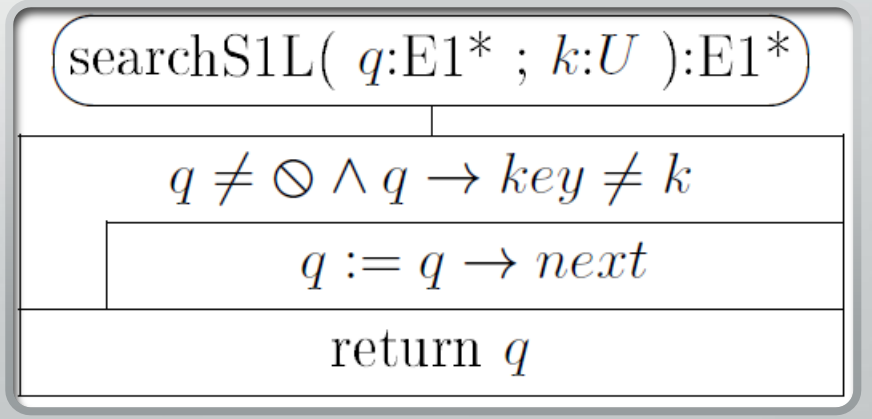
(search and update operations)





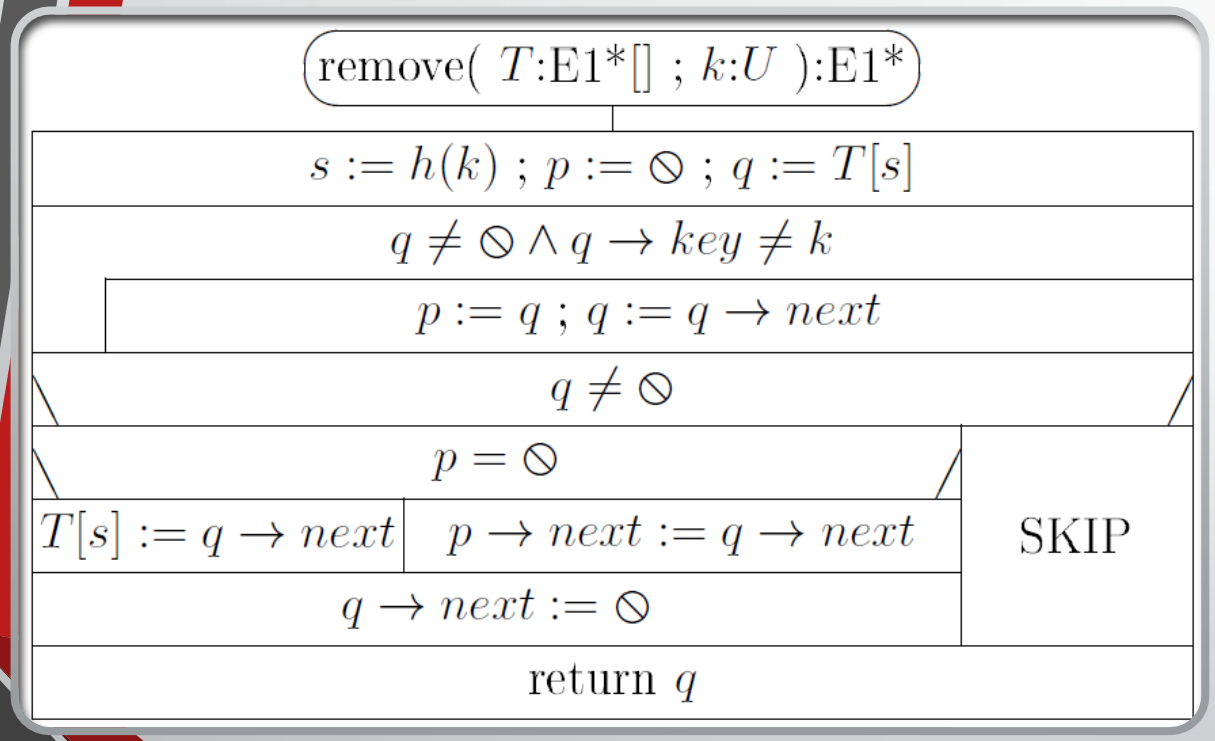






Kulcsütközés feloldása láncolással: műveletek

(search and update operations)



- Műveletigény:
  - $T_{init}(m) \in \Theta(m)$
  - $T_{műveletek}(m)$:
    - $mT \in \Theta(1)$
    - $MT(n) \in \Theta(n)$
    - $AT(n,m) \in \Theta(1 + \frac{n}{m})$
- $AT(n,m)$ feltétele:
  - $h : U \rightarrow 0..(m-1)$ fv egyszerű egyenletes hasítás
  - a résekhez tartozó listák átlagos hossza = $\frac{n}{m}$ =α.
  - $\frac{n}{m} \in O(1)$ => $AT(n,m) \in \Theta(1)$ is teljesül

Láncolt hashelés műveletek C# kódja *

```
using System;

public class E1
{
    public int Key;
    public E1 Next;

    public E1(int key)
    {
        Key = key;
        Next = null;
    }
}
```

```
public class HashTable
{
    private E1[] table;
    private int size;

    public HashTable(int size)
    {
        this.size = size;
        table = new E1[size];
    }

    private int Hash(int key)
    {
        return key % size;
    }

    public void Init()
    {
        for (int i = 0; i < size; i++)
        {
            table[i] = null;
        }
    }
```

```
public bool Insert(E1 p)
{
    int k = p.Key;
    int s = Hash(k);

    if (SearchS1L(table[s], k) != null)
    {
        return false;
    }

    p.Next = table[s];
    table[s] = p;
    return true;
}

public E1 Search(int key)
{
    return SearchS1L(table[Hash(key)], key);
}

private E1 SearchS1L(E1 q, int key)
{
    while (q != null && q.Key != key)
    {
        q = q.Next;
    }
    return q;
}
```

```
    public E1 Remove(int key)
    {
        int s = Hash(key);
        E1 p = null;
        E1 q = table[s];

        while (q != null && q.Key != key)
        {
            p = q;
            q = q.Next;
        }

        if (q != null)
        {
            if (p == null)
            {
                table[s] = q.Next;
            }
            else
            {
                p.Next = q.Next;
            }

            q.Next = null;
        }

        return q;
    }
}
```

A hasítófüggvény megválasztása (good hash functions)

- Osztó módszer (division method)
  - $k \in \mathbb{Z}$
  - $h(k) = k \bmod m$
  - m olyan prím, amely nincs közel a kettő hatványhoz
    - egyenletesen szór [0..(m-1)]
  - Pl. 2000 rekord és $\alpha \approx 3$
    - 701
      - (közel esik a 2000/3-hoz, de távol 512 és 1024

A hasítófüggvény megválasztása

- Kulcsok a [0; 1) intervallumon (egyszerű szorzó módszer)
  - Ha a kulcsok egyenletesen oszlanak el
    - $h(k) = \lfloor k * m \rfloor$
- Szorzó módszer (multiplication method)
  - $k \in \mathbb{R}$
  - $h(k) = \lfloor \{k * A\} * m \rfloor$ ($0 < A < 1$)
  - Nem minden konstanssal szór egyformán jól
  - Knuth: $A = \frac{\sqrt{5}-1}{2} \approx 0{,}618$
  - Osztó módszerrel szembeni előny
    - nem érzékeny a hasító tábla méretére

A hasítófüggvény megválasztása

- Előjel nélküli egész kulcsok esetén a szorzó módszer helyett
  - Táblaméret: kettő hatvány
  - Tfh kulcsok: *w* biten ábrázolt természetes számok
    - $0 \le k \le 2^w - 1$
  - Ábrázoljuk szintén *w* biten az $s = \lfloor A * 2^w \rfloor$ konstanst
  - L! $m = 2^p$, $q = 2^w - 1$ és $r = w - p$
  - dupla szavas, előjel nélküli egész aritmetikát használunk
    - &: a bitenkénti és művelet
    - >>: bitléptetés jobbra

- $h(k) = ((s * k) \,\&\, q) >> r$

A hasítófüggvény megválasztása

- Mindhárom módszer feltételezi, hogy a kulcsok számok

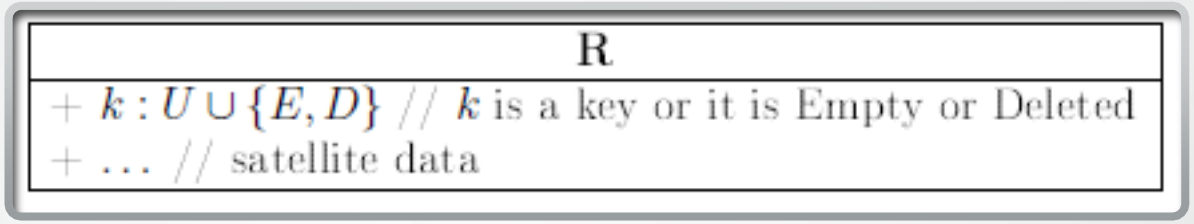
Pl. sztringek esetén

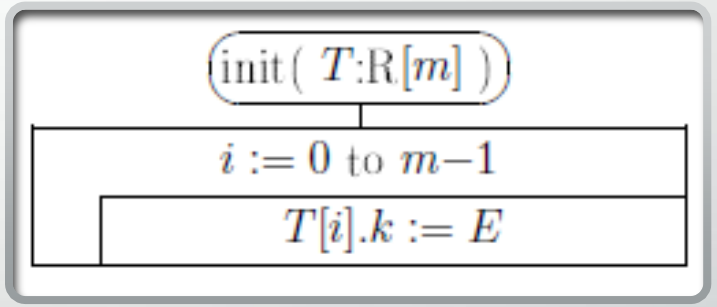
a karakterek ~ megfelelő számrendszerbeli előjel nélküli egész számok számjegyei

a sztringek ~ nagy természetes számok

Nyílt címzéses (open addressing) hashelés

- Az adatrekordok közvetlenül a résekben vannak
  - hasító tábla: $T : R[m]$
  - rekordok: R típus
  - Egyedi kulcsok
  - Rések (slots)
    - Szabad (free) (üres (Empty) / törölt (Deleted))
    - Foglalt (occupied)





- Hashfüggvény
  - próbafüggvény (probing function): $h : U$ x 0..(m-1) ->0.. m -1
  - potenciális próbasorozat (potential probing sequence: <$h(k,0)$, $h(k,1)$,..., $h(k,m-1)$>
  - Első elem: $h_1(k)$
  - m darab hasító függvény: $h(\cdot, i) : U \rightarrow 0..(m-1)$ ($i \in 0..(m-1)$)

Nyílt címzés: beszúrás és keresés, ha nincs törlés

- *k* kulcsú *r* adat beszúrásának/ keresésének lépései:
  - Próbák(foglalt és a kulcsa nem k,): $h(k, 0)$, $h(k, 1)$….
  - Leállás:
    - üres rést találunk
    - *k* kulcsú foglalt rést találunk
    - kimerítjük az összes lehetséges próbát
      - ($\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$ potenciális próbasorozatot)

Nyílt címzés: beszúrás és keresés, ha nincs törlés

Keresés:

- Sikeres: megtaláltuk a keresett kulcsú foglalt rést
- Sikertelen: különben

Beszúrás:

- Sikeres: üres rés
  - ebbe tesszük az adatot
- Sikertelen: különben

Próbasorozatok (probe sequence)

- Potenciális próbasorozat: $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$
  - műveletek elvégzésekor ténylegesen csak egy prefixét állítjuk elő
  - !! $\langle 0, 1, \ldots, (m-1) \rangle$ egy permutációja (permutation) legyen
    - az egész hasító táblát lefedje
    - ne hivatkozzon kétszer vagy többször ugyanarra a résre
- aktuális próbasorozat (actual probing sequence):
  - a potenciális próbasorozatnak a ténylegesen előállított prefix sorozata
- a beszúrás/keresés a $h(k, i-1)$ próbánál áll meg ⇔
  a beszúrás/keresés (azaz az aktuális próbasorozat) hossza i

Egyenletes hasítás, várható lépésszám

- Egyenletes hasítás (uniform hashing)
  - Egy tetszőleges potenciális próbasorozat a $\langle 0, 1, \ldots, (m-1) \rangle$ sorozatnak mind az $m!$ permutációját azonos valószínűséggel állítja elő
- Várható lépésszám
  - (egyenletes hasítás, nincsenek törölt rések, kitöltöttségi arány: $0 < \alpha < 1$)
  - egy sikertelen keresés illetve egy sikeres beszúrás várható hossza legfeljebb $\frac{1}{1-\alpha}$
  - egy sikeres keresés illetve sikertelen beszúrás várható hossza legfeljebb $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
    - 50%-os kitöltöttség mellett egy sikertelen keresés (illetve egy sikeres beszúrás) várható hossza legfeljebb 2, míg egy sikeres keresésé (illetve egy sikertelen beszúrásé) kisebb, mint 1,387
    - 90%-os kitöltöttség mellett pedig egy sikertelen keresés (illetve egy sikeres beszúrás) várható hossza legfeljebb 10, míg egy míg egy sikeres keresésé (illetve egy sikertelen beszúrásé) kisebb, mint 2,559

Nyílt címzésű hasítótábla műveletei, ha van törlés is

- Törlés:
  - Sikeres keresés
  - A megtalált *i* rés kulcsát töröltre (*D*) állítjuk
    - Miért nem üres-re (E) állítjuk?
      - a *k* kulcsú adatot kulcsütközés miatt a $T[h(k, 1)]$ helyre tettük
      - majd töröltük a $h(k, 0)$ helyen levő adatot (azaz a $T[h(k, 0)]$ rést üresre állítottuk)
      - ➢ egy ezt követő keresés nem találná meg a *k* kulcsú adatot

Nyílt címzésű hasítótábla műveletei, ha van törlés is

- Keresés:
  - Átlépjük a törölt réseket (deleted slots) is
  - Leállás
    - sikeres keresés
      - Keresett kulcsú elem megtalálása
    - Sikertelen keresés
      - üres rés (empty slot)
      - a potenciális próbasorozat kimerítése

Nyílt címzésű hasítótábla műveletei, ha van törlés is

- Beszúrás
  - Teljes keresés a beszúrandó adat kulcsára
    - Közben az első törölt rés megjegyzése
  - Sikertelen beszúrás
    - Sikeres keresés (duplikált kulcsot nem engedünk meg)
    - A potenciális próbasorozat kimerítése (a hasítótábla tele van)

Nyílt címzésű hasítótábla műveletei, ha van törlés is

- Sikeres beszúrás: sikertelen keresés
  - Ha találtunk közben törölt rést
    - a beszúrandó adatot az elsőként talált törölt résbe tesszük
    - jövőbeli keresések hossza a lehető legkisebb legyen
  - Ha nem talál törölt rést-
    - üres résen áll meg
    - a beszúrandó adatot ebbe az üres résbe tesszük

Nyílt címzésű hasítótábla műveletei, ha van törlés is

- Hasító táblák karbantartása
  - Ha elég sokáig használunk egy nyílt címzésű hasító táblát
    - Elszaporodhatnak a törölt rések, és elfogyhatnak az üres rések
    - A tábla esetleg közel sincs tele
  - A sikertelen keresések az egész táblát végig fogják nézni
  - Érdemes a táblát időnkénti frissíteni
    - Törölt rések megszüntetése
      - Kimásolja az adatokat egy temporális területre
      - Üresre inicializálja a táblát
      - A kimentett adatokat egyesével újra beszúrja

Próbasorozat előállítása:

- $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$ próbasorozat (részleges) előállítása:
  - Egy **elsődleges** $h_1 : U \rightarrow 0..(m-1)$ **hasítófüggvény**
    - az értéke egyben az első próba indexét, $h(k, 0)$-t adja
  - Szükség esetén : ebből kiindulva lépkedünk a réseken továb:b
  - Elvárás: h_1 **egyszerű egyenletes hasítás (uniform hashing)** legyen
  - Feltesszük: a kulcsok természetes számok (így lehet pl. $D = -1$ és $E = -2$)
- Három stratégia:
  - Lineáris próba
  - Négyzetes próba
  - Kettős hasítás

Lineáris próba (linear probing)

- $h(k, i) = (h_1(k) + i) \bmod m$ $\quad (i \in 0..m\text{-}1)$
- Előnye:
  - Könnyű implementálni
- Hátrányai:
  - Ha két kulcsra $h(k_1, 0) = h(k_2, 0)$
    - másodlagos csomósodás (secondary clustering):
      - az egész próbasorozatuk megegyezik

➢ Összesen csak m db különböző próbasorozat van

- az egyenletes hasításhoz szükséges $m!$ db próbasorozathoz képest

Lineáris próba (linear probing) (animációval)

- Hátrányai:
  - Elsődleges csomósodás (primary clustering):
    - a különböző próba sorozatok összekapcsolódásával foglalt rések hosszú, összefüggő sorozatai alakulhatnak ki
    - megnöveli a várható keresési időt
    - pl. két szabad rés között (ciklikusan értve) *i* db foglalt rés
      - legalább $(i + 2)/m$ valószínűséggel a következő sikeres beszúráskor ez a csomó még hosszabb lesz
      - előfordulhat, hogy közben összekapcsolódik egy másik csomóval.
- Használata:
  - Ha a kulcsütközés valószínűsége elenyészően kicsi

A kitöltöttségi arányszám: $\alpha = 6/11$

Sikeres keresés várható lépésszáma (egyenletes hasítás esetén): $\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right) \approx 1{,}73$

Lineáris próba

0	1	2	3	4	5	6	7	8	9	10
21		24	2	15	16					32

- $h(k, i) = (h_1(k) + i) \bmod m$ $\quad (i \in 0..m\text{-}1)$
- Példa: $m = 11$, $h_1 : U \rightarrow 0..(m-1)$, $h_1(k) = k \bmod m$

Művelet	k	$h_1(k)$	Próbasorozat
Beszúr	24	2	2_E ✓
Beszúr	16	5	5_E ✓
Beszúr	57	2	$2, 3_E$ ✓
Beszúr	32	10	10_E ✓
Beszúr	15	4	4_E ✓
Töröl	57	2	$2, 3_E$ ✓

Művelet	k	$h_1(k)$	Próbasorozat
Beszúr	21	10	$10, 0_E$ ✓
Keres	2	2	$2, 3_D, 4, 5, 6_E$ X
Beszúr	2	2	$2, 3_D, 4, 5, 6_E$ ✓
Beszúr	2	2	$2, 3_2$ X
Keres	21	10	$10, 0_{21}$ ✓

- Táblaméretet célszerű prímszámnak választani

A kitöltöttségi arányszám: α =6/11

Sikeres keresés várható lépésszáma (egyenletes hasítás esetén): $\frac{1}{\alpha}\ln(\frac{1}{1-\alpha}) \approx 1{,}73$

Lineáris próba nem animált változata

- $h(k, i) = (h_1(k) + i) \bmod m$ $\quad (i \in 0..m\text{-}1)$
- Példa: $m = 11$, $h_1 : U \rightarrow 0..(m-1)$, $h_1(k) = k \bmod m$

Művelet	k	$h_1(k)$	Próbasorozat
Beszúr	24	2	2_E ✓
Beszúr	16	5	5_E ✓
Beszúr	57	2	$2,3_E$ ✓
Beszúr	32	10	10_E ✓
Beszúr	15	4	4_E ✓
Töröl	57	2	$2, 3_E$ ✓

0	1	2	3	4	5	6	7	8	9	10
		24								
		24			16					
		24	57		16					
		24	57		16					32
		24	57	15	16					32
		24	D	15	16					32

- Táblaméretet célszerű prímszámnak választani

A kitöltöttségi arányszám: $\alpha = 6/11$

Sikeres keresés várható lépésszáma (egyenletes hasítás esetén): $\frac{1}{\alpha} \ln(\frac{1}{1-\alpha}) \approx 1{,}73$

Lineáris próba nem animált változata

- $h(k, i) = (h_1(k) + i) \bmod m$ $\quad (i \in 0..m\text{-}1)$
- Példa: $m = 11$, $h_1 : U \rightarrow 0..(m-1)$, $h_1(k) = k \bmod m$

Művelet	k	$h_1(k)$	Próbasorozat
Beszúr	21	10	$10, 0_E$ ✓
Keres	2	2	$2, 3_D, 4, 5, 6_E$ X
Beszúr	2	2	$2, 3_D, 4, 5, 6_E$ ✓
Beszúr	2	2	$2, 3_2$ X
Keres	21	10	$10, 0_{21}$ ✓

0	1	2	3	4	5	6	7	8	9	10
21		24	D	15	16					32

0	1	2	3	4	5	6	7	8	9	10
21		24	D	15	16					32

0	1	2	3	4	5	6	7	8	9	10
21		24	2	15	16					32

Négyzetes próba (quadratic probing)

- $h(k, i) = (h(k, 0) + c_1 \cdot i + c_2 \cdot i^2) \mod m$ $(i \in 0..m\text{-}1)$
  - $(h_1 : U \rightarrow 0..(m-1)$ hasító fv.), $c_1, c_2 \in \mathbb{R}; c_2 \neq 0$
- A különböző próbasorozatok nem kapcsolódnak össze
- Másodlagos csomósodás (secondary clustering):
  - Ha két kulcsra $h(k_1, 0) = h(k_2, 0)$ -> az egész próba sorozatuk megegyezik

➢ Itt is csak m db különböző próbasorozat

- (az egyenletes hasításhoz szükséges $m!$ db próbasorozathoz képest)

Négyzetes próba (quadratic probing)

- c_1, c_2 konstansok megválasztása:
  - m kettőhatvány -> $c_1 = c_2 = 1/2$
    - $h(k,i) = \left(h_1(k) + \frac{i+i^2}{2}\right) mod\ m$ ($i \in 0..m-1$)
    - $(h(k,i+1) - h(k,i))\ mod\ m = \left(\frac{(i+1)+(i+1)^2}{2} - \frac{i+i^2}{2}\right) mod\ m = (i+1)\ mod\ m$
    - $h(k,i+1) = (h(k,i) + i + 1)\ mod\ m$

Négyzetes próba (animációval)

0	1	2	3	4	5	6	7
79		4		12	21		28

- Példa: $m = 8$, Példa: $m = 11$, $h_1 : U \rightarrow 0..(m-1)$, $h_1(k) = k \bmod m$

Művelet	K	$h_1(k)$	Próbasorozat
Beszúr	21	5	5_E ✓
Beszúr	12	4	4_E ✓
Beszúr	39	7	7_E ✓
Beszúr	79	7	$7, 0_E$ ✓
Beszúr	4	4	$4, 5, 7, 2_E$ ✓
Töröl	39	7	7_{39} ✓

Művelet	K	$h_1(k)$	Próbasorozat
Keres	4	4	$4, 5, 7_D, 2_4$ ✓
Keres	15	7	$7_D, 0, 2, 5, 1_E$ X
Beszúr	28	4	$4, 5, 7_D, 2, 6_E$ ✓
Töröl	18	2	$2, 3_E$ X

Négyzetes próba nem animált változata

- Példa: $m = 8$, Példa: $m = 11$, $h_1 : U \rightarrow 0..(m-1)$, $h_1(k) = k \bmod m$

Művelet	K	$h_1(k)$	Próbasorozat
Beszúr	21	5	5_E ✓
Beszúr	12	4	4_E ✓
Beszúr	39	7	7_E ✓
Beszúr	79	7	$7, 0_E$ ✓
Beszúr	4	4	$4, 5, 7, 2_E$ ✓
Töröl	39	7	7_{39} ✓

0	1	2	3	4	5	6	7
					21		
				12	21		
				12	21		39
79				12	21		39
79		4		12	21		39
79		4		12	21		D

Négyzetes próba nem animált változata

- Példa: $m = 8$, Példa: $m = 11$, $h_1 : U \rightarrow 0..(m-1)$, $h_1(k) = k \bmod m$

Művelet	K	$h_1(k)$	Próbasorozat
Keres	4	4	$4, 5, 7_D, 2_4$ ✓
Keres	15	7	$7_D, 0, 2, 5, 1_E$ X
Beszúr	28	4	$4, 5, 7_D, 2, 6_E$ ✓
Töröl	18	2	$2, 3_E$ X

0	1	2	3	4	5	6	7
79		4		12	21		D
79		4		12	21		D
79		4		12	21		28
79		4		12	21		28

Négyzetes próba

A kitöltöttségi arányszám: α =5/8

0	1	2	3	4	5	6	7
79		10		12	21		28

- Példa: $m = 8$, $h : \mathrm{N} \longrightarrow 0..7$, $h(k, n) = (k + \frac{n+n^2}{2}) \bmod 8$

Művelet	*K*	$h_1(k)$	Próbasorozat
Beszúr	21	5	5_E ✓
Beszúr	12	4	4_E ✓
Beszúr	39	7	7_E ✓
Beszúr	79	7	$7, 0_E$ ✓
Beszúr	4	4	$4, 5, 7, 2_E$ ✓
Töröl	39	7	7_{39} ✓

Művelet	K	$h_1(k)$	Próbasorozat
Keres	4	4	$4,5,7_D,2_4$ ✓
Keres	15	7	$7_D, 0, 2, 5, 1_E$ X
Beszúr	28	4	$4, 5, 7_D, 2, 6_E$ ✓
Töröl	18	2	$2, 3_E$ X
Töröl	4	4	$4, 5, 7, 2_E$ ✓
Beszúr	10	2	$2_D, 3, 5, 0, 4, 1_E$ ✓

Kettős hasítás (double hashing) (animációval)

- $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m \quad (i \in 0..m\text{-}1)$
  - ($h_1 : U \rightarrow 0..(m-1)$ és $h_2 : U \rightarrow 0..(m-1)$ hasító fv.-ek)
- A próbasorozat lefedi az egész hasító táblát $\Leftrightarrow$ $h_2(k)$ és m relatív prímek
  - m kettő hatvány és $h_2(k)$ minden lehetséges kulcsra páratlan szám
  - m prímszám
- Példa:
  - ha m prímszám (ami lehetőleg ne essen kettő hatvány közelébe)
  - m' kicsit kisebb (mondjuk $m' = m - 1$ vagy $m' = m - 2$)
  - $h_1(k) = k \bmod m \quad h_2(k) = 1 + (k \bmod m')$

Kettős hasítás (double hashing)

- Minden különböző $(h_1(k), h_2(k))$ pároshoz különböző próbasorozat tartozik
  - $\Theta(m^2)$ különböző próbasorozat lehetséges
    - messze van az ideális $m!$ számú próbasorozattól
    - mégis jól közelíti annak működését
    - Példa
      - m=1609 és m'=1607 (ikerpírmek)
      - 2.585.663 különböző potenciális próbasorozat
        - Lineáris és négyzetes próba estén: 1609

Kettős hasítás

- $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$ $(i \in 0..m\text{-}1)$
- Példa: $m = 11$, $h_1(k) = k \bmod 11$, $h_2(k) = 1 + (k \bmod 10)$

0	1	2	3	4	5	6	7	8	9	10
141	34	24	25			D	29		53	

Próbasorozat: számtani sorozat

Első elem: $h_1 : N \rightarrow 0..m\text{-}1$

Differencia: kulcsonként eltérő: $h_2 : U \rightarrow 0..m\text{-}1$

Művelet	k	$h_1(k)$	$h_2(k)$	Próbasorozat
Beszúr	34	1	5	1_E ✓
Beszúr	53	9	4	9_E ✓
Beszúr	141	9	2	$9, 0_E$ ✓
Beszúr	0	0	1	$0, 1, 2_E$ ✓
Beszúr	55	0	6	$0, 6_E$ ✓
Töröl	55	0	6	$0, 6_{55}$ ✓

Művelet	k	$h_1(k)$	$h_2(k)$	Próbasorozat
Keres	83	6	4	$6_D, 10_E$ X
Beszúr	25	3	6	3_E ✓
Beszúr	29	7	0	7_E ✓
Töröl	22	0	3	$0, 3, 6_D, 9, 1, 4_E$ X
Töröl	0	0	1	$0, 1, 2_0$ ✓
Beszúr	24	2	5	$2_D, 7, 1, 6_D, 0, 5_E$ ✓

Kettős hasítás nem animált változata

- $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$ $(i \in 0..m\text{-}1)$
- Példa: $m = 11$, $h_1(k) = k \bmod 11$, $h_2(k) = 1 + (k \bmod 10)$

Művelet	k	$h_1(k)$	$h_2(k)$	Próbasorozat
Beszúr	34	1	5	1_E ✓
Beszúr	53	9	4	9_E ✓
Beszúr	141	9	2	$9, 0_E$ ✓
Beszúr	0	0	1	$0, 1, 2_E$ ✓
Beszúr	55	0	6	$0, 6_E$ ✓
Töröl	55	0	6	$0, 6_{55}$ ✓

0	1	2	3	4	5	6	7	8	9	10
	34									
	34								53	
141	34								53	
141	34	0							53	
141	34	0				55			53	
141	34	0				D			53	

Kettős hasítás nem animált változata

- $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$ $(i \in 0..m\text{-}1)$
- Példa: $m = 11$, $h_1(k) = k \bmod 11$, $h_2(k) = 1 + (k \bmod 10)$

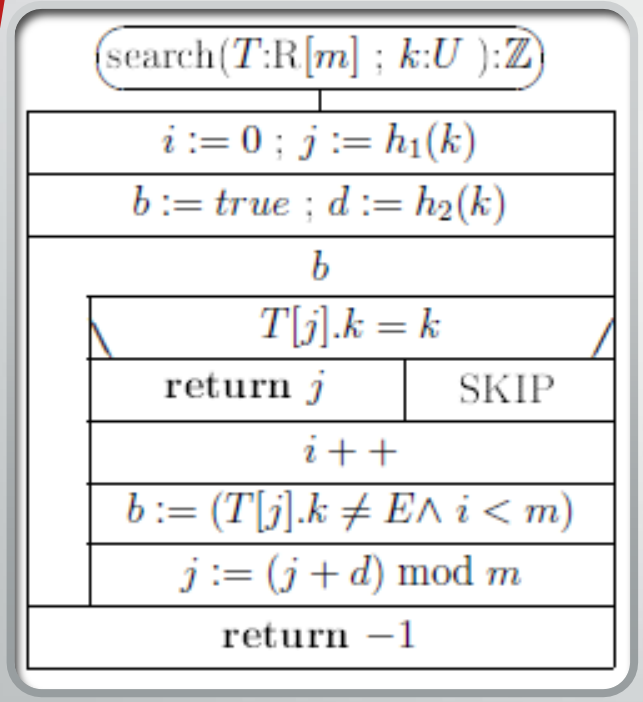
Művelet	k	$h_1(k)$	$h_2(k)$	Próbasorozat
Keres	83	6	4	$6_D, 10_E$ ✗
Beszúr	25	3	6	3_E ✓
Beszúr	29	7	0	7_E ✓
Töröl	22	0	3	$0, 3, 6_D, 9, 1, 4_E$ ✗
Töröl	0	0	1	$0, 1, 2_0$ ✓
Beszúr	24	2	5	$2_D, 7, 1, 6_D, 0, 5_E$ ✓

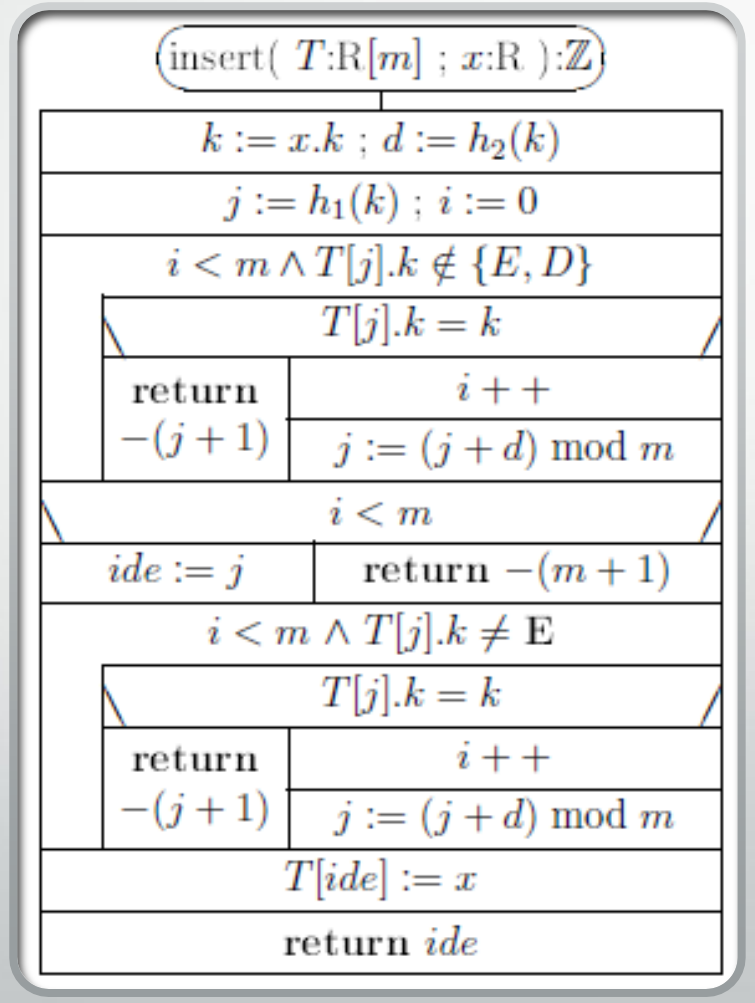
0	1	2	3	4	5	6	7	8	9	10
141	34	0				D			53	
141	34	0	25			D			53	
141	34	0	25			D	29		53	
141	34	0	25			D	29		53	
141	34	D	25			D	29		53	
141	34	24	25			D	29		53	

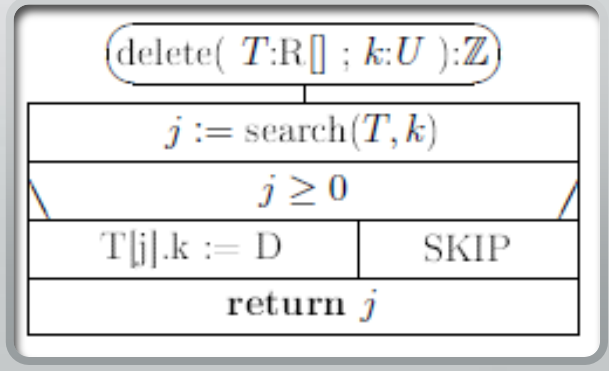
A kettős hasítás programozása - feladat

- Írjuk meg a beszúrás, a keresés és a törlés struktogramjait
- Paraméterek:
  - *x*: beszúrandó adat
  - *k*: a keresett kulcs,
    illetve a törlendő adat kulcsa
  - A hasító tábla: T[o..*m*)
    - nullától indexeljük
    - *m* a mérete
- Eredmény:
  - Sikeres keresés: a keresett kulcsú adat pozíciója
  - Sikertelen keresés: -1
  - Sikeres beszúrás: a beszúrás pozíciója
  - Sikertelen beszúrás:
    - Ha nincs elég hely a táblában:-(*m*+1)
    - Ha a *j* indexű résben megtaláljuk a táblában a beszúrandó adat kulcsát:−(j + 1)
  - Sikeres törlés: a törölt adat pozíciója
  - Sikertelen törlés:−1
- A nyílt címzésű hasító tábla üresre inicializálásának és az adott kulcsú foglalt rés törlésének struktogramja nem függ attól, hogy a beszúrást és a keresést milyen stratégiával hajtjuk végre.

A kettős hasítás programozása - megoldás







Kettős hasítás C# kód*

```
using System;

public enum EntryState { Empty, Deleted, Occupied }

public class R
{
    public int Key;
    public EntryState State;

    public R()
    {
        State = EntryState.Empty;
    }
}

public class OpenAddressHashTable
{
    private R[] table;
    private int m;

    public OpenAddressHashTable(int size)
    {
        m = size;
        table = new R[m];
        Init();
    }
```

```
public void Init()
{
    for (int i = 0; i < m; i++)
    {
        table[i] = new R();
    }
}

private int H1(int key) => key % m;
private int H2(int key) => 1 + (key % (m - 1));
```

```
public bool Remove(int key)
{
    int index = Search(key);
    if (index == -1)
        return false;

    table[index].State = EntryState.Deleted;
    return true;
}
}
```

Kettős hasítás
C# kód*

```
public int Search(int key)
{
    int d = H2(key);
    int j = H1(key);
    int i = 0;

    while (table[j].State != EntryState.Empty && i < m)
    {
        if (table[j].State == EntryState.Occupied && table[j].Key == key)
            return j;

        i++;
        j = (j + d) % m;
    }

    return -1; // not found
}
```

```
public int Insert(int key)
{
    int d = H2(key);
    int j = H1(key);
    int i = 0;
    int? ide = null;

    while (i < m && table[j].State == EntryState.Occupied && table[j].Key != key)
    {
        if (ide == null && table[j].State == EntryState.Deleted)
            ide = j;

        i++;
        j = (j + d) % m;
    }

    if (i < m && table[j].Key == key && table[j].State == EntryState.Occupied)
        return -(i + 1); // key already exists

    if (ide == null)
        ide = j;

    if (i >= m || table[j].State == EntryState.Occupied)
        return -(m + 1); // table full or error

    table[ide.Value].Key = key;
    table[ide.Value].State = EntryState.Occupied;

    return ide.Value;
}
```

Ellenőrző kérdések

1. A $T[0..M-1]$ hasító tábla rései (slot-jai) egyirányú, nemciklikus, fejelem nélküli, rendezetlen láncolt listák pointerei. Adott a $k \bmod M$ hasító függvény. A kulcsütközéseket láncolással oldjuk fel. Feltehető, hogy mindegyik kulcs csak egyszer szerepel T-ben.
   - Írjuk meg az *ins(T[],M,k,a)* : 0..2 értékű függvényt, ami beszúrja a $T[0..M-1]$ hasító táblába a (k,a) kulcs-adat párt! Akkor és csak akkor ad vissza 0 értéket, ha sikeres volt a beszúrás. Ilyenkor az új listaelemet a megfelelő lista elejére szúrjuk be! Ha a táblában már volt k kulcsú elem, a beszúrás meghiúsul, és 2 hibakódot adjunk vissza. Különben, ha nem tudjuk már a szükséges listaelemet allokálni, 1 hibakódot adjunk vissza!
   - Írjuk meg a *search(T[],M,k)* függvényt, ami visszaadja a T-beli, k kulcsú elem címét, vagy a _ pointert, ha ilyen nincs!
   - Írjuk meg a *del(T[],M,k)* logikai függvényt, ami törli a $T[0..M-1]$ hasító táblából a k kulcsú adatot! Akkor és csak akkor ad vissza true értéket, ha sikeres volt a törlés, azaz a táblában volt k kulcsú elem. Ilyenkor a feleslegessé váló listaelemet deallokáljuk!
   - Legyen $M = 9$ Mutassuk be, mi történik, ha sorban beszúrjuk a hasító táblába a következő kulcsokat: 5; 28; 19; 15; 20; 33; 12; 17; 10.

Ellenőrző kérdések

1. A T[0..M-1] hasító táblát nyílt címzéssel ábrázoltuk. Adott a k mod M hasító függvény. Mindegyik kulcs csak egyszer szerepel T-ben.
   - A kulcsütközések feloldására használjuk a következő módszereket:
     - Lineáris próba
     - Négyzetes próba ($c_1 = c_2 = ½$)
     - Kettős hash-elés esetére! A másodlagos hash függvény: $h_2(k) = 1+(k \bmod (M-1))$
   - Írjuk meg az *ins(T[],M,k,a)* : 0..2 értékű függvényt, ami beszúrja a T[0..M-1] hasító táblába a (k,a) kulcs-adat párt! Akkor és csak akkor ad vissza 0 értéket, ha sikeres volt a beszúrás. Ilyenkor az új listaelemet a megfelelő lista elejére szúrjuk be! Ha a táblában már volt k kulcsú elem, a beszúrás meghiúsul, és 2 hibakódot adjunk vissza. Különben, ha nem tudjuk már a szükséges listaelemet allokálni, 1 hibakódot adjunk vissza!
   - Írjuk meg a *search(T[],M,k)* függvényt, ami visszaadja a T-beli, k kulcsú elem címét, vagy a _ pointert, ha ilyen nincs!
   - Írjuk meg a *del(T[],M,k)* logikai függvényt, ami törli a T[0..M-1] hasító táblából a k kulcsú adatot! Akkor és csak akkor ad vissza true értéket, ha sikeres volt a törlés, azaz a táblában volt k kulcsú elem. Ilyenkor a feleslegessé váló listaelemet deallokáljuk!
   - Legyen M = 9 Mutassuk be, mi történik, ha sorban beszúrjuk a hasító táblába a következő kulcsokat 10; 22; 31; 4; 15; 28; 17; 88; 59. Ezután töröljük a 17-et, majd próbáljuk megkeresni a 18-at és az 59-et, végül pedig szúrjuk be a 18-at!

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek I. előadásjegyzete és az Algoritmusok I. gyakorló feladatok (2024) alapján készült.

Algoritmusok és adatszerkezetek I.
11. Előadás

(Összehasonlító) rendezések műveletigényének alsó korlátai

Tartalom

Rendezések alsókorlát-elemzése (Lower bounds for sorting)

1. Tétel: Tetszőleges rendező algoritmusra $mT(n) \in \Omega(n)$.

- **Bizonyítás**
  - ∀ helyes rendező algoritmusnak mind az *n* rendezendő elem értékét ellenőriznie kell
  - Mindegyik alprogram hívás és ciklusiteráció is csak korlátos számú elemet ellenőriz
    - a belé ágyazott alprogram hívások és ciklusiterációk nélkül
  - L! ezeknek a korlátoknak a maximuma *k*
    - $mT(n) * k \geq n$
    - $mT(n) \geq \frac{1}{k} n$
    - $mT(n) \in \Omega(n)$

Összehasonlító rendezések (Comparison sorts)

2. Definíció.

- Tetszőleges rendező algoritmus akkor összehasonlító rendezés (comparison sort), ha az input elemeinek rendezéséhez csak az elemek kulcsainak összehasonlításából nyer információt.
  - adott az $\langle a_1, a_2, \ldots, a_n \rangle$ input kulcssorozat, és ennek két kulcsát akarjuk összehasonlítani
    - az $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \neq a_j$, $a_i \geq aj$, vagy $a_i > a_j$ kulcs-összehasonlítások valamelyikét végezzük el
  - Nem vizsgáljuk meg a kulcsok belső szerkezetét, és más egyéb módon sem szerzünk róluk információt.
- Insertion sort, Heap sort, Merge sort és a Quicksort

Az összehasonlító rendezések alsókorlát-elemzése

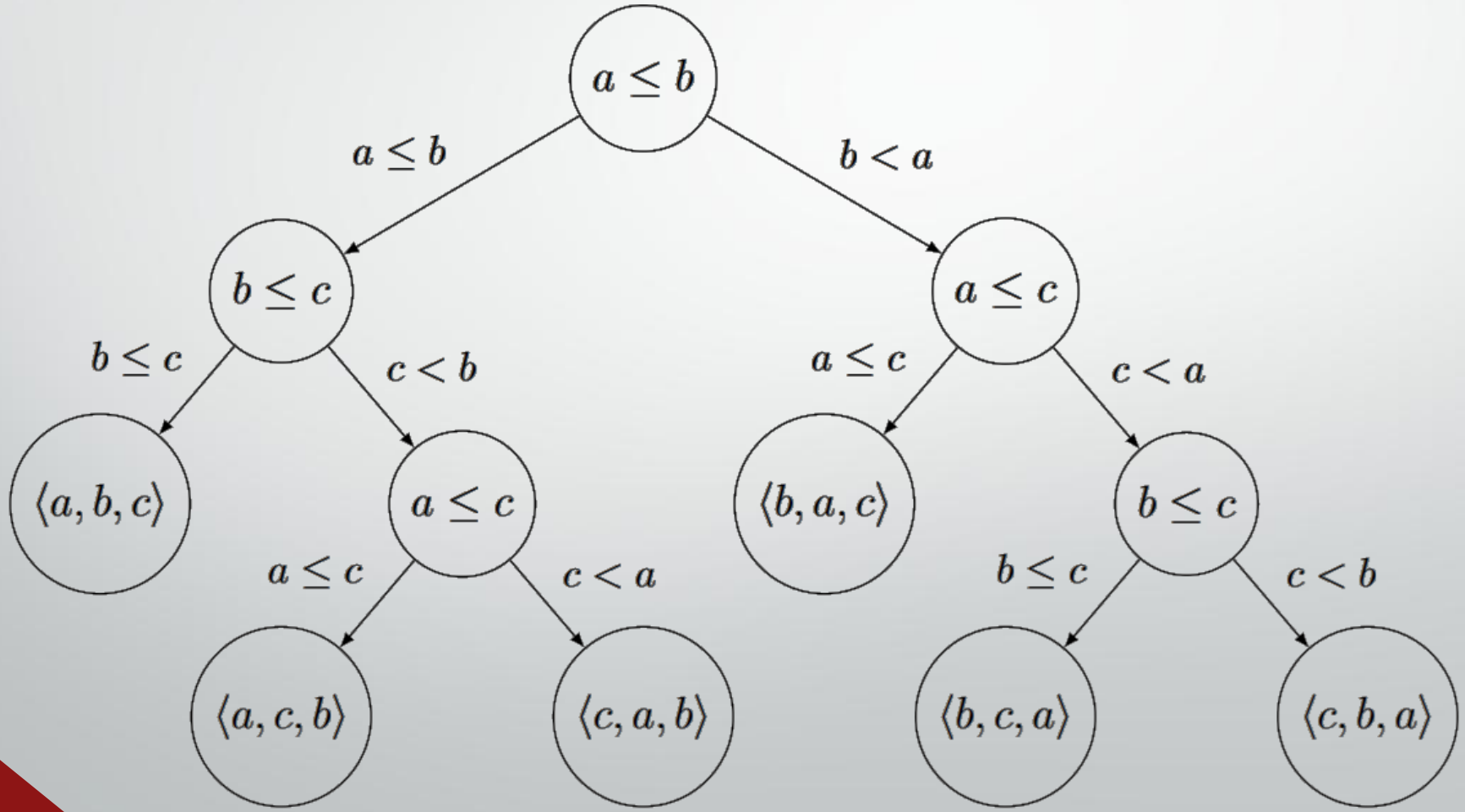
- Feltehető: a rendezendő elemek kulcsai különböznek egymástól
  - A lehetséges inputok halmazát csökkentjük
  - A kisebb halmazon adott alsó korlát ($MC(n)$-re/ $MT(n)$-re) -> a teljes halmazon is alsó becslés
- Alsó becslések:
  - A rendező algoritmus által elvégzett kulcs összehasonlítások maximális számára ($MC(n)$)
  - A maximális műveletigényre ($MT(n)$)
  - $a_i < a_j$, $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ alakú kulcs összehasonlítások maximális számára
    - Ez az összes kulcsösszehasonlítás maximális számára is alsó becslés

Döntési fa modell (decision tree model)

- Az összehasonlító rendezések modellezésének eszköze
- Belső csúcsok: kulcs összehasonlítások
- Levelek: a rendezés eredményeként adódó lehetséges sorozatok
- Az egyes részfákban a felette levő kulcs összehasonlítások eredményeivel összhangban levő levelek vannak
  - feltéve, hogy a rendező algoritmus helyes
- Az unáris belső csúcsokat törölhetjük
  - Ezek nem adnak információt a program további működéséhez.

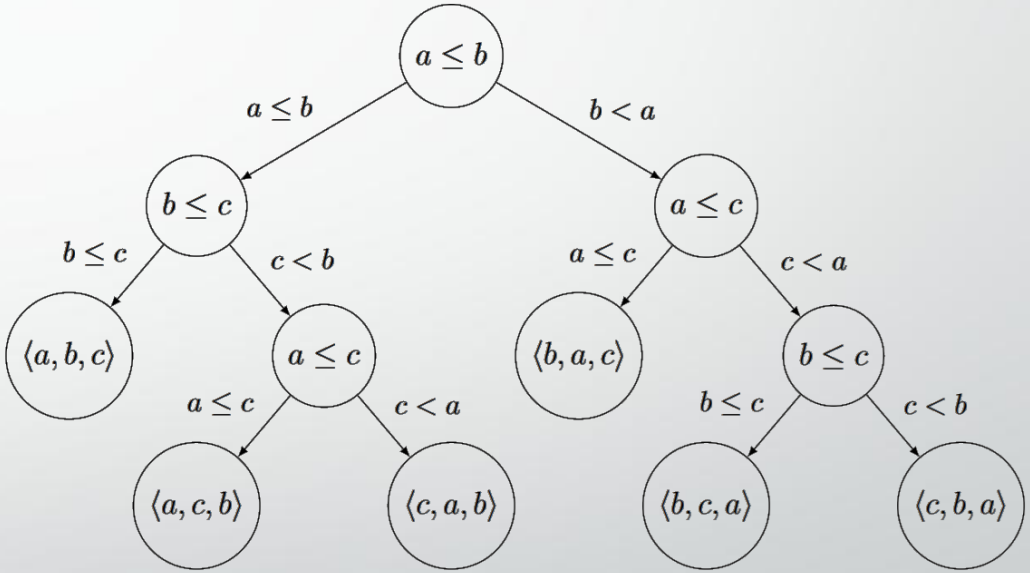
➢ Feltehető: a döntési fa szigorúan bináris

Beszúró rendezés döntési fája az <*a*, *b*, *c*> input sorozatra (3! = 6)



Alsó becslés a kulcsösszehasonlítások maximális számára ($MC(n)$)

- Általánosságban: L! $\langle a_1, a_2, \ldots a_n \rangle$ a rendezendő sorozat
- A döntési fában:
  - ∀ belső csúcs: egy kulcsösszehasonlítás
  - ∀ levél: $\langle a_1, a_2, \ldots a_n \rangle$ permutációja
- Tetszőleges rendezés:
  - egy út megtétele a döntési fában
  - a gyökértől valamelyik levélig
    - belső csúcsok: a közbenső kulcsösszehasonlítások és döntések
    - a levelek: a rendezés lehetséges eredményei



Alsó becslés a kulcs-összehasonlítások maximális számára ($MC(n)$)

- Pl. egy belső csúcs: az $a_i \leq a_j$ öh:
  - a bal részfában a további összehasonlítások: már tudjuk, hogy $a_i \leq a_j$,
  - a jobb részfában a további összehasonlítás: már tudjuk, hogy $a_i > a_j$.
- Egy tetszőleges levélhez érünk: a rendezés meghatározza az $\langle a_1, a_2, \ldots a_n \rangle$ megfelelő sorba rendezését
  - $MC(n)$ = a leghosszabb körmentes út hosszával a döntési fában a gyökértől levélig
  - Ez éppen a döntési fa h magassága
  - $h = MC(n)$
- A rendezés inputja a rendezett sorozat tetszőleges permutációja lehet
  - Bármelyik helyes rendező algoritmus képes kell legyen az input tetszőleges permutációját előállítani
  - Az n elemű input sorozat mind az $n!$ permutációja meg kell jelenjen egy-egy levélen
- A döntési fának legalább $n!$ levele van.

Alsó korlát a legrosszabb esetre (A lower bound for the worst case)

3. Tétel. Bármely összehasonlító rendezés végrehajtásához a legrosszabb esetben $MC(n) \in \Omega(n \log n)$ kulcsösszehasonlítás szükséges.

- **Bizonyítás.** A fentiek miatt elég: alsó becslést adni a döntési fa $h = MC(n)$ magasságára
  - L! a levelei száma: l
  - $n! \leq l$
  - $l \leq 2^h$ (egy h magasságú bináris fának legfeljebb 2^h levele van)

$$n! \leq l \leq 2^h$$

- $n! \leq 2^h$

$$MC(n) = h \geq \log n! = \sum_{i=1}^{n} \log i \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \log i \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \log \left\lceil \frac{n}{2} \right\rceil \geq \left\lceil \frac{n}{2} \right\rceil * \log \left\lceil \frac{n}{2} \right\rceil \geq$$

$$\geq \frac{n}{2} * \log \frac{n}{2} = \frac{n}{2} * (\log n - \log 2) = \frac{n}{2} * (\log n - 1) = \frac{n}{2} \log n - \frac{n}{2} \in \Omega(n \log n)$$

Alsó korlát a legrosszabb esetre

- $\left(\sum_{i=\lceil \frac{n}{2} \rceil}^{n} \log \left\lceil \frac{n}{2} \right\rceil \geq \left\lceil \frac{n}{2} \right\rceil * \log \left\lceil \frac{n}{2} \right\rceil\right)$ magyarázata:
  - Az összegnek legalább $\left\lceil \frac{n}{2} \right\rceil$ tagja van.
  - A tagok száma: $n - \left(\left\lceil \frac{n}{2} \right\rceil - 1\right) = \left(n - \left\lceil \frac{n}{2} \right\rceil\right) + 1 = \left\lfloor \frac{n}{2} \right\rfloor + 1$
    - Ha *n* páratlan: $\left\lceil \frac{n}{2} \right\rceil$
    - Ha *n* páros: $\left\lceil \frac{n}{2} \right\rceil + 1$

Alsó korlát a legrosszabb esetre

4. Tétel. Tetszőleges összehasonlító rendezésre $MT(n) \in \Omega(n \log n)$

- **Bizonyítás.**
  - Mindegyik alprogram hívás és ciklusiteráció is csak korlátos számú elemet ellenőriz
    - a belé ágyazott alprogram hívások és ciklusiteránciók nélkül
  - L! ezeknek a korlátoknak a maximuma *k*
    - $MT(n) * k \geq MC(n)$
    - $MT(n) \geq \frac{1}{k} MC(n)$
    - $MT(n) \in \Omega(MC(n))$

❖ 3. tétel
❖ $\bullet \in \Omega(\bullet)$ reláció tranzitív

- $MT(n) \in \Omega(n \log n)$

- heap sort és a merge sort: aszimptotikusan optimálisak
  - $MT(n) \in O(n \log n)$ összeillik a $MT(n) \in \Omega(n \log n)$ alsó korláttal. (Ld. 4. tétel)
  - $MT(n) \in \Theta(n \log n)$

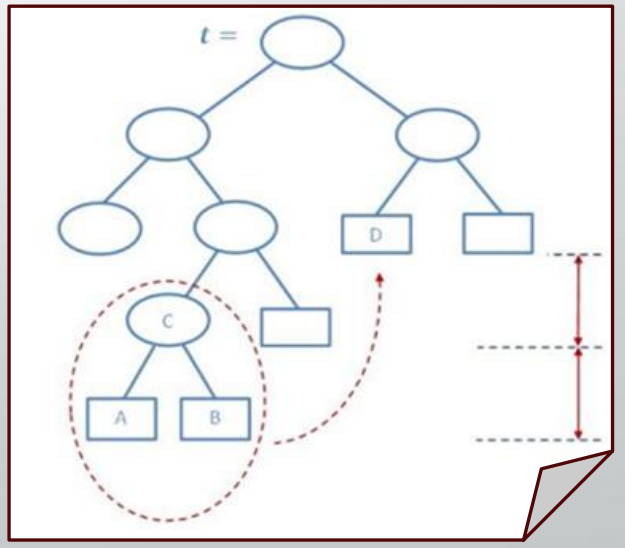
Alsó korlát az összehasonlítások számára átlagos esetben*

- A rendező eljárás (adott *n* méret melletti) átlagos összehasonlítási száma = a döntési fa levélmagasságainak az átlaga
- Jelölések:
  - *lhsum(t) :* A levélmagasság összeg
  - *Tökéletes döntési fa:* minden belső pontnak pontosan két gyermeke van: igen/nem
    - ~(szigorúan bináris fa)
    - Egy *t* tökéletes döntési fához nem feltétlenül tartozik olyan algoritmus, amelynek *t* a döntési fája lenne, de becsléshez felhasználható

5. **Lemma.** Az azonos számú levelet tartalmazó tökéletes döntési fák közül levélmagasság összeg azokra a legkisebb, amelyek egyben majdnem teljes bináris fák.

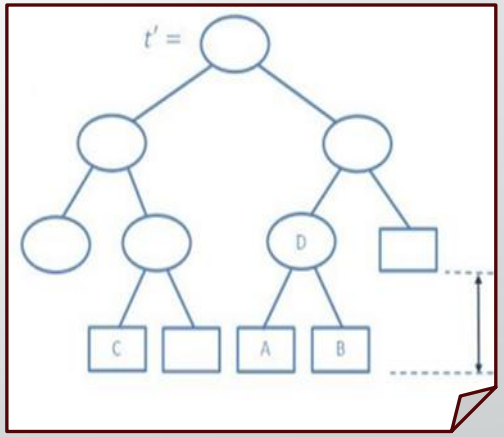
Alsó korlát az összehasonlítások számára átlagos esetben*

- **Bizonyítás.**
  - L! *t* : tökéletes döntési fa, de nem majdnem teljes bináris fa
    - => levelei között legalább két szintkülönbség található
    - Pl. A *t* fában pl. az *A* és *B* levelek és a *D* levél közötti szintkülönbség 2
    - $h(A)-h(D) = h(B)-h(D) \geq 2$
  - L! *t' t-nek egy átalakítása*
    - A és B leveleket áthelyezzük legalább két szinttel magasabbra
    - A levélmagasságösszeg legalább 1-gyel csökken
    - $lhsum(t) - lhsum(t') = 2(h(C) + 1)) + h(D) - (h(C) + 2(h(D) + 1))$
      $= 2\,h(C) + 2 + h(D) - h(C) - 2(h(D)) - 2 = h(C) - h(D) \geq 1$



Alsó korlát az összehasonlítások számára átlagos esetben*

- **Bizonyítás folytatása**
  - Az alsó szintű testvér levélpárok – szülővel együtt történő – véges sokszori áthelyezése
    - ➢ majdnem teljes bináris fához jutunk
    - = tökéletes döntési fa
    - Magasságösszege < bármely (ugyanannyi levélcsúcsot számláló) nem majdnem teljes tökéletes döntési fa



- Az *lsum*(*t*) érték a majdnem teljes bináris fákra egyértelmű
  - Pl. 6 levelű fáknál:
    - 4 az alsó színtű levelek száma
    - 2 az eggyel magasabb szinten levők száma

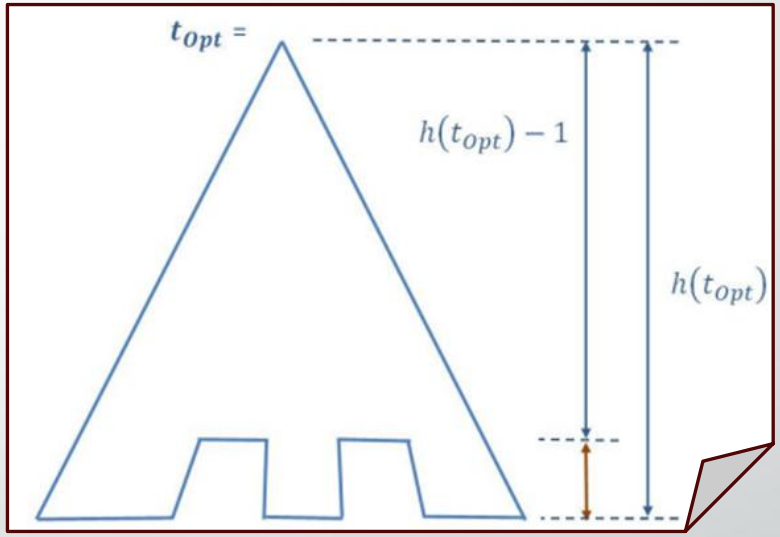
Alsó korlát az összehasonlítások számára átlagos esetben*

6. Tétel. Bármely összehasonlító rendezés végrehajtásához átlagos esetben $AC(n) \in \Omega(n \log n)$ kulcsösszehasonlítás szükséges.

- **Bizonyítás.**
  - L! $t_R(n)$:
    - Az R összehasonlításos rendező algoritmus adott n input mérethez tartozó döntési fája
    - Leveleinek száma: $n!$
  - L! t_{Opt}:
    - $n!$ levelet tartalmazó majdnem teljes tökéletes döntési fa

Alsó korlát az összehasonlítások számára átlagos esetben*

- **Bizonyítás folytatása**
  1. 5. lemma szerint: $lhsum(t_R(n)) \geq lhsum(t_{Opt})$
  2. Továbbá: $lhsum(t_{Opt}) \geq n!(lhsum(t_{Opt}) - 1)$ (lásd ábra)



- Átlagos esetre térve:
  - $AC(n) = \frac{lhsum(t(n))}{n!} \geq \frac{lhsum(t_{opt})}{n!} \geq \frac{n!\,(lhsum(t_{opt})-1)}{n!}$

(1) (2)

$= h(t_{Opt}) - 1 \in \Omega(n \log n)$

Ellenőrző kérdések

1. Mit tud mondani a rendezések minimális műveletigényének alsó korlátjáról? Miért?
2. Egy tetszőleges rendezést mikor nevezünk összehasonlító rendezésnek?
3. Az összehasonlító rendezéseket mivel modellezzük? Mit tud erről mondani?
4. Mondja ki az összehasonlító rendezések maximális műveletigényének alsó korlátjára vonatkozó alaptételt!
5. Bizonyítsa be a kulcsösszehasonlítások számának maximumára vonatkozó lemmát!
6. Osztályozza a tanult rendezéseket műveletigény és tárigény szempontjából!

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek I. előadásjegyzete és Fekete István: Algoritmusok és adatszerkezetek Előadásjegyzete alapján készült.