Kupac, prioritásos sor HeapSort, Leszámláló rendező

9. előadás

Prioritásos sor

Egyszerű sor $\rightarrow$ prioritásos sor

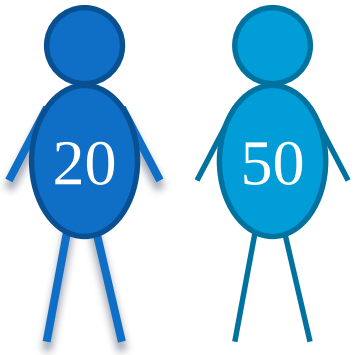
- Egyszerű sor:
  - FIFO szemantika
  - Elem hozzáadása, törlése konstans ($\mathcal{O}(1)$) igényű
- Mit tegyünk, ha a sorban lévő elemeknek valamifajta rendezése is van?
  - Ezt általában prioritásnak nevezzük.
  - Úgy kell rendezzük az elemeket, hogy a legnagyobb (legkisebb) prioritású elemet töröljük először.

Elsőbbségi (prioritásos) sor

- Példa
  - sürgősségi osztály
    - különböző súlyosságú esetek

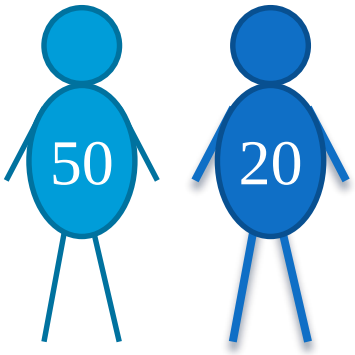
Elsőbbségi (prioritásos) sor

- Példa
  - sürgősségi osztály
    - különböző súlyosságú esetek



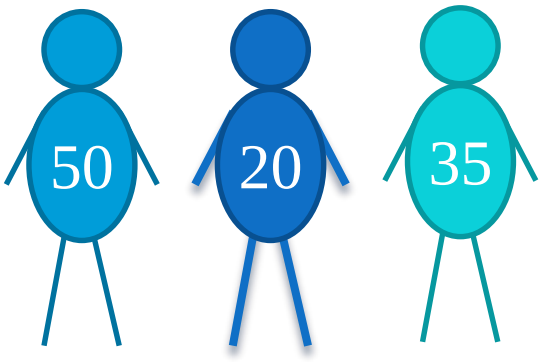
Elsőbbségi (prioritásos) sor

- Példa
  - sürgősségi osztály
    - különböző súlyosságú esetek



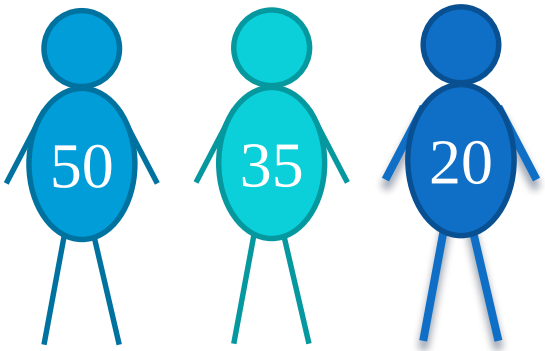
Elsőbbségi (prioritásos) sor

- Példa
  - sürgősségi osztály
    - különböző súlyosságú esetek



Elsőbbségi (prioritásos) sor

- Példa
  - sürgősségi osztály
    - különböző súlyosságú esetek



Elsőbbségi (prioritásos) sor

- Mire lehet használni?
  - Tennivalók, melyiket kell először elvégezni
  - Operációs rendszer, mely prioritásos feladatokat (jobokat) dolgoz fel
    - Itt különböző további algoritmusok, amelyek a prioritást meghatározzák egy-egy folyamat (job) számára
  - Telekommunikációban a csomagok továbbításánál is

Elsőbbségi (prioritásos) sor

- Az elsőbbségi sor ADT axiomatikus leírása
- E alaptípus feletti P elsőbbségi sor típus jellemzése:
  - Egyszerűsítés: csak prioritásokat teszünk bele (N)
  - Műveletek „full" nem szerepel
    - empty $\rightarrow P$ az üres prior. sor konstruktor – létrehozás
    - isempty $P \rightarrow L$ üres a prioritásos sor?
    - insert: $P \times N \rightarrow P$ elem betétele a prioritásos sorba
    - delmax: $P \rightarrow P \times N$ maximális elem kivétele a pr. sorból
    - max: $P \rightarrow N$ maximális elem lekérdezése
  - Megszorítások:
    - delMax és max értelmezési tartománya $P \setminus \{empty\}$

Elsőbbségi (prioritásos) sor

- Axiómák:
  1. $\text{isempty}(\text{empty})$ vagy $p = \text{empty} \rightarrow \text{isempty}(p)$
  2. $\text{isempty}(p) \rightarrow p = \text{empty}$
  3. $\neg\text{isempty}(\text{insert}(p, n))$
  4. $\text{insert}(\text{delmax}(p)) = p$
  5. $\max(p) = \text{delmax}(p)_2$
  6. $\text{delmax}(p)_1 \neq \text{empty} \rightarrow \max(p) \geq \max(\text{delmax}(p)_1)$
  7. $n \geq \max(p) \rightarrow \text{delmax}(\text{insert}(p, n))_1 = p \wedge \max(\text{insert}(p, n)) = n$
  8. $n < \max(p) \rightarrow \max(\text{insert}(p, n)) = \max(p)$
  9. $\text{delmax}(\text{insert}(\text{empty}, n)) = (\text{empty}, n)$
  10. $\max(\text{insert}(\text{empty}, n)) = n$
  - (7. és 8.-nál feltettük, hogy nem üres a prioritásos sor – ez az értelmezési tartomány megszorítása!)

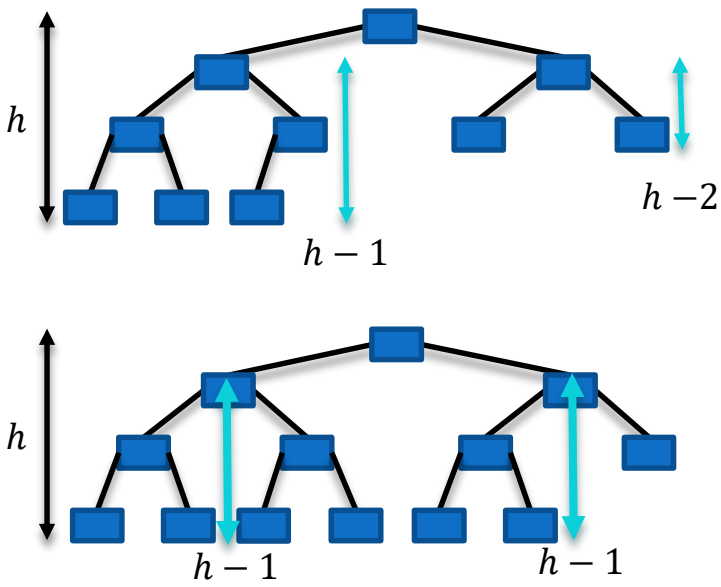
Elsőbbségi (prioritásos) sor

- Kérdés: hogyan ábrázoljuk, mivel reprezentáljuk?
  - Rendezetlen tömbbel, a beérkezési idő szerint
    - max műveletigénye mindig egy maxker, vagyis $\Theta(n)$
  - Rendezett tömbbel
    - insert műveletigénye:
      - A hely megkeresése → logaritmikus keresés $\Theta(\log_2 n)$
      - Tőle jobbra léptetés $\Theta(n)$
      - Összesen $\Theta(n)$
  - Heap (kupac) adatszerkezettel

Kupac (Heap)

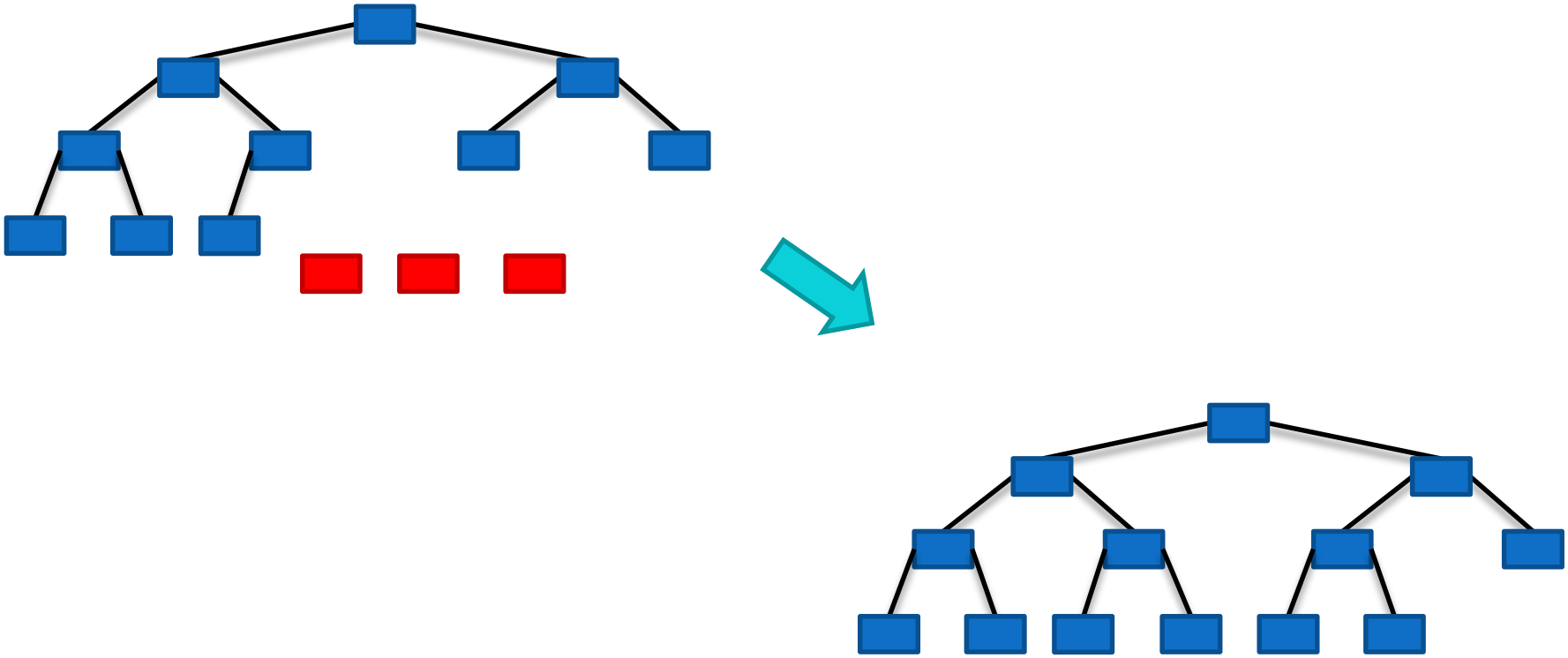
Teljes fák – majdnem teljes fák

- Egy bináris fa teljes, ha
  - a magassága h, és
  - $2^{h+1} - 1$ csomópontja van
- Egy h magasságú bináris fa majdnem teljes, ha
  - Üres, vagy
  - A magassága h, és a bal részfája $h - 1$ magas és majdnem teljes és jobb részfája $h - 2$ magas és teljes, vagy
  - A magassága h, és a bal részfája $h - 1$ magas és teljes és jobb részfája $h - 1$ magas és majdnem teljes



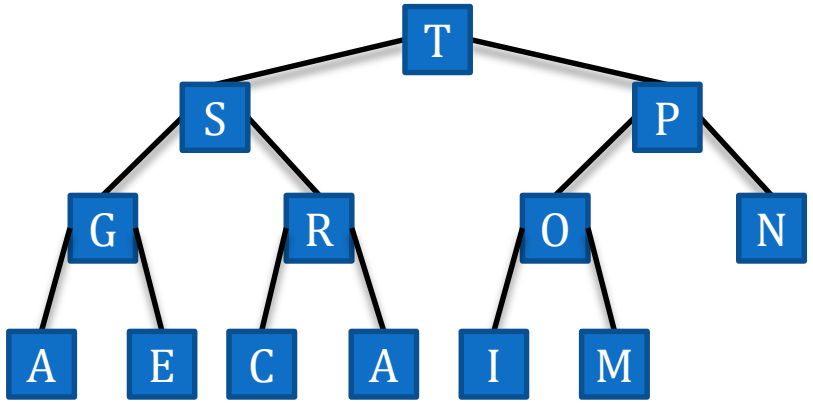
Majdnem teljes fák

- ha megvizsgáljuk a példákat, látjuk, hogy a majdnem teljes fákat balról „töltjük fel”



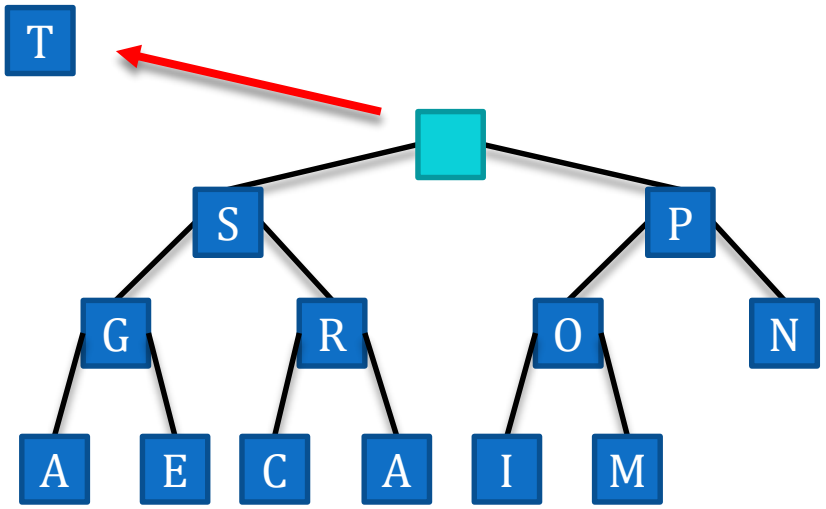
Kupac (Heap)

- Egy majdnem teljes (bináris) fa heap tulajdonságú, ha
  - Üres, vagy
  - A gyökérben lévő kulcs nagyobb, mint mindkét gyerekében, és mindkét részfája is heap tulajdonságú



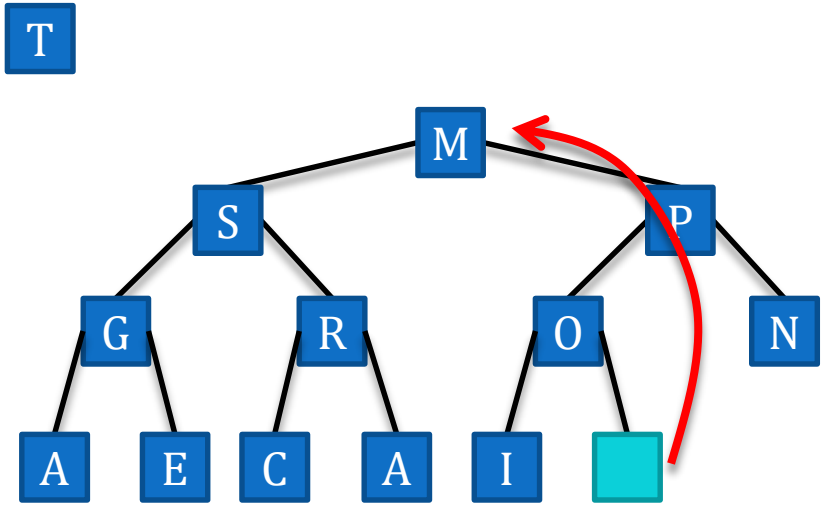
Kupac (Heap)

- A kupacok használhatóak
  - A prioritásos sorok megvalósítására, mivel a gyökérben lévő elem a maximális és ez az, amit a delmax kitöröl
  - A törlés után helyre kell állítani a heap tulajdonságot



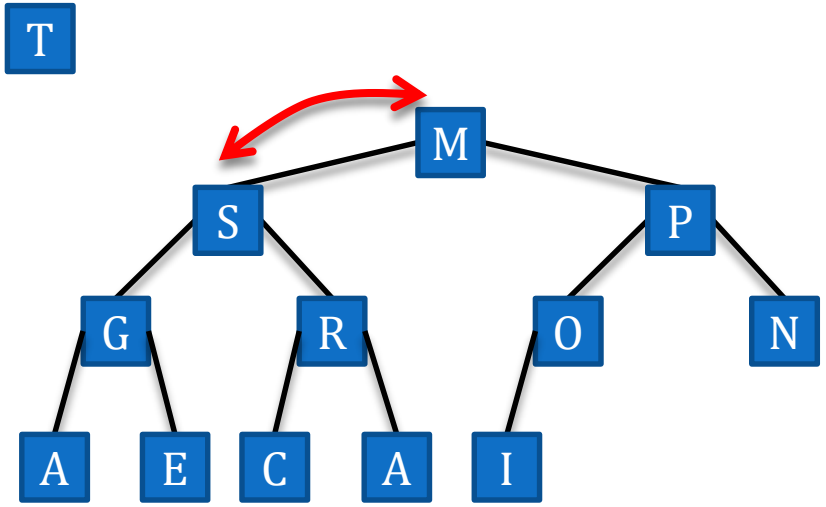
Kupac – helyreállítás

- Az első lépésében a majdnem teljes tulajdonságot állítjuk helyre
  - Vigyük fel a legutolsó elemet a gyökérbe
    - Ezzel a majdnem teljes tulajdonságot teljesítettük
    - Azért működik ez, mert az eredeti fa majdnem teljes volt



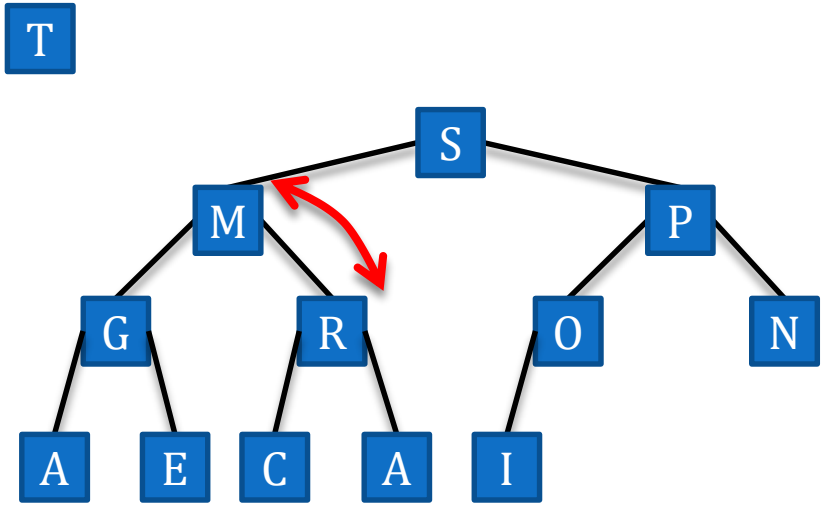
Kupac – helyreállítás

- A kapott majdnem teljes fa azonban nem kupac
  - Tehát a gyökér nem nagyobb a gyerekeinél
  - A helyreállításhoz cseréljük fel a gyökeret a nagyobb gyerekével



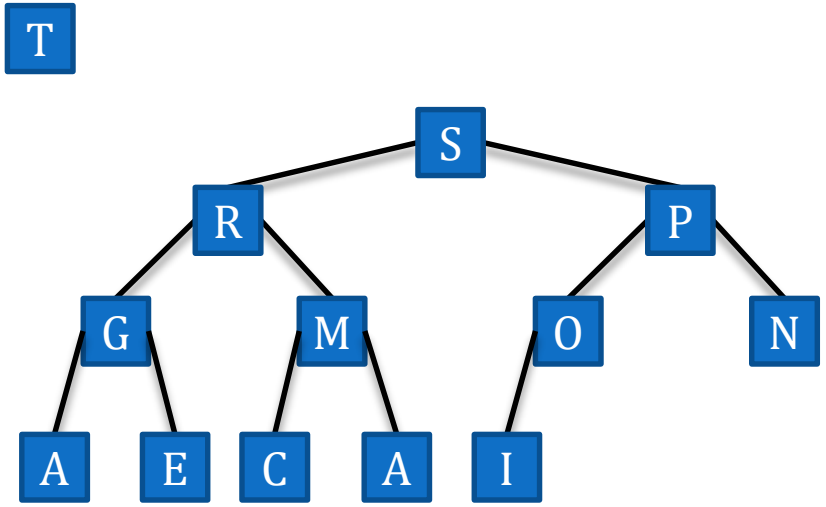
Kupac – helyreállítás

- A jobb részfa kupac, a bal részfa azonban nem teljesíti a kupac tulajdonságot
  - Ismételjük meg az előző lépést a bal részfára



Kupac – helyreállítás

- Ezt a lépést végrehajtva a kapott fa kupac

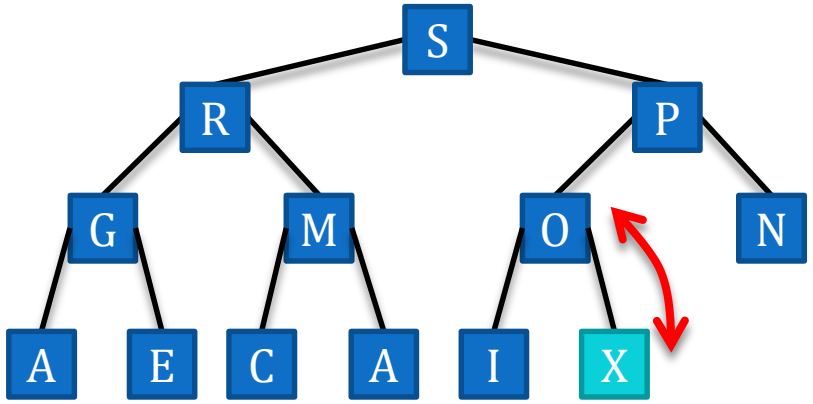


Kupac – Törlés ideje

- Gyökér eltávolítása $\mathcal{O}(1)$
- Utolsó elem a gyökérbe $\mathcal{O}(1)$
- Nagyobb gyerekkel csere $\mathcal{O}(h) = \mathcal{O}(\log n)$
- Összesen (ignorálva a konstansokat) $\mathcal{O}(\log n)$

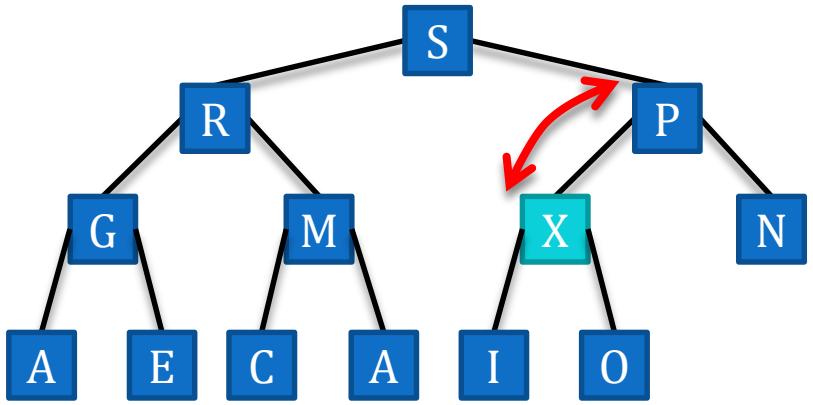
Kupac – elem beszúrása

- Adjunk egy elemet a kupachoz
  - Helyezzük a következő üres pozícióra a jobb szélén
    - Ez kell legyen a következő kitöltendő hely
  - Vigyük felfelé, amíg nagyobb, mint a szülei
  - Újra maximum h csere tehát a beszúrás is $\mathcal{O}(\log n)$



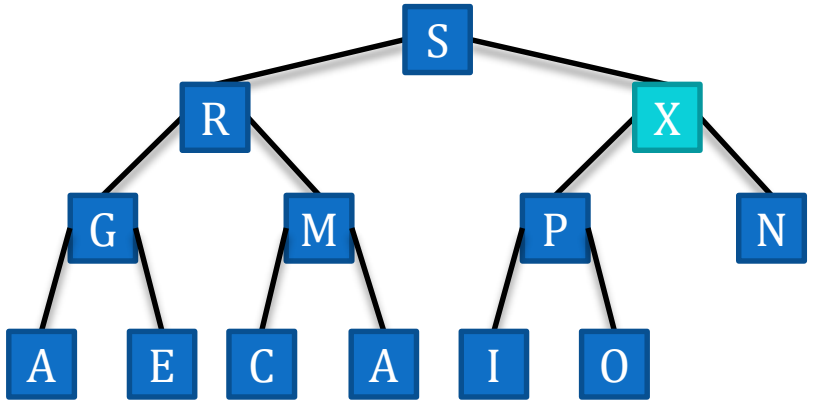
Kupac – elem beszúrása

- Adjunk egy elemet a kupachoz
  - Helyezzük a következő üres pozícióra a jobb szélén
    - Ez kell legyen a következő kitöltendő hely
  - Vigyük felfelé, amíg nagyobb, mint a szülei
  - Újra maximum h csere tehát a beszúrás is $\mathcal{O}(\log n)$



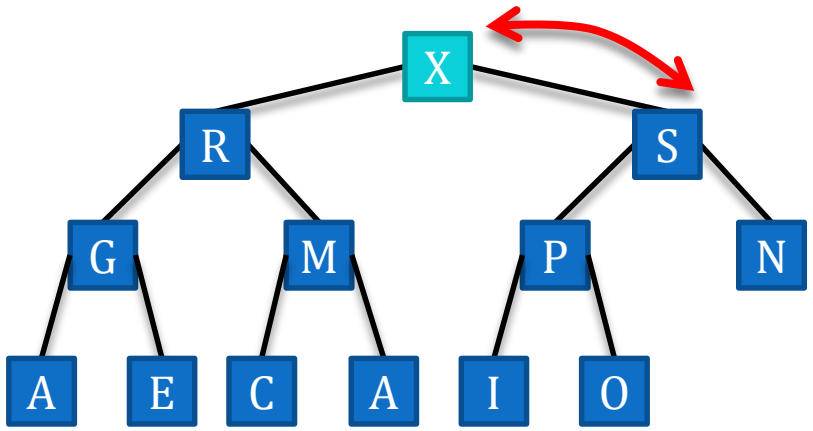
Kupac – elem beszúrása

- Adjunk egy elemet a kupachoz
  - Helyezzük a következő üres pozícióra a jobb szélén
    - Ez kell legyen a következő kitöltendő hely
  - Vigyük felfelé, amíg nagyobb, mint a szülei
  - Újra maximum h csere tehát a beszúrás is $\mathcal{O}(\log n)$



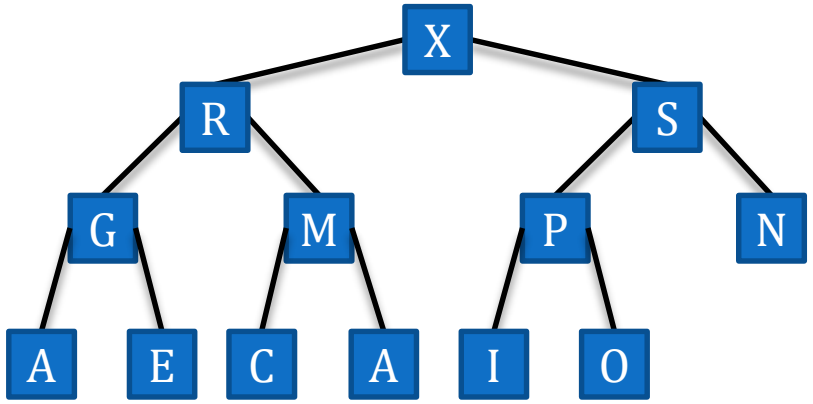
Kupac – elem beszúrása

- Adjunk egy elemet a kupachoz
  - Helyezzük a következő üres pozícióra a jobb szélén
    - Ez kell legyen a következő kitöltendő hely
  - Vigyük felfelé, amíg nagyobb, mint a szülei
  - Újra maximum h csere tehát a beszúrás is $\mathcal{O}(\log n)$



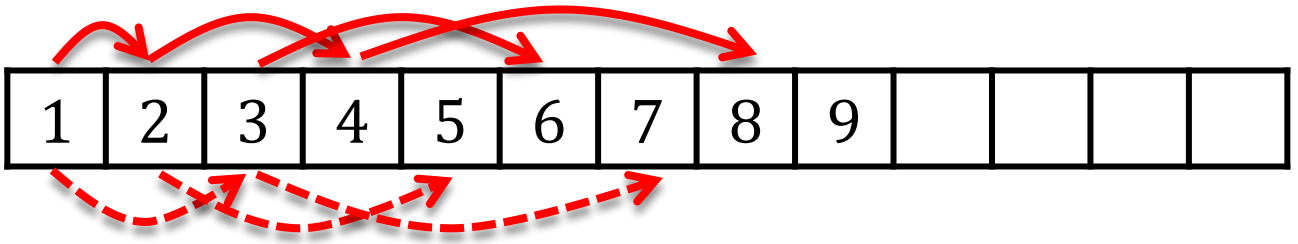
Kupac – elem beszúrása

- Adjunk egy elemet a kupachoz
  - Helyezzük a következő üres pozícióra a jobb szélén
    - Ez kell legyen a következő kitöltendő hely
  - Vigyük felfelé, amíg nagyobb, mint a szülei
  - Újra maximum h csere tehát a beszúrás is $\mathcal{O}(\log n)$



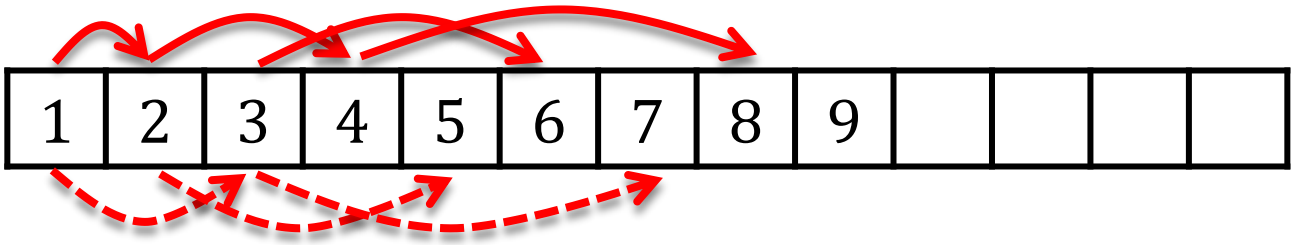
Kupacok – reprezentáció

- Dinamikusan allokált csomópontok és mutatók
  - mint bármilyen más láncolt lista vagy fa
- Használjunk egy tömböt és
  - használjuk ki a „majdnem teljes” tulajdonságot
  - a k csomópont gyerekei a $2k, 2k + 1$-nél vannak
  - a k szülője a $\frac{k}{2}$-nél van
  - ha $k > n$, akkor a csomópont nem létezik



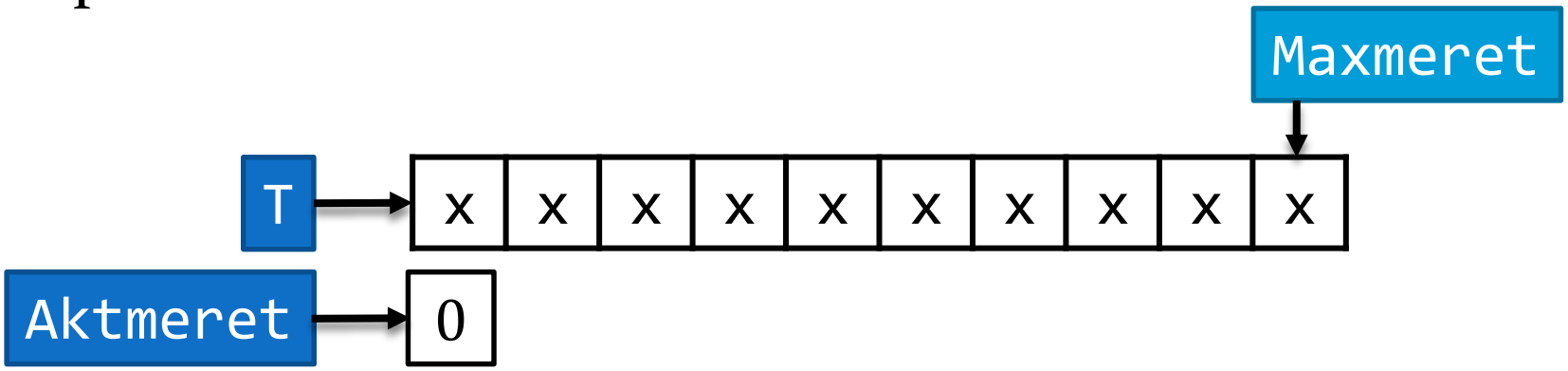
Kupac – hatékonyság

- A beszúró és törlő műveletek egy h magasságú fában
  - $h \leq \log_2 n$
  - Vagyis $\mathcal{O}(\log n)$ az időigénye



Kupac műveletei

- Reprezentáció



- Empty: $\rightarrow K$ az üres kupac létrehozása
  - A reprezentáló tömb létrehozása a `Maxmeretnek` megfelelően
  - `Aktmeret` beállítása nullára
- IsEmpty: $K \rightarrow L$ üres a kupac?
  - `return (Aktmeret = 0)`

Kupac műveletei

- Insert: $K \times N \rightarrow K$ elem betétele a kupacba

Aktmeret≠Maxmeret	
Aktmeret ← Aktmeret + 1 T[Aktmeret] ← ujelem Szulo ← Aktmeret / 2 Gyerek ← Aktmeret	Tele!
$(Szulo \geq 1) \wedge (T[\text{Szulo}] < T[\text{Gyerek}])$	
Csere(T[Szulo], T[Gyerek]) Gyerek ← Szulo Szulo ← Szulo/2	

Kupac műveletei

- Max: $K \rightarrow E$ maximális elem lekérdezése

Aktmeret≠ 0	
return $T[1]$	Üres!

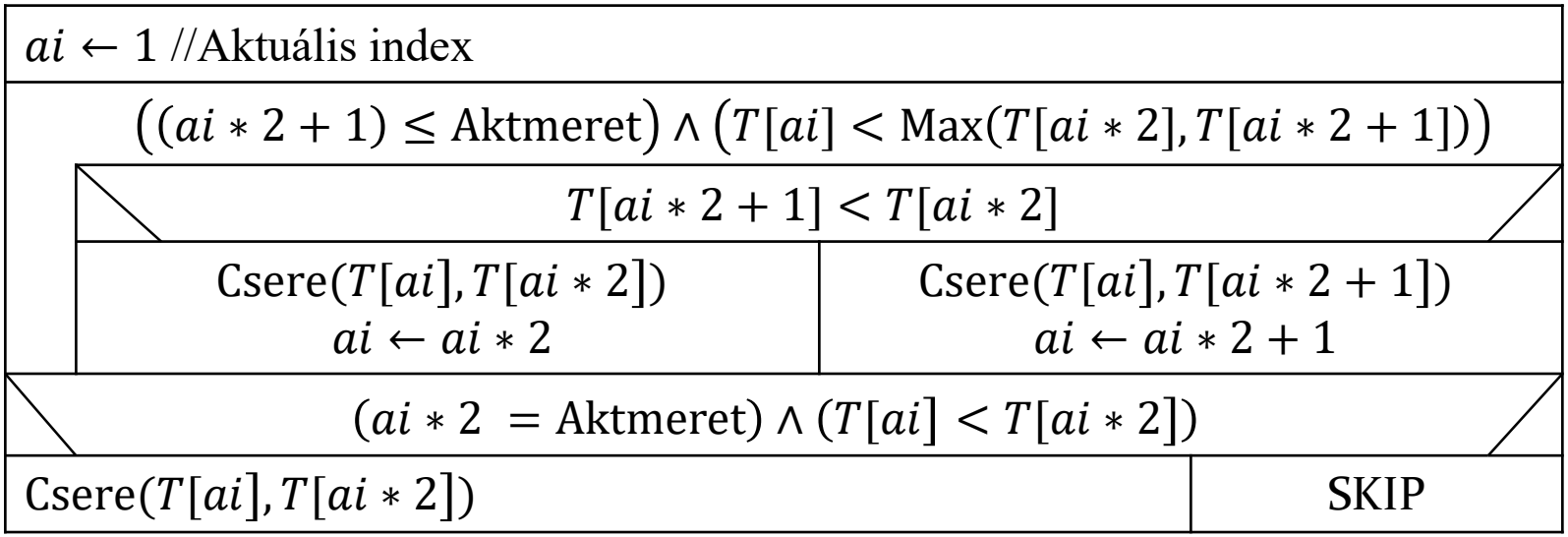
Kupac műveletei

- DelMax: $K \rightarrow K \times N$ maximális elem kivétele a kupacból
  - A maximális elem értékét a maxelem-ben kapjuk

Aktmeret≠ 0	
Maxelem← $T[1]$ $T[1] \leftarrow T[\text{Aktmeret}]$ Aktmeret ← Aktmeret − 1 Sullyeszt return Maxelem	Üres!

Kupac műveletei

- Sullyeszt: feltételezzük, hogy a kupacban két jó részfa van, de a $T[1]$-t megfelelően le kell süllyeszteni



Prioritásos sor megvalósítása kupaccal

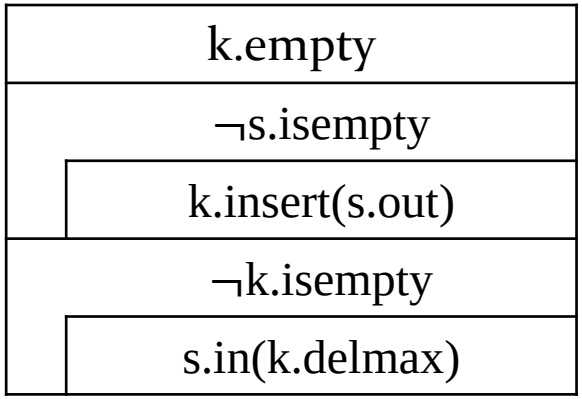
- `Empty` (az üres prioritásos sor konstruktor – létrehozás)
  - `Empty` (üres kupac)
- `isempty` (üres a prioritásos sor?)
  - `IsEmpty` (üres a kupac?)
- `insert` (elem betétele a prioritásos sorba)
  - `Insert` (elem betétele a kupacba)
- `delmax` (maximális elem kivétele a prioritásos sorból)
  - `DelMax` (maximális elem kivétele a kupacból)
- `max` (maximális elem lekérdezése a prioritásos sorból)
  - `Max` (maximális elem lekérdezése a kupacból)

Kupacrendezés

- Használjuk a kupacokat rendezésre!
  - Szúrd be az elemeket egy kupacba!
  - Amíg a sor ki nem ürül, vedd ki a kupacból a maximális elemet, és tedd az eredmény (rendezett) sorba!

Kupacrendezés

- Az *s* sorban lévő elemeket rendezzük a *k* kupac segítségével!



Kupacrendezés

- A kupacok hatékony rendezőt adnak
  - Szúrjunk be minden elemet a kupacba n elem, ezekhez kell $\mathcal{O}(\log_2 n)$
    - összesen: $\mathcal{O}(n \log_2 n)$
  - sorrendben távolítsuk el az elemeket n elem, ezekhez kell $\mathcal{O}(\log_2 n)$
    - összesen: $\mathcal{O}(n \log_2 n)$
- Összesen $\mathcal{O}(n \log n)$
  - ignorálva a 2-es konstanst, stb.

Heapsort vs. Quicksort

- Használjuk inkább a Heapsort-ot?
- A Quicksort általában gyorsabb
  - kevesebb összehasonlítás és csere
  - néhány empirikus adat

	Quick		Heap		Beszúró	
	Összehasonlítás	Csere	Összehasonlítás	Csere	Összehasonlítás	Csere
100	712	148	2842	581	2596	899
200	1682	328	9736	9736	10307	3503
500	5102	919	53113	4042	62746	21083

Quicksort ⇔ Heap Sort

- Quicksort
  - általában gyorsabb
  - néha $O(n^2)$
    - jobb pivot választás csökkenti a valószínűségét
    - ha átlagosan jó végrehajtási időt akarunk, használjuk ezt
    - üzleti alkalmazások, információs rendszerek
- Heap Sort
  - általában lassúbb
  - Garantált O(n log n) … lehet rá építeni, ezt a tervezésben felhasználni!
  - használjuk például real-time rendszerekhez
    - Az idő egy megszorítás

QuickSort

Másik felosztási algoritmus

Gyorsrendezés – másik felosztási algoritmus

- Többféle algoritmus létezik a felosztásra
  - Nézzünk meg még egyet – pszeudokódban
- FELOSZT(A,p,r)

```
x ← A[r]
i ← p - 1
for j ← p to r-1
  do if A[j] ≤ x
    then i ← i+1
      A[i] ↔ A[j] csere
A[i+1] ↔ A[r] csere
return i+1
```

Gyorsrendezés – másik felosztási algoritmus

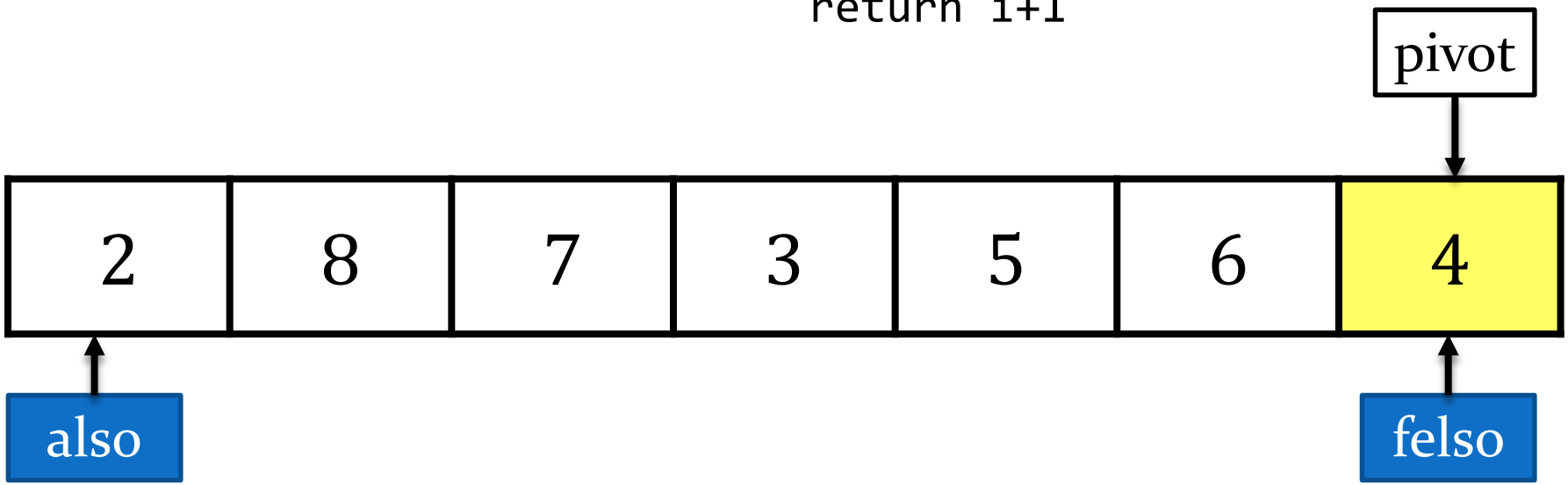
- Többféle algoritmus létezik a felosztásra
  - Nézzünk meg még egyet – pszeudokódban
- FELOSZT(A,also,felso)

```
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```

Quicksort – Megosztás

- Bármelyik elem jó strázsának, válasszuk a felsőt

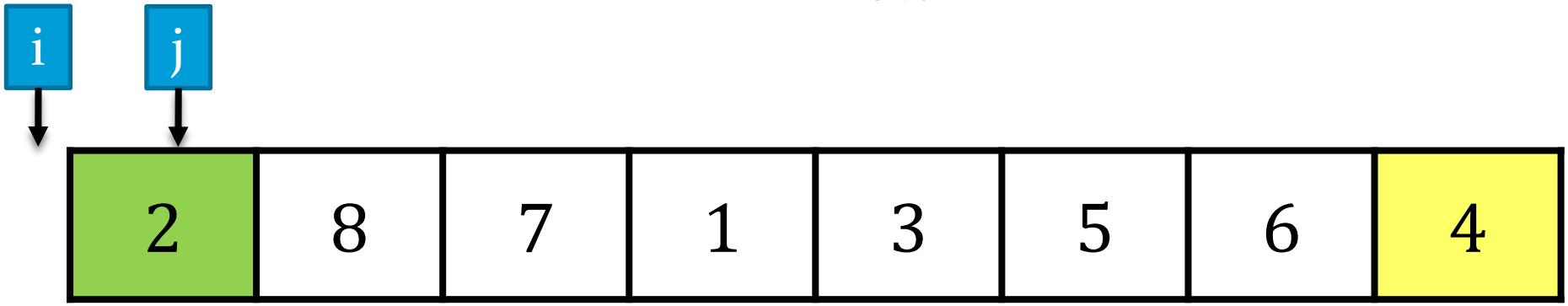
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- Induljunk el az i és j indexszel!

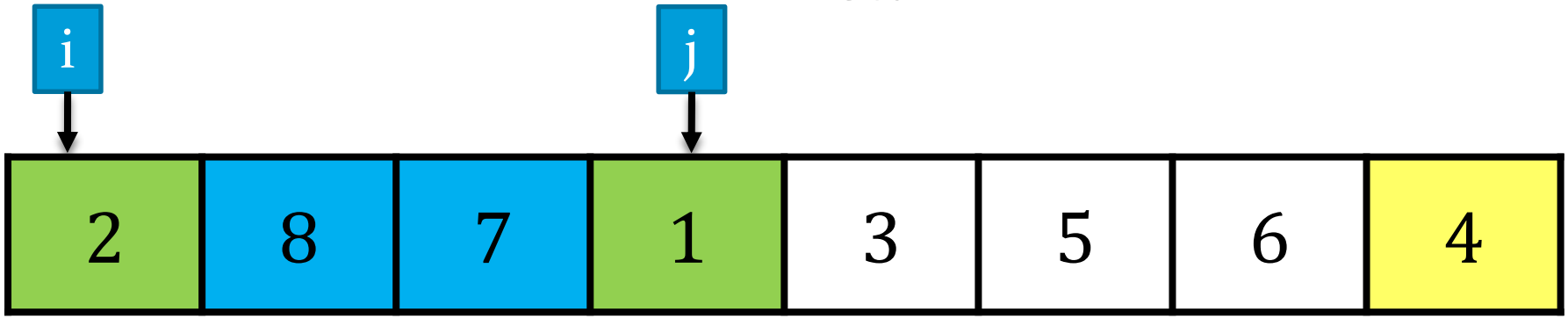
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- A j pozíción találunk a strázsánál kisebb elemet
- Növeljük az i-t

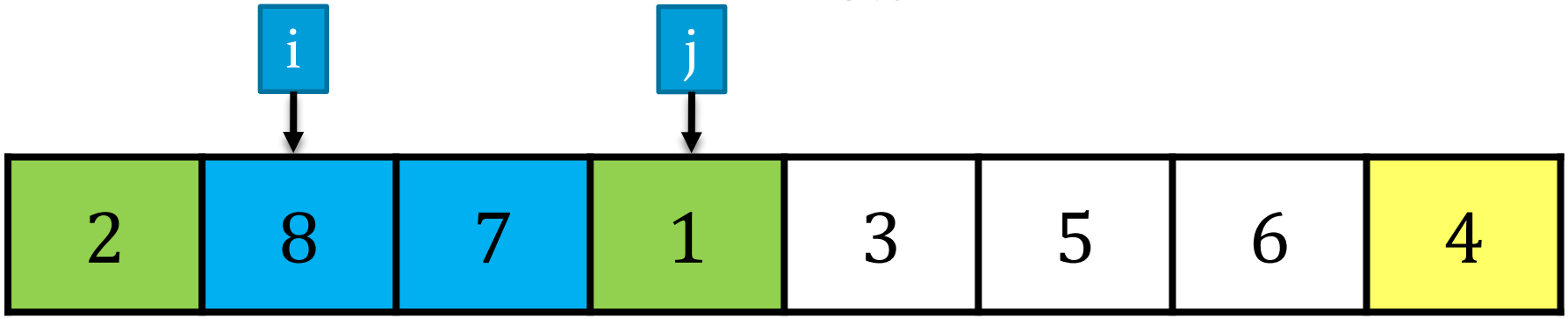
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- És megcseréljük az i és a j elemet

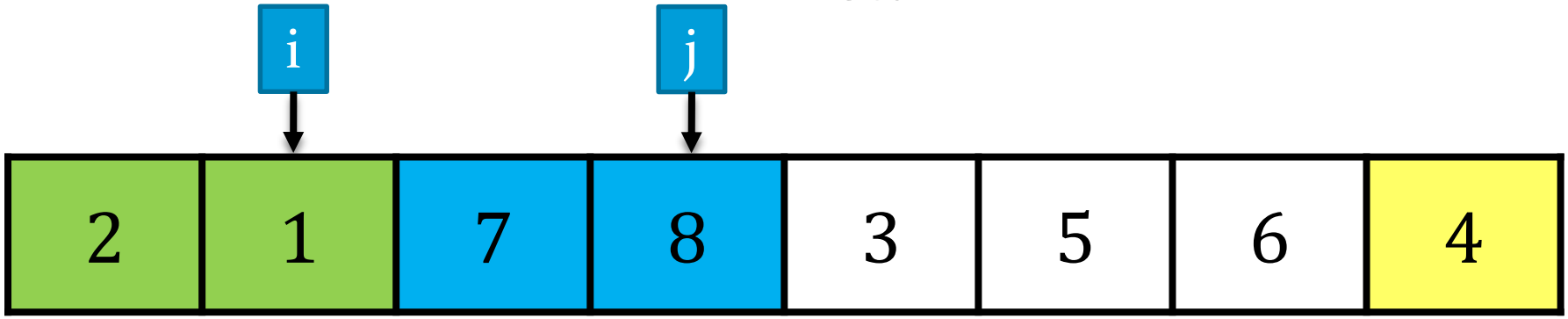
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- Majd folytatjuk a j növelését

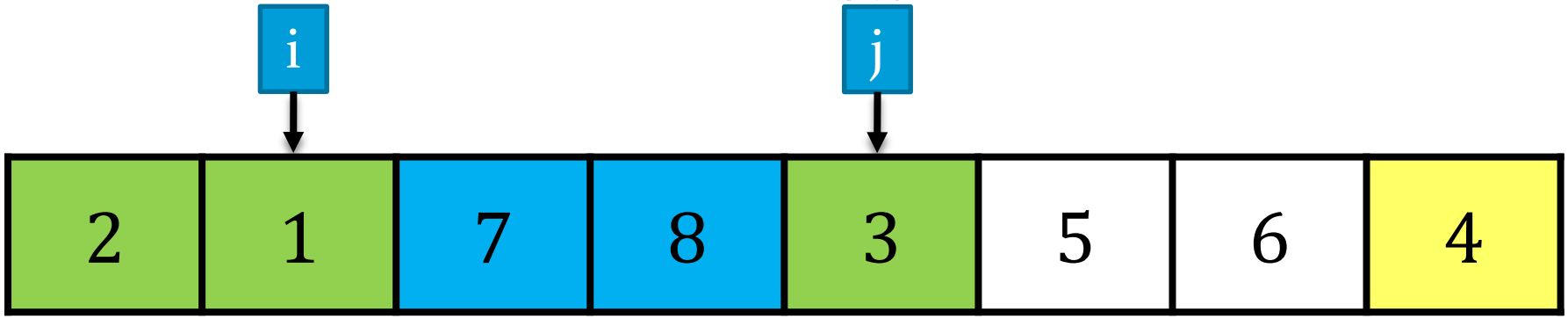
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- Újabb találat

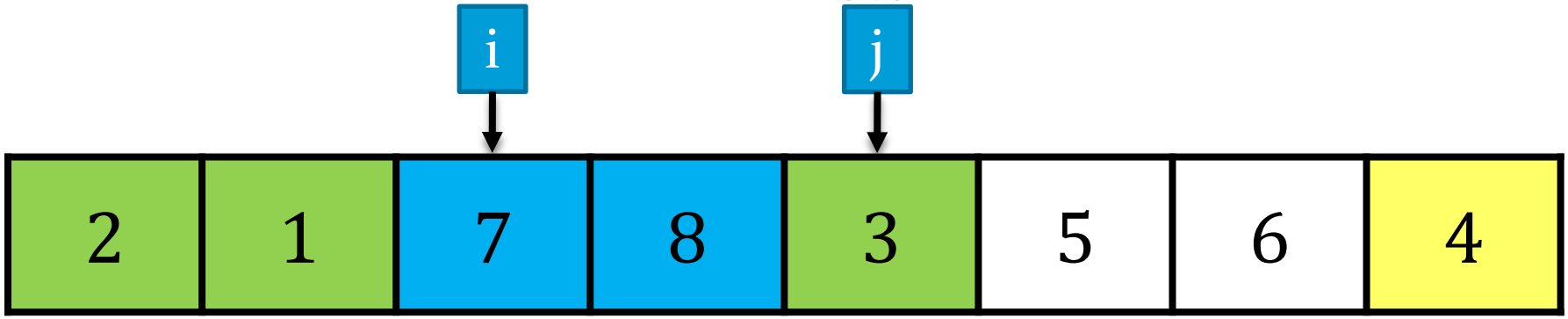
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- Újabb csere

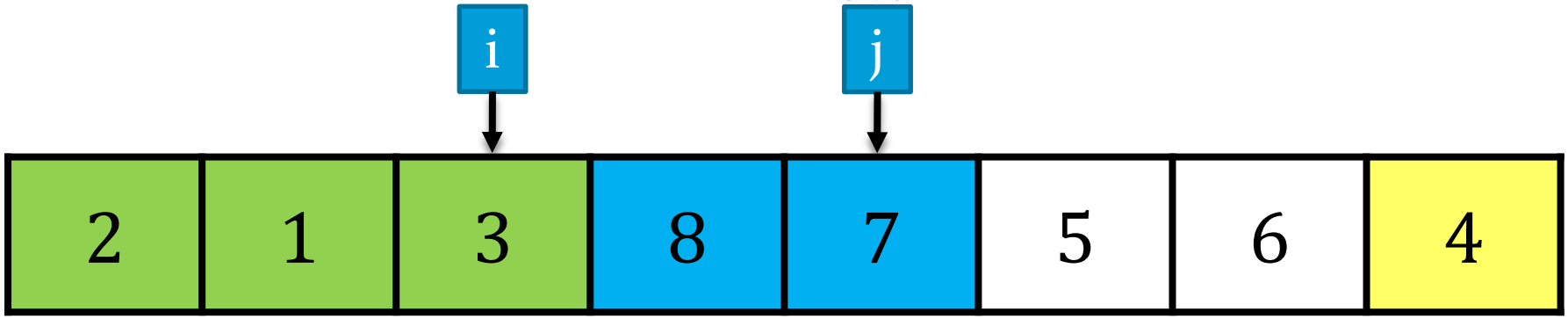
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- …

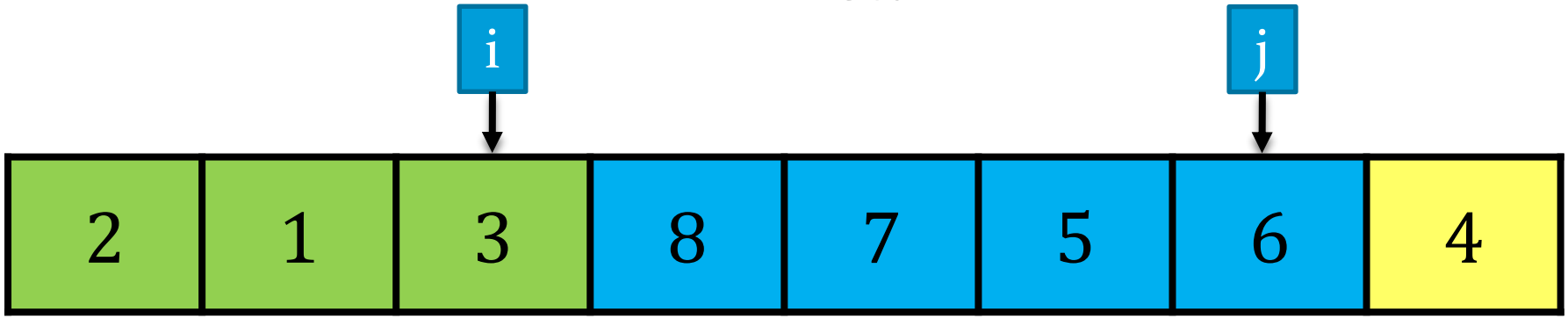
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- Végezetül a strázsa elemet a helyére tesszük

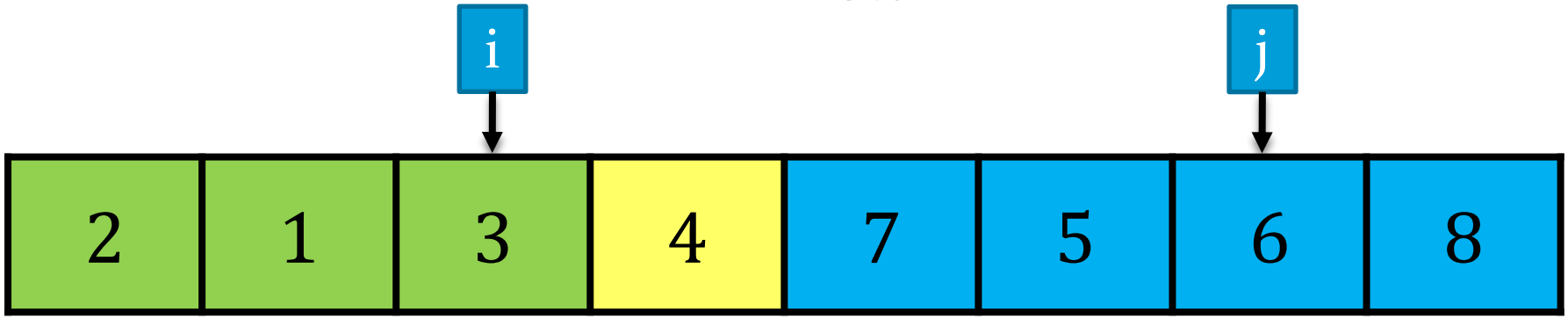
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- …

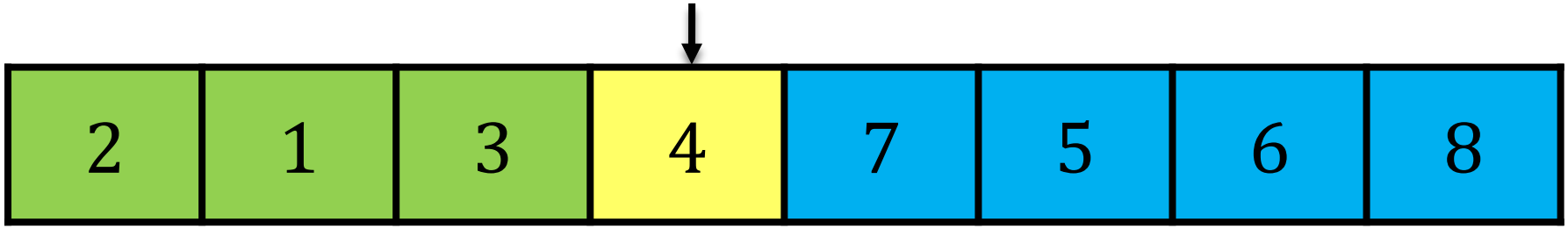
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- Legvégül visszatérünk a strázsa pozíciójával

```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Leszámláló rendezés

Leszámláló rendezés

- Tegyük fel, hogy van n db bemeneti elem, s ezek mindegyike 1 és k közötti egész szám
- Az alapötlet: meghatározzuk minden egyes x bemeneti elemre azoknak az elemeknek a számát, amelyek kisebbek, mint az x
- Ezután x-et közvetlenül a saját pozíciójára tudom helyezni
- Legyen a bemenet az $A[1..n]$ tömb, a kimenet a $B[1..n]$ tömb
  - Mindkettő hossza: $\text{hossz}[A] = \text{hossz}[B] = n$
- Szükség van még egy $C[1..k]$ tömbre átmeneti munkaterületként

Leszámláló rendezés

1. Végigmegyünk az A-n, és ha egy elem értéke i, akkor megnöveljük $C[i]$ értékét eggyel.
2. Minden i-re $1..k$ között meghatározzuk, hogy hány olyan bemeneti elem van, amelyiknek az értéke $\leq$ i (összegzés C-n)
3. Minden i-re $n..1$ között $A[i]$-t betesszük B megfelelő pozíciójába – ezt a C-ből állapítjuk meg

- Ha betettük, akkor a $C[A[i]]$ értékét csökkentjük, így a következő vele egyenlő elem már elé kerül, vagyis így stabil lesz a rendezés, az egyenlő elemeknél megtartja az eredeti sorrendet

Leszámláló rendezés

- Az A tömb elemeit leszámláljuk a C tömbbe
  - A C-ben az i helyen az i-vel egyenlő elemek száma szerepel
  - Végigmegyünk az A-n és ha egy elem értéke i, akkor a $C[i]$ értéket megnöveljük
- A

3	6	4	1	3	4	1	4

- C

2	0	2	3	0	1

Leszámláló rendezés

- Minden i-re 1 … k között meghatározzuk, hogy hány olyan bemeneti elem van, amelyiknek az értéke $\leq i$ (összegzés C-n)

- A

3	6	4	1	3	4	1	4

- C

2	2	4	7	7	8

Leszámláló rendezés

- Minden i-re n … 1 között $A[i]$-t betesszük B megfelelő pozíciójába - ezt a C-ből állapítjuk meg.

- A

3	6	4	1	3	4	1	4

- C

2	2	4	7	7	8

- B

						4	

Leszámláló rendezés

- Minden i-re n … 1 között $A[i]$-t betesszük B megfelelő pozíciójába - ezt a C-ből állapítjuk meg.

- A

3	6	4	1	3	4	1	4

- C

2	2	4	6	7	8

- B

	1					4	

Leszámláló rendezés

- Minden i-re n … 1 között $A[i]$-t betesszük B megfelelő pozíciójába - ezt a C-ből állapítjuk meg.

- A

3	6	4	1	3	4	1	4

- C

1	2	4	6	7	8

- B

	1				4	4	

Leszámláló rendezés

- Minden i-re n … 1 között $A[i]$-t betesszük B megfelelő pozíciójába - ezt a C-ből állapítjuk meg.

- A

3	6	4	1	3	4	1	4

- C

1	2	4	5	7	8

- B

	1		3		4	4	

Leszámláló rendezés

- Minden i-re n … 1 között $A[i]$-t betesszük B megfelelő pozíciójába - ezt a C-ből állapítjuk meg.

- A

3	6	4	1	3	4	1	4

- C

1	2	3	5	7	8

- B

1	1		3		4	4	

Leszámláló rendezés

- Minden i-re n … 1 között $A[i]$-t betesszük B megfelelő pozíciójába - ezt a C-ből állapítjuk meg.

- A

3	6	4	1	3	4	1	4

- C

0	2	3	5	7	8

- B

1	1		3	4	4	4	

Leszámláló rendezés

- Minden i-re n … 1 között $A[i]$-t betesszük B megfelelő pozíciójába - ezt a C-ből állapítjuk meg.

- A

3	6	4	1	3	4	1	4

- C

0	2	3	4	7	8

- B

1	1		3	4	4	4	6

Leszámláló rendezés

- Minden i-re n … 1 között $A[i]$-t betesszük B megfelelő pozíciójába - ezt a C-ből állapítjuk meg.

- A

3	6	4	1	3	4	1	4

- C

0	2	3	4	7	7

- B

1	1	3	3	4	4	4	6

Leszámláló rendezés

- Minden i-re n … 1 között $A[i]$-t betesszük B megfelelő pozíciójába - ezt a C-ből állapítjuk meg.

- A

3	6	4	1	3	4	1	4

- C

0	2	2	4	7	7

- B

1	1	3	3	4	4	4	6

Leszámláló rendezés

Az algoritmus pszeudokódja:

```
for i←1 to k do
    C[i]←0
for i←1 to hossz(A) do
    C[A[i]]←C[A[i]]+1
for i←2 to k do
    C[i]←C[i]+C[i-1]
for i←hossz(A) downto 1 do
    B[C[A[i]]]←A[i]
    C[A[i]]←C[A[i]]-1
```

C-ben az i-vel egyenlő elemek száma

C-ben az i-nél kisebb, vagy egyenlő elemek száma

Leszámláló rendezés

- Futási idő:
  - 1. for ciklus: $\Theta(k)$
  - 2. for ciklus: $\Theta(n)$
  - 3. for ciklus: $\Theta(k)$
  - 4. for ciklus: $\Theta(n)$
- Így a teljes időigény: $\Theta(k + n)$
  - Ha $k = \Theta(n)$, akkor a rendezés futási ideje $\Theta(n)$!
- Ez nem összehasonlító rendezés
  - A helyigénye viszont nagyobb ☹