Rendezések 2.

10. előadás

QuickSort – újra

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

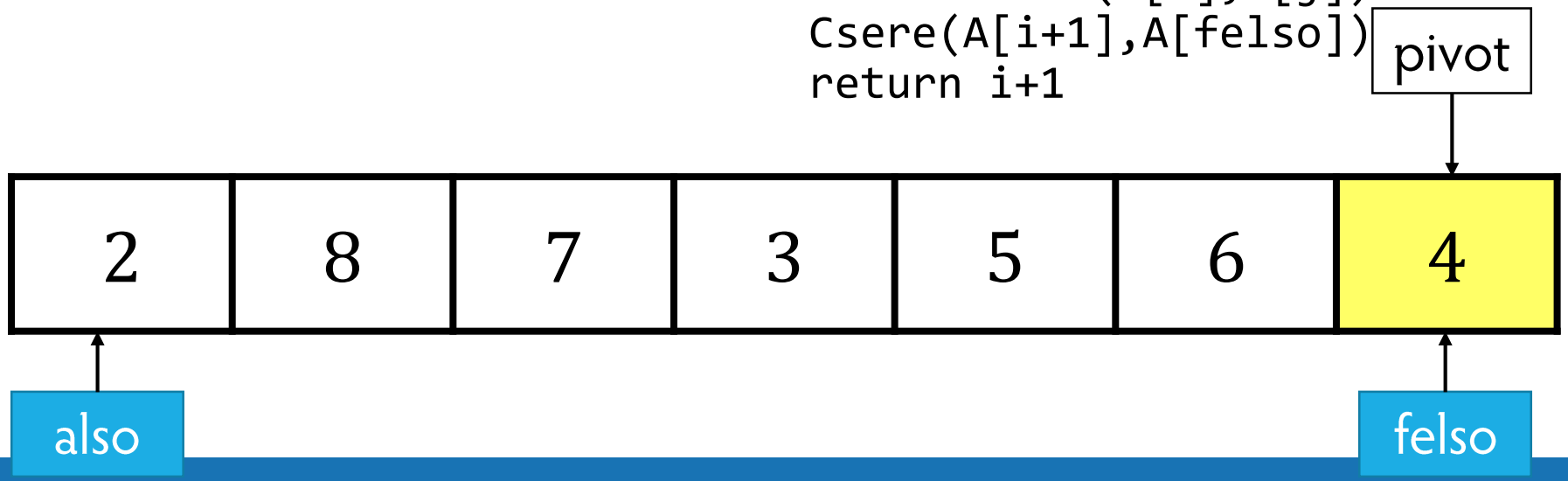
- Többféle algoritmus létezik a felosztásra
  - Nézzünk meg még egyet – pszeudokódban
- FELOSZT(A,also,felso)

```
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

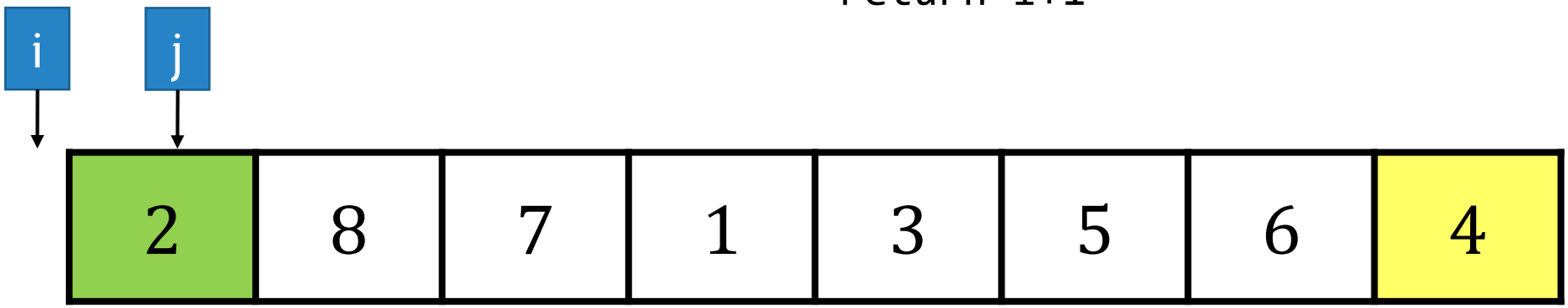
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- Induljunk el az i és j indexszel!

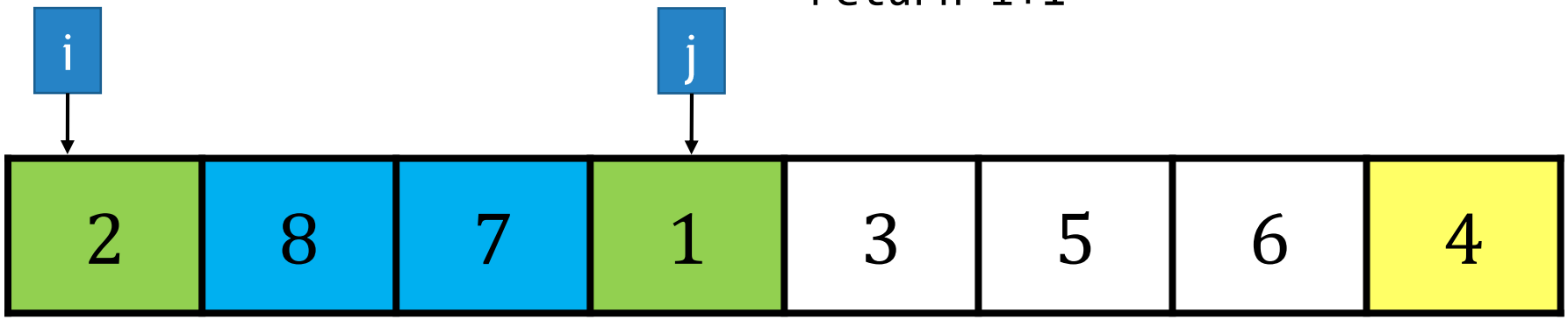
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

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

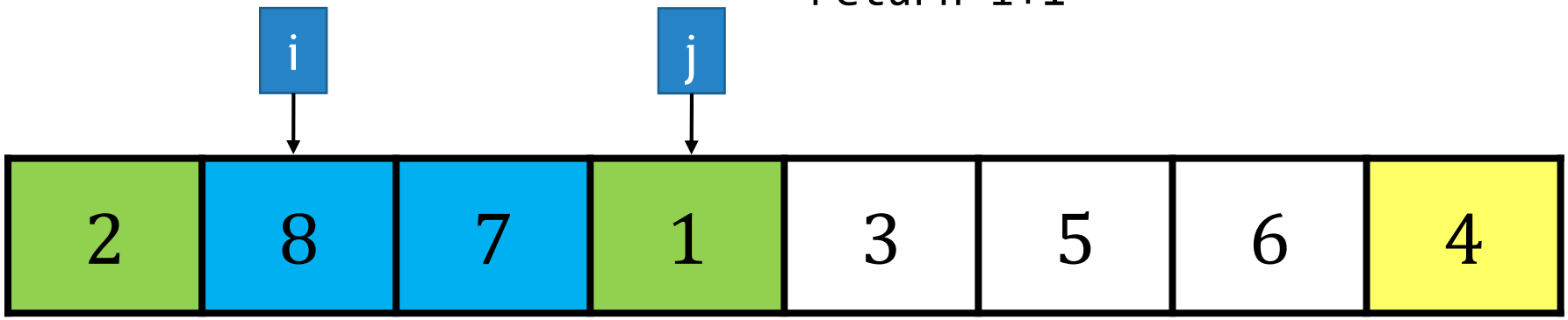
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- És megcseréljük az i és a j elemet

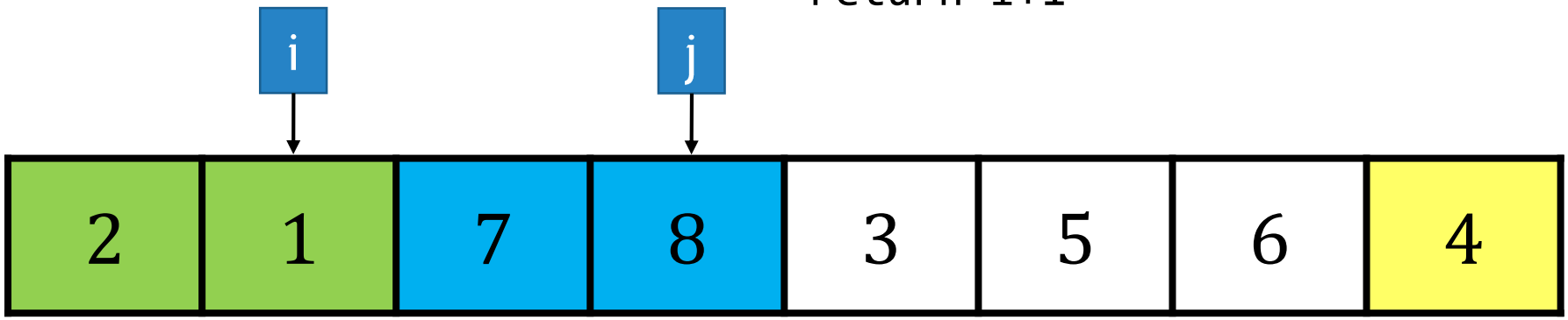
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- Majd folytatjuk a j növelését

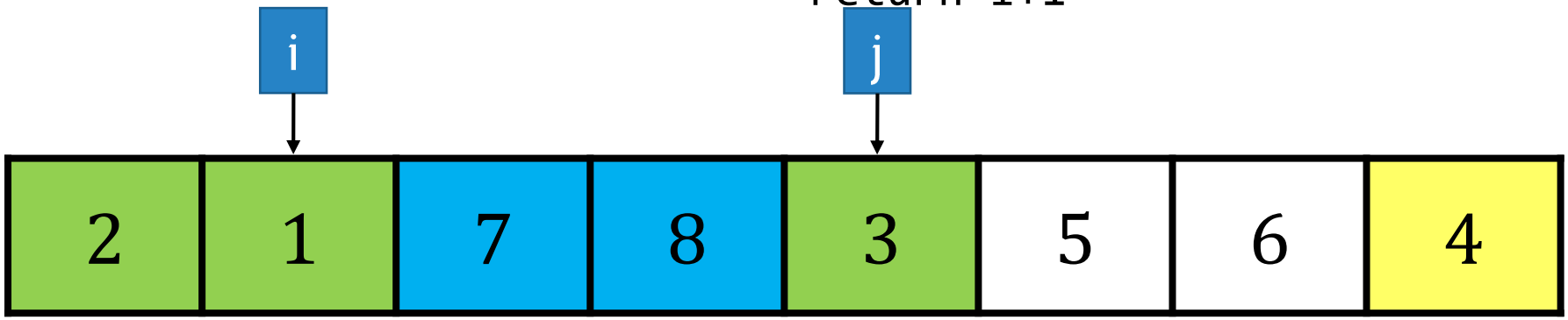
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- Újabb találat

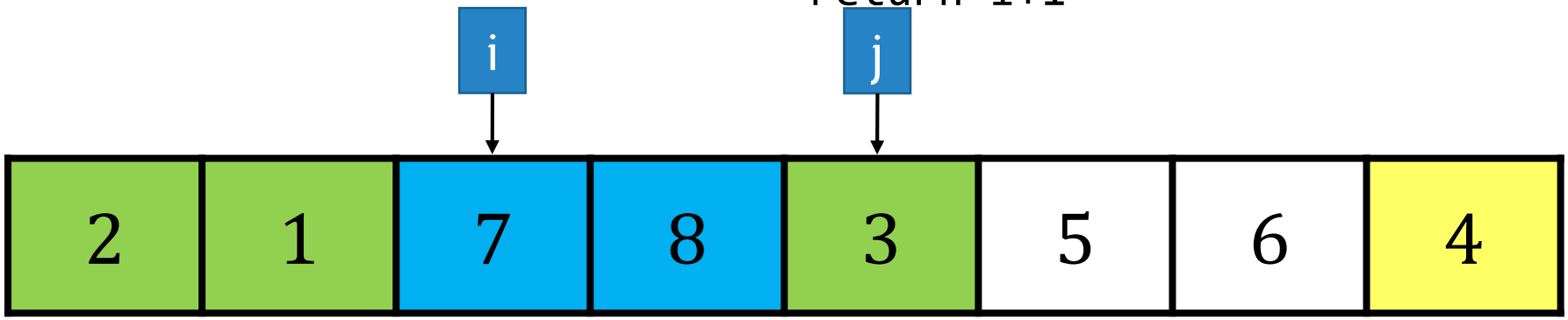
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- Újabb csere

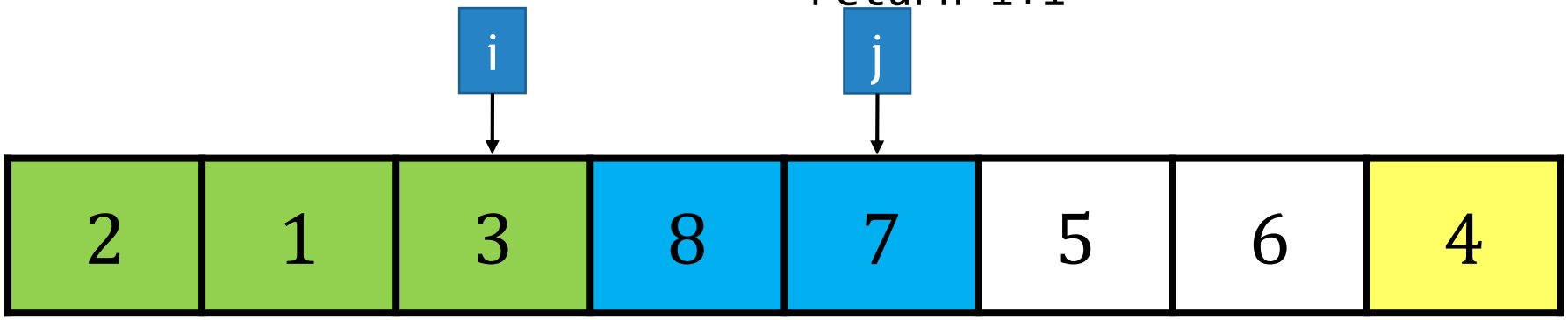
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- …

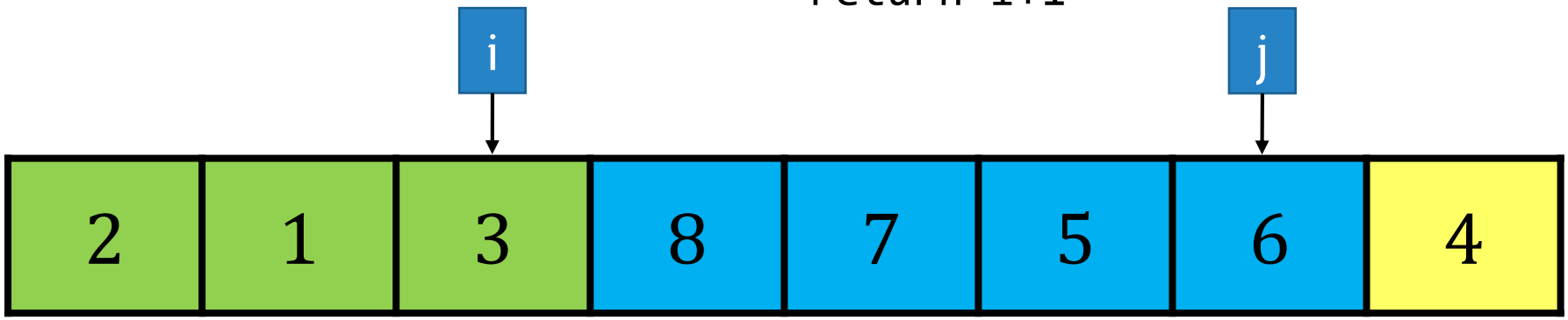
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- Végezetül a strázsa elemet a helyére tesszük

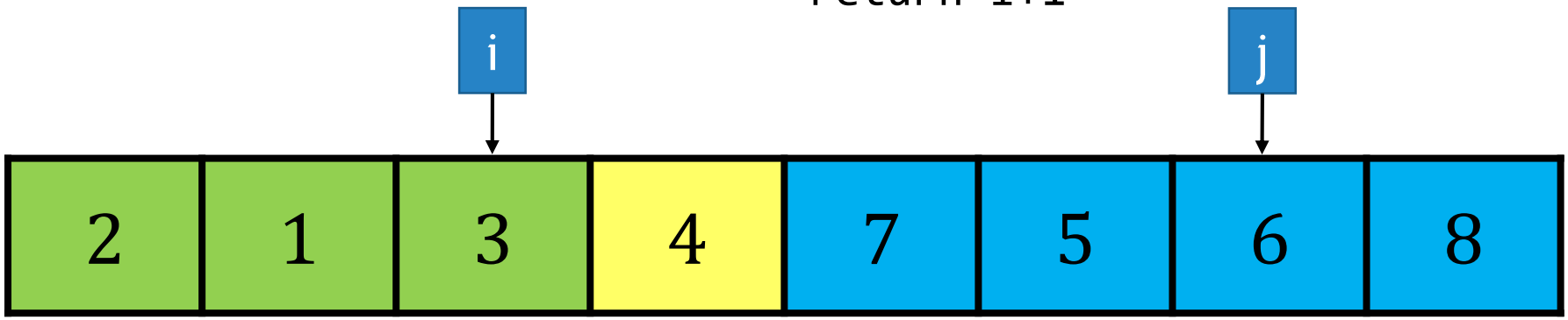
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- …

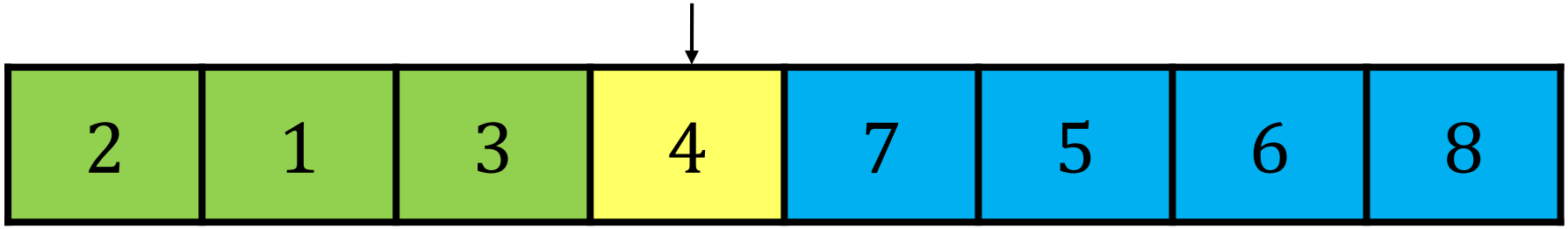
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Quicksort – Megosztás

- Legvégül visszatérünk a strázsa pozíciójával

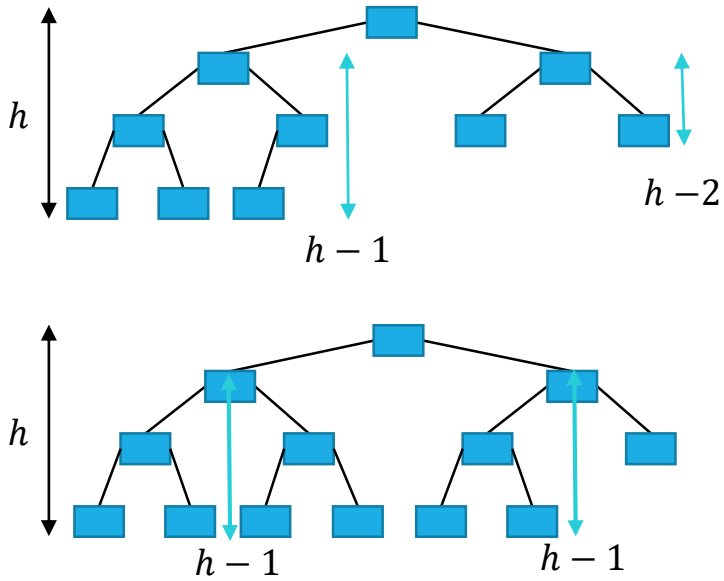
```
FELOSZT(A,also,felso)
str←A[felso]
i←also-1
for j←also to felso-1
  do if A[j]≤str
    then i←i+1
      Csere(A[i],A[j])
Csere(A[i+1],A[felso])
return i+1
```



Kupac (Heap)

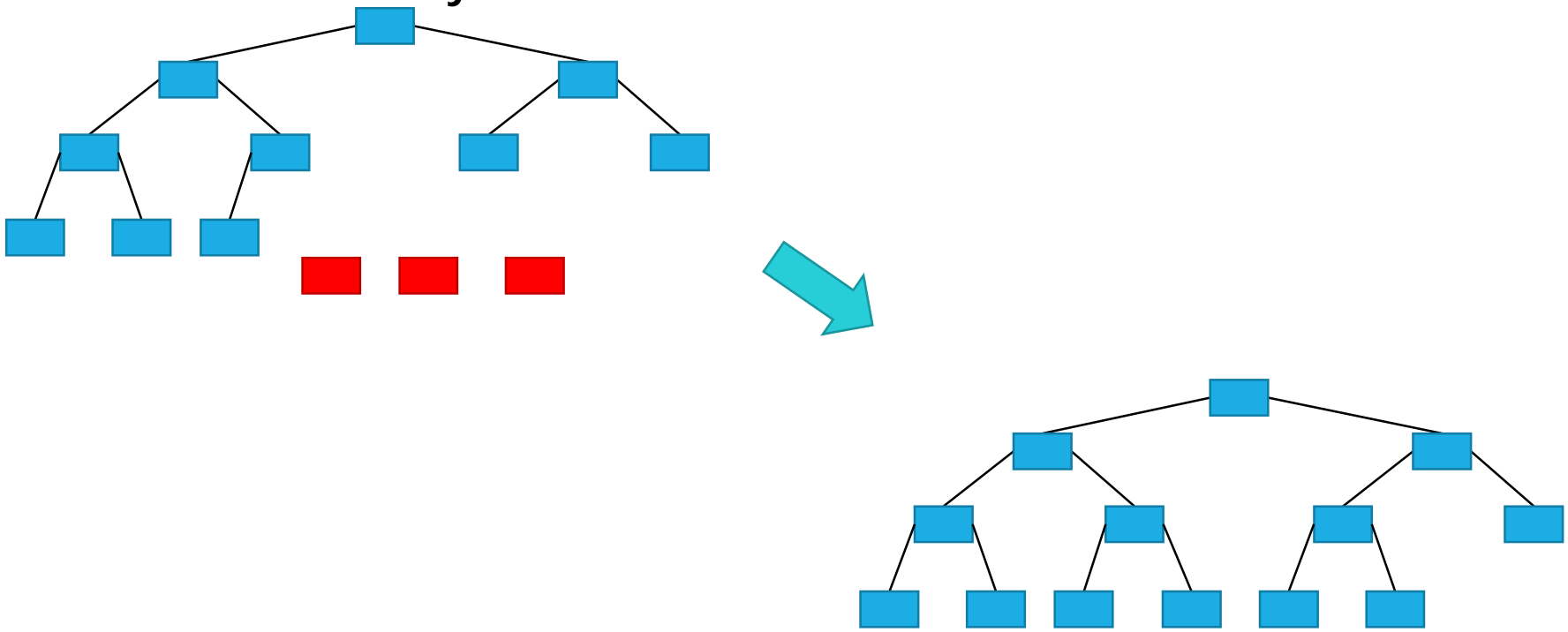
Teljes fák – majdnem teljes fák

- Egy bináris fa teljes, ha
  - a magassága h, és
  - $2^h - 1$ 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



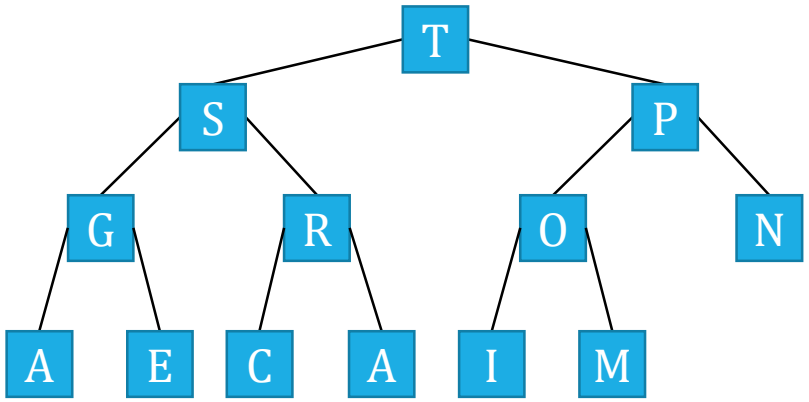
Majdnem teljes fák

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



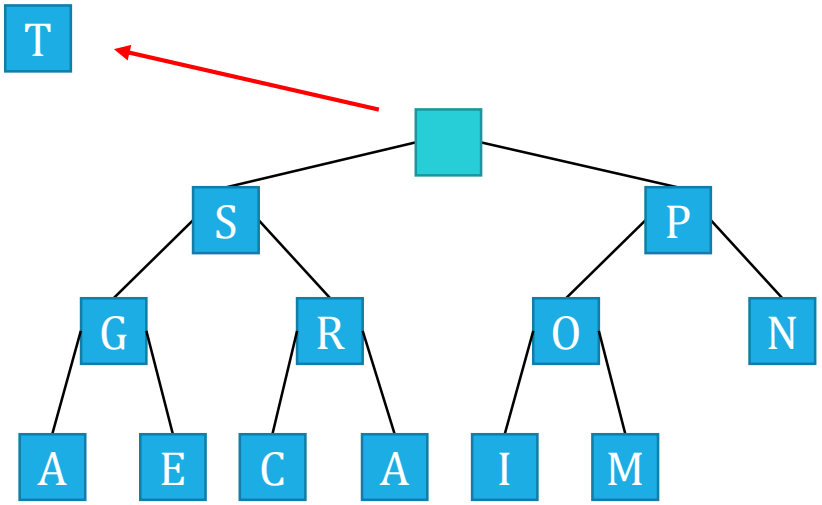
Kupac (Heap)

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



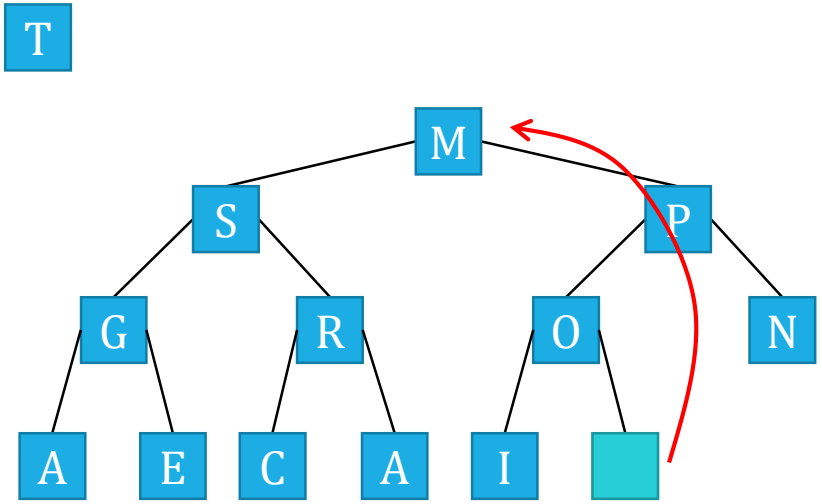
Kupac (Heap)

- 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



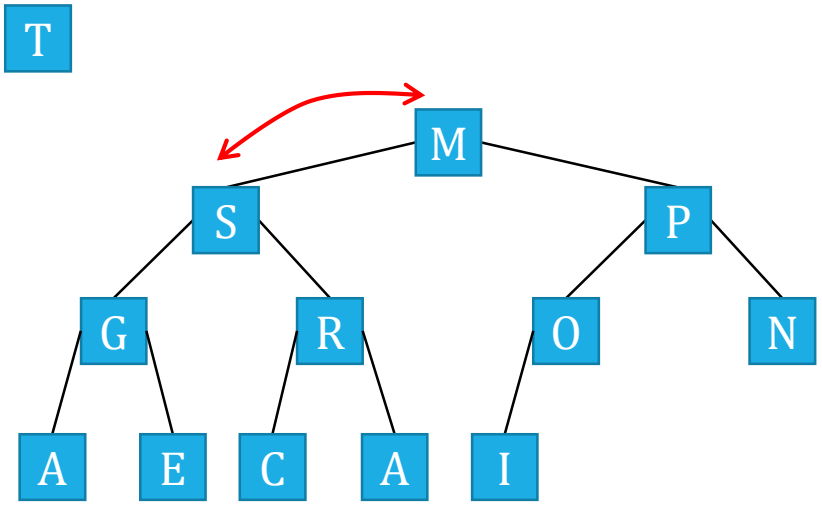
Kupac – helyreállítás

- 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



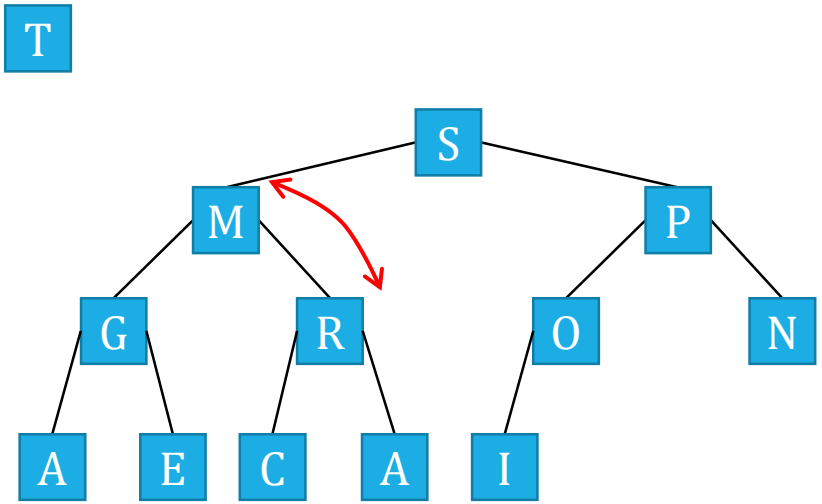
Kupac – helyreállítás

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



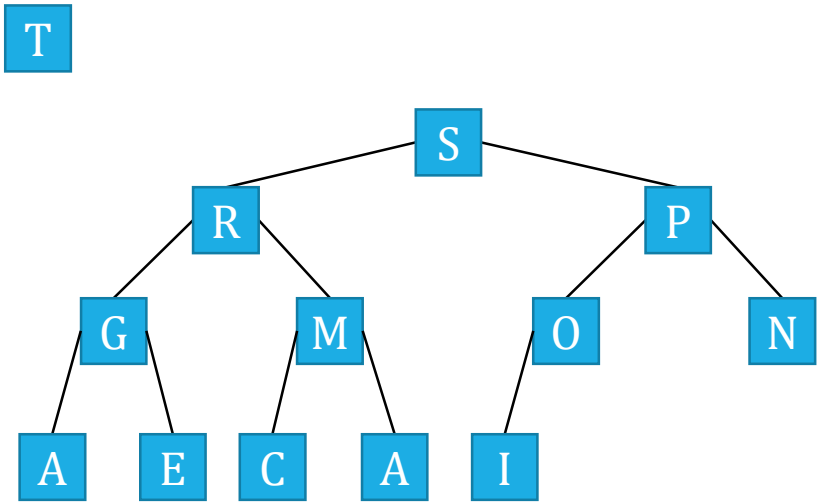
Kupac – helyreállítás

- 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

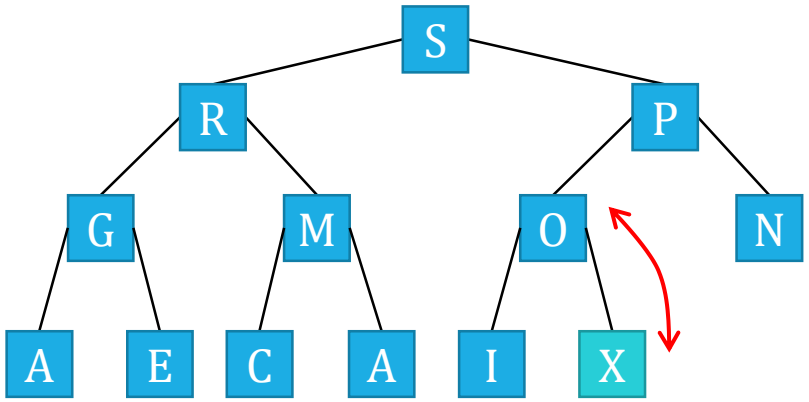


Kupac – Törlés ideje

- 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) $\mathcal{O}(\log n)$

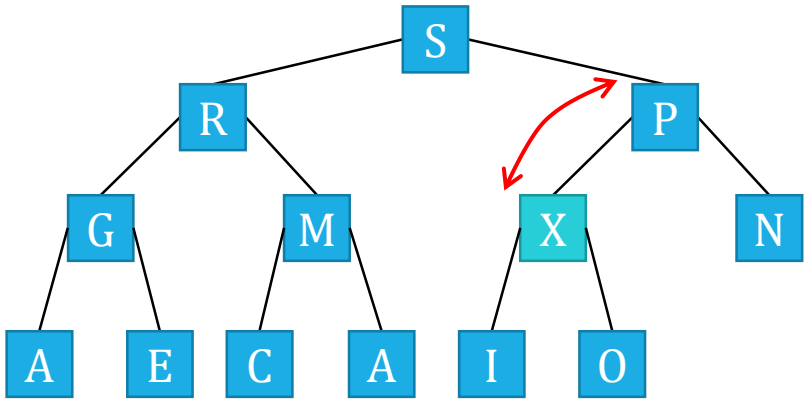
Kupac – elem beszúrása

- 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 $\mathcal{O}(\log n)$



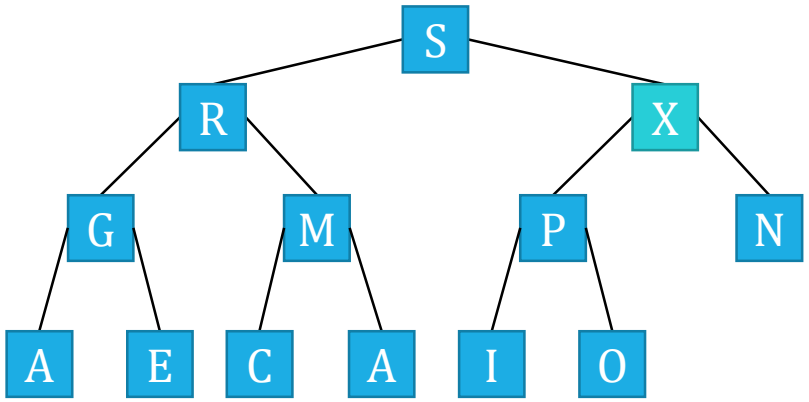
Kupac – elem beszúrása

- 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 $\mathcal{O}(\log n)$



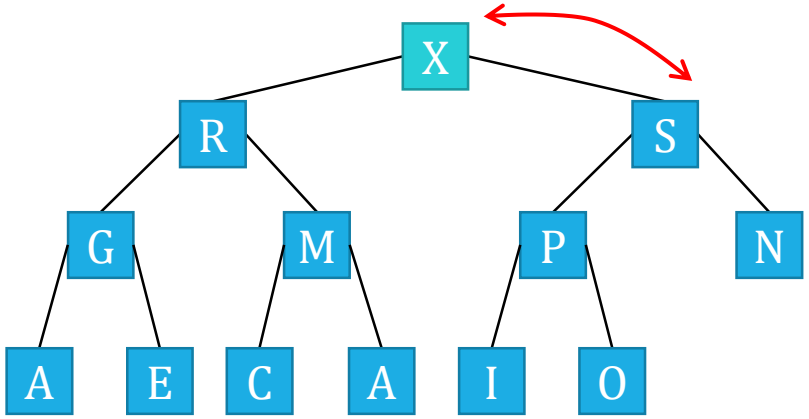
Kupac – elem beszúrása

- 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 $\mathcal{O}(\log n)$



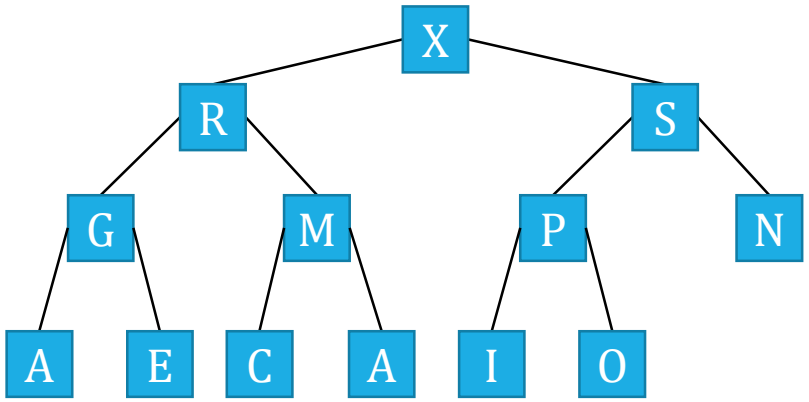
Kupac – elem beszúrása

- 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 $\mathcal{O}(\log n)$



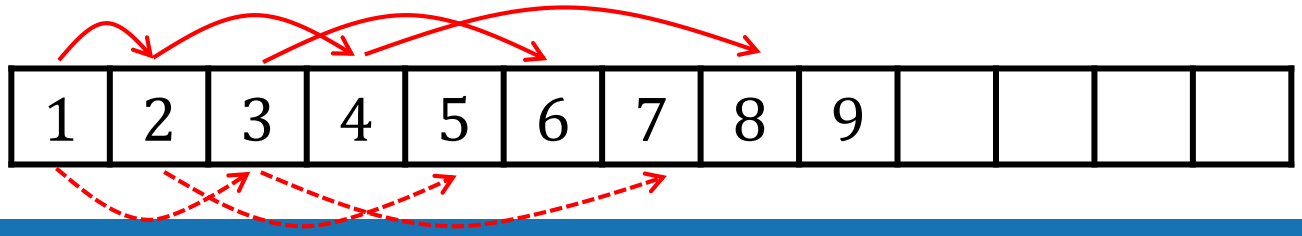
Kupac – elem beszúrása

- 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 $\mathcal{O}(\log n)$



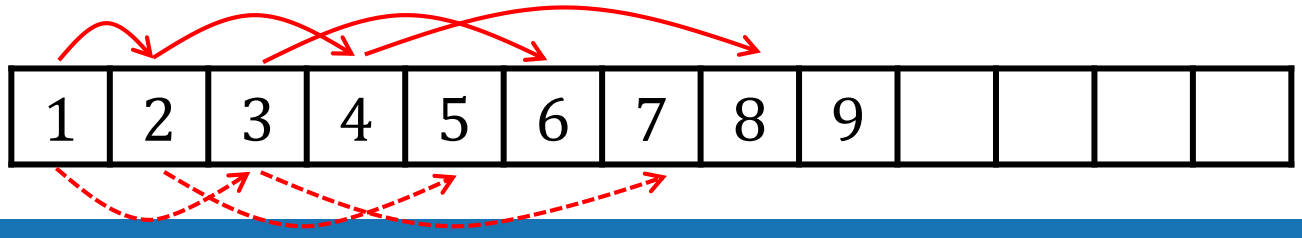
Kupacok – reprezentáció

- 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



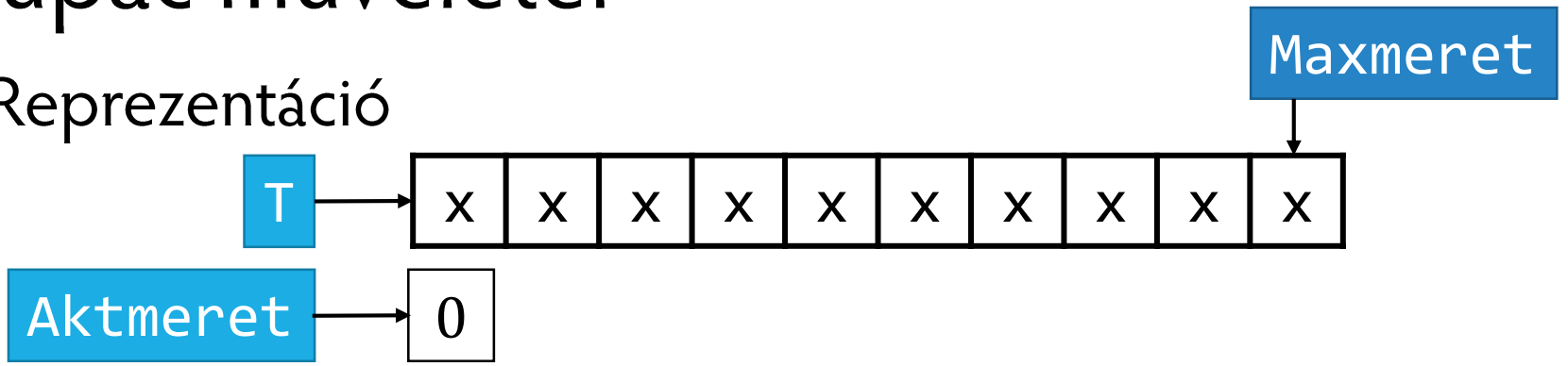
Kupac – hatékonyság

- 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$\neq 0$	
return $T[1]$	Üres!

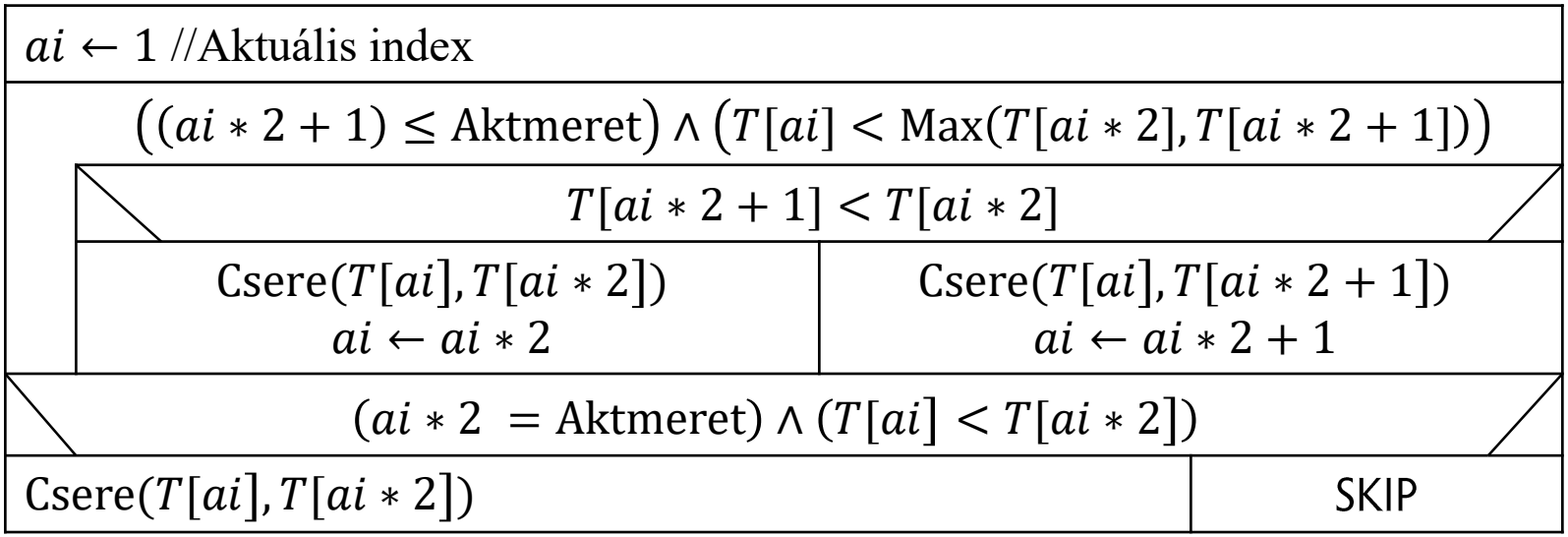
Kupac műveletei

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

Aktmeret≠ 0	
Maxelem← $T[1]$ $T[1] \leftarrow T[\text{Aktmeret}]$ $\text{Aktmeret} \leftarrow \text{Aktmeret} - 1$ 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

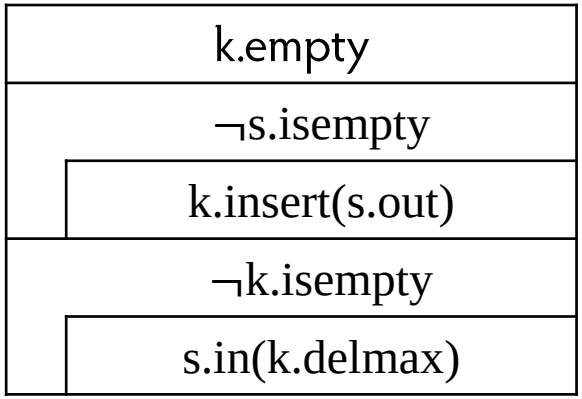
- `Empty` (az üres prioritásos sor konstruktor – létrehozás)
  - `Empty` (üres kupac)
- `isempty` (üres a prioritásos sor?)
  - `IsEmpty` (üres a kupac?)
- `insert` (elem betétele a prioritásos sorba)
  - `Insert` (elem betétele a kupacba)
- `delmax` (maximális elem kivétele a prioritásos sorból)
  -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

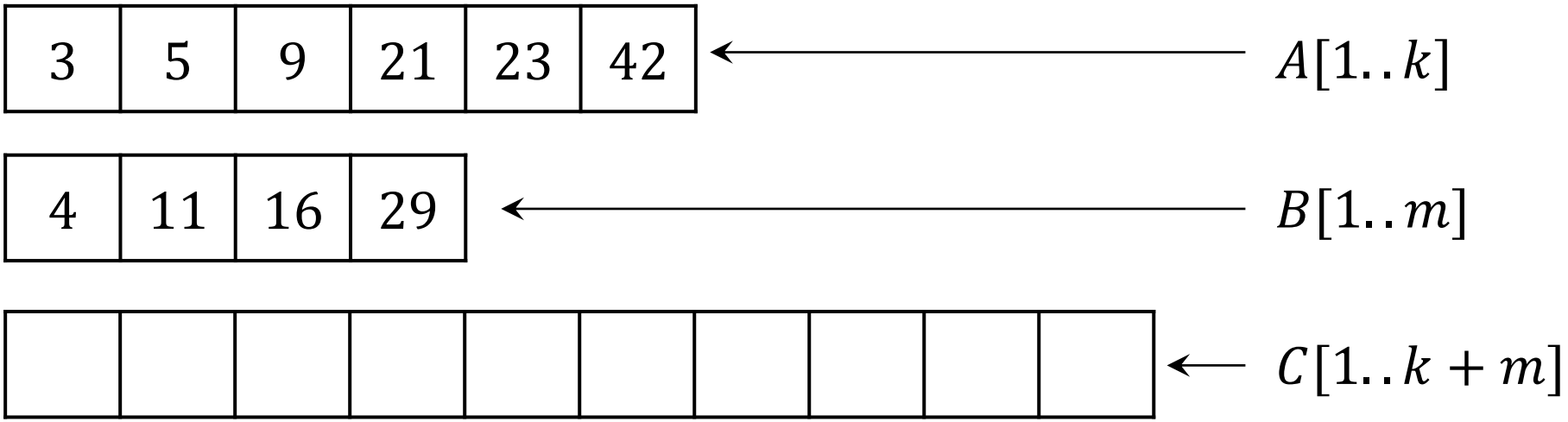
Quicksort ⇔ Heap Sort

- 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

Összefuttatásos rendezés

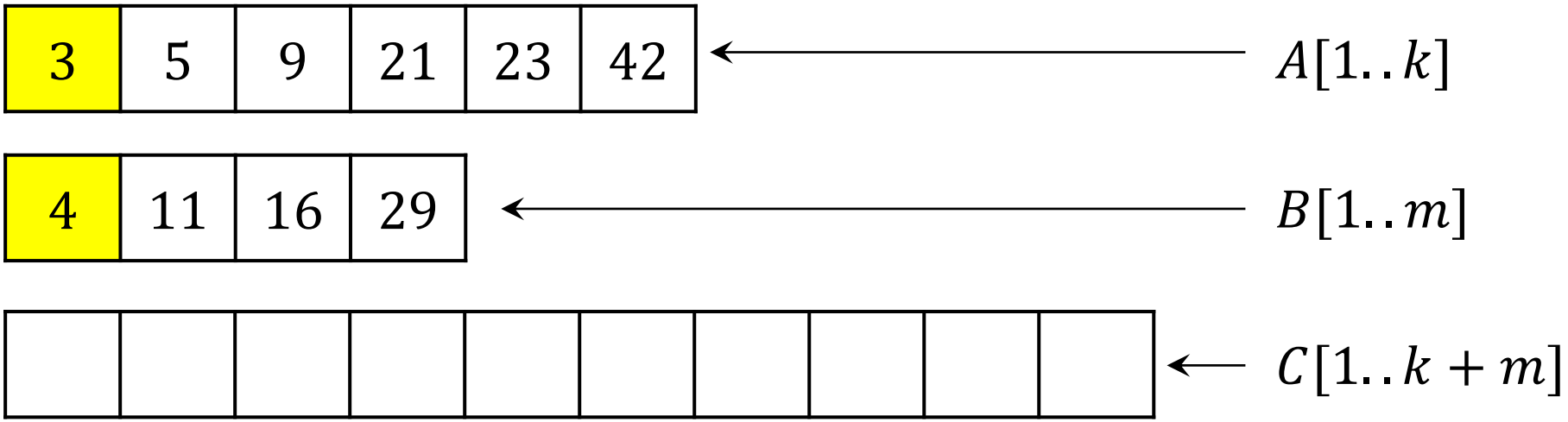
Összefuttatásos rendezés

- Alap: két rendezett sorozat összefuttatása
- Legyen $A[1..k]$ és $B[1..m]$ két rendezett vektor, ekkor:



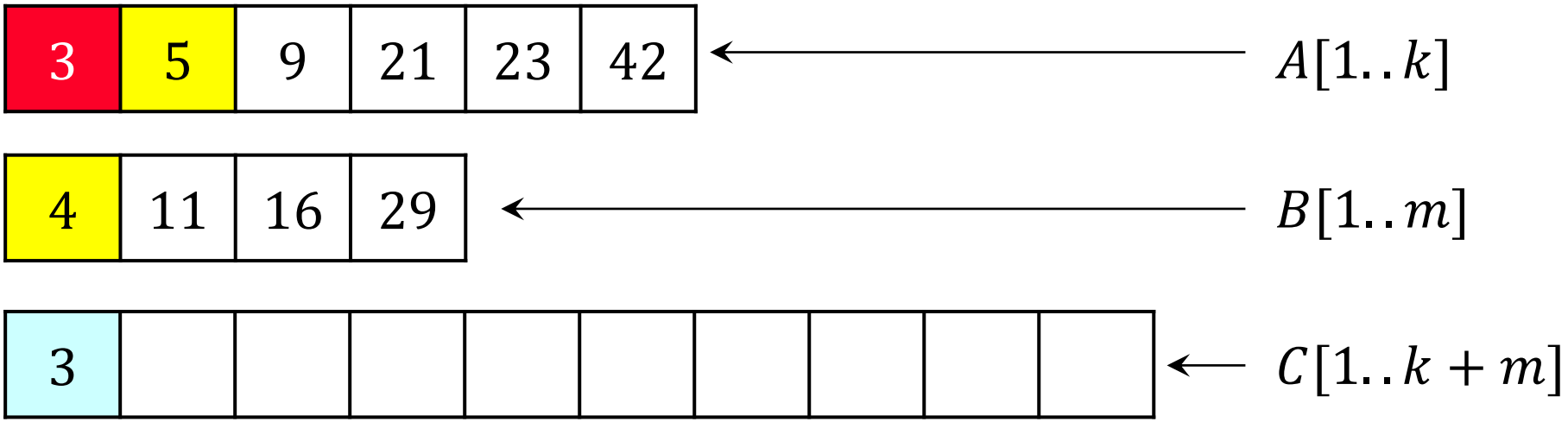
Összefuttatásos rendezés

- Alap: két rendezett sorozat összefuttatása
- Legyen $A[1..k]$ és $B[1..m]$ két rendezett vektor, ekkor:



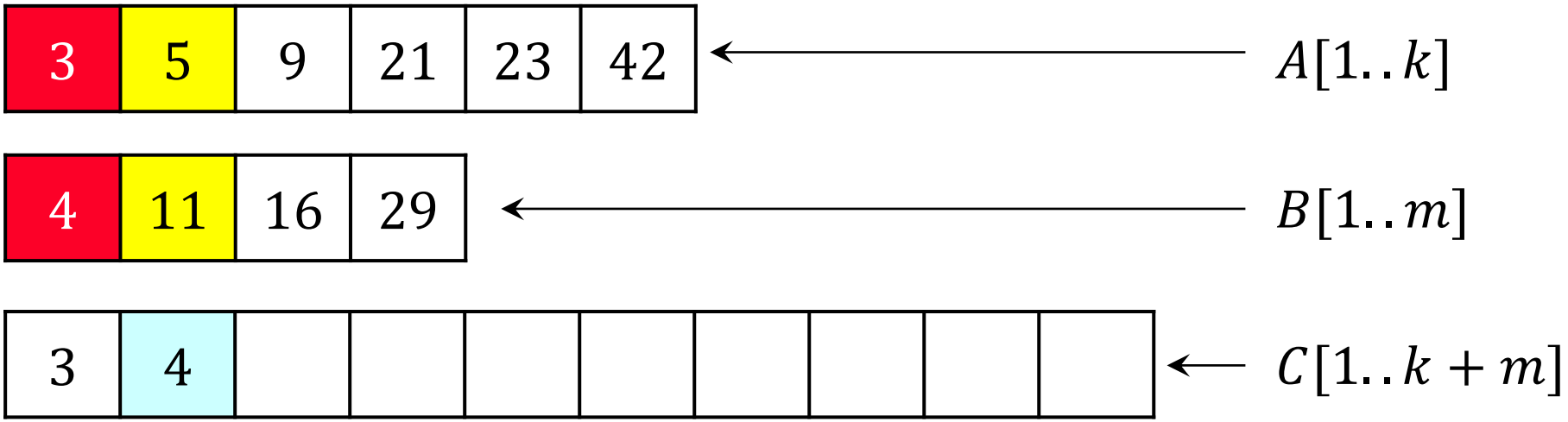
Összefuttatásos rendezés

- Alap: két rendezett sorozat összefuttatása
- Legyen $A[1..k]$ és $B[1..m]$ két rendezett vektor, ekkor:



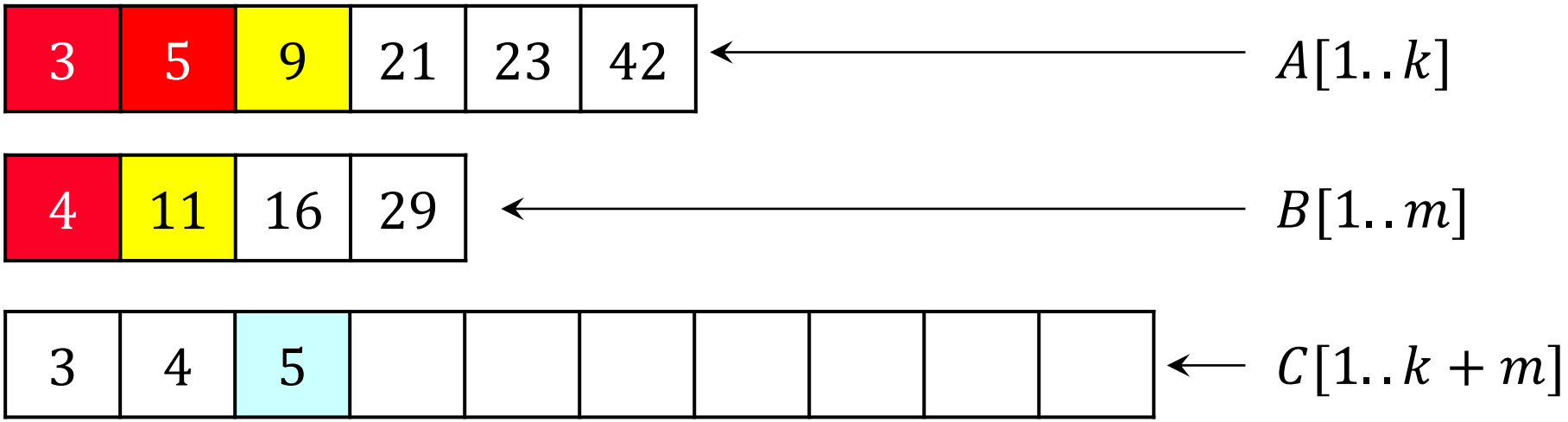
Összefuttatásos rendezés

- Alap: két rendezett sorozat összefuttatása
- Legyen $A[1..k]$ és $B[1..m]$ két rendezett vektor, ekkor:



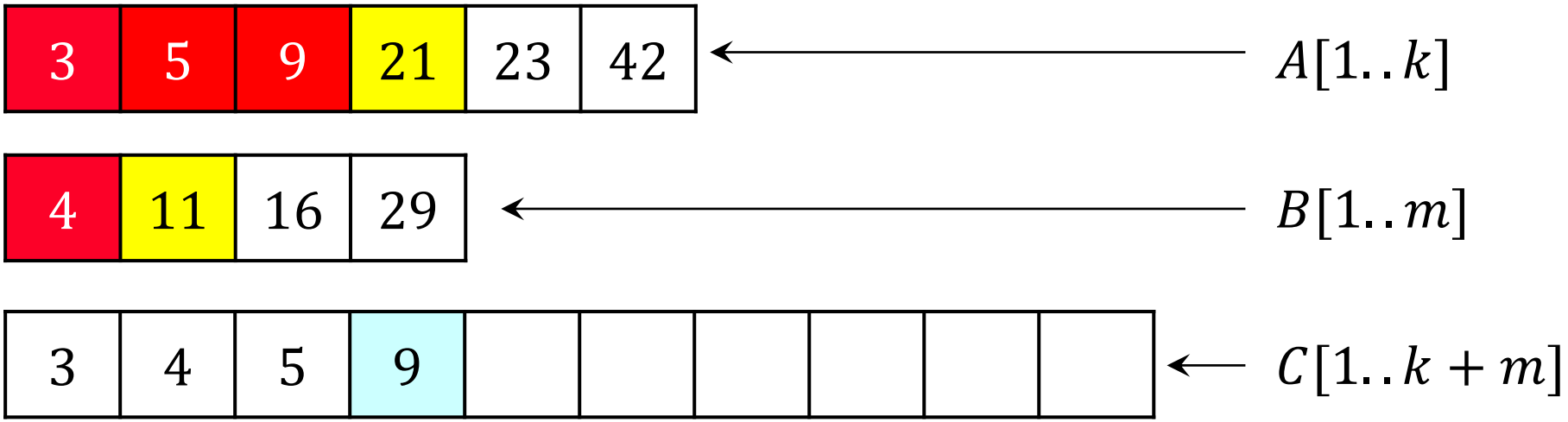
Összefuttatásos rendezés

- Alap: két rendezett sorozat összefuttatása
- Legyen $A[1..k]$ és $B[1..m]$ két rendezett vektor, ekkor:



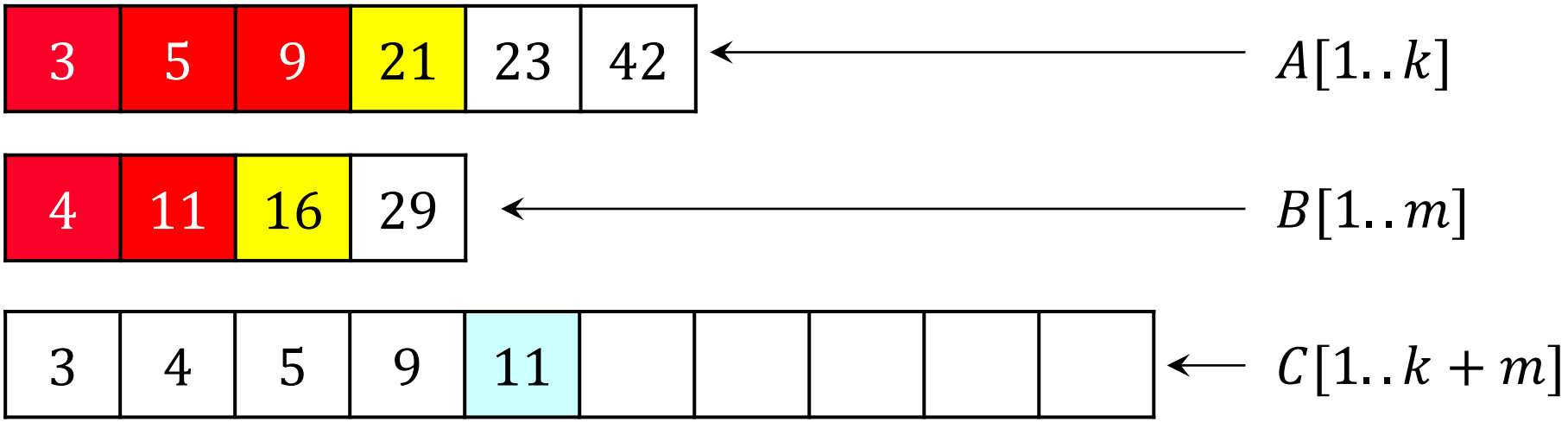
Összefuttatásos rendezés

- Alap: két rendezett sorozat összefuttatása
- Legyen $A[1..k]$ és $B[1..m]$ két rendezett vektor, ekkor:



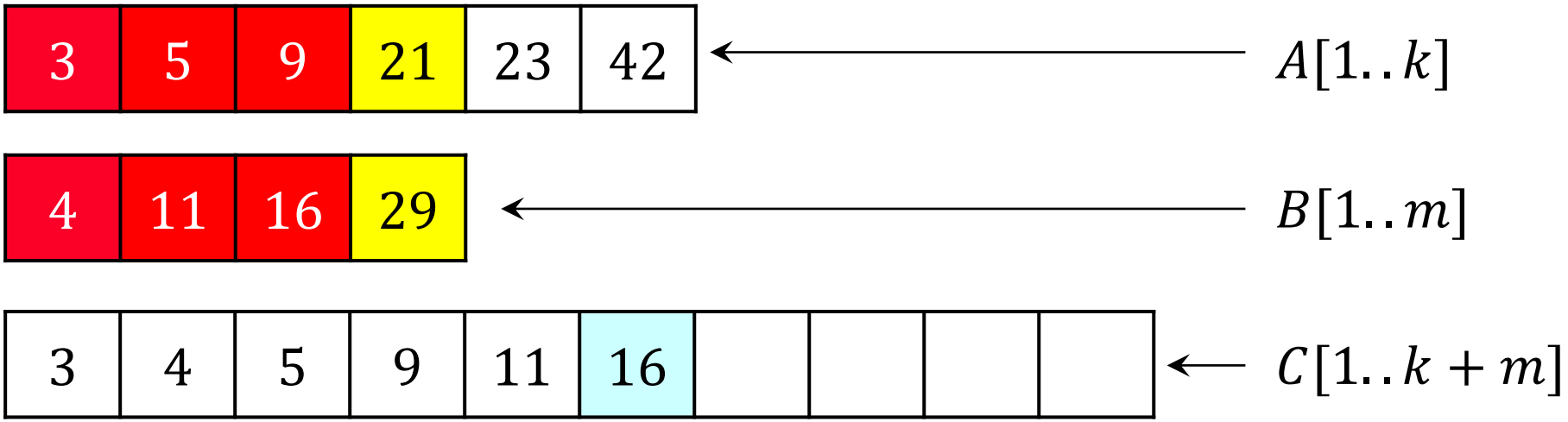
Összefuttatásos rendezés

- Alap: két rendezett sorozat összefuttatása
- Legyen $A[1..k]$ és $B[1..m]$ két rendezett vektor, ekkor:



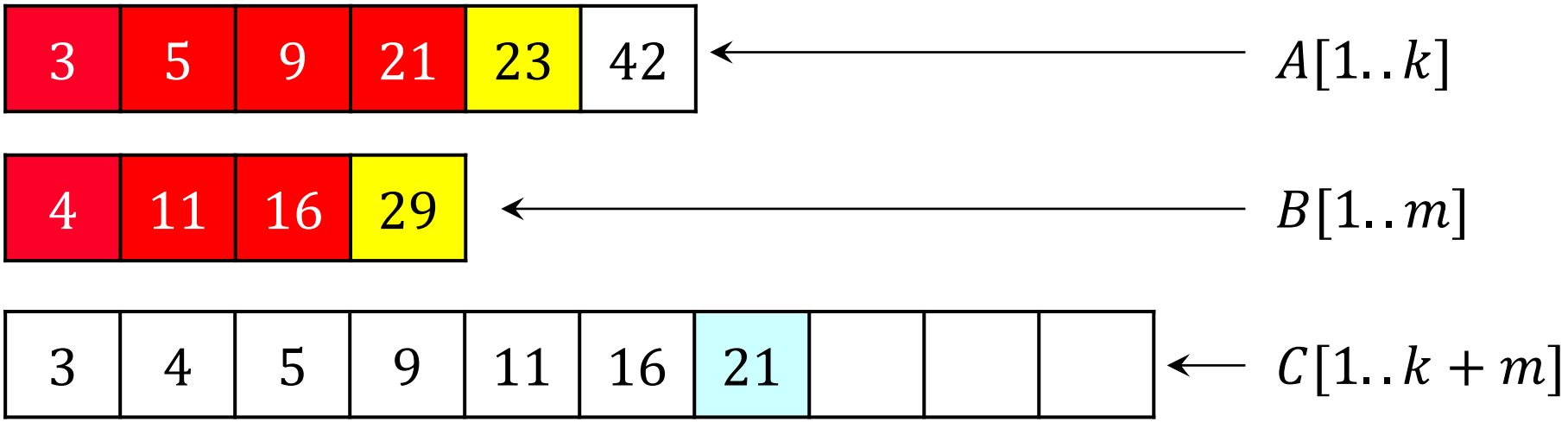
Összefuttatásos rendezés

- Alap: két rendezett sorozat összefuttatása
- Legyen $A[1..k]$ és $B[1..m]$ két rendezett vektor, ekkor:



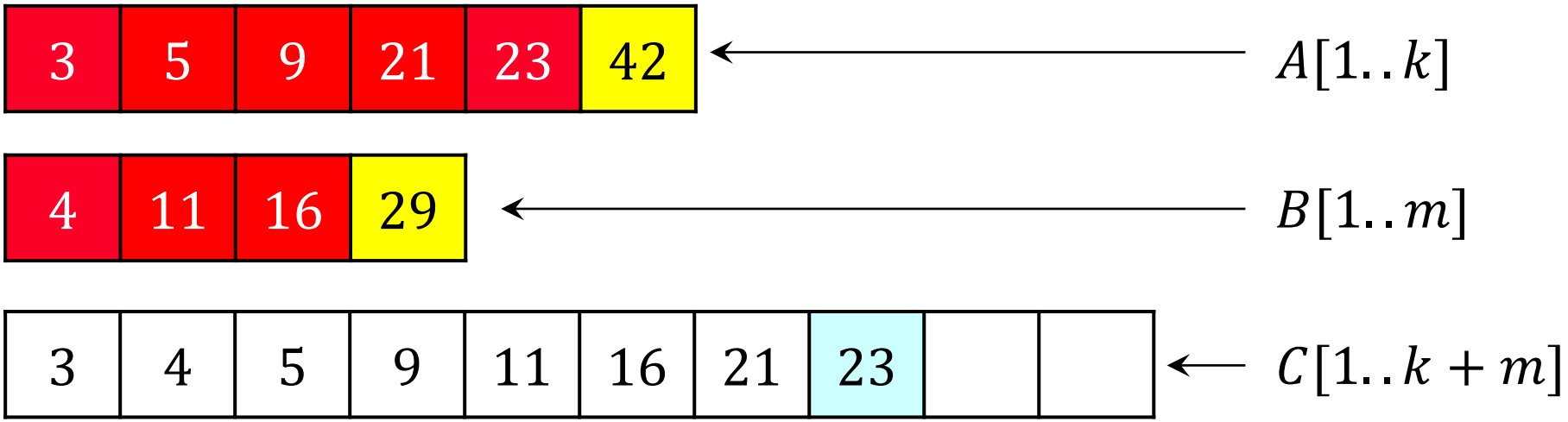
Összefuttatásos rendezés

- Alap: két rendezett sorozat összefuttatása
- Legyen $A[1..k]$ és $B[1..m]$ két rendezett vektor, ekkor:



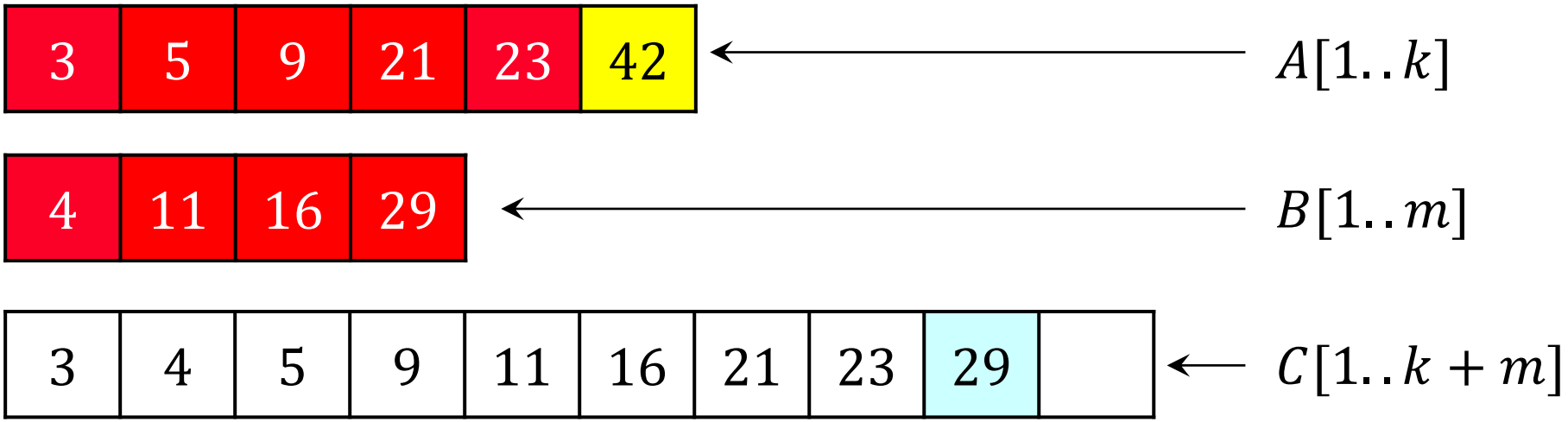
Összefuttatásos rendezés

- Alap: két rendezett sorozat összefuttatása
- Legyen $A[1..k]$ és $B[1..m]$ két rendezett vektor, ekkor:



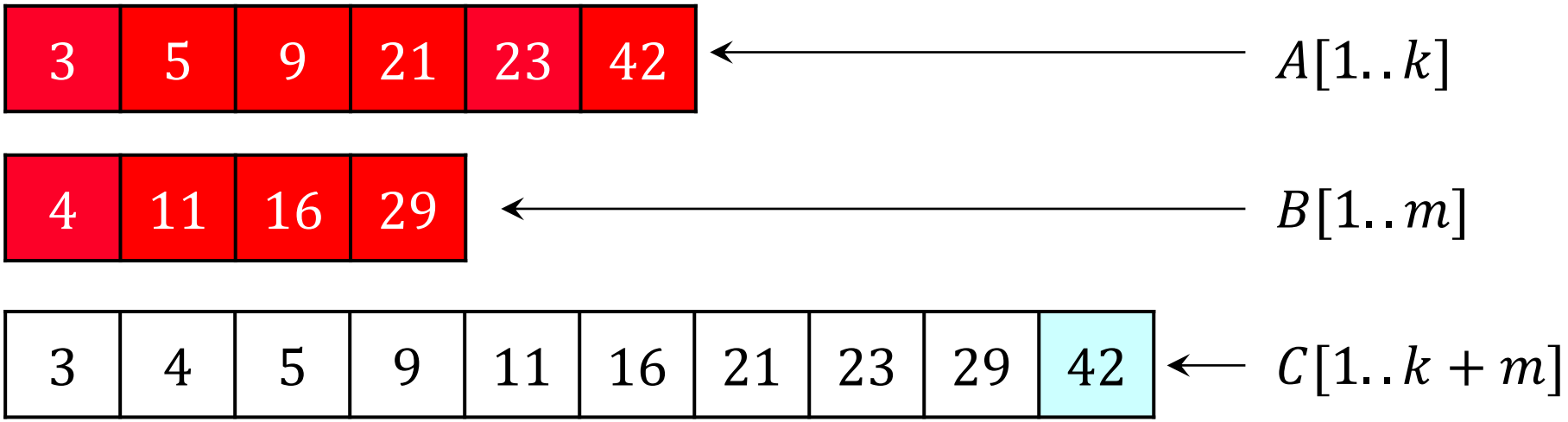
Összefuttatásos rendezés

- Alap: két rendezett sorozat összefuttatása
- Legyen $A[1..k]$ és $B[1..m]$ két rendezett vektor, ekkor:



Összefuttatásos rendezés

- Alap: két rendezett sorozat összefuttatása
- Legyen $A[1..k]$ és $B[1..m]$ két rendezett vektor, ekkor:



Összefuttatás algoritmusa

- Összefuttat(A, B, C)

$ai \leftarrow 1, bi \leftarrow 1, ci \leftarrow 1$		
$ai \leq k \wedge bi \leq m$		
$A[ai] < B[bi]$	$A[ai] = B[bi]$	$A[ai] > B[bi]$
$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$	$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$ $ci \leftarrow ci + 1$ $C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$	$C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$
$ci \leftarrow ci + 1$		
C tömbbe tegyük a maradékot		

Összefuttatás algoritmusa

- Összefuttat(A, B, C)

$ai \leftarrow 1, bi \leftarrow 1, ci \leftarrow 1$			
$ai \leq k \wedge bi \leq m$			
	$A[ai] < B[bi]$	$A[ai] = B[bi]$	$A[ai] > B[bi]$
	$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$	$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$ $ci \leftarrow ci + 1$ $C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$	$C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$
	$ci \leftarrow ci + 1$		
C tömbbe tegyük a maradékot			

Összefuttatás algoritmusa

- Összefuttat(A, B, C)

$ai \leftarrow 1, bi \leftarrow 1, ci \leftarrow 1$			
$ai \leq k \wedge bi \leq m$			
	$A[ai] < B[bi]$	$A[ai] = B[bi]$	$A[ai] > B[bi]$
	$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$	$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$ $ci \leftarrow ci + 1$ $C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$	$C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$
	$ci \leftarrow ci + 1$		
C tömbbe tegyük a maradékot			

Összefuttatás algoritmusa

- Összefuttat(A, B, C)

$ai \leftarrow 1, bi \leftarrow 1, ci \leftarrow 1$			
$ai \leq k \wedge bi \leq m$			
	$A[ai] < B[bi]$	$A[ai] = B[bi]$	$A[ai] > B[bi]$
	$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$	$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$ $ci \leftarrow ci + 1$ $C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$	$C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$
	$ci \leftarrow ci + 1$		
C tömbbe tegyük a maradékot			

Összefuttatás algoritmusa

- Összefuttat(A, B, C)

$ai \leftarrow 1, bi \leftarrow 1, ci \leftarrow 1$		
$ai \leq k \wedge bi \leq m$		
$A[ai] < B[bi]$	$A[ai] = B[bi]$	$A[ai] > B[bi]$
$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$	$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$ $ci \leftarrow ci + 1$ $C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$	$C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$
$ci \leftarrow ci + 1$		
C tömbbe tegyük a maradékot		

Összefuttatás algoritmusa

- Összefuttat(A, B, C)

$ai \leftarrow 1, bi \leftarrow 1, ci \leftarrow 1$		
$ai \leq k \wedge bi \leq m$		
$A[ai] < B[bi]$	$A[ai] = B[bi]$	$A[ai] > B[bi]$
$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$	$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$ $ci \leftarrow ci + 1$ $C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$	$C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$
$ci \leftarrow ci + 1$		
C tömbbe tegyük a maradékot		

Összefuttatás algoritmusa

- Összefuttat(A, B, C)

$ai \leftarrow 1, bi \leftarrow 1, ci \leftarrow 1$		
$ai \leq k \wedge bi \leq m$		
$A[ai] < B[bi]$	$A[ai] = B[bi]$	$A[ai] > B[bi]$
$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$	$C[ci] \leftarrow A[Ai]$ $ai \leftarrow ai + 1$ $ci \leftarrow ci + 1$ $C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$	$C[ci] \leftarrow B[Bi]$ $bi \leftarrow bi + 1$
$ci \leftarrow ci + 1$		
C tömbbe tegyük a maradékot		

- Ez egy külön ciklus

Összefuttatásos rendezés

- Összefuttatásnál
  - összehasonlítások száma: $k + m - 1$
- RENDEZÉS:
  - Az üres és az egyelemű sorozat biztosan rendezett.
  - $S[1..n]$ $(n > 1)$ tömböt ha rendezni szeretnénk:
    - szétvágjuk két részre (S_1, S_2) – elfelezzük
    - ezeket külön-külön rendezzük
    - összefuttatjuk
  - MergeSort-nak is hívják

Összefuttatásos rendezés

- MergeSort(S)

Length(S) > 1	
Szétvág(S, S_1, S_2)	SKIP
MergeSort(S_1) MergeSort(S_2)	
Összefuttat(S_1, S_2, S)	

Összefuttatásos rendezés – tömbre

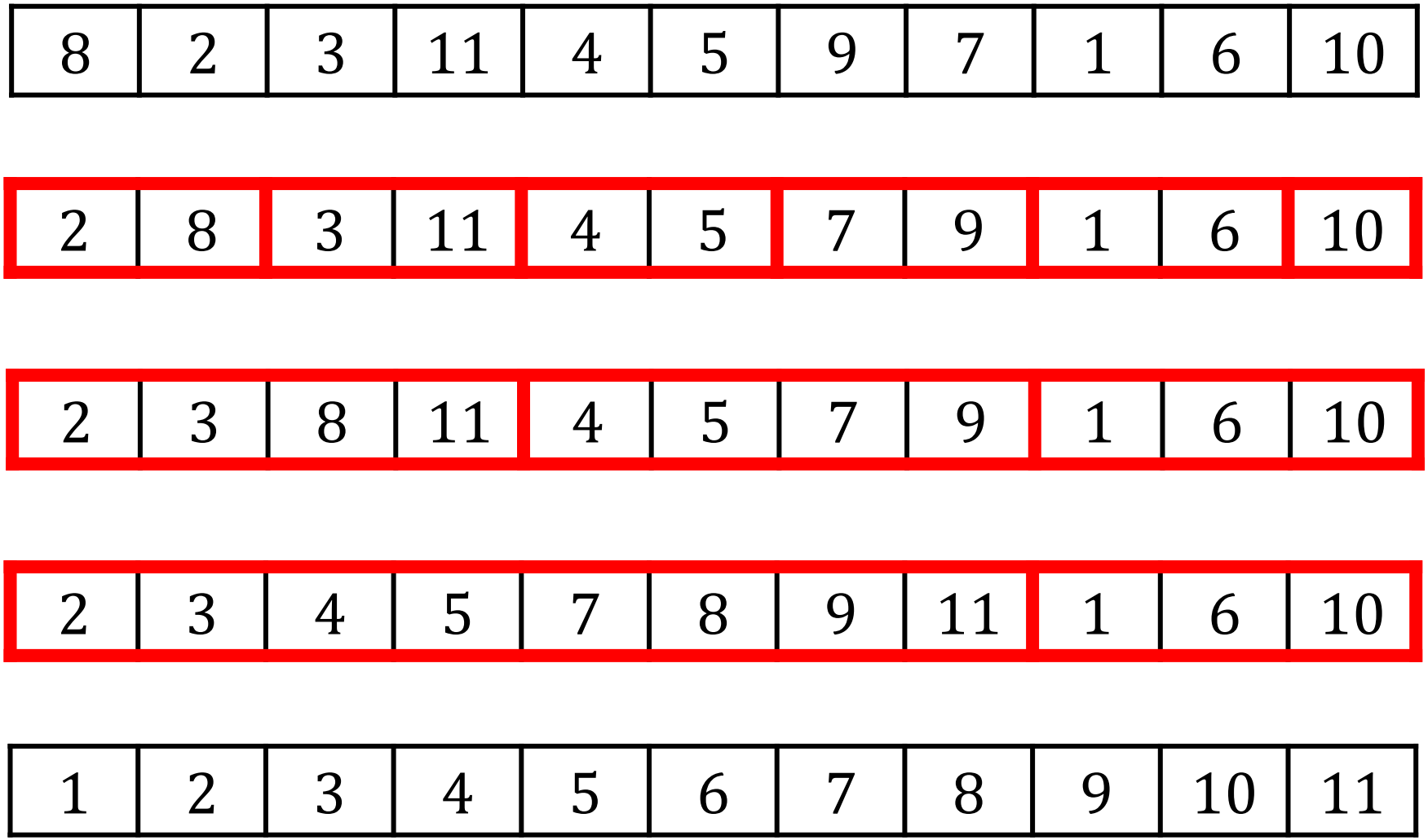
- MergeSort(A, k, v)

$k < v$	
$h \leftarrow \frac{k+v}{2}$ Szétvág(S, S_1, S_2, h)	SKIP
MergeSort(A, k, h) MergeSort$(A, h+1, v)$	
Összefuttat$(A[k, h], A[h+1, v], A[k, v])$	

- A külső hívás
  - MergeSort$(A, 1, n)$
- Az összefuttatásnál szükséges egy segédtömb használata

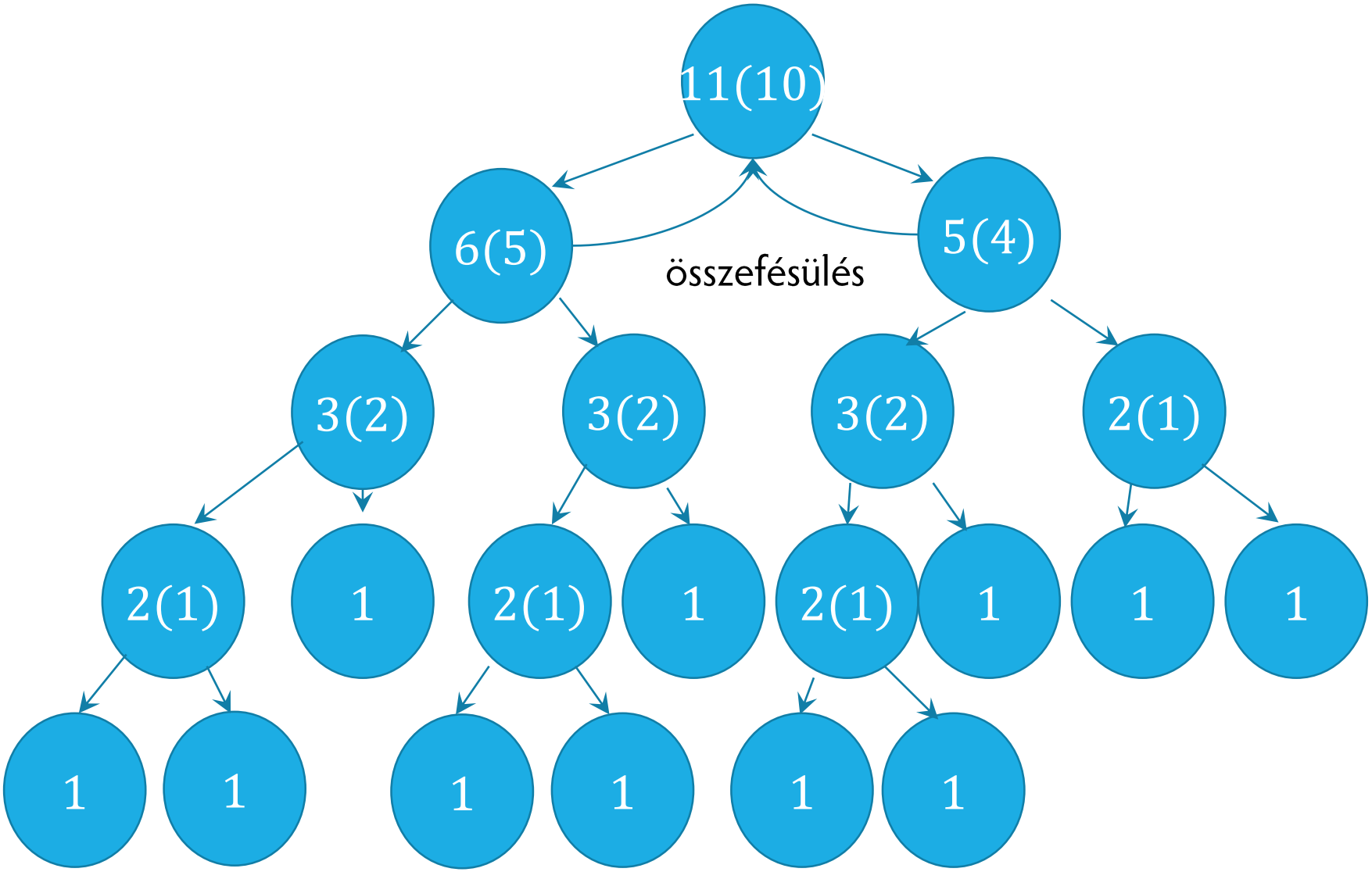
Összefuttatásos rendezés

- Tömbökre szokás egy másik, iteratív stratégiát is követni:
  - Minden lépésben végigmegyünk rajta egyre növekvő $h = 1, 2, 2^2, 2^3, \ldots\ 2^{\lceil \log_2 n \rceil - 1}$ lépésközzel, és az ilyen hosszú szomszédos tömbrészleteket fésüljük össze, egy segédtömböt használva
- Példa …


8 2 3 11 4 5 9 7 1 6 10
2 8 3 11 4 5 7 9 1 6 10
2 3 8 11 4 5 7 9 1 6 10
2 3 4 5 7 8 9 11 1 6 10
1 2 3 4 5 6 7 8 9 10 11

Összefuttatásos rendezés

- Hatékonyságelemzés:
  - Az összehasonlítások számát becsüljük a legrosszabb esetben
    - Az összefuttatás k és m hosszú sorozatok összefésülésekor $k + m - 1$ összehasonlítást végez
      - $\text{MÖö}_{\text{sszefuttat}}(k, m) = k + m - 1$
    - ha n a két sorozat együttes hossza
      - $\text{MÖö}_{\text{sszefuttat}}(n) = n - 1$
    - A MergeSort eljárásnál a közel egyenlő szétvágást alapul véve: milyen hosszú részekre vágja az eljárás a (rész)sorozatokat a rekurzív hívások szintjein.



$n = 11$-re

( )-ben:öh-k száma $\quad MÖ = 10 + (5 + 4) + (2 + 2 + 2 + 1) + (1 + 1 + 1) = 29$

Összefuttatásos rendezés

- Az egyenlő szétvágások bináris fája majdnem teljes.
- A fa magassága: $4 = \lceil \log_2 11 \rceil$
  - Általában is: $h = \lceil \log_2 n \rceil$
- A fa leveleihez nem tartozik összehasonlítás, így éppen h szinten kell összegezni az összehasonlítások számát
  - Ezek összege szintenként egyre csökken
    - mindenütt $\leq n - 1$
- Így $\text{MÖ}_{\text{MS}}(n) \leq (n-1) * \lceil \log_2 n \rceil = \Theta(n * \log_2 n)$

Összehasonlításos rendezők elméleti korlátja

Rendezések

- Összehasonlításos rendezőknél mit tudunk mondani a rendezés időigényére? Tudunk-e alsó becslést adni?
  - Ugyanaz, mintha barkóbázva kellene kitalálni, hogy az elemek melyik sorrendje (permutációja) az igazi sorrend
  - Kezdetben $n!$ lehetséges sorrend jön szóba.
  - Két elemet összehasonlítva, a válasz két részre osztja a sorrendeket
    - Ha például azt kapjuk, hogy $x < y$, akkor az olyan sorrendek, amikben x hátrébb van y-nál, már nem jönnek szóba
    - Ha a válasz mindig olyan, hogy minél több sorrend maradjon meg, akkor k kérdés után még szóba jövő sorrendek száma

$$\frac{n!}{2^k}$$

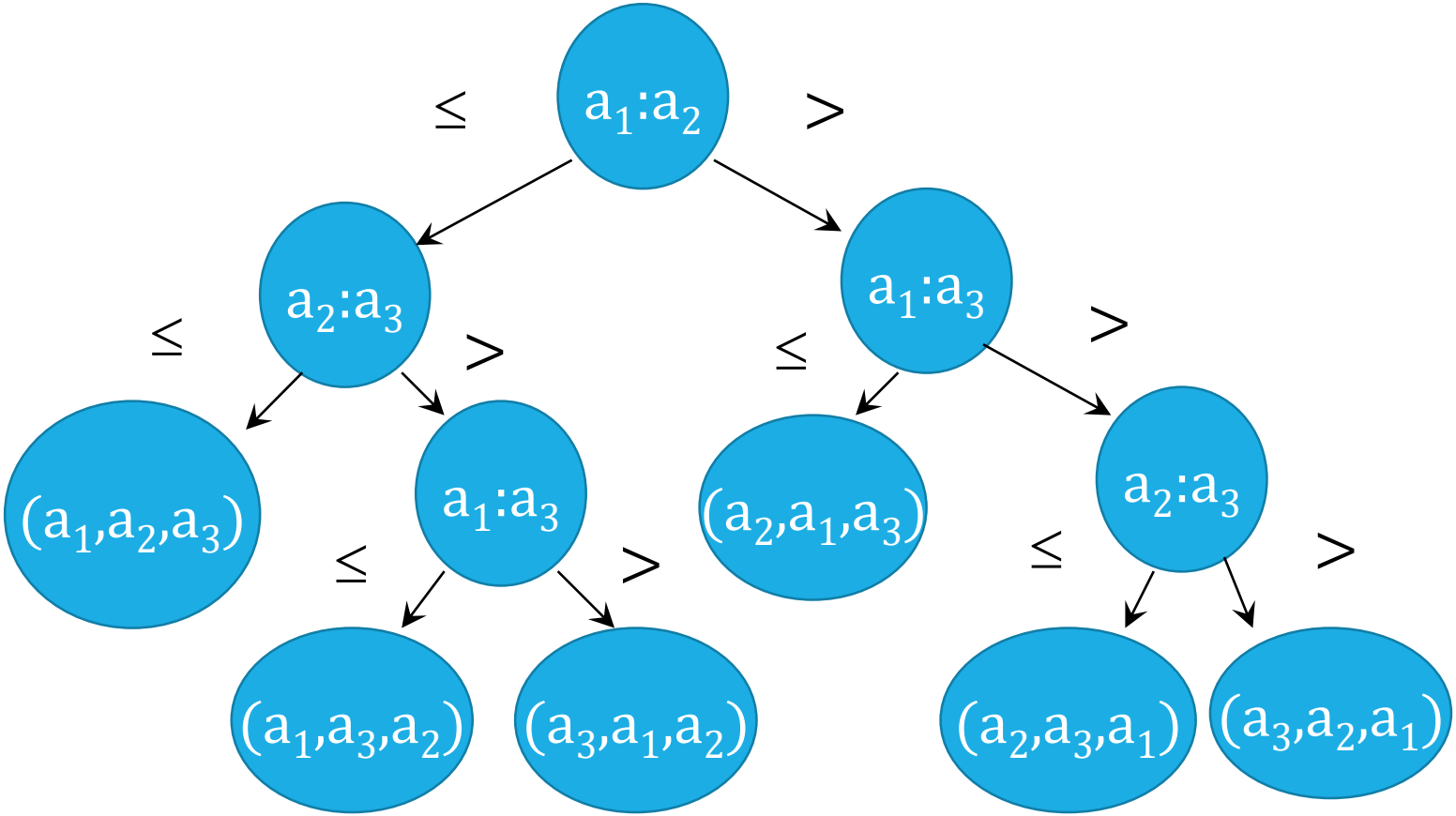
Rendezések

- Döntési fa modell
  - Az összehasonlítások tekinthetők döntési fáknak, amik a rendezés során történt összehasonlításokat ábrázolják
  - Tegyük fel, hogy minden elem különböző
  - Egy belső csúcsot egy $a_i : a_j$ párral jelölhetünk, ahol
$$1 \leq i, j \leq n$$
  - A levelek egy-egy permutációnak feleltethetők meg

Rendezések

- Baloldali részfa: az $a_i \leq a_j$ után szükséges összehasonlítások
- Jobboldali részfa: az $a_i > a_j$ után szükséges összehasonlítások
- Levél: a rendezett sorrend

Rendezések


a1:a2
≤
>
a2:a3
a1:a3
≤
>
≤
>
(a1,a2,a3)
a1:a3
(a2,a1,a3)
a2:a3
≤
>
≤
>
(a1,a3,a2)
(a3,a1,a2)
(a2,a3,a1)
(a3,a2,a1)

Alsó korlát a legrosszabb esetben

- Tétel: Bármely n elemet rendező döntési fa magassága
$$\Omega(n * \log_2 n)$$
  - Bizonyítás: Vegyük az n elemet rendező döntési fát, jelöljük ennek magasságát h-val.
  - A fának $n!$ levele van – ezek a permutációk.
  - A h mélységű bináris fa leveleinek száma $\leq 2^h$, ezért
$$n! \leq 2^h$$
  - $h \geq \log_2(n!)$
    - felhasználva, hogy a log függvény monoton növő

Alsó korlát a legrosszabb esetben

- Ismert a Stirling formula (~1730!):

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

  - ahol e a természetes logaritmus alapja
- Ennek alapján

$$n! > \left(\frac{n}{e}\right)^n$$

- Behelyettesítve:
- $h \geq \log_2\left(\left(\frac{n}{e}\right)^n\right) = n * \log_2 n - n * \log_2 e = \Omega(n * \log_2 n)$

Alsó korlát a legrosszabb esetben

- Következmény 1.
  - A kupacrendezés és az összefésüléses rendezés aszimptotikusan optimális összehasonlító rendezések.
- Bizonyítás
  - A kupacrendezés és az összefésüléses rendezés futási idejének felső korlátja $\Theta(n * \log_2 n)$ a legrosszabb esetre megadott $\Omega(n * \log_2 n)$ felső korláttal.
- Következmény 2.
  - Nincs lineáris idejű összehasonlításos rendezés
  - ☹

Rendező algoritmusok III.

Következő alkalommal