Algoritmusok és adatszerkezetek II.

Előadás

Bemutató

Tartalom

Algoritmusok és adatszerkezetek II.
1. Előadás

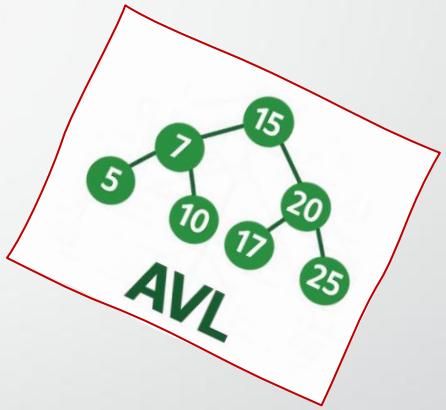
AVL fa I.

fogalma, miértje, láncolt és szöveges ábrázolása, kulcs beszúrása, legkisebb kulcsú csúcs kivétele.

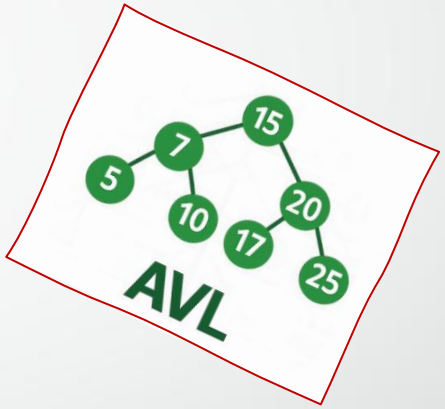
Tartalom

Kiegyensúlyozott keresőfák (balanced search trees)

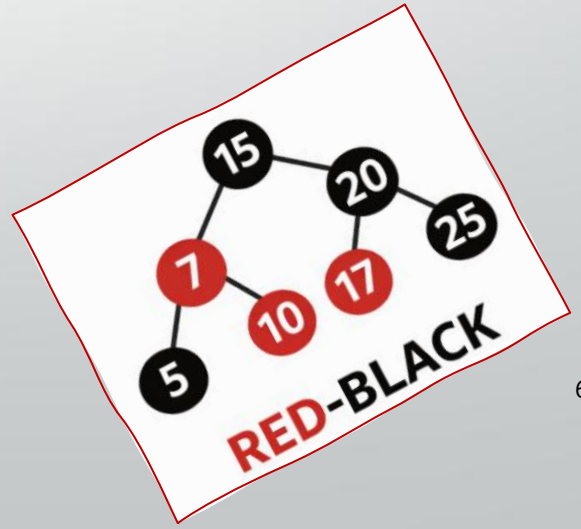
- Bináris keresőfák (BST)
  - Futási ideje ~ fa magasságával
    - Átlagos esetben megfelelő hatékonyságúak ($\Theta(\log n)$)
  - Legrosszabb esetben a fa magassága n-1
    - Szótárműveletek nem elég hatékonyak
- Kiegyensúlyozott keresőfák
  - A fa magassága minden esetben $\Theta(\log n)$
  - Garantálja a szótárműveletek megfelelő hatékonyságát



AVL fa

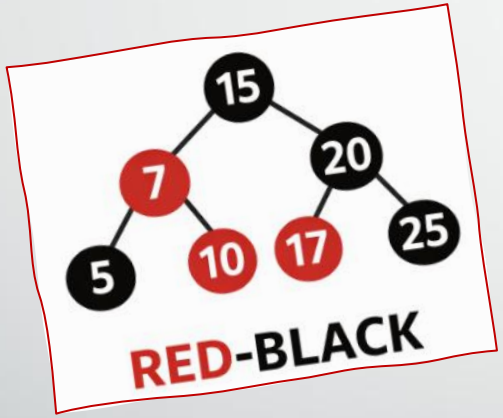


- **Az AVL fa előnyösebb a piros-fekete fával szemben**
  - ha lényegesen több a keresés, mint a módosítás,
  - ➢ mert az AVL erősebben kiegyensúlyozott

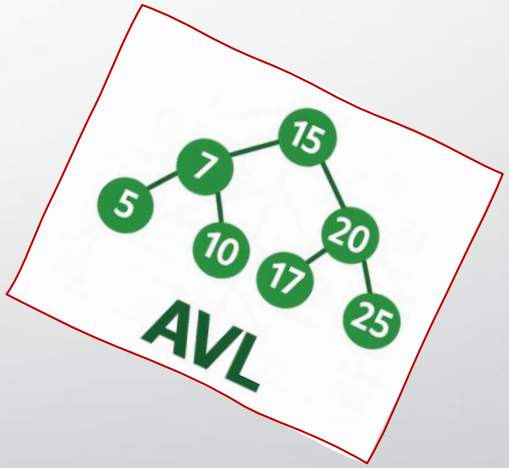


Kiegyensúlyozott keresőfák típusai

Piros-fekete fák



AVL fák



- **Az AVL fa előnyösebb a piros-fekete fával szemben**
  - ha lényegesen több a keresés, mint a módosítás,
  - ➢ mert az AVL erősebben kiegyensúlyozott

AVL fa fogalma

(Gregory Adelszon-Velszkij és Evgenii Landisz, 1963)

- **Definíció:** Az AVL fák magasság szerint kiegyensúlyozott bináris keresőfák.

- Egy bináris *fa magasság szerint kiegyensúlyozott* (height-balanced BST)
  - ha minden csúcsa kiegyensúlyozott.

- kiegyensúlyozott ~ magasság szerint kiegyensúlyozott

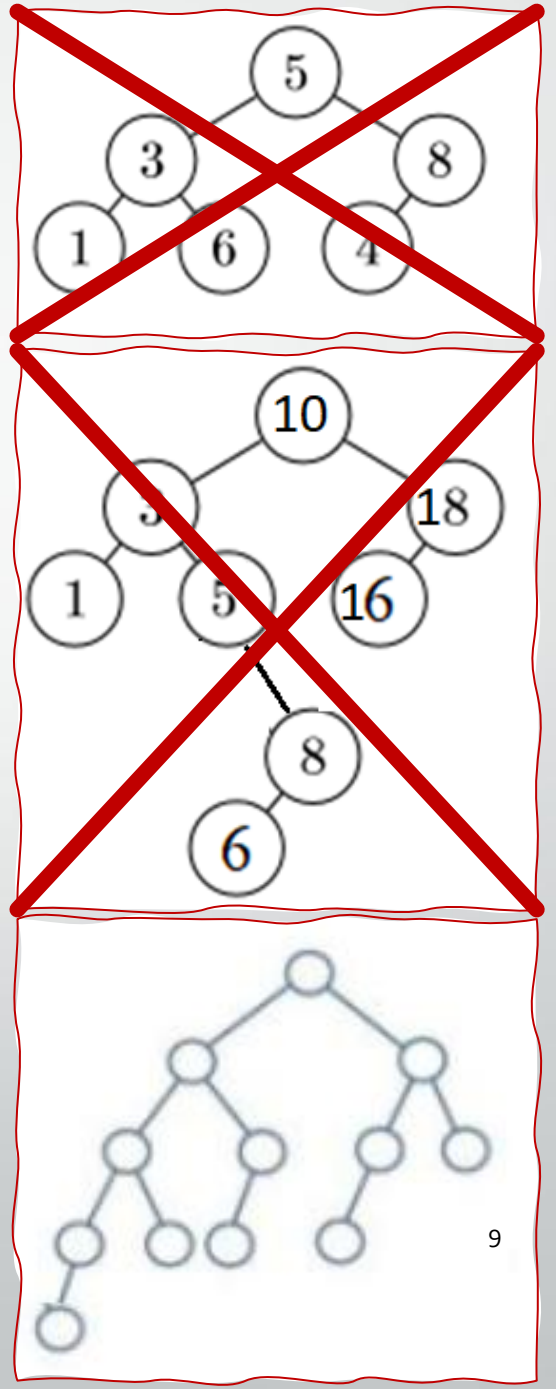
AVL fa fogalma

(Adelszon-Velszkij és Landisz, 1962)

- Egy bináris fa egy (**p*) csúcsa *kiegyensúlyozott (*balanced node*):*
  - ha a csúcs (*p -> b*) egyensúlyára (balance) $|p \text{->} b| \leq 1$.

- **Definíció:** A (**p*) csúcs egyensúlya (the (∗p) node's balance):

  $p \text{->} b = h(p \text{->} right) - h(p \text{->} left)$



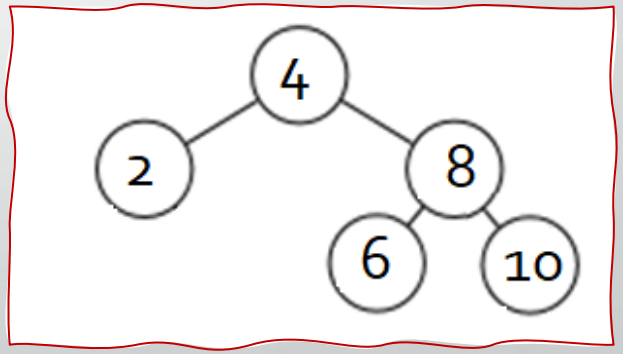
Az AVL fák láncolt reprezentálása

Node
+ $key : \mathcal{T}$ // $\mathcal{T}$ is some known type + $b : -1..1$ // the balance of the node + $left, right$: Node*
+ Node() { $left := right := \oslash$; $b := 0$ } // create a tree of a single node + Node(x:$\mathcal{T}$) { $left := right := \oslash$; $b := 0$; $key := x$ }

- A csúcsokban a *b* egyensúly attribútumot expliciten tároljuk
- //Más lehetőségek:
  - a két részfa magasságainak tárolása
  - csak az aktuális részfa magasságának tárolása

Az AVL fák szöveges reprezentálása

- Keresőfák
  - (bal_részfa gyökér jobb_részfa) jelölés
  - az üres részfák elhagyása
  - a könnyebb olvashatóság kedvéért [ ] és { } zárójelek alkalmazása
  - ➢ Az így ábrázolt fák inorder bejárása a zárójelek elhagyásával adódik
- AVL fák
  - A belső csúcsok egyensúlyainak jelölése *kb* formában
    - *k:* a csúcsot azonosító kulcs
    - *b:* a csúcs egyensúlya: **0:◦, 1:+, 2:++, -1:, -2**
  - a leveleknél nem jelöljük az egyensúlyt
    - a levélcsúcsok egyensúlya mindig nulla



{ [2] 4 [(6) 8 (10)] }

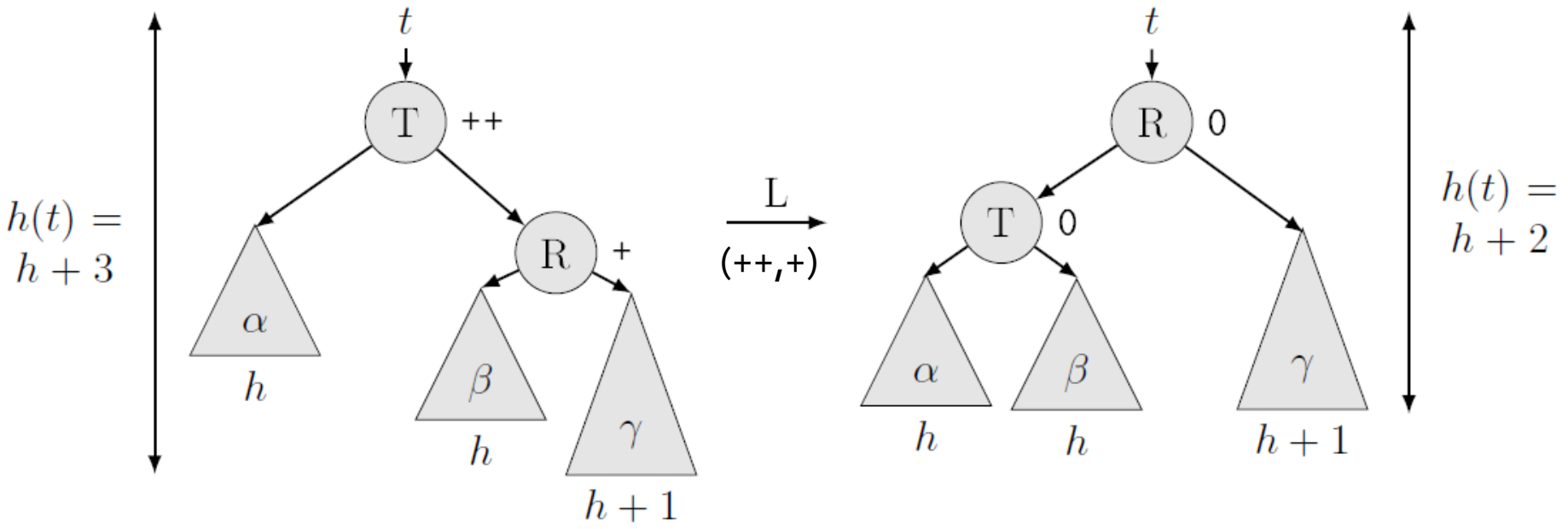
{ [2] 4+ [(6) 8◦ (10)] }

AVL fa kiegyensúlyozó forgatások (balancing rotations)

- Jelölések
  - görög kisbetűk: részfák (ezek mindig AVL fák)
  - latin nagybetűk: csúcsok kulcsait (egyensúlyok nélkül)
- Forgatások:
  - Balra forgatás (Left rotation): [α T (β R γ)] -> [(α T β) R γ]
  - Jobbra forgatás (Right rotation): [(α L β) T γ] -> [α L (β T γ)]
- A bináris keresőfa tulajdonságot a kiegyensúlyozások is megtartják
  - a fa inorder bejárását egyik forgatás sem változtatja -> a bináris keresőfa tulajdonságot is megtartják
  - a kiegyensúlyozatlan részfák kiegyensúlyozása minden esetben egy vagy két forgatásból áll

AVL fa forgatások: (++,+) forgatás

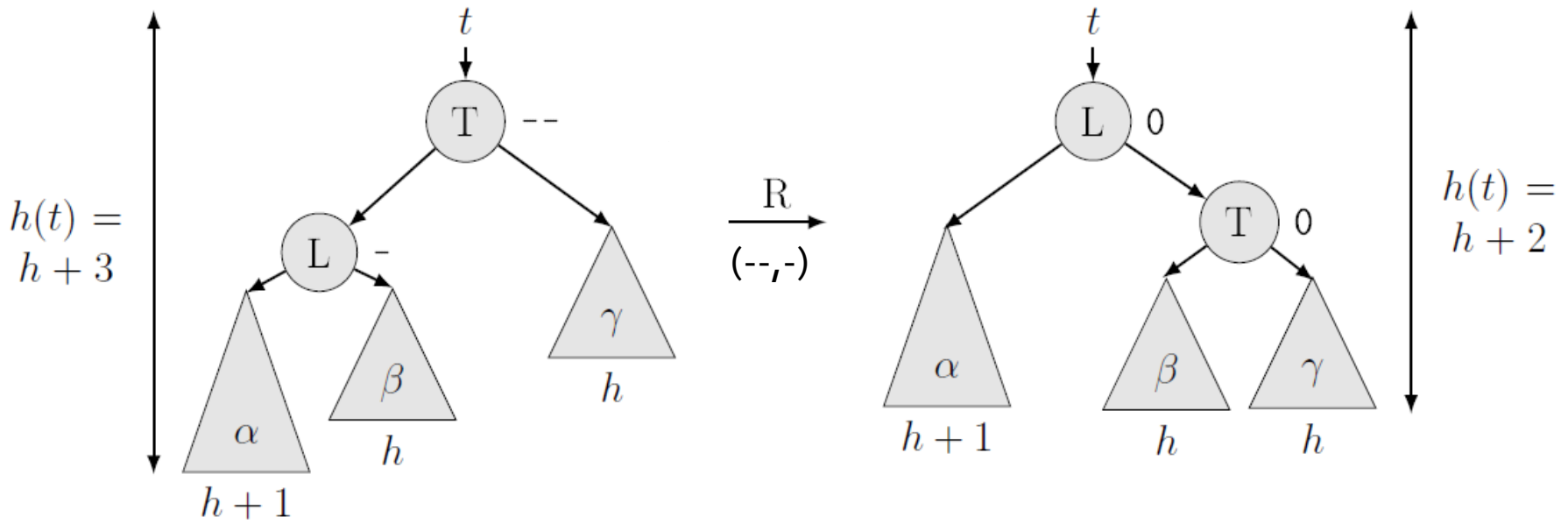
A kiegyensúlyozatlan (T) csúcsnak és magasabb részfája (R) gyökércsúcsának egyensúlya nem ellentétes előjelű. Így a magasabb részfájához tartozó gyermeke, (R), a kiegyensúlyozó forgatás után az új gyökércsúcs. A többiek helyét a BST tulajdonság már meghatározza.



[α T (β R γ)] -> [(α T β) R γ]

AVL fa forgatások: (--,-) forgatás

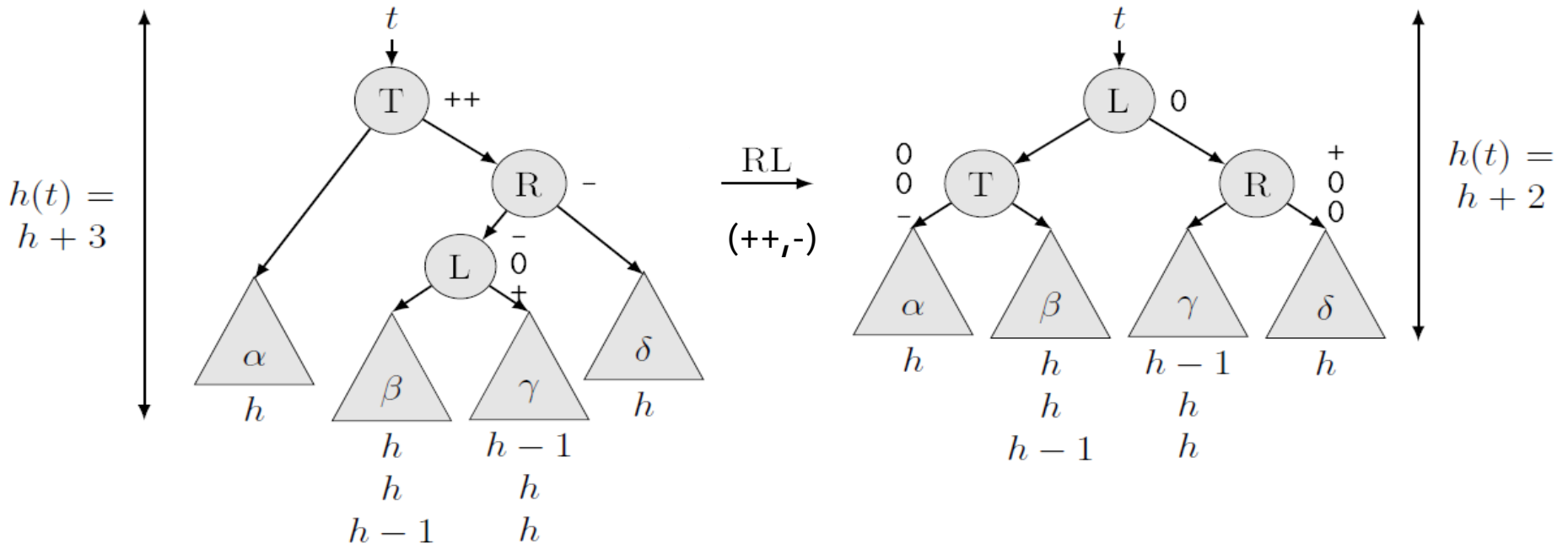
A kiegyensúlyozatlan (T) csúcsnak és magasabb részfája (L) gyökércsúcsának egyensúlya nem ellentétes előjelű. Így a magasabb részfájához tartozó gyermeke, (L), a kiegyensúlyozó forgatás után az új gyökércsúcs. A többiek helyét a BST tulajdonság már meghatározza.



$[(\alpha\ \mathrm{L}\ \beta)\ \mathrm{T}\ \gamma]$ -> $[\alpha\ \mathrm{L}\ (\beta\ \mathrm{T}\ \gamma)]$

AVL fa forgatások: (++,-) forgatás

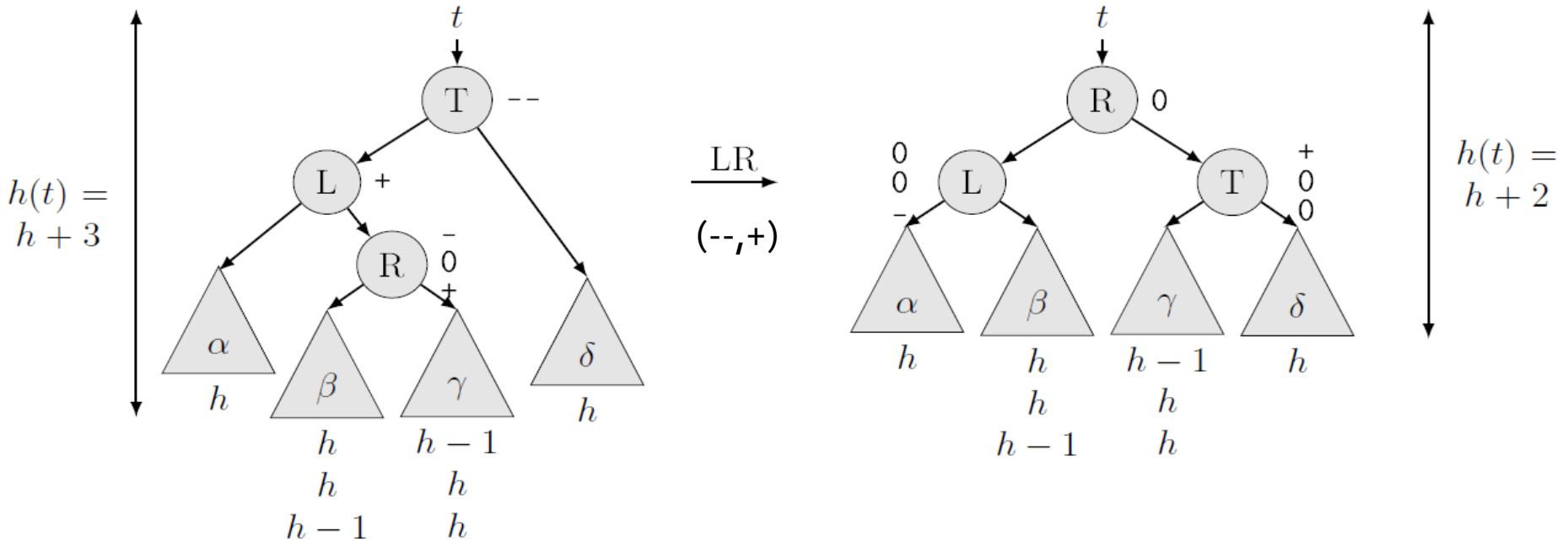
A kiegyensúlyozatlan (T) csúcsnak és magasabb részfája (R) gyökércsúcsának egyensúlya ellentétes előjelű. Így a magasabb részfájához tartozó unokája, (L), a kiegyensúlyozó forgatás után az új gyökércsúcs. A többiek helyét a BST tulajdonság már meghatározza.



{ α T [(β L γ) R δ] } -> { [α T β] L [γ R δ] }

AVL fa forgatások: (--,+) forgatás

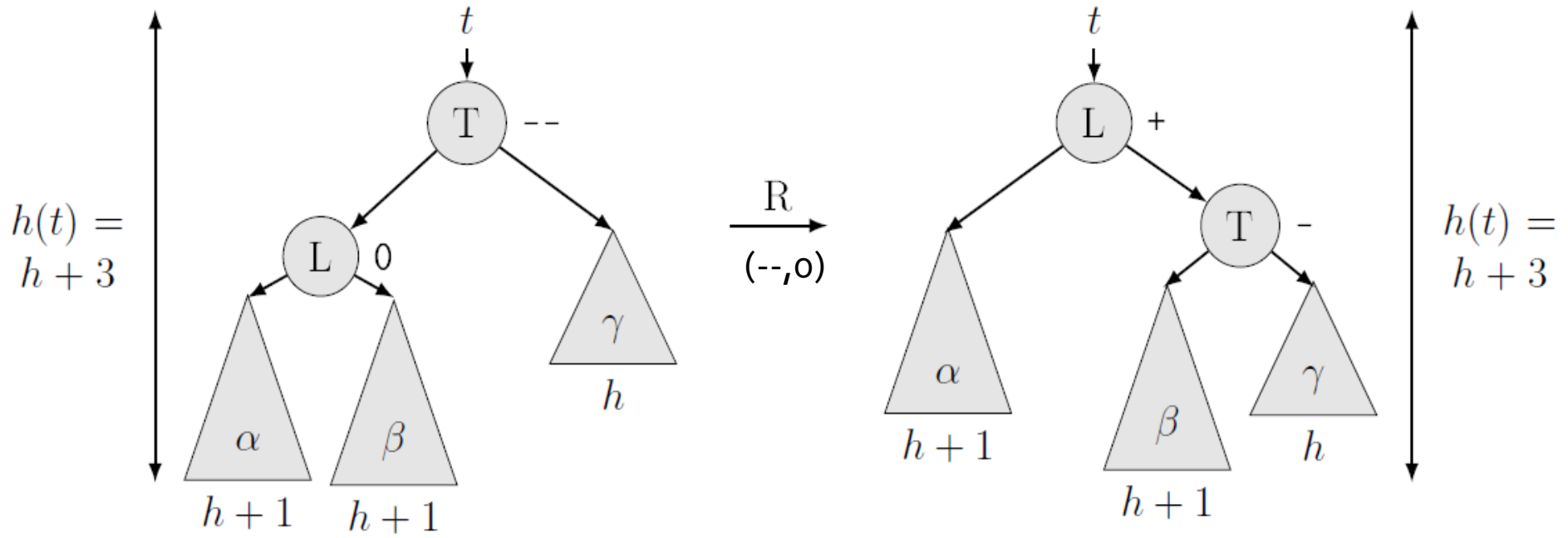
A kiegyensúlyozatlan (T) csúcsnak és magasabb részfája (L) gyökércsúcsának egyensúlya ellentétes előjelű. Így a magasabb részfájához tartozó unokája, (R), a kiegyensúlyozó forgatás után az új gyökércsúcs. A többiek helyét a BST tulajdonság már meghatározza.



{ [α L (β R γ)] T δ } -> { [α L β] R [γ T δ] }

AVL fa forgatások: (--,o) forgatás

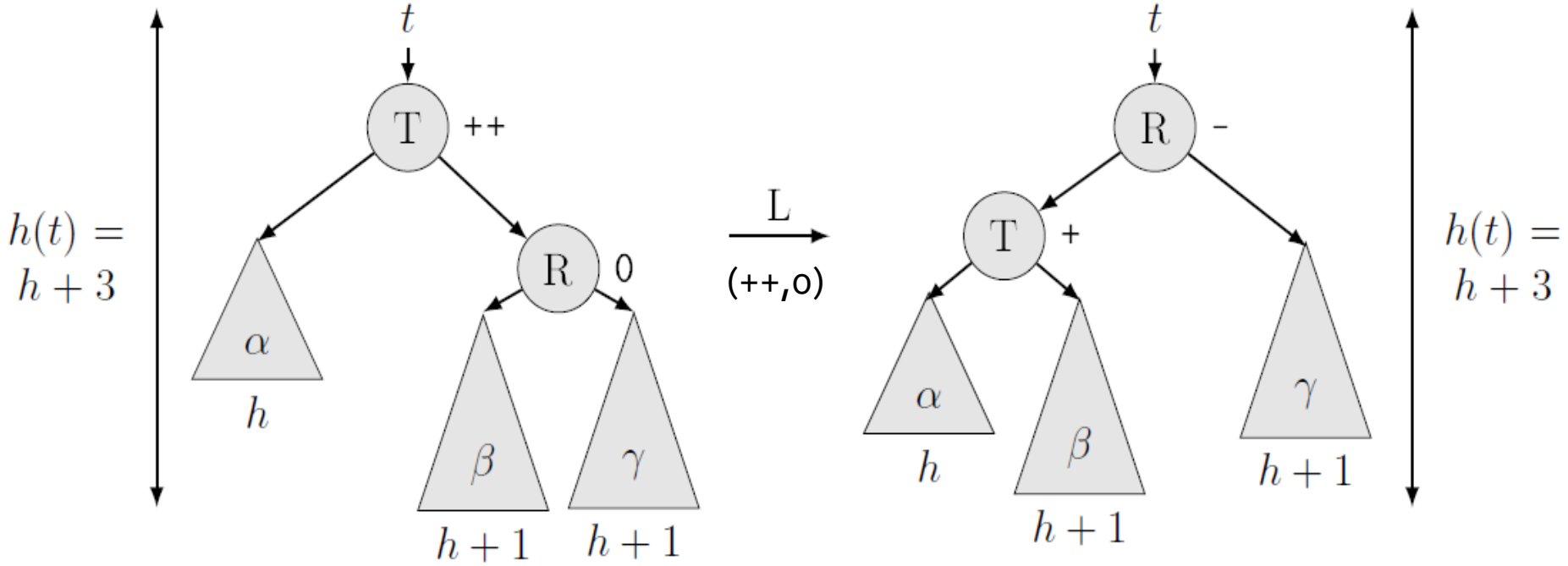
A kiegyensúlyozatlan (T) csúcsnak és magasabb részfája (L) gyökércsúcsának egyensúlya nem ellentétes előjelű. Így a magasabb részfájához tartozó gyermeke, (L), a kiegyensúlyozó forgatás után az új gyökércsúcs. A többiek helyét a BST tulajdonság már meghatározza.



$[(\alpha\ \mathrm{L}\ \beta)\ \mathrm{T}\ \gamma]$ -> $[\alpha\ \mathrm{L}\ (\beta\ \mathrm{T}\ \gamma)]$

AVL fa forgatások: (++,0) forgatás

A kiegyensúlyozatlan (T) csúcsnak és magasabb részfája (R) gyökércsúcsának egyensúlya nem ellentétes előjelű. Így a magasabb részfájához tartozó gyermeke, (R), a kiegyensúlyozó forgatás után az új gyökércsúcs. A többiek helyét a BST tulajdonság már meghatározza.



[α T (β R γ)] -> [(α T β) R γ]

AVL fa magassága

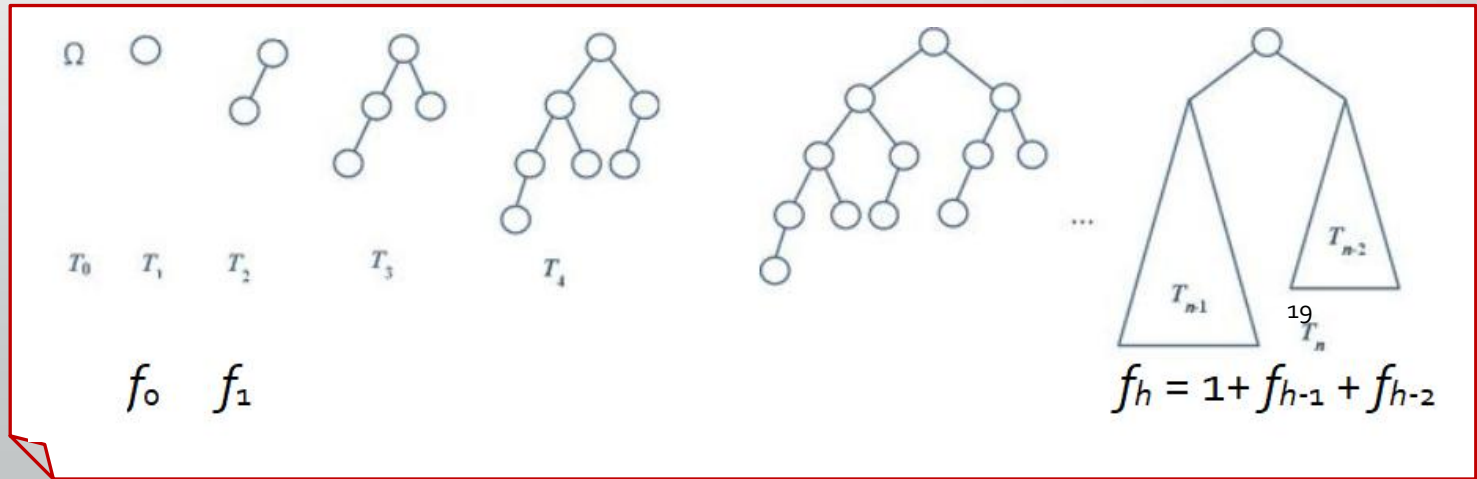
- **Tétel:** Tetszőleges *n* csúcsú nemüres AVL fa *h* magasságára:

$$\lceil \log n \rceil \leq h \leq 1{,}45 \log n, \text{ azaz } h \in \Theta(\log n)$$

- **Bizonyítás** vázlata: 1. 2.

1. alsó és felső becslés a *h* magasságú, nemüres KBF-ek *n* méretére
   - KBF: kiegyensúlyozott, bináris fák
   - $n < 2^{h+1} \rightarrow \lceil \log n \rceil \leq h$
2. a *h* mélységű, legkisebb méretű KBF-ek csúcsainak f_h számának meghatározása:
   - $f_0 = 1, f_1 = 2, f_h = 1 + f_{h-1} + f_{h-2}$. (**Fibonacci fák**)
   - tetszőleges *h* magasságú KBF *n* méretére: $n \geq f_h$

➢ $h \leq 1{,}45 \log n$

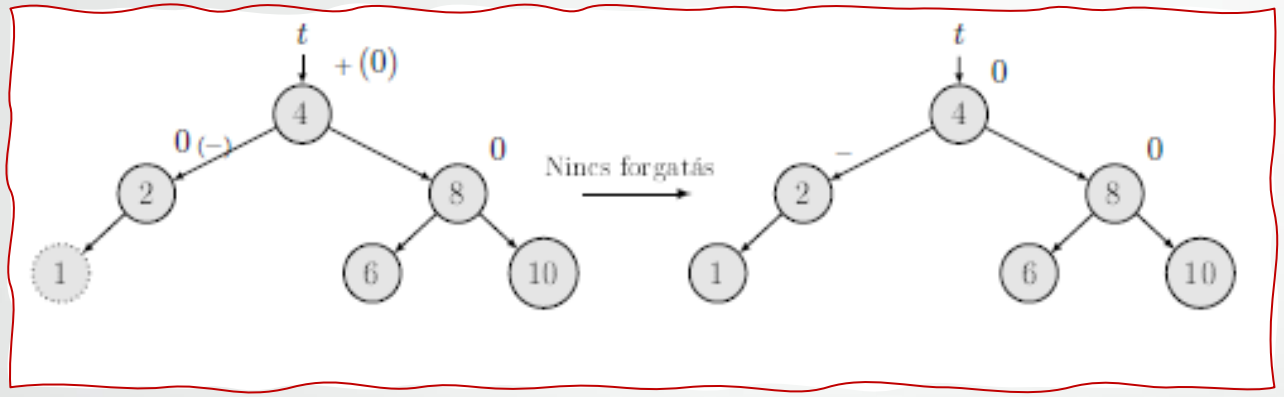


Műveletigény

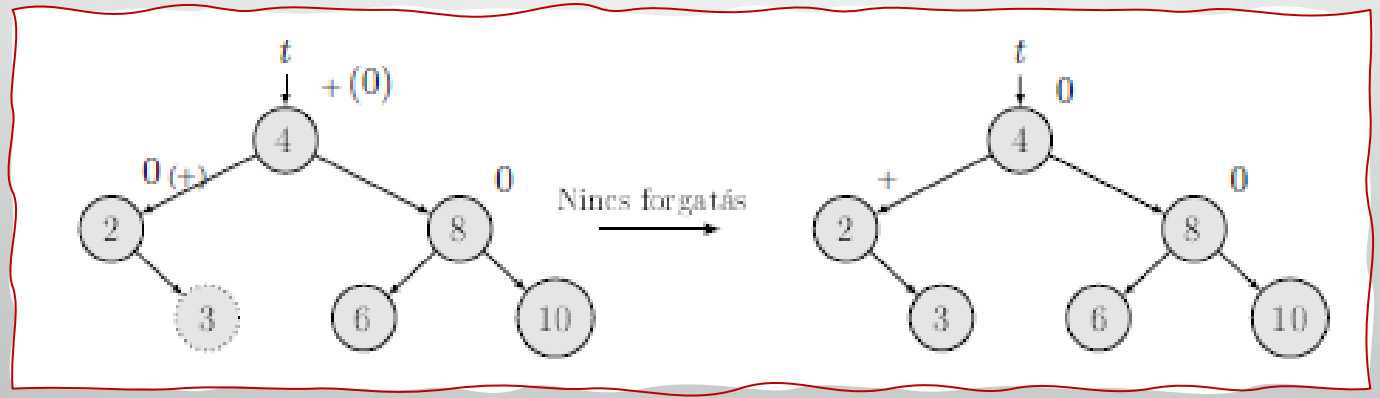
- **search(*t*; *k*), min(*t*), max(*t*)**
  - $MT(n) \in \Theta(\log n)$, ahol $n = |t|$
  - az AVL fák magassága: $\Theta[\log n]$
- **insert(*t*; *k*), del(*t*; *k*), remMin(*t*; *minp*), remMax(*t*; *maxp*)**
  - változtatják a fa alakját -> elromolhat a kiegyensúlyozottság
    - már nem garantált a fenti műveletigény
  - Elkerülésére:
    - Minden rekurzív eljáráshívás után ellenőrizés: hogyan változott a megfelelő részfa magassága
    - Ez hogyan befolyásolta a felette levő csúcs kiegyensúlyozottságát
    - Szükség esetén: helyreállítás
    - minden szinten **legfeljebb konstans** mennyiségű **extra műveletet** fog jelenteni
  - $MT(n) \in \Theta(\log n)$, ahol $n = |t|$

Példa AVL fába beszúrás

- Kiinduló fa: {[2]4+[(6)8◦(10)]}



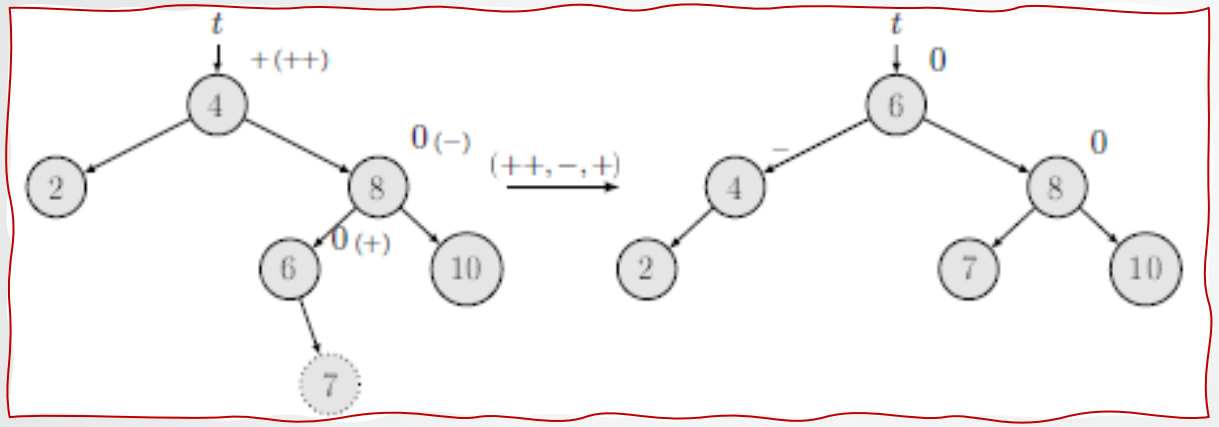
- 1 beszúrása -> {[(1)2−()]4◦[(6)8◦(10)]}



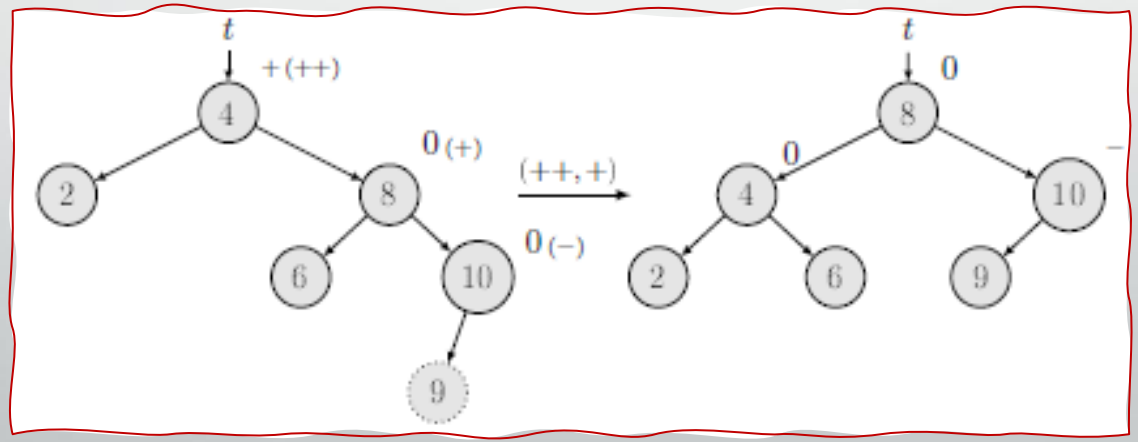
- 3 beszúrása - > {[()2+(3)]4◦[(6)8◦(10)]}

Példa AVL fába beszúrás

- Kiinduló fa: {[2]4+[(6)8◦(10)]}



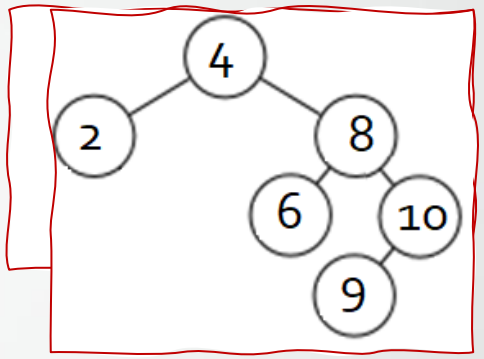
- 7 beszúrása -> {[2]4++[(⟨⟩6+⟨7⟩)8−(10)]}



- 9 beszúrása -> {[2]4++[(6)8+(⟨9⟩10−⟨⟩)]}

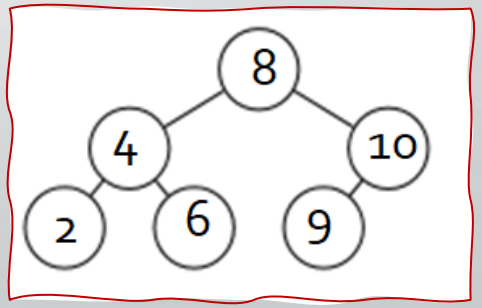
AVL fák: beszúrás (balra forgatás)

- A transzformáció helyességének belátása
  - $h = h(\alpha)$ jelölés
  - a kiinduló fa:
    - T++ és R+ egyensúlyok
      - $h((\beta\ R\ \gamma)) = h+2$
      - $h(\gamma) = h+1$
      - $h(\beta) = h$
    - az eredmény fára:
      - $h(\alpha) = h = h(\beta)$ -> T◦
      - $h((\alpha\ T◦\ \beta)) = h + 1 = h(\gamma)$ -> R◦



{ [2] 4++ [(6) 8+ ({9} 10-)] }

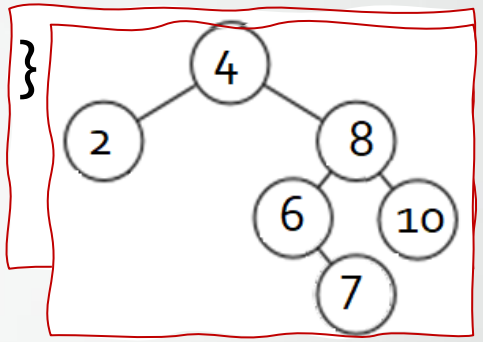
Balra forgatás (Left rotation):
[α T (β R γ)] -> [(α T β) R γ]



{ [(2) 4 (6)] 8 ◦ [(9) 10-] }

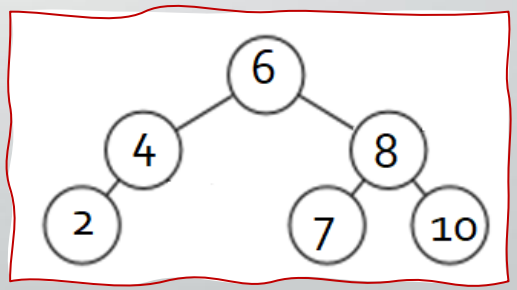
AVL fák: beszúrás (kettős forgatás)

- { α T++ [(β L-∘+ γ) R- δ] } -> { [α T∘∘- β] L∘ [γ R+∘∘ δ] }
  - T=4, α = [2], R=8, δ = (10), L=6+, β = $\emptyset$ γ = {7}
- Kettős forgatás (8. 9. dia)
  - { α T [(β L γ) R δ] } fára az R csúcsnál alkalmaztunk egy jobbra forgatást
  - { α T [β L (γ R δ)] } eredményfát balra forgatjuk
  - Eredményfa: { [α T β] L [γ R ∘ δ] }



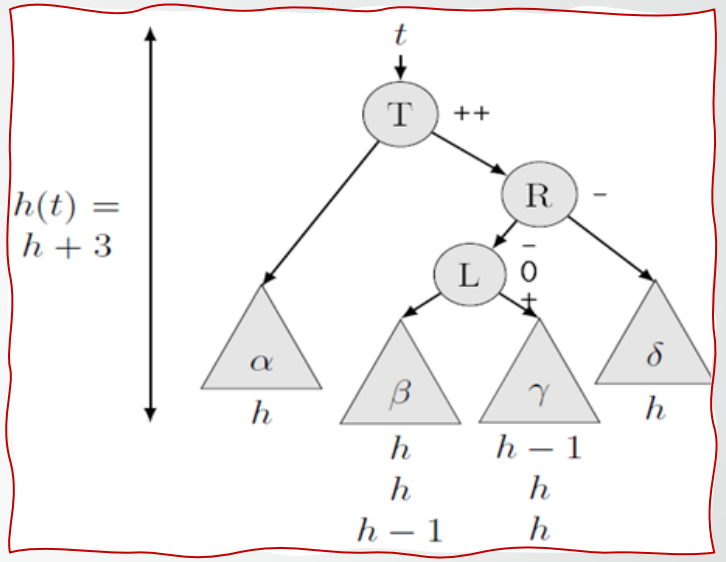
{ [2] 4++ [(6 + {7}) 8- (10)] }

Kettős (RL) forgatás:
{ α T [(β L γ) R δ] } -> { [α T β] L [γ R δ] }



{ [(2) 4-] 6∘ [(7) 8∘ (10)] }

Kettős forgatás helyessége:



- $h = h(\alpha)$ jelölés
- T++ -> $h((\beta\ L\ \gamma)\ R\ \delta) = h+2$
- R- -> $h(\beta\ L\ \gamma) = h+1$ *és* $h(\delta) = h$
- L lehetséges egyensúlyai szerint:
  - L- -> $h(\beta) = h$ és $h(\gamma) = h-1$
  - L∘ -> $h(\beta) = h$ és $h(\gamma) = h$
  - L+ -> $h(\beta) = h-1$ és $h(\gamma) = h$

➢ Eredményfában: T- és R∘

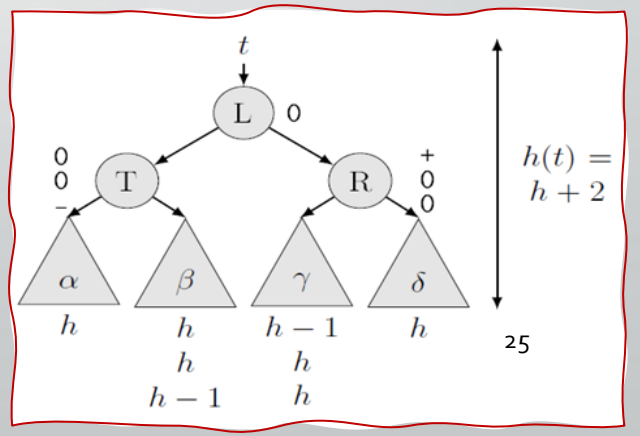
- Mindhárom esetben:

  $h([\alpha\ T\ \beta]) = h+1 = h([\gamma\ R\ \delta])$

  ➢ L∘

Kettős (RL) forgatás:

$\{\alpha\ T\ [(\beta\ L\ \gamma)\ R\ \delta]\}$ -> $\{[\alpha\ T\ \beta]\ L\ [\gamma\ R\ \delta]\}$



Beszúrás

- Összefüggések a kettős forgatás utáni egyensúlyokra
  - Jelölések:
    - az L csúcsnak a kettős forgatás előtti egyensúlya: *s*
    - a T és a R csúcsoknak a kettős forgatás utáni egyensúlyai: s_t, s_r
  - $s_t = -\left\lfloor \frac{s+1}{2} \right\rfloor$ és $s_r = \left\lfloor \frac{1-s}{2} \right\rfloor$
- A fent említett két kiegyensúlyozási séma mellett igazak a tükörképei:
  - a forgatások előtt a T-- csúccsal a gyökérben
  - $[(\alpha\ L - \beta)T - -\gamma] \rightarrow [\alpha L\circ\ (\beta T \circ \gamma)\ ]$
  - $\{\ [\alpha\ L + (\beta\ R - \circ\ +\ \gamma)\ ]\ T - - \delta \rightarrow [\ (\alpha\ L \circ\circ \beta)\ R\circ\ (\gamma\ T + \circ\circ \delta)\ ]$
  - $s_l = -\left\lfloor \frac{s+1}{2} \right\rfloor$ és $s_t = \left\lfloor \frac{1-s}{2} \right\rfloor$

Beszúrás

Kérdések:

- Az AVL fában az új csúcs beszúrása után **mely csúcsoknak és milyen sorrendben** számoljuk újra az egyensúlyát?
- Mikor ütemezzük be a kiegyensúlyozást?

Válaszok

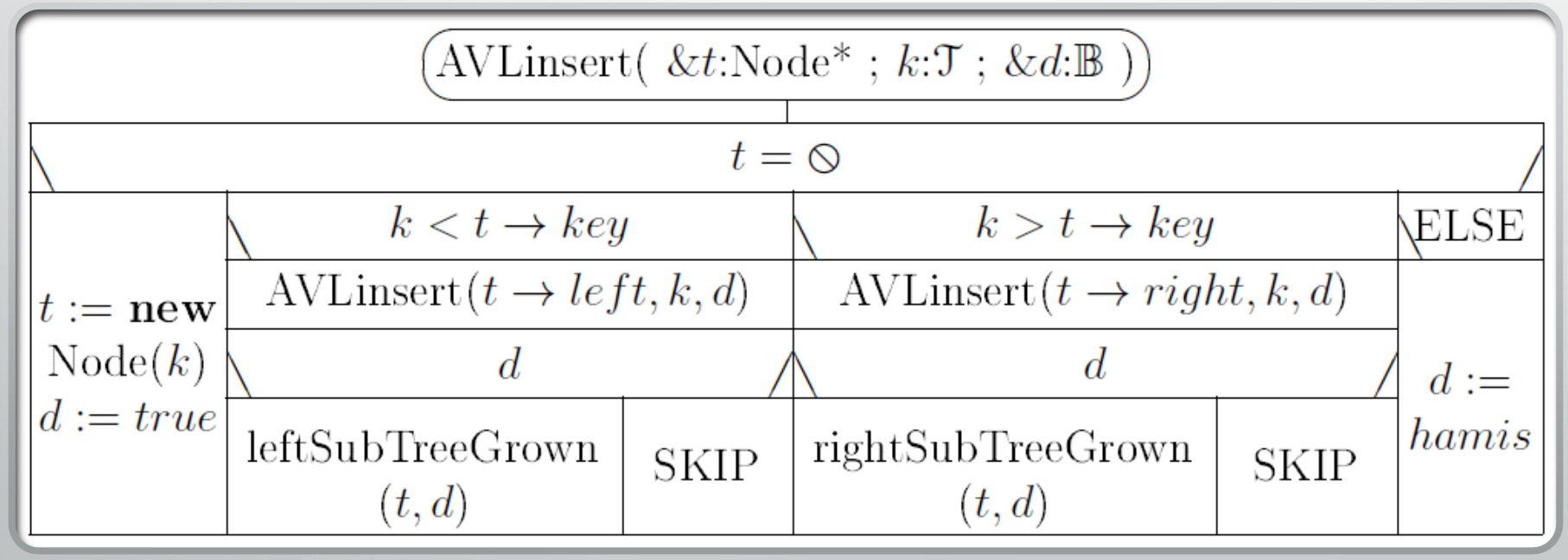
- Csak a beszúrás nyomvonalán visszafelé haladva kell újra számolnunk az egyensúlyokat
- Csak itt kell kiegyensúlyoznunk, ha kiegyensúlyozatlan csúcsot találunk

➢ Futási idő a fa magasságával arányos marad

- AVL fákra: $O(\log n)$

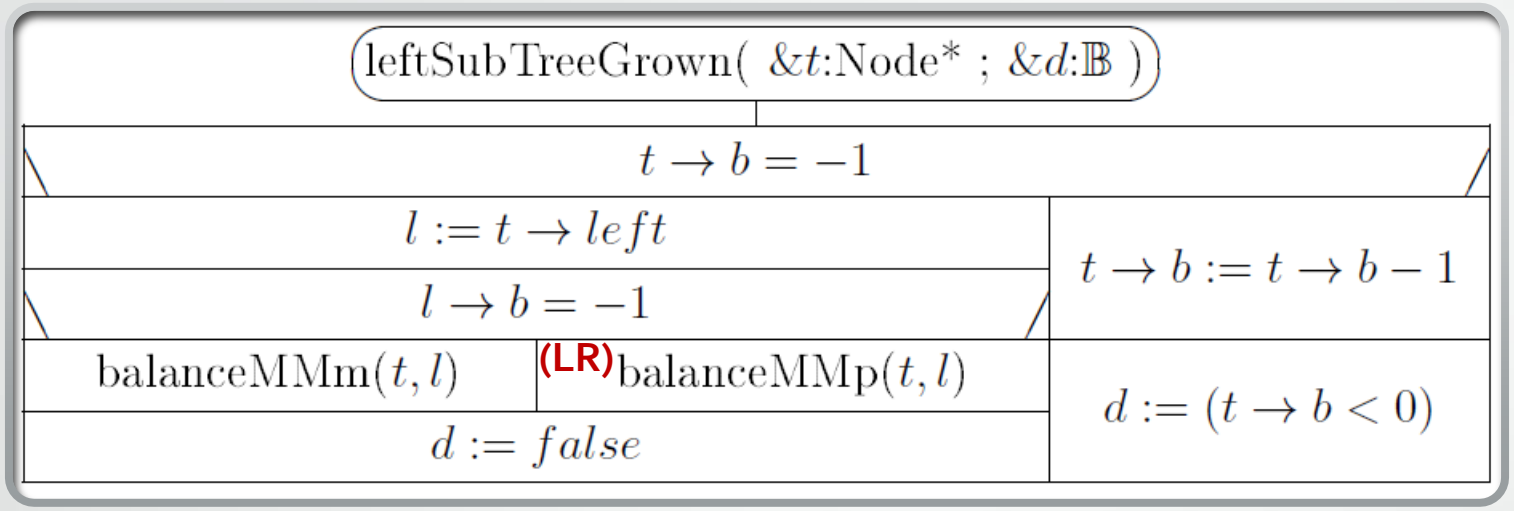
Beszúró és kiegyensúlyozó algoritmus működése

- A fa magassága:
  - eggyel növekszik (*d* = true)
  - ugyanannyi marad (*d* = false).

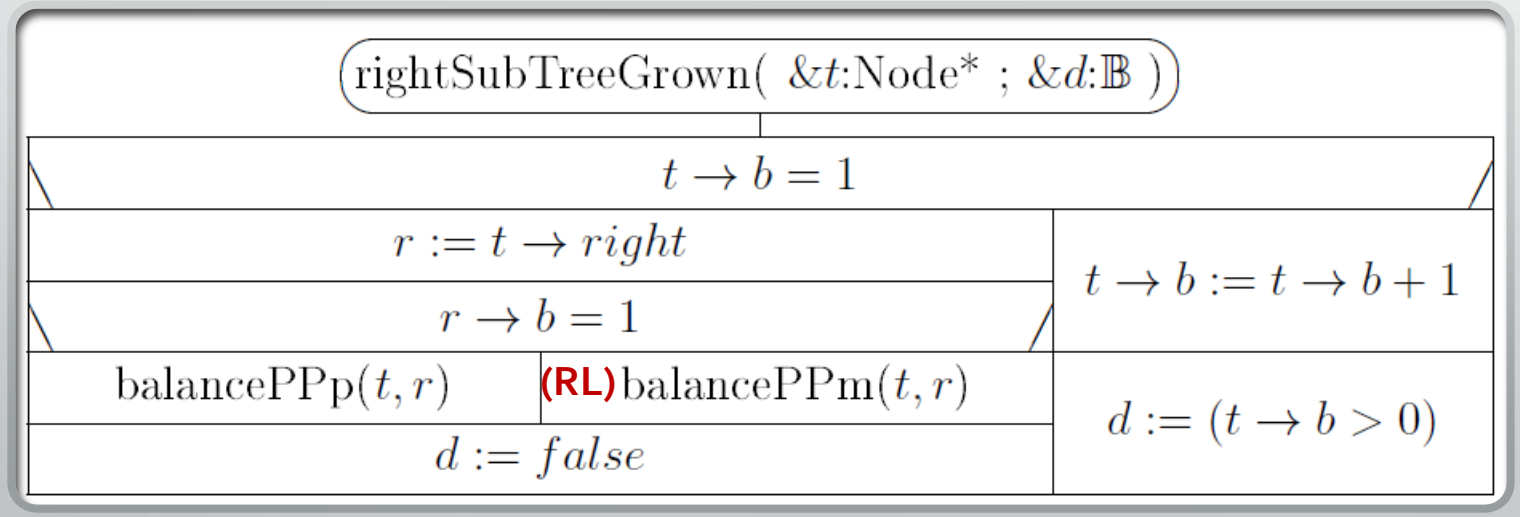


Beszúró és kiegyensúlyozó algoritmusok II.

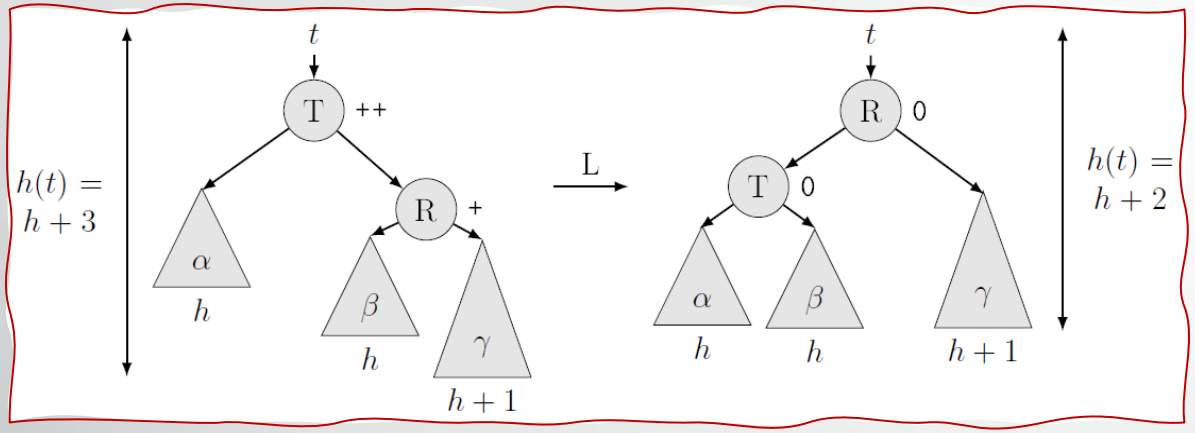
(R)



(L)



Beszúró és kiegyensúlyozó algoritmusok III.

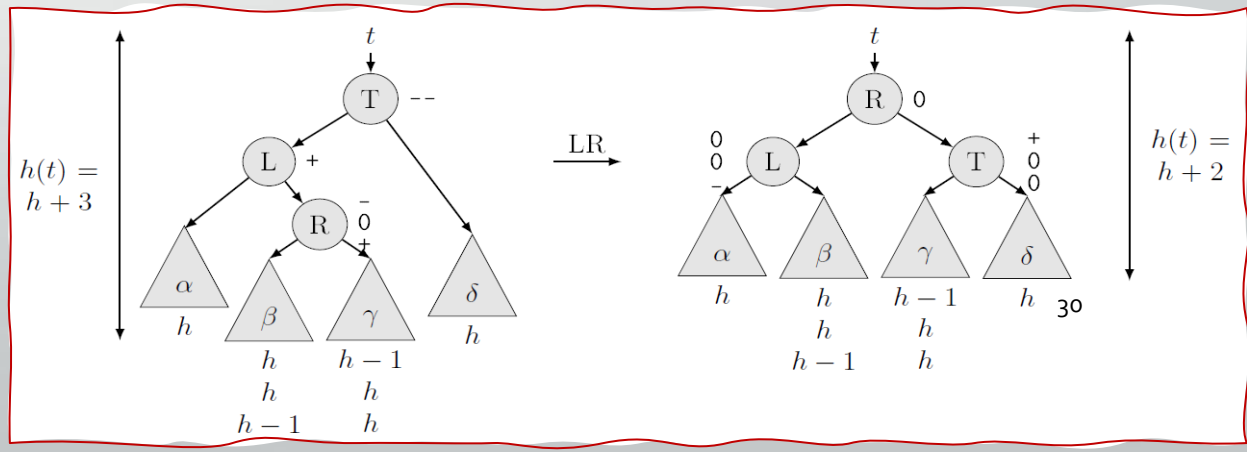


balancePPp(&t, r : Node*)

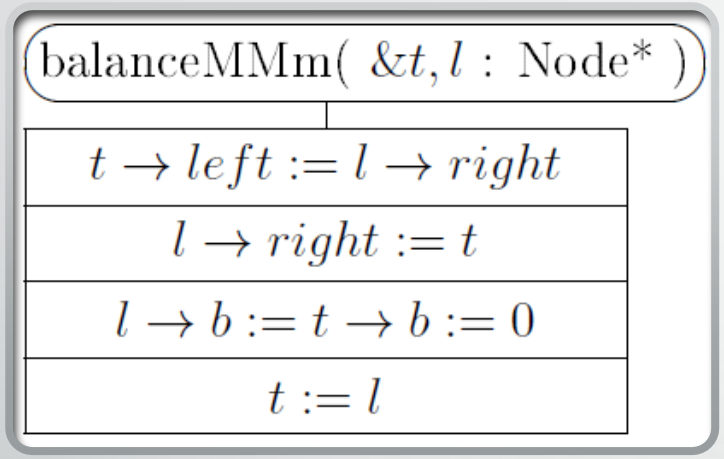
$t \to right := r \to left$
$r \to left := t$
$r \to b := t \to b := 0$
$t := r$

balanceMMp(&t, l : Node*)

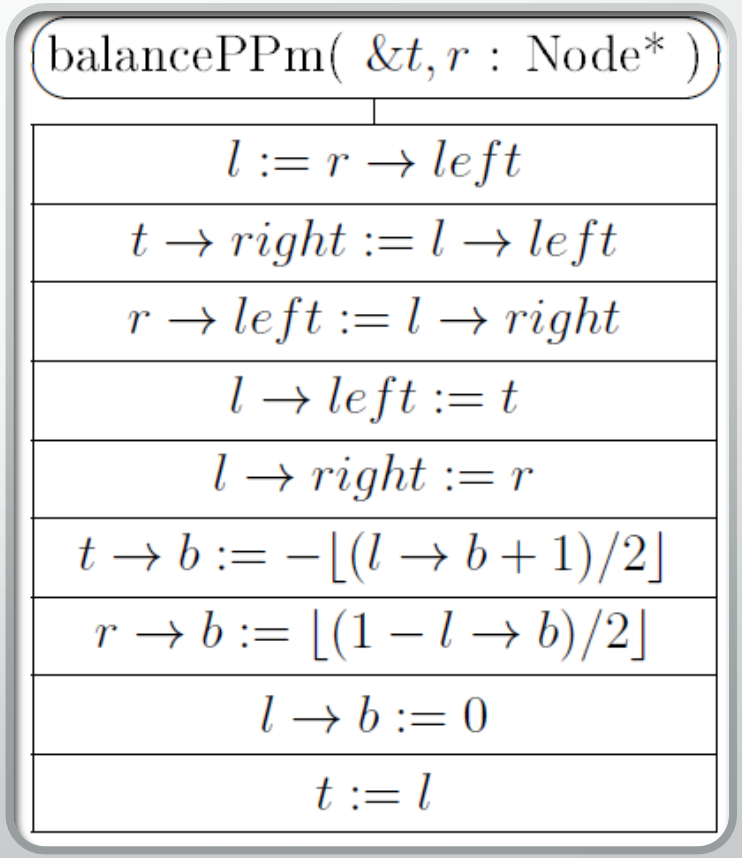
$r := l \to right$
$l \to right := r \to left$
$t \to left := r \to right$
$r \to left := l$
$r \to right := t$
$l \to b := -\lfloor (r \to b + 1)/2 \rfloor$
$t \to b := \lfloor (1 - r \to b)/2 \rfloor$
$r \to b := 0$
$t := r$



Beszúró és kiegyensúlyozó algoritmusok IV. (tükörképek)



~balancePPp (R forgatás)



~balanceMMp (RL forgatás)

Az AVL fába való beszúrás rövid összefoglalása

1. Megkeressük a kulcs helyét a fában.
2. Ha a kulcs benne van a fában
   - ➢ STOP.
3. Ha a kulcs helyén egy üres részfa található,
   - Beszúrunk az üres fa helyére egy új, a kulcsot tartalmazó levélcsúcsot.
   - Ez a részfa eggyel magasabb lett.

Az AVL fába való beszúrás rövid összefoglalása

4. Ha a gyökércsúcsnál vagyunk,
   - STOP.
   - Különben egyet fölfelé lépünk a keresőfában.
     - Mivel az a részfa, amiből fölfele léptünk, eggyel magasabb lett:
       - Az aktuális csúcs egyensúlyát megfelelőképp módosítjuk:
         - Ha a jobb részfa lett magasabb: +1, ha a bal: -1
5. Ha az aktuális csúcs egyensúlya o lett
   - az aktuális csúcshoz tartozó részfa alacsonyabb ága hozzá nőtt a magasabbikhoz
   - az aktuális részfa magassága = a beszúrás előtti állapottal
   - egyetlen más csúcs egyensúlyát sem kell módosítani: STOP.

Az AVL fába való beszúrás rövid összefoglalása

6. Ha az aktuális csúcs új egyensúlya 1 vagy -1
   - előtte 0 volt -> az aktuális részfa magasabb lett eggyel. -> a 4. ponttól folytatjuk.
7. Ha az aktuális csúcs új egyensúlya 2 vagy -2
   - a hozzá tartozó részfát ki kell egyensúlyozni.
     - A kiegyensúlyozás után az aktuális részfa visszanyeri a beszúrás előtti magasságát
     - már egyetlen más csúcs egyensúlyát sem kell módosítani: STOP.
   - A 2 és -2 eseteket a struktogramban nem számoltuk ki expliciten, hogy az egyensúly tárolására elég legyen két bit.

Az algoritmusban a kiegyensúlyozás után az aktuális részfa visszanyeri a beszúrás előtti magasságát

- **Állítás:** Az algoritmusban a kiegyensúlyozás után az aktuális részfa visszanyeri a beszúrás előtti magasságát

- **Bizonyítás:** A kiegyensúlyozandó részfa gyökere T++ eset: (~ T -- eset)
  - A T++ esethez tartozó kiegyensúlyozási sémák:
    - [α T++ (β R+ γ)] -> [(α T $\circ$ β) R $\circ$ γ]
    - { α T++ [(β L-$\circ$+ γ) R- δ] } -> { [α T$\circ\circ$- β] L$\circ$ [γ R+$\circ\circ$ δ] }

Bizonyítás folytatása

1. Ha a T csúcs jobboldali R gyereke + vagy - súlyú -> a fenti sémák közül a megfelelő alkalmazható:
   - a beszúró algoritmus a fában a beszúrás helyétől egyesével fölfele lépked
   - az első kiegyensúlyozatlan csúcsnál azonnal kiegyensúlyoz
   - ez alatt nincs kiegyensúlyozatlan csúcs (azaz az $\alpha\beta\gamma\delta$ részfák is kiegyensúlyozottak)
   - ➢ ez a fenti forgatások feltétele,
   - Keresőfa marad: a kiegyensúlyozás nélküli beszúró algoritmus garantál
2. A fenti sémák minden esetet lefednek, azaz R+ vagy R -:
   - R nem lehet a beszúrás által létrehozott új csúcs
     - mert különben T-nek a beszúrás előtti jobboldali részfája üres lett volna -> most nem lehetne T++
   - Ha a fölfele lépkedés során nulla egyensúlyú csúcs áll elő, akkor a fölötte levő csúcsok egyensúlya már nem módosul, így kiegyensúlyozatlan csúcs sem állhat elő. Márpedig most T++. Így tehát az új csúcstól T-ig fölfelé vezető úton minden csúcs, azaz R egyensúlya is + vagy -.

Bizonyítás folytatása

3. A kiegyensúlyozások visszaállítják a részfa beszúrás előtti magasságát.
   - A T++ kiegyensúlyozatlan csúcs a beszúrás előtt kiegyensúlyozott volt
   - A beszúrás, a beszúrás helyétől kezdve T-ig fölfelé vezető úton mindegyik részfa magasságát pontosan eggyel növelte ->, a beszúrás előtt T+ volt
   - A beszúrás előtt, $h = h(\alpha)$ jelöléssel, a T gyökerű részfa $h+2$ magas volt
   - A beszúrás után T++ lett -> a T gyökerű részfa $h+3$ magas lett
   - A beszúrás utáni, de még a kiegyensúlyozás előtti állapotot tekintve a T jobboldali gyerekére megkülönböztetjük:
     - R+ eset
     - R- eset

Bizonyítás folytatása

- R+ eset:
  - $h(\alpha) = h = h(\beta)$ és $h(\gamma) = h + 1$
  - ➢ A kiegyensúlyozás után: $h([(\alpha\ T\ \beta)\ R\ \gamma\ ]) = h + 2$
- R- eset:
  - $h(\alpha) = h = h(\delta)$ és $h(\beta\ L\ \gamma) = h + 1$
  - ➢ $h(\beta), h(\gamma) \leq h$
  - ➢ A kiegyensúlyozás után: $h(\{[\alpha\ T\ \beta]\ L\ [\gamma\ R\ \delta]\}) = h + 2$

➢ Ezzel beláttuk, hogy a kiegyensúlyozások mindkét esetben visszaállítják a részfa beszúrás előtti magasságát

- úgy, hogy a beszúrás által eggyel megnövelt magasságot eggyel csökkentik.

- Szimmetria okokból ez hasonlóan látható be T-- esetén is
  - figyelembe véve a L- és a L+ eseteket.

Ellenőrző kérdések

- Mit jelent az, hogy egy bináris fa magasság szerint kiegyensúlyozott?
- Hogyan számoljuk ki egy csúcs egyensúlyát?
- Hogyan változik a fa magassága beszúrás után?
- Miért csak a beszúrási útvonalon kell ellenőrizni a kiegyensúlyozottságot?
- Milyen sorrendben számoljuk újra az egyensúlyokat beszúrás után?
- Rajzoljuk le a következő AVL fát a belső csúcsok egyensúlyaival együtt!
  - { [() 1 (2)] 4 [(5) 6 (⟨7⟩ 8 ⟨⟩)] }
  - Szemléltessük a 3 beszúrását és a 4 törlését,
    - mindkét esetben az eredeti fára!
    - Jelöljük, ha ki kell egyensúlyozni, a kiegyensúlyozás helyét, és a kiegyensúlyozás után is rajzoljuk újra fát! A rajzokon jelöljük a belső csúcsok egyensúlyait is, a szokásos módon!
  - Rajzoljuk le a hat általános kiegyensúlyozási séma közül azokat, amiket alkalmaztunk!

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek II. Előadásjegyzete (Fák) és Fekete István: Algoritmusok és adatszerkezetek / AVL-Fák előadásjegyzete alapján készült.

Algoritmusok és adatszerkezetek II.
2. Előadás

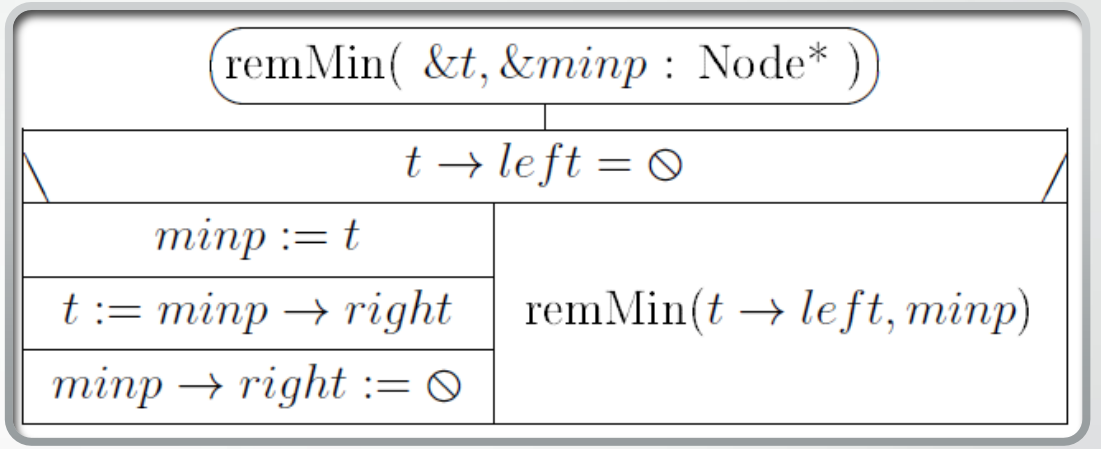
AVL fa II.

AVL fa II.: adott kulcsú csúcs törlése, Fibonacci fák, AVL fa magassága. Általános fák.

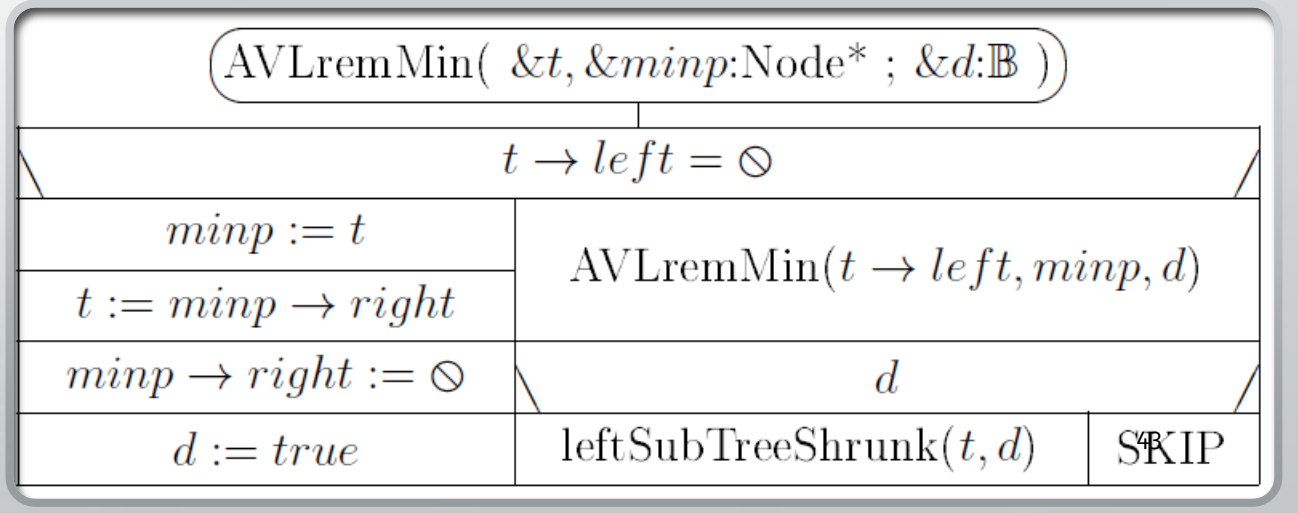
Tartalom

AVL fák: a remMin(*t*; *minp*) eljárás

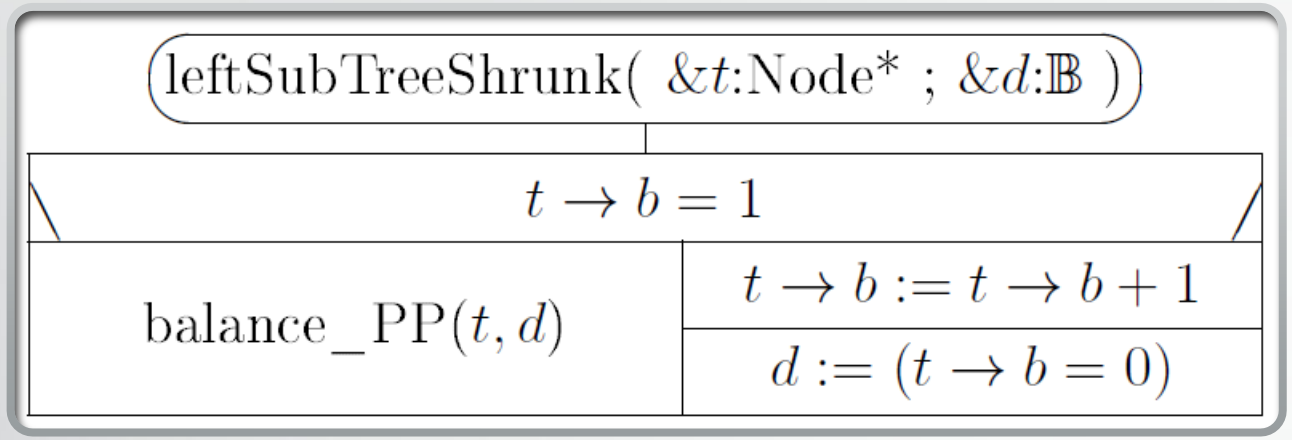
- Bináris keresőfákra a remMin eljárás:
  - Pl. { [2]4+[(6)8◦ (10)] } AVL fára alkalmazva
  - Eredmény: *minp*-> *key* = 2 és { 4++[(6)8◦ (10)] } kiegyensúlyozatlan binkerfa
  - Balra forgatás-> kiegyensúlyozza a fát
  - Nem csökkenti az aktuális részfa magasságát

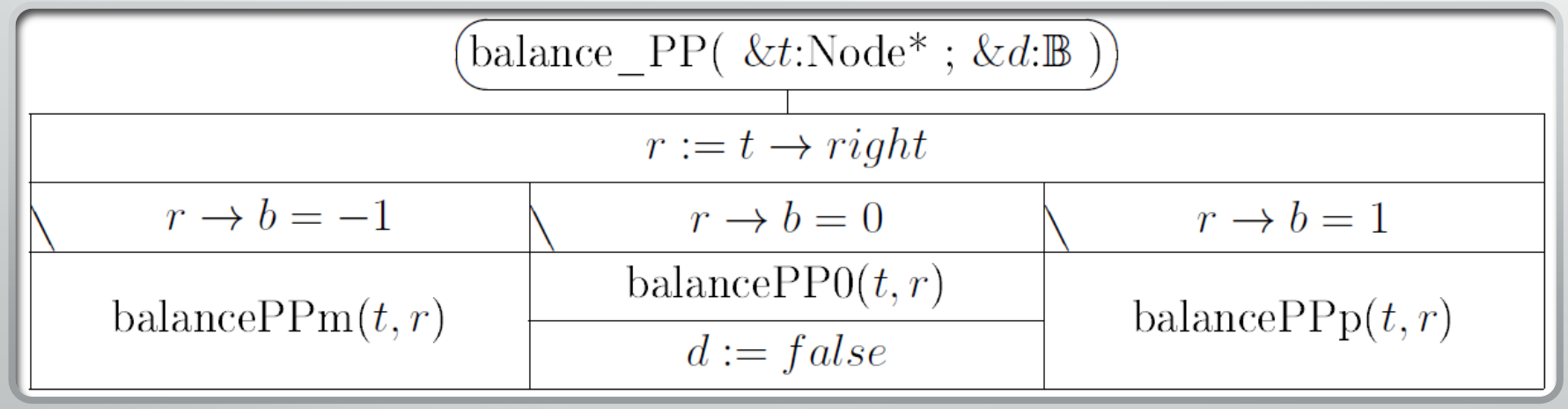


- AVL fákra alkalmazott, a megfelelő kiegyensúlyozásokkal kiegészített AVLremMin eljárás:
  - *d*: csökkent-e az aktuális részfa magassága

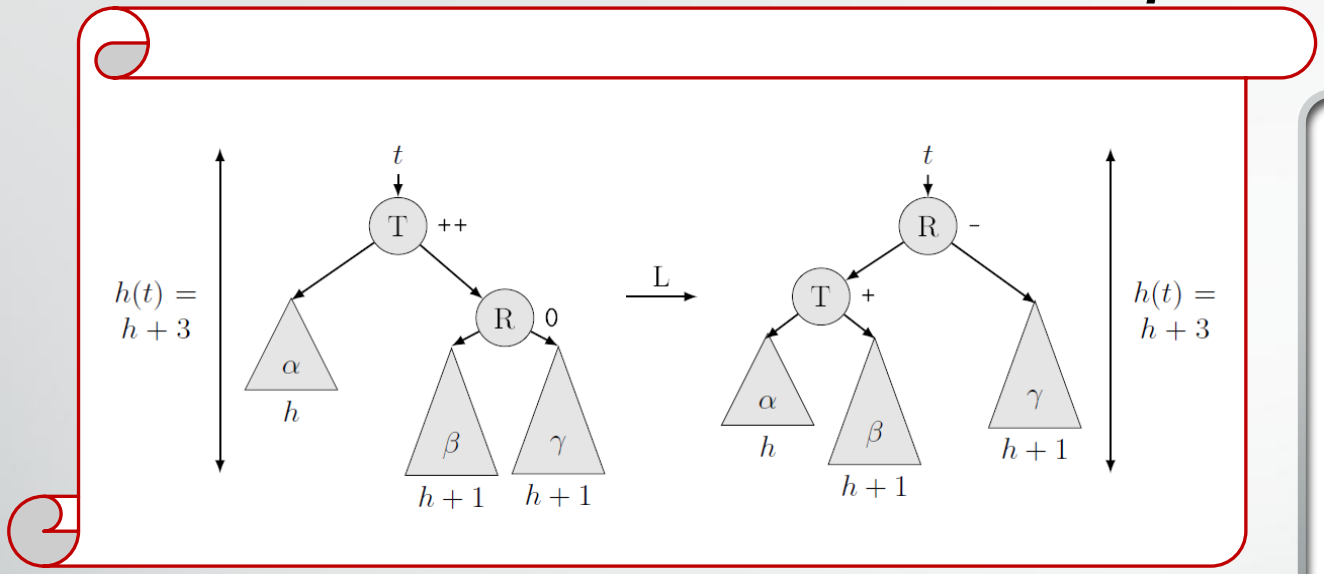


remMin(*t*; *minp*) eljárás segédeljárásai





remMin(*t*; *minp*) folytatás



balancePP0(&t, r : Node*)

$t \to right := r \to left$
$r \to left := t$
$t \to b := 1$
$r \to b := -1$
$t := r$

- Algoritmus helyességének igazolása:
  - ~ beszúrás
- Lényeges különbség:
  - Beszúrásnál: minden beszúrást csak egyetlen kiegyensúlyozás követ
  - Minimum eltávolításnál: minden felette lévő szinten ki kell egyensúlyozni

AVL fák: törlés

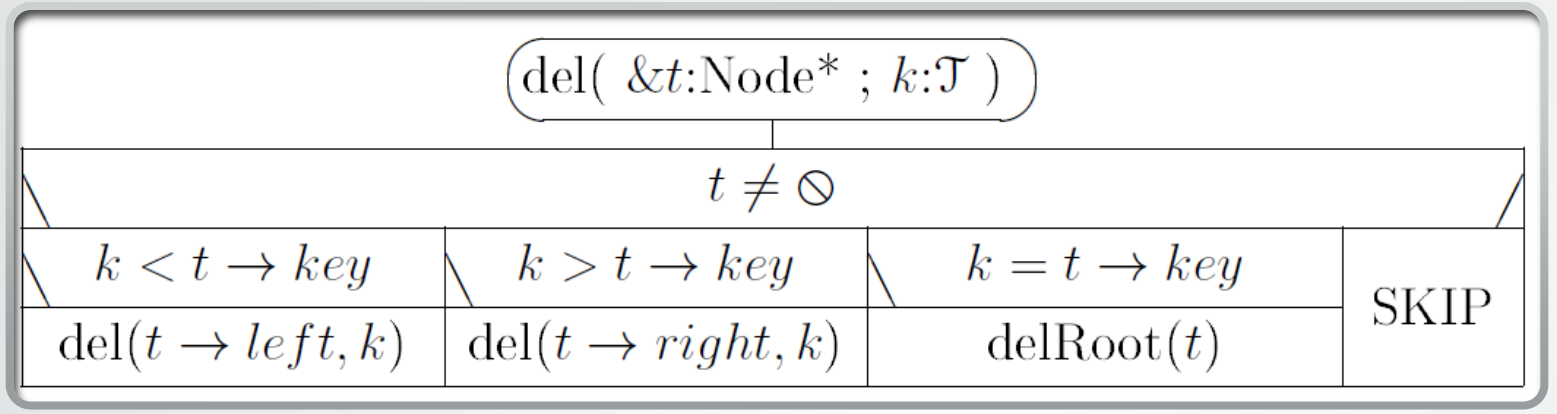
- Bináris keresőfákra a del(*t*; *k*) és a delRoot(*t*) eljárások:
- Változtatások:
  - ~ AVLremMin(*t*; *minp*; *d*)
  - + *d* paraméter

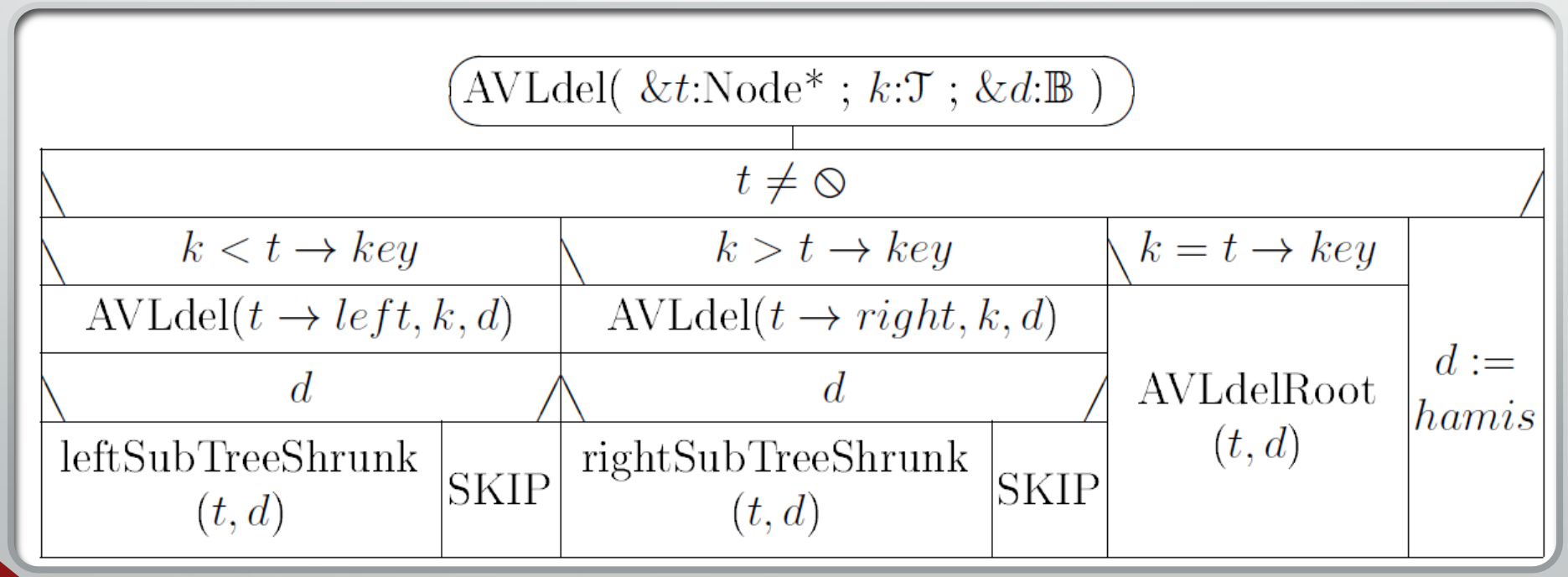
del(&t:Node* ; k:$\mathcal{T}$)			
$t \neq \oslash$			
$k < t \to key$	$k > t \to key$	$k = t \to key$	SKIP
$\mathrm{del}(t \to left, k)$	$\mathrm{del}(t \to right, k)$	$\mathrm{delRoot}(t)$	

  - ha csökkent az aktuális részfa magassága:
    - módosítjuk a *t* -> *b* értéket
    - ha kell: kiegyensúlyozunk
    - ha kell: *d*-t beállítjuk.

delRoot(&t:Node*)		
$t \to left = \oslash$	$t \to right = \oslash$	$t \to left \neq \oslash \wedge t \to right \neq \oslash$
$p := t$	$p := t$	$\mathrm{remMin}(t \to right, p)$
$t := p \to right$	$t := p \to left$	$p \to left := t \to left$
		$p \to right := t \to right$
delete p	**delete** p	**delete** t ; $t := p$

del(*t*, *k*) eljárás





delRoot(*t*) eljárás

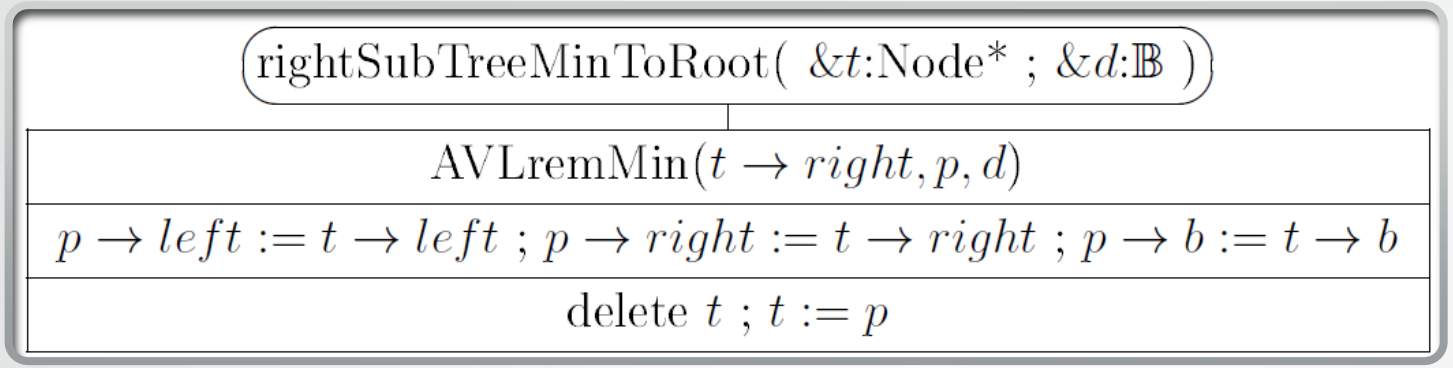
delRoot(&t:Node*)

$t \to left = \oslash$	$t \to right = \oslash$	$t \to left \neq \oslash \wedge t \to right \neq \oslash$
$p := t$	$p := t$	$\text{remMin}(t \to right, p)$
$t := p \to right$	$t := p \to left$	$p \to left := t \to left$
		$p \to right := t \to right$
delete p	**delete** p	**delete** t ; $t := p$

AVLdelRoot(&t:Node* ; &d:$\mathbb{B}$)

$t \to left = \oslash$	$t \to right = \oslash$	$t \to left \neq \oslash \wedge t \to right \neq \oslash$	
$p := t$	$p := t$	$\text{rightSubTreeMinToRoot}(t, d)$	
$t := p \to right$	$t := p \to left$		
delete p	**delete** p	d	
$d := true$	$d := true$	$\text{rightSubTreeShrunk}(t, d)$	SKIP

delRoot(t) eljárás folytatás



- Előfordulhat, hogy több szinten ki kell egyensúlyozni
  - a legrosszabb esetben akár minden felette lévő szinten
- Futási idő
  - egyetlen kiegyensúlyozás sem tartalmaz se rekurziót, se ciklust => konstans számú eljáráshívásból áll
  - sem itt, sem az AVLremMin eljárásnál nem befolyásolja a futási időnek az AVLinsert eljárásra is érvényes nagyságrendjét

➢ $\boldsymbol{MT(n) \in \Theta(\log n)}$, ahol $n = |t|$

AVL fa magassága*

- **Tétel:** Tetszőleges *n* csúcsú nemüres AVL fa *h* magasságára:

$$\lfloor \log n \rfloor \leq h \leq 1{,}45 \log n$$

- **Bizonyítás:**
  - $\lfloor \log n \rfloor \leq h$
    - elég azt belátni, hogy ez tetszőleges nemüres bináris fára igaz
      - Egy tetszőleges bináris fa 0.szintjén legfeljebb $2^0 = 1$ csúcs van, az 1. szintjén maximum $2^1 = 2$ csúcs, a 2. szintjén nem több, mint $2^2 = 4$ csúcs.
      - Általában: ha az *i*-edik szinten 2^i csúcs van, akkor az $i + 1$-edik szinten legfeljebb 2^{i+1} csúcs, hiszen minden csúcsnak maximum két gyereke van.

Bizonyítás folytatása

- egy h mélységű bináris fa csúcsainak n számára:
  $n \leq 2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1 < 2^{h+1}$
- $\lfloor \log n \rfloor \leq h < \log 2^{h+1} = h + 1$
- $\lfloor \log n \rfloor \leq h$

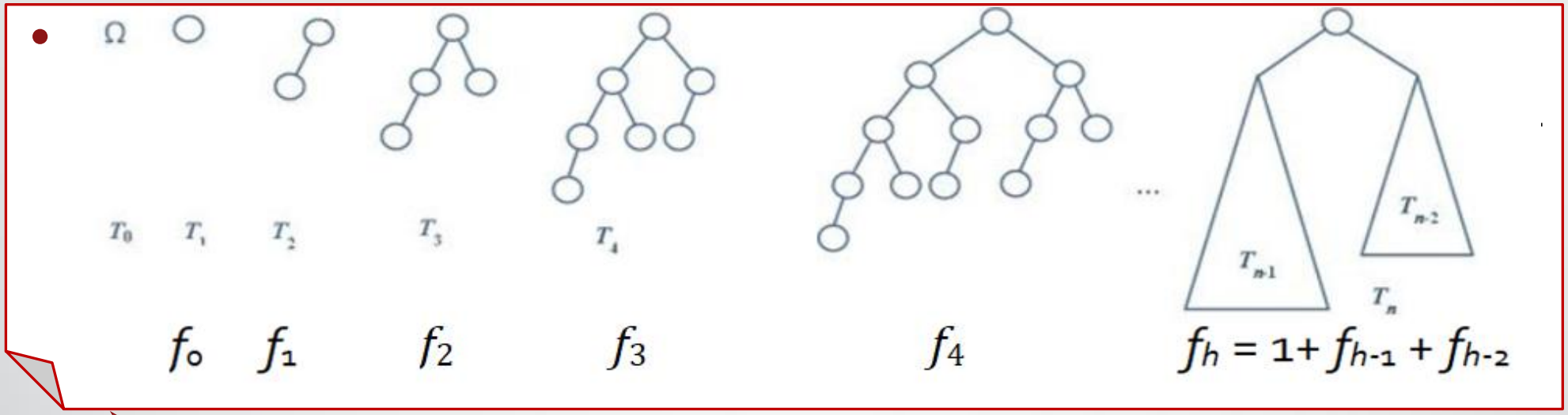
- $h \leq 1{,}45 \log n$ bizonyítása:
  - elég azt belátni, hogy ez tetszőleges nemüres KBF-ra igaz
    - KBF: kiegyensúlyozott, bináris fák

Bizonyítás folytatása

- a $h \geq 0$ magasságú, KBF-ek csúcsainak minimális f_h számának meghatározása
  - $f_0 = 1$ (egy nulla magasságú KBF csak a gyökércsúcsból áll)
  - $f_1 = 2$ (egy magasságú KBF-ek pedig ((B)G(J)), ((B)G), vagy (G(J)) alakúak)
  - az $< f_0; f_1; f_2; \ldots >$ sorozat szigorúan monoton növekvő
    - Igazolásához vegyünk egy $i + 1$ magas, f_{i+1} csúcsú t KBF-et!
    - Ekkor a t bal és jobb részfái közül az egyik magassága i.
    - Legyen ez az f részfa! Jelölje most $s(b)$ egy tetszőleges b bináris fa csúcsainak számát!
      - $f_{i+1} = s(t) > s(f) \geq f_i$
  - $h \geq 2$ esetén: $f_h = 1 + f_{h-1} + f_{h-2}$

Bizonyítás folytatása

- *h* ≥ 2 esetén: $f_h = 1 + f_{h-1} + f_{h-2}$



- $f_h = 1 + f_{h-1} + f_{h-2}$
  - A képlet emlékeztet a Fibonacci sorozatra: $F_0 = 0$, $F_1 = 1$, $F_h = F_{h-1} + F_{h-2}$, ha $h \geq 2$
- A *h* magasságú, és f_h méretű KBF-eket Fibonacci fáknak hívjuk.
- Az előbbiek szerint egy *h*-2 magasságú Fibonacci fa
  - (φ- 1 G φ- 2) vagy (φ- 2 G φ- 1) alakú
  - ahol φ- 1 és φ- 2 *h* - 1, illetve *h* -2 magasságú Fibonacci fák

Összefüggés az $< f_h : h \in \mathbb{N} >$ és az $< F_h : h \in \mathbb{N} >$ sorozatok között

h	0	1	2	3	4	5	6	7	8	9
F_h	0	1	1	2	3	5	8	13	21	34
f_h	1	2	4	7	12	20	33			

- $f_h = F_{h+3} - 1$
  $h \geq 0$

- Bizonyítás: Teljes indukcióval
  - Első néhány értékre: táblázat ($0 \leq h \leq k \geq 1$)
  - $h = k + 1$-re: $f_h = f_{k+1} = 1 + f_k + f_{k-1} = 1 + F_{k+3} - 1 + F_{k+2} - 1 = F_{k+4} - 1 = F_{h+3} - 1.$

$$F_h = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^h - \left(\frac{1-\sqrt{5}}{2} \right)^h \right]$$

$$n \geq f_h = F_{h+3} - 1 = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right] - 1 \geq$$

Összefüggés az $< f_h : h \in \mathbb{N} >$ és az $< F_h : h \in \mathbb{N} >$ sorozatok között

$$\geq \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - \left| \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right| \right] - 1$$

- $2 < \sqrt{5} < 3 \quad \rightarrow \quad \frac{|1-\sqrt{5}|}{2} < 1$ és $\left| (\frac{1-\sqrt{5}}{2})^{h+3} \right| < 1$

➢ $$n > \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1 \right] - 1 = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{3} \left(\frac{1+\sqrt{5}}{2} \right)^{h} - \frac{1+\sqrt{5}}{\sqrt{5}}$$

- Mivel $$\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{3} = \frac{1+3\sqrt{5}+3(\sqrt{5})^2+(\sqrt{5})^3}{8\sqrt{5}} = \frac{16+8\sqrt{5}}{8\sqrt{5}} = \frac{2+\sqrt{5}}{\sqrt{5}}$$

- Behelyettesítve az előbbi *n*-re vonatkozó összefüggésbe: $$n > \frac{2+\sqrt{5}}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h} - \frac{1+\sqrt{5}}{\sqrt{5}}$$

Összefüggés az $< f_h : h \in \mathbb{N} >$ és az $< F_h : h \in \mathbb{N} >$ sorozatok között

- Az első együtthatót tagokra bontva, a disztributív szabállyal:

$$n > \left(\frac{1+\sqrt{5}}{2}\right)^h + \frac{2}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - \frac{1+\sqrt{5}}{\sqrt{5}}$$

- Eszerint $h \geq 1$ esetén:

$$n > \left(\frac{1+\sqrt{5}}{2}\right)^h$$

- $h = 0$ -> $n = 1$

$$n \geq \left(\frac{1+\sqrt{5}}{2}\right)^h$$

➢ Azaz tetszőleges, nemüres KBF n méretére és h magasságára:

- Vegyük mindkét oldal kettes alapú logaritmusát, majd osszunk $\log\frac{1+\sqrt{5}}{2}$ -tel:

$$h \leq \frac{1}{\log\frac{1+\sqrt{5}}{2}} \log n$$

- $1{,}44 < 1{,}44042 < \frac{1}{\log\frac{1+\sqrt{5}}{2}} < 1{,}4404201 < 1{,}45$

➢ $\boldsymbol{h \leq 1{,}45 \log n}$

Általános fák

- Egy csúcsnak tetszőlegesen sok gyereke lehet
- Tetszőleges csúcshoz tartozó részfák száma pontosan egyenlő a gyerekek számával
  azaz nem tartoznak hozzá üres részfák
- Ha a gyerekek sorrendje lényeges: rendezett fákról beszélünk
- Általános fák használata:
  - modellezhetjük vele a számítógépünkben a mappák hierarchiáját, a programok blokkstruktúráját, a függvénykifejezéseket, a családfákat és bármelyik hierarchikus struktúrát.

Általános fák

- Általános fa nem általánosítása az *r*-áris fa fogalmának
  - ugyan minden konkrét általános fában van az egy csúcshoz tartozó gyerekek számára valamilyen *r* felső korlát
  - de ez a korlát nem abszolút: a fa tetszőleges csúcsa gyerekeinek száma tetszőlegesen növelhető
  - Másrészt: itt nem értelmezzük az üres részfa fogalmát
- Speciális csúcsok:
  - gyökércsúcs: nincs szülője
  - Levélcsúcs: nincs gyereke

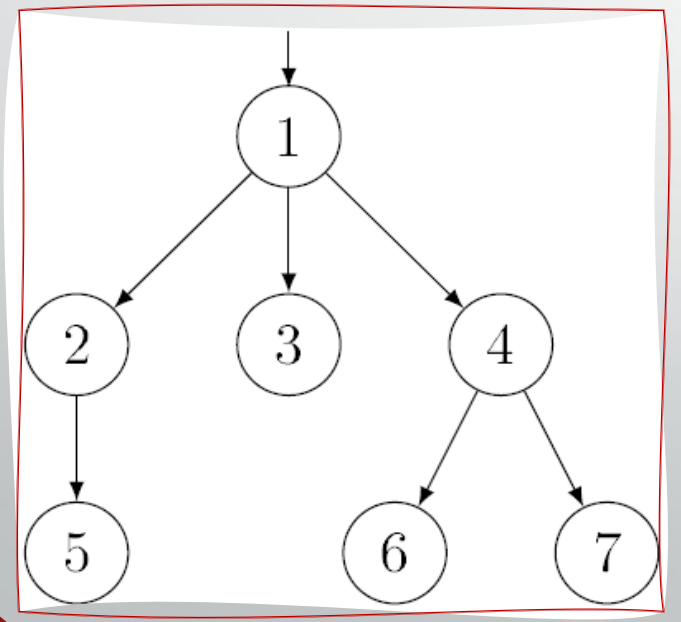
Általános fák ábrázolása

- Természetes ábrázolási módja: a bináris láncolt reprezentáció
  - egy csúcs szerkezete:
    - *child1*: az első gyerekre mutat
    - *sibling*: a következő testvérre mutat
    - *szülő* (opcionális): a gyerekek közös szülőjére mutat vissza
  - A **p* csúcs levél -> *p->child1* = ⊘
  - A **p* csúcs utolsó testvér -> *p->sibling* = ⊘

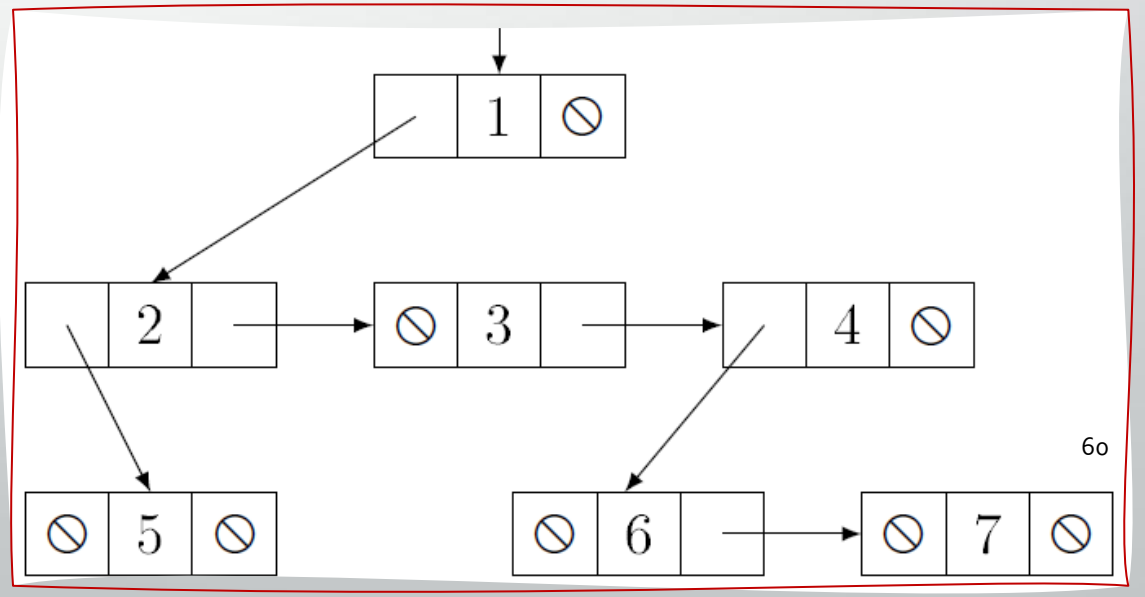
Node
+ $child1, sibling$: Node* // $child1$: első gyerek; $sibling$: következő testvér + key : T // T ismert típus
+ Node() { $child1 := sibling := \oslash$ } // egycsúcsú fát képez belőle + Node(x:T) { $child1 := sibling := \oslash$; $key := x$ }

Általános fák ábrázolása

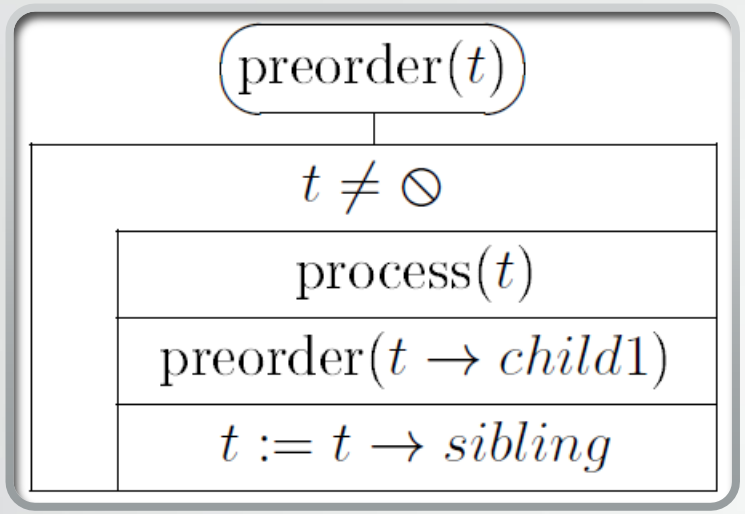
- Szöveges (zárójeles) reprezentáció:
  - A gyökér előre kerül
  - Egy nemüres fa általános alakja ($G\ t_1 \ldots t_n$)
    ahol G a gyökércsúcs tartalma, $t_1 \ldots t_n$ pedig a részfák
- { 1 [2 (5)] (3) [4 (6) (7)] } általános fa:
- Az általános fa absztrakt szerkezete:



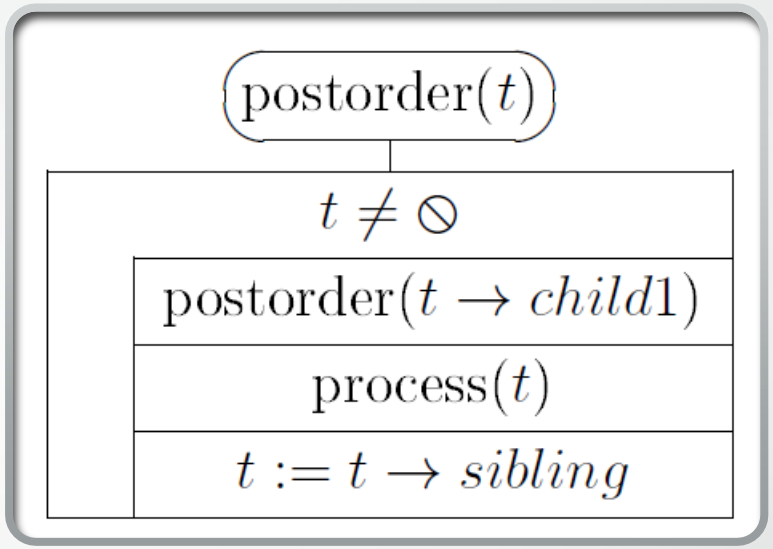
Az általános fa bináris reprezentációja:



Általános fa bejárásai



- Megfeleltetések:
  - *child1 ~ left*
  - *sibling ~ right*
- Felhasználása:
  - Fájlrendszereknél keresés



- Megfeleltetések:
  - *Postorder ~ inorder*
- Felhasználása:
  - függvénykifeJezések kiértékelése (kivéve a lusta kiértékelést)

- Inorder bejárás:
  - nem szimulálható a bináris reprezentáció egyik nevezetes bejárásával sem
  - A (G $t_1 \ldots t_n$) fa bejárása n>0 esetén: t_1, G $t_2 \ldots t_n$
    - n=0 esetén : G

Általános fa C# kód*

```
using System;

public class Node<T>
{
    public T Key { get; set; }
    public Node<T>? Child { get; set; }   // első gyerek
    public Node<T>? Sibling { get; set; } // következő testvér

    // Üres konstruktor
    public Node()
    {
        Child = null;
        Sibling = null;
    }

    // Kulcsot kapó konstruktor
    public Node(T key)
    {
        Key = key;
        Child = null;
        Sibling = null;
    }
}
```

```
public class GeneralTree<T>
{
    private readonly Action<T> _process;

    public GeneralTree(Action<T> process)
    {
        _process = process;
    }
```

```
    // Preorder bejárás (gyökér -> gyerekek -> testvérek)
    public void Preorder(Node<T>? t)
    {
        while (t != null)
        {
            _process(t.Key);          // process(t)
            Preorder(t.Child);        // preorder(t->child1)
            t = t.Sibling;            // t := t->sibling
        }
    }
```

```
    // Postorder bejárás (gyerekek -> gyökér -> testvérek)
    public void Postorder(Node<T>? t)
    {
        while (t != null)
        {
            Postorder(t.Child);       // postorder(t->child1)
            _process(t.Key);          // process(t)
            t = t.Sibling;            // t := t->sibling
        }
    }
}
```

```
// Példa használatra
class Program
{
    static void Main()
    {
        // Példa fa:
        //          A
        //        / | \
        //       B  C  D
        //      / \
        //     E   F

        var root = new Node<string>("A");
        root.Child = new Node<string>("B");
        root.Child.Sibling = new Node<string>("C");
        root.Child.Sibling.Sibling = new Node<string>("D");
        root.Child.Sibling.Child = new Node<string>("E");
        root.Child.Sibling.Child.Sibling = new Node<string>("F");

        var tree = new GeneralTree<string>(Console.WriteLine);

        Console.WriteLine("Preorder:");
        tree.Preorder(root);

        Console.WriteLine("\nPostorder:");
        tree.Postorder(root);
    }
}
```

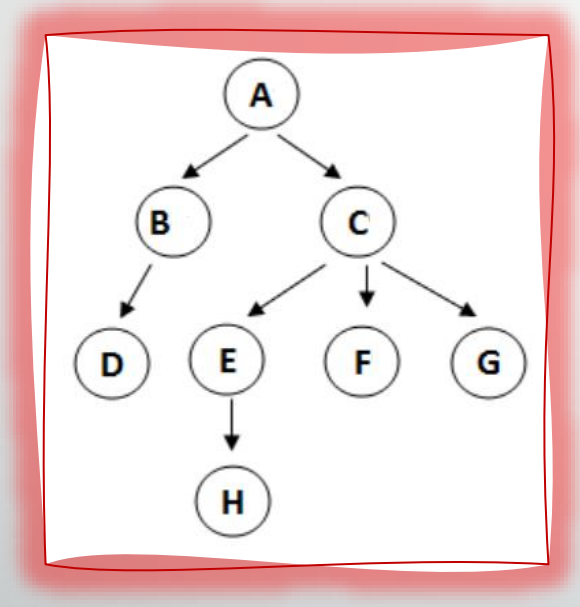
Kimenet:

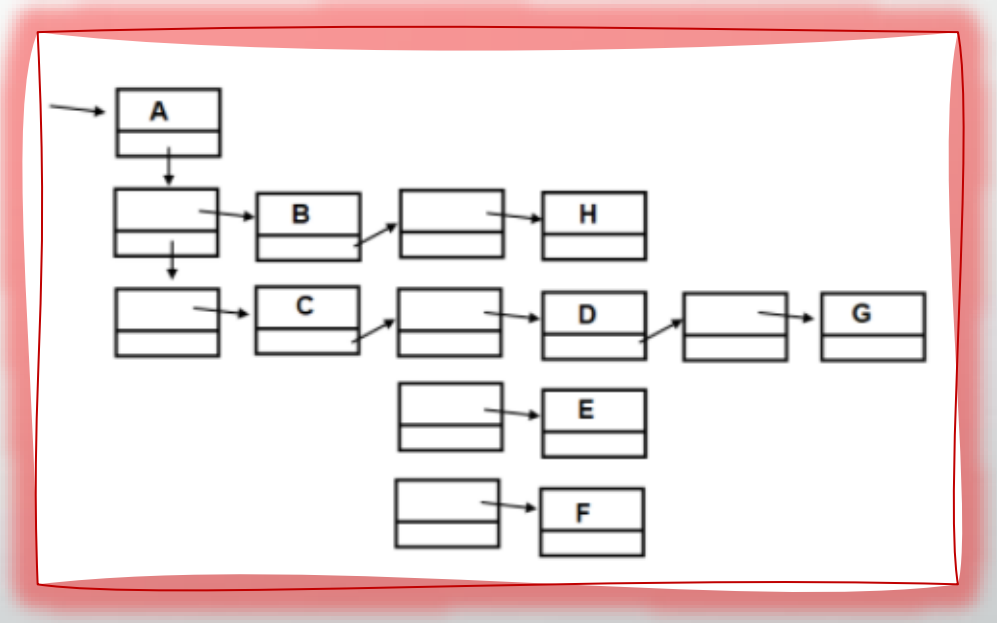
```
Preorder:
A
B
C
E
F
D
```

```
Postorder:
B
E
F
C
D
A
```

Általános fák ábrázolása másképp*

Multilistás ábrázolás





Minden csomópont egy láncolt lista. A lista első eleme tartalmazza az adatot, a többi csomópont már csak hivatkozásokat a leszármazottakra

Ellenőrző kérdések

1. Mit jelöl a **d** változó az AVLremMin eljárásban?
2. Hogyan biztosítja az **AVLremMin** algoritmus, hogy a fa magasság-különbsége mindig legfeljebb 1 maradjon?
3. Adj példát arra, mikor hívódik meg a **leftSubTreeShrunk(*t*, *d*)** eljárás!
4. Szemléltessük az 1 2 7 3 5 6 8 9 4 számok egymás utáni beszúrását egy, kezdetben üres AVL fába! Az így adódó fából indulva, ezután szemléltessük az 1 3 4 8 9 2 számok egymás utáni törlését! Minden egyes beszúrás illetve törlés után rajzoljuk újra fát! Jelöljük, ha ki kell egyensúlyozni, a kiegyensúlyozás helyét, és a kiegyensúlyozás után is rajzoljuk újra fát! A rajzokon jelöljük a belső csúcsok egyensúlyait is, a szokásos módon!
5. Rajzoljunk 0, 1, 2, 3, 4 és 5 magasságú Fibonacci fákat, amelyek egyben AVL fák is!
6. Mutassunk olyan, öt magasságú AVL fát, amelynek adott levelét törölve, minden, a fában a törölt csúcs feletti szinten ki kell egyensúlyozni!
7. Az előadásról ismert **leftSubTreeShrunk(*t*, *d*)** eljárás mintájára dolgozzuk ki az ott csak felhasznált **rightSubTreeShrunk(*t*, *d*)** eljárást, ennek segédeljárásait, és az ehhez szükséges kiegyensúlyozási sémát!

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek II. Előadásjegyzete (Fák) és Fekete István: Algoritmusok és adatszerkezetek / AVL-Fák előadásjegyzete alapján készült.

Algoritmusok és adatszerkezetek II.
3. Előadás

B+ fa fogalma, láncolt és szöveges ábrázolása, elhelyezkedése a háttértáron, magassága, műveletei.

Tartalom

B+ fák

- Nagy mennyiségű adatok tárolására kiválóan használható adatszerkezet
- Többszintű hierarchikus indexstruktúra
  - a listákból nem közvetlenül rekordokra, hanem újabb indexlistákra történik hivatkozás
  - minden közvetlenül a rekordokra mutató listája a fa azonos szintjén helyezkedik el
    - Minden rekordot közel azonos idő elérni
- Az indexszerkezet kiegyensúlyozott
  - az indexlisták kihasználtsága is jó
    - minden lista a legelsőt kivéve a kapacitásának minimum a felét kihasználja

Miért jó a B+ fa?

- Rendezett tárolás
  - Nagy adathalmazokon hatékonyabb így a keresés és módosítás
- Szekvenciális elhelyezés – nem praktikus
  - A legtöbb beszúrás és törlés kapcsán a tároló jelentős részének újraírása válna szükségessé
- Láncolt lista használata
  - Keresés nem hatékony

Miért jó a B+ fa?

- Keresőfa (pl. AVL fák, vagy piros-fekete fák)
  - Hátrány: Ha az adatokat egy véletlen elérésű háttértáron, pl. egy mágneslemezen kívánjuk elhelyezni.
    - A mágneslemezek: egyszerre az adatok egy egész blokkját (512 byte vagy 4 KB) adatot mozgatja
    - Egy bináris keresőfa egy csúcsa ennek csak egy töredékét használná.
- B+ fa:
  - Jobban kihasználja a mágneslemez blokkjait

A fokszám megválasztása

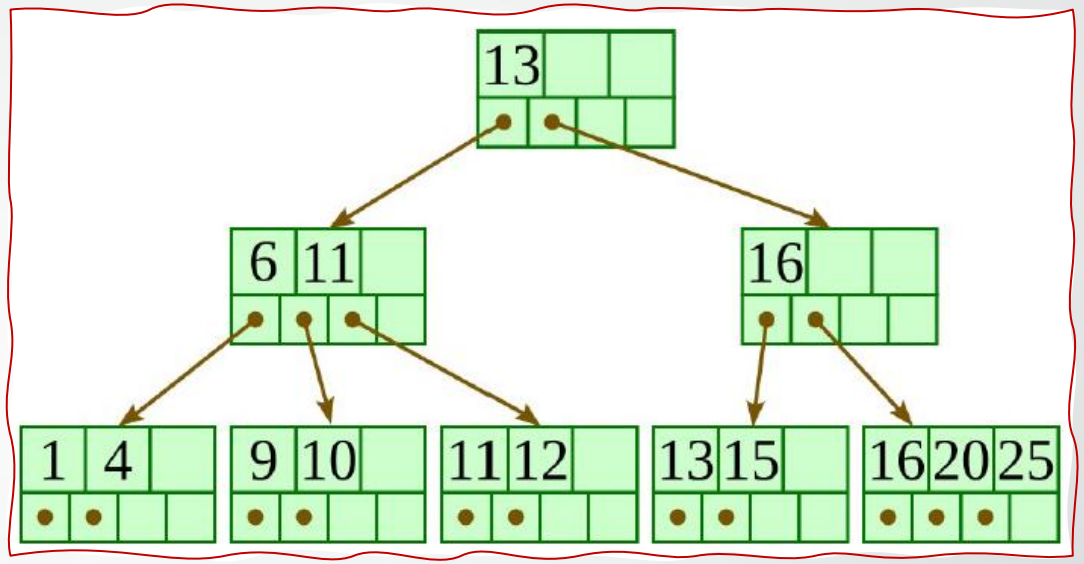
- Egy *d*-edfokú B+ fa csúcsaiban 4 bájtos kulcsok és 6 bájtos pointerek vannak. A B+ fát mágneslemezen tároljuk, ahol a blokkméret 4096 bájt. Mekkorának érdemes választani a B+ fa *d* fokszámát?
  - Egy csúcs legfeljebb *d*-1 kulcsot és *d* mutatót tartalmaz
    - $4(d\text{-}1) + 6d \leq 4096$

      $10d - 4 \leq 4096$

      $10d \leq 4100$

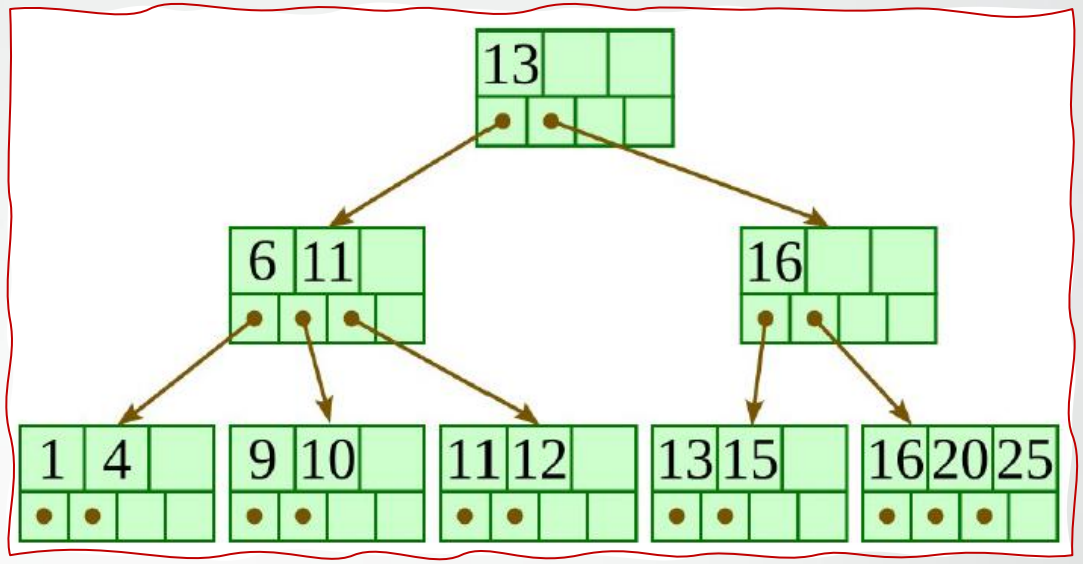
- *d* =410 –nek érdemes választani a B+ fa fokszámát

B+ fák



- Minden csúcs
  - legfeljebb d mutató ($4 \leq d$)
  - legfeljebb d-1 kulcs
  - ahol d a B+ fa fokszáma
    - fára jellemző állandó
- A belső csúcsok: hasító kulcsok (split key)
  - mindegyik referencia két kulcs "között" van:
    - egy olyan részfa gyökerére mutat, amiben minden érték a két kulcs között található
    - (mindegyik csúcshoz hozzáképzelve balról egy "mínusz végtelen", jobbról egy "plusz végtelen" értékű kulcsot)
    - $k_i \leq k < k_{i+1}$
- Az adatok a levélszinten vannak
  - A levélszinten minden kulcshoz tartozik egy mutató, ami a megfelelő adatrekordra hivatkozik. (A leveleket a d-edik mutatókkal gyakran listába fűzik.)
  - A gyökércsúcstól mindegyik levél azonos távolságra van

A B+ fa szöveges (zárójeles) ábrázolása



- Hierarchikus jelölésmód:
  - A fa szintjeit különböző zárójelekkel különítjük el a könnyebb olvashatóság érdekében:
- Strukturális felépítés:
  - Belső csúcs: (*T1*, *k1*, *T2*, *k2*, *T3*), ahol *Tj* a részfa, *kj* pedig a hasító kulcs.
  - Levél: (*k1*, *k2*, *k3*), ahol a kulcsok sorrendben szerepelnek.
- Példa (negyedfokú fa): { [(1, 4) 6 (9, 10) 11 (11, 12)] 13 [(13, 15) 16 (16, 20, 25)] }
  - A 13-as kulcs a gyökérben választja el a két fő részfát
  - A baloldali részfa két hasító kulcsa: 6, 11
  - A jobboldali részfa hasító kulcs: 16

B+ fa műveletidő

- Keresés:
  - A belső csúcsokban található *hasító kulcsok* segítségével
  - A csúcsokban is logaritmikusan keresve, és mindig a megfelelő ágon tovább haladva
  - O(lg *n*) lépésben (ahol *n* a B+ fával ábrázolt adathalmaz mérete)
- A hasító kulcsok nem feltétlenül szerepelnek a levelekben.
- A fa magassága: O(log *n*)
  - A gyakorlatban a fa *h* magassága az log *n* érték töredéke:
    - $\log_d(n/(d-1)) \leq h \leq \log_{\lfloor d/2 \rfloor}(n/2)$

A B+ fa magasságának (*h*) elméleti korlátai*

- A fa *h* magassága (a gyökértől a levelekig vezető élek száma)
  - *n* elemű adathalmaz és a *d* fokszám függvényében:

- Alsó becslés (Optimális eset – teli csúcsok)
  - Legalacsonyabb a fa:
    - ha minden csúcs maximálisan ki van használva
      - Ekkor minden belső csúcsnak *d* gyereke van
      - minden csúcs maximális számú kulcsot tartalmaz (legfeljebb $d-1$)
  - Az elemek száma ekkor:
    - A csúcsok száma:
      - $1 + d + d^2 + \cdots + d^h = \frac{d^{h+1}-1}{d-1}$
    - Ha minden csúcsban $d-1$kulcs van, akkor:
      - $n = (d-1) \cdot \frac{d^{h+1}-1}{d-1} = d^{h+1} - 1$

A B+ fa magasságának (h) elméleti korlátai*

$$n = d^{h+1} - 1$$

- Innen:

$$n < d^{h+1}$$

- Osszuk $d - 1$-gyel:

$$\frac{n}{d-1} < \frac{d^{h+1}}{d-1}$$

- Mivel $\frac{d^{h+1}}{d-1} \leq d^{h+1}$, ezért:

$$\frac{n}{d-1} < d^{h+1}$$

- Logaritmizálva:

$$\log_d\left(\frac{n}{d-1}\right) < h + 1$$

- Innen:

$$\boxed{\log_d\left(\frac{n}{d-1}\right) \leq h}$$

A B+ fa magasságának (*h*) elméleti korlátai*

- Felső becslés (Legrosszabb eset – minimális kitöltöttség)
  - Legmagasabb a fa:
    - ha minden csúcs csak a minimumot tartalmazza
      - A gyökérnek legalább 2 gyereke van
      - Minden más belső csúcsnak legalább floor($d/2$) gyereke
      - Minden levélnek legalább floor($d/2$) kulcsa van
  - Az elemek száma ekkor legalább: $n \geq 2 \cdot (\text{floor}(d/2))^{h-1} \cdot \text{floor}(d/2)$.
    - Ebből adódik: $h \leq \log_{\text{floor}(d/2)}(n/2)$

B+ fa műveletidő a gyakorlatban

- Példa:
  - d=410 értékkel:
    - $\log[410](n/409) \leq h \leq \log[205](n/2)$
  - n=1.000.000.000 -> 2,4 < h < 3,8 -> h=3
    - a fának pontosan négy szintje van
  - Egy-egy adat eléréséhez összesen maximum 3-szor olvasunk a lemezről
    - A felső két szintet az adatbázis megnyitásakor betöltjük a központi tárba
    - A levélszintről még egyet lépünk a tényleges adatrekord eléréséhez.
    - Ha a keresett kulcsú rekord nincs az adatbázisban: csak kétszer olvasunk a lemezről.

d-ed fokú B+ fa invariánsai ($4 \leq d$ állandó)

- Minden levélben legfeljebb $d-1$ kulcs, és ugyanennyi, a megfelelő (azaz ilyen kulcsú) adatrekordra hivatkozó mutató található.
- A gyökértől mindegyik levél ugyanolyan távol található.
  (Más szavakkal, minden levél azonos mélységben, a legalsó szinten van.)
- Minden belső csúcsban eggyel több mutató van, mint kulcs, ahol d a felső határ a mutatók számára.

d-ed fokú B+ fa invariánsai ($4 \leq d$ állandó)

- Minden *Cs* belső csúcsra, ahol k a *Cs* csúcsban a kulcsok száma: az első gyerekhez tartozó részfában minden kulcs kisebb, mint a *Cs* első kulcsa; az utolsó gyerekhez tartozó részfában minden kulcs nagyobb-egyenlő, mint a *Cs* utolsó kulcsa; és az *i*-edik gyerekhez tartozó részfában ($2 \leq i \leq k$) lévő tetszőleges *r* kulcsra $Cs.kulcs[i-1] \leq r < Cs.kulcs[i]$.
- A gyökércsúcsnak legalább két gyereke van (kivéve, ha ez a fa egyetlen csúcsa, következésképpen az egyetlen levele is).

d-ed fokú B+ fa invariánsai ($4 \leq d$ állandó)

- Minden, a gyökértől különböző belső csúcsnak legalább $\lfloor d/2 \rfloor$ gyereke van.
- Minden levél legalább $\lfloor d/2 \rfloor$ kulcsot tartalmaz (kivéve, ha a fának egyetlen csúcsa van).
- A B+ fa által reprezentált adathalmaz minden kulcsa megjelenik valamelyik levélben, balról jobbra szigorúan monoton növekvő sorrendben.

B+ fa beszúrás

- Ha a fa üres
  - új levélcsúcs létrehozása (gyökércsúcs)
  - beszúrjuk a kulcs/mutató párt
- Különben keressük meg a kulcsnak megfelelő levelet
- Ha a levélben már szerepel a kulcs,
  - a beszúrás sikertelen.
- Különben menjünk az 1. pontra!

B+ fa beszúrás

1. Ha a csúcsban van üres hely
   - szúrjuk be a megfelelő kulcs/mutató párt kulcs szerint rendezetten ebbe a csúcsba!
2. Ha a csúcs már tele van
   - vágjuk szét két csúccsá
   - osszuk el a *d* darab kulcsot egyenlően a két csúcs között!

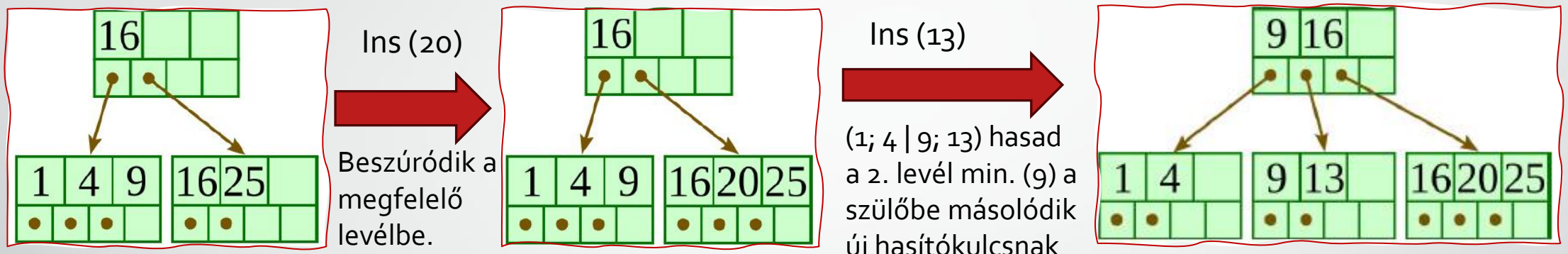
B+ fa beszúrás

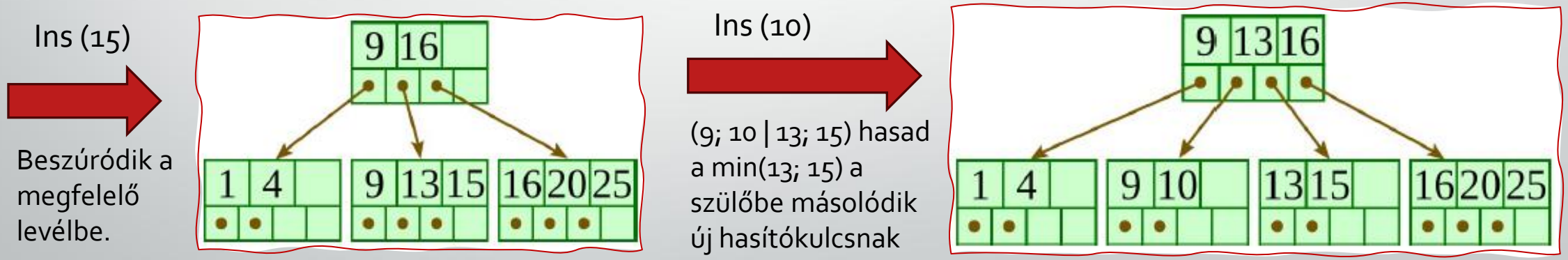
- Ha a csúcs egy levél,
  - vegyük a második csúcs legkisebb értékének másolatát
  - és ismételjük meg ezt a beszúró algoritmust, hogy beszúrjuk azt a szülő csúcsba!
- Ha a csúcs nem levél,
  - vegyük ki a középső értéket a kulcsok elosztása során,
  - és ismételjük meg ezt a beszúró algoritmust, hogy beszúrjuk ezt a középső értéket a szülő csúcsba!
- (Ha kell, a szülő csúcsot előbb létrehozzuk. Ekkor a B+ fa magassága nő.)

B+ fa műveletek:

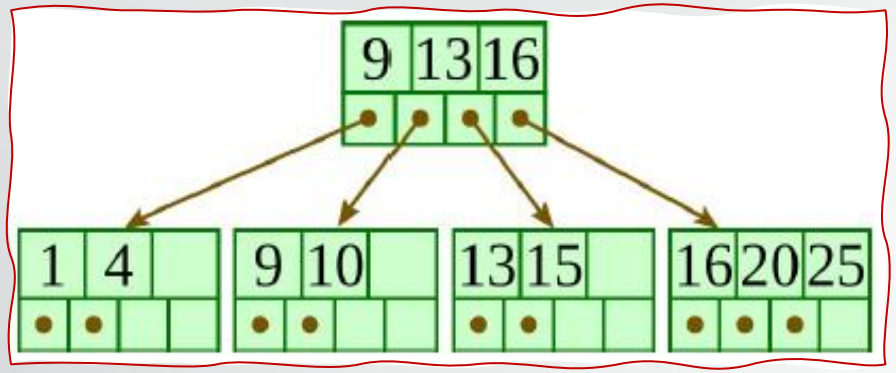
- Beszúrás, törlés
- Konkrét algoritmust nem tárgyalunk
- A példákban általában d=4 értékkel dolgozunk
  - A fenti invariánsok alapján:
  - Minden levél legalább két kulcsot tartalmaz
  - Minden belső csúcsnak legalább két gyereke és legalább egy hasító kulcsa van

B+ fa beszúrás példa



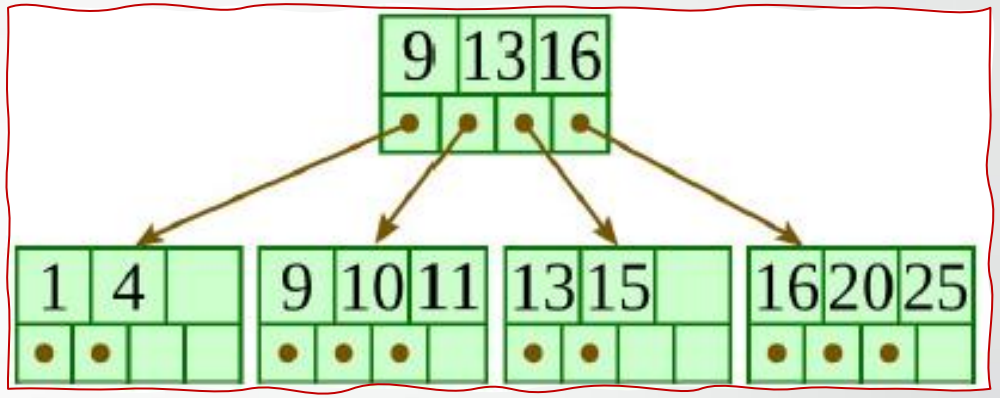


B+ fa beszúrás példa



Ins (11)

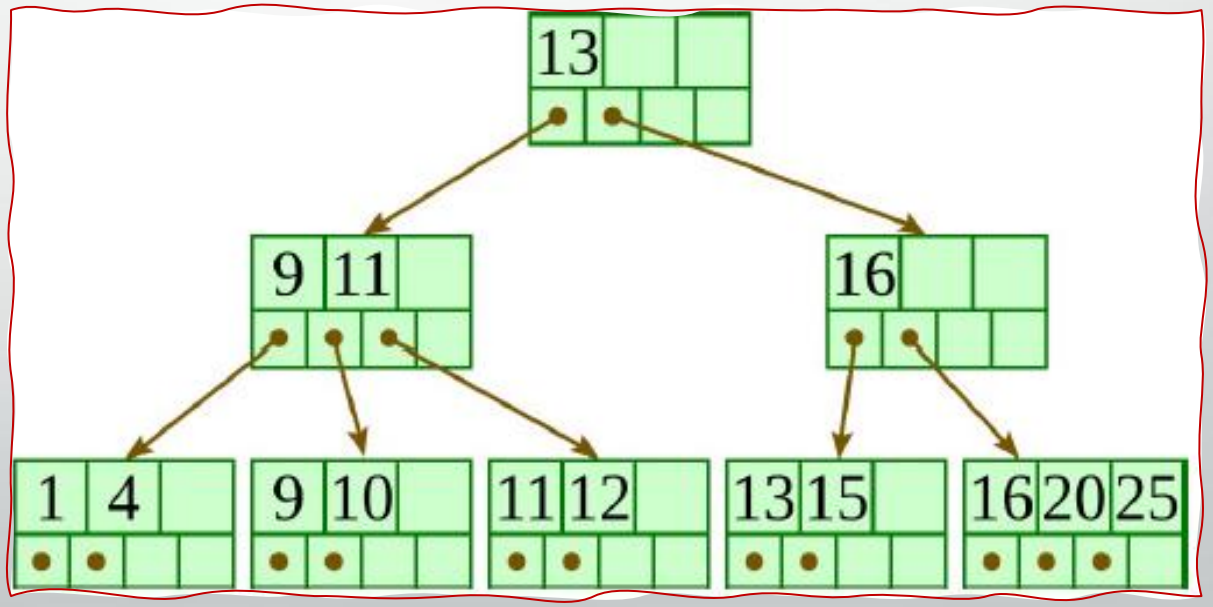
Beszúródik a megfelelő levélbe.



Ins (12)

(9; 10 | 11; 12) hasad a min(11; 12) a szülőbe másolódik új hasítókulcsnak

(9; 11 | 13; 16) hasad a min(13; 16) felmegy az újonnan létrehozott gyökérbe



B+ fa törlés

- Keressük meg a törlendő kulcsot tartalmazó levelet.
  - Ha ilyen nincs, a törlés meghiúsul
- Különben
  - a törlő algoritmus futása
    - vagy az A esettel fejeződik be;
    - vagy a B esettel folytatódik,
    - ami után a C eset (nullaszor, egyszer, vagy többször) ismétlődhet,
    - és még a D eset is sorra kerülhet végül.

B+ fa törlés: A eset

A. A keresés során megtalált levélcsúcs egyben a gyökércsúcs is:

- Töröljük a megfelelő kulcsot és a hozzá tartozó mutatót a csúcsból.
- Ha a csúcs tartalmaz még kulcsot
  - kész vagyunk.
- Különben töröljük a fa egyetlen csúcsát és üres fát kapunk.

B+ fa törlés: B eset folytatás

B. A keresés során megtalált levélcsúcs nem a gyökércsúcs:

- Töröljük a megfelelő kulcsot és a hozzá tartozó mutatót a csúcsból.
- Ha a csúcs még tartalmaz elég kulcsot és mutatót, hogy teljesítse az invariánsokat
  - ➢ kész vagyunk.

B+ fa törlés: B eset folytatás

- Ha a levélcsúcsban már túl kevés kulcs van ahhoz, hogy teljesítse az invariánsokat, de a következő, vagy a megelőző testvérének több van, mint amennyi szükséges:
  - Osszuk el a kulcsokat közte és a megfelelő testvére között.
  - Javítsuk ki a testvérek szülőjében a két testvérhez tartozó hasító kulcsot **két testvér közül a második (jobb oldali) legkisebb elemére** (minimumára)

B+ fa törlés: B eset folytatás

- Ha a levélcsúcsban már túl kevés kulcs van ahhoz, hogy teljesítse az invariánst, és a következő, valamint a megelőző testvére is a minimumon van, hogy teljesítse az invariánst:
  - egyesítsük egy vele szomszédos testvérével

B+ fa törlés B eset: testvérek egyesítése

- Testvérek egyesítése
  - Ennek során a két testvér közül a (balról jobbra sorrend szerinti) másodikból a kulcsokat és a hozzájuk tartozó mutatókat sorban átmásoljuk az elsőbe, annak eredeti kulcsai és mutatói után
  - Majd a második testvért töröljük
  - Ezután meg kell ismételnünk a törlő algoritmust a szülőre, hogy eltávolítsuk a szülőből a hasító kulcsot (ami eddig elválasztotta a most egyesített levélcsúcsokat), a most törölt második testvérre hivatkozó mutatóval együtt.

B+ fa törlés: C eset

C. Belső — a gyökértől különböző — csúcsból való törlés:

- Töröljük a belső csúcs éppen most egyesített két gyereke közti hasító kulcsot és az egyesítés során törölt gyerekére hivatkozó mutatót a belső csúcsból!

- Ha a belső csúcsnak van még $\lfloor d/2 \rfloor$ gyereke, (hogy teljesítse az invariánsokat)
  - kész vagyunk.

- Ha a belső csúcsnak már túl kevés gyereke van ahhoz, hogy teljesítse az invariánsokat, de a következő, vagy a megelőző testvérének több van, mint amennyi szükséges:
  - osszuk el a gyerekeket és a köztük levő hasító kulcsokat egyenlően közte és a megfelelő testvére között, a hasító kulcsok közé a testvérek közti (a közös szülőjükben lévő) hasító kulcsot is beleértve!

B+ fa törlés: C eset

- A gyerekek és a hasító kulcsok újraelosztása során, a középső hasító kulcs a testvérek közös szülőjében a két testvérhez tartozó régi hasító kulcs helyére kerül úgy, hogy megfelelően reprezentálja a köztük megváltozott vágási pontot!
- (Ha a két testvérben a gyerekek összlétszáma páratlan, akkor az újraelosztás után is annak a testvérnek legyen több gyereke, akinek előtte is több volt!)

- Ha a belső csúcsnak már túl kevés gyereke van ahhoz, hogy teljesítse az invariánst, és a következő, valamint a megelőző testvére is a minimumon van, hogy teljesítse az invariánst:
  - egyesítsük egy vele szomszédos testvérével!
  - Az egyesített csúcsot a két testvér közül a (balról jobbra sorrend szerinti) elsőből hozzuk létre

B+ fa törlés

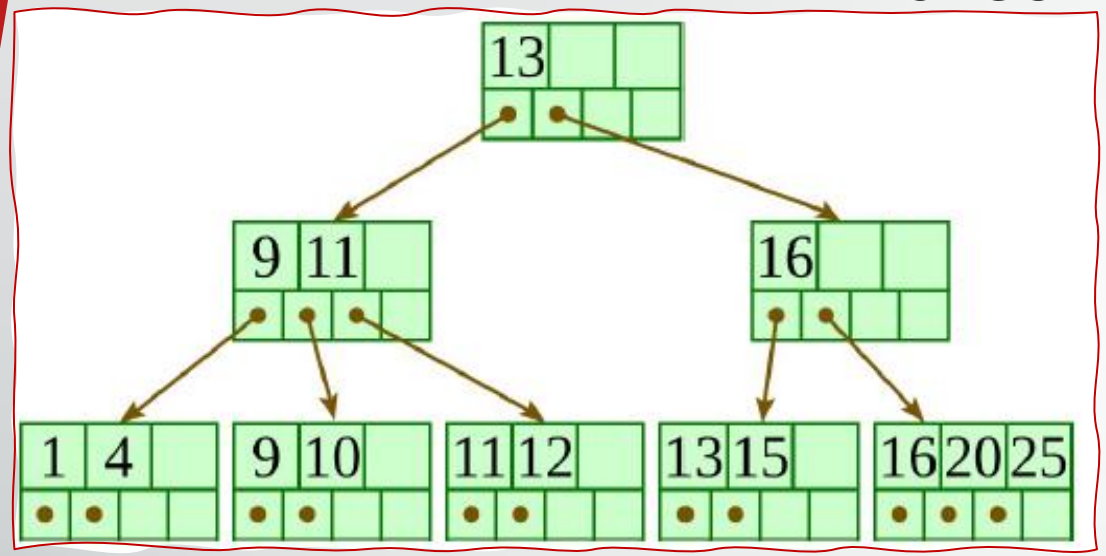
- Gyerekei és hasító kulcsai:
  - először a saját gyerekei és hasító kulcsai az eredeti sorrendben,
  - amiket a két testvér közti (a közös szülőjükben lévő) hasító kulcs követ,
  - és végül a második testvér gyerekei és hasító kulcsai jönnek, szintén az eredeti sorrendben.
- Ezután töröljük a második testvért.
- A két testvér egyesítése után meg kell ismételnünk a törlő algoritmust a közös szülőjükre,
  - hogy eltávolítsuk a szülőből a hasító kulcsot (ami eddig elválasztotta a most egyesített testvéreket), a most törölt második testvérre hivatkozó mutatóval együtt.

B+ fa törlés: D eset

D. A gyökércsúcsból való törlés, ha az nem levélcsúcs

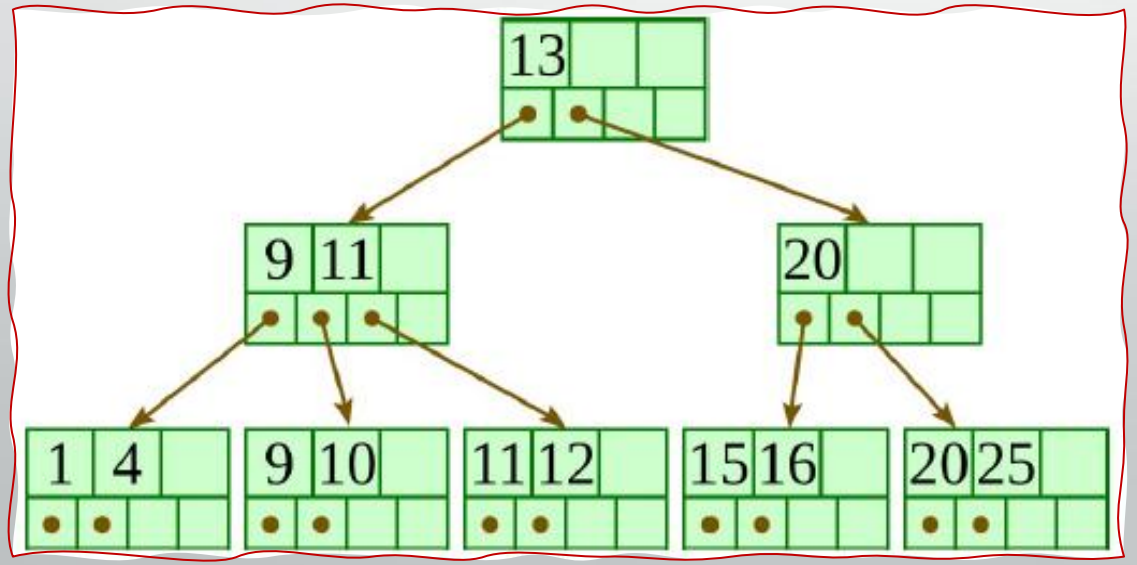
- Töröljük a gyökércsúcs éppen most egyesített két gyereke közti hasító kulcsot és az egyesítés során törölt gyerekére hivatkozó mutatót a gyökércsúcsból!
- Ha a gyökércsúcsnak van még 2 gyereke
  - kész vagyunk.
- Ha a gyökércsúcsnak csak 1 gyereke maradt
  - töröljük a gyökércsúcsot
  - és a megmaradt egyetlen gyereke legyen az új gyökércsúcs!
  - (Ekkor a B+ fa magassága csökken.)

B+ fa törlés példa



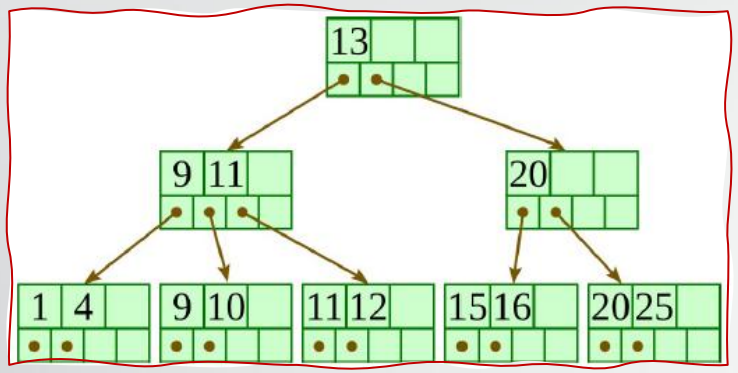
Del (13)

A 15 egyedül marad,
A 16-ot átveszi a jobboldali testvérétől,
Majd a szülőjükben átíródik a megfelelő hasító kulcs.



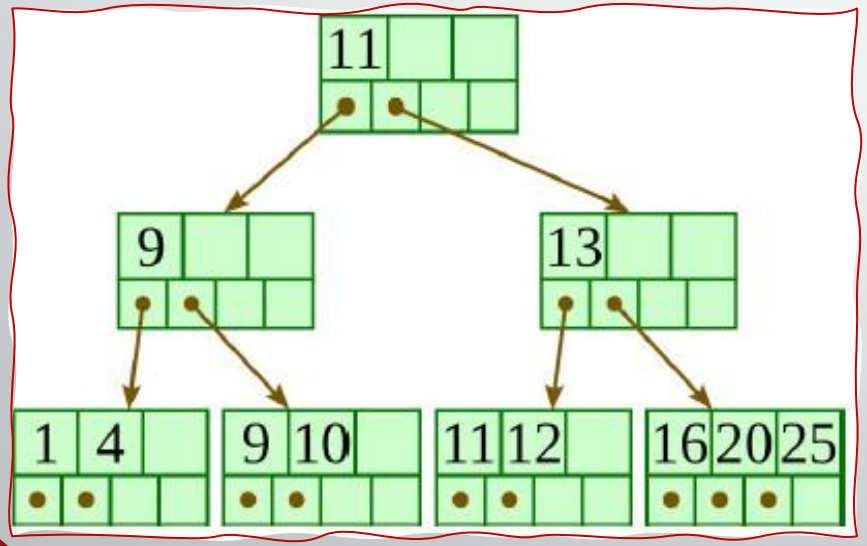
Del (15)

B+ fa törlés példa



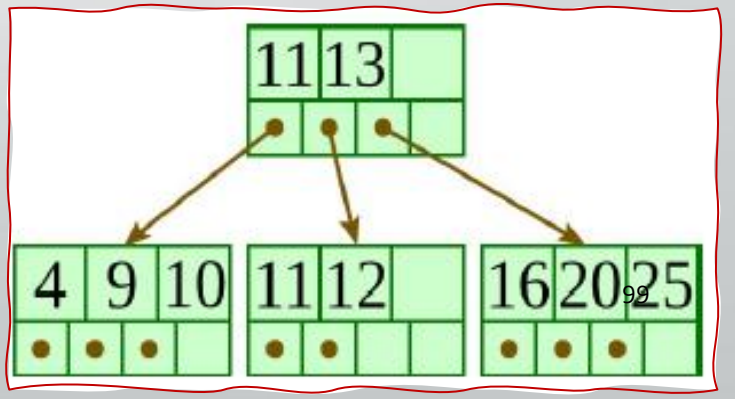
Del (15)

A két testvér levél egyesül,
A 20 hasító kulcs törlődik a szülőből, aminek 1 gyereke marad,
és egyet átvesz a testvérétől;
A 13 a nagyszülőből lejön, és a 11 a 13 helyére megy.



A (4; 9; 10) egyesül,
A szülők egyesülnek,
11 lejön az egyesült szülőbe,
A gyökérnek 1 gyereke marad: törlődik.

Del (1)

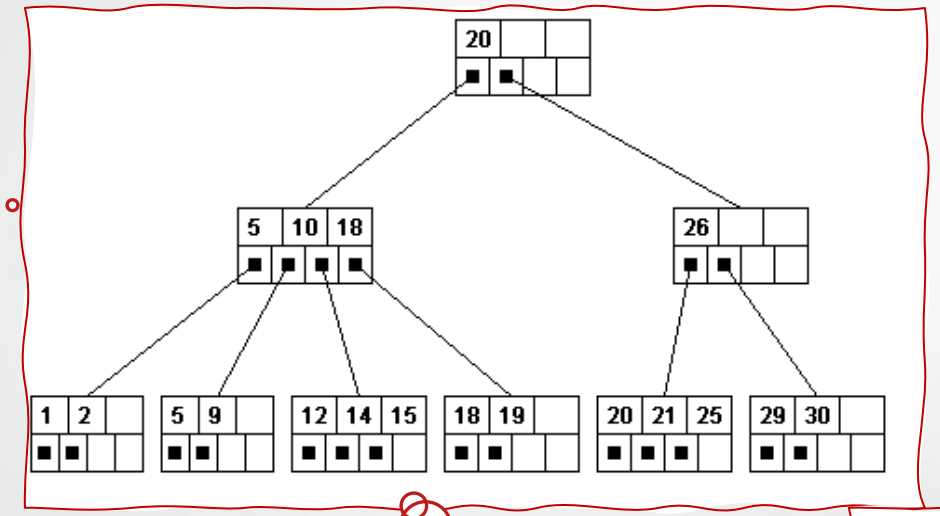


Ellenőrző kérdések

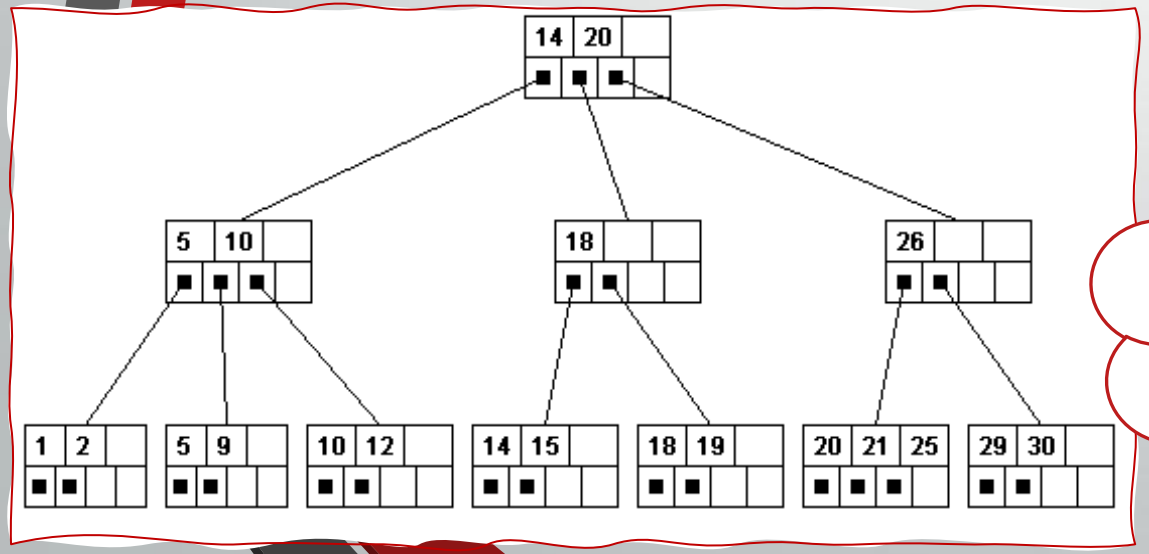
1. A *d*-edfokú B+ fák **belső** csúcsainak milyen tulajdonságát ismeri?
2. A *d*-edfokú B+ fák **levél** csúcsainak milyen tulajdonságát ismeri?
3. Adott a { [(1 2) 5 (5 9) 10 (12 14 15) 18 (18 19)] 20 [(20 21 25) 26 (29 30)] } negyedfokú B+fa.
   - Rajzolja le a fát!
   - Szemléltessünk a 10 beszúrását, valamint a 2 és a 26 törlését, **mindhárom esetben az eredeti fára**!
4. Tegyük fel, hogy egy *d*-edfokú B+ fában egy *n* méretű kulcshalmazt tároltunk! Adjon alsó és felső becslést a fa *h* magasságára!

3. Kérdés megoldása

- A { [(1 2) 5 (5 9) 10 (12 14 15) 18 (18 19)] 20 [(20 21 25) 26 (29 30)] } fa képe:

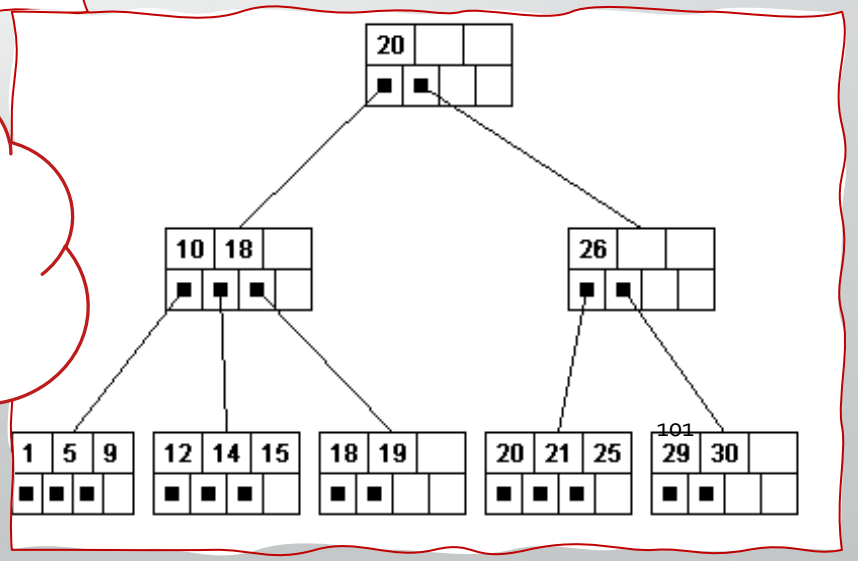


- 10 beszúrása



- 26 törlése:
  - Mivel 26 hasító kulcs, nem lehet törölni.
  - Nem változik a fa

- 2 törlése:



Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek II. Előadásjegyzete (B+ Fák)
és Fekete István: Algoritmusok és adatszerkezetek / B-Fák
előadásjegyzete alapján készült.

Algoritmusok és adatszerkezetek I.
4. Előadás

Gráf ábrázolások: Szomszédossági csúcsmátrix és éllista, ezek tárigénye. Szélességi gráfkeresés.

Tartalom

Gráfelméleti alapfogalmak

- Gráfok használata
  - Pl. hálózatokat, folyamatokat modellezhetünk

1. Definíció. **Gráf** alatt egy $G = (V, E)$ rendezett párost értünk, ahol

- V a csúcsok (vertices) tetszőleges, véges halmaza,
- $E \subseteq V \times V \setminus \{(u; v) : u \in V\}$ pedig az élek (edges) halmaza.
- Ha $V = \{\}$, akkor üres gráfról,
- ha $V \neq \{\}$, akkor nemüres gráfról beszélünk.

Gráfelméleti alapfogalmak

- Megjegyzés:
  - Már a definíció szintjén kizárjuk a gráfokból párhuzamos éleket és a hurokéleket.
    - Nincs ugyanis semmilyen eszközünk arra, hogy két $(u; v)$ élet megkülönböztessünk (párhuzamos élek),
    - az $(u; u)$ alakú, ún. hurokéleket pedig expliciten kizártuk
  - ➢ Így a továbbiakban gráf alatt tulajdonképpen **egyszerű gráfot** értünk.

2. Definíció. A $G = (V, E)$ gráf **irányítatlan**, ha tetszőleges $(u, v) \in E$ élre $(u, v) = (v, u)$.

3. Definíció. A $G = (V, E)$ gráf **irányított**, ha tetszőleges $(u, v), (v, u) \in E$ élpárra $(u, v) \neq (v, u)$.

- Ilyenkor azt mondjuk, hogy az (u, v) él fordítottja a (v, u) él, és viszont.

Gráfelméleti alapfogalmak

- Megjegyzés:
  - Az irányítatlan gráfoknál tetszőleges (u, v) éllel együtt (v, u) is a gráf éle, hiszen ez a két él egyenlő.
  - Irányított gráfoknál általában lesz a gráfnak olyan (u, v) éle, hogy ennek fordítottja (v, u) nem éle a gráfnak.

4. Definíció. A $G = (V, E)$ gráf csúcsainak (V) egy $\langle u_0, u_1, \ldots u_n \rangle$ ($n \in \mathbb{N}$) sorozata a gráf egy **út**ja, ha tetszőleges $i \in 1..n$-re $(u_{i-1}, u_i) \in E$.

- Ezek az (u_{i-1}, u_i) élek az út élei.
- Az út hossza ilyenkor n, azaz az utat alkotó élek számával egyenlő.

Gráfelméleti alapfogalmak

5. Definíció. Tetszőleges $\langle u_0, u_1, \ldots u_n \rangle$ út **rész-útja** $0 \leq i \leq j \leq n$ esetén az $\langle u_i, u_{i+1}, \ldots u_j \rangle$ út

- A **kör:** olyan út
  - kezdő és végpontja (csúcsa) azonos
  - a hossza > 0
  - az élei páronként különbözőek
- Az **egyszerű kör:** olyan kör
  - csak a kezdő és a végpontja azonos

Gráfelméleti alapfogalmak

- Tetszőleges **út akkor tartalmaz kört**
  - ha van olyan rész-útja, ami kör
- **Körmentes út:** olyan út
  - ami nem tartalmaz kört
- **Körmentes gráf** : olyan gráf
  - amiben csak körmentes utak vannak

- Megjegyzés:
  - Az utak köröket is tartalmazhatnak! A fentiek szerint tetszőleges kör hossza ≥ 2.

Gráfelméleti alapfogalmak

6. Definíció. **DAG** alatt irányított, körmentes gráfot értünk (directed acyclic graph).

- A DAG-ok modellezhetnek például összetett folyamatokat, ahol a gráf csúcsai elemi műveletek, az élei pedig az ezek közötti rákövetkezési kényszerek.

7. Definíció. Tetszőleges $G = (V, E)$ **irányított gráf irányítatlan megfelelője** az a $G' = (V, E')$ irányítatlan gráf, amire $E' = \{(u, v) : (u, v) \in E \vee (v, u) \in E\}$.

Gráfelméleti alapfogalmak

8. Definíció. A G **irányítatlan gráf összefüggő**, ha G tetszőleges csúcsából bármelyik csúcsába vezet út.
A G **irányított gráf összefüggő**, ha az irányítatlan megfelelője összefüggő.

9. Definíció. Az irányítatlan, körmentes, összefüggő gráfokat szabad fáknak, más néven **irányítatlan fák**nak nevezzük.

10. Definíció. Az u csúcs a G irányított gráf **generátor csúcsa**, ha u-ból a G tetszőleges v csúcsa elérhető, azaz létezik $u \rightsquigarrow v$ út.

Gráfelméleti alapfogalmak

11. Tulajdonság. Ha a *G* irányított gráfnak van generátor csúcsa, akkor összefüggő,
de fordítva nem igaz az állítás.

12. Definíció. *T* **gyökeres fa**, más néven **irányított fa**, ha *T* olyan irányított gráf, aminek van generátor csúcsa, és a *T* irányítatlan megfelelője körmentes.
Ilyenkor a generátor csúcsot a fa gyökér csúcsának is nevezzük.

13. Tulajdonság. Tetszőleges (gyökeres vagy szabad) nemüres fának pontosan eggyel kevesebb éle van, mint ahány csúcsa.

Gráfelméleti alapfogalmak

14. Definíció. A $G = (V, E)$ gráfnak **részgráf**ja a $G' = (V', E')$ gráf, ha $V' \subseteq V \wedge E' \subseteq E$, és mindkét gráf irányított, vagy mindkettő irányítatlan.
A G gráfnak **valódi részgráfja** a G' gráf, ha G-nek részgráfja G', de $G \neq G'$.

15. Definíció. Két (rész)gráf **diszjunkt**,

- ha nincs közös csúcsuk (és ebből következően közös élük sem)

16. Definíció. A G gráf **összefüggő komponense** a G' gráf,

- ha G-nek részgráfja G' és G' összefüggő
- de G-nek nincs olyan összefüggő részgráfja, aminek G' valódi részgráfja.

Gráfelméleti alapfogalmak

15. Tulajdonság. Tetszőleges gráf vagy összefüggő vagy felbontható (egymástól diszjunkt) összefüggő komponensekre (amelyek együtt kiadják a teljes gráfot)

16. Definíció. A *G* gráf **erdő**, ha összefüggő komponensei fák (vagy egyetlen fából áll).

17. Tulajdonság. A *G* irányítatlan gráf erdő $\Leftrightarrow$ *G* körmentes.
A *G* irányított gráf erdő $\Leftrightarrow$ *G* irányítatlan megfelelője körmentes, és *G* mindegyik összefüggő komponensének van generátor csúcsa.

Gráfábrázolás

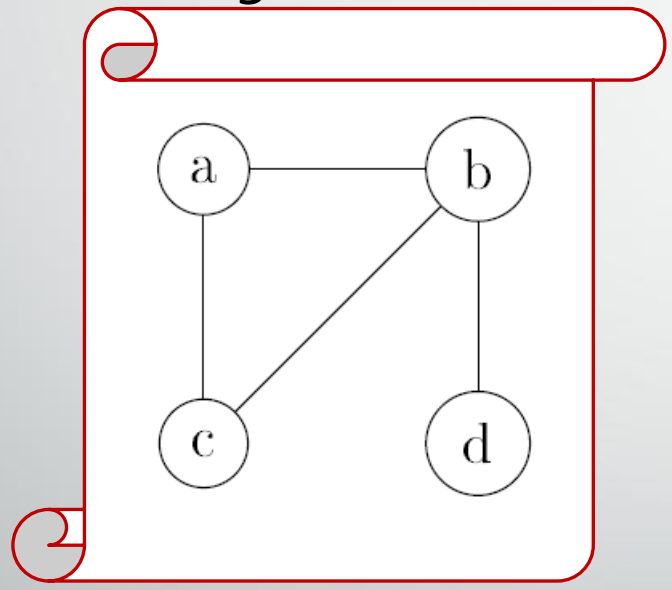
- Jelölések
  - $G = (V, E)$ gráfról általában föltesszük, hogy $V = \{v_1, \ldots, v_n\}$, ahol $n \in \mathbb{N}$, azaz hogy a gráf csúcsait egyértelműen azonosítják az $1..n$ sorszámok.
  - a csúcsok sorszámait a szemléletesség kedvéért gyakran az angol ábécé kisbetűivel jelöljük
- Grafikus ábrázolás
  - Csúcsok: kis körök
  - Élek
    - irányított gráfoknál: a körök közti nyilak
    - irányítatlan esetben: a köröket összekötő vonalak
  - A csúcsok sorszámát (illetve az azt reprezentáló betűt) általában a körökbe írjuk

Gráfábrázolás

- Szöveges ábrázolás
  - irányítatlan gráfoknál: „u - v_{u1}; . . . v_{uk}.” jelentése:
    - a gráfban az u csúcsnak szomszédai: v_{u1}; . . . v_{uk} csúcsok
    - Azaz: (u, v_{u1}), . . .; (u, v_{uk}) élei a gráfnak.
  - irányított gráfoknál: „u -> v_{u1}; . . . v_{uk}.” jelentése:
    - a gráfban az u csúcsból (u, v_{u1}), . . .; (u, v_{uk}) irányított élek indulnak ki
    - Azaz: az u csúcs rákövetkezői (gyerekei): v_{u1}; . . . v_{uk} csúcsok

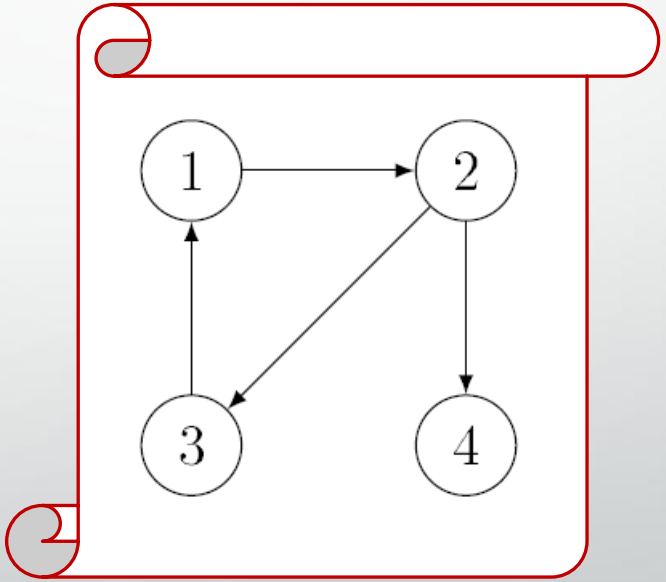
Gráfábrázolás példa

- Irányítatlan gráf grafikus és szöveges ábrázolása



a – b ; c.
b – c ; d.

- Irányított gráf grafikus és szöveges ábrázolása

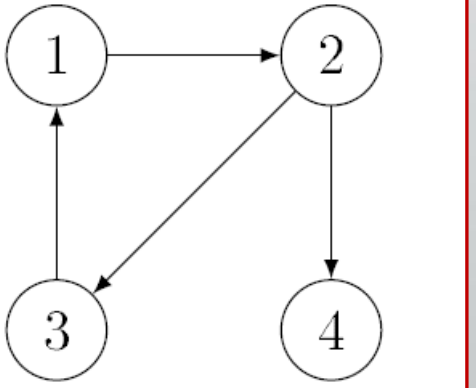


1 → 2.
2 → 3 ; 4.
3 → 1.

Szomszédossági mátrixos (adjacency matrix), más néven csúcsmátrixos reprezentáció

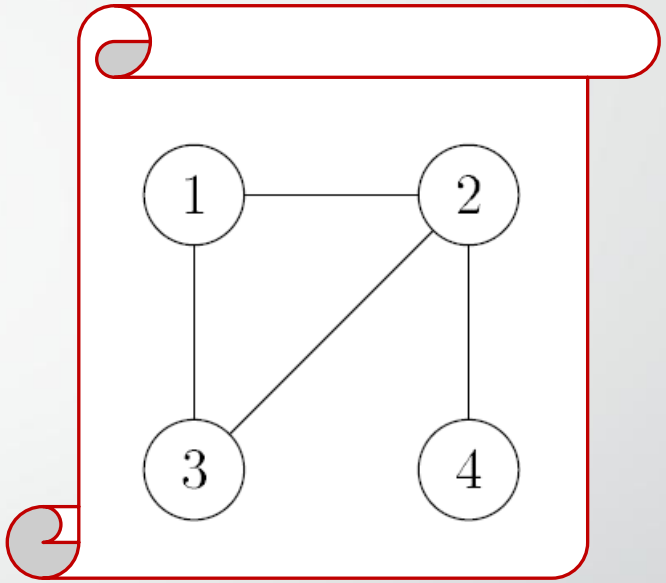
- A $G = (V, E)$ gráfot ($V = \{v_1, \ldots, v_n\}$) egy ***A*/1 : *bit*[*n*, *n*]** mátrix reprezentálja, ahol:
  - $n = |V|$ a csúcsok száma
  - $1..n$ a csúcsok sorszámai, azaz azonosító indexei,
  - **type** *bit* **is** {0, 1}
  - tetszőleges $i, j \in 1..n$ csúcssorszámokra
    - $A[i, j] = 1 \Leftrightarrow (v_i, v_j) \in E$
    - $A[i, j] = 0 \Leftrightarrow (v_i, v_j) \notin E$

A	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	1	0	0	0
4	0	0	0	0



Szomszédossági mátrixos reprezentáció

- A főátlókban mindig nullák vannak
  - csak egyszerű gráfokkal foglalkozunk (nincsenek hurokélek)
- Irányítatlan esetben a szomszédossági mátrixos reprezentáció mindig szimmetrikus
  - $(v_i, v_j) \in E \Leftrightarrow (v_j, v_i) \in E$
  - $\forall\ v_i$ és v_j csúcsokra $A[i,j] = A[j,i]$
- Elég alsóháromszög (felsőháromszög) mátrixban ábrázolni
  - Főátló nélkül
  - Pl. sorfolytonosan



A	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	1	1	0	0
4	0	1	0	0

Mátrix- tömb megfeleltetés

- Helyfoglalás:
  - 2. sor 1 elem, 3. sor 2 elem, … utolsó sor (n - 1) elem.
  - ➢ n^2 bit helyett csak 1 + 2 + … + (n - 1) = n * (n - 1)/2 bit
- A/1 : $bit[n, n]$ mátrix -> B: $bit[n * (n - 1)/2]$ tömb
  - $a_{ij} = A[i, j]$ jelöléssel
  - $<a_{21}, a_{31}, a_{32}, a_{41}, a_{42}, a_{43}, \dots, a_{n1}, \dots, a_{n(n-1)}>$
- Innét
  - $A[i, j] = B[(i-1)*(i-2)/2+(j-1)]$ ha $i > j$ (az alsóháromszög mátrixban)
  - $A[i, j] = A[j, i]$ ha $i < j$ ($A[i, j]$ a felsőháromszög mátrixban)
  - $A[i, i] = 0$ ($A[i, i]$ a főátlón van)

$a_{ij} = A[i, j]$ elem helyének meghatározása a B tömbben

- Azt kell megszámolnunk, hogy hány elem előzi meg sorfolytonosan az a_{ij} elemet a B tömbben
- a_{ij} indexe a B-ben = az a_{ij} -t megelőző elemek számával
  - Mivel a B tömböt nullától indexeljük
- Az alsóháromszög mátrixban az a_{ij} elemet az alábbi elemek előzik meg sorfolytonosan:

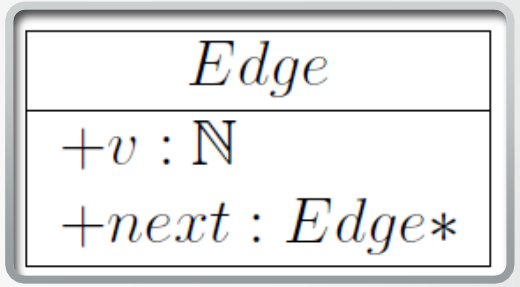
  a_{21},
  a_{31}, a_{32},
  a_{41}, a_{42}, a_{43},
  … ,
  $a_{(i-1)1}$, … , $a_{(i-1)\,(i-2)}$,
  a_{i1}, … , $a_{i(j-1)}$

- Ez pedig összesen $(1+2+3+\ldots+(i-2))+(j-1) = (i-1)*(i-2)/2+(j-1)$ elem

Műveletidő csúcsmátrixos ábrázolásnál

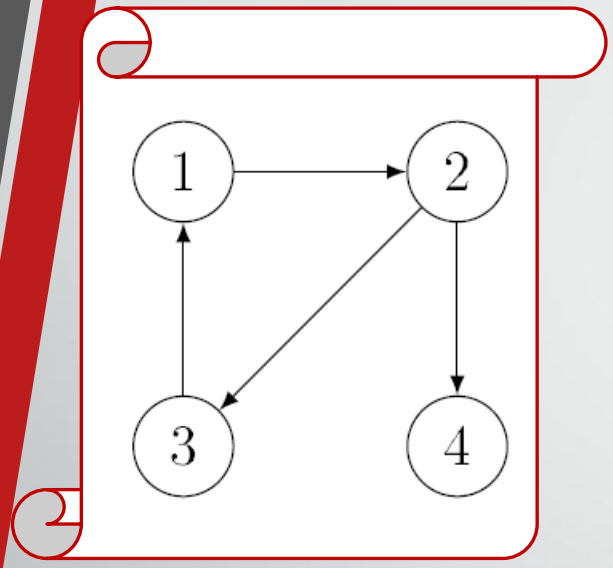
- $(v_i, v_j) \in E$ kérdés: $\Theta(1)$
  - Olyan algoritmusoknál előnyös, ahol gyakori ez a művelet
- Adott csúcs (irányított gráfoknál) gyerekeinek, vagy (irányítatlan gráfoknál) szomszédainak felsorolása: n lépés
  - Ez általában lényegesen több, mint ahány gyerek vagy szomszéd ténylegesen van

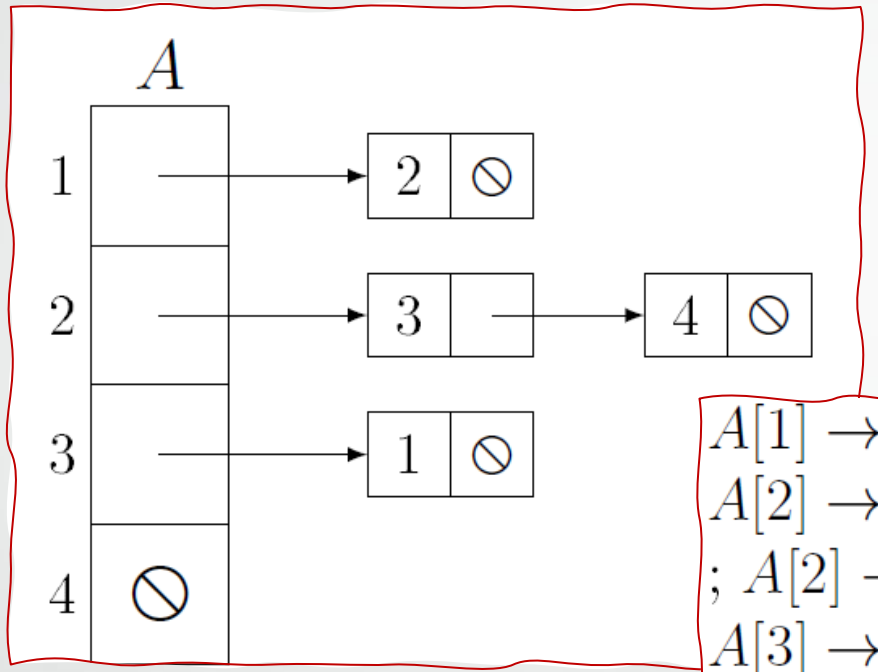
Szomszédossági listás (adjacency list) reprezentáció



- *A G* = (*V*, *E*) gráfot (*V* = {v_1, . . ., v_n}) ***A*[1 : *Edge**[*n*]** pointertömb reprezentálja, ahol:
  - Irányítatlan gráf esetében a v_i csúcs szomszédjainak sorszámait az *A*[*i*] S1L tartalmazza (i ∈1..*n*)
    - A v_i csúcs szomszédjainak indexeit tehát az *A*[*i*] lista elemeinek *v* attribútumai tartalmazzák
    - Így az *A*[*i*] lista elemei a v_i csúcshoz kapcsolódó éleknek felelnek meg
    - ➢ Minden élet kétszer ábrázolunk,
      - pl. a v_i csúcsnak szomszédja v_j -> a v_j csúcsnak is szomszédja v_i
  - Irányított gráfok esetén hasonló a reprezentáció, de az *A*[*i*] S1L csak az v_i csúcs gyerekeinek (más néven közvetlen rákövetkezőinek) sorszámait tartalmazza (i ∈1..*n*)
    - ➢ Mindegyik élet csak egyszer kell ábrázolnunk

Szomszédossági lista példa





$A[1] \to v = 2$; $A[1] \to next = \oslash$
$A[2] \to v = 3$; $A[2] \to next \to v = 4$
; $A[2] \to next \to next = \oslash$
$A[3] \to v = 1$; $A[3] \to next = \oslash$
$A[4] = \oslash$

- A szomszédossági listás ábrázolásnál S1L-ek helyett természetesen másfajta listákat is alkalmazhatunk.

Műveletidő szomszédossági listás ábrázolásnál

- $(v_i, v_j) \in E$ kérdés
  - meg kell keresnünk a j indexet az $A[i]$ listán
  - ➢ algoritmusoknál, ahol gyakori ez a művelet: érdemes a csúcsmátrixos reprezentációt választani
- Adott csúcs (irányított gráfoknál) gyerekeinek, vagy (irányítatlan gráfoknál) szomszédainak felsorolása
  - pontosan annyi lépésre van szükségünk, mint ahány gyerek vagy szomszéd ténylegesen van
  - A legtöbb gráfalgoritmusnak ez a leggyakoribb művelete
    - ➢ ezt a reprezentációt gyakran előnyben részesítjük a többi ábrázolással szemben.

Számítógépes gráfábrázolások tárigénye

- A $G = (V, E)$ gráfban:
  - csúcsok száma: $n = |V|$
  - élek száma: $m = |E|$
  - $0 \leq m \leq n * (n - 1) \leq n^2$

➢ $m \in O(n^2)$

- Gráfok osztályozása:
  - ritka gráf (sparse graph):
    - $m \in O(n)$
  - sűrű gráf (dense graph):
    - $m \in \Theta(n^2)$

Számítógépes gráfábrázolások tárigénye

- **Szomszédossági mátrixos** (csúcsmátrixos) ábrázolás tárigénye
  - Alapesetben n^2 bit
  - Irányítatlan gráfoknál, csak az alsóháromszög mátrixot tárolva:
    - $n * (n - 1)/2$ bit $\in \Theta(n^2)$ ->

➢ Az aszimptotikus tárigény mindkét esetben $\in \boldsymbol{\Theta(n^2)}$

Számítógépes gráfábrázolások tárigénye

- **Szomszédossági listás** reprezentációnál
  - A pointertömb *n* db mutatóból áll
  - A szomszédossági listáknak összesen *m* vagy 2 * *m* elemük van
    - (a gráf irányított vagy irányítatlan)

➢ Az aszimptotikus tárigény mindkét esetben $\Theta(n + m)$

Számítógépes gráfábrázolások tárigénye

Ritka gráfoknál

- (a gyakorlati alkalmazások többségénél)
- $m \in O(n) \rightarrow \Theta(n + m) = \Theta(n)$
  - A szomszédossági listás reprezentáció tárigénye aszimptotikusan kisebb, mint a csúcsmátrixosé

Sűrű gráfoknál

- $m \in \Theta(n^2) \rightarrow \Theta(n + m) = \Theta(n^2)$
  - szomszédossági listás és a csúcsmátrixos reprezentáció tárigénye aszimptotikusan ekvivalens

Számítógépes gráfábrázolások tárigénye

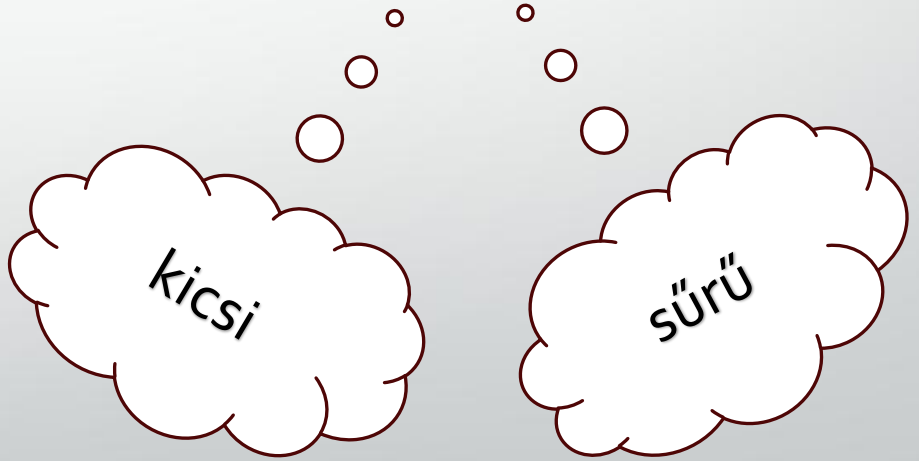
- Teljes gráfoknál
  - A szomszédossági listáknak összesen $n*(n-1)$ elemük van
  - Egy-egy listaelem sok bitből áll

➢ Teljes vagy közel teljes gráfoknál a szomszédossági listás ábrázolás tényleges tárigénye jelentősen nagyobb lehet, mint a csúcsmátrixosé, ahol a mátrix egy-egy eleme akár egyetlen biten is elfér.

Melyik ábrázolást válasszuk?

Szomszédossági lista

Szomszédossági mátrix



Az absztrakt halmaz, az absztrakt sorozat és az absztrakt gráf típus

- Típuskonstruktorok bevezetése: ($\mathcal{T}$ tetszőleges típus)
  - $\mathcal{T}\{\}$: $\mathcal{T}$ típusú elemek tetszőleges, véges halmaza
  - $\mathcal{T}<>$: $\mathcal{T}$ típusú elemek tetszőleges, véges sorozata
- *u* **from** *S* művelet: (*S* tetszőleges nemüres halmaz)
  - kiválasztjuk az *S* halmaz egy tetszőleges elemét,
  - *u*-nak értékül adjuk,
  - majd eltávolítjuk *S*-ből
- Az üres halmaz: önmagában álló {}

Az absztrakt halmaz, az absztrakt sorozat és az absztrakt gráf típus

- Sorozatok:
  - a matematikában szokásos módon, egytől indexeljük
  - az indexet alsó indexként jelöljük
  - ha u; $v : \mathcal{T}$<>, akkor az $u + v$ kifejezés a konkatenáljukat jelöli
  - <> önmagában az üres sorozat

Az absztrakt halmaz, az absztrakt sorozat és az absztrakt gráf típus

- A halmaz és a sorozat típusú változókat is deklarálni kell
  - $s : \mathcal{T}$ <> $\rightarrow$ s üres sorozattal inicializálódik
  - $h : \mathcal{T}$ {} $\rightarrow$ h üres halmazzal inicializálódik
  - Ha a zárójelek között megadunk - pontosvesszőkkel elválasztva - néhány $\mathcal{T}$ típusú elemet $\rightarrow$
    a halmaz illetve sorozat a deklaráció kiértékelődése után ezeket fogja tartalmazni

Az absztrakt halmaz, az absztrakt sorozat és az absztrakt gráf típus

- L! $\mathcal{V}$: (vertex, azaz csúcs) absztrakt típus
  - A gráfok csúcsainak absztrakt típusa (A gráfok absztrakt algoritmusainak leírásához)
  - Mindegyik csúcshoz tetszőlegesen sok, névvel jelölt címke társítható (*name*(v))
  - Mindegyik címkéhez tartozik valamilyen érték
    - értékadása: *name*(v) := *x*
    - a *name* címke (azaz a $\mathcal{V}$ halmazon értelmezett parciális függvény) a *name*(v) := *x* értékadással a v csúcsnál az *x* értéket veszi fel.

Az absztrakt halmaz, az absztrakt sorozat és az absztrakt gráf típus

- A $\mathcal{V}$ halmazt az algoritmusok implementációiban legtöbbször az $\mathbb{N}$ halmaz reprezentálja
  - egy *n* csúcsú gráf csúcsait pedig egyszerűen az 1..*n* vagy a 0..(*n*-1) halmaz
    - a tömböket egytől vagy nullától kezdve indexeljük
  - A címkéket ui. gyakran tömbök reprezentálják
    - A láthatóságuk, a hatáskörük és az élettartamuk is korlátozott
    - Az ebből fakadó problémákat az absztrakt algoritmus implementálásakor kell megoldani

Az absztrakt halmaz, az absztrakt sorozat és az absztrakt gráf típus

- Az értékkel bíró címkék mellett használni fogunk egyszerű címkéket is, amikor a gráfok csúcsaihoz és/vagy éleihez egyszerű számértékeket vagy neveket társítunk

- $\mathcal{E}$: élek

$\mathcal{E}$
$+\ u, v : \mathcal{V}$

- $\mathcal{G}$: élsúlyozatlan absztrakt gráfok
  - Egyszerű gráfok (nem tartalmaznak sem párhuzamos, sem hurokéleket)

$\mathcal{G}$
$+\ V : \mathcal{V}\{\}$ $+\ E : \mathcal{E}\{\}\ //\ E \subseteq V \times V \setminus \{(u,u) : u \in V\}\ //\ \text{edges}$

Elemi gráfalgoritmusok

- Elemi gráfalgoritmusok:
  - Élsúlyozatlan gráfokon értelmezett algoritmusok
  - Szélességi gráfkeresés
- Élsúlyozatlan gráfokban tetszőleges út hossza = az út mentén érintett élek száma
- Az út tartalmazhat kört
- Irányított/irányítatlan gráf:
  - irányított gráf: Ha a gráfban tetszőleges *u* és *v* csúcsokra az (*u*; *v*) élt megkülönböztetjük a (*v*; *u*) éltől
  - irányítatlan gráf: ezeket definíció szerint egyenlőknek tekintjük

Szélességi gráfkeresés (BFS: Breadth-first Search)

- Értelmezése:
  - irányított és irányítatlan gráfok
- Feladata:
  - Meghatározza a start csúcsból (*s*)
  - a gráf minden, *s*-ből elérhető csúcsába
  - a legkevesebb élet tartalmazó utat
  - (ha több ilyen van, akkor az egyiket)

Szélességi gráfkeresés (BFS: Breadth-first Search)

- Élsúlyozatlan gráfokban ezeket tekintjük
  - az *s*-ből induló *legrövidebb*, (*optimális*) utaknak
- A csúcsok fontosabb címkéi:
  - *d* - a megtalált úttal hány élen keresztül jutunk a csúcsba
  - π - melyik csúcsból jutunk közvetlenül a csúcsba (ki a szülője)
  - *color*

Szélességi gráfkeresés (BFS: Breadth-first Search)

- *color* - csak szemléletes tartalmuk van:
  - a **fehér** csúcsokat a gráfbejárás/keresés még nem találta meg
  - a **szürke** csúcsokat már megtalálta, de még nem dolgozta fel
  - a **fekete** csúcsokkal viszont már nincs további tennivalója
  - az értéke nem befolyásolja a program futását
    - a rá vonatkozó utasítások az algoritmusból elhagyhatók

Szélességi gráfkeresés
(BFS: Breadth-first Search)

- Az *s*-től *k* távolságra levő csúcsok vannak a gráf *k*-adik szintjén

➢ BFS a gráfot szintenként járja be

  - először a nulladik szintet, aztán az elsőt, majd a másodikat stb.

- Minden szintet teljesen feldolgoz
  - közben a következő szinten levő csúcsokat találja meg.

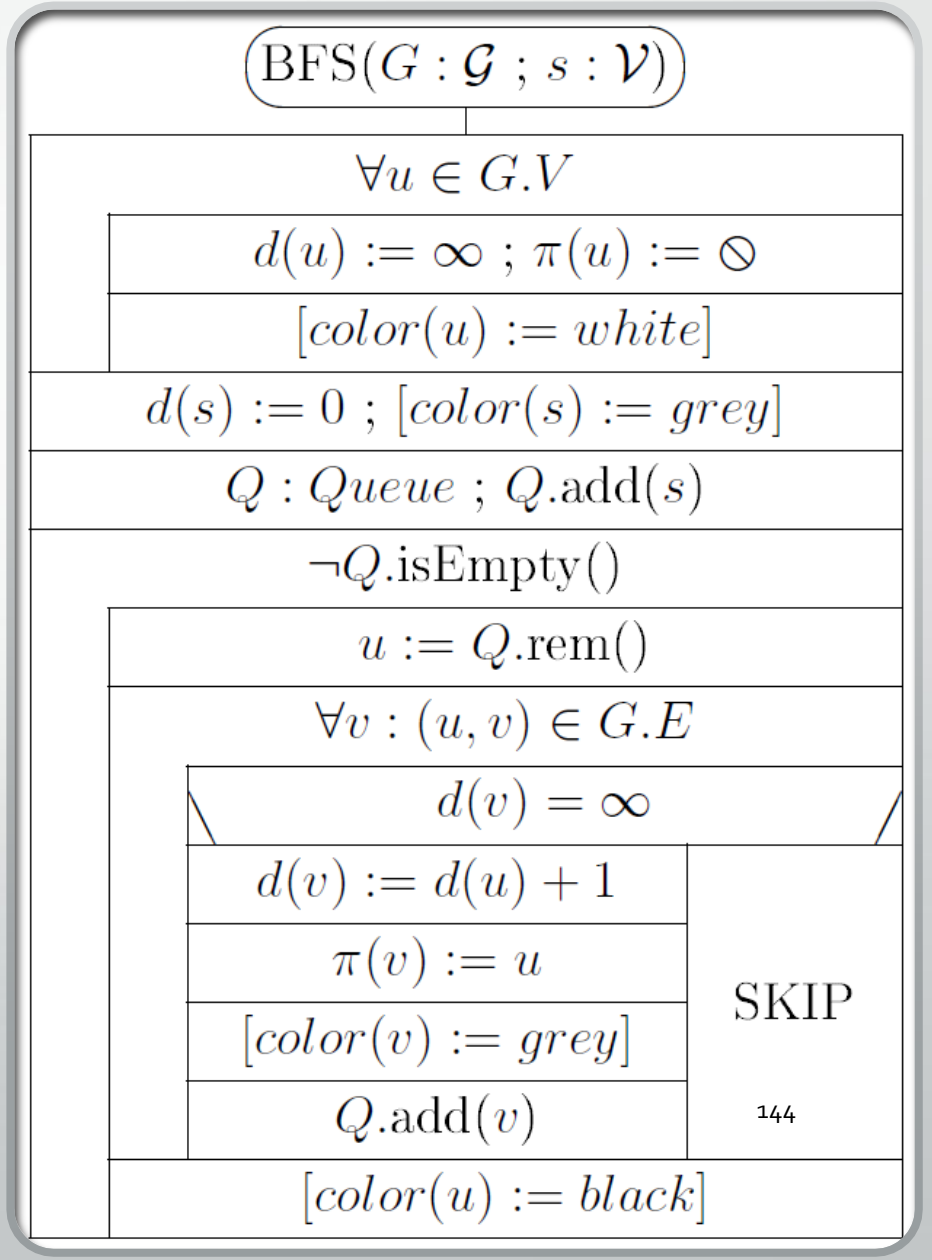
➢ Innét jönnek a szélességi bejárás és a szélességi keresés elnevezések.

Szélességi gráfkeresés (BFS: Breadth-first Search)

- Ha az u csúcs s-ből nem érhető el :
  - $d(u) = \infty$
  - $\pi(u) = \varnothing$
- $\pi(s) = \varnothing$
  - a legrövidebb $s \rightsquigarrow s$ út csak az s csúcsot tartalmazza
  - nincs benne él
  - ➢ s-nek szülője sincs

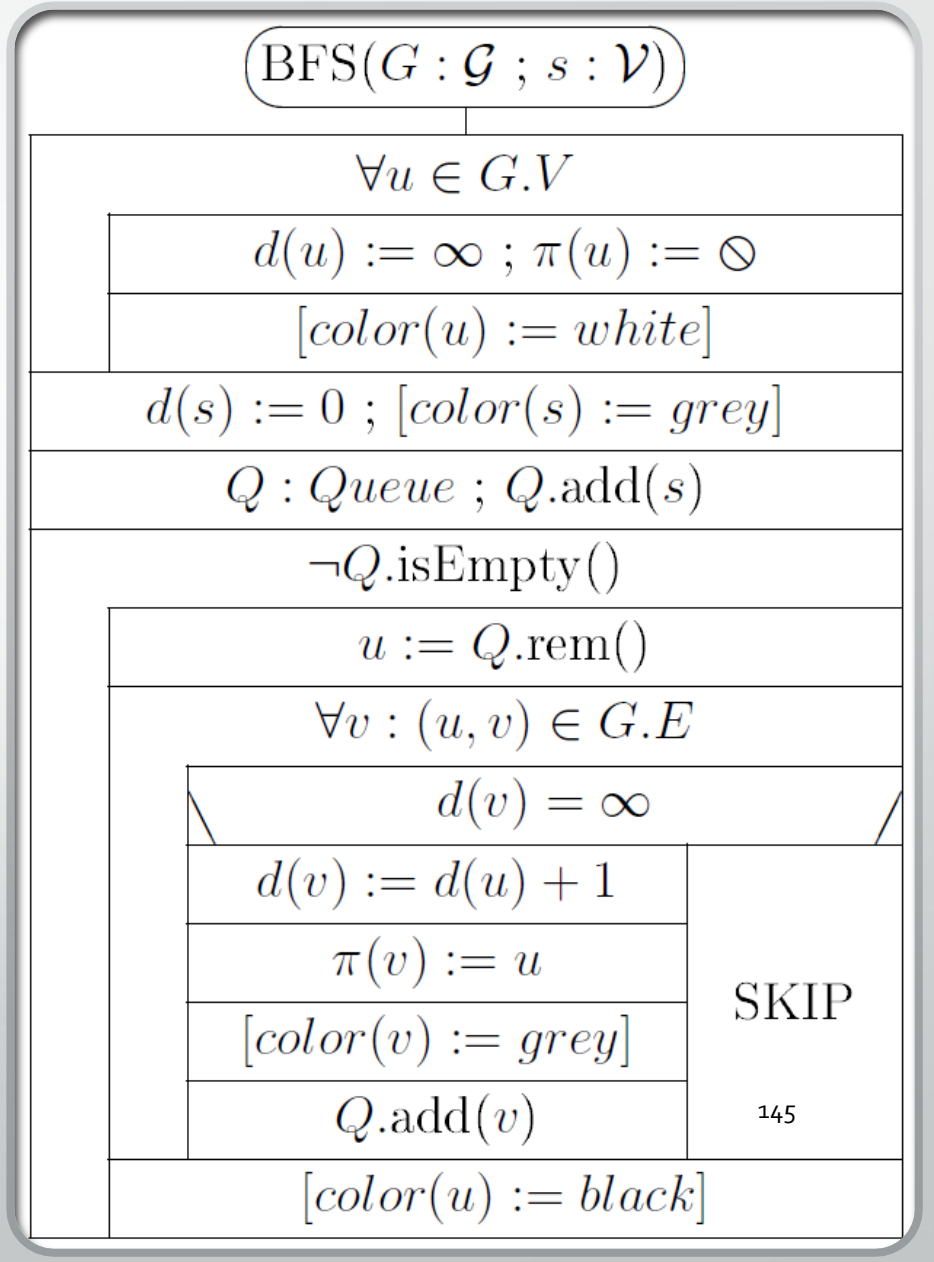
Szélességi gráfkeresés

- 1. ciklus: inicializálás
  - $d(u) = \infty$ (még egyik csúcsot sem értük el)
  - $\pi(u) = \oslash$
- Q: Azok a csúcsok
  - amiket már elértünk
  - de még a gyerekeiket nem néztük meg
- A második, azaz a fő ciklus addig fut, amíg van már elért, de még fel nem dolgozott csúcs.
  - Kiveszi a sorból az *u* csúcsot és kiterjeszti



Szélességi gráfkeresés

- Ha valamelyik *v* gyerekcsúcsra az (*u*; *v*) él feldolgozásakor már d(*v*) ≠ ∞
  - ez a csúcs már korábbról ismert
  - az újonnan *v*-be talált út ennél biztosan nem rövidebb
  - figyelmen kívül is hagyja
- Ha $d(v) = \infty$
  - beállítjuk a címkéiket
  - a sor végére tesszük őket
- Mire az *s*-től *k* távolságra levő csúcsokat feldolgozzuk, a sort már az *s*-től *k* + 1 távolságra levő csúcsok alkotják



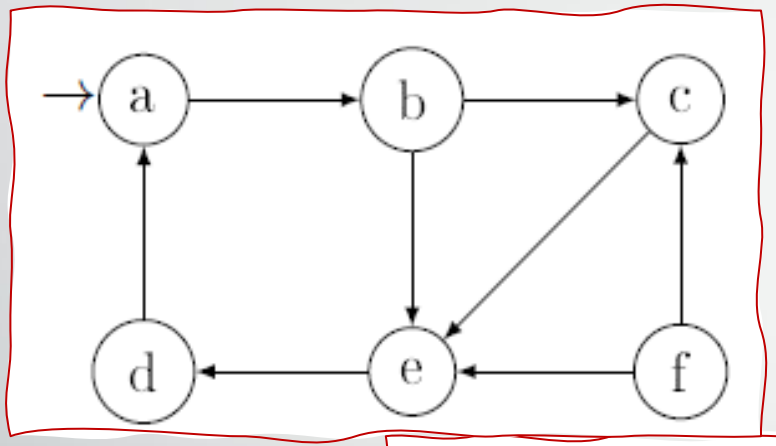
A szélességi fa (Breadth-first Tree)

- A BFS minden *s*-ből elérhető, de tőle különböző csúcsra, annak π címkéjével hivatkozik annak szülőjére, az *s*-ből a csúcsba vezető (a BFS által meghatározott) legrövidebb úton
- Több csúcsnak is lehet ugyanaz a szülője, a szülőcsúcs viszont a BFS által meghatározott legrövidebb utakon egyértelmű
- Az *s*-ből elérhető csúcsok π hivatkozásai -> egy általános fát definiálnak
  - a gyökere *s* -> $\pi(s) = \varnothing$
  - ***szélességi fának*** és ***legrövidebb utak fájának*** is nevezzük
    - fordított irányban mindegyik, az *s*-ből elérhető csúcsra a BFS által meghatározott legrövidebb utakat tartalmazza
    - Fordított ábrázolás célszerűsége
      - minden csúcsnak legfeljebb egy szülője, viszont több gyereke is lehet

      ➢ tömörebb ábrázolás érhető el

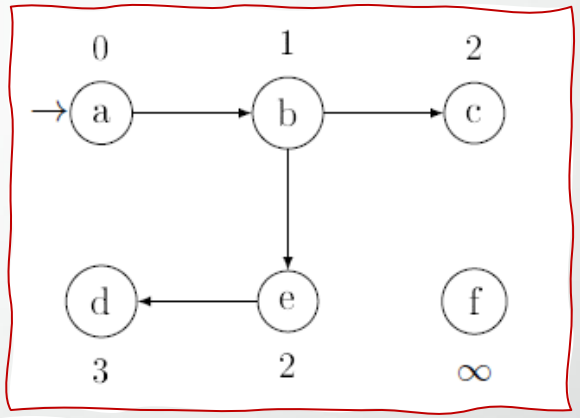
A szélességi gráfkeresés szemléltetése

- Most is és a későbbiekben is, indeterminisztikus esetekben a kisebb indexű csúcsot részesítjük előnyben.
- Kezdő csúcs: a



a → b.
b → c ; e.
c → e.
d → a.
e → d.
f → c ; e.

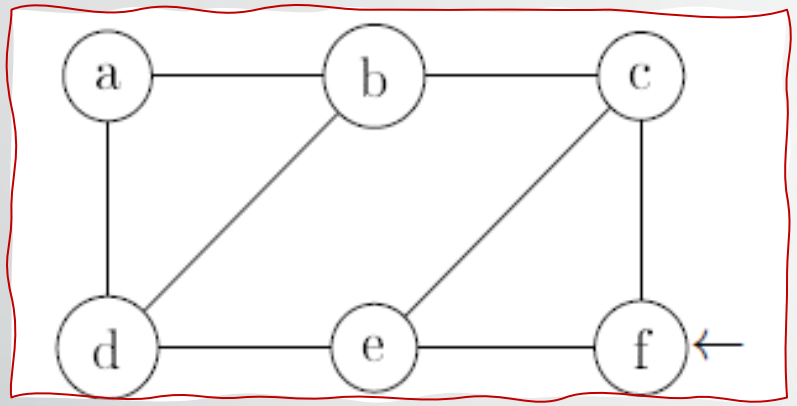
A gráf szélességi fája:



ex-pand dV	changes of d and π						Q : Queue
	a	b	c	d	e	f	
	0 ⊘	∞ ⊘	∞ ⊘	∞ ⊘	∞ ⊘	∞ ⊘	⟨ a ⟩
0 a		1 a					⟨ b ⟩
1 b			2 b		2 b		⟨ c, e ⟩
2 c							⟨ e ⟩
2 e				3 e			⟨ d ⟩
3 d							⟨ ⟩

A szélességi gráfkeresés szemléltetése

- Most is és a későbbiekben is, indeterminisztikus esetekben a kisebb indexű csúcsot részesítjük előnyben.
- Start csúcs: f



A gráf szélességi fája:

a – b ; d.
b – c ; d.
c – e ; f.
d – e.
e – f.

Szélességi gráfkeresés hatékonysága

- L! $n = |G.V|$ és $m = |G.E|$
  - Az első (az inicializáló ciklus) n-szer iterál
  - A második, a fő ciklus annyiszor iterál, ahány csúcs elérhető s-ből (önmagát is számítva)
    - legfeljebb n
    - minimum 1
    - ➢ a belső ciklus legfeljebb m-szer (irányítatlan esetben $2m$-szer) iterál összesen
      - Maximum: ha s- ből mindegyik csúcs elérhető -> minden él sorra kerül
      - Minimum: ha s-ből nem megy ki egyetlen él sem ->egyszer sem

$$\sum: MT(n,m) \in \Theta(n+m) \text{ és } mT(n,m) \in \Theta(n)$$

Szélességi gráfkeresés implementációja szomszédossági listás és szomszédossági mátrixos gráfábrázolás esetén

- A $G = (V, E)$ gráfról föltesszük, hogy $V = \{v_1, \ldots, v_n\}$, ahol $n \in \mathbb{N}$
  - azaz a gráf csúcsait egyértelműen azonosítják az $1..n$ sorszámok
- Az absztrakt v_i csúcsok ábrázolása:
  - d és π címkéinek megfeleltetjük a $d/1$; $\pi/1 : \mathbb{N}\ [n]$ tömböket
    - ahol $d(v_i)$ reprezentációja $d[i]$ és $\pi(v_i)$ reprezentációja $\pi[i]$

Szélességi gráfkeresés implementációja szomszédossági listás és szomszédossági mátrixos gráfábrázolás esetén

- A *color* címkék reprezentálása felesleges
- A $\varnothing$ reprezentációja lehet például a o számérték
  - pl. $\pi[s]$ = o absztrakt jelentése $\pi(v_s) = \varnothing$
- BFS(A/1 : Edge*[n] ; s : 1..n ; d/1; π /1 : $\mathbb{N}$ [n]) és
  BFS(A/1 : bit [n,n] ; s : 1..n ; d/1; π /1 : $\mathbb{N}$ [n])

Ellenőrző kérdések

1. A $G[1..n]$ egy irányított gráf szomszédossági listás ábrázolása.
   - Adja meg a listaelem típus leírását!
   - Írja meg a **transzponál**(*G,n,GT)* eljárást, ami előállítja a gráf transzponáltjának a szomszédossági listás reprezentációját!
   - Írja meg a **kibeFokok**(*G,n,be,ki)* eljárást, ami minden $u \in 1..n$ csúcsra a $be[u]$ –ban kiszámítja a csúcs bemeneti fokszámát, $ki[u]$ –ban pedig a csúcs kimeneti fokszámát!
   - Mindkét eljárásban $MT(n,m) \in \Theta(n + m)$, ahol m a gráf éleinek száma

Ellenőrző kérdések

2. Szélességi gráfkeresés

- Mit számol ki a Szélességi gráfkeresés?
- Adja meg az algoritmus absztrakt struktogramját!
- A Szélességi gráfkeresés a gráf mely csúcsaiba talál optimális utat, és a végrehajtás során mikor?
- Mit tud a Szélességi gráfkeresés műveletigényéről?
- Szemléltesse az algoritmust az **a** csúcsból indítva, az **a – b ;d. b – c;d. c-e. d-e.** Irányítatlan gráfon! Rajzolja le az eredményül adódó szélességi fát is!

Köszönöm a figyelmet!

Pusztai Kinga

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

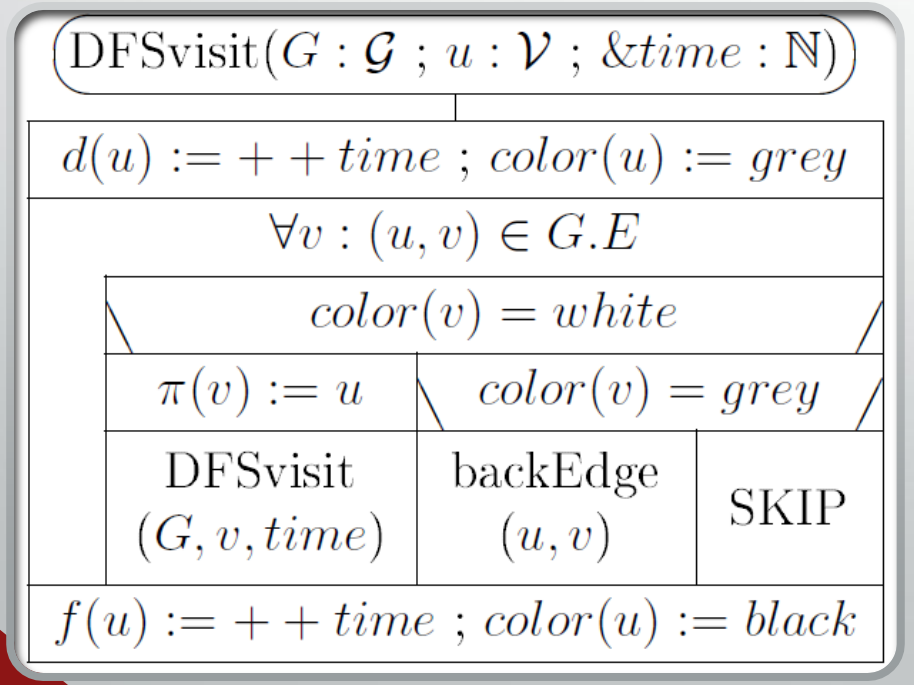
Algoritmusok és adatszerkezetek I.
5. Előadás

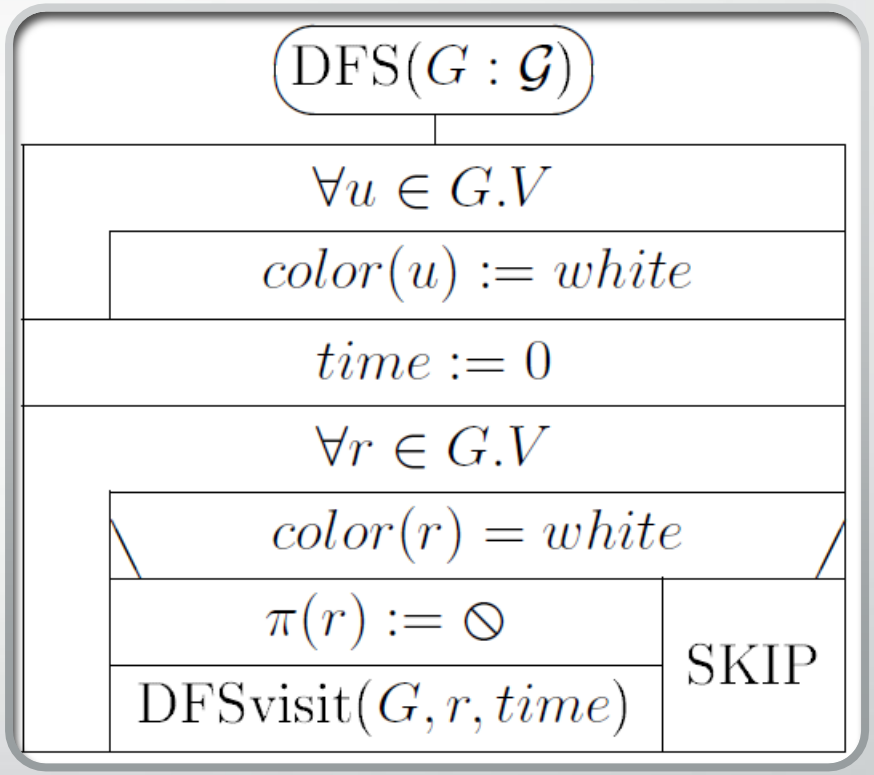
Mélységi gráfkeresés, élek osztályozása, DAG, topologikus rendezés, irányított kör keresése.

Tartalom

Mélységi gráfkeresés (DFS. Depth-first Search)

- Csak egyszerű irányított gráfokra értelmezzük
- Csúcsok színei:
  - Fehér csúcs: érintetlen
  - Szürke csúcs: belőle elérhető csúcsokat járunk be éppen
  - Fekete csúcs: befejeztük





- *d*(u): elérési idő (discovery time)
- *f*(u): befejezési idő (finishing time)

Mélységi feszítő erdő (Depth-first forest)

- Mindegyik, a DFS eljárásból indított DFSvisit egy-egy *mélységi fát* számol ki
- Ezek összessége: *mélységi feszítő erdő*
  - $r \in G.V$ egy mélységi fa gyökere $\Leftrightarrow \pi(r) = \varnothing$
  - $(u, v) \in G.E$ egy mélységi fa éle $\Leftrightarrow u = \pi(v)$

Az élek osztályozása (Classification of edges)

1. Definíció. A gráf éleinek osztályozása:

- (u, v) fa-él (tree edge) $\Leftrightarrow$ (u, v) valamelyik mélységi fa egyik éle.
  - (A fa-élek mentén járjuk be a gráfot.)
- (u, v) vissza-él (back edge) $\Leftrightarrow$ v az u őse egy mélységi fában.
- (u, v) előre-él (forward edge) $\Leftrightarrow$ (u, v) nem fa-él, de v az u leszármazottja egy mélységi fában.
- (u, v) kereszt-él (cross edge) $\Leftrightarrow$ u és v két olyan csúcs, amelyek ugyanannak a mélységi fának két különböző ágán vannak, vagy két különböző mélységi fában találhatók.

Az élek osztályozása (Classification of edges)

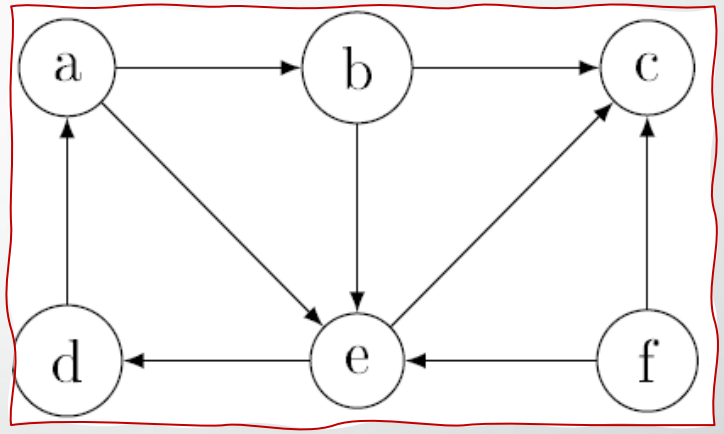
2. Tétel. A gráf éleinek felismerése:

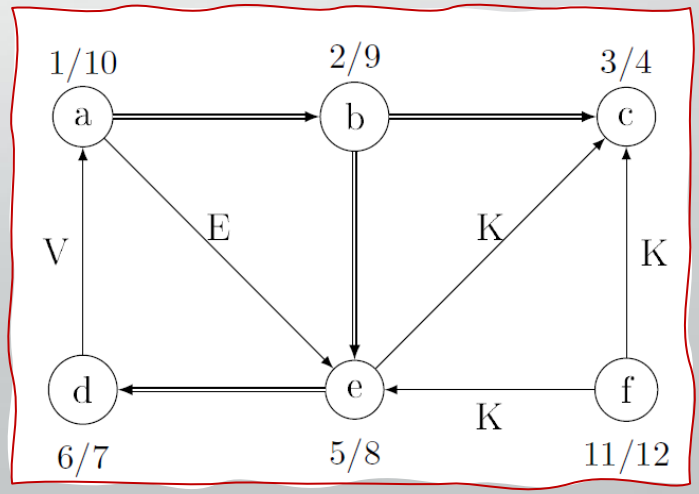
- A mélységi bejárás során tetszőleges (u, v) él feldolgozásakor az él a következő kritériumok alapján osztályozható:
  - (u, v) fa-él (tree edge) $\Leftrightarrow$ a v csúcs még fehér.
  - (u, v) vissza-él (back edge) $\Leftrightarrow$ a v csúcs éppen szürke.
  - (u, v) előre-él (forward edge) $\Leftrightarrow$ a v csúcs már fekete $\wedge$ $d(u) < d(v)$.
  - (u, v) kereszt-él (cross edge) $\Leftrightarrow$ a v csúcs már fekete $\wedge$ $d(u) > d(v)$.

A mélységi bejárás szemléltetése

a → b ; e.
b → c ; e.
d → a.
e → c ; d.
f → c ; e.

- A bejárás indeterminisztikus abban, hogy az egyes ciklusokban a csúcsokat milyen sorrendben veszi sorra
  - Szemléltetésben: csúcsokat ábécé rendben, illetve indexek szerint monoton növekvően dolgozzuk fel
- *d*/*f* alakú címkék:
  a csúcs *elérési/befejezési* idő
  (discovery/finishing time)
- Élek osztályozása:
  - fa-élek: dupla szárú nyíl
  - vissza- él: V
  - előre-él: E
  - kereszt-él: K



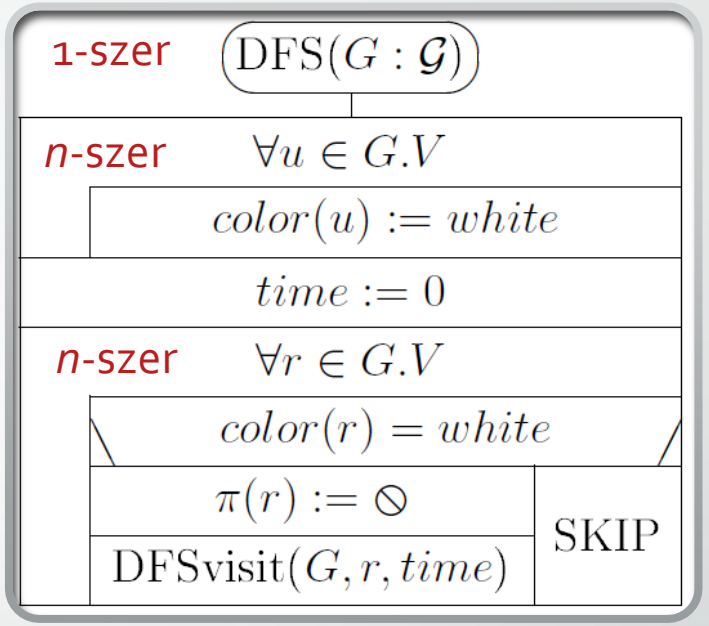


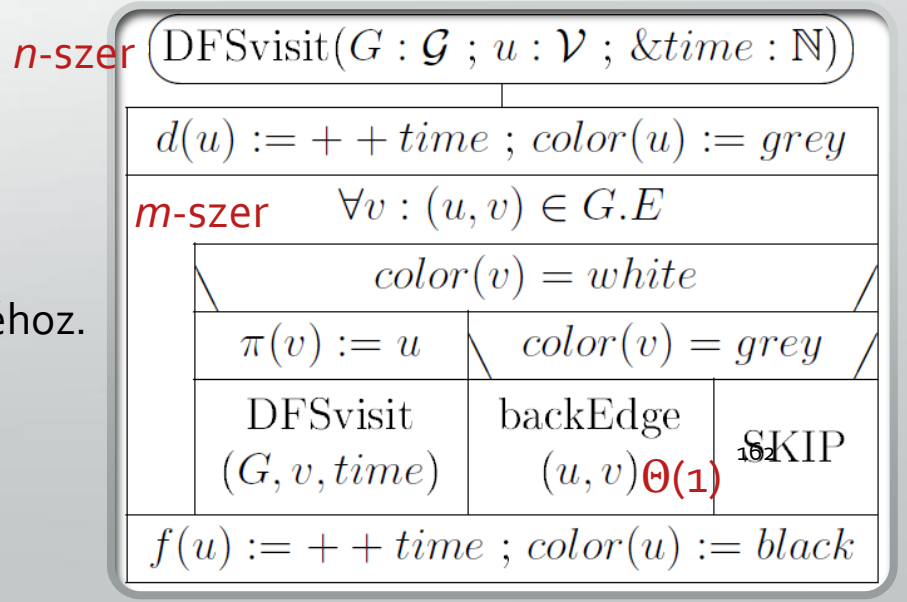
A mélységi gráfkeresés futási ideje

- $n = |G.V|$ és $m = |G.E|$
- DFS mindkét ciklusa *n*-szer fut le.
- DFSvisit rekurzív eljárás:
  - Mindegyik csúcsra, amikor először érjük el (fehér)egyszer
  - **összesen *n*-szer** hívódik meg
- A DFSvisit ciklusa
  - mindegyik csúcsra annyit iterál, amennyi a kimeneti fokszáma.
  - Ez a ciklus tehát a gráf éleit dolgozza fel, mindegyiket egyszer.
  - Összesen ***m*** iterációt végez.
- A DFS csak egyszer hívódik meg.
- Tfh! a backEdge eljárás: csak megjelöl egy visszaélet
  - Θ(1) műveletigényű
  - és műveletigénye hozzávehető az őt meghívó ciklusiterációéhoz.

➢ Pontosan: (1+2*n*)+(*n*+*m*) = 3*n* + *m* + 1 lépés

➢ $MT(n); mT(n) \in \Theta(n + m)$





A DAG tulajdonság eldöntése

3. Definíció. A *G* irányított gráf akkor DAG (Directed Acyclic Graph = körmentes irányított gráf), ha nem tartalmaz irányított kört.

- A DAG-ok a gráfok fontos osztályát képezik: sok hasznos algorimus DAGot vár a bemenetén
  - az input ellenőrzése is szükséges lehet
- Ha a mélységi bejárás egy (*u*, *v*) vissza-élet talál -> irányított kört is talált a gráfban
  - ekkor a vissza-él a definíciója szerint egy mélységi fában az *u* csúcs egyik őse a *v* csúcs
  - és a *v*-ből *u*-ba vezető, fa-élekből álló út az (*u*, *v*) éllel együtt irányított kört alkot

A DAG tulajdonság eldöntése

- Bebizonyítható, hogy ha a *G* irányított gráf nem DAG (irányított kört tartalmaz)
  ->a DFS fog vissza-élet, és ezzel irányított kört találni
  - Az viszont nem garantált, hogy az összes irányított kört megtalálja

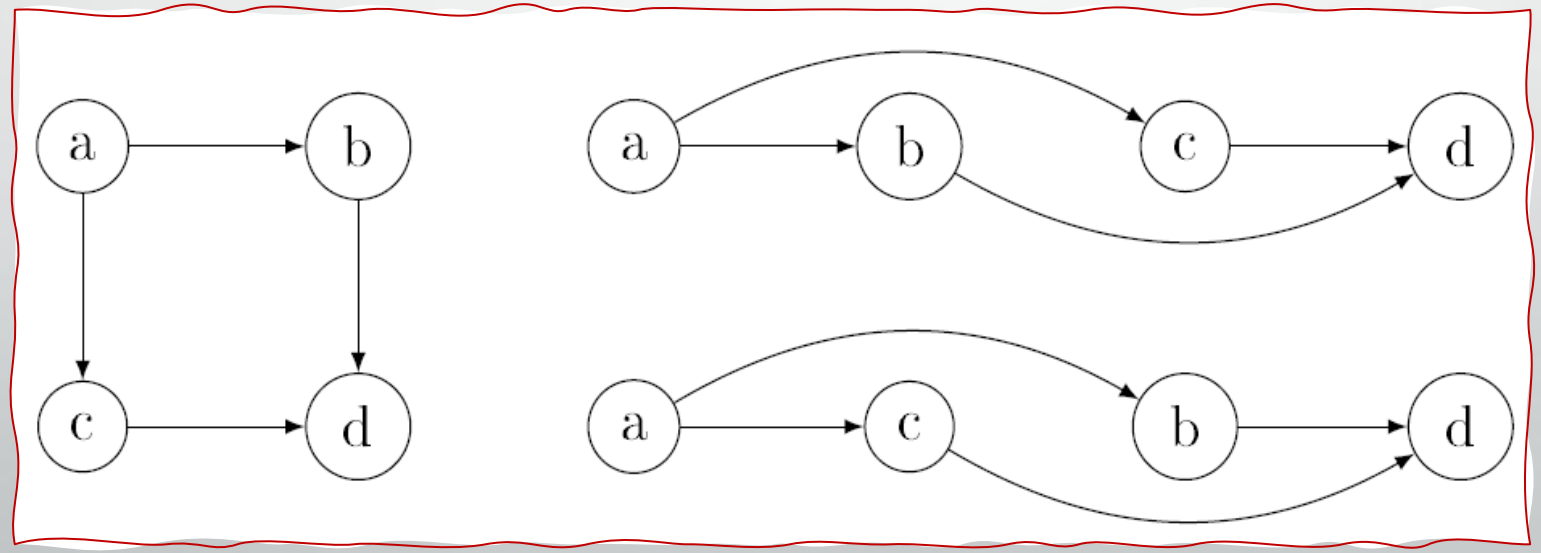
4. Tétel. A *G* irányított gráf DAG ⟺ a mélységi bejárás nem talál G-ben vissza-élet.
Ha a DFS talál egy (*u*, *v*) vissza-élet, akkor az <*u*, $\pi(u)$, $\pi(\pi(u))$,… *v*; *u*> csúcssorozat visszafelé olvasva egyszerű irányított kört ad.

- A tétel alapján a backEdge eljárás akár ki is nyomtathatja a megtalált irányított kört.
  - Ebben az esetben a maximális futási ideje nyilván $\Theta(n)$ lesz

Topologikus rendezés

5. Definíció. Irányított gráf topologikus rendezése alatt a gráf csúcsainak olyan sorba rendezését értjük, amelyben minden él egy-egy később jövő csúcsba (szemléletesen: balról jobbra) mutat.

- Ugyanannak a gráfnak lehet egynél több topologikus rendezése



Topologikus rendezés

5. Tétel. Tetszőleges irányított gráfnak pontosan akkor van topologikus rendezése, ha nincs irányított kör a gráfban, azaz a gráf DAG.

- **Bizonyítás.**
  - ⇒ (Ha van irányított kör a gráfban, akkor nincs a gráfnak topologikus rendezés.)
    - Ha van irányított kör a gráfban, jelölje $<u_1, u_2, \ldots, u_k, u_1>$!
    - Ekkor egy tetszőleges topologikus rendezésben u_1 után jön valahol u_2, az után valahol u_3, és így tovább, végül is u_1 után jön u_k,
    - és u_k után u_1 => ellentmondás
    - ➢ Ekkor nincs topologikus rendezés (a gráf csúcsain).

Topologikus rendezés

- **Bizonyítás.**
  - ⇐ (Ha nincs irányított kör a gráfban, akkor van a gráfnak topologikus rendezés.)
    - Ha nincs irányított kör a gráfban → van olyan csúcs, aminek nincs megelőzője
    - Ha veszünk egy megelőzővel nem rendelkező csúcsot, és töröljük a gráfból →a maradék gráfban nem keletkezik irányított kör
    - ➢ lesz megint legalább egy olyan, amelyiknek nincs megelőzője
    - Sorban a törölt csúcsok adják a topologikus rendezést

Tetszőleges DAG-ra a topologikus rendezés algoritmusa

- A DAG topologikus rendezésésében a csúcsok a befejezési idők szerint szigorúan monoton csökkenően jelennek meg.
- Ha az algoritmus indeterminizmusát más (az alfabetikustól különböző) konvenció mentén oldanánk fel,
  gyakran a példákban kiszámoltaktól különböző topologikus rendezést kapnánk!
- A topologikus rendezés egy kézenfekvő alkalmazása: az ún. egygépes ütemezési probléma megoldása:
  - A csúcsok: (munka)folyamatok
  - Az élek: a köztük lévő rákövetkezési kényszerek
  - A folyamatokat ezeknek megfelelően kell sorba rakni.

Tetszőleges DAG-ra a topologikus rendezés algoritmusának lépései

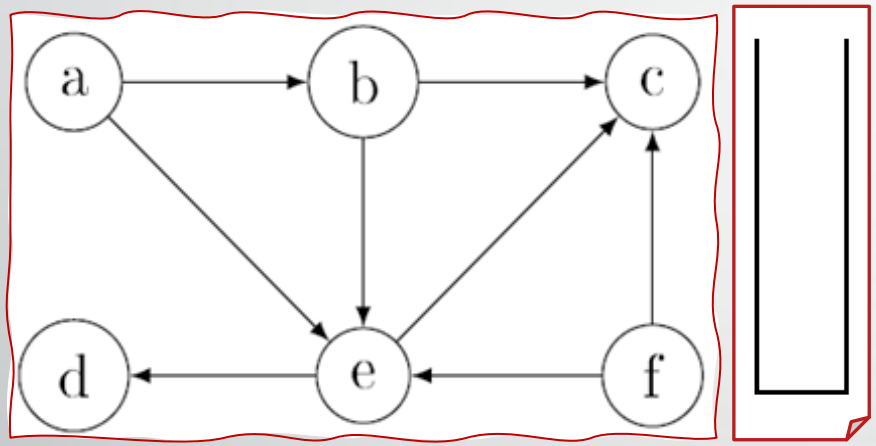
1. Létrehozunk egy üres vermet
2. Végrehajtjuk a gráf mélységi bejárását
   - valahányszor befejezünk egy csúcsot, a verem tetejére tesszük
3. A verem tartalmát kiolvasva megkapjuk a gráf csúcsainak topologikus rendezését

Tetszőleges DAG-ra a topologikus rendezés algoritmusa

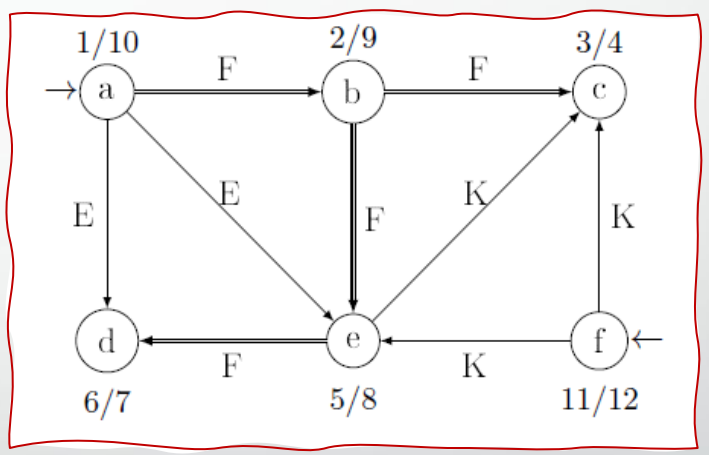
- Az algoritmus képes ellenőrizni a saját előfeltételét
  - Ha a DFS vissza-élt talál -> a gráf irányított kört tartalmaz ->az algoritmus eredménye: nincs topologikus rendezés
  - (Ilyenkor a verem tartalma nem használható fel.)
- *Feladat:* Írja meg a topologikus rendezés stuktogramját! (szomszédossági listás és szomszédossági mátrixos gráfábrázolások esetén!)
  - Mit tud mondani az algoritmusok hatékonyságáról?

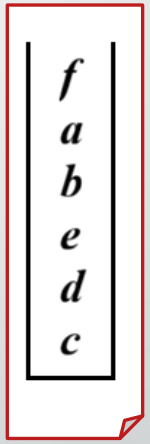
Topologikus rendezés algoritmus szemléltetése

- Bemeneti gráf:



- Mélységi bejárás:





- Eredmény (verem tartalmának kiírítása) :
  - A bemeneti gráf topologikus rendezése: ⟨*f*, *a*, *b*, *e*, *d*, *c*⟩

Erősen összefüggő komponensek meghatározása = A redukált gráf előállításának algoritmusa*

Lépések:

1. A gráfot bejárjuk mélységi bejárással,

   a csúcsokat a befejezési számok sorrendjében kiírjuk egy verembe.

2. Transzponáljuk a gráfot, azaz megfordítjuk az élek irányítását.

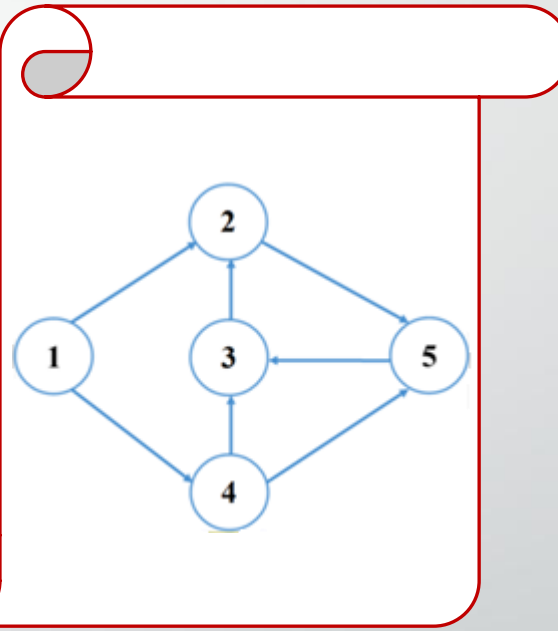
3. Bejárjuk a transzponáltat mélységi bejárással, de nem az alapvető sorrend szerint vesszük a csúcsokat kiinduló csúcsnak, hanem az 1. lépésben készített veremből történő kivétel sorrendjében.

Erősen összefüggő komponensek meghatározása = A redukált gráf előállításának algoritmusa

- A lépések végrehajtásával egy mélységi erdőt kapunk:
  - A fák a gráf erős komponensei lesznek
- Redukált gráf előállítása:
  - EÖK fáinak a csúcsait összevonjuk egy csúccsá
  - A csúcsok között az éleket az eredeti gráf éleinek megfelelően megadjuk

Erősen összefüggő komponensek*

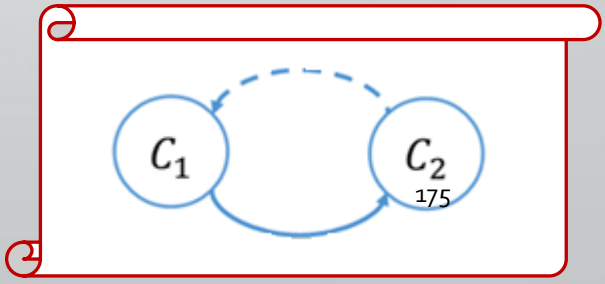
- **Definíció.**
  - Legyen *G=(V,E)* irányított, véges gráf. *G összefüggő*, ha *G* irányítás nélkül összefüggő.
  - *G erősen összefüggő*, ha $\forall\, u, v \in E$ esetén $\exists\, u \rightsquigarrow v$ út.
- **Definíció.** Legyen *G=(V,E)* irányított, véges gráf, $u, v \in V$.
  - Ekkor legyen $u \approx v$, ha léteznek $u \rightsquigarrow v$, illetve $v \rightsquigarrow u$ utak a gráfban.
- **Állítás.** A $\approx$ reláció ekvivlenciareláció
  - t $\approx$ osztályozza a *V* csúcshalmazt.



Összefüggő, de nem erősen összefüggő gráf

Erősen összefüggő komponensek*

- **Definíció.** A $\approx$ reláció ekvivalencia osztályait nevezzük a gráf *erős komponense*inek.
- **Állítás.** Egy összefüggő gráf két erős komponense között az élek csak egy irányba mehetnek.
- **Bizonyítás.** Indirekt tegyük fel, hogy két erős komponens között oda-vissza is mehetnek élek.
  - Legyen a két komponens: C_1 és C_2
  - Ekkor $u \in C_1$ és $v \in C_2$ csúcsok között léteznek $u \rightsquigarrow v$ és $v \rightsquigarrow u$ utak, ugyanis:
    - az erős komponenseken belül: definíció szerint
    - C_1 és C_2 között: az indirekt feltevés szerint létezik út
  - $u \approx v$, ami ellentmondás



Erősen összefüggő komponensek*

- **Műveletigény:** $T(n) \in O(n + m)$
  - Mélységi bejárás: $O(n + m)$
  - Gráf megfordítása: $O(m)$
  - Veremműveletek
- **Megjegyzés:** Ha a gráf DAG, akkor az algoritmus 3. lépésében említett bejárási sorrend a gráf egy topologikus rendezése.
- **Állítás:** Bármely mélységi bejárás során, egy erős komponens összes csúcsa ugyanabba a mélységi fába kerül.
- **Állítás:** Futtassuk le a fenti algoritmust a $G=(V,E)$ gráfon.
  Legyenek $u,v \in V$ a gráf csúcsai, és tekintsük az algoritmus 3. lépésében kapott mélységi fákat.

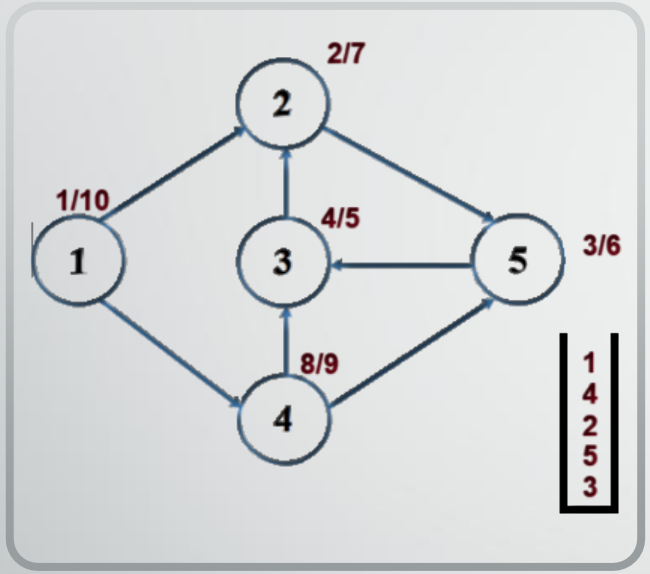
  Ekkor $u \approx v \iff$ ha u és v ugyanabban a mélységi fában vannak

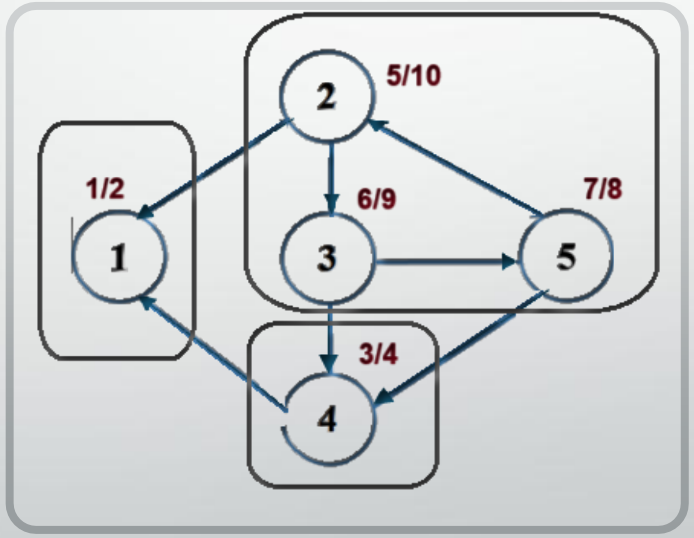
Erősen összefüggő komponensek*

- **Definíció.** Legyen *G=(V,E)* irányított, véges gráf. *G redukált gráf*ja olyan irányított gráf,
  - amelynek csúcsai *G* erős komponensei,
  - és a redukált gráf két csúcsa között, akkor halad él, ha csúcsoknak megfelelő *G* erős komponensei között halad él,
  - továbbá az él irányítása megegyezik az erős komponensek között haladó él(ek) irányításával.
- **Állítás.** A redukált gráf DAG, azaz körmentes irányított gráf.
- **Bizonyítás.** Indirekt tegyük fel, hogy a redukált gráf nem DAG, azaz létezik benne irányított kör.
  - Legyen egy ilyen kör $C_1 \rightarrow C_{1-} \rightarrow C_2 \rightarrow \ldots \rightarrow C_k \rightarrow C_1$
  - Ekkor a kör mentén lévő komponensek kölcsönösen elérhetők,
  - vagyis $C_1 \approx C_2 \approx \ldots \approx C_k$, ami ellentmondás.

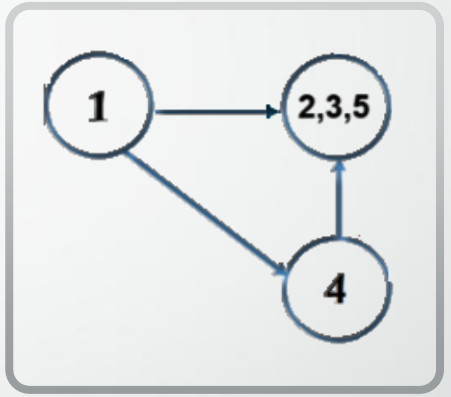
Példa

Erősen összefüggő komponensek





Redukált gráf



Ellenőrző kérdések: Mélységi gráfkeresés

1. Rajzolja le a Mélységi gráfkeresés absztrakt struktogramját!
   - Mit tud a műveletigényéről? (Indokolja is az állítást!)
2. Adja meg az éltípusok definícióját és mondja ki az osztályozásukkal kapcsolatos tételt!
3. Szemléltesse a Mélységi keresést az alábbi irányított gráfon!
   - nemdeterminisztikus esetekben mindig a kisebb indexű csúcsot részesítse előnyben!
   - Jelölje a bejárás során a különböző éltípusokat is!
   - *a* -> *b* ; *d*. *b* -> *c* ; *d*. *c*-> *e*. *d*.-> *e*. *e*-> *b*. *f*-> *c* ; *e*.

Ellenőrző kérdések: topologikus rendezés

1. Mit értünk egy irányított gráf csúcsainak topologikus rendezése alatt?
   - Mondja ki és bizonyítsa be az ezzel kapcsolatos tételt!
2. Mutassa be az alábbi gráf csúcsai topologikus rendezésének a gráf mélységi bejárására épülő algoritmusát!
   - *a*-> *b*; *d*. *b* -> *c*; *d*. *c*-> *e*. *d*-> *e*. *f*-> *c*; *e*.
   - Nemdeterminisztikus esetekben előny: az alfabetikusan kisebb indexű csúcs
   - Módosítsa egyetlen él behúzásával úgy a gráfot, hogy ne legyen topologikus rendezése!
     - A módosított gráfnak miért nincs topologikus rendezése?
     - Mikor derül ez ki a fent szemléltetett algoritmus végrehajtása során?
3. Mit tud a mélységi bejárásra épülő topologikus rendezés műveletigényéről?

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek II. előadásjegyzet: Elemi gráfalgoritmusok és Fekete István: Algoritmusok és adatszerkezetek jegyzete alapján készült.

Algoritmusok és adatszerkezetek I.
6. Előadás

MST, általános algoritmus, biztonsá-gos élek, Kruskal algoritmus, unió-holvan adatszerkezet.

Tartalom

Élsúlyozott gráfok és ábrázolásaik

1. Definíció. **Élsúlyozott gráf** alatt egy $G = (V, E)$ gráfot értünk a $w : E \to \mathbb{R}$ súlyfüggvénnyel, ahol

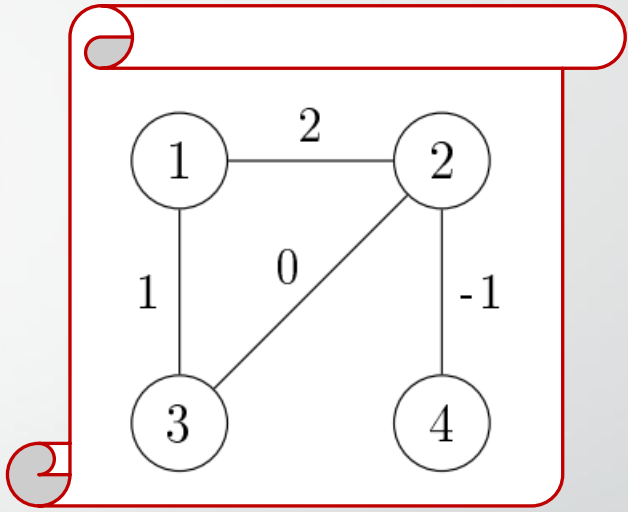
- V a csúcsok (vertices) tetszőleges, véges halmaza,
- $E \subseteq V \times V \setminus \{(u; v) : u \in V\}$ az élek (edges) halmaza.
- A $w : E \to \mathbb{R}$ minden egyes élhez hozzárendeli annak súlyát, más néven hosszát, illetve költségét.
  - (Az élsúly, élhossz és élköltség elnevezések szinonimák.)

2. Definíció. Élsúlyozott gráfban tetszőleges **út hossza** (más néven **költsége**, **súlya**) az út mentén található élek összsúlya.
Hasonlóképpen tetszőleges **gráf/fa súlya** az élei súlyainak összege.

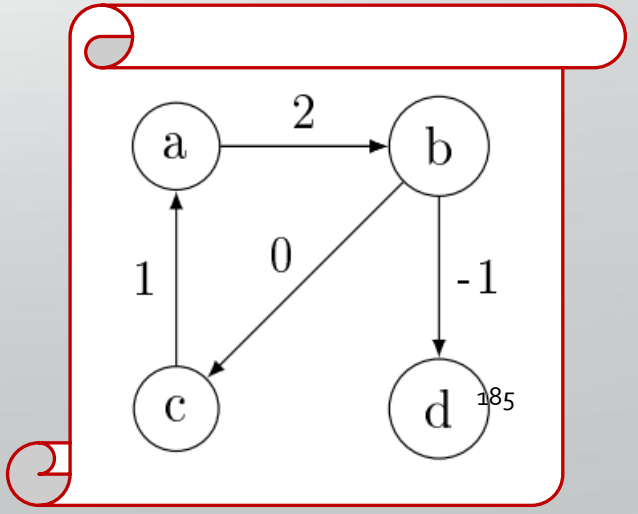
Élsúlyozott gráfok ábrázolásai

$1 - 2,\ 2\ ;\ 3,\ 1.$
$2 - 3,\ 0\ ;\ 4,\ -1.$

- Grafikus ábrázolás:
  - Az éleket súlyukkal címkézzük
- Szöveges ábrázolás
  - irányítatlan gráfoknál: „u - v_{u1}, w_{u1} ; . . . v_{uk} ,w_{uk}.” jelentése:
    - (u, v_{u1}), . . .; (u, v_{uk}) élei a gráfnak
      $w(u, v_{u1}) = w_{u1}$ … , $w(u, v_{uk}) = w_{uk}$ súlyokkal.
  - irányított gráfoknál: „u -> v_{u1}, w_{u1} ; . . . v_{uk} ,w_{uk}.” jelentése:
    - a gráfban az u csúcsból (u, v_{u1}), . . .; (u, v_{uk}) irányított élek indulnak ki, azaz az u csúcs rákövetkezői (gyerekei): v_{u1}; . . . v_{uk} csúcsok.



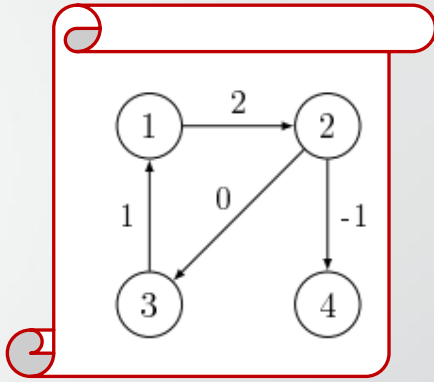
$a \rightarrow b,\ 2.$
$b \rightarrow c,\ 0\ ;\ d,\ -1.$
$c \rightarrow a,\ 1.$



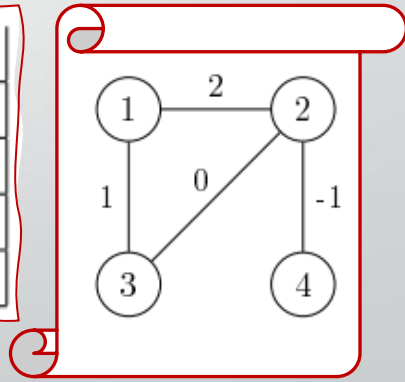
Szomszédossági mátrixos (adjacency matrix), más néven csúcsmátrixos reprezentáció

- $G = (V, E)$ gráf $w : E \to \mathbb{R}$ súlyfüggvénnyel ($V = \{v_1, \cdots, v_n\}$) reprezentációja:
  - $A/1 : \mathbb{R}_\infty [n, n]$ mátrix, ahol
    - $n = |V|$ a csúcsok száma
    - $1..n$ a csúcsok sorszámai
    - tetszőleges $i, j \in 1..n$ csúcssorszámokra:
      - $A[i, j] = w(v_i, v_j) \Leftrightarrow (v_i, v_j) \in E$
      - $A[i, i] = 0$
      - $A[i, j] = \infty \Leftrightarrow (v_i, v_j) \notin E \wedge i \neq j$

A	1	2	3	4
1	0	2	∞	∞
2	∞	0	0	-1
3	1	∞	0	∞
4	∞	∞	∞	0



A	1	2	3	4
1	0	2	1	∞
2	2	0	0	-1
3	1	0	0	∞
4	∞	-1	∞	0

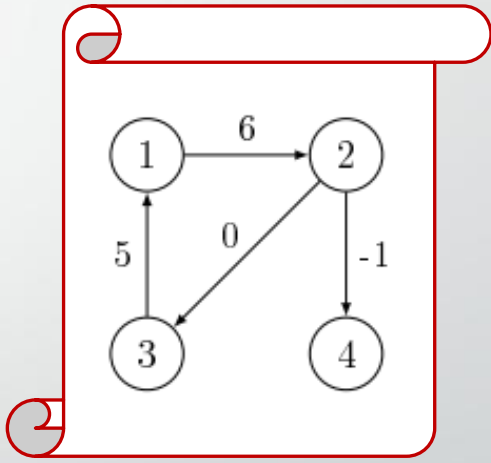


- *Megjegyzések:*
  - A főátlóban mindig nullák vannak
    - csak egyszerű gráfokkal foglalkozunk (amelyekben nincsenek hurokélek)
    - tetszőleges csúcsból önmaga közvetlenül, nulla költségű úton érhető el.
  - Irányítatlan esetben a szomszédossági mátrixos reprezentáció mindig szimmetrikus,
    - $(v_i, v_j) \in E$ esetén $(v_j, v_i) = (v_i, v_j) \in E$

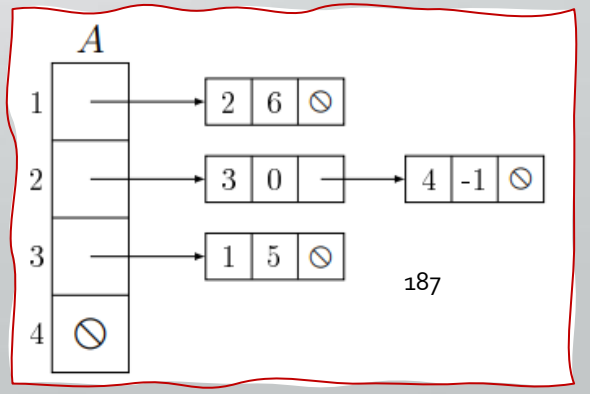
Szomszédossági listás (adjacency list) reprezentáció

- $G = (V, E)$ gráf $w : E \to \mathbb{R}$ súlyfüggvénnyel ($V = \{v_1, \cdots, v_n\}$) reprezentációja:
  - $A/1 : \mathbb{R}_\infty [n, n]$, ahol
  - A : *Edge**[n] pointertömb, ahol
    - az *Edge* típus a következő:
    - A *next* és a *v* attributumok szerepe ugyanaz, mint az élsúlyozatlan gráfoknál, *w* pedig a megfelelő él súlya.

Edge
$+v : \mathbb{N}$
$+w : \mathbb{R}$
$+next : Edge*$

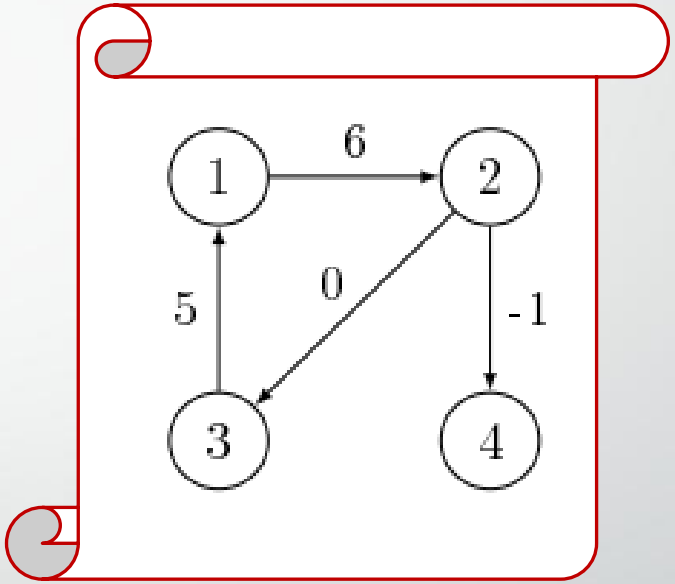


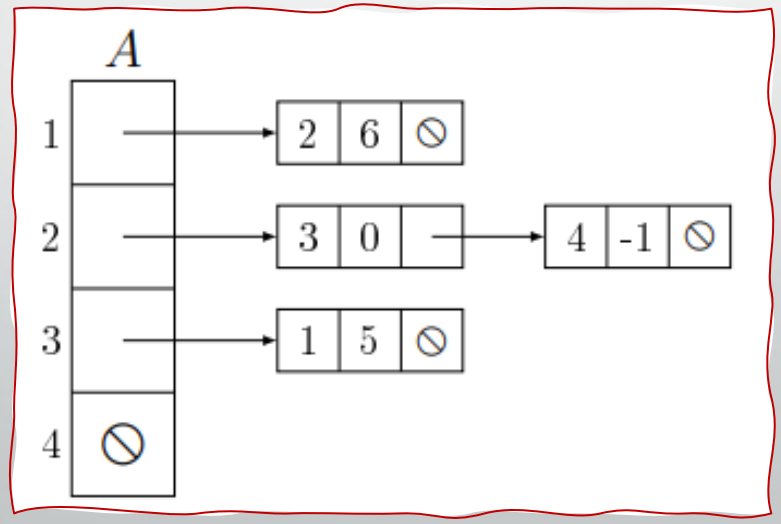
- Az élsúlyozatlan gráfokhoz hasonlóan az élsúlyozott gráfoknál is
  - Irányítatlan gráfok esetén minden élet kétszer ábrázolunk
  - Irányított gráfok esetén csak egyszer



Szomszédossági listás reprezentáció szemléltetés

- A[1] → v = 2 ; A[1] → w = 6 ; A[1] → next = ⊘
- A[2] → v = 3 ; A[2] → w = 0 ; A[2] → next → v = 4;
- A[2] → next → w = −1 ; A[2] → next → next = ⊘
- A[3] → v = 1 ; A[3] → w = 5 ; A[3] → next = ⊘
- A[4] = ⊘





Élsúlyozott gráfábrázolások tárigénye

- Szomszédossági mátrixok:
  - Tárigényük hasonlóan számolható, mint élsúlyozatlan esetben
  - Tfh. egy valós számot egy gépi szóban tárolunk
  - Alapesetben: n^2 szó
  - Irányítatlan gráfoknál:
    - Csak az alsóháromszög mátrixot tárolva
    - $n * (n - 1) / 2$ szó

➢ $n * (n - 1) / 2 \in \Theta(n^2)$ -> az aszimptotikus tárigény mindkét esetben: $\Theta(n^2)$

Élsúlyozott gráfábrázolások tárigénye

- Szomszédossági listák:
  - Mindegyik élben eggyel több mező van, mint az élsúlyozatlan esetben
  - Ez az aszimptotikus tárigényt nyilván nem befolyásolja: $\Theta(n + m)$

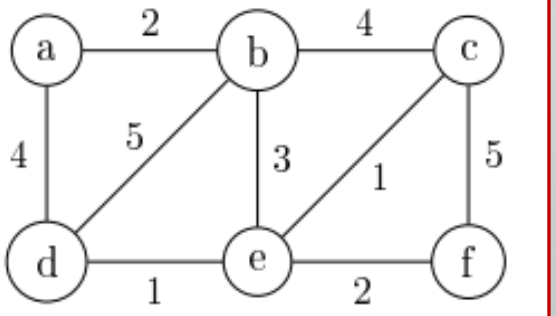
Élsúlyozott gráfok absztrakt osztálya

$\mathcal{G}_w$
$+\ V : \mathcal{V}\{\}$
$+\ E : \mathcal{E}\{\}\ //\ E \subseteq V \times V \setminus \{(u,u) : u \in V\}$
$+\ A : V \to 2^V\ //\ A(u) = \{\, v \in V \mid (u,v) \in E \,\}$
$//\ A(u) =$ the adjacent vertices of vertex u.
$+\ w : E \to \mathbb{R}\ //$ weights of edges

Minimális feszítőfák
(MST = Minimum Spanning Tree)

a – b, 2 ; d, 4.
b – c, 4 ; d, 5 ; e, 3.
c – e, 1 ; f, 5.
d – e, 1.
e – f, 2.

- Feladat minimális feszítőfa keresésére
  - A csúcsok pl. városok,
  - az élek lehetséges összeköttetések, a megépítésük költségeivel.
  - A lehető legkisebb építési költséggel szeretnénk elérni, hogy tetszőleges városból bármelyik másikba el lehessen jutni.
- Gráfok
  - összefüggő,
  - irányítatlan,
  - Élsúlyozott. (Az élsúlyok negatívak is lehetnek.)



Minimális feszítőfák

1. Definíció. A $G = (V, E)$ irányítatlan gráf feszítő erdeje a $T = (V, F)$ gráf

- ha $F \subseteq E$
- valamint T (irányítatlan) erdő (azaz T olyan irányítatlan gráf, aminek mindegyik komponense irányítatlan fa, a fák páronként diszjunktak, és együtt éppen lefedik a G csúcshalmazát)

2. Definíció. A $G = (V, E)$ irányítatlan, összefüggő gráf feszítőfája a $T = (V, F)$ gráf

- ha $F \subseteq E$
- és T (irányítatlan) fa.

Minimális feszítőfák

3. Definíció. Amennyiben $G = (V, E)$ élsúlyozott gráf (fa, erdő stb.) a $w : E \to R$ súlyfüggvénnyel, akkor a G súlya az élei súlyainak összege:

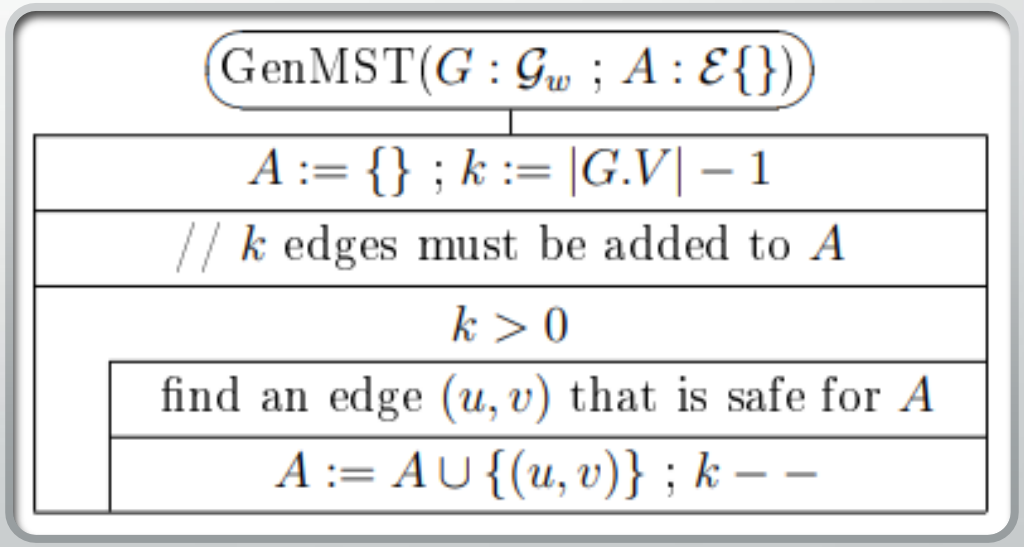
$$w(G) = \sum_{e \in E} w(e)$$

4. Definíció. A G irányítatlan, összefüggő, élsúlyozott gráf minimális feszítőfája T

- ha T a G feszítőfája
- és G bármely T' feszítőfájára: $w(T) \leq w(T')$.

Általános módszer

- Az *A* = { } üres élhalmazból indul
- Ezt bővíti újabb és újabb élekkel
  - Úgy, hogy *A* végig a *G* összefüggő, irányítatlan, élsúlyozott gráf valamelyik minimális feszítőfája élhalmazának a részhalmaza marad
- Az így választott éleket nevezzük az ***A* élhalmazra nézve biztonságosnak (safe for *A*)**
- Amikor az élek száma eléri a |*G.V* |–1 értéket
  - az *A* szükségszerűen feszítőfa
  - és így minimális feszítőfa is lesz



Általános módszer

5. Definíció. Tegyük fel, hogy $G = (V, E)$ élsúlyozott, irányítatlan, összefüggő gráf, és $A \subseteq$ a G valamelyik minimális feszítőfája élhalmazának!

- Ekkor az $(u, v) \in E$ él biztonságosan hozzávehető az A élhalmazhoz (safe for A)
  - ha $(u, v) \notin A$
  - és $A \cup \{(u, v)\} \subseteq$ a G valamelyik (az előzővel nem okvetlenül egyező) minimális feszítőfája élhalmazának.

6. Következmény. Tegyük fel, hogy $G = (V, E)$ élsúlyozott, irányítatlan, összefüggő gráf!

- Ha egy kezdetben üres A élhalmazt
- újabb és újabb biztonságosan hozzávehető éllel bővítünk,

➢ akkor $|G.V|-1$ bővítés után éppen a G egyik minimális feszítőfáját kapjuk meg.

Általános módszer

- Hogyan tudunk mindig biztonságos élet választani az A élhalmazhoz?
  - Ehhez lesz szükségünk az alábbi fogalmakra és tételre.

7. Definíció. Ha $G = (V, E)$ gráf és $\{\} \subsetneq S \subsetneq V$

➢ akkor a G gráfon $(S, V \setminus S)$ egy **vágás**.

8. Definíció. $G = (V, E)$ gráfon az $(u, v) \in E$ él **keresztezi** az $(S, V \setminus S)$ **vágást**,

- ha $(u \in S \wedge v \in V \setminus S) \vee (u \in V \setminus S \wedge v \in S)$.

Általános módszer

9. Definíció. $G = (V, E)$ élsúlyozott gráfon az $(u, v) \in E$ **könnyű él** az (S, V\S) vágásban,

- ha (u, v) keresztezi a vágást,
- és $\forall(p, q)$ a vágást keresztező élre $w(u, v) \leq w(p, q)$.

10. Definíció. A $G = (V, E)$ gráfban az $A \subseteq E$ élhalmazt **elkerüli** az $(S, V \setminus S)$ **vágást**,

- ha az A egyetlen éle sem keresztezi a vágást.

Általános módszer

11. Tétel. Ha a $G = (V, E)$ irányítatlan, összefüggő, élsúlyozott gráfon

1. A részhalmaza a G valamelyik minimális feszítőfája élhalmazának,
2. Az $(S, V \setminus S)$ vágás elkerüli az A élhalmazt, és
3. Az $(u, v) \in E$ könnyű él az $(S, V \setminus S)$ vágásban,

➢ az (u, v) él biztonságosan hozzávehető az A élhalmazhoz.

Általános módszer

- **Bizonyítás.** $(u, v) \notin A$, ui. (u, v) keresztezi az A-t elkerülő vágást.
  - Legyen $T = (V, T_E)$ olyan MST, amire $A \subseteq T_E$

  a) $(u, v) \in T_E$

  ➢ készen vagyunk.

  b) $(u, v) \notin T_E$ esetén:

  - T feszítőfa -> T-ben el lehet jutni u-ból v-be,
  - a T-ben az u-ból a v-be vezető út egyértelmű.

  ➢ Ezen az úton van olyan él, ami keresztezi az $(S, V \setminus S)$ vágást.

  - Legyen (p, q) egy ilyen él!

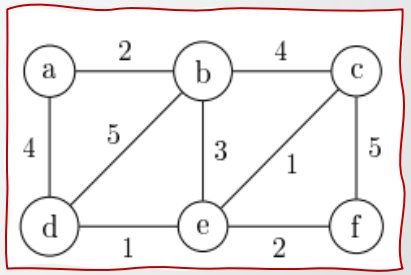
  ➢ $w(p, q) \geq w(u, v) \wedge (p, q) \notin A$.

Általános módszer

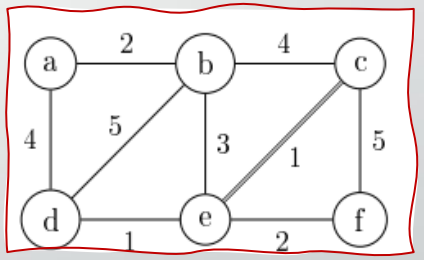
- A (p, q) él törlésével
  - a T MST szétesik (azaz két fából álló „feszítő erdő” lesz, az egyik fában az u, a másikban a v csúccsal)
  - de ha az eredményhez az (u, v) élet hozzávesszük
  - ➢ az így adódó T' újra feszítőfa lesz
- $T' := T \setminus (p, q) \cup (u, v)$.
  - $w(T') = w(T) - w(p, q) + w(u, v) \leq w(T)$
  - T MST volt -> $w(T') \geq w(T)$
  - ➢ $w(T') = w(T)$
  - ➢ T' is MST kell, hogy legyen

Általános módszer szemléltetése

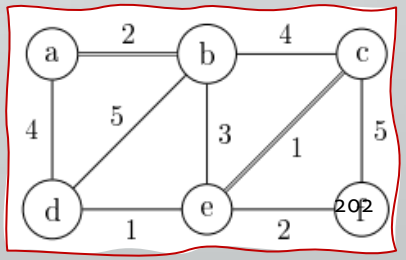
- Az előbbi tétel segítségével már módszeresen tudunk minimális feszítőfát építeni.
  - Jelölés: a gráfban dupla vonallal az *A* élhalmaz elemei
  - Minden lépésben választunk egy *A*-t elkerülő vágást
    - majd ebben egy könnyű élt, amit hozzáveszünk *A*-hoz.
  - A hozzávétel eredménye a következő lépés kiinduló pontja



1. *A*={}
   - L! pl. az ({*a*, *b*, *c*}, {*d*, *e*, *f*}) vágás
   - Egyik könnyű, tehát biztonságos él: (*c*, *e*) -> hozzávesszük *A*-hoz

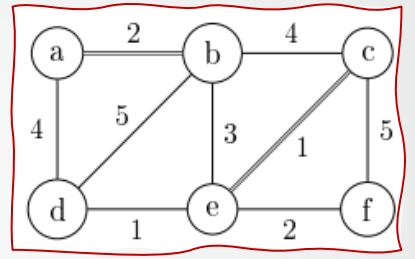


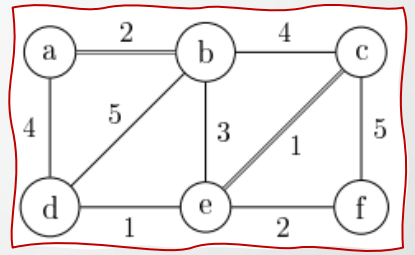
2. *A*={(*c*, *e*)}
   - L! pl. az ({*a*, *f*}, {*b*, *c*, *d*, *e*}) vágás
   - Egyik könnyű, tehát biztonságos él: (*a*, *b*) -> hozzávesszük *A*-hoz

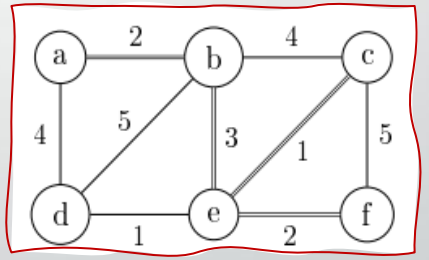


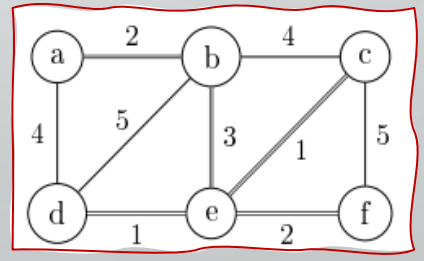
Általános módszer szemléltetése

3. $A=\{(a, b), (c, e)\}$
   - L! pl. az $(\{a, b, c, d, e\}, \{f\})$ vágás
   - Egyik könnyű, tehát biztonságos él: (e, f) -> hozzávesszük A-hoz
4. $A=\{(a, b), (c, e), (e, f)\}$
   - L! pl. az $(\{a, b\}, \{c, d, e, f\})$ vágás
   - Egyik könnyű, tehát biztonságos él: (b, e) -> hozzávesszük A-hoz
5. $A=\{(a, b), (b, e), (c, e), (e, f)\}$
   - L! pl. az $(\{a, b, c, e, f\}, \{d\})$ vágás
   - Egyik könnyű, tehát biztonságos él: (d, e) -> hozzávesszük A-hoz
- $A = \{(a, b),(b, e),(c, e),(d, e),(e, f)\}$.
- $|A| = 5 = |G.V|-1$ -> A egy minimális feszítőfa (MST) élhalmaza









Általános módszer

- A fenti módszer még nem tekinthető szigorú értelemben vett algoritmusnak
  - nem adtuk meg, hogyan válasszunk ki az *A* halmazt elkerülő vágást, és abban könnyű élt
- A **Kruskal** és **Prim-Jarník** algoritmusok éppen ezt teszik
  - $O(m * \lg n)$ (nagyon jó) maximális műveletigénnyel
    - *n:* a gráf csúcsainak száma
    - *m:* az éleinek száma
  - Egyik sem adja meg az *A*-t elkerülő vágást expliciten, a vágásban (az egyik) könnyű élt viszont igen
  - Ilyen módon, mivel lokálisan optimalizálnak, mindkettő **mohó algoritmus**
    - A 11. tétel szerint viszont mindkét algoritmus minimális feszítőfát számol

Kruskal algoritmusa

- $G = (V, E)$ gráf éleit a súlyuk (hosszuk) szerint monoton növekvően veszi sorba
  - Azokat az éleket eldobja, amelyek az A bizonyos éleivel együtt kört képeznének
  - A többit hozzáveszi A-hoz
- Az explicit körkeresés azonban nem hatékony
  - minden egyes e élre bejárást kellene indítani a $(V, A \cup \{e\})$ gráfon.
  - Helyette a következő definíción és invariánson alapuló megoldáshoz folyamodunk:

Kruskal algoritmusa

13. Definíció. A $G = (V, E)$ gráf **feszítő erdeje** a (V, A) gráf

- ha egymástól diszjunkt fákból, mint komponensekből áll,
- és $A \subseteq E$
- (Két fa egymástól diszjunkt, ha nincs közös csúcsuk [és így közös élük sem].)

- A Kruskal algoritmus **invariánsa**
  - (V, A) a $G = (V, E)$ összefüggő, irányítatlan, élsúlyozott gráf feszítő erdeje
  - és A részhalmaza a G valamelyik minimális feszítőfája élhalmazának

Kruskal algoritmusa szövegesen

- $A = \{\}$-val indulunk
  - Jelentése: a kezdeti feszítő erdő fái a $G = (V, E)$ gráf egycsúcsú fái.
- A G gráf éleit a súlyuk (hosszuk) szerint monoton növekvően vesszük sorba
- Tetszőleges élet pontosan akkor veszünk hozzá A-hoz, ha a (V, A) erdő két fáját köti össze
  - Azaz nem egy fán belül fut, és így nem zár be egyetlen kört sem

➢ Minden egyes él hozzávételével eggyel csökken az erdő fáinak száma, de továbbra is feszítő erdőt alkotnak

Kruskal algoritmusa szövegesen

- *G* összefüggő -> bármelyik két fa között van út -> előbb-utóbb összekapcsolódnak -> egy *T* fából áll az erdő, T feszítőfa
- A fenti invariáns miatt
  - a *T* élhalmaza részhalmaza valamelyik *M* minimális feszítőfa élhalmazának
  - A *G* minden feszítőfájának $|V| - 1$ éle van -> a *T* és *M* élhalmaza megegyezik
  - Mindkettő csúcshalmaza *V*

➢ *T* = *M*, azaz *T* minimális feszítőfa.

Invariáns igazságának belátása

- Kezdetben a (*V*, *A*) úgy feszítő erdő, hogy A = {}
  - *A* részhalmaza a *G* bármelyik minimális feszítőfája élhalmazának
  - ➢ az invariáns igaz
- Tekintsünk most az algoritmus futása során egy olyan pillanatot, amikor az invariáns még igaz, és egy újabb *e* élet készülünk megvizsgálni
  - Az *e* a jelenlegi feszítő erdő valamelyik fáján belül fut
  - Az *e* a jelenlegi feszítő erdő két fáját köti össze

Invariáns igazságának belátása

- Ha *e* a jelenlegi feszítő erdő valamelyik fáján belül fut
  - eldobjuk
  - a feszítő erdő nem változik
  - az invariáns igaz marad
- Ha *e* a jelenlegi feszítő erdő két fáját köti össze
  - legyen *S* az egyik fa csúcshalmaza, és tekintsük az (*S*, *V**S*) vágást
    - ez nyilván elkerüli *A*-t
  - Ekkor *e* könnyű él a vágásban
    - A G gráf éleit a súlyuk szerint monoton növekvően vesszük sorba

Invariáns igazságának belátása

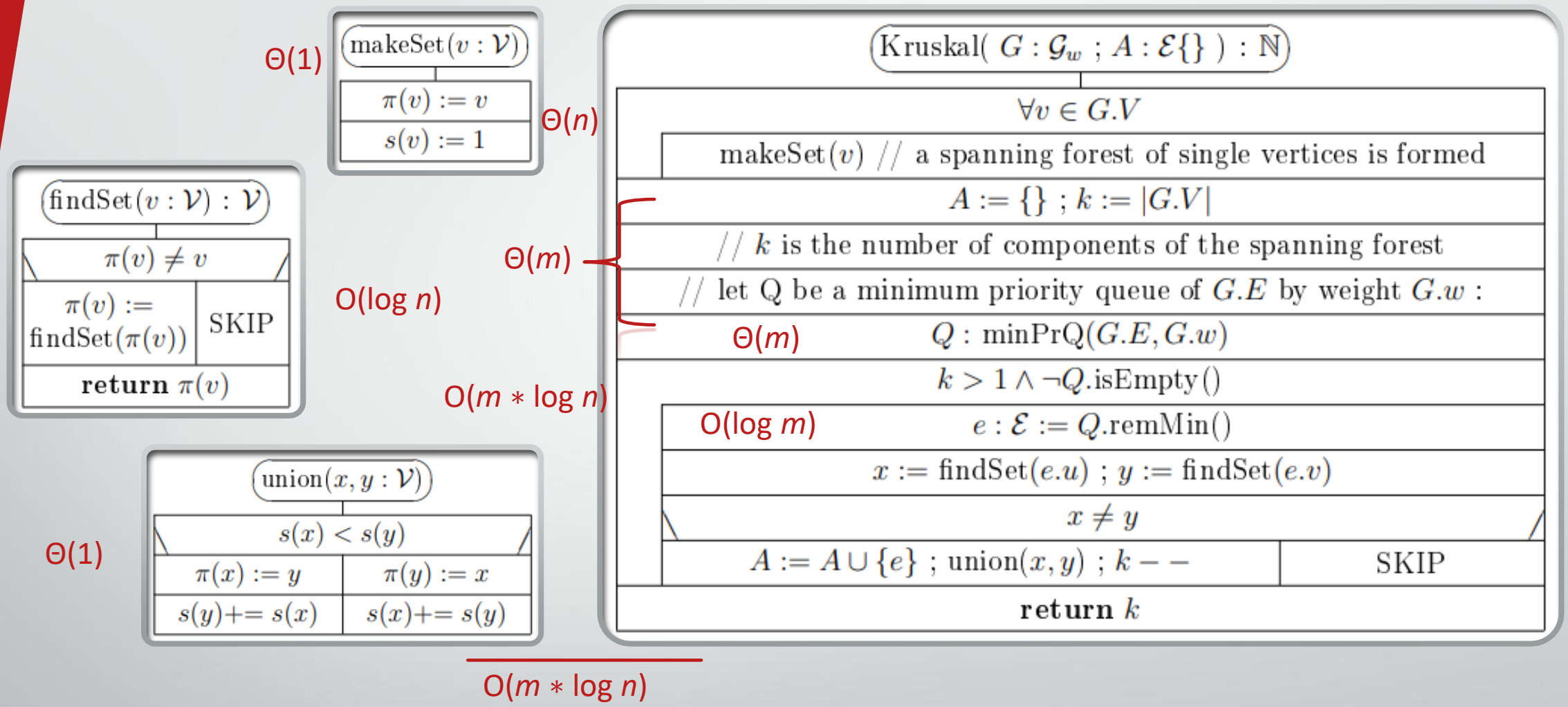
- Ekkor *e* könnyű él a vágásban
  - A G gráf éleit a súlyuk szerint monoton növekvően vesszük sorba
  - ➢ az aktuális élnél kisebb súlyú éleket már korábban feldolgoztuk
    - ezek vagy benne vannak *A*-ban; vagy nincsenek benne *A*-ban
    - de a (*V*, *A*) feszítő erdő egyik fájának két csúcsát kötik össze
    - ➢ eldobtuk őket
- Teljesülnek tehát a 11. tétel feltételei
  - ➢ *e* biztonságosan hozzávehető *A*-hoz, és hozzá is vesszük.

➢ Ez alapján az invariáns az *e* él feldolgozása után is igaz marad

Kruskal algoritmus műveletigénye

- Feltételezések (Biz. 29. dia)
  - makeSet(v) $\in \Theta(1)$
  - union(x, y) $\in \Theta(1)$
  - findSet(v) $\in O(\log n)$
- Tfh. Q prioritásos sort bináris kupaccal valósítjuk meg
  - inicializálása $\in \Theta(m)$
    - a gráf éleit tároljuk benne
  - remMin() $\in O(\log m)$
- ➢ Az első ciklus műveletigénye: $\Theta(n)$
- A két ciklus közti részé: $\Theta(m)$

Kruskal algoritmus stuktogramja és műveletidő



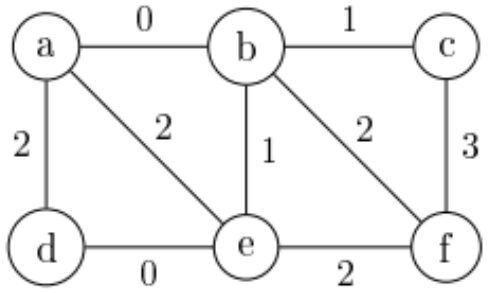
- az algoritmus ellenőrzi, hogy *G* összefüggő-e
  - Ha összefüggő: k = 1
  - Ha nem összefüggő: k > 1

Kruskal algoritmus szemléltetése

a – b, 0 ; d, 2 ; e, 2.
b – c, 1 ; e, 1 ; f, 2.
c – f, 3.
d – e, 0.
e – f, 2.

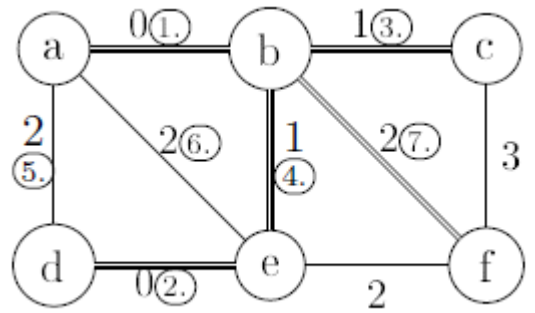
összefüggő

élsúlyozott



irányítatlan gráf

lépés	komponensek	él	biztonságos?
①	a, b, c, d, e, f	a –0– b	+
②	ab, c, d, e, f	d –0– e	+
③	ab, c, de, f	b –1– c	+
④	abc, de, f	b –1– e	+
⑤	abcde, f	a –2– d	–
⑥	abcde, f	a –2– e	–
⑦	abcde, f	b –2– f	+
-	abcdef	-	-

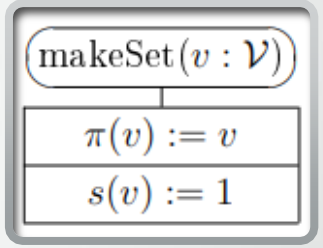


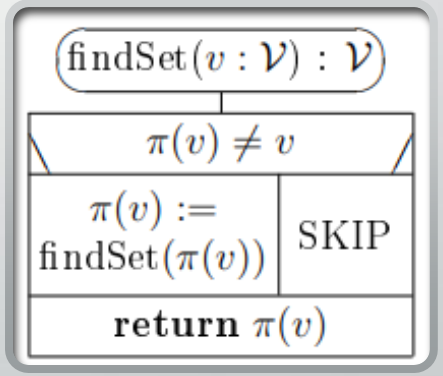
Kruskal algoritmus műveletigénye

- *Fő ciklus*
  - *Egy iterációja* $\in O(\log n)$
    - $e := Q$.remMin() $\in O(\log m) = O(\log n)$
    - G összefüggő és irányítatlan
    - így $n - 1 \leq m \leq n * (n - 1)/2 < n^2$
    - $(\log n) - 1 < \log(n - 1) \leq \log m < \log n^2 = 2 * \log n$
    - $\log m \in \Theta(\log n)$ -> $\Theta(\log m) = \Theta(\log n)$ -> $O(\log m) = O(\log n)$
  - legfeljebb m-szer iterál
  - a teljes műveletigénye $O(m * \log n)$
- Ezt aszimptotikus értelemben már nem módosítja a fő ciklust megelőző inicializálások nála nagyságrenddel kisebb $\Theta(n + m)$ futási ideje.

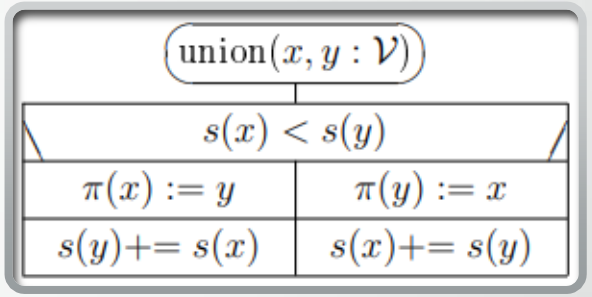
A Kruskal algoritmus halmazműveletei

- makeSet(*v*)
  - a gráf mindegyik *v* csúcsából egyelemű irányított fát képez
- findSet(*v*)
  - megállapítja, hogy a csúcs melyik irányítatlan fában van
    - azaz megkeresi a csúcsot tartalmazó irányított fa gyökércsúcsát
    - [Közben a *v* csúcs és mindegyik őse π mutatóját a gyökércsúcsra állítja]
      - ➢ a későbbi findSet-hívások már majd hatékonyabbak legyenek
      - út-rövidítés tényleges hatékonyságnövekedést hoz magával
        - Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Új Algoritmusok, Scolar Kiadó, Budapest, 2003. ISBN 963 9193 90 9 könyvben részletes elemzés





A Kruskal algoritmus halmazműveletei



- union(*x*, *y*)
  - a megfelelő irányított fák gyökércsúcsait köti össze
    - *x* := findSet(*e.u*) ; *y* := findSet(*e.v*) utasítások az *e* él két végpontjához tartozó gyökércsúcsokat határozzák meg
  - a kisebb méretű irányított fa gyökerét a nagyobb (vagy vele egyenlő méretű) irányított fa gyökere alá köti be
  - Belátjuk, hogy ezzel a módszerrel sosem lesz az egyes fák magassága > log (*fa mérete*)
    - Ebből pedig közvetlenül következik majd a findSet(*v*) függvény műveletigényével kapcsolatos állításunk

Unió-hol van adatszerkezet

- Az irányítatlan feszítőerdő nyilvántartásához egy másik, irányított erdőt is kezelünk
- Irányított erdő
  - Az irányítatlan feszítőerdő minden egyes (irányítatlan) fájának megfelel az irányított erdő egy (irányított) fája
    - Ugyanaz a csúcshalmaza
  - Az irányított fák a gyökerük felé irányítottak

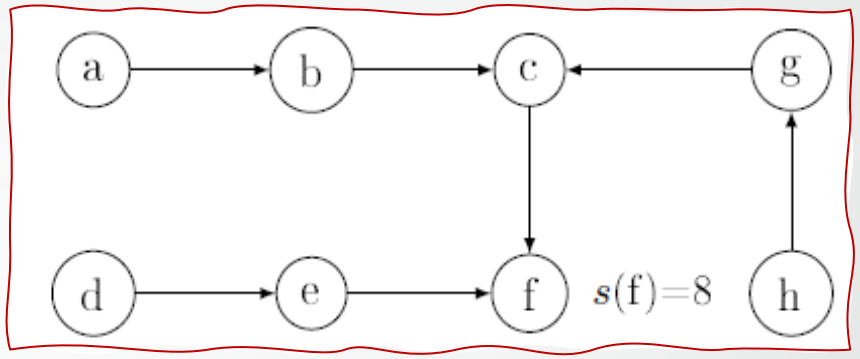
Unió-hol van adatszerkezet

- Mindegyik csúcsnak van egy π címkéje
  - értéke az ő szülője
  - kivéve az irányított fák gyökércsúcsait
    - tetszőleges r gyökércsúcsra viszont $\pi(r) = r$
- $s(r)$: r-hez tartozó fa mérete

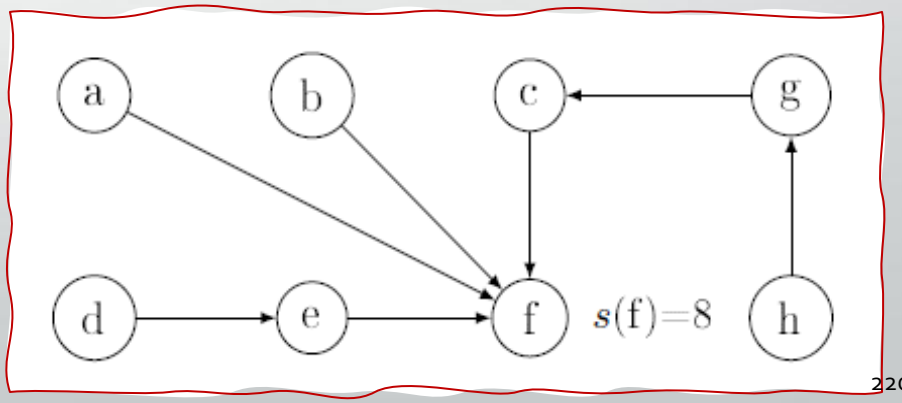
➢ Így mindegyik irányítatlan fát a neki megfelelő irányított fa gyökércsúcsával azonosítjuk

Példa findSet műveletre (útrövidítés)

- findSet(a)
  - Eredménye: *f*
  - Közben módosítja a fát
    - $\pi(a) = \pi(b) = f$
- A példában az önmagára mutató pointerek nincsenek berajzolva

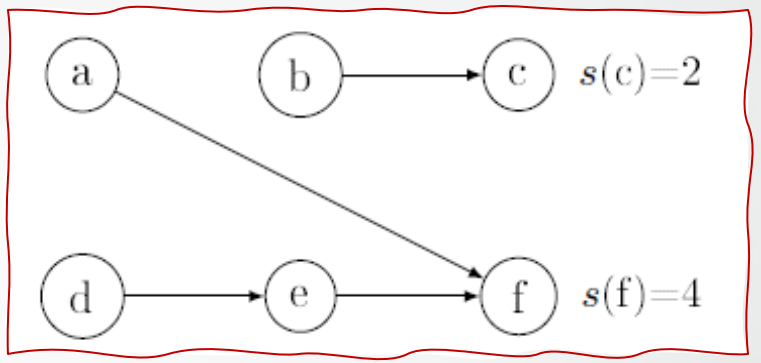


findSet(a)

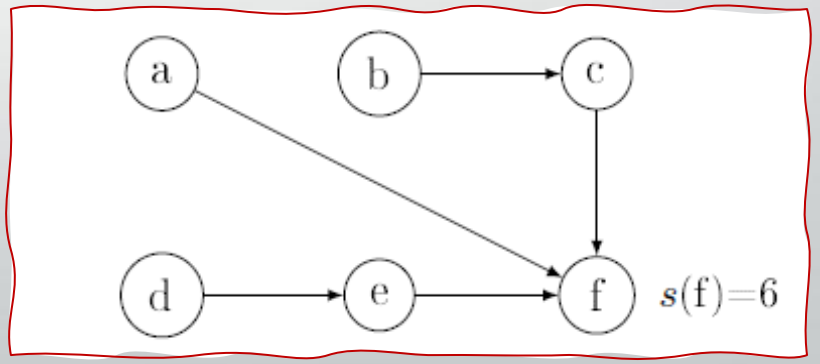


Példa union műveletre

- union(*c*,*f*)
  - *s*(*c*) < *s*(*f*)
    - *f* csúcs alá csatoljuk a *c* csúcsot



union(*c*,*f*)



Unió-hol van asz. fáinak magassága

17. Tulajdonság. A makeSet(v) és a union(x, y) eljárások hatására létrejövő, gyökércsúcsuk felé irányított fák magassága sosem lesz nagyobb, mint a méretük kettes alapú logaritmusa, azaz, ha r egy ilyen fa gyökércsúcsa, $s(r)$ a mérete és $h(r)$ a magassága, akkor $\log s(r) \geq h(r)$.

- **Bizonyítás.**
  - makeSet(v) : egy csúcsú, nulla magasságú fákat hoz létre
    - $\log 1 = 0$
    - ➢ a bizonyítandó tulajdonság kezdetben igaz
  - Elegendő belátni, hogy a union(x, y) eljárás is tartja a fenti egyenlőtlenséget
    - Legyen r a union(x, y) eljárás hatására létrejövő fa gyökere
    - Feltehető, hogy az eljáráshívás előtt $\log s(x) \geq h(x) \wedge \log s(y) \geq h(y)$
    - Azt kell belátnunk, hogy a union(x, y) eljáráshívás után $\log s(r) \geq h(r)$

Unió-hol van asz. fáinak magassága

- Feltehető még, hogy $s(x) \geq s(y)$
  - Ekkor az x gyökércsúcs alá csatoljuk be az y gyökércsúcsot, így $h(r) = \max(h(x), h(y) + 1)$
    - $h(x) > h(y)$
      - $h(r) = h(x) \wedge s(r) \geq s(x)$
      - $\log s(r) \geq \log s(x) \geq h(x) = h(r)$
      - $\log s(r) \geq h(r)$
    - $h(x) \leq h(y)$
      - $h(r) = h(y) + 1 \wedge s(r) = s(x) + s(y) \geq 2 * s(y)$
      - $\log s(r) \geq \log(2 * s(y)) = 1 + \log s(y) \geq 1 + h(y) = h(r)$
      - $\log s(r) \geq h(r)$

Halmazműveletek műveletigénye

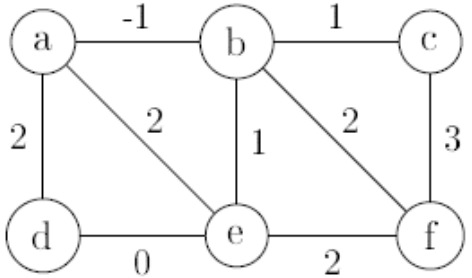
- Kruskal algoritmus műveletigény-számításával kapcsolatos feltételezések jogosságának belátása
  - Jelölések
    - *h:*= a *v* csúcsot tartalmazó irányított fa magassága
    - *s:*= a *v* csúcsot tartalmazó irányított fa mérete
  - findSet(v) ∈ O(log *n*)
    - findSet(v) fv műveletigénye O(*h*)
    - a 17. tulajdonság alapján $h \leq \log s$
    - $s \leq n$

findSet(*v*) fv. futási ideje durva felső becsléssel O(log *n*)

  - makeSet(*v*) ∈ Θ(1) ∧ union(*x*, *y*) ∈ Θ(1)
    - Sem ciklust, sem eljáráshívást nem tartalmaznak

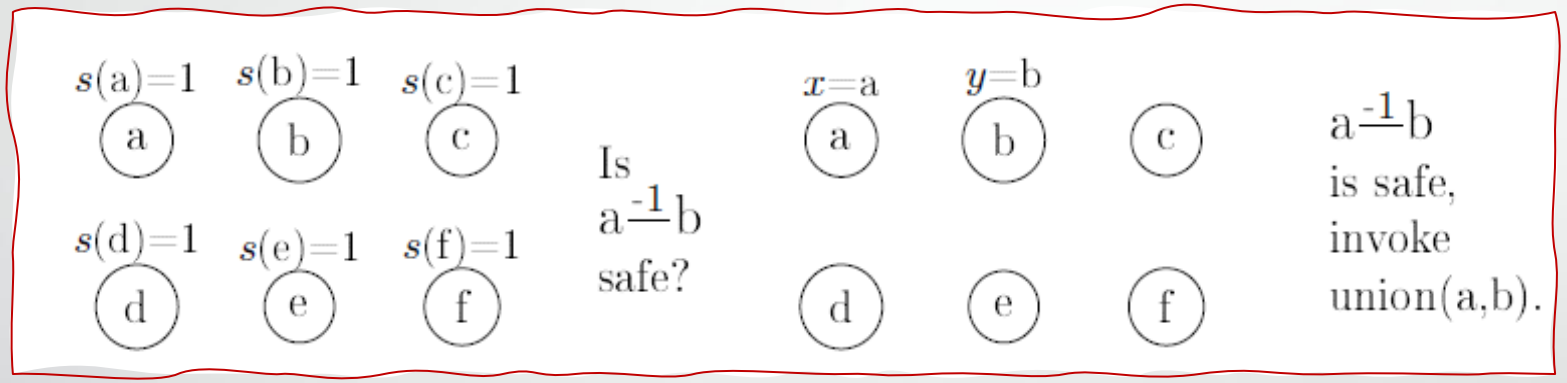
A Kruskal algoritmus grafikus szemléltetése a halmazműveletekkel együtt

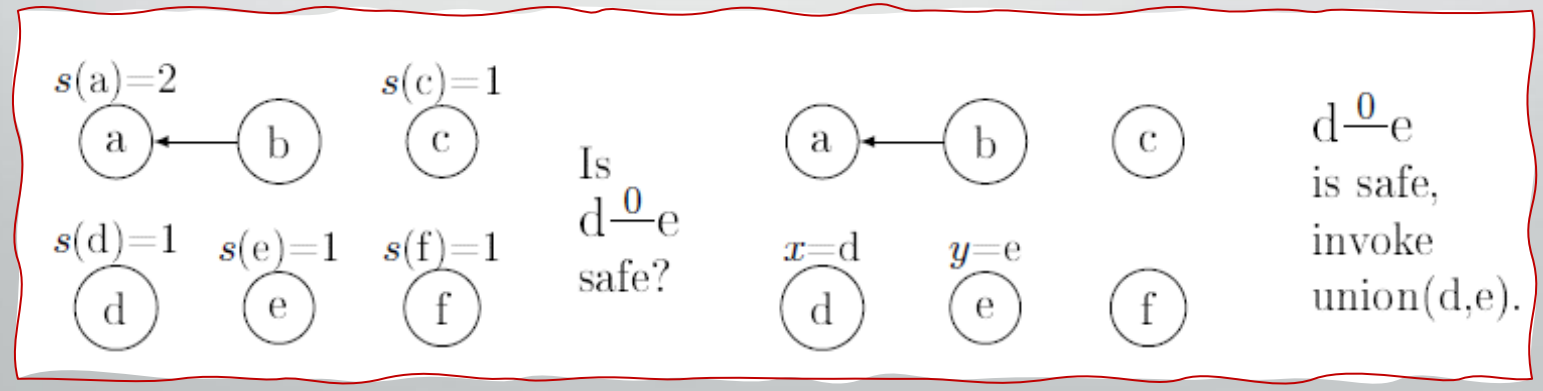
a – b, -1 ; d, 2 ; e, 2.
b – c, 1 ; e, 1 ; f, 2.
c – f, 3.
d – e, 0.
e – f, 2.



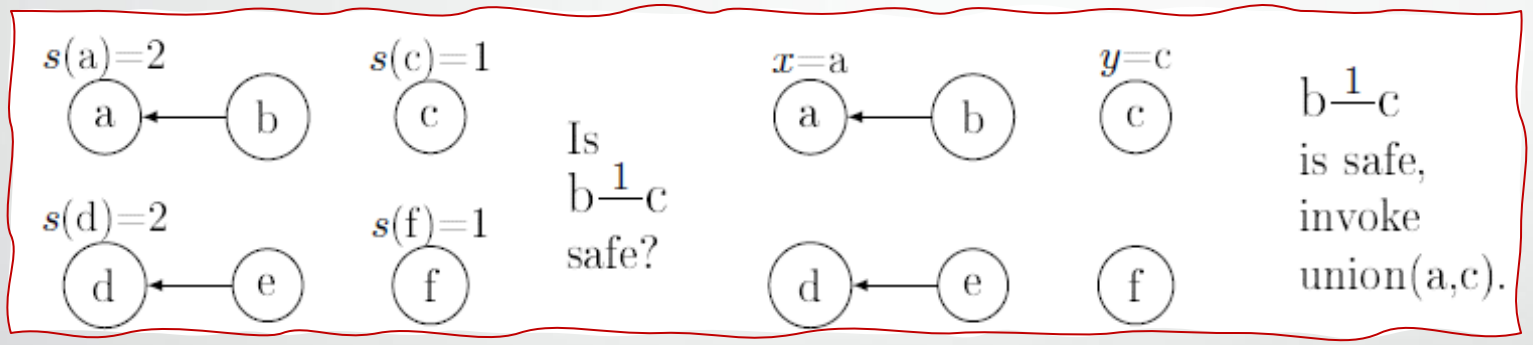
komponensek	a	b	c	d	e	f	$u \overset{w}{—} v$	$u\pi^* s$	$v\pi^* s$	$\pm$
a, b, c, d, e, f	a1	b1	c1	d1	e1	f1	$a \overset{-1}{—} b$	a1	b1	+
ab, c, d, e, f	b	b2					$d \overset{0}{—} e$	d1	e1	+
ab, c, de, f				e	e2		$b \overset{1}{—} c$	b2	c1	+
abc, de, f		b3	b				$b \overset{1}{—} e$	b3	e2	+
abcde, f		b5			b		$a \overset{2}{—} d$	ab5	deb5	–
abcde, f				b			$a \overset{2}{—} e$	ab5	eb5	–
abcde, f							$b \overset{2}{—} f$	b5	f1	+
abcdef		b6				b	-			

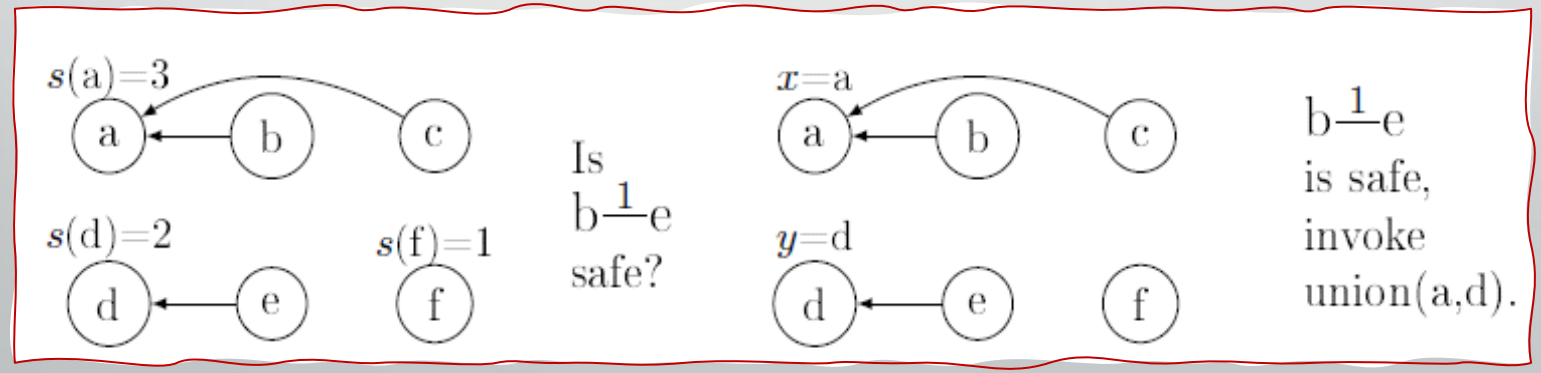
A Kruskal algoritmus grafikus szemléltetése a halmazműveletekkel együtt



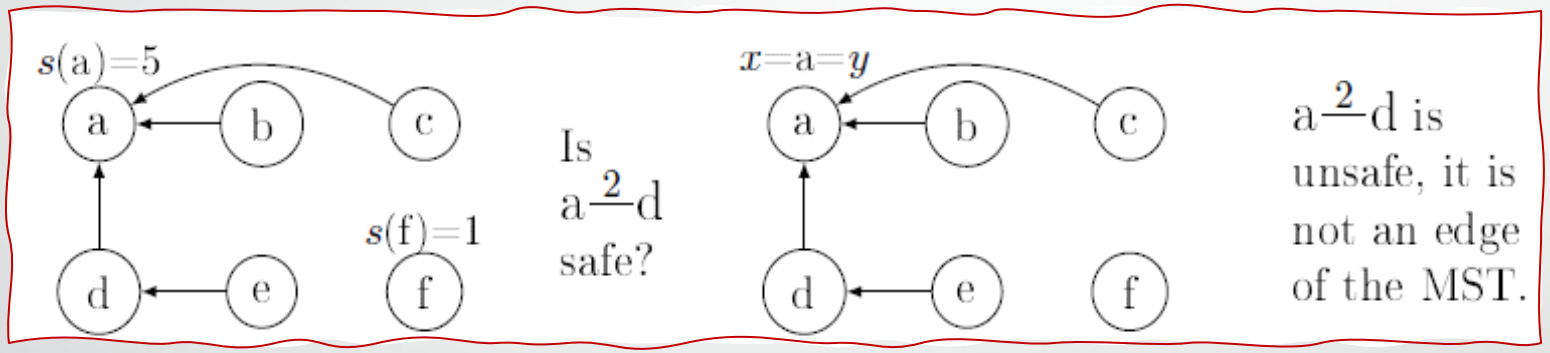


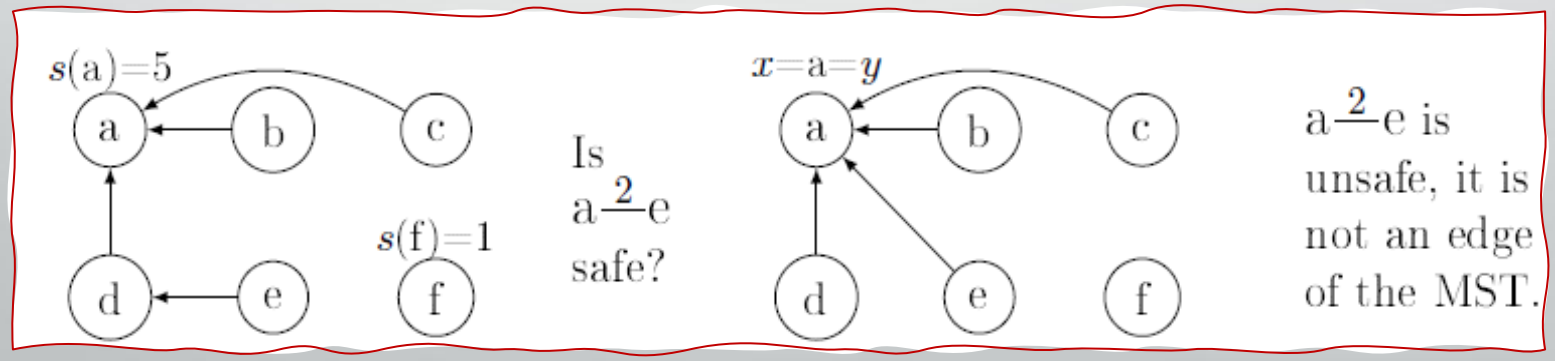
A Kruskal algoritmus grafikus szemléltetése a halmazműveletekkel együtt



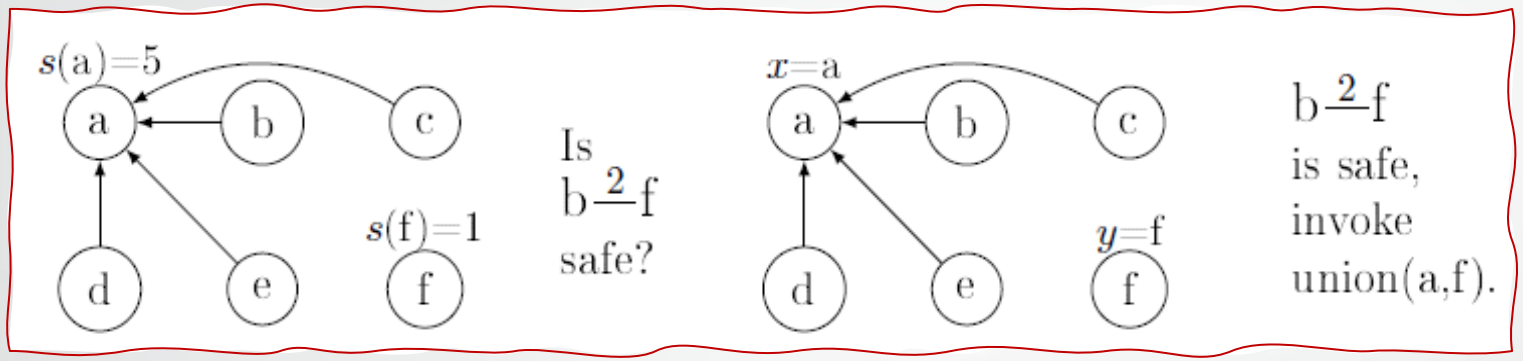


A Kruskal algoritmus grafikus szemléltetése a halmazműveletekkel együtt

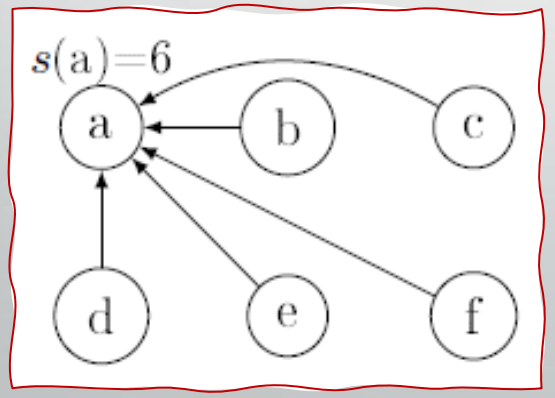


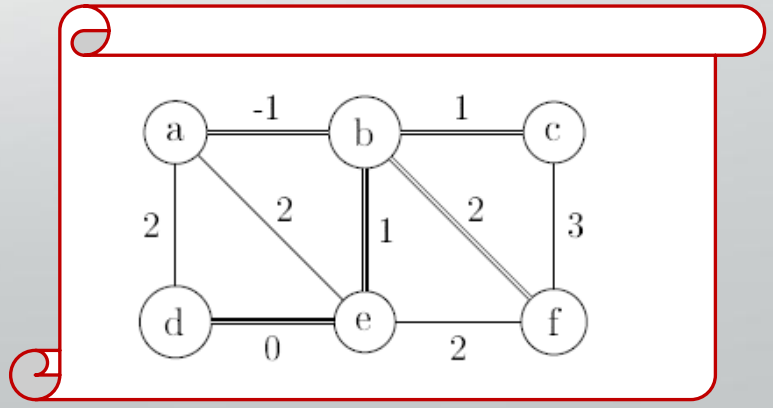


A Kruskal algoritmus grafikus szemléltetése a halmazműveletekkel együtt



Végeredmény:





Ellenőrző kérdések

1. Mit számol ki a *Kruskal* algoritmus?
2. Szemléltesse a működését az alábbi gráfon!
   - *a* - *b*, 0 ; *d*, 1. *b* - *c*, 5 ; *d*, 2 ; *e*, 3. *d* - *e*, 4.
   - Mekkora az algoritmusnak az előadásról ismert műveletigénye, és milyen feltételekkel?
3. Mondja ki a *biztonságos élekről* és a *minimális feszítőfákról* szóló tételt!
   - Definiálja a tételben szereplő *vágás* és könnyű él fogalmakat!
   - Hogyan következik a *Kruskal* algoritmus helyessége ebből a tételből?

Köszönöm a figyelmet!

Pusztai Kinga

A bemutató Ásványi Tibor: Algoritmusok és adatszerkezetek II. eladásjegyzet:Élsúlyozott gráfok és algoritmusaik alapján készült.

Algoritmusok és adatszerkezetek I.
7. Előadás

MST, Prim-Jarník algoritmus.
Legrövidebb utak egy forrásból,
Dijkstra algoritmus.

Tartalom

Prim-Jarník algoritmus

- Kijelölünk a $G = (V, E)$ összefüggő, élsúlyozott, irányítatlan gráfban egy tetszőleges $r \in V$ csúcsot.
- A $T = (\{r\}, \{\})$, egyetlen csúcsból álló fából kiindulva építünk fel egy minimális (V, F) feszítőfát:
  - Minden lépésben a $T = (N, A)$ fához adunk:
    - egy újabb biztonságos élet
    - és hozzá tartozó csúcsot
  - $T = (N, A)$ fa végig ennek a minimális feszítőfának a része marad
    - végig igaz az $N \subseteq V \wedge A \subseteq F$ invariáns
  - Ehhez mindegyik lépésben egy könnyű élet választunk ki az $(N, V \setminus N)$ vágásban. (Ld. 11. tételt!)

Prim-Jarník algoritmus

- A megfelelő könnyű él hatékony meghatározása:
  - egy Q minprioritásos sorban tartjuk nyilván a $V \setminus N$ csúcshalmazt
  - Mindegyik $u \in V \setminus N$ csúcshoz tartozik egy $c(u)$ és egy $p(u)$ címke
  - Ha van él az u csúcs és a T fa N csúcshalmaza között (azaz az $(N, V \setminus N)$ vágásban)
    - a $(p(u), u)$ a gráf egyik éle a vágásban
    - $c(u) = w(p(u), u)$
    - és $\forall\, x$ csúcsra, amelyre (x, u) vágásbeli él, $w(x, u) \geq w(p(u), u)$
    - Ez azt jelenti, hogy $(p(u), u)$ minimális súlyú él azok között, amelyek az u csúcsot a T fához kapcsolják
  - Ha nincs él az u csúcs és a T fa között
    - $p(u) = \varnothing \wedge\ c(u) = \infty$

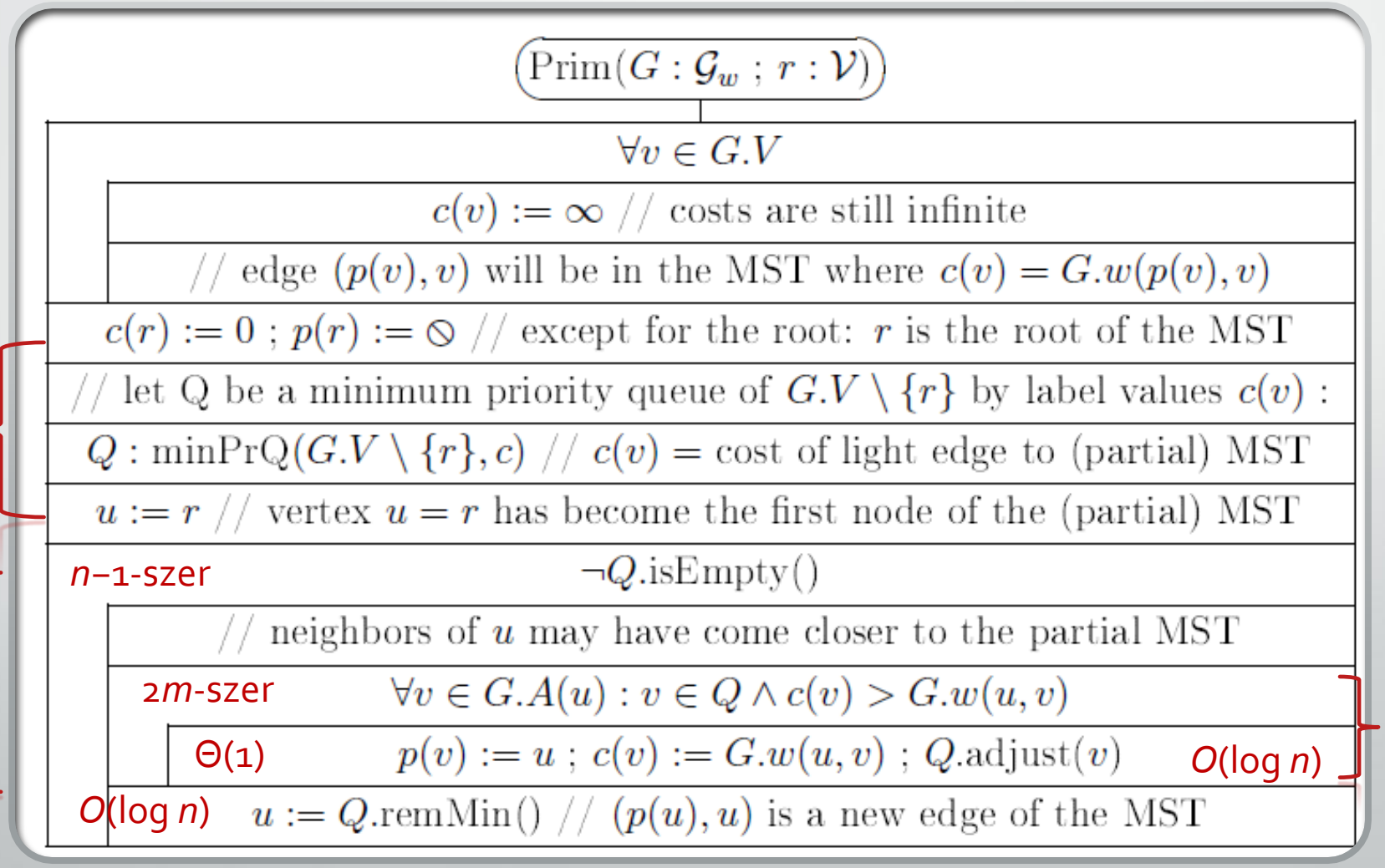
Prim-Jarník algoritmus

- A Q min-prioritásos sorban a csúcsokat a $c(u)$ értékeik szerint tartjuk nyilván
  - u := Q.remMin()
    - Kivesz egy olyan u csúcsot Q-ból, amire a $(p(u), u)$ könnyű él az $(N, V \setminus N)$ vágásban
  - Ezt az élt (11. tétel sz.) biztonságosan adjuk hozzá a T fához
    - azaz T továbbra is kiegészíthető minimális feszítőfává, ha még nem az
- Eredmény:
  - eredetileg egyetlen csúcsból álló T fa
  - $n-1$ db ilyen $(p(u), u)$ él hozzáadásával
  - MST (minimális feszítőfa) lesz

Prim-Jarník algoritmus

- Amikor *u* bekerül a *T* fába, a *Q*-beli *v* szomszédai közelebb kerülhetnek a fához
  - ezeket meg kell vizsgálni
    - ha $w(u, v) < c(v)$
      - -> $c(v) := w(u, v)$ és a $p(v) := u$ utasításokkal aktualizálni kell a *v* csúcs címkéit
    - Ilyen módon a c(v) érték csökkenése miatt a Q helyreállítása is szükségessé válhat
      - Ha például *Q* reprezentációja egy minimum kupac, a *v* csúcsot lehet, hogy meg kell cserélni a szülőjével, esetleg újra és újra, mígnem a szülőjének *c* értéke ≤ $c(v)$ lesz, vagy *v* fölér kupac gyökerébe
- Az alábbi algoritmusban a T fa implicit reprezentációja
  - $N = V \setminus Q$
  - *T* élei: az $N\setminus\{r\}$-beli *x* csúcsok és $p(x)$ címkéik segítségével $(p(x), x)$

Prim algoritmus stuktogramja és műveletigénye



Θ(*n*)

Θ(*n*)

n−1-szer

O(*n* ∗ log *n*)

2*m*-szer

Θ(1)

O(log *n*)

O(log *n*)

O(*m* ∗ log *n*)

∑: O((*n*+*m*) ∗ log *n*) a gráf öf -> m ≥ n−1 -> $MT_{Prim}(n, m) \in O(m * \log n)$

A Prim-Jarník algoritmus műveletigénye

- Feltételek
  - Q reprezentációja bináris minimum kupac
  - $n = |G.V| \wedge m = |G.E|$
- Az első ciklus futási ideje: $\Theta(n)$
- Az első és a második ciklus közti rész műveletigénye
  - a Q kupac inicializálása határoroza meg -> $\Theta(n)$
- A második ciklus
  - mindegyik iterációja kiveszi a Q legkisebb c-értékű elemét
    - $n-1$-szer fut l
    - a belső ciklustól eltekintve mindegyik iterációja $O(\log n)$ időt igényel
    - $O(n * \log n)$.
  - + a belső ciklus futási idejét: $O(m * \log n)$

A Prim-Jarník algoritmus műveletigénye

- A belső ciklus
  - a gráf mindegyik élére legfeljebb kétszer fog lefutni
    - mindegyik élet mindkét végpontja felől megtaláljuk, kivéve azokat, amelyek egyik végpontja a Q-ban utoljára megmaradt csúcs.
    - (A ciklusfejben a $\forall v : (u, v) \in G.A(u)$ rész után következő $v \in Q \wedge c(v) > G.w(u, v)$ szűrő feltétel valójában egy implicit, beágyazott elágazás feltétel része.)
  - Q.adjust(v): $O(\log n)$
  - Többi: $\Theta(1)$
  - ➢ $O(m * \log n)$
- ➢ $\Theta(n) + \Theta(n) + O(n * \log n) + O(m * \log n) = O((n+m) * \log n)$
- Mivel a gráf összefüggő
  - ➢ $m \geq n-1$

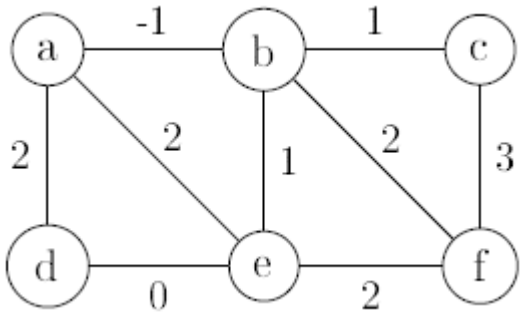
➢ $MT_{\text{Prim}}(n, m) \in O(m * \log n)$

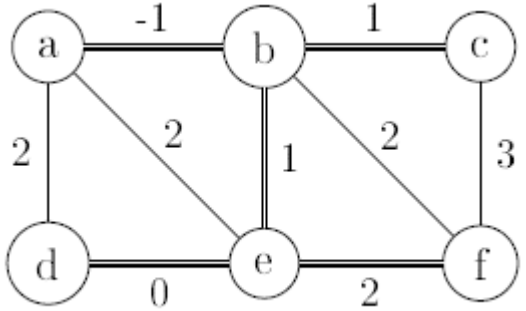
A Prim algoritmus működésének szemléltetése

- A *d* csúcsból indítva

a – b, -1 ; d, 2 ; e, 2.
b – c, 1 ; e, 1 ; f, 2.
c – f, 3.
d – e, 0.
e – f, 2.

in-to MST	changes of c and p labels					
	a	b	c	d	e	f
	∞ ⊘	∞ ⊘	∞ ⊘	0 ⊘	∞ ⊘	∞ ⊘
d	2 d			|	0 d	
e —0— d		1 e		|	|	2 e
b —1— e	-1 b	|	1 b	|	|	
a —-1— b	|	|		|	|	
c —1— b	|	|	|	|	|	
f —2— e	|	|	|	|	|	|





Legrövidebb utak egy forrásból

- **Feladat:** A $G : \mathcal{G}_w$ gráf tetszőlegesen rögzített *s* start csúcsából (source vertex) legrövidebb, azaz optimális utat keresünk mindegyik, az *s*-ből *G*-ben elérhető csúcsba.
  - A most következő algoritmusokban az irányítatlan gráfokat olyan irányított gráfoknak tekintjük, ahol a gráf tetszőleges (*u*, *v*) élével együtt (*v*, *u*) is éle a gráfnak, és *w*(*u*, *v*) = *w*(*v*, *u*).
- A feladat megoldható ⇔ *G*-ben nem létezik *s*-ből elérhető negatív kör
  - azaz olyan kör, amely mentén az élsúlyok összege negatív
  - Irányítatlan gráfoknál: ha nincs a gráfban *s*-ből elérhető negatív él
    - pl. az irányítatlan gráfban (*u*, *v*) *s*-ből elérhető negatív él => <u, *v*, *u*> - *s*-ből elérhető negatív kör

Legrövidebb utak egy forrásból

- Ha a feladat megoldható =>$\forall$ $v \in G.V \setminus \{s\}$ csúcsra két lehetőség
  - Ha létezik út *s*-ből *v*-be
    - $d(v)$: az optimális út hossza
    - $\pi(v)$: egy ilyen optimális úton a *v* csúcs közvetlen megelőzője, szülője
  - Ha nem létezik út s-ből v-be
    - $d(v) = \infty$ és $\pi(v) = \emptyset$
  - Az *s* csúcsra
    - $d(s) = 0$ és $\pi(s) = \emptyset$ (az optimális út csak az *s* csúcsból áll)

Legrövidebb utakat kereső algoritmusok közös vonásai

- L! $s \rightsquigarrow v$: az adott pillanatig kiszámolt s-ből v-be vezető legrövidebb út
- Az inicializálásuk után már teljesülő invariáns
  - $d(v)$ ennek a hossza
  - $\pi(v)$ ($v \neq s$ esetén) ezen az úton a v csúcs közvetlen megelőzője
- Ha még nem számoltunk ki $s \rightsquigarrow v$ utat
  - végtelen hosszúnak tekintjük
  - azaz $d(v) = \infty$ és $\pi(v) =$ Ø

Legrövidebb utakat kereső algoritmusok közös vonásai

- Az optimális utakat fokozatosan közelítjük
  - a gráf (*u*, *v*) éleit szisztematikusan vizsgáljuk
    - ha $s \rightsquigarrow u \longrightarrow v$ rövidebb mint $s \rightsquigarrow v$ => az előbbi lesz az új $s \rightsquigarrow v$ (*közelítés* (relaxation))
    - Kód szinten:

$d(v) > d(u) + G.w(u,v)$	
$\pi(v) := u \ ; \ d(v) := d(u) + G.w(u,v)$	SKIP

- Ismételten kiválasztanak és ki is vesznek az ún. *feldolgozandó* csúcsok halmazából egy csúcsot, és a csúcsból kimenő összes élre végeznek közelítést.
  - Ezeket a közelítéseket együtt a csúcs *kiterjesztésének* nevezzük
  - A csúcs kiválasztását (és kivételét a feldolgozandók közül) a kiterjesztésével együtt a *feldolgozásának* nevezzük.

Legrövidebb utakat kereső algoritmusok különbségei

- Dijkstra algoritmus
  - Kezdetben kiválasztja kiterjesztésre *s*-et
  - A feldolgozandó csúcsok halmazát pedig *G.V* \{*s*}-sel inicializálja
  - Először s-et kiterjeszti
  - Majd a feldolgozandók közül kiválaszt egy olyan csúcsot
    - amibe az adott időpontig a többihez képest legrövidebb utat talált
  - Mohó algoritmus
    - Lokálisan optimalizál
      - Mindig a legígéretesebb csúcsot dolgozza fel
  - Ezáltal mindig olyan csúcsot terjeszt ki, amibe már eddig optimális utat talált
    - egy csúcsot sem kell kétszer kiterjesztenie

Legrövidebb utakat kereső algoritmusok különbségei

- DAGshP algoritmus
  - Először meghatározza az *s*-ből elérhető csúcsokat és egyúttal sorba rendezi ezeket
    - Az adott sorrend szerint feldolgozva őket, mindegyik kiválasztásának az időpontjában már megvan bele az optimális út
  - ➢ Ez is csak egyszer terjeszti ki, és csak az *s*-ből elérhető csúcsokat.
- Az Dijkstra és DAGshP előnyei: a legrosszabb esetben is hatékonyak
  - DAGshP: $\Theta(n + m)$
  - Dijkstra: $O(n + m) * \log n$

Legrövidebb utakat kereső algoritmusok különbségei

- Az előbbi két algoritmusnak speciális előfeltételei vannak.
  - Egyik sem tudja a lehető legáltalánosabb esetben megoldani a legrövidebb utak egy forrásból problémát.
  - A Dijkstra algoritmus elégséges előfeltétele
    - A gráfban ne legyen *s*-ből elérhető negatív súlyú él
  - DAGshP algoritmus elégséges előfeltétele
    - A gráf *s*-ből elérhető része DAG legyen

Legrövidebb utakat kereső algoritmusok különbségei

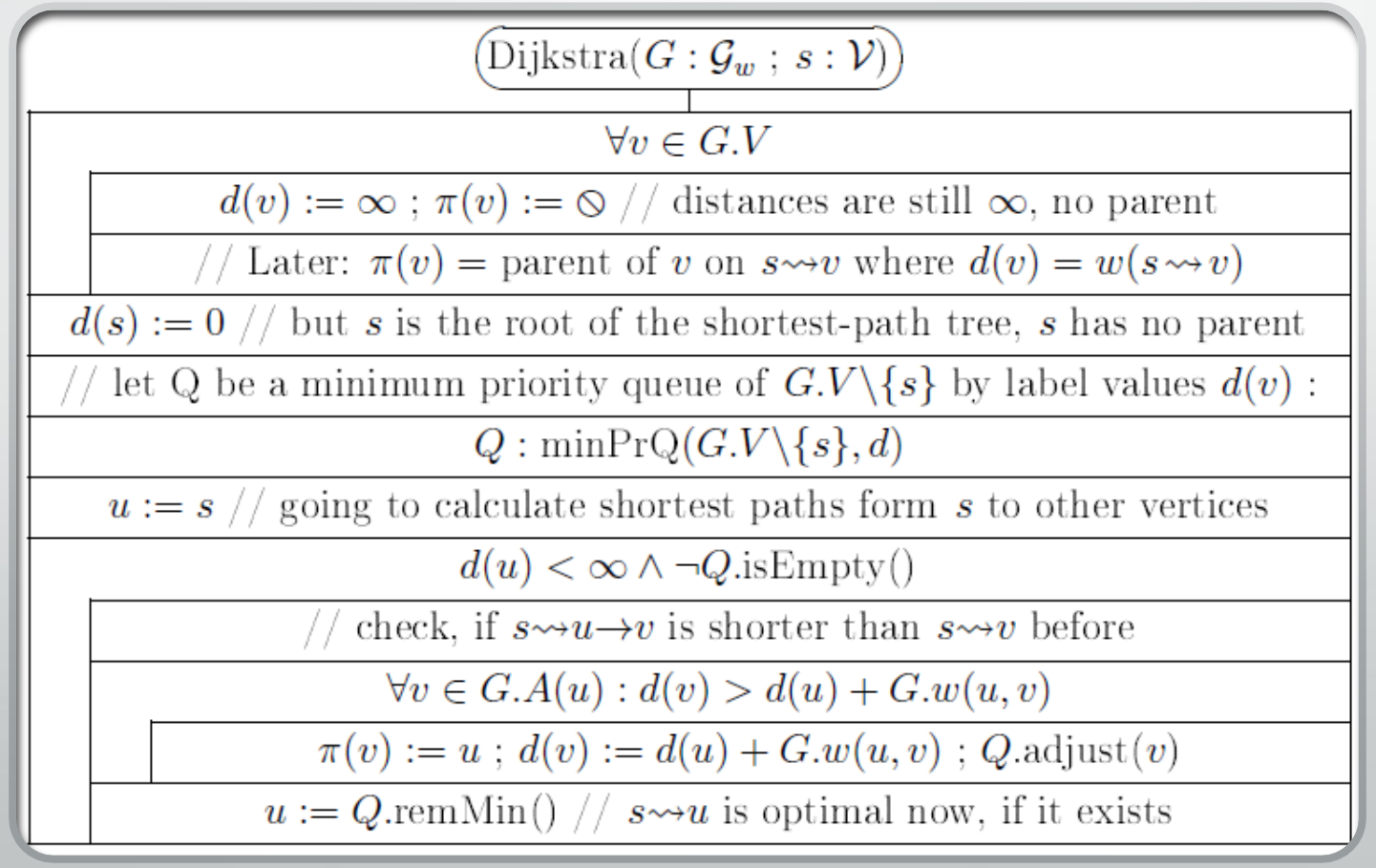
- QBF algoritmus
  - A többivel szemben a lehető legáltalánosabb esetben oldja meg a feladatot
  - Szükséges és elégséges előfeltétele:
    - Ne legyen az *s*-ből elérhető negatív kör
  - Cserébe csak $O(n * m)$ műveletigényt tudunk garantálni
    - ➢ Ugyanazt a csúcsot többször is feldolgozhatja
    - Átlagos esetben ennél sokkal hatékonyabb
      - Implementációikat összehasonlítva sok nemnegatív élsúlyú gráfon a Dijkstra algoritmusnál is gyorsabb

Legrövidebb utakat kereső algoritmusok különbségei

- A DAGshP és a QBF algoritmusok előfeltétele nemtriviális
  - Ezeknél így az algoritmus része
    - az előfeltételének az ellenőrzése
    - a nem megfelelő bemenetek visszautasítása
  - Módja
    - mindkettő megad egy olyan kört a gráfban, ami miatt nem tudja a feladatot megoldani
    - (A QBF ebben az esetben negatív kört ad meg.)

Dijkstra algoritmusa

- **Előfeltétel:** A $\mathcal{G} : \mathcal{G}_w$ gráfban $\forall(u, v) \in G.E$-re $G.w(u, v) \geq 0$, azaz a gráf mindegyik élének élsúlya nemnegatív.
  - Nincs a gráfban negatív kör => a legrövidebb utak egy forrásból feladat megoldható.



Dijkstra algoritmus műveletidő

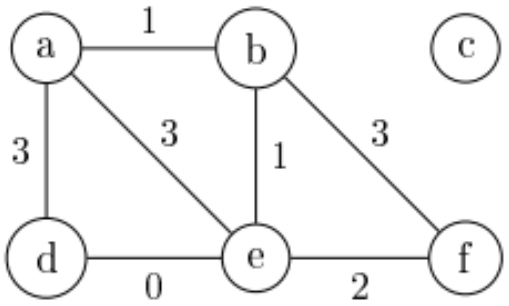
- $MT_{Dijkstra}(n, m) \in O((n + m) * \lg n)$
  - ~ Prim-Jarník algoritmus
  - (két alg. közötti hasonlóság)

```
Dijkstra(G : G_w ; s : V)
    ∀v ∈ G.V
        d(v) := ∞ ; π(v) := ⊘ // distances are still ∞, no parent
        // Later: π(v) = parent of v on s⇝v where d(v) = w(s⇝v)
    d(s) := 0 // but s is the root of the shortest-path tree, s has no parent
    // let Q be a minimum priority queue of G.V\{s} by label values d(v) :
    Q : minPrQ(G.V\{s}, d)
    u := s // going to calculate shortest paths form s to other vertices
    d(u) < ∞ ∧ ¬Q.isEmpty()
        // check, if s⇝u→v is shorter than s⇝v before
        ∀v ∈ G.A(u) : d(v) > d(u) + G.w(u, v)
            π(v) := u ; d(v) := d(u) + G.w(u, v) ; Q.adjust(v)
        u := Q.remMin() // s⇝u is optimal now, if it exists
```

- $mT_{Dijkstra}(n, m) \in \Theta(n)$
  - A 2. ciklus előtti rész műveletigénye már $\Theta(n)$
  - Ehhez hozzáadódik
    - \+ a 2. ciklus egyetlen iterációja
    - \+ a belső ciklus 0-szori iterációja
    - Ha nincs a gráfban *s*-nek rákövetkezője
    - = A 2. ciklus műveletigénye $O(\log n)$ (pontosabban $\Theta(1)$)
      - ez nagysárenddel kisebb, mint $\Theta(n)$

Dijkstra algoritmus szemléltetése

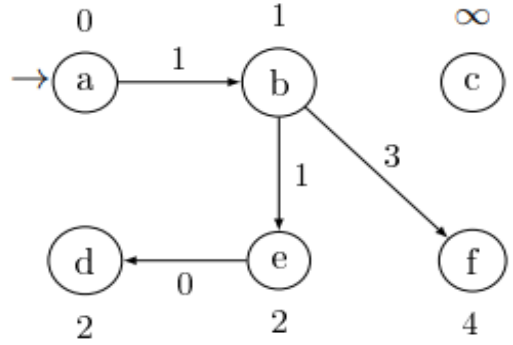
- A Dijkstra algoritmusnál a ki nem terjesztett csúcsokat röviden nyílt csúcsoknak nevezzük.



a – b, 1 ; d, 3 ; e, 3.
b – e, 1 ; f, 3.
c.
d – e, 0.
e – f, 2

ex-pand dv	changes of d and π labels					
	a	b	c	d	e	f
	$0\ \varnothing$	$\infty\ \varnothing$	$\infty\ \varnothing$	$\infty\ \varnothing$	$\infty\ \varnothing$	$\infty\ \varnothing$
0 a	|	1 a		3 a	3 a	
1 b	|	|			2 b	4 b
2 e	|	|		2 e	|	
2 d	|	|		|	|	
4 f	|	|		|	|	|

- A legrövidebb utak fája $s = a$ esetén
- Az s-ből elérhetetlen csúcsot vagy csúcsokat is feltüntetjük, ∞ d értékkel



Dijkstra algoritmus tételek

1. **Tétel.** Amikor az u csúcsot kiválasztjuk kiterjesztésre (először az u := s, később az u := Q.remMin() utasítással), akkor u-ba már optimális utat találtunk, feltéve, hogy d(u) < ∞.

- **Bizonyítás.**
  - u = s-re nyilván igaz az állítás.
  - Tegyük fel indirekt módon, hogy nem mindegyik csúcsra igaz!
    - ∃ v csúcs, amire az algoritmus futása során először lesz igaz, hogy kiválasztjuk kiterjesztésre, de $w(s \overset{opt}{\rightsquigarrow} v) < d(v) < \infty$
      - ahol $s \overset{opt}{\rightsquigarrow} v$ egy *s*-ből *v*-be vezető optimális út, $w(s \overset{opt}{\rightsquigarrow} v)$ pedig ennek a hossza

➢ v ≠ s

Dijkstra algoritmus tételek

- Továbbá, az $s \overset{opt}{\rightsquigarrow} v$ úton az s csúcsot már kiterjesztettük, a v csúcsot viszont még nem
  - Van tehát az $s \overset{opt}{\rightsquigarrow} v$ úton egy első csúcs, amit még nem terjesztettünk ki
    - L! t : $s \overset{opt}{\rightsquigarrow} v$ úton az első csúcs, amit még nem terjesztettünk ki!
    - Szelméletesen: $s \overset{opt}{\rightsquigarrow} t \overset{opt}{\rightsquigarrow} v$.
- Mivel s-et már kiterjesztettük: $t \neq s$.
- Továbbá:
  - az $s \overset{opt}{\rightsquigarrow} t$ úton t közvetlen x megelőzőjét is kiterjesztettük már
  - és az x kiterjesztésekor még teljesült, hogy $d(x) = w(s \overset{opt}{\rightsquigarrow} x)$

Dijkstra algoritmus tételek

- Akkor viszont az $x \longrightarrow t$ él feldolgozása óta már $d(t) = w(s \overset{opt}{\rightsquigarrow} t)$ is teljesül
- Mivel
  - a $t \overset{opt}{\rightsquigarrow} v$ úton minden él súlya (azaz hossza) nemnegatív
  - t az $s \overset{opt}{\rightsquigarrow} v$ úton van, azaz $s \overset{opt}{\rightsquigarrow} t \overset{opt}{\rightsquigarrow} v$

➢ $w(s \overset{opt}{\rightsquigarrow} t) \leq w(s \overset{opt}{\rightsquigarrow} v)$

∑ : $d(t) = w(s \overset{opt}{\rightsquigarrow} t) \leq w(s \overset{opt}{\rightsquigarrow} v) < d(v) < \infty$, azaz $d(t) < d(v)$

- ez ellentmond az indirekt feltételezésnek, ami szerint a v csúcsot választottuk kiterjesztésre.

Dijkstra algoritmus tételek

2. Tétel. Ha a kiterjesztésre kiválasztott u csúcsra $d(u) = \infty$, akkor $Q \cup \{u\}$ egyetlen eleme sem érhető már el az s csúcsból.

- **Bizonyítás.**
  - u az első és egyetlen olyan csúcs, amit $d(u) = \infty$ értékkel választottunk ki, mert ezzel megáll a fő ciklus
  - A korábban kiterjesztésre kiválasztott x csúcsokra tehát $d(x) < \infty$
    - ezekbe az előző tétel szerint már optimális utat találtunk

Dijkstra algoritmus tételek

- Tegyük fel indirekt módon, hogy $Q \cup \{u\}$-nak van olyan v eleme, amely elérhető s-ből
  - Ekkor létezik $s \overset{opt}{\rightsquigarrow} v$ is
  - ezen kell lennie egy első t csúcsnak:
    - eleme $Q \cup \{u\}$-nak
    - de az előző tétel bizonyítása szerint erre már $d(t) = w(s \overset{opt}{\rightsquigarrow} t) < \infty = d(u)$
    - $d(t) < d(u)$
    - ez ellentmond annak, hogy az u csúcsot választottuk ki kiterjesztésre

A tétel következménye

- Amíg tehát a kiterjesztésre kiválasztott u csúcsra $d(u) < \infty$
  - addig olyan csúcsot választunk ki, amelyikbe már optimális utat találtunk
- Ha $d(u) < \infty$ mindegyik kiválasztott csúcsra igaz ⇔
  - a fenti algoritmus 2. ciklusa $n-1$ iterációval kiüríti Q –t
  - megáll, miután a gráf mindegyik csúcsába optimális utat talált
- Ha pedig egyszer a kiterjesztésre kiválasztott u csúcsra már $d(u) = \infty$
  - $Q \cup \{u\}$ egyetlen eleme sem érhető már el az s csúcsból
  - megállhatunk:
    - mindegyik d-értéke végtelen, π-érteke pedig ⦸
    - míg a korábban, véges d-értékkel kiterjesztett csúcsok éppen azok, amelyek elérhetők s-ből, és ezekbe találtunk is optimális utat

Ellenőrző kérdések

1. Implementálja a Dijkstra/Prim-Jarník algoritmust szomszédossági mátrixos / szomszédossági listás gráfábrázolás esetén! A prioritásos sort rendezetlen tömbbel reprezentálja!
   - Mekkora lesz a műveletigény?
   - Lehet-e az aszimptotikus műveletigényen javítani a prioritásos sor kifinomultabb megvalósításával?
2. Mit számol ki a Prim-Jarník algoritmus?
   - Szemléltesse a működését a következő irányítatlan gráfon, a ***d*** csúcsból indítva!
     - *a* - *b*, 0 ; *d*, 1. *b* - *c*, 5 ; *d*, 2 ; *e*, 3. *c* - *e*, 2. *d* - *e*, 2.
   - Mondja ki a biztonságos élekről és a minimális feszítőfákról szóló tételt! Definiálja a tételben szereplő vágás és könnyű él fogalmakat! Hogyan következik a Prim-Jarník algoritmus helyessége ebből a tételből?
   - Mekkora az algoritmusnak az előadásról ismert műveletigénye, és milyen feltételekkel?

Köszönöm a figyelmet!

Pusztai Kinga

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

Algoritmusok és adatszerkezetek I.
8. Előadás

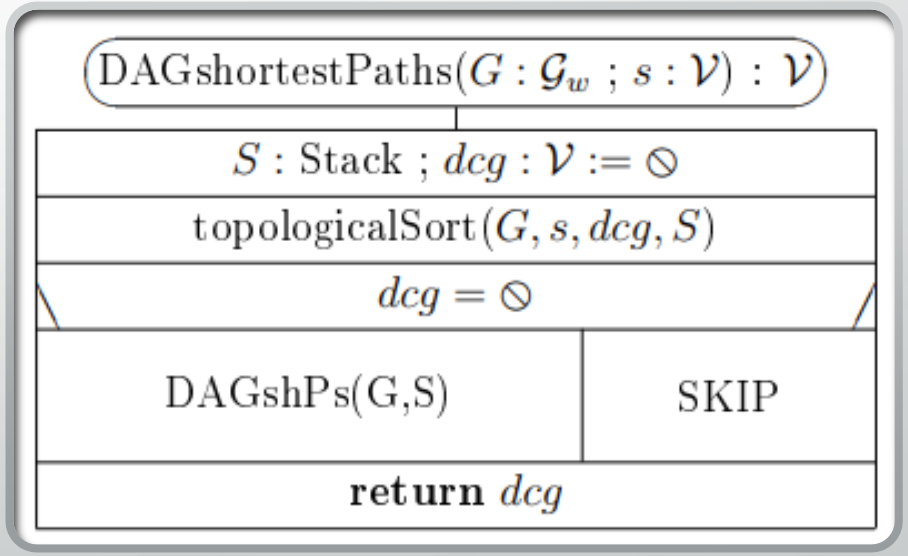
DAG legrövidebb utak egy forrásból. Negatív kör, Sor-alapú Bellman-Ford algoritmus, menetek, negatív kör keresése.

Tartalom

DAG legrövidebb utak egy forrásból (DAGshP)

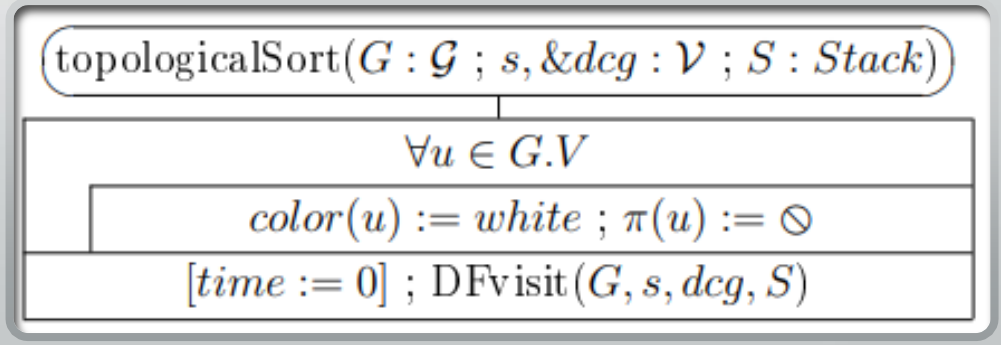
- **Előfeltétel:** A $\mathcal{G} : \mathcal{G}_w$ gráf irányított, és a gráfban nincs s-ből elérhető irányított kör
  - Nincs a gráfban *s*-ből elérhető negatív kör -> a legrövidebb utak egy forrásból feladat megoldható.

- Az algoritmus ellenőrzi az előfeltételét
  - Ha teljesül, a DAGshortestPaths() függvény $\oslash$ értékkel tér vissza
  - Különben megtalál egy irányított kört, aminek egyik csúcsával tér vissza
  - Ebből indulva a π címkék mentén a kör fordított irányban bejárható



DAG legrövidebb utak egy forrásból (DAGshP)

- A topologikus rendezés csak az *s*-ből elérhető részgráfot próbálja topologikusan rendezni.
- A *time* nevű változóra és a hozzá kapcsolódó utasításokra csak a topologikus rendezés szemléltetésének megkönnyítése végett van szükségünk.
  - Ezek az implementációkból elhagyhatók
    - a vonatkozó utasítások szögletes zárójelben
  - Time: az egyszerűség kedvéért globális



DAG legrövidebb utak egy forrásból (DAGshP)

DFvisit$(G : \mathcal{G}\ ;\ u : \mathcal{V}\ ;\ \&time : \mathbb{N})$

- $d(u) := ++time\ ;\ color(u) := grey$
- $\forall v \in G.A(u)$
  - $color(v) = white$
    - $\pi(v) := u$
    - DFvisit $(G, v, time)$
  - $color(v) = grey$
    - backEdge (u, v)
  - SKIP
- $f(u) := ++time\ ;\ color(u) := black$

$$MT(n,m) \in \Theta(n+m)$$

$$mT(n,m) \in \Theta(n)$$

DAGshPs$(G : \mathcal{G}_w\ ;\ S : \text{Stack})$

- $\forall v \in G.V$
  - $d(v) := \infty\ ;\ \pi(v) := \oslash$ // distances are still ∞, parents undefined
  - // $\pi(v)$ = parent of v on $s \rightsquigarrow v$ where $d(v) = w(s \rightsquigarrow v)$
- $s := S.\text{pop}()$
- $d(s) := 0$ // s is the root of the shortest-path tree
- $u := s$ // going to calculate shortest paths form s to other vertices
- $\neg S.\text{isEmpty}()$
  - // check, if $s \rightsquigarrow u \rightarrow v$ is shorter than $s \rightsquigarrow v$ before
  - $\forall v \in G.A(u) : d(v) > d(u) + G.w(u,v)$
    - $\pi(v) := u\ ;\ d(v) := d(u) + G.w(u,v)$
  - $u := S.\text{pop}()$ // $s \rightsquigarrow u$ is optimal now

DAGshP tételek

1. **Tétel.** A legrövidebb utak egy forrásból algoritmus az *s*-ből elérhető csúcsokba optimális utat talál. Az *s*-ből el nem érhető *x* csúcsokra $d(x) = \infty$ és $\pi(x) = \oslash$ lesz.

- **Bizonyítás.**
  - Az *s*-ből indított részleges topologikus rendezés
    - az *s*-ből elérhető csúcsokat teszi az *S* verembe
      - a megfelelő sorrendben

Bizonyítás folytatása

- pontosan ezeket a csúcsokat terjesztjük ki, a topologikus sorrendnek megfelelően
- A többi x csúcsra az inicalizálás eredményeként $d(x) = \infty$ és $\pi(x) = \varnothing$ marad
  - ui. ha egy csúcs nem érhető el s-ből, akkor egyetlen G-beli megelőzője sem
- Elegendő belátni, hogy amikor egy u csúcsot kiválasztunk kiterjesztésre
  - (először az $u := s$, később az $u := S.pop()$ utasítással)
  - u-ba már optimális utat találtunk
- Az előbbi állítás $u = s$ esetben nyilván igaz

Bizonyítás folytatása

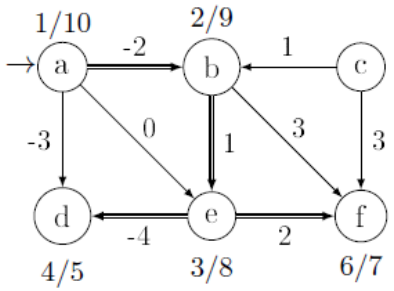
- Tegyük fel, hogy adott $v \neq s$ csúcs kiválasztása előtt mindegyik kiterjesztésre kiválasztott csúcsra igaz volt
- Mivel a v csúcsnak topologikus sorrend szerinti és így a G-beli megelőzőit is a v kiválasztásának pillanatában már kiterjesztettük a v csúcsnak adott $s \overset{opt}{\rightsquigarrow} v$ úton vett közvetlen u megelőzőjét is
  - mégpedig úgy, hogy u-ba a kiterjesztésekor már optimális utat találtunk
  - így legkésőbb az $u \longrightarrow v$ él feldolgozásakor v-be is optimális utat találtunk már

A DAG legrövidebb utak egy forrásból algoritmus szemléltetése

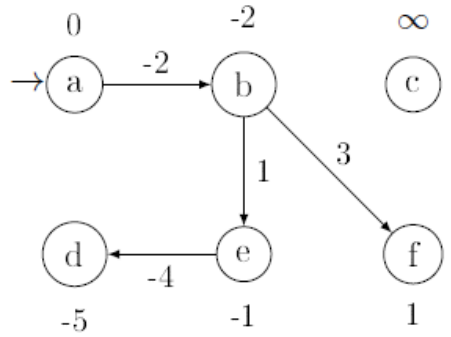
- $s = a$
- Először topologikusan rendezzük az s-ből elérhető részgráfot
  - (A dupla nyilak a mélységi fa fa-éleit jelölik.)
- Ezután a szokásos inicializálásokat követően a csúcsokat a topologikus sorrendjüknek (S) megfelelően terjesztjük ki

ex-pand $d\,\mathcal{V}$	changes of d and π labels					
	a	b	c	d	e	f
	$0\,\oslash$	$\infty\,\oslash$	$\infty\,\oslash$	$\infty\,\oslash$	$\infty\,\oslash$	$\infty\,\oslash$
0 a		-2 a		-3 a	0 a	
-2 b					-1 b	1 b
-1 e				-5 e		
1 f						

a → b, -2 ; d, -3 ; e, 0.
b → e, 1 ; f, 3.
c → b, 1 ; f, 3.
e → d, -4 ; f, 2.
$S = \langle$ a, b, e, f, d $\rangle$



- A legrövidebb utak fája $s = a$ esetén
- Az s-ből elérhetetlen csúcsot vagy csúcsokat is feltüntetjük, ∞ d értékkel



Sor-alapú (Queue-based) Bellman-Ford algoritmus (QBF)

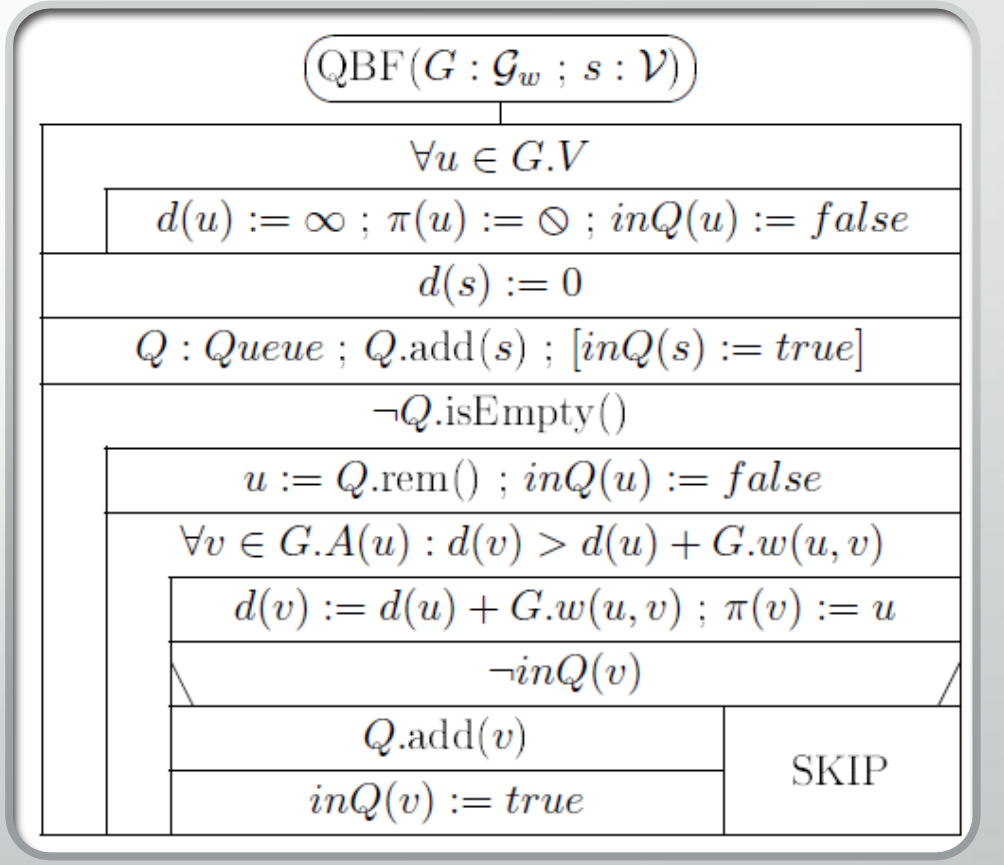
- **Előfeltétel:** Nincs az *s*-ből elérhető negatív kör.

  (Irányított gráf esetén lehetnek negatív élsúlyok.)

- Tarján Róbert Endre amerikai matematikus elemezte és publikálta
  - Breadth-first Scanning néven
- QBF ~ szélességi keresés
  - de az utak hosszát a Dijkstra, a DAGshP és a Bellman-Ford algoritmushoz hasonlóan
  - mint az út menti élsúlyok összegét határozza meg

Sor-alapú (Queue-based) Bellman-Ford algoritmus (QBF) működése

- Kezdetben csak az *s* start (forrás) csúcsot teszi a sorba.
- Inicializálások ~BFS-hez
- Fő ciklus: a sor első elemét kiveszi és kiterjeszti,
- Különbség: mindegyik közvetlen rákövetkezőjére vizsgálat
  - nem talált-e bele rövidebb utat mint eddig
- Ha igen: megfelelően módosítja a *d* és a *π* értékét
- Ha a módosított *d* és *π* értékű csúcs pillanatnyilag nincs benne a sorban, beteszi a sor végére



QBF működése

- A sor (Queue) adattípusra nincs „∈” műveletünk
  - vagy átlátszó sor típust kellene bevezetnünk és megvalósítanunk
  - vagy minden *v* csúcsra értelmezünk egy $inQ(v)$ logikai értékű címkét
    - Igaz ⇔ amikor *v* eleme a sornak
  - Implementációja:
    - feltéve, hogy a csúcsokat 1-től *n*-ig sorszámoztuk
    - $inQ/1 : \mathbb{B}[n]$

QBF működése

- Eredmény:
  - Ha nincs a gráfban *s*-ből elérhető negatív kör:
    - minden *s*-ből elérhető *v* csúcsra meghatározott egy $s \overset{opt}{\rightsquigarrow} v$ utat
    - üres sorral megáll
  - Ha a gráf tartalmaz az *s*-ből elérhető negatív kört
    - a kiterjesztések egy ilyen mentén „körbejárnak”
    - ➢ a sor sosem ürül ki
    - ➢ és az algoritmus végtelen ciklusba kerül

A negatív körök kezelése

- Bevezetjük a gráf v csúcsaira az $e(v)$ címkét
  - Ennyi élt tartalmaz s $\rightsquigarrow$ v
- Ha nincs a gráfban s-ből elérhető negatív kör $\Leftrightarrow$ QBF futása során minden, a fő ciklusban kezelt v csúcsra $e(v) < n$
  - Fordítva: Ha van a gráfban s-ből elérhető negatív kör $\Leftrightarrow$ a QBF futása során van olyan, a fő ciklusban kezelt v csúcs, amelyre $e(v) \geq n$.
- Ha $e(v) \geq n$
  - v csúcs része egy negatív körnek, amit a csúcsok π címkéi mutatnak meg
  - vagy a π címkéken keresztül el lehet belőle jutni egy ilyen negatív körbe, ami gráf csúcsainak n száma miatt összesen legfeljebb n lépésben azonosítható

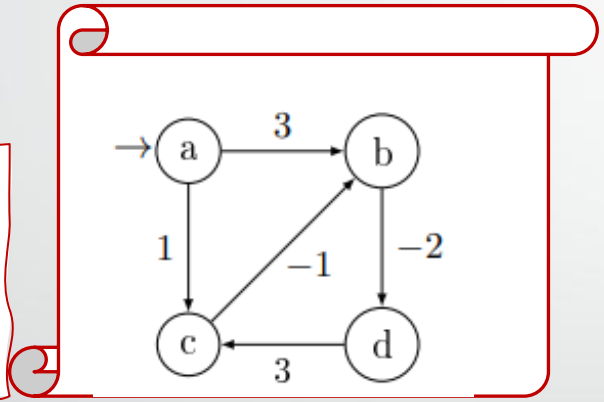
Változtatások az algoritmusban a negatív körök kezeléséhez

- Az *s*-ből elért *v* csúcsokra a $d(v)$, a $\pi(v)$ és az $inQ(v)$ mellett az $e(v)$ címkét is nyilvántartjuk
- Ha valamely (u, v) él mentén végzett közelítés során $e(v) \geq n$
  - meghatározzuk valamelyik negatív kör egyik csúcsát
  - az algoritmus ezzel tér vissza
- Ha a QBF futása során mindegyik közelítés után $e(v) < n$
  - üres sorral állunk meg
  - Az algoritmus ⦸ extremális hivatkozással tér vissza
- Az így kiegészített QBF műveletigénye: $O(n*m)$

A legrövidebb utak fája meghatározásának szemléltetése

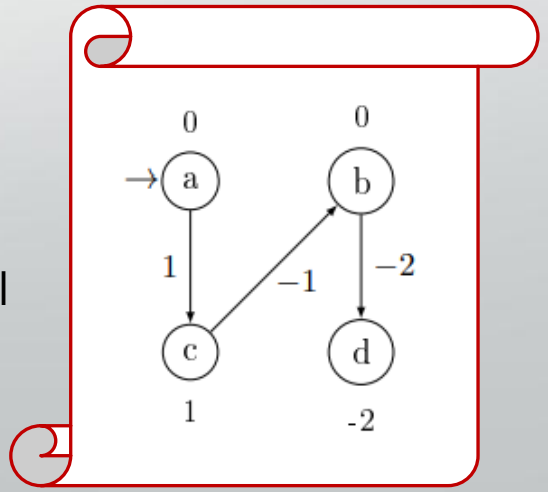
- Mindegyik *s*-ből elérhető *v* csúcsra kiszámítunk egy $s \overset{opt}{\rightsquigarrow} v$ utat

$a \to b,\ 3\ ;\ c,\ 1.$
$b \to d,\ -2.$
$c \to b,\ -1.$
$d \to c,\ 3.$



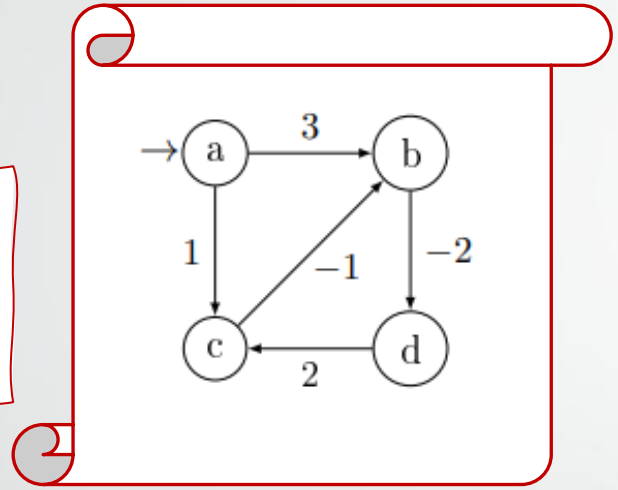
ex-pand	changes of $d\,\pi\,e$				Q :
	a	b	c	d	Queue
$d\,\mathcal{V}\,e$	0 ⊘ 0	∞ ⊘	∞ ⊘	∞ ⊘	⟨ a ⟩
0 a 0		3 a 1	1 a 1		⟨ b, c ⟩
3 b 1				1 b 2	⟨ c, d ⟩
1 c 1		0 c 2			⟨ d, b ⟩
1 d 2					⟨ b ⟩
0 b 2				−2 b 3	⟨ d ⟩
−2 d 3					⟨ ⟩

- A legrövidebb utak fája *s* = *a* esetén
- A csúcsoknál csak a *d* értékeket tüntetjük fel



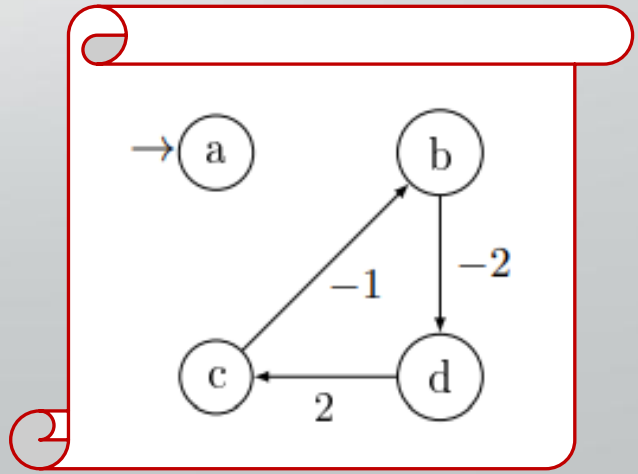
A negatív körök kezelésének szemléltetése

a → b, 3 ; c, 1.
b → d, −2.
c → b, −1.
d → c, 2.



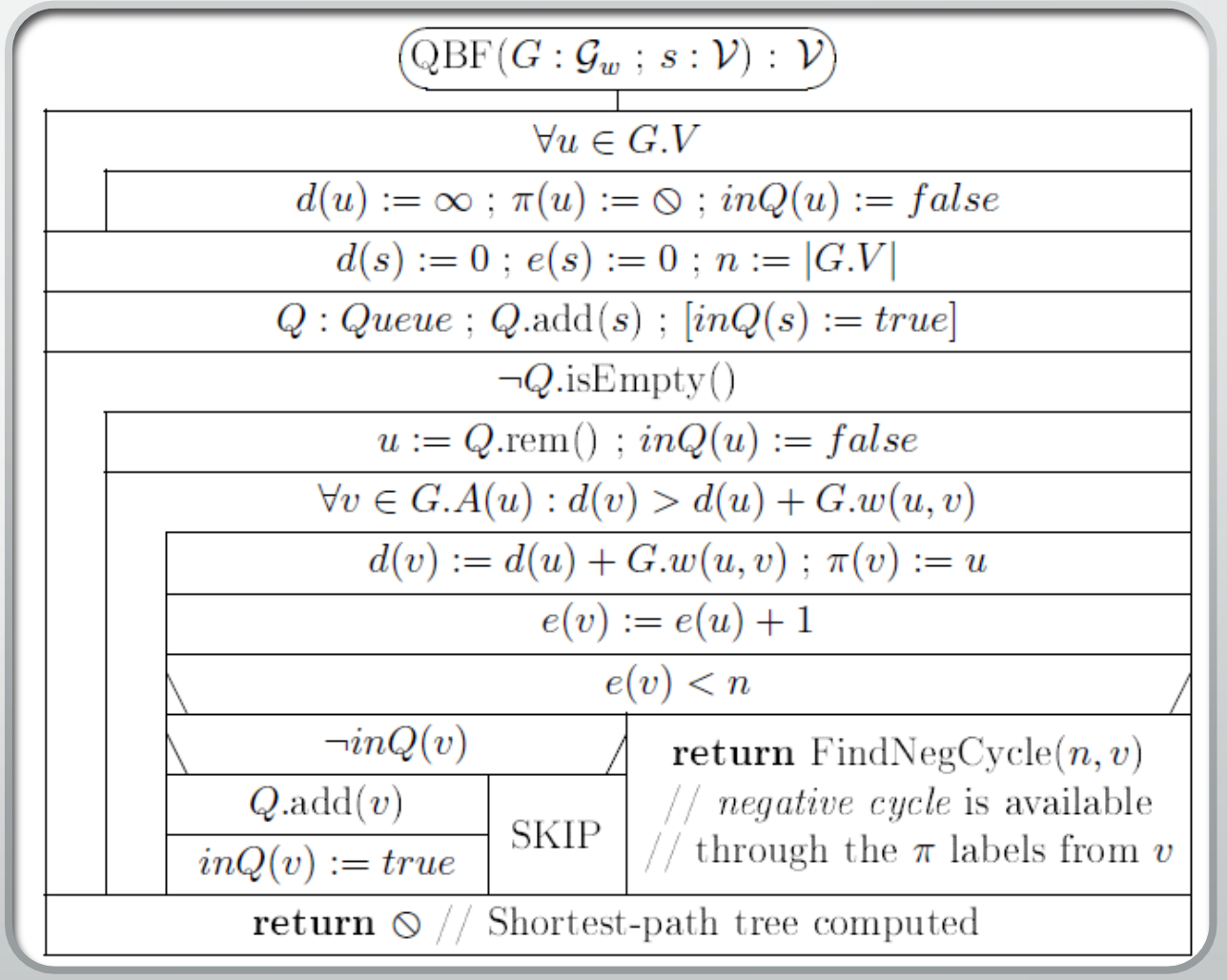
ex-pand $d\,\mathcal{V}\,e$	changes of $d\,\pi\,e$ a	b	c	d	Q : Queue
	0 ⊘ 0	∞ ⊘	∞ ⊘	∞ ⊘	⟨ a ⟩
0 a 0		3 a 1	1 a 1		⟨ b, c ⟩
3 b 1				1 b 2	⟨ c, d ⟩
1 c 1		0 c 2			⟨ d, b ⟩
1 d 2					⟨ b ⟩
0 b 2				−2 b 3	⟨ d ⟩
−2 d 3			0 d ④		⟨ ⟩

- Itt megállunk, mert $e(c) = 4 = n$
  - negatív kör található a *c* csúcs ősei közt
- A π értékek szerint visszafelé haladva megtalálhatjuk a negatív kört

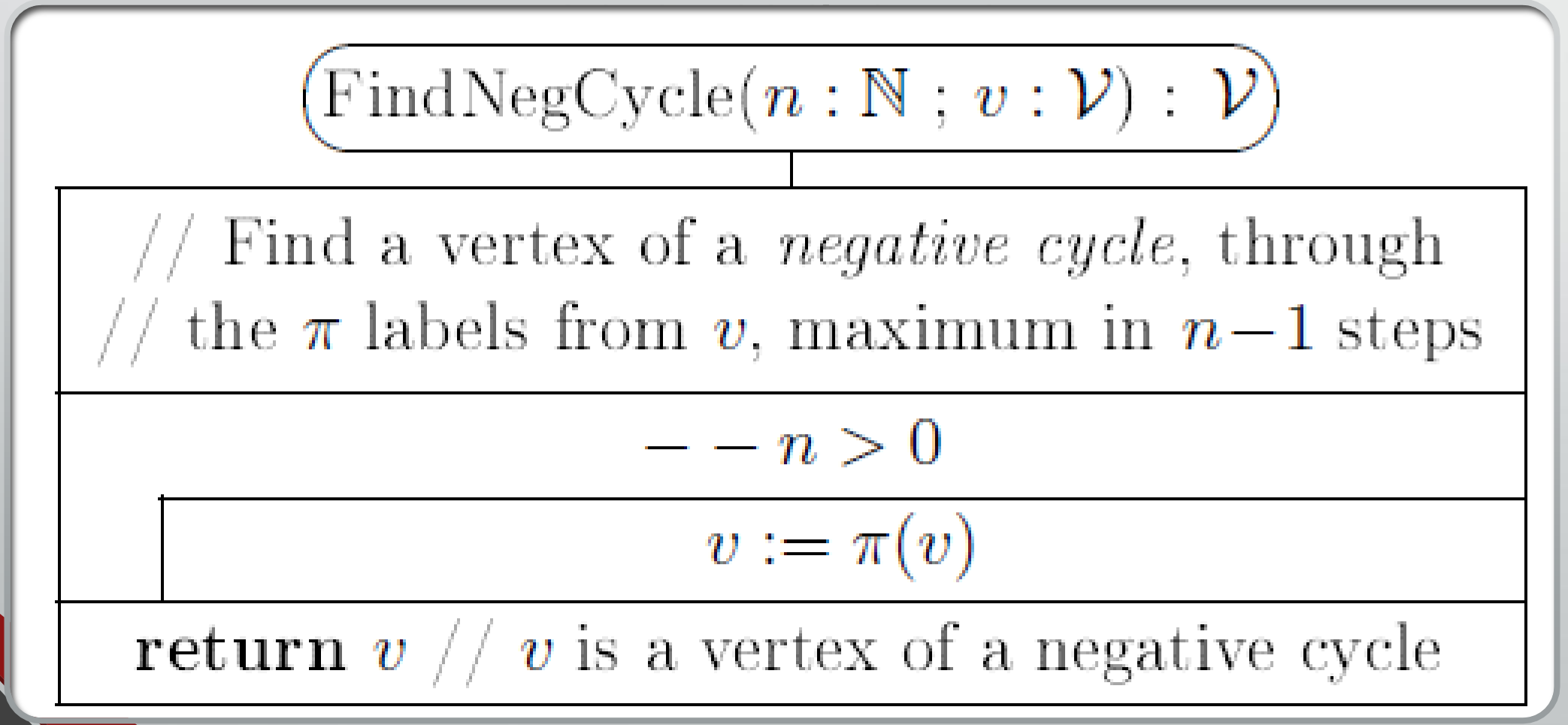


A negatív körök kezelésével kiegészített struktogramok

- Egy tetszőleges *v* csúcs *d*, *π* és *e* címkéjének módosítása után a *v* csúcsot a sor végére tesszük ⇔
  - $e(v) < n$
  - *v* nincs benne a sorban
- Ha biztosan nincs a gráfban *s*-ből elérhető negatív kör
  - ➢ az algoritmus fentebbi változata alkalmazható



A negatív körök kezelésével kiegészített struktogramok



A sor-alapú Bellman-Ford algoritmus (QBF) elemzése

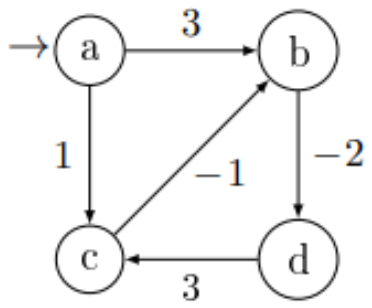
- Az alábbi elemzésben arra az esetre szorítkozunk, amikor nincs a gráfban *s*-ből elérhető negatív kör.
- Az algoritmus helyességének bizonyítása és az algoritmus hatékonyságának vizsgálata szempontjából is alapvető a QBF futásának menetekre bontása

10. Definíció. Menet rekurzív definíciója:

- 0. menet: a start csúcs (s) feldolgozása
- (i+1). menet: az i. menet végén a sorban levő csúcsok feldolgozása

Az s $\overset{opt}{\rightsquigarrow}$ v utak megkeresésének szemléltetése menetszámlálással

a → b, 3 ; c, 1.
b → d, −2.
c → b, −1.
d → c, 3.

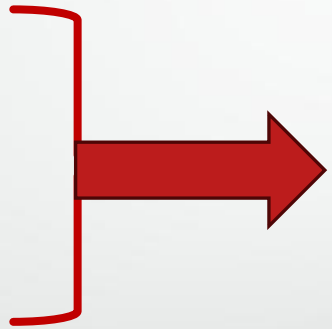


ex-pand $d\,\mathcal{V}\,e$	changes of $d\,\pi\,e$ a	b	c	d	Q : Queue	Pass
	0 ⊘ 0	∞ ⊘	∞ ⊘	∞ ⊘	⟨ a ⟩	
0 a 0		3 a 1	1 a 1		⟨ b, c ⟩	0.
3 b 1				1 b 2	⟨ c, d ⟩	1.
1 c 1		0 c 2			⟨ d, b ⟩	1.
1 d 2					⟨ b ⟩	2.
0 b 2				−2 b 3	⟨ d ⟩	2.
−2 d 3					⟨ ⟩	3.

QBF elemzése

11. Tulajdonság. A gráf tetszőleges *u* csúcsára

- ha *s*-ből nem érhető el negatív kör
- és az *u* csúcsba vezet *k* élből álló $s \overset{opt}{\rightsquigarrow} u$ út



- a *k*. menet elején már $d(u) = w(s \overset{opt}{\rightsquigarrow} u)$
- és $<s, \ldots, \pi^2(u), \pi(u), u>$ egy optimális út

- **Bizonyítás.** *k* szerinti teljes indukcióval

QBF elemzése

12. Lemma. Ha *s*-ből nem érhető el negatív kör

- tetszőleges *u* elérhető csúcsra ∃ $s \overset{opt}{\rightsquigarrow} u$ út, ami legfeljebb $n-1$ élből áll

- **Bizonyítás.**
  - Ebben az esetben az *s*-ből induló körmentes utak hossza ≤ mint a kört is tartalmazók
  - Tetszőleges körmentes útnak viszont legfeljebb $n-1$ éle van
  - Véges sok körmentes út van a gráfban
  - *s*-ből tetszőleges *v* csúcsba is véges sok körmentes út van
  - ezek között létezik egy optimális

QBF elemzése

13. Tétel. (A fenti Lemma és Tulajdonság következménye)
Ha s-ből nem érhető el negatív kör

- Tetszőleges s-ből elérhető *u* csúcsra van olyan $s \overset{opt}{\rightsquigarrow} u$ út, amit az *n*−1. menet elejére már a QBF kiszámolt
- Az *n*−1. menet végére kiürül a sor, az algoritmus pedig $O(n * m)$ időben megáll
  - $MT(n, m) \in O(n * m)$
- Az elméleti műveletigény becslés: rossz hatékonyság

QBF elemzése

- Gyakorlati tesztek: $\Theta(n)$ átlagos műveletigény, ha
  - Pozitív élsúlyú
  - Véletlenszerűen generált
  - Szomszédossági listás gráfábrázolás esetén
  - Nagyméretű
  - *s*-ből összefüggő
  - ritka gráfokra: ($m \in O(n)$)

  = aszimptotikusan az elméleti minimummal

➢ Hálózatokon való útkeresésnél alkalmazzák

- A hálózatok jelentős része nagyméretű, ritka gráffal modellezhető

Ellenőrző kérdések

1. Mit számol ki a Sor-alapú (Queue-based) Bellman-Ford algoritmus?
   - Adja meg a struktogramját!
   - Mit értünk a fenti program futásának menetei alatt? Mi a menetekhez kapcsolódó alapvető tulajdonság?
   - Adjon az algoritmus műveletigényére aszimptotikus felső becslést, és indokolja is állítását!
   - Honnét ismerhető fel, hogy van-e a gráfban a startcsúcsból elérhető negatív kör? Hogyan lokalizálható egy ilyen negatív kör?
   - Szemléltesse az algoritmus működését az alábbi gráfon, a *b* csúcsból indítva!
     - *b*-> *c*, 2 ; *e*, 4. *c*-> *d*, -1 ; *e*, 1. *d*-> *b*, -1 ; *e*, 2 ; *f*, 2. *e* -> *f*, -2.

Ellenőrző kérdések

2. Írja le röviden, szóban vagy struktogrammal, azt a tanult algoritmust, mellyel irányított, súlyozott, körmentes gráfokra a leghatékonyabb módon határozhatjuk meg a start csúcsból a többi csúcsba vezető legrövidebb utak fáját! (Negatív élköltségek is megengedettek.)
   - Mekkora a műveletigénye? Miért?
   - Szemléltesse a működését az alábbi gráfon, ahol a csúcsok topologikus rendezése <*a*, *b*, *c*, *d*, *e*, *f*>, és a *b* a startcsúcs! A legrövidebb utak fáját rajzolja is le!
     - *a*-> *d*, 2; *f*, 3. *b*-> *c*, 2; *e*, 4. *c*-> *d*, -1; *e*, 1. *d*-> *e*, 2; *f*, 2. *e* -> *f*, -2.

Köszönöm a figyelmet!

Pusztai Kinga

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

Algoritmusok és adatszerkezetek I.
9. Előadás

Legrövidebb utak minden csúcspárra. D és Pi mátrixok, Floyd-Warshall algoritmus; gráf tranzitív lezártja.

Tartalom

Utak minden csúcspárra: (Floyd-Warshall és Tranzitív lezárt algoritmusok)

- Két algoritmus hasonlósága

A gráfok csúcsmátrixos reprezentációjára épülnek

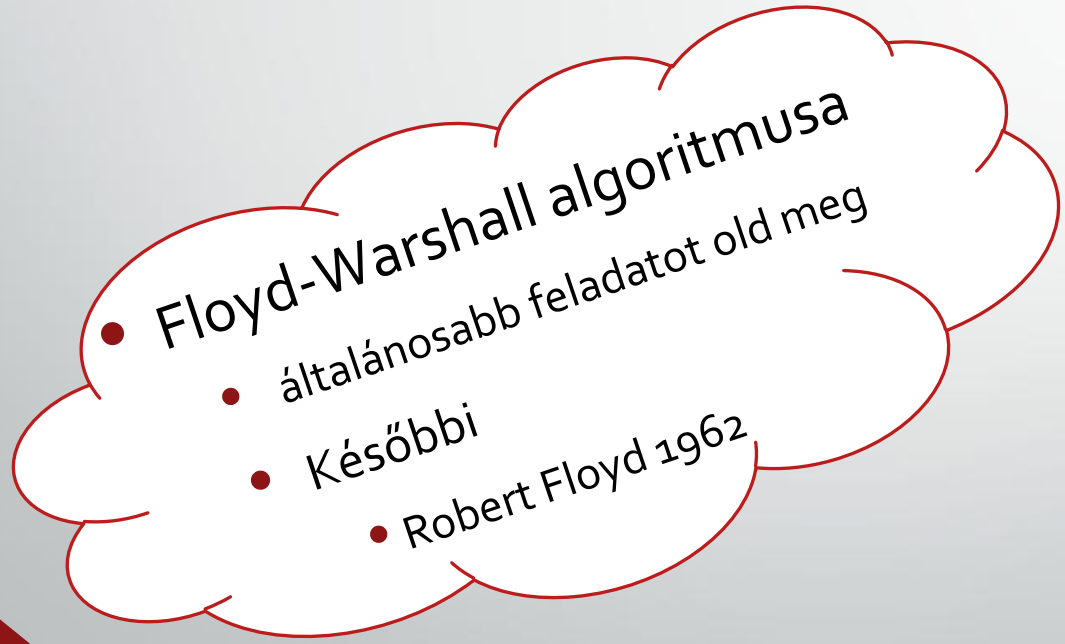
Műveletigényük: $\Theta(n^3)$

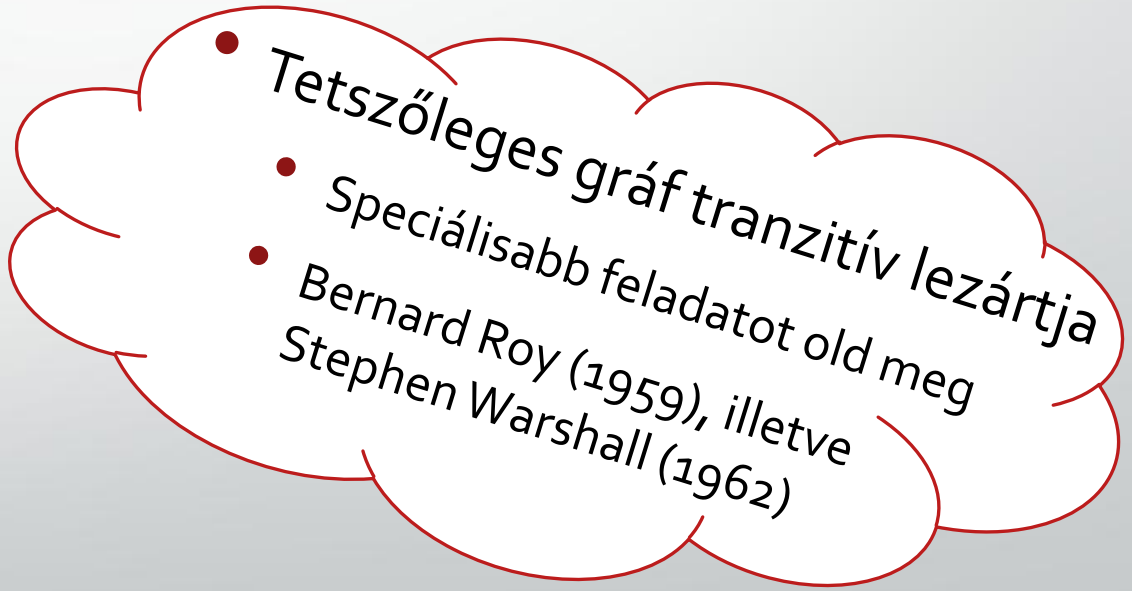
Az irányítatlan gráf értelmezése: Olyan irányított gráf, ahol minden (u, v) élnek megvan az ellentétes irányú (v, u) élpárja és $w(u, v) = w(v, u)$

Hasonló működési elv: Dinamikus programozás

Utak minden csúcspárra: (Floyd-Warshall és Tranzitív lezárt algoritmusok)

- Két algoritmus különbözősége





Legrövidebb utak minden csúcspárra: a Floyd-Warshall algoritmus (FW)

1. Jelölés. $\mathbb{R}_\infty = \mathbb{R} \cup \{\infty\}$

- Az $A/1 : \mathbb{R}_\infty [n, n]$ csúcsmátrixszal adott élsúlyozott gráf alapján meghatározza:
  - $D/1 : \mathbb{R}_\infty [n, n]$ mátrixot ahol $D[i, j]$
    - az i indexű csúcsból a j indexű csúcsba vezető optimális út hossza
    - vagy ∞, ha nincs út i-ből j-be.
  - $\pi/1 : \mathbb{N}[n, n]$ mátrixot is, ahol $\pi[i, j]$
    - az algoritmus által kiszámolt, az i indexű csúcsból a j indexű csúcsba vezető optimális úton a j csúcs közvetlen megelőzője, ha $i \neq j$ és létezik út i-ból j-be
    - Különben $\pi[i, j] = 0$.

Floyd-Warshall algoritmus (FW)

- **Előfeltétel.**
  - A gráfban nincs negatív kör. (Ezt az algoritmus ellenőrzi.)
- A továbbiakban gráf csúcsait
  - 1-től *n*-ig indexeljük
  - a csúcsokat az indexükkel azonosítjuk
- Az algoritmus a $<(D^{(0)}, \pi^{(0)}),(D^{(1)}, \pi^{(1)}), \ldots ,(D^{(n)}, \pi^{(n)})>$ mátrixpársorozatot állítja elő, amit a következőképpen definiálunk

Floyd-Warshall algoritmus (FW)

2. Jelölés. $i \overset{k}{\rightsquigarrow} j$ az i csúcsból a j csúcsba vezető út, azzal a megszorítással, hogy

- az úton a közbenső csúcsok indexe ≤ k (i > k és j > k megengedett, továbbá i, j ∈ 1..n és k ∈ 0..n).

3. Jelölés. $i \overset{k}{\underset{opt}{\rightsquigarrow}} j$ az i csúcsból a j csúcsba vezető legrövidebb körmentes út, azzal a megszorítással, hogy

- az úton a közbenső csúcsok indexe ≤ k.
- Ha több ilyen út lenne, azt vesszük, ahol a közbenső csúcsok indexeinek maximuma a lehető legkisebb.

Floyd-Warshall algoritmus (FW)

4. Megjegyzés. k szerinti indukcióval adódik, hogy

- Ha létezik $i \overset{k}{\rightsquigarrow} j$ út $\Rightarrow$ az $i \overset{k}{\underset{opt}{\rightsquigarrow}} j$ út egyértelműen definiált (mivel nincs negatív kör)
  - $i = j$ esetén $i \overset{k}{\underset{opt}{\rightsquigarrow}} j = \langle i \rangle$
  - $i \neq j$ esetén pedig
    - $\exists\, i \overset{0}{\underset{opt}{\rightsquigarrow}} j = \langle i, j \rangle \Leftrightarrow (i, j) \in G.E$
    - $k \in 1..n$ esetén pedig egy $i \overset{k}{\underset{opt}{\rightsquigarrow}} j$ úttal kapcsolatban két lehetőség van:
      - $i \overset{k}{\underset{opt}{\rightsquigarrow}} j = i \overset{k-1}{\underset{opt}{\rightsquigarrow}} j$ $\quad \vee \quad$ $i \overset{k}{\underset{opt}{\rightsquigarrow}} j = i \overset{k-1}{\underset{opt}{\rightsquigarrow}} k \overset{k-1}{\underset{opt}{\rightsquigarrow}} j$

Floyd-Warshall algoritmus (FW)

5. Definíció. $D_{ij}^{(k)} = \begin{cases} w\begin{pmatrix} k \\ i \rightsquigarrow j \\ opt \end{pmatrix} \text{ ha létezik } i \overset{k}{\rightsquigarrow} j \\ \infty \text{ ha nem létezik } i \overset{k}{\rightsquigarrow} j \end{cases}$

6. Definíció. $\pi_{ij}^{(k)} = \begin{cases} \text{az } i \overset{k}{\underset{opt}{\rightsquigarrow}} j \text{ úton a } j \text{ csúcs közvetlen megelőzője, ha i} \neq \text{ j és létezik } i \overset{k}{\rightsquigarrow} j, \\ 0 \qquad \text{ha } i = j \text{ vagy nem létezik } i \overset{k}{\rightsquigarrow} j \end{cases}$

Floyd-Warshall algoritmus (FW)

7. Tulajdonság.

$D^{(0)}$ mátrix: megegyezik a gráf A csúcsmátrixával

$D^{(n)}$ mátrix: a kiszámítandó D mátrix

8. Tulajdonság.

$$\pi_{ij}^{(0)} = \begin{cases} i & \text{ha } i \neq j \wedge (i,j) \text{ a gráf éle} \\ 0 & \text{ha } i = j \vee (i,j) \text{ nem éle a gráfnak} \end{cases}$$

$\pi^{(n)}$ mátrix: megegyezik a kiszámítandó π mátrixszal

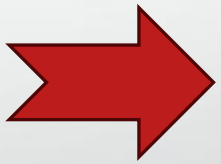
Floyd-Warshall algoritmus (FW)

9. Tulajdonság. A 4. megjegyzés alapján, $k \in 1..n$ esetén:

- Ha $D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{ij}^{(k-1)}$
  - akkor $D_{ij}^{(k)} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \wedge \pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$
  - különben $D_{ij}^{(k)} = D_{ij}^{(k-1)} \wedge \pi_{ij}^{(k)} = \pi_{ij}^{(k-1)}$

10. Következmény. $k \in 1..n$ esetén:

- $D_{ik}^{(k)} = D_{ik}^{(k-1)} \wedge \pi_{ik}^{(k)} = \pi_{ik}^{(k-1)} \wedge$
  $D_{kj}^{(k)} = D_{kj}^{(k-1)} \wedge \pi_{kj}^{(k)} = \pi_{kj}^{(k-1)}$
- Jelentése:
  $D^{(k)}$ mátrix k-adik sora és oszlopa = $D^{(k-1)}$
  $\pi^{(k)}$ mátrix k-adik sora és oszlopa = $\pi^{(k-1)}$



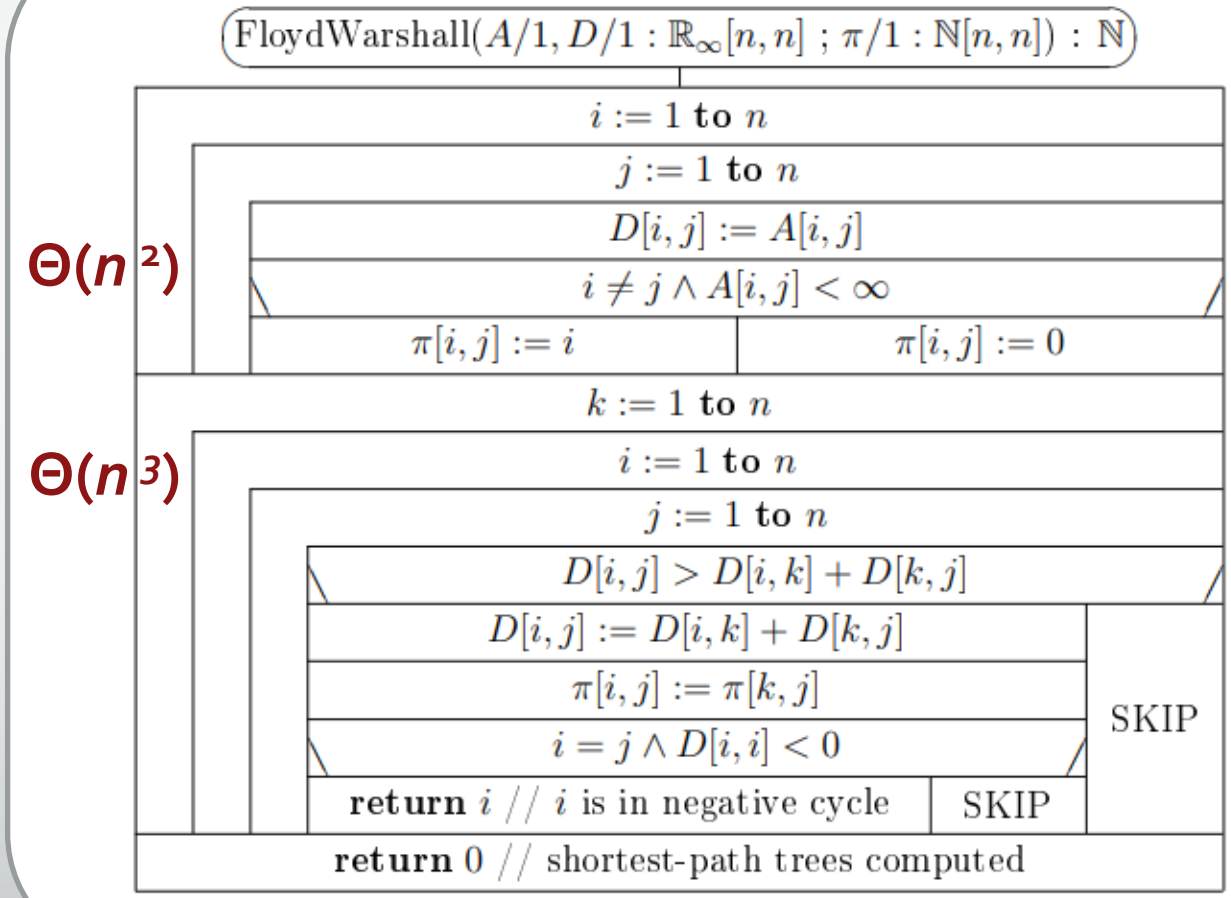
a D mátrix kiszámításához nem kell segédmátrixokat tárolni, mert

- $D_{ij}^{(k)}$ csak $D_{ij}^{(k-1)}$, $D_{ik}^{(k-1)}$ és $D_{kj}^{(k-1)}$ értékektől függ

➢ $D_{ik}^{(k)} = D_{ik}^{(k-1)}$ és $D_{kj}^{(k)} = D_{kj}^{(k-1)}$,

Floyd-Warshall algoritmus (FW)

- 1. kettős ciklus
  - A fizikailag is létező *D* mátrixot az elméleti $D^{(0)}$ mátrix elemeivel inicializáljuk
- 2. háromszorosan beágyazott ciklus
  - k = 1-től n-ig kiszámoljuk $D^{(k-1)}$-ből $D^{(k)}$ –t
  - fizikailag végig ugyanazt a *D* és π mátrixot használva
- Ha a *D* mátrix főátlójában negatív érték jelenik meg
  - ➢ negatív kört találtunk
- Műveletidő
  - $MT(n) \in \Theta(n^3)$
  - $mT(n) \in \Theta(n^2)$
    - Ha van a gráfban negatív kör
      - 2. külső ciklus 1.iterációja alatt kiderülhet



A legrövidebb utak minden csúcspárra probléma visszavezetése a legrövidebb utak egy forrásból feladatra

- A 2 probléma közötti kapcsolat
  - Floyd-Warshall (FW) algoritmus által kiszámolt D és π mátrixok i-edik sora a legrövidebb utak egy forrásból feladat megoldása $s = i$ esetére
  - Megfordítva, ha minden $s \in 1..n$ értékre megoldjuk a legrövidebb utak egy forrásból feladatot => megkapjuk a legrövidebb utak minden csúcspárra probléma megoldását is

Néhány speciálisabb eset

- Ha a gráf DAG
  - $MT(n, m) \in \Theta(n * (n + m))$
  - Ritka gráfokon: $\Theta(n^2)$
    - ➢ nagysárenddel gyorsabb, mint az FW algoritmus
  - Sűrű gráfokon: $\Theta(n^3)$
    - feltéve, hogy a csúcsok többségéből a gráf nagy része elérhető
    - ➢ a gyakorlatban a nagyobb rejtett konstansok miatt lassúbb futást eredményez
- Ha a gráf élsúlyozatlan
  - Szélességi keresést alkalmazunk
  - Hasonló eredmények

Néhány speciálisabb eset

- Ha a gráfunk élsúlyai nemnegatívak
  - a Dijkstra algoritmust minden $s \in 1..n$ értékre végrehajtva
  - $MT(n, m) \in O(n * (n + m) * \log n)$
  - Ritka gráfokra: $O(n^2 * \log n)$
    - Aszimptotikusan lényegesen jobb, mint az FW algoritmus $\Theta(n^3)$ műveletigénye
  - Sűrű gráfokra: $MT(n, m) \in O(n^3 * \log n)$
    - Aszimptotikusan rosszabb felső becslés, mint az FW esetén

Néhány speciálisabb eset

- Ha csak annyit tudunk, hogy nincs a gráfunkban negatív kör
  - QBF algoritmust minden $s \in 1..n$ értékre végrehajtva
  - $MT(n, m) \in O(n * n * m)$
    - a legrosszabb esetet tekintve,
      és feltételezve, hogy nincs lényegesen kevesebb él, mint csúcs a gráfban

    ➢ sosem ad jobb felső becslést, mint amit az FW algoritmus esetén kaptunk

Gráf tranzitív lezártja (TC)

Feladat. egy hálózatban (gráfban) honnét hova lehet eljutni.
Az eljutás módja és költsége most nem érdekel bennünket.

11. Definíció. A $G = (V, E)$ gráf *tranzitív lezártja* alatt a $T \subseteq V \times V$ relációt értjük, ahol
$(u, v) \in T \Leftrightarrow$ ha G-ben az u csúcsból elérhető a v csúcs.

12. Jelölés. $\mathbb{B} = \{0; 1\}$

- A gráf ábrázolása az $A/1 : \mathbb{B}[n, n]$ csúcsmátrixszal
  - Ha a gráfunk csúcsai az 1..n indexekkel azonosíthatók
  - (az esetleges élsúlyoktól eltekintve)

Gráf tranzitív lezártja (TC)

- A tranzitív lezártja ábrázolása: a $T/1 : \mathbb{B}[n, n]$ mátrixszal
  - Ahol $T[i, j] \Leftrightarrow$ ha az A mátrixszal ábrázolt gráfban van út az i csúcsból a j csúcsba
  - A T mátrix kiszámításához definiáljuk a $<T^{(0)}, T^{(1)}, T^{(n)}>$ mátrixsorozatot, aminek utolsó tagja magával a T mátrixszal lesz egyenlő

13. Definíció. $T_{ij}^{(k)} \Leftrightarrow \exists\, i \overset{k}{\rightsquigarrow} j \; (k \in 0..n \wedge i, j \in 1..n)$

14. Tulajdonság. (A T mátrixok rekurzív megadása.)

- $T_{ij}^{(0)} = A[i, j] \vee (i = j) \qquad (i, j \in 1..n)$
- $T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee T_{ik}^{(k-1)} \wedge T_{kj}^{(k-1)} \; (k \in 1..n \wedge i, j \in 1..n)$

Gráf tranzitív lezártja (TC)

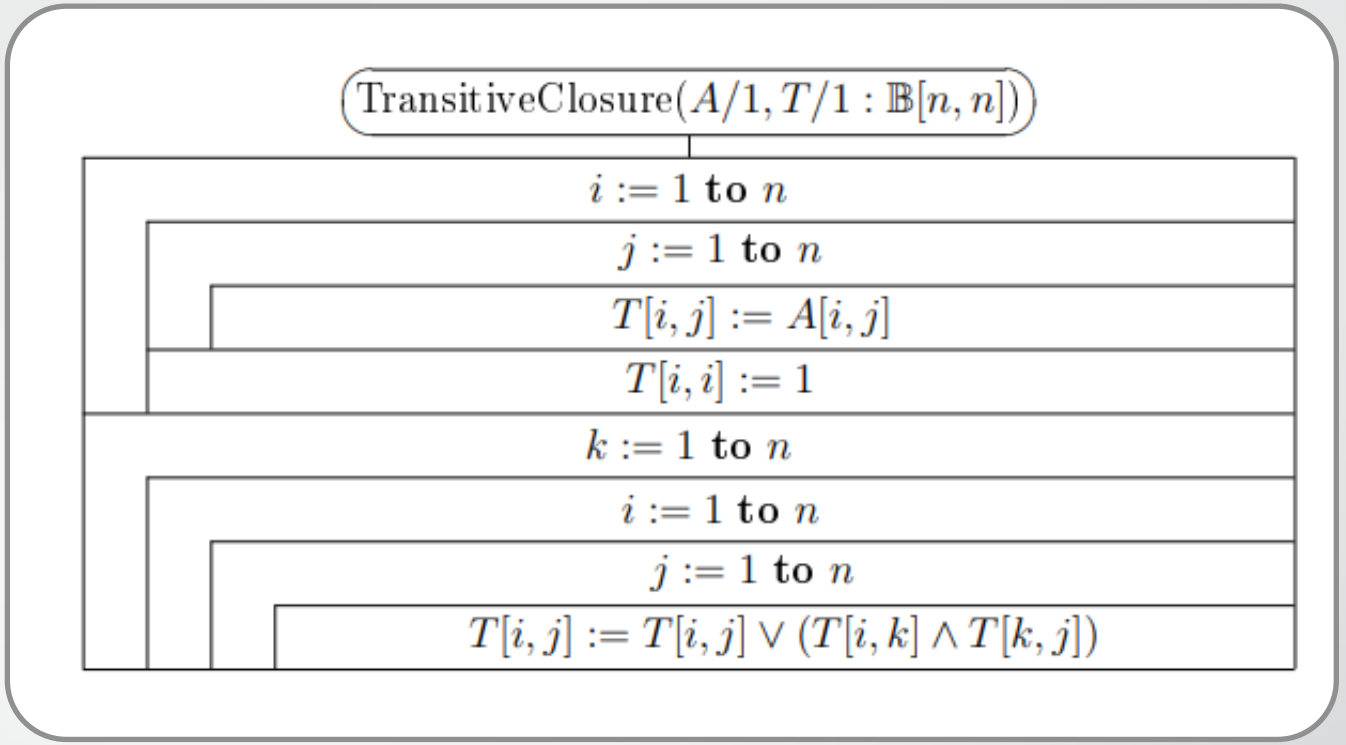
15. Következmény.

- $T_{ik}^{(k)} = T_{ik}^{(k-1)} \wedge T_{kj}^{(k)} = T_{kj}^{(k-1)}$ ($k \in 1..n \wedge i, j \in 1..n$)
  - $T^{(k)}$ mátrix k-adik oszlopa és sora = $T^{(k-1)}$ mátrixé. ($k \in 1..n$)

➢ a T mátrix kiszámításához nem kell segédmátrixokat tárolni, mert

- $T_{ij}^{(k)}$ csak a $T_{ij}^{(k-1)}$, a $T_{ik}^{(k-1)}$ és a $T_{kj}^{(k-1)}$ értékektől függ
- $T_{ik}^{(k)} = T_{ik}^{(k-1)}$, $T_{kj}^{(k)} = T_{kj}^{(k-1)}$

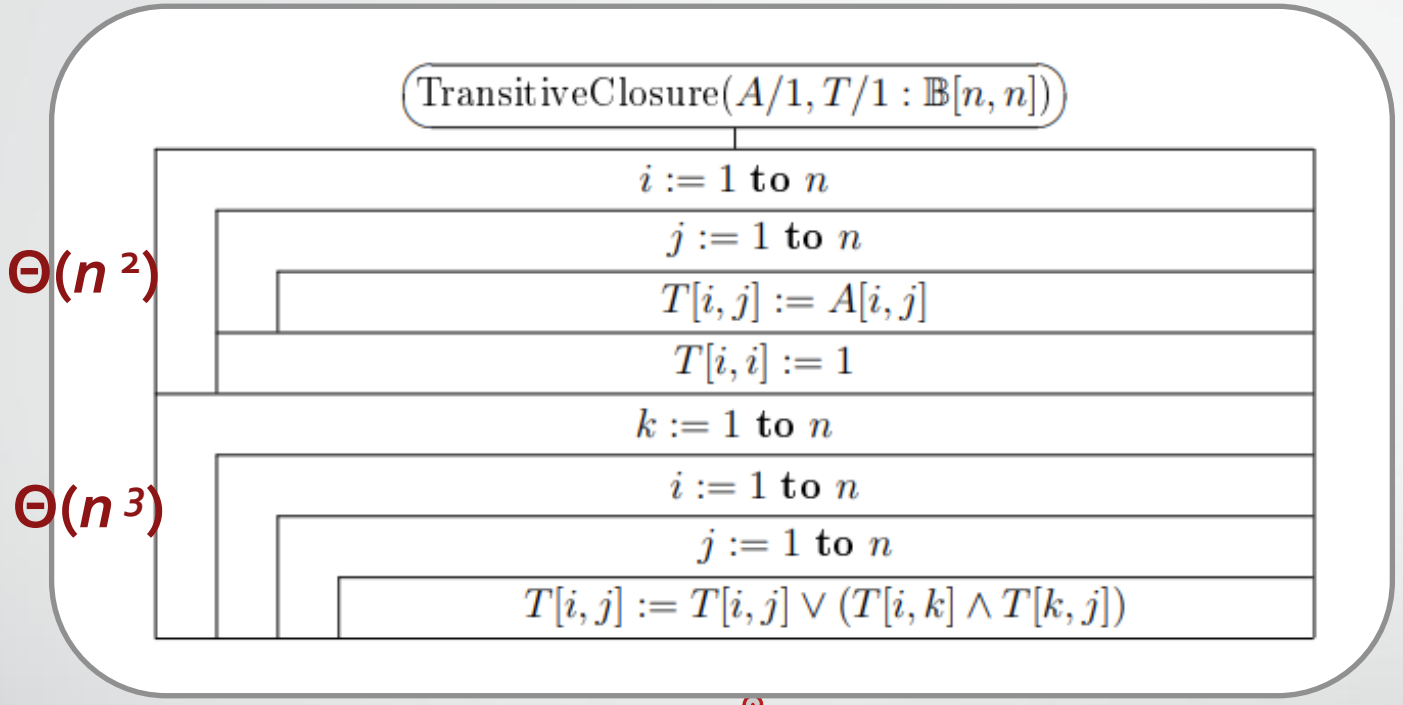
A tranzitív lezárt algoritmusa



- 1. kettős ciklusa
  - A fizikailag is létező *T* mátrixot az elméleti $T^{(0)}$ mátrix elemeivel inicializáljuk
- 2. háromszorosan beágyazott ciklusa
  - *k* = 1-től *n*-ig kiszámoljuk $T^{(k-1)}$-ből $T^{(k)}$ –t
  - Fizikailag végig ugyanazt a *T* mátrixot használva
    - *T*[*i*, *j*] új értékének kiszámolásakor az elméleti mátrixsorozat $T_{ij}^{(k)}$ elemét határozzuk meg
    - Az értékadás végrehajtása előtt még
      - $T[i, j] = T_{ij}^{(k-1)}$,
      - $T[i, k] = T_{ik}^{(k)} = T_{ik}^{(k-1)}$ és $T[k, j] = T_{kj}^{(k)} = T_{kj}^{(k-1)}$

➢ mindegy, hogy a második külső ciklus adott iterációján belül a *T*[*i*, *k*], illetve a *T*[*k*, *j*] már értéket kapott-e

A tranzitív lezárt algoritmus elemzése



- Műveletidő: $T(n) \in \Theta(n^3)$
- A TC és az FW kapcsolata: $T[i, j] \Leftrightarrow D[i, j] < \infty$ $(i, j \in 1..n)$
  - TC: egyszerűbb feladat, hatékonyabb megoldás (az egyszerűbb ciklusmagok miatt)

A tranzitív lezárt kiszámításának visszavezetése szélességi keresésre

- A szélességi keresés után
  - $s = i$ esetben éppen azok a csúcsok lesznek feketék, amelyek elérhetők i-ből
- A szélességi keresés egyszerűsített változatát alkalmazva
  - ahol $T[i, j] \Leftrightarrow \text{color}(j) \neq \text{white} \Rightarrow$ a T mátrix megfelelő sorát kapjuk meg
  - $MT(n, m) \in \Theta(n + m)$ műveletigénnyel
- A mátrix összes sorát így $\Theta(n * (n + m))$ maximális futási idővel kapjuk meg
  - Ritka gráfok esetén $\Theta(n^2)$
    - azaz aszimptotikusan jobb, mint a TC algoritmus
  - Sűrű gráfok esetén $\Theta(n^3)$
    - ami a gyakorlatban hosszabb futási időt eredményez, a nagyon egyszerű TC algoritmussal összehasonlítva

Ellenőrző kérdések

1. Mit számol ki a Floyd-Warshall algoritmus? Mekkora a műveletigénye n csúcsú gráf esetén? Miért?
   - Szemléltessük a működését az alábbi irányított gráfon a $(D^{(0)}; \pi^{(0)}); \ldots ; (D^{(3)}; \pi^{(3)})$ mátrix párok megadásával!
   - Rajzoljuk le a legrövidebb utak fáit, amiket az eredményből kiolvashatunk!
     - 1-> 3, 1. 2-> 1, 0 ; 3, 2. 3-> 1, 1 ; 2, 2.
2. Mit jelent egy gráf tranzitív lezártja, amit Warshall algoritmusa számol ki?
   - Tegyük fel, hogy a gráfnak *n* csúcsa van! Mi a $<T^{(k)}: k \in 0..n>$ mátrix sorozat definíciója? Mi a rekurzív képlete?
   - Adja meg az algoritmus struktogramját! Mekkora a műveletigénye n csúcsú gráf esetén? Miért elegendő egyetlen T mátrix a programban?
   - Mutassa be az algoritmus működését a 4 -> 3. 3->2. 2 -> 1 ; 4. irányított gráfon!

Köszönöm a figyelmet!

Pusztai Kinga

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

Algoritmusok és adatszerkezetek I.
10. Előadás

Mintaillesztés

Tartalom

Mintaillesztés (String-matching)

1. Jelölések. (Notations)

- $\mathbb{N} = \{ i \in \mathbb{Z} \mid i \geq 0 \}$ $\quad i \,..\, k = \{ j \in \mathbb{N} \mid i \leq j \leq k \}$
- $[i \,..\, k) = \{ j \in \mathbb{N} \mid i \leq j < k \}$ $\quad (i \,..\, k) = \{ j \in \mathbb{N} \mid i < j < k \}$
- $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_d\}$ az ábécé (alphabet), ahol d $\in \mathbb{N} \wedge d > 0$
- $T : \Sigma[n]$ a szöveg (text), ami betűk egy sztringje 0-tól $n-1$-ig indexelve.
- $T[i \,..\, j) = T[i \,..\, j-1]$
- $P : \Sigma[m]$ a minta (pattern), ahol $0 < m \leq n$.

- A továbbiakban feltesszük, hogy
  - $T : \Sigma[n]$ és $P : \Sigma[m]$,
  - a hosszuk, azaz m és n változatlanok, ahol $1 \leq m \leq n$ (a minta nem üres, és legfeljebb olyan hosszú, mint a szöveg)
- Feladat:
  - A $T : \Sigma[n]$ szövegben keressük a $P : \Sigma[m]$ minta előfordulásait (occurrences)
  - Más szavakkal a T szövegben P minta azon eltolásait keressük, amelyekre $T[s \,..\, s+m) = P[0 \,..\, m)$
  - Ezekre az eltolásokra $s \in 0 \,..\, n-m$ nyilvánvalóan teljesül.

Mintaillesztés (String-matching)

2. Definíció. $s \in 0 \,..\, n-m$ a *P minta* lehetséges eltolása (possible shift) a *T szövegen*

- $s \in 0 \,..\, n-m$ **érvényes eltolás** (valid shift), ha
  - $T[s \,..\, s+m) = P[0 \,..\, m)$
- Különben $s \in 0 \,..\, n-m$ **érvénytelen eltolás** (invalid shift)

3. Probléma. (Mintaillesztés.)

- Az érvényes eltolások V halmazát szeretnénk meghatározni
- Ehhez eltolások egy S halmazába gyűjtjük a mintaillesztő keresések futása során talált érvényes eltolásokat
- A problémát megoldó algoritmusok közös utófeltétele $S = V$, ahol
  - $V = \{ s \in 0 \,..\, n-m \mid T[s \,..\, s+m) = P[0 \,..\, m) \}$

Egyszerű mintaillesztés (Naive string-matching)

- **Példa:** A $P[0..4)$ = *BABA* mintát keressük a $T[0..11)$ = *ABABB ABABAB* szövegben
  - $\underline{B}$: *B*-t sikeresen illesztette a szöveg megfelelő betűjére
  - $\not{B}$: sikertelenül illesztette.)

- Brute-force (nyers erő) algoritmus:
  - Sorban megvizsgáljuk a lehetséges eltolásokat, és kiszűrjük közülük az érvényeseket

```
BruteForce(T : Σ[n] ; P : Σ[m] ; S : ℕ{})
S := {}
s := 0 to n − m
    match(T, s, P) // T[s..s+m) = P[0..m)
    S := S ∪ {s}  |  SKIP
```

$i=$	0	1	2	3	4	5	6	7	8	9	10
$T[i]=$	*A*	*B*	*A*	*B*	*B*	*A*	*B*	*A*	*B*	*A*	*B*
	$\not{B}$	*A*	*B*	*A*							
		$\underline{B}$	$\underline{A}$	$\underline{B}$	$\not{A}$						
			$\not{B}$	*A*	*B*	*A*					
				$\underline{B}$	$\not{A}$	*B*	*A*				
s=4					$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$			
						$\not{B}$	*A*	*B*	*A*		
s=6							$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	
								$\not{B}$	*A*	*B*	*A*

$S = \{4; 6\} = V$

```
match( T : Σ[] ; s : ℕ ; P : Σ[m] ) : 𝔹
// check, whether T[s..s+m) = P[0..m)
j := 0
j < m ∧ T[s + j] = P[j]
    j++
return j ≥ m
```

Algoritmus műveletigénye

- A $T[s\,..\,s+m) = P[0\,..\,m)$ egyenlőségvizsgálatra:
  - $MT_e(m) \in \Theta(m)$
  - $mT_e(m) \in \Theta(1)$

➢ Innét a teljes algoritmusra:

  - $MT_{BF}(n, m) \in \Theta((n - m + 1) * m)$
  - $mT_{BF}(n, m) \in \Theta(n - m + 1)$

```
match( T : Σ[] ; s : ℕ ; P : Σ[m] ) : 𝔹
// check, whether T[s..s+m) = P[0..m)
j := 0
j < m ∧ T[s + j] = P[j]
    j++
return j ≥ m
```

- Ha egy feladatosztályon valamely $0 < k < 1$ konstansra $m \leq k * n$

  ➢ $1 * n \geq n - m + 1 > n - k * n = (1 - k) * n$,
  ahol $1 > 1 - k > 0$ konstans

➢ A $\Theta(\cdot)$ függvényosztályok definíciója szerint:

  - $n - m + 1 \in \Theta(n)$
  - $(n - m + 1) * m \in \Theta(n * m)$

```
BruteForce(T : Σ[n] ; P : Σ[m] ; S : ℕ{})
S := {}
s := 0 to n − m
    match(T, s, P) // T[s..s+m) = P[0..m)
        S := S ∪ {s}
    SKIP
```

Műveletigény*

- az $\cdot \in \Theta(\cdot)$ reláció tranzitivitása miatt:

4. Következmény. Ha egy feladatosztályon valamely k konstansra $0 < k < 1 \wedge m \leq k * n$

- $mT_{BF}(n,m) \in \Theta(n)$
- $MT_{BF}(n,m) \in \Theta(n * m)$

- Ha egy feladatosztályon m az n-hez képest nem is elhanyagolható
  - azaz valamely $\varepsilon > 0$ konstansra $m \geq \varepsilon * n \Rightarrow \varepsilon * n * n \leq n * m \leq n * n$
- Következésképp $n * m \in \Theta(n^2)$
  - Ebből + 4 következménnyel adódik az alábbi eredmény:

5. Következmény.
Ha egy feladatosztályon az ε és a k konstansokra $0 < \varepsilon \leq k < 1 \wedge \varepsilon * n \leq m \leq k * n \Rightarrow$ $MT_{BF}(n,m) \in \Theta(n^2)$

Quicksearch és KMP algoritmusok

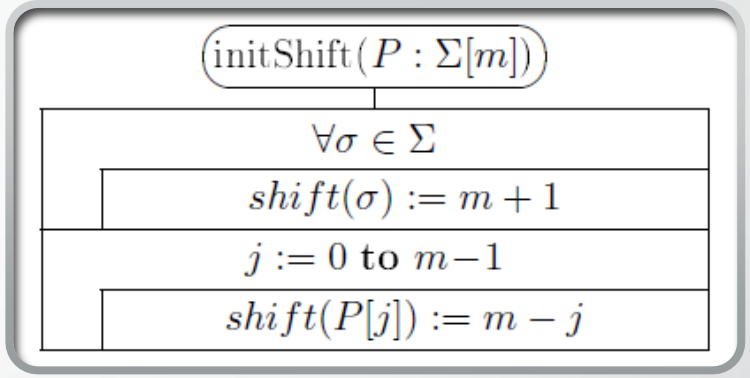
- A gyorsabb keresés érdekében:
  - általában egynél nagyobb lépésekben növeljük a P[o . .*m*) minta eltolását a T[o . . *n*) szöveghez képest úgy
    - biztosan ne ugorjunk át egyetlen érvényes eltolást sem
- A tényleges mintaillesztés előtt egy előkészítő fázist hajt végre
  - nem függ a szövegtől, csak a mintától

Quicksearch előkészítő fázis

- Quicksearch előkészítő fázisában
  - az ábécé σ elemeihez shift(σ) $\in 1 \ldots m+1$ címkéket társítunk, ahol P[0 . .m) a keresett minta
  - Tfh σ = T[$s + m$]
  - shift(σ) érték: megmondja, hogy a T[s . . s + m) = P[0 . .m) öh után legalább mennyivel kell (jobbra) eltolni a P mintát a szövegen ahhoz
    - a T[$s + m$] alapján legyen esély a mintának a megfelelő szövegrészhez való illeszkedésére
  - $\sigma \in$ P[0 . .m) esetén a shift(σ) $\in 1 \ldots m$ érték azt mondja meg, hogy legalább mennyivel kell tovább tolni a P mintát ahhoz, hogy a T[$s + m$] betűhöz kerülő karaktere maga is σ legyen. Világos, hogy a σ legjobboldali P-beli előfordulásához tartozik a legkisebb ilyen eltolás.
  - $\sigma \notin P$[0 . .m) esetén shift(σ) = $m + 1$ lesz, azaz a minta átugorja a T[$s + m$] karaktert.

Quicksearch: initShift

- shift(σ) jelentése
  - $\sigma \in P[0 \,..\, m)$ esetén: shift(σ) ∈ 1 . . m
    - legalább mennyivel kell tovább tolni a P mintát ahhoz, hogy a $T[s + m]$ betűhöz kerülő karaktere maga is σ legyen
    - a σ legjobboldali P-beli előfordulásához tartozik a legkisebb ilyen eltolás
  - $\sigma \notin P[0 \,..\, m)$ esetén: shift(σ) = m + 1
    - azaz a minta átugorja a $T[s + m]$ karaktert
- Az ábécé méretét konstansnak tekintve: $T_{\text{initShift}}(m) \in \Theta(m)$
- A $P[0 \,..\, 4)$=*CADA* mintával az initShift() működése:



σ	A	B	C	D
initial $shift(\sigma)$	5	5	5	5
C			4	
A	3			
D				2
A	1			
final $shift(\sigma)$	1	5	4	2

Quicksearch algoritmus

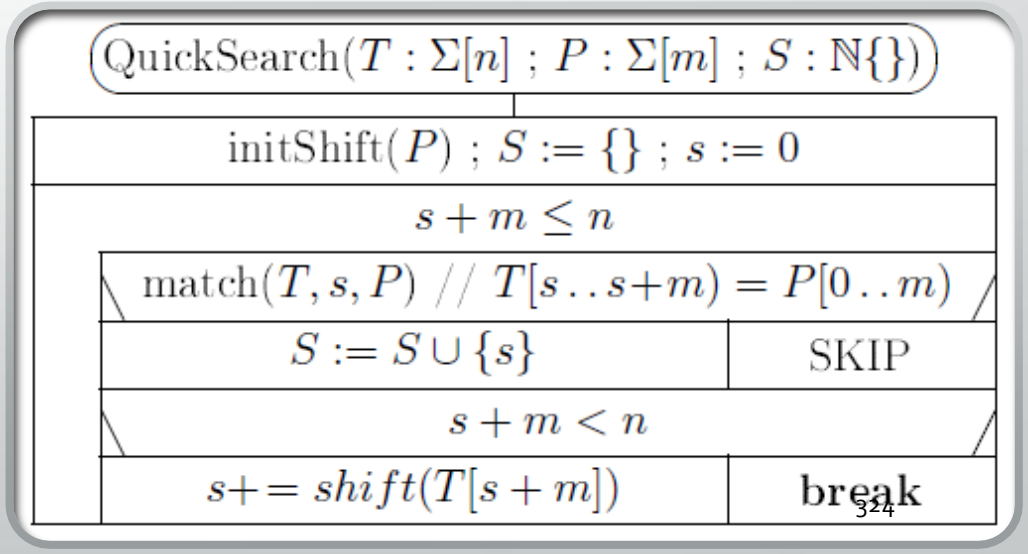
σ	A	B	C	D
$shift(\sigma)$	1	5	4	2

$i=$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$T[i]=$	A	D	A	B	A	B	C	A	D	A	B	C	A	B	A	D	A	C	A	D	A	D	A
	∅	A	D	A																			
		∅	A	D	A																		
$s=6$							C	A	D	A													
												C	A	∅	A								
														∅	A	D	A						
$s=17$																		C	A	D	A		
																				∅	A	D	A

$S = \{6; 17\} = V$

- $mT(n,m) \in \Theta\left(\frac{n}{m+1} + m\right)$
  - (pl. ha $T[0\,..\,n)$ és $P[0\,..\,m)$ diszjunktak)
- $MT(n,m) \in \Theta((n - m + 2) * m)$
  - (pl. ha $T = AA\ldots A$ és $P = A\ldots A$)

- Összehasonlítás az egyszerű mintaillesztéssel
  - $mT(n)$ nagyságrenddel jobb
  - $MT(n)$ egy kicsit rosszabb
  - Tapasztaltok szerint az átlagos futási idő inkább a legjobb esethez áll közel
- Időkritikus alkalmazásokban
  - Egy stabilabb futási idejű, minden esetben hatékony algoritmusra lenne szükségünk



Mintaillesztés lineáris időben (Knuth-Morris-Pratt, azaz KMP algoritmus)

- A KMP algoritmus futási ideje : $\Theta(n)$ (minden esetben)
  - a legjobb és a legrosszabb eset között körülbelül kétszeres szorzóval
  - ➢ hatékony működés és nagyon stabil futási idő
- Sohasem lép vissza a *T* szövegen
  - Ellentétben az előző megoldásokkal
  - ➢ a KMP algoritmus könnyen implementálható szekvenciális fájlokra.

Speciális jelölések:

6. Jelölések . *Ha x és y két sztring =>*

- *x + y a konkatenáltjuk.*
  - *Pl. x = AB és y = BA esetén x + y = ABBA és y + x = BAAB.*
- *ε az üres sztring.*
  - *Nyilván tetszőleges x sztringre x + ε = x = ε + x.*
- *$x \sqsubseteq y$ (x az y prefixe) jelentése: $\exists z$ sztring : x + z = y.*
  - *(ε, A,AB,ABB,ABBA $\sqsubseteq$ ABBA)*
- *$x \sqsubset y$ (x az y valódi prefixe [proper prefix]) jelentése: $x \sqsubseteq y \wedge x \neq y$.*
  - *(ε, A,AB,ABB $\sqsubset$ ABBA)*

Speciális jelölések:

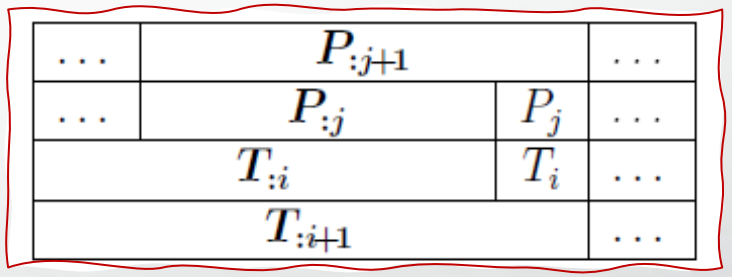
6. Jelölések. *Ha x és y két sztring ⇒*

- *$x \sqsupseteq y$ (x az y szuffixe) jelentése: $\exists z$ sztring : $z + x = y$.*
  - *(ABBA,BBA,BA, A, $\varepsilon \sqsupseteq$ ABBA)*
- *$x \sqsupset y$ (x az y valódi szuffixe [proper suffix]) jelentése: $x \sqsupseteq y \wedge x \neq y$.*
  - *(BBA,BA, A, $\varepsilon \sqsupset$ ABBA)*
- *$P_{:j} = P[0 \,..\, j) = P[0 \,..\, j-1]$ (csak ebben az alfejezetben)*
  - *$P_{:j}$ a P sztring j hosszúságú prefixe, azaz kezdőszelete.*
  - *$P_{:0} = \varepsilon$ az üres prefixe.*
  - ≈ *$T_{:i} = T[0 \,..\, i)$.*
  - ➢ *minden sztringnek szuffixe az üres sztring, azaz $P_{:0} \sqsupseteq P_{:j}$ $(j \in 0 \,..\, m)$,*
  - ➢ *minden nemüres sztringnek valódi szuffixe az üres sztring, azaz $P_{:0} \sqsupset P_{:j}$ $(j \in 1 \,..\, m)$.*

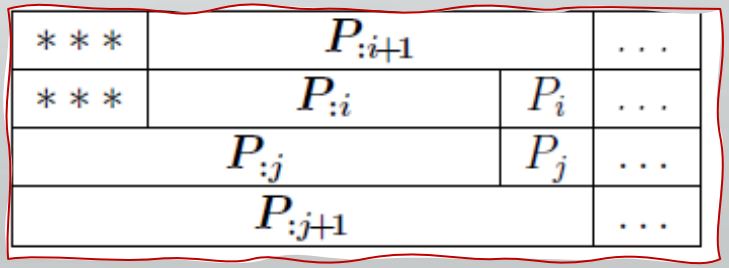
Sztringek tulajdonságai

7. Lemma. Szuffix kiterjesztési (Suffix-extension) lemma

- $P_j \sqsupseteq T_i \wedge P[j] = T[i] \Leftrightarrow P_{:j+1} \sqsupseteq T_{i+1}$



- $P_{:i} \sqsupset P_{:j} \wedge P[i] = P[j] \Leftrightarrow P_{:i+1} \sqsupset P_{:j+1}$



Bevezetés a KMP algoritmushoz

- Példa:
  - Feltesszük, hogy van egy hosszabb *T* szövegünk, de most ennek csak a *T*[*i*−5 . . *i*+2] = *BABA BABB* részét vesszük figyelembe.
  - A minta a $P_{:6}$ = *BABABB*
  - Az aktuális eltolás *i* − 5
  - A sikeresen illesztett betűket aláhúztuk
  - A sikertelenül illesztett karaktert pedig áthúztuk

…	T_{i-5}	T_{i-4}	T_{i-3}	T_{i-2}	T_{i-1}	T_i	T_{i+1}	T_{i+2}
…	B	A	B	A	B	A	B	B
$P_{:6} =$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	~~B~~		
			B	A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$

Példa megoldása

$\ldots$	T_{i-5}	T_{i-4}	T_{i-3}	T_{i-2}	T_{i-1}	T_i	T_{i+1}	T_{i+2}
$\ldots$	B	A	B	A	B	A	B	B
$P_{:6} =$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\not{B}$		
			B	A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$

- $P_{:5} = T[i-5 \ldots i)$, de $P[5] \neq T[i]$
  - $i-5$ egy érvénytelen eltolás (3. sor)
  - P minta (i-5)-nél nagyobb eltolását keressük
- Elvárások az eltolással:
  - $T_{:i}$ alapján esélyes legyen, hogy érvényes
  - Ne ugorjon át esélyes eltolást.
  - T szöveg korábban megvizsgált, első i betűjét ($T_{:i}$ kezdőszeletét vesszük figyelembe)
- Ígéretes eltolások:
  - (i -4), (i -3), (i -2), (i -1), i

Példa megoldása

- $P_{:5} = T[i-5\,..\,i)$, de $P[5] \neq T[i]$
- Ígéretes eltolások:
  - (i -4) és (i-2) eltolás
    - (i-4): $T_{i-4} = A$ betűvel a $P_0 = B$ szeretnénk illeszteni
    - (i-2): $T_{i-2} \neq P_0$
    - ➢ Sikertelen => érvénytelen eltolások
  - (i -3) , (i-1) és i eltolás
    - (i-3): $T[i-3..i) = BAB = P_{:3}$
    - (i-1): $T[i-1..i) = B = P_{:1}$
    - i: $T[i..i) = P_{:0}$
    - ➢ Sikeres => esélyes, hogy érvényes eltolások
    - Ha a három közül a nem a legkisebb eltolást választanánk => átugornánk egy érvényes eltolást

$\ldots$	T_{i-5}	T_{i-4}	T_{i-3}	T_{i-2}	T_{i-1}	T_i	T_{i+1}	T_{i+2}
$\ldots$	B	A	B	A	B	A	B	B
$P_{:6} =$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\not{B}$		
			B	A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$
					B	A	B	$\ldots$

Példa megoldása

...	T_{i-5}	T_{i-4}	T_{i-3}	T_{i-2}	T_{i-1}	T_i	T_{i+1}	T_{i+2}
...	B	A	B	A	B	A	B	B
$P_{:6} =$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\not{B}$		
			B	A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$

- Tehát *P* mintának a legkisebb, az aktuálishoz képest további eltolását keressük
  - hogy a minta $P_{:5}$ kezdőszeletének az a $P_{:k}$ ($0 \le k < 5$) prefixe, ami még a szöveg $T[i-k \,..\, i)$ része alatt van, illeszkedjen arra
  - (azaz $P_{:k} = T[i-k \,..\, i)$, vagy $P_{:0} \sqsupset T_{:i}$) (4. sor)
- Ez a $P_{:k}$ -t meghatározó legkisebb további eltolás legfeljebb 5
  - mert $P_{:0} \sqsupset T_{:i.}$
  - Ezzel a $P_{:k}$-t meghatározó eltolással biztosan nem ugrunk át egyetlen érvényes eltolást sem.
- A példában két betűvel kell tovább tolni a mintát és $k = 3$
  - $P[3] = T[i]$, $P[4] = [i+1]$ és $P[5] = T[i+2]$
  - ➢ $i-3$ egy érvényes eltolás
  - Bármely ennél nagyobb eltolás átugorta volna az $i-3$ érvényes eltolást.

Bevezetés a KMP algoritmushoz

- Hogyan lehetne a k értékét hatékonyan meghatározni általános esetben?
  - ≈ az előző gondolatmenethez
  - Általánosítás:
    - Minta: $P_{:m}$
    - Szöveg: $T_{:n}$
    - $j \in 1 \,..\, m$, ahol
      - az aktuális eltolás : $i–j$
      - $P_{:j} = T[i–j \,..\, i)$ azaz $P_{:j} \sqsupseteq T_{:i}$
      - $(P[j] \neq T[i] \vee j = m)$
  - A *további eltolás* + $k = j$
    - Nagyobb *további eltolás*hoz kisebb k érték, míg kisebb további eltoláshoz nagyobb k érték tartozik

Bevezetés a KMP algoritmushoz

- k a legkisebb *további eltolás*hoz tartozik, amelyre $P_{:k} \sqsupset T_{:i}$
  - ahol ez a *további eltolás* $\leq j$ (ui. $P_{:0} \sqsupset T_{:i}$)
  - ➢ k a legnagyobb h, amire $P_{:h} \sqsupset T_{:i} \wedge 0 \leq h < j$ teljesül
    - Ezt a leghosszabb $P_{:h}$ -t keressük
- Továbbá $P_{:h} \sqsupset T_{:i}$ ekvivalens a $P_{:h} \sqsupset P_{:j}$ relációval

$\ldots$	T_{i-j}	$\ldots$	T_{i-h}	$\ldots$	T_{i-1}	T_i	$\ldots$
$P_{:j} =$	P_0	$\ldots$	P_{j-h}	$\ldots$	P_{j-1}	$\ldots$	$\ldots$
			P_0	$\ldots$	P_{h-1}	$\ldots$	$\ldots$

∑ Azaz: a leghosszabb $P_{:h}$ -t keressük, amire $P_{:h} \sqsupset P_{:j}$

- Ennek hosszát a π prefix függvény határozza meg.

π prefix függvény

9. Definíció. $\pi(j) = \max\{ h \in 0\,..\,j-1 \mid P_{:h} \sqsupset P_{:j} \}$ $(j \in 1\,..\,m)$

- Azaz a leghosszabb $P_{:h}$ -t keressük, amire $P_{:h} \sqsubset P_{:j} \wedge P_{:h} \sqsupset P_{:j}$
- ($P_{:h}$ a $P_{:j}$ prefix-szuffixe)

10. Tulajdonság. $j \in 1\,..\,m \Rightarrow \pi(j) \in 0\,..\,j-1$

- **Bizonyítás.**
  - 9. definíció szerint: $\pi(j)$ a $0\,..\,j-1$ egy részhalmazának a maximuma.

π prefix függvény

11. Lemma. $j \in [1\,..m) \Rightarrow \pi(j+1) \leq \pi(j) + 1$

- **Bizonyítás.**
  - Ha $\pi(j+1) = 0 \Rightarrow \pi(j+1) = 0 \leq 0+1 \leq \pi(j)+1$
    - mivel $\pi(j) \geq 0$ (12. tul.)
  - Ha $\pi(j+1) > 0$
    - (9. def. szerint) $P_{:(\pi(j+1)-1)+1} = P_{:\pi(j+1)} \sqsupset P_{:j+1}$
    - (7. lemma) $P_{:\pi(j+1)-1} \sqsupset P_{:j}$
    - (9. def) $\pi(j+1) - 1 \leq \pi(j)$
    - $\pi(j+1) \leq \pi(j) + 1$.

Példa: π függvény kiszámítása

- $P_{:8} = P[0\,..\,8) = BABA\ BBAB$ mintára
  - Alkalmazzuk:

    9. Definíció. $\pi(j) = \max\{ h \in 0\,..\,j-1 \mid P_{:h} \sqsupset P_{:j} \}$ $(j \in 1\,..\,m)$

    10. Tulajdonság. $j \in 1\,..\,m \Rightarrow \pi(j) \in 0\,..\,j-1$

    11. Lemma. $j \in [1\,..\,m) \Rightarrow \pi(j+1) \leq \pi(j) + 1$

  - Megoldás:

$P[j-1] =$	B	A	B	A	B	B	A	B
$j =$	1	2	3	4	5	6	7	8
$\pi(j) =$	0	0	1	2	3	1	2	3

Példa: π függvény kiszámítása

$P[j-1] =$	B	A	B	A	B	B	A	B
$j =$	1	2	3	4	5	6	7	8
$\pi(j) =$	0	0	1	2	3	1	2	3

- Megoldás:
  - Minden esetben: $\pi(1) = 0$ (10. sz.: $\pi(1) \in 0\,..\,1-1$) — *B*
  - $\pi(2) = 0$ (11. sz: *π fv.*max 1-gyel nő, $B \not\sqsupset BA$) — *BA*
  - $\pi(3) = 1$ (11. sz: $\pi(3) \leq 1$ $P_{:1} = B \sqsupset BAB = P_{:3}$ => *9. sz.* $\pi(3) \geq 1$) — *BAB*
  - $\pi(4) = 2$ (11. sz: $\pi(4) \leq 2$ $P_{:2} = BA \sqsupset BABA = P_{:4}$ => *9. sz.* $\pi(4) \geq 2$) — *BABA*
  - $\pi(5) = 3$ (~) — *BABAB*
  - $\pi(6) = 1$ ($\pi(6) \leq 4$; $P_{:1} = B \sqsupset BABABB = P_{:6}$) — *BABABB*
    - $P_{:4}$ = BABA$\not\sqsupset$ BABABB = $P_{:6}$, $P_{:3}$ = BAB$\not\sqsupset$BABABB = $P_{:6}$, $P_{:2}$ = BA$\not\sqsupset$ BABABB = $P_{:6}$
  - $\pi(7) = 2$ (~) — *BABABBA*
  - $\pi(8) = 3$ (~) — *BABABBAB*

Példa:

- Minta: $P_{:8} = P[0 \ldots 8) = BABA\ BBAB$
- Szöveg: $T_{:18} = T[0 \ldots 18) = ABABAB\ ABBABA\ BABBAB$

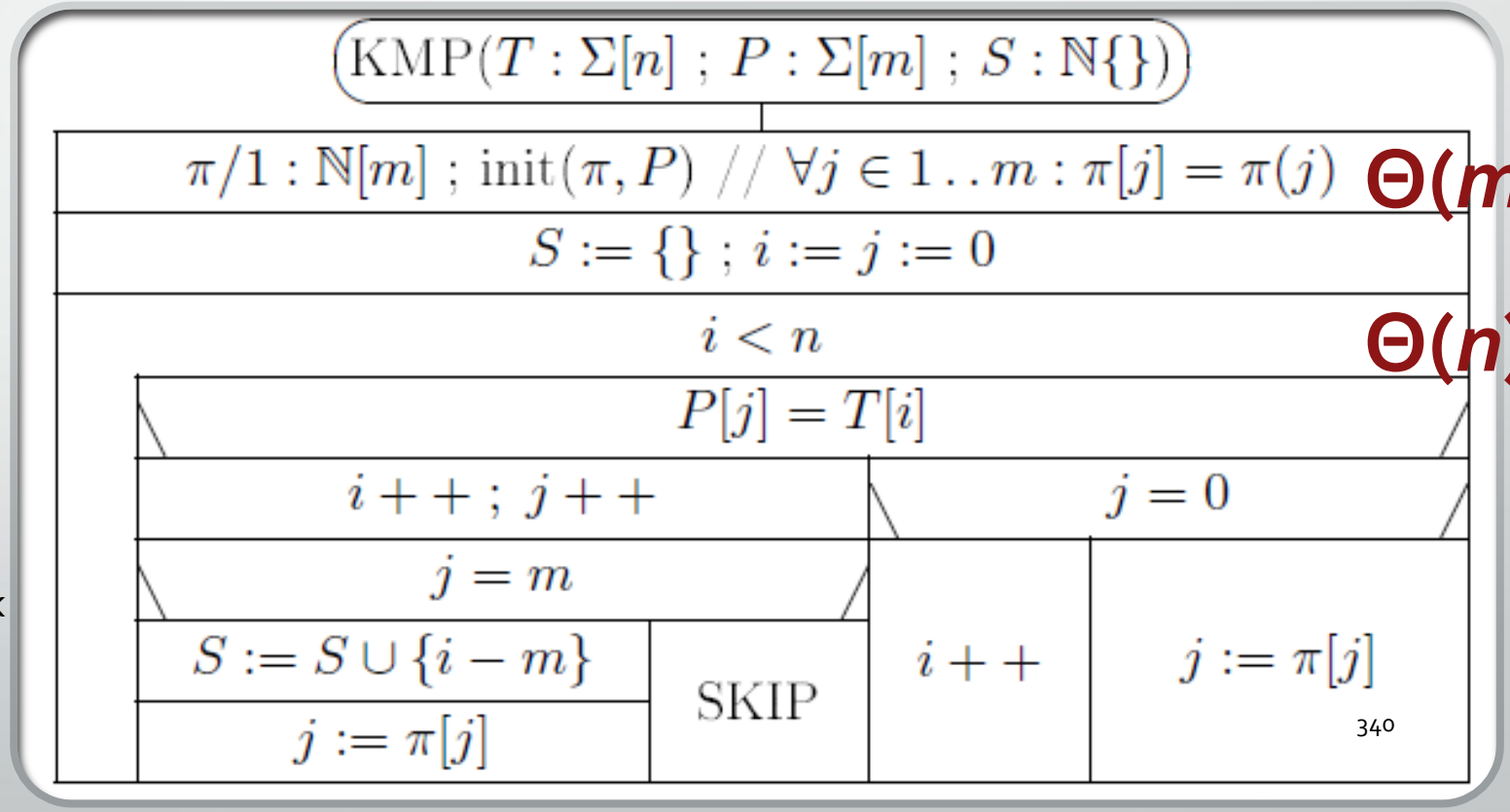
$i=$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$T[i]=$	A	B	A	B	A	B	A	B	B	A	B	A	B	A	B	B	A	B
	~~B~~																	
		B	A	B	A	B	~~B~~											
$s=3$				B	A	B	A	B	B	A	B							
									B	A	B	A	B	~~B~~				
$s=10$											B	A	B	A	B	B	A	B
																B	A	B

$$S = \{3; 10\} = V$$

A KMP algoritmus fő eljárása

- Ciklus globális invariánsa:
  - i–j a minta aktuális eltolása $\wedge$
  - $P_{:j} \sqsupseteq T_{:i}$ $\wedge$
  - $0 \le j \le i \le n$ $\wedge$
  - $j \le m$

- init(π, P)
  - π prefix-függvény értékeit összegyűjti a π tömbbe
  - Műveletigény: $\Theta(m)$
  - Utófeltétele:
    - $\forall j \in 1\,..\,m : \pi[j] = \pi(j)$
- Fő eljárás:
  - ciklusa az invariánsán alapszik
  - ahol i–j a P minta aktuális eltolása a T szövegen



A KMP(T, P, S) eljárás megállása és hatékonysága

- A terminálás igazolása
  - init(π, P) eljárás műveletigénye: $\Theta(m)$ (bizonyítás később)
  - KMP(T, P, S) eljárás műveletigénye: $\Theta(n)$
- KMP(T, P, S) eljárás ciklusa legfeljebb $2n$ iterációt hajt végre
  - Elég belátni, hogy a fő ciklusának a futási ideje $\Theta(n)$
    - $m \in 1\,..\,n$ => az init(π, P) eljárás és egyéb inicializálások $\Theta(m)$ műveletigénye aszimptotikus nagyságrendben már nem módosít
  - A fő ciklus legalább n és legfeljebb $2n$ iterációt hajt végre
    - A ciklus (Inv) invariánsa szerint: $i \in 0\,..\,n \wedge j \in 0\,..\,(m-1) \wedge j \leq i$

A KMP(*T*, *P*, *S*) eljárás megállása és hatékonysága

- legalább *n* iteráció
  - az első iteráció előtt $i = 0$
  - mindegyik iteráció legfeljebb eggyel növeli az *i* változót
  - a ciklusfeltétel $i < n$

legalább *n* iteráció szükséges ahhoz, hogy $i = n$ legyen, és az eljárás befejeződjék

- legfeljebb $2n$ iteráció
  - Tekintsük a $2i-j$ kifejezést!
    - Az $i \in 0 \,..\, n \wedge j \in 0 \,..\, (m-1) \wedge j \leq i$ inv. tul. => $2i-j \in 0 \,..\, 2n$
      - $2i-j = i + (i-j) \in 0+0 \,..\, n+n = 0 \,..\, 2n$
    - Az első iteráció előtt $2i-j = 0$
    - mindegyik iterációval (mind a négy programágon) szig. mon. nő
    - végig $2i-j \leq 2n$

legfeljebb $2n$ iteráció után megáll a ciklus

A π prefix tömb inicializálása

- A π prefix tömböt az init(π, P) eljárás tölti fel úgy, hogy:

$$\forall k \in [1..m]: \pi[k] = \pi(k)$$

- A π függvény definíciója:

$$\pi(j) = \max\{\, h \in [0..j-1] \mid P_{:h} \sqsupset P_{:j} \,\}$$

- Ennek következménye:
  - **$\pi(1) = 0$**, minden mintára
  - ezért inicializáláskor:
    - $\pi[1] := 0$

A π prefix tömb inicializálása

- **Ciklus invariáns**
  - A ciklust a következő invariáns szerint szervezzük:

$$1 \leq j \leq m \ \wedge \ \left(\forall k \in [1..j]: \pi[k] = \pi(k)\right)$$

  - Kezdés:
    - $j := 1$
    - ekkor az invariáns már igaz (mert $\pi[1] = 0$)
  - Ciklusfeltétel:
    - $j < m$ (addig igaz, amíg a π tömböt még nem töltöttük fel.)

A π prefix tömb inicializálása

- **Következő érték számítása**
  - Feladat: $\pi(j+1)$ meghatározása (= $\pi[j+1]$)

$$\pi(j+1) = \max\{\, h \in [0..j] \mid P_{:h} \sqsupset P_{:j+1} \}$$

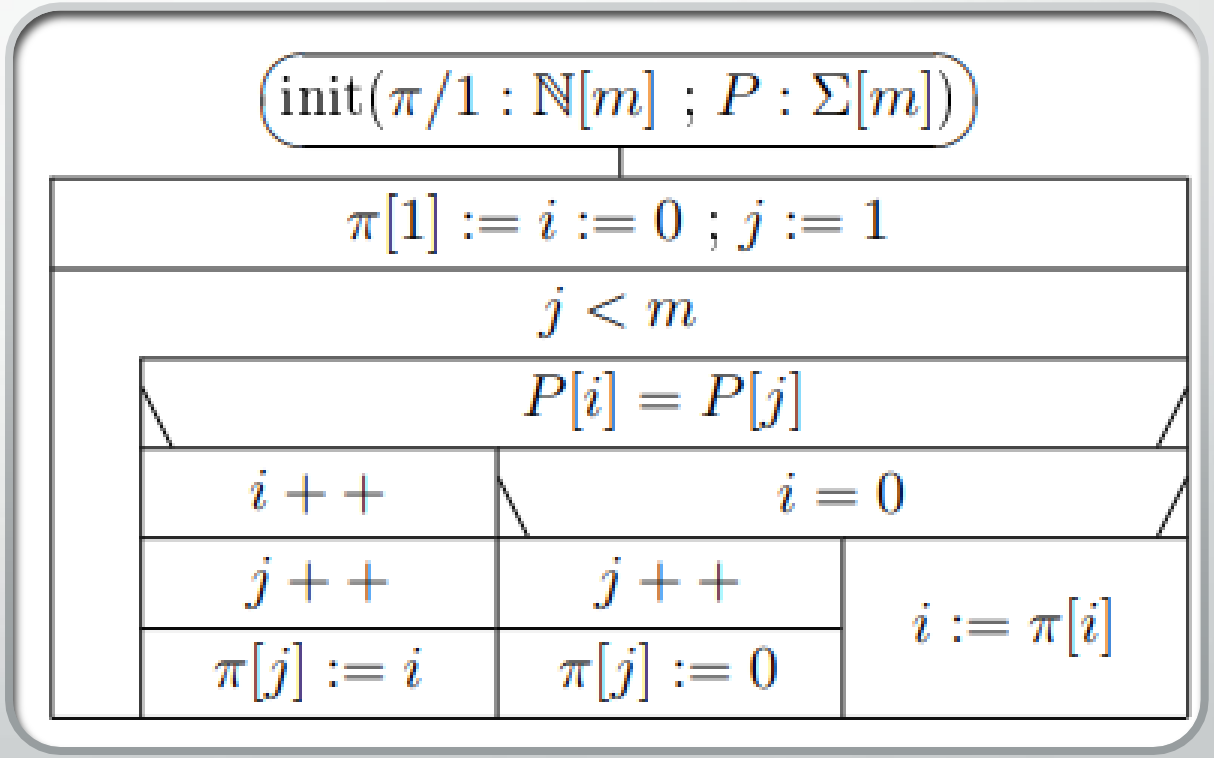
  - Két eset:
    - pozitív érték
      - Ha $\pi(j+1) > 0$, => $\pi(j+1) = \max\{\, i+1 \in [1..j] \mid P_{:i+1} \sqsupset P_{:j+1} \}$
      - A szuffixkiterjesztési lemma alapján: $P_{:i+1} \sqsupset P_{:j+1} \iff P_{:i} \sqsupset P_{:j} \wedge P[i] = P[j]$
    - **0**, ha nincs megfelelő prefix-szuffix

A π prefix tömb inicializálása

- **Gyakorlati következmény**
  - A keresést a $P_{:j}$ leghosszabb $P_{:i}$ prefix-szuffixéből indítjuk
  - Ha nem jó:
    - visszalépünk egy rövidebb prefix-szuffixre
  - Ezt addig folytatjuk, amíg:
    - találunk megfelelőt $\longrightarrow \pi(j+1) > 0$
    - vagy nincs $\longrightarrow \pi(j+1) = 0$

Az init(π, P) eljárás

- A ciklus invariánsa
  - $P_{:i} \sqsupset P_{:j} \wedge$
  - $[\ \forall k \in 1\,..\,j : \pi[k] = \pi(k)\ ] \wedge$
  - $0 \le i < j \le m \wedge$
  - $[\forall h \in (i\,..\,j) : P_{:h} \sqsupset P_{:j} \Rightarrow P_{:h+1} \not\sqsupset P_{:j+1}]$



A π tömb számításának esetei

1. Egyezés esete: $P[i] = P[j]$

- Ha az aktuális prefix-szuffixnél teljesül, hogy: $P[i] = P[j]$
  - A szuffixkiterjesztési lemma alapján: $\pi(j+1) = i+1$
- Bal oldali programág
- Következmény:
  - növeljük *i*-t és *j*-t
  - teljesül: $i = \pi(j)$

👉 A következő lépésben a keresést a **megnövelt prefix-szuffixből** folytatjuk

A π tömb számításának esetei

- **2. Nem egyezés, de van még prefix:** $P[i] \neq P[j],\ i > 0$
  - Ha nincs egyezés, de az aktuális prefix-szuffix **nem üres**:
    - $i := \pi[i]$
  - Jobb oldali programág
  - Jelentése:
    - visszalépünk a **következő leghosszabb prefix-szuffixre**

👉 Így nem kezdjük újra a keresést elölről,
hanem **a megfelő helyre visszaugrunk**

A π tömb számításának esetei

- **3. Nem egyezés és nincs prefix:** $i = 0$
  - Ha már az összes lehetőséget kipróbáltuk és nincs egyezés:

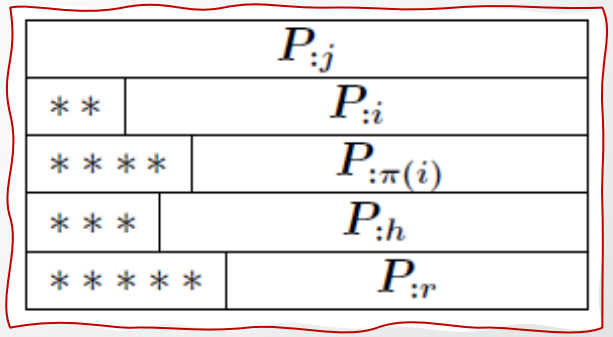
$$\pi(j + 1) = 0$$

  - Középső programág
  - Következmény:
    - nincs nemüres prefix-szuffix
    - a keresés az **üres sztringből indul újra**

A π tömb számításának esetei

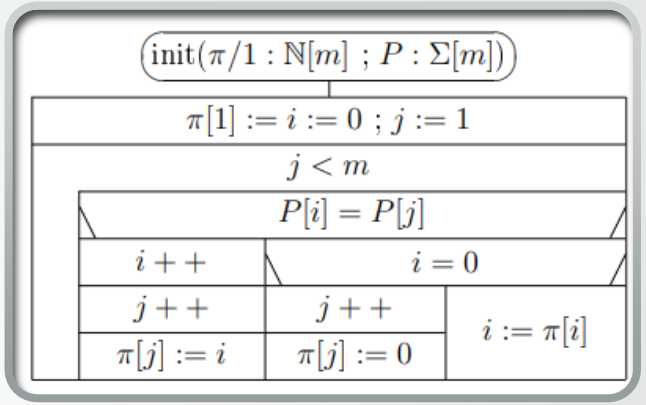
14. Lemma. Ha $P_{:i}$ a $P_{:j}$ –nek egy nemüres prefix-szuffixe (tehát o < *i* < *j*), akkor $P_{:\pi(i)}$ a $P_{:j}$ –nek a $P_{:i}$ után következő leghosszabb prefix-szuffixe

- **Bizonyítás.**
  - (π fv. def.) $P_{:\pi(i)}$ a $P_{:i}$ leghosszabb prefix-szuffixe
  - A $P_{:i}$ a $P_{:j}$ prefix-szuffixe
  - ➢ $P_{:\pi(i)}$ szintén szuffixe lesz $P_{:j}$-nek
  - $\pi(i) < i < j$
    - ➢ $P_{:\pi(i)}$ a $P_{:j}$ -nek a $P_{:i}$ -nél rövidebb prefix-szuffixe
  - $P_{:j}$ -nek a $P_{:i}$ után következő leghosszabb prefix-szuffixe a $P_{:i}$ -nek is prefix-szuffixe
    - ➢ Nem lehet $P_{:\pi(i)}$ -nél hosszabb (az ábrán: $P_{:h}$) és rövidebb sem ((az ábrán: $P_{:r}$)



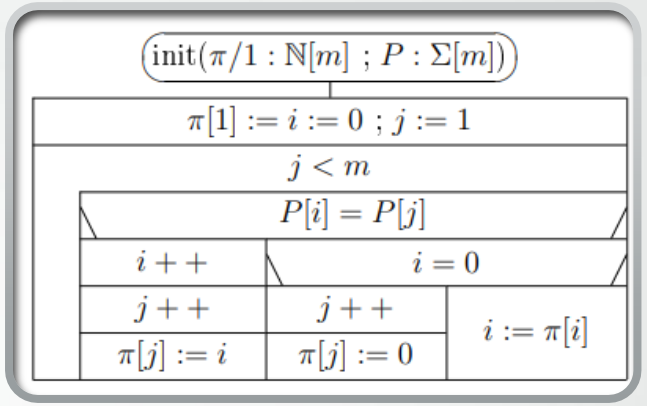
- Az értékadás ($i := \pi[i]$) helyes, mert:
  - (14. Lemma sz.) $i := \pi[i]$ a következő leghosszabb prefix-szuffixének hosszát adja
    - Feltéve, hogy $\pi[i]$ már rendelkezésünkre áll. Ez pedig kövekezik:
      - Init() eljárás invariáns tulajdonságából
      - És a jobb oldali programág i ≠ o feltételéből

Az init(π, P) eljárás megállása és hatékonysága: $\Theta(m)$



- Az eljárás ciklusa legfeljebb $2m-2$ iterációt hajt végre:
  - A ciklus minden egyes iterációja $\Theta(1)$ műveletigényű
  - Az init(π, P) eljárás ciklusa legalább $m-1$ iterációt hajt végre
    - A ciklus (inv) invariánsa szerint $0 \le i < j \le m$
    - Az első iteráció előtt $j = 1$
    - \+ Mindegyik iteráció legfeljebb eggyel növeli a j változót
    - \+ a ciklusfeltétel $j < m$
    - ➢ legalább $m-1$ iteráció szükséges ahhoz, hogy $j = m$ legyen, és az eljárás befejeződjék
  - Az init(π, P) eljárás ciklusa legfeljebb $2m-2$ iterációt hajt végre

Az init(π, P) eljárás megállása és hatékonysága:



- Az init(π, P) eljárás ciklusa legfeljebb $2m-2$ iterációt hajt végre
  - Tekintsük a $2j-i$ kifejezést!
    - $0 \le i < j \le m \Rightarrow 2j-i \in 2 \ldots 2m$
      - $j-i \ge 1 \wedge j \ge 1 \Rightarrow 2j-i \ge 2$
      - $j \le m \Rightarrow 2j \le 2m \Rightarrow 2j-i \le 2m$, mivel $i \ge 0$
    - Az első iteráció előtt $2j-i = 2$

    +mindegyik iterációval (3 programágon) szigorúan monoton nő

    +végig $2j-i \le 2m$

    ➢legfeljebb $2m-2$ iteráció után megáll a ciklus

A KMP algoritmus összegzése

- KMP algoritmusnál legjobb és a legrosszabb eset absztrakt műveletigénye között kétszeres szorzó van
  - a futási idő is nagyon stabil
  - a legrosszabb esetben is meglepően hatékony, a szöveg hosszának közelítőleg lineáris függvénye
- Online alkalmazásoknál
  - a hatékonyság (a legrosszabb esetben is) és a futási idő stabilitása együtt, határozott előny a másik két algoritmussal szemben
- Offline alkalmazásoknál
  - a Quicksearch várható futási ideje jobb, mint a KMP algoritmusé => ez lehet előnyösebb

A KMP algoritmus összegzése

- A KMP(*T*, *P*, *S*) eljárás i változója sohasem csökken => a szövegen sohasem kell visszalépni
  - Mivel a *T*[o . . *n*) szövegre csak a *T*[*i*] kifejezésen keresztül hivatkozunk
  - ➢ a KMP algoritmus kényelmesen, hatékonyan implementálható akkor is, ha a szöveg egy szekvenciális fájlban van
  - a Brute-force és a Quicksearch algoritmusoknál van visszalépés
    - a szövegen esetleg *m*−2 karakternyit is vissza kell lépni
    - ➢ szekvenciális input fájl:
      - az utoljára beolvasott *m*−1 karakterét folyamatosan nyilván kell tartani egy átmeneti tárolóban

A KMP algoritmus szemléltetése

- Az init(π, P) algoritmus szemléltetése az *ABAB BABA* mintán:
- (A három programág mindegyikének az elején kezdünk új sort.)

i	j	$\pi[j]$	^{0}A	^{1}B	^{2}A	^{3}B	^{4}B	^{5}A	^{6}B	^{7}A
0	1	0		~~A~~						
0	2	0			$\underline{A}$					
1	3	1			A	$\underline{B}$				
2	4	2			A	B	~~A~~			
0	4	2					~~A~~			
0	5	0						$\underline{A}$		
1	6	1						A	$\underline{B}$	
2	7	2						A	B	$\underline{A}$
3	8	3								

A végeredmény:

$P[j-1] =$	A	B	A	B	B	A	B	A
$j =$	1	2	3	4	5	6	7	8
$\pi[j] =$	0	0	1	2	0	1	2	3

A KMP algoritmus szemléltetése

- A $P[0\,..\,8)$ = *ABAB BABA* mintát keressük
- A $T[0\,..\,17)$ = *ABABAB BABABB ABABA* szövegben
- A mintához tartozó $\pi/1 : \mathbb{N}[8]$ tömböt fentebb már kiszámoltuk.

A végeredmény:

$P[j-1] =$	A	B	A	B	B	A	B	A
$j =$	1	2	3	4	5	6	7	8
$\pi[j] =$	0	0	1	2	0	1	2	3

A keresés:

$i = 0$		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$T[i]=$	A	B	A	B	A	B	B	A	B	A	B	B	A	B	A	B	A
	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\not{B}$												
$s=2$			A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$							
$s=7$								A	B	A	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$		
													A	B	A	$\underline{B}$	$\not{B}$
															A	B	$\underline{A}$

$$S = \{2; 7\} = V$$

Függelék*
Invariáns helyességéhez szükséges lemma*

15.Lemma. A szuffixum reláció tranzitivitása (Transitivity of the suffix relation)

- $x \sqsupset y \wedge y \sqsupseteq z \Rightarrow x \sqsupset z$.

16.Lemma. Átfedő szuffix (Overlapping-suffix) lemma

- Tegyük fel, hogy x, y és z olyan sztringek, amelyekre $x \sqsupseteq z$ és $y \sqsupseteq z$
- $|x| \leq |y| \Rightarrow x \sqsupseteq y$
- $|x| < |y| \Rightarrow x \sqsupset y$
- $|x| = |y| \Rightarrow x = y$

Invariáns helyességéhez szükséges lemma*

17. Lemma. $j \in 1\,..\,m \wedge P_{:j} \sqsupseteq T_{:i} \Rightarrow$ nincs érvényes eltolás az $(i–j\,..\,i–\pi(j))$ intervallumban

- **Bizonyítás.**
  - Indirekt: $k \in (\pi(j)\,..\,j)$ és $i–k$ érvényes eltolás
  - ➢ $T[i–k\,..\,i–k+m = P[0\,..\,m)$
  - Nyilván $k < j \leq m$
  - ➢ $k < m$
  - ➢ $i–k+m > i$

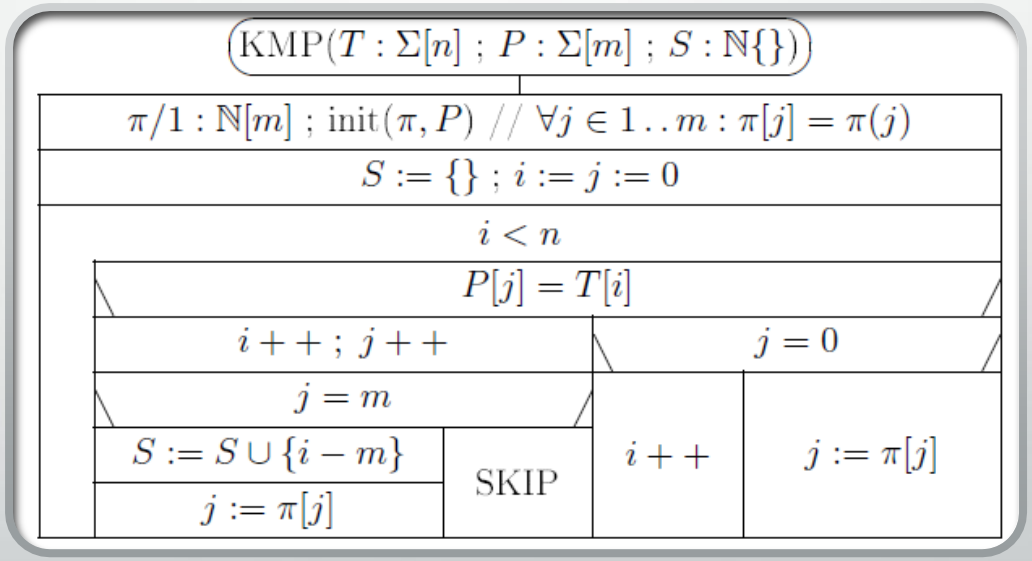
Invariáns helyességéhez szükséges lemma*

- $T[i-k \,..\, i) = P[o \,..\, k)$, vagy másképp írva $P_{:k} \sqsupseteq T_{:i}$
- Továbbá $P_{:j} \sqsupseteq T_{:i} \wedge k < j$
- Következésképpen $P_{:k} \sqsupset P_{:j}$ (8. Átfedő szuffix lemma miatt)
- $k > \pi(j)$
- $P_{:k} \not\sqsupset P_{:j}$
  - ui. $P_{:\pi(j)}$ a $P_{:j}$ leghosszabb valódi prefixe, ami egyben szuffixe is, a π prefix függvény definíciója (11) alapján.

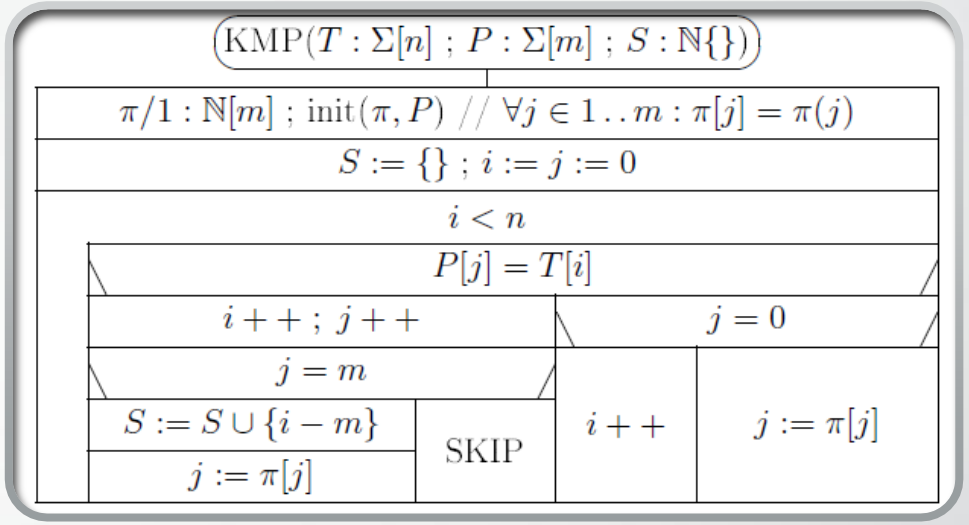
Invariáns helyessége*

18. Tétel. Az (Inv) állítás a KMP(T, P, S) eljárás ciklusának invariánsa.

1. $P_{:j} \sqsupseteq T_{:i} \wedge$
2. $o \leq j \leq i \leq n \wedge$
3. $j < m \wedge$
4. $S = V \cap [o\,..\,i–j)$



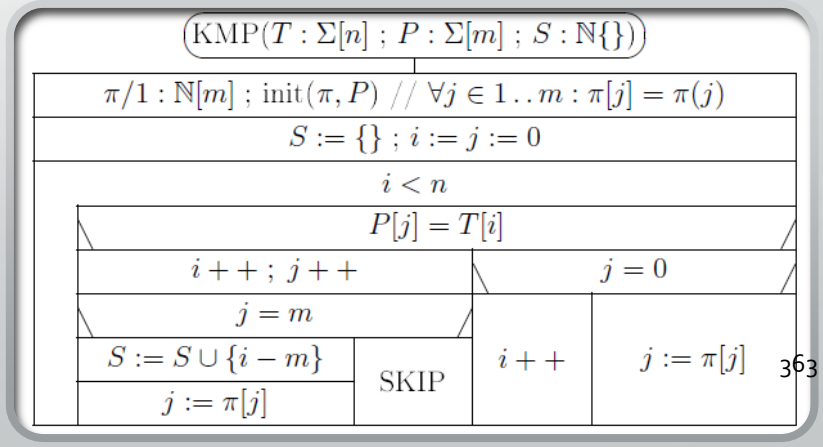
Invariáns helyessége*



- **Bizonyítás.**
  - Kezdetben:
    - $i = j = 0 \Rightarrow P_{:0} \sqsupseteq T_{:0} \wedge 0 \leq o \leq o \leq n \wedge o < m \wedge S = V \cap [o\,..\,o) = V \cap \{\} = \{\}$
  - A továbbiakban:
    - igazoljuk, hogy a ciklusmag végrehajtása (mind a négy programágon) tartja (Inv)-et.
    - Állításainkhoz mindig hozzáértjük init(π, P) utófeltételét.
    - Ha $i < n$ => belépünk a ciklusba
      => (Inv1) $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \wedge S = V \cap [o\,..\,i–j)$ teljesül

Invariáns helyességének bizonyítása*

- Ha $P[j] = T[i]$ =>(9. Szufix kiterjesztési lemma szerint) $P_{:j+1} \sqsupseteq T_{:i+1}$
  - majd i és j növelése után (Inv2) $P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \wedge S = V \cap [0\,..\,i-j)$
  1. Ha $j = m$ => $P_{:m} \sqsupseteq T_{:i}$ (másképpen írva: $P[0\,..m) = T[i-m\,..\,i)$
     - $i-m$ érvényes eltolás, amit S-hez hozzávéve (Inv3) $P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \wedge S = V \cap [0\,..\,i-j]$
     - $j \in 1\,..m \wedge P_{:j} \sqsupseteq T_{:i}$, =>(16. lemma szerint) nincs érvényes eltolás az $(i-j\,..\,i-\pi(j))$ intervallumban
     - $P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \wedge S = V \cap [0\,..\,i-\pi(j))$



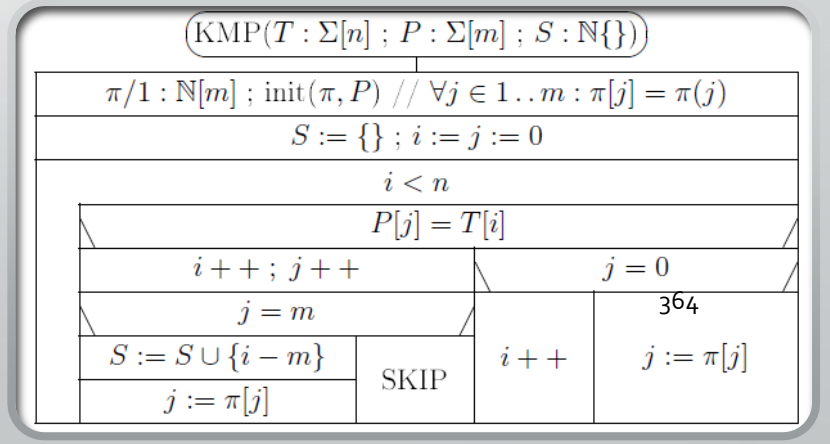
Invariáns helyességének bizonyítása*

- $P_{:\pi(j)} \sqsupset P_{:j} \sqsupseteq T_{:i}$,
- (a szufixum reláció tranzitivitása 7. lemma)
- $\pi[j] = \pi(j)$

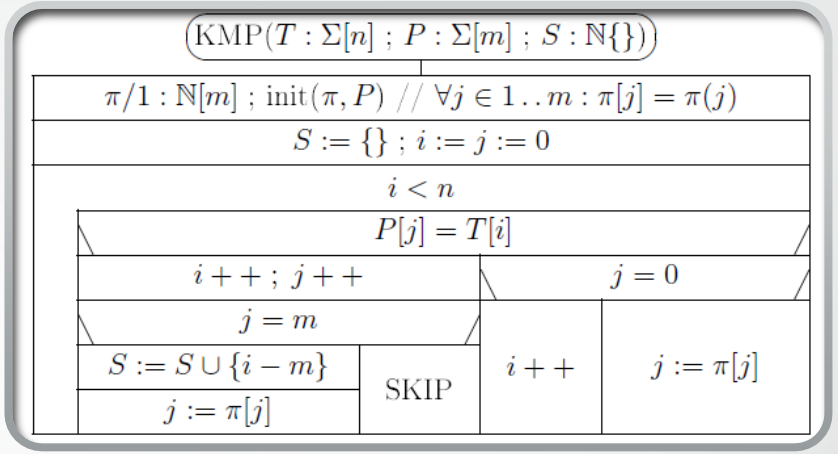
$\Rightarrow$ $P_{:\pi(j)} \sqsupset T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \wedge S = V \cap [0\,..\,i - \pi[j])$

- $\pi[j] = \pi(j) \in [0\,..\,j) \Rightarrow j := \pi[j]$ értékadás után már $0 \leq j < i \wedge j < m$ teljesül
- $P_{:j} \sqsupset T_{:i} \wedge 0 \leq j < i \leq n \wedge j < m \wedge S = V \cap [0\,..\,i - j)$
- az első programág végén közvetlenül következik (Inv).

2. Ha $j \neq m \Rightarrow$ (Inv2 szerint) $j < m$
   - a második programág végén is teljesül (Inv).

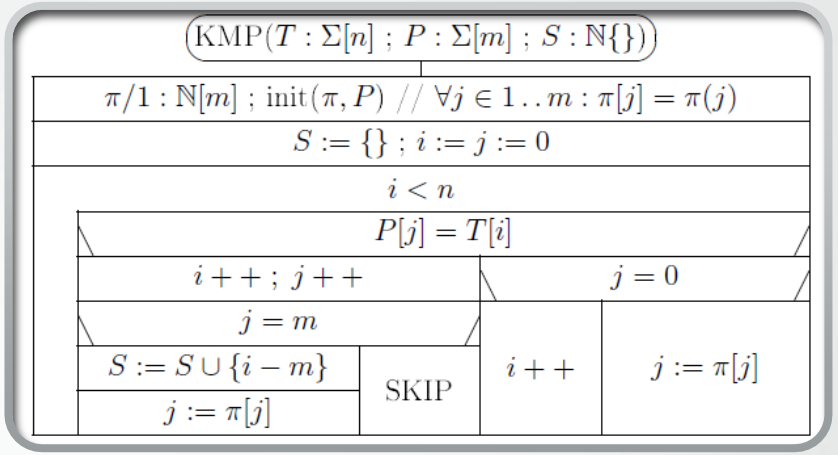


Invariáns helyességének bizonyítása*



- Ha $P[j] \neq T[i]$
  - (9. Szufix kiterjesztési lemma szerint) $P_{:j+1} \not\sqsupseteq T_{:i+1}$ (másképpen írva: $P[0..j] \neq T[i-j..i]$
    - Inv1-ből $j < m$ miatt => $P[0..m) \neq T[i-j..i-j+m)$
    - $i-j$ érvénytelen eltolás
    - \+ Inv1 (azaz a $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \wedge S = V \cap [0..i-j)$ tulajdonsággal
    - (Inv4) $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \wedge S = V \cap [0..i-j]$

Invariáns helyességének bizonyítása*



3. Ha $j = 0$ => (Inv4 figyelembevételével) $P_{:0} \sqsupseteq T_{:i} \wedge 0 = j \leq i < n \wedge j < m \wedge S = V \cap [0\,..\,i\text{–}j]$
   - i növelése után => $P_{:0} \sqsupseteq T_{:i} \wedge 0 = j \leq i \leq n \wedge j < m \wedge S = V \cap [0\,..\,i\text{–}j)$
   - a harmadik programág végén is teljesül (Inv) $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i \leq n \wedge j < m \wedge S = V \cap [0\,..\,i\text{–}j)$
4. Ha $j \neq 0$ 7> (Inv4 figyelembevételével) $P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i < n \wedge j < m \wedge S = V \cap [0\,..\,i\text{–}j]$
   - Ennek közvetlen következménye (Inv3) $P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \wedge S = V \cap [0\,..\,i\text{–}j]$
   - (lásd 1. pr.ág vizsgálata) (Inv3) esetén a $j := \pi[j]$ értékadást végrehajtva teljesül (Inv)
   - 4. programág végén is igaz.

A KMP(T, P, S) eljárás parciális helyessége*

19. Tétel. Ha a KMP algoritmus megáll, akkor megoldja az 3. (mintaillesztési) problémát,

- azaz $S = V$ teljesül, amikor a KMP(T, P, S) eljárás befejeződik

- **Bizonyítás.**
  - A KMP(T, P, S) eljárás ciklusának (Inv) invariánsa (17)
    - azaz $P_{:j} \sqsupseteq T_{:i} \wedge 0 \le j \le i \le n \wedge j < m \wedge S = V \cap [0 \,..\, i-j)$
  - a ciklusfeltétel tagadása
    - vagyis $i \ge n$ szerint

$i = n \wedge j < m \wedge S = V \cap [0 \,..\, n-j)$ teljesül, mikor a ciklus befejeződik

\+ $j < m \Rightarrow n-j > n-m \Rightarrow [0 \,..\, n-j) \supseteq 0 \,..\, n-m \supseteq \{ s \in 0 \,..\, n-m \mid T[s \,..\, s+m) = P[0 \,..\, m) \} = V \Rightarrow [0 \,..\, n-j) \supseteq V$

➢ $S = V \cap [0 \,..\, n-j) = V$

➢ $S = V$ teljesül, amikor a KMP(T, P, S) eljárás befejeződik

Az init(π, P) eljárás parciális helyességének bizonyítása az invariánsok módszerével*

- **Matemaikai igényességű bizonyítások**
- **A** *π prefix függvényre vonatkozó lemma és bizonyítása:*

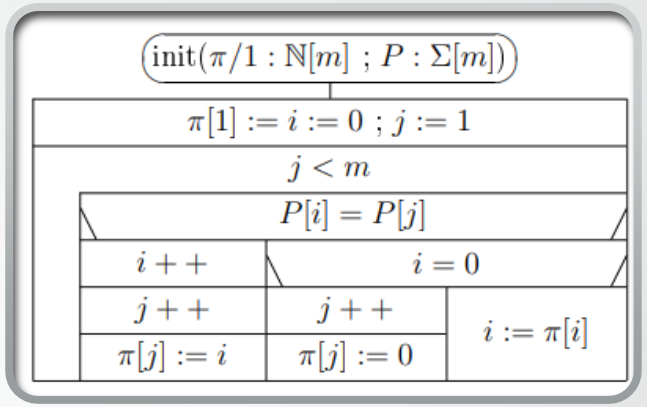
20.Lemma. $P_{:i} \sqsupset P_{:j} \wedge 0 < i < j < m \wedge \pi(j+1) \leq i \Rightarrow \pi(j+1) \leq \pi(i)+1$

- **Bizonyítás.**
  - Ha $\pi(j+1) = 0$
    - $\pi(j+1) < 0+1 \leq \pi(i)+1$
    - definíció szerint $\pi(i) \geq 0$

Az init(π, P) eljárás parciális helyességének bizonyítása az invariánsok módszerével*

- Ha $\pi(j + 1) > 0$
  - $k := \pi(j + 1) - 1$
  - A π függvény definíciója szerint: $P_{:k+1} \sqsupset P_{:j+1} \Rightarrow P_{:k} \sqsupset P_{:j}$
  - \+ $P_{:i} \sqsupset P_{:j}$ és $k < i$
  - $P_{:k} \sqsupset P_{:i}$
  - $k \leq \pi(i)$
  - $\pi(j + 1) = k + 1 \leq \pi(i) + 1$ adódik

Az init(π, P) eljárás ciklusinvariánsa*

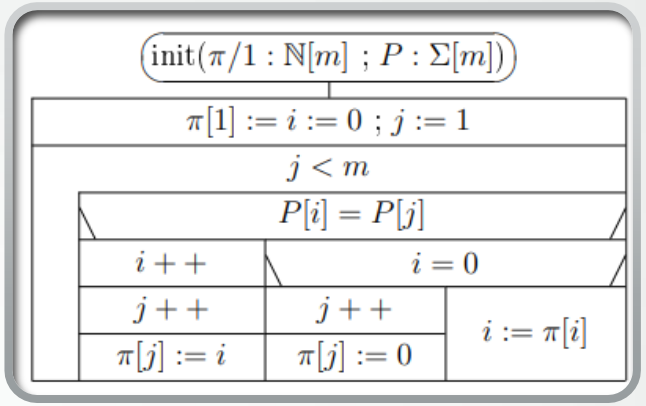


21.Tétel. Az (inv) állítás az init(π, P) eljárás ciklusának invariánsa.

- (inv) $P_{:i} \sqsupset P_{:j} \wedge (\forall k \in 1 \,..\, j : \pi[k] = \pi(k)) \wedge 0 \le i < j \le m \wedge (j < m \rightarrow \pi(j+1) \le i+1)$

- Bizonyítás.
  - Közvetlenül az 1. ciklusiteráció előtt a $\pi[1] := i := 0$; $j := 1$ inicializálások miatt (inv):
    - $P_{:0} \sqsupset P_{:1} \wedge (\forall k \in 1 \,..\, 1 : \pi[k] = \pi(k) = \pi(1) = 0) \wedge 0 \le 0 < 1 \le m \wedge (1 < m \rightarrow \pi(2) \le 1)$
    - Igazak, mert
      - $P_{:0}$ üres sztring bármely nem nemüres sztringnek valódi szufixe
      - (12. tulajdonság szerint) $\pi(1) = 0$
      - a P minta m mérete nem nulla
      - (13. lemma szerint) $\pi(1+1) \le \pi(1) + 1 = 1$, feltéve, hogy $m > 1$

Az init(π, P) eljárás ciklusinvariánsa*

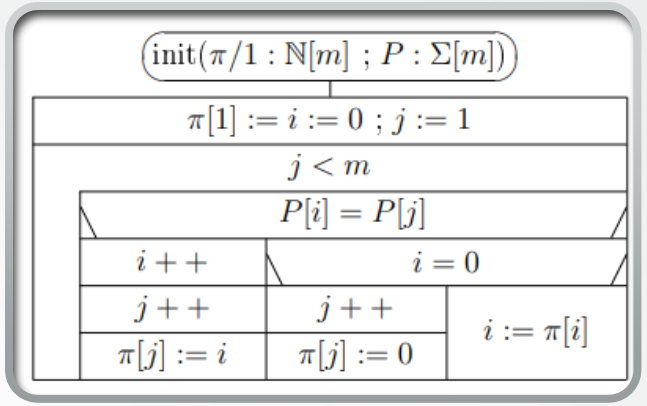


- Bizonyítás folytatása
  - A ciklusiterációk tartják az (inv) tulajdonságot
    - azaz, ha tetszőleges ciklusiteráció előtt (inv) és a ciklusfeltétel igaz => a ciklusmag bármelyik ágának a végére jutva ugyancsak igaz lesz
    - Amikor belépünk a ciklusmagba => (inv) és a ciklusfeltétel alapján (inv1) teljesül:
      - (inv1) $P_{:i} \sqsupset P_{:j} \wedge (\forall k \in 1 \,..\, j : \pi[k] = \pi(k)) \wedge 0 \leq i < j < m \wedge \pi(j+1) \leq i+1$.

    1. Ha $P[i] = P[j]$
       - (9. lemma és inv1-ből a $P_{:i} \sqsupset P_{:j}$ alapján) $P_{:i+1} \sqsupset P_{:j+1}$
       - (a π prefix függvény definíciója (11) szerint) $\pi(j+1) \geq i+1$
       - (inv1 szerint) $\pi(j+1) \leq i+1$.

       ➢ $\pi(j+1) = i+1$

Az init(π, P) eljárás ciklusinvariánsa*

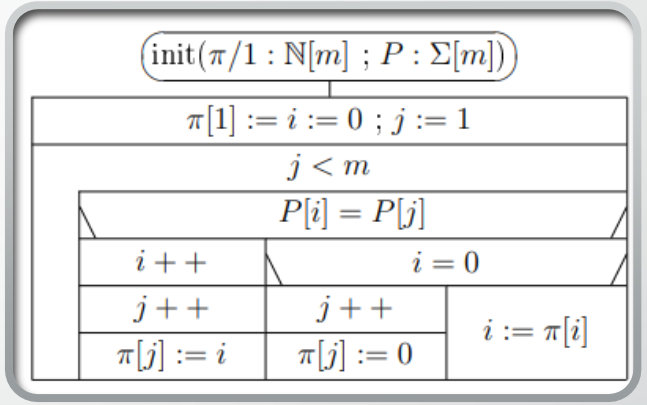


- Bizonyítás folytatása
  - Az i + +; j + +; $\pi[j] := i$ értékadások után pedig:
    - $P_{:i} \sqsupset P_{:j} \wedge (\forall k \in 1 . . j : \pi[k] = \pi(k)) \wedge 0 < i < j \leq m \wedge \pi(j) = i$
  - Ha $j < m$ => (13. lemma és $\pi(j) = i$ szerint) $\pi(j + 1) \leq \pi(j)+1 = i+1$
  - A ciklusmag első ágának végén tehát teljesül az (inv) állítás

2-3. Ha $P[i] \neq P[j]$

- (1.9. lemma szerint) $P_{:i+1}\, 1 \not\sqsupset P_{:j+1}$
- (a π prefix függvény definíciója (11) szerint) $\pi(j+1) \neq i+1$
- (inv1 alapján) $\pi(j + 1) \leq i + 1$
- $\pi(j + 1) \leq i$
- Ezt (inv1)-gyel összevetve, a belső elágazás előtt (inv2) teljesül
  - (inv2) $P_{:i} \sqsupset P_{:j} \wedge (\forall k \in 1 . . j : \pi[k] = \pi(k)) \wedge 0 \leq i < j < m \wedge \pi(j + 1) \leq i.$

Az init(π, P) eljárás ciklusinvariánsa*



- Bizonyítás folytatása

2. Ha $i = 0$

- (inv2)-ből + (a $\pi(j + 1) \leq i$ állítás) => $\pi(j + 1) = 0$
  - mivel a π függvény nemnegatív

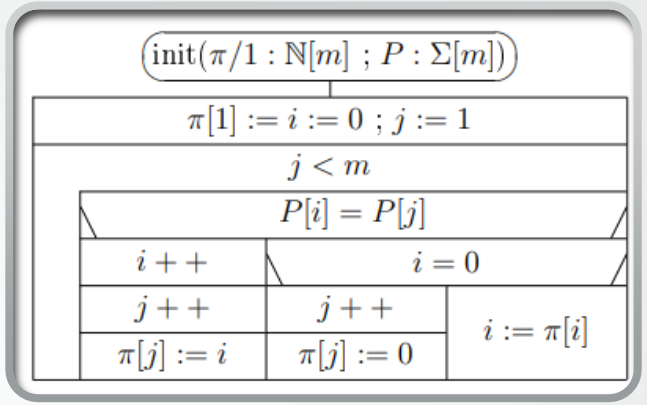
+(inv2) a j+ +; $\pi[j] := 0$ értékadások után:

- $P_{:i} \sqsupset P_{:j} \wedge (\forall k \in 1 \,..\, j : \pi[k] = \pi(k)) \wedge 0 = i < j \leq m \wedge \pi(j) = i.$

- Ha $j < m$ => (13. lemma és $\pi(j) = i$ szerint) $\pi(j + 1) \leq \pi(j) + 1 = i + 1$

- A ciklusmag második ágának végén is teljesül az (inv) állítás.

Az init(π, P) eljárás ciklusinvariánsa*



- Bizonyítás folytatása

2. Ha $i \neq 0$

- (inv2 szerint) inv3 teljesül
  - (inv3) $P_{:i} \sqsupset P_{:j} \wedge (\forall k \in 1 \,..\, j : \pi[k] = \pi(k)) \wedge 0 < i < j < m \wedge \pi(j+1) \leq i$
- $i > 0$ => az 1.19. lemma feltételei teljesülnek => $\pi(j+1) \leq \pi(i) + 1$
- (inv3 2. és 3. tényezője figyelembevételével) $\pi[i] = \pi(i)$
- + π prefix függvény definíciója (11) => $P_{:\pi[j]} \sqsupset P_{:i}$
- +(az inv3 $P_{:i} \sqsupset P_{:j}$ állítása + az 7. tranzitivitási lemma => $P_{:\pi[j]} \sqsupset P_{:j}$
- (inv3) alapján, az $i := \pi[i]$ értékadás után:
  - $P_{:i} \sqsupset P_{:j} \wedge (\forall k \in 1 \,..\, j : \pi[k] = \pi(k)) \wedge 0 \leq i < j < m \wedge \pi(j+1) \leq i + 1$
- A ciklusmag utolsó ágának a végén is teljesül az (inv) állítás

Az init(π, P) eljárás parciális helyessége*

22. Következmény. Ha az init(π, P) eljárás megáll, akkor a visszatérésekor teljesül az utófeltétele:

- $\forall k \in 1 \,..\, m : \pi[k] = \pi(k)$

- **Bizonyítás.**
  - Az init(π, P) eljárásnak a 20. tételből ismerős (inv) invariánsa és a ciklusfeltétel tagadása:
    - azaz $j \geq m$ alapján $j = m$
  - \+ az (inv) invariáns ($\forall k \in 1 \,..\, j : \pi[k] = \pi(k)$) tényezője
  - ➢ az init(π, P) eljárás fenti utófeltétele azonnal adódik.

Ellenőrző kérdések: QuickSearch

1. Mit számol ki a Quick Search algoritmus?
   - Mi az előnye, illetve hátránya a naiv mintaillesztő algoritmussal összehasonlítva?
   - Mutassa be a Quick Search algoritmus (a) előkészítő eljárásának működését
     - az {*A;B;C;D*} ábécé-vel
     - az *ABACABA* mintán
   - És e mintát illesztő eljárását
     - az *ABABA CABAC ABADA BACAB ABA* szövegen!
   - Mekkora az egyes eljárások aszimptotikus műveletigénye?

Ellenőrző kérdések: Knuth-Morris-Pratt (KMP)

1. Definiálja a KMP algoritmusban a keresett P[1:m] mintához tartozó π függvényt (nem a struktogramot)!
   - Adja meg a π függvényt a definíciója alapján
     - a *BABA ABAB* mintán!
2. Szemléltesse a KMP algoritmus működését,
   - ahogy a fenti minta előfordulásait keresi
   - az *ABABA BAABA BAABA BABAA BABBA BA* szövegben!

Köszönöm a figyelmet!

Pusztai Kinga

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

Algoritmusok és adatszerkezetek I.
11. Előadás

Veszteségmentes adattömörítés.

Tartalom

Információtömörítés

- Veszteségmentes (lossless) tömörítések
  - Pontosan rekonstruálható az eredeti bemenet.
  - Pl. ZIP, ARJ, PNG, FLAC, RAR, 7Z tömörítések
  - Ezekkel foglalkozunk
- Veszteséges (lossy) tömörítések
  - Pl. MP3, MP4, JPG
  - Elhagyják azokat a részleteket, amiket egy átlagember már nem érzékel
  - ➢ Drasztikusan csökkentik a fájlméretet
- A tárgyalt módszerek előfeltételei
  - Bemenet egy véges, nemüres $\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_d \rangle$ ábécé feletti szöveg

Naiv módszer

- A tömörítendő szöveget karakterenként, x hosszúságú bitsorozatokkal kódoljuk.
- $\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_d \rangle$ az ábécé.
- Egy-egy karakter $\lceil \lg d \rceil$ bittel kódolható
  - $\lceil \lg d \rceil$ biten $2^{\lceil \lg d \rceil}$ különböző bináris kód ábrázolható
  - $2^{\lceil \lg d \rceil} \geq d > 2^{\lceil \lg d \rceil - 1}$ => $\lceil \lg d \rceil$ biten ábrázolható d-féle különböző kód, de eggyel kevesebb biten már nem.
- T: $\Sigma\langle\rangle$ a tömörítendő szöveg. $n = |T|$ jelöléssel $n * \lceil \lg d \rceil$ bittel kódolható
- A tömörített fájl a kódtáblázatot is tartalmazza
- Kitömörítés:
  - A kódtáblázat alapján $\lceil \lg d \rceil$ bites szakaszokra bontható
- A kódtáblázat mérete miatt a gyakorlatban csak hosszabb szövegeket érdemes így tömöríteni

Példa:

karakter	kód
A	000
B	001
D	010
K	011
R	100

- ABRAKADABRA szöveg
  - $d = 5$ és $n = 11$
  - ➢ a tömörített kód hossza $11 * \lceil \lg 5 \rceil = 11 * 3 = 33$ bit.
  - (A 3-bites kódok közül tetszőleges 5 kioszthető az 5 betűnek)
  - A fenti ABRAKADABRA szöveg kódtáblázata lehet pl. a következő:
  - A fenti kódtáblázattal a tömörített kód a következő lesz:
    - 000001100000011000010000001100000
    - $\lceil \lg d \rceil$ Ez a tömörített fájlba foglalt kódtáblázat alapján könnyedén 3 bites szakaszokra bontható és kitömöríthető

Huffman-kód

- Karakterenkénti kódoló
  - Huffman-kód hossza minimális
- Változó hosszúságú bitsorozatok
  - A gyakrabban előforduló karakterek kódja rövidebb
  - A ritkábban előfordulóké hosszabb
- **Prefix-mentes kód:** Egyetlen karakter kódja sem prefixe semelyik másik karakter kódjának sem

Huffman-kód

- A szöveghez többféle kódfa és hozzátartozó kódtáblázat építhető
  - Mindegyik segítségével az input szövegnek ugyanolyan hosszú a tömörített kódja
- Betömörítés:
  - a kódtáblával
- Kitömörítés
  - a kódfával

➢ A tömörített fájl a kódfát is tartalmazza

Huffman-kód

- A tömörítendő fájlt, illetve szöveget kétszer olvassa végig
  1. Olvasásnál:
     - Meghatározza a szövegben előforduló karakterek halmazát
     - az egyes karakterek gyakoriságát
     - majd ennek alapján kódfát
     - abból pedig kódtáblázatot épít
  2. Olvasásnál:
     - A kódtábla alapján kiírja az output fájlba sorban a karakterek bináris kódját

Kódfa

- szigorúan bináris fa
  - Levél: az ábécé karakterei és annak gyakoriságai (előfordulásainak száma)
  - A belső csúcsok: a csúcshoz tartozó részfa leveleit címkéző karakterek gyakoriságainak összegével címkézzük.
  - ➢ A kódfa gyökere: a tömörítendő szöveg hossza

A kódfa felépítése:

- Kiindulás: egy csúcsú fák – egy csúcs és annak gyakorisága
- Minimum prioritásos sor: a fák gyökerét címkéző gyakoriságértékek szerint

A kódfa felépítése:

- Ciklus, amíg a kupac még legalább kettő fából áll:
  - Kiveszünk a kupacból egy olyan fát, amelyeknek gyökerét a legkisebb gyakoriság címkézi
  - Ezután a maradék kupacra ezt még egyszer megismételjük
  - Összeadjuk a két gyakoriságot
  - Az összeggel címkézünk egy új csúcsot, amelynek bal és jobb részfája az előbb kiválasztott két fa lesz
    - A bal ágat a 0, a jobb ágat az 1 címkézi
  - Az így képzett új fát visszatesszük a minimumprioritásos sorba
- A minimum-prioritásos sorban maradó egyetlen bináris fa: Huffman-féle kódfa

Kódtáblázat, kódolás

- A kódfából -> kódtáblázat
- Egy karakter kódja:
  - a kódfa gyökerétől elindulva és a karakterhez tartozó levélig lefelé haladva a kódfa éleit címkéző biteket összeolvassuk
  - (pl. a kódfa preorder bejárásával, az aktuális csúcshoz vezető bitsorozat folyamatos nyilvántartásával, és levélhez érve, a kódtáblázatba írásával.)
- Kódolás:
  - a tömörítendő szöveg 2. végigolvasása
  - a kódtáblázat segítségével sorban mindegyik karakter bináris kódját a (kezdetben üres) tömörített bitsorozat végéhez fűzzük
- A tömörített fájl a kódfát is tartalmazza

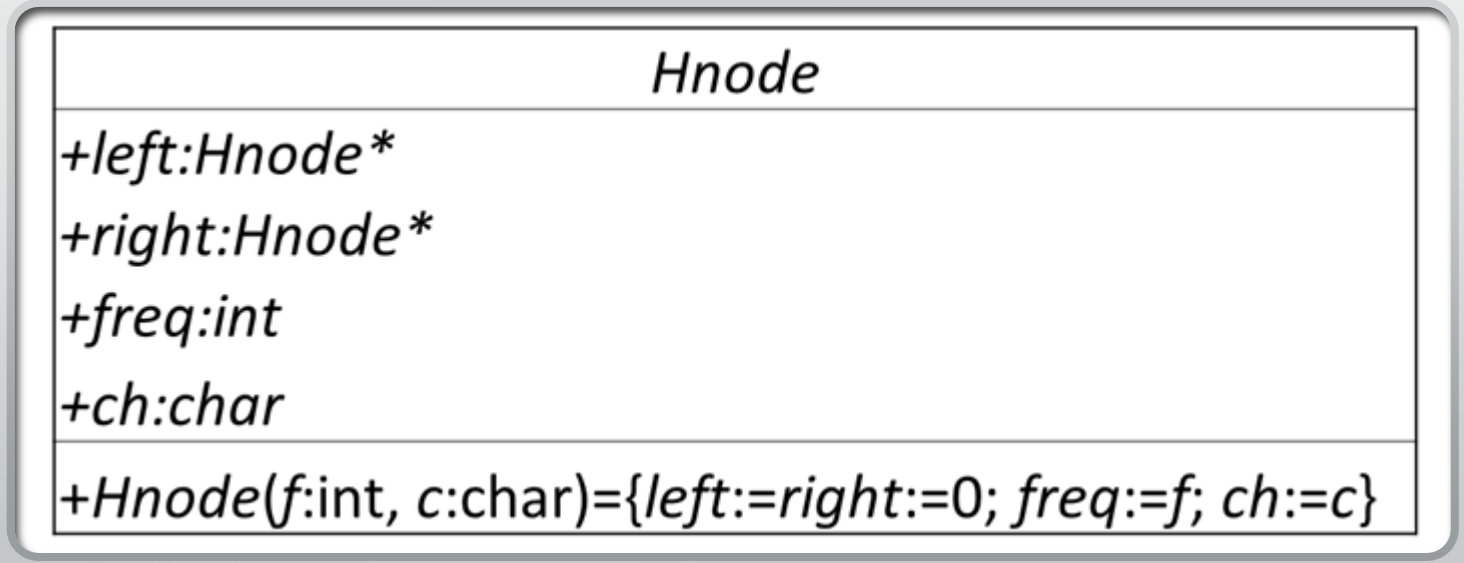
➢ A gyakorlatban Huffman-kódolással is csak hosszabb szövegeket érdemes tömöríteni

Kitömörítés

- Karakterenként
  - Mindegyik karakter kinyeréséhez a kódfa gyökerétől indulunk
  - majd a tömörített kód sorban olvasott bitjeinek hatására
    - 0: balra lépünk
    - 1: jobbra lépünk lefelé a fában
  - Amíg levélcsúcshoz érünk
  - Ekkor kiírjuk a levelet címkéző karaktert
  - Folytatjuk az eljárást a következő bittől és újra a kódfa gyökerétől folytatjuk
  - amíg a tömörített kódon végig nem érünk.

Huffman kódolás – algoritmus*

- A kódfa típusa: *Hnode*



Huffman kódolás*

HuffmanEncode(text:String; &root:Hnode*; &encoded:String)

```
F:dictionary<char,N>
foreach c in text
    F.ContainsKey(c)
    T: F[c]++
    F: F.Add(c,1)
root:=HuffmanCodeTree(F)
D:Item{} //D is the dictionary, initially empty
BuildCodes(root,"",D)
encoded:bit< >
foreach c in text
    encoded:=encoded+D[c]
```

HuffmanCodeTree(F:dictionary<char,N>):Hnode*

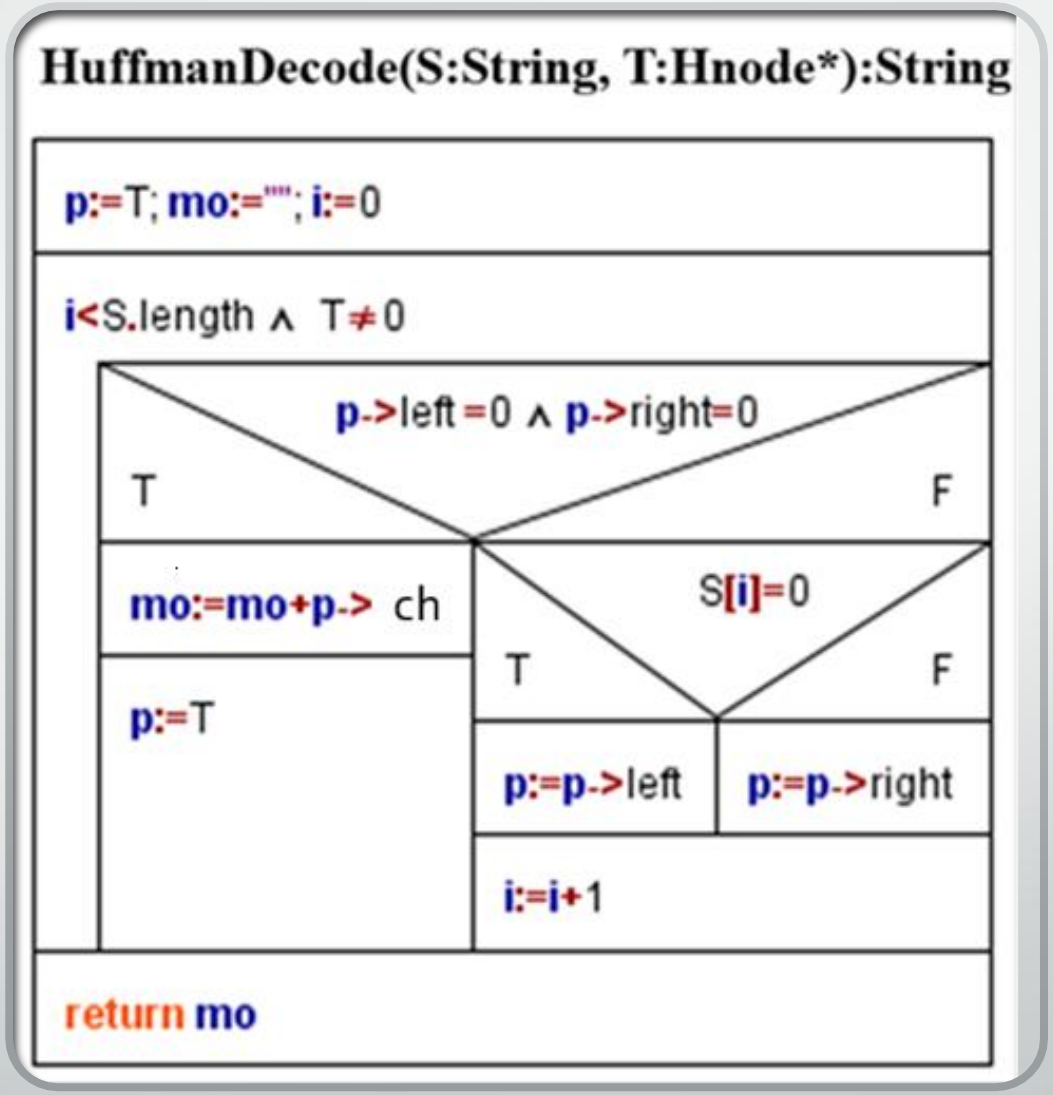
```
Q:minPrQueue // a Hnode-ok freq adattagja szerint
foreach σ ∈ F
    p:=new Hnode(σ.Value, σ.Key)
    Q.add(p)
Q.length() > 1
    p:=Q.remMin()
    q:=Q.remMin()
    s:=new Hnode(p->freq+q->freq,' ')
    s->left:=p
    s->right:=q
    Q.add(s)
return Q.remMin()
```

BuildCodes (n:Hnode*; s:bit< >; D: Item{})

n->left=0 and n->right=0

T	F
x:=Item(n->ch, s); D:= D U {x}	BuildCodes(n->left, s+<0>, D)
	BuildCodes(n->right, s+<1>, D)

Huffman dekódolás*



Huffman kódolás C# kód*

```
public (string encoded, Node root) HuffmanEncode(string text)
{
    var freq = text.GroupBy(c => c).ToDictionary(g => g.Key, g => g.Count());
    var pq = new PriorityQueue<Node, int>();

    foreach (var kv in freq)
        pq.Enqueue(new Node(kv.Key, kv.Value), kv.Value);

    while (pq.Count > 1)
    {
        pq.TryDequeue(out var a, out _);
        pq.TryDequeue(out var b, out _);

        var parent = new Node(null, a.Freq + b.Freq, a, b);
        pq.Enqueue(parent, parent.Freq);
    }
```

Huffman kódolás C# kód folytatás*

```
pq.TryDequeue(out var root, out _);

var codes = new Dictionary<char, string>();
BuildCodes(root, "", codes);

var sb = new StringBuilder();
foreach (var c in text)
    sb.Append(codes[c]);

return (sb.ToString(), root);
}
```

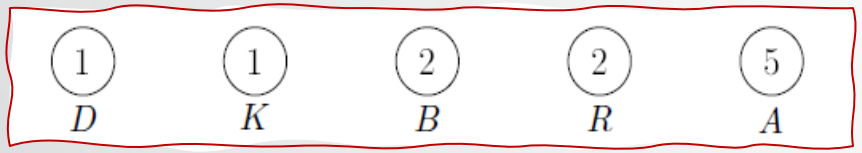
```
void BuildCodes(Node n, string s, Dictionary<char, string> d)
{
    if (n.Ch != null)
    {
        d[n.Ch.Value] = s;
        return;
    }
    BuildCodes(n.Left,  s + "0", d);
    BuildCodes(n.Right, s + "1", d);
}
```

Huffman dekódolás C# kód*

```
public string HuffmanDecode(string encoded, Node root)
{
    var sb = new StringBuilder();
    var node = root;

    foreach (var bit in encoded)
    {
        node = (bit == '0') ? node.Left : node.Right;

        if (node.Ch != null)
        {
            sb.Append(node.Ch.Value);
            node = root;
        }
    }

    return sb.ToString();
}
```

Huffman-kódolás szemléltetése

- A szöveg: ABRAKADABRA
- Gyakoriságtáblázat:
- Prioritásos sor:

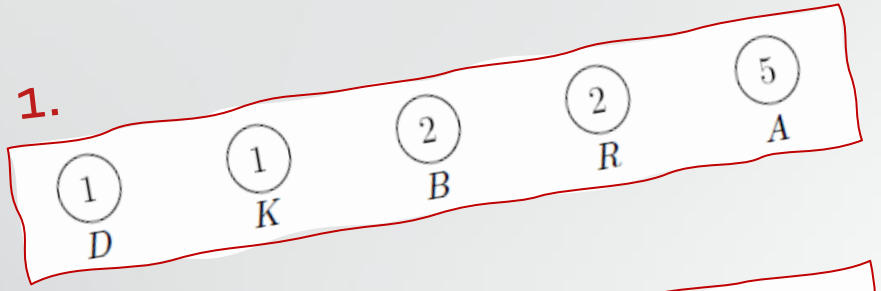


szöveg:	A	B	R	A	K	A	D	A	B	R	A
A	1			2		3		4			5
B	-	1							2		
D	-	-	-	-	-	-	1				
K	-	-	-	-	1						
R	-	-	1							2	

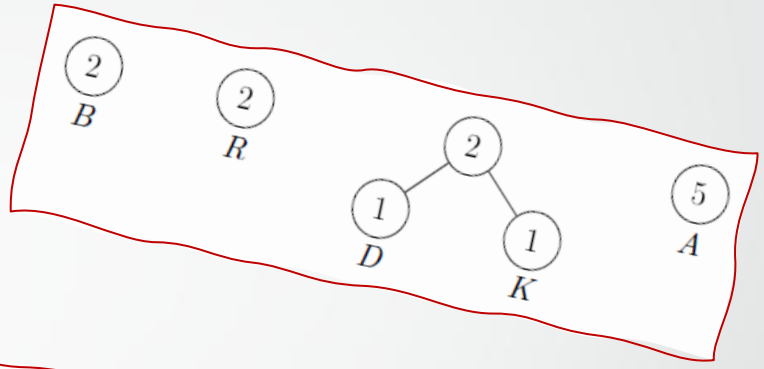
- Faépítés:
  - Kivesszük a két legkisebb gyakoriság-értékű fát
    - Azonos gyakoriság esetén elsősorban a fák magassága, másodsorban a fák frontját címkéző szavak alfabetikus sorrendje szerint rendezzük.
  - Egy új gyökércsúcs alá tesszük őket bal- és jobboldali részfának
  - Az új gyökércsúcs: a két fa-gyakoriság-érték összegével címkézzük
  - Visszatesszük az új fát a minimum-prioritásos sorba.

Kódfa építése

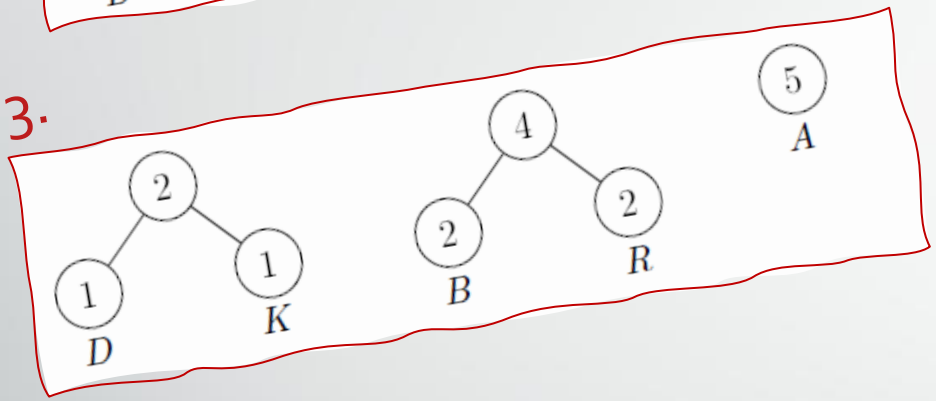
1.



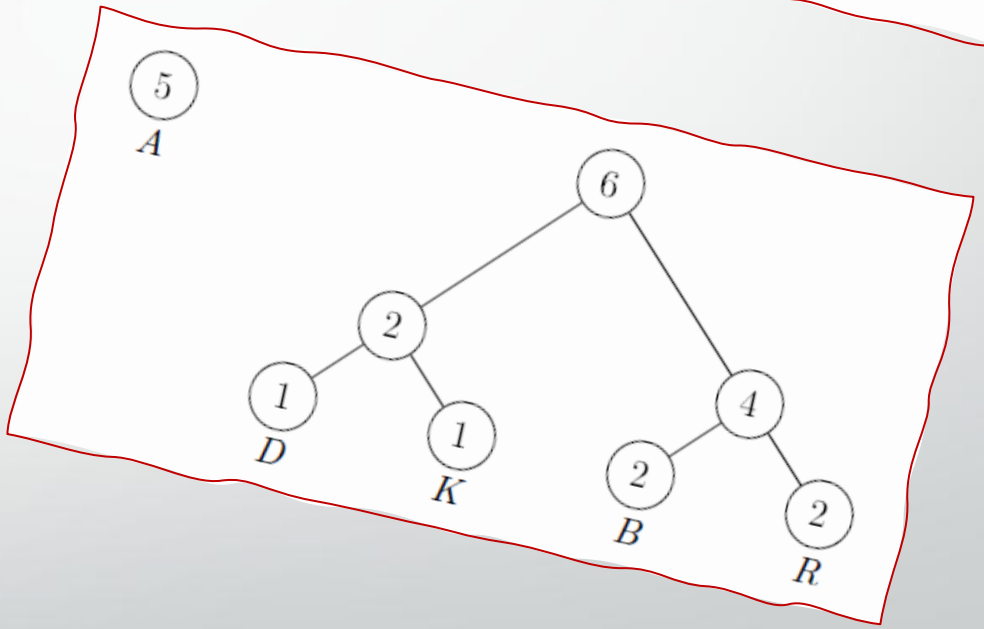
2.



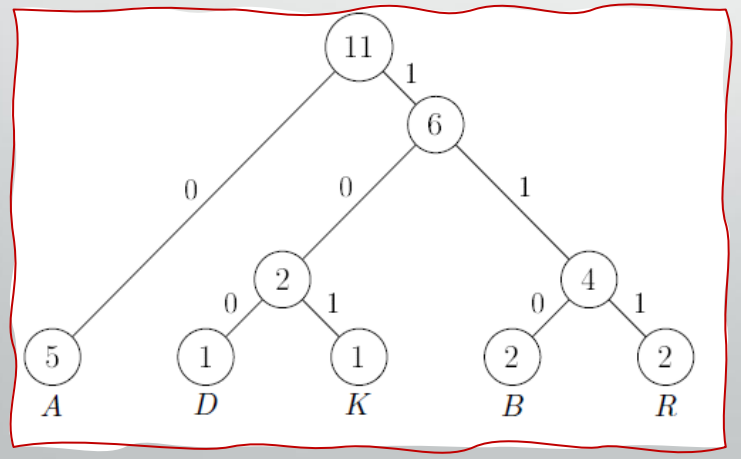
3.



4.



5.



Huffman-kódolás szemléltetése

- Kódtáblázat:
  - Az út éleit címkéző biteket összeolvasva adódik a levelet címkéző karakter Huffman-kódja
- Szöveg Huffman kódja:
  - 01101110101010001101110
  - 23 bit
- Kitömörítés:
  - Huffman-kód és a kódfa alapján:
    - 0=>A
    - 110 => B
    - 111 => R
    - 0 => A
    - 101 =>K
    - 0 => A
    - 100 =>D
    - 0 => A
    - 110 => B
    - 111 => R
    - 0 => A

betű	kód
A	0
B	110
D	100
K	101
R	111

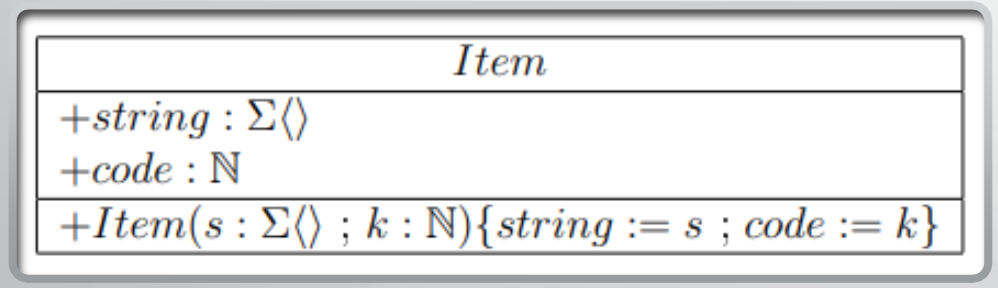
Lempel-Ziv-Welch (LZW) módszer

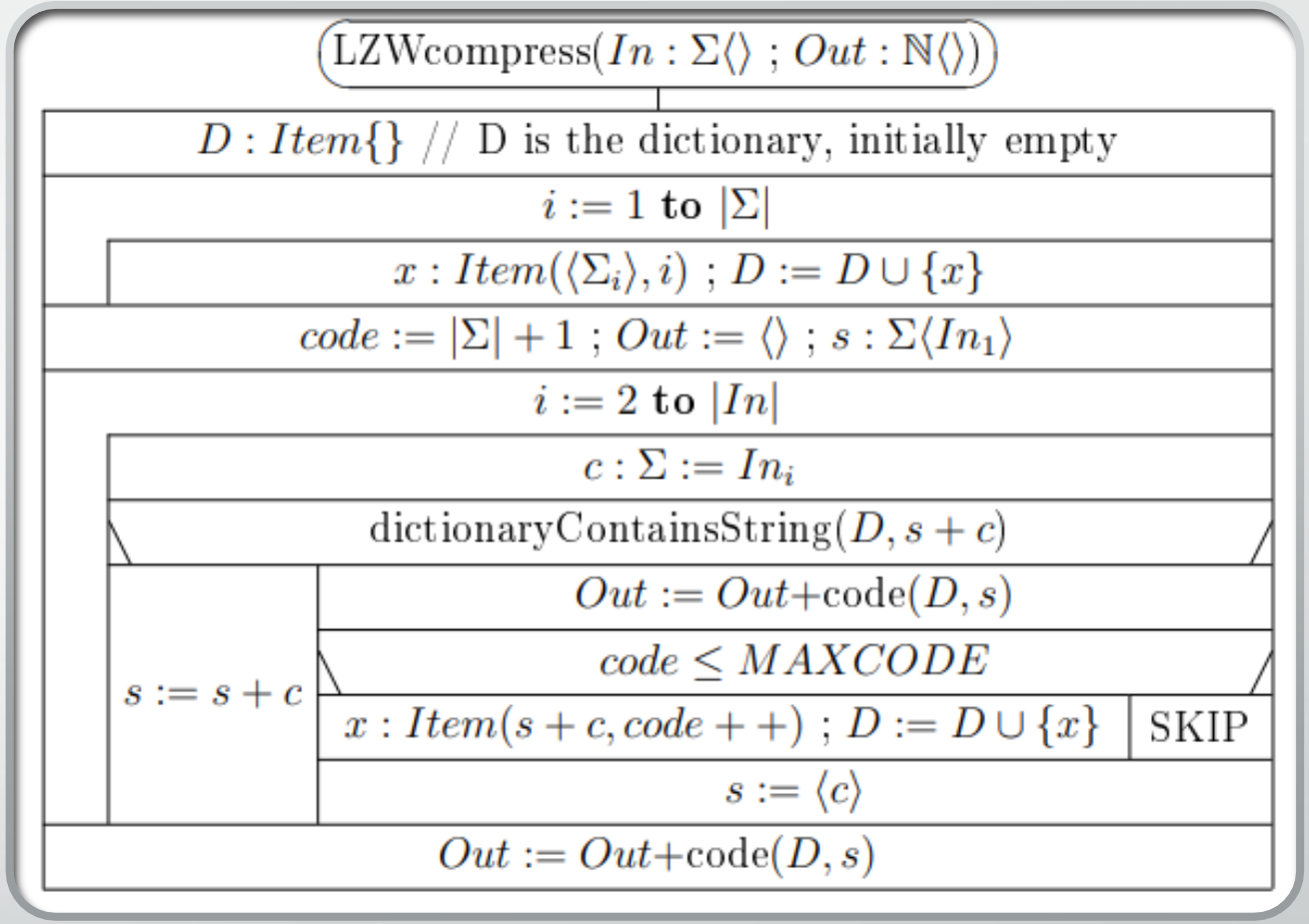
- Heurisztikus algoritmus
- Optimalitást nem garantál, de tapasztalat alapján 30%-kal jobb a Huffman-kódolásnál
- ZIP tömörítés az LZW elven működik.
- Az input szöveget ismétlődő mintákra (változó hosszúságú sztringekre) bontja
- Mindegyik mintát ugyanolyan hosszú bináris kóddal helyettesíti
- A tömörített fájl a kódtáblázatot nem tartalmazza
  - A kitömörítés rekonstruálja az ábécé és a tömörített kód alapján

Jelölések az absztrakt struktogramokhoz

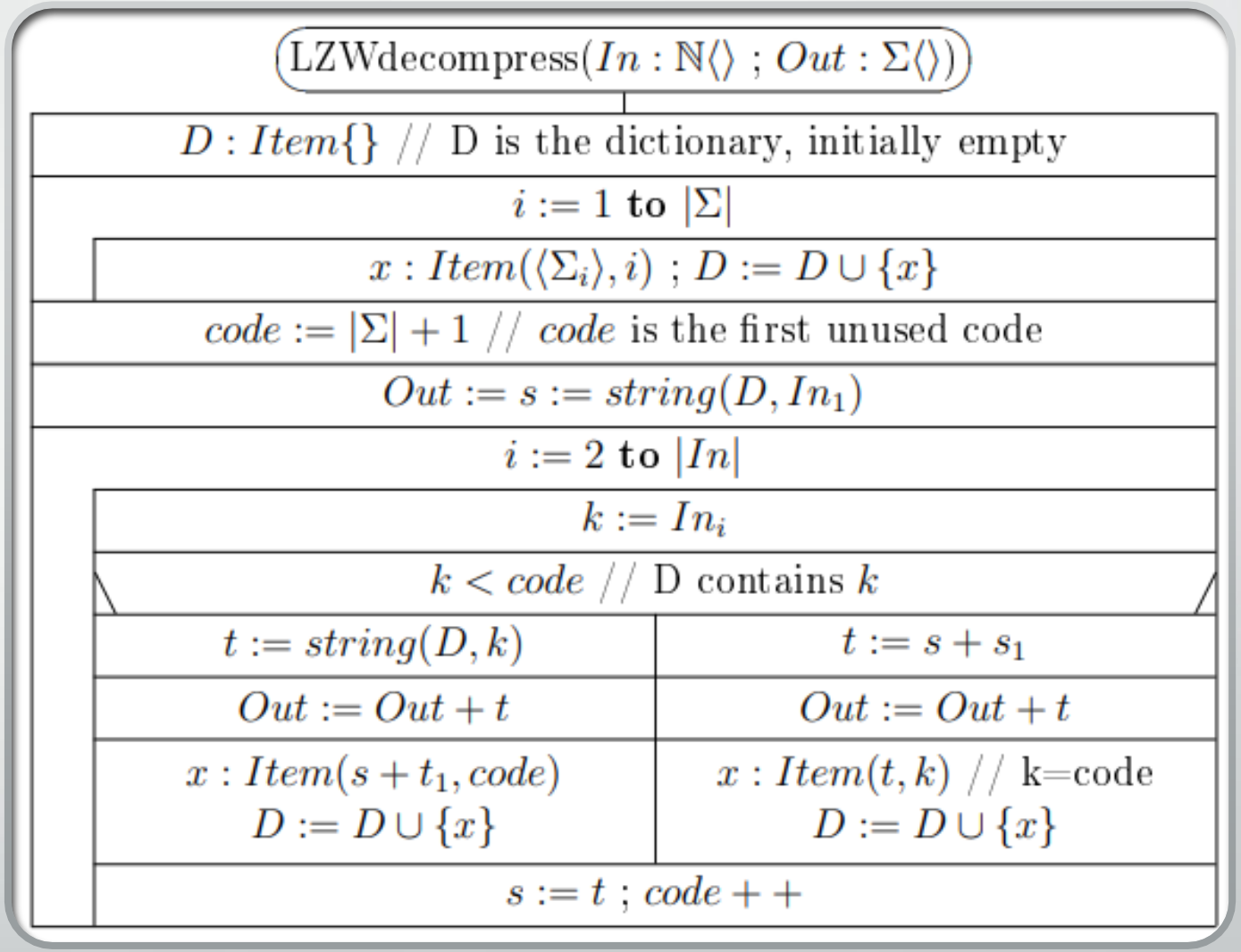
- Ha a kódok *b* bitesek => MAXCODE = 2^b-1 globális konstans a kódként használható legnagyobb számérték
- Pl. $b = 12$ => MAXCODE = $2^{12} - 1 = 4095$
- $\Sigma = <\sigma_1, \sigma_2, \ldots, \sigma_d>$ sorozat tartalmazza az ábécé karaktereit $(d \in \mathbb{N} \wedge d > 0)$
- Tömörítésnél:
  - *In*: a tömörítendő szöveg
  - *Out*: a tömörítés eredménye: kódok sorozata
- Kitömörítésnél:
  - *In – Out* fordítva
  - *D*: szótár, ami (string, code) rendezett párok, azaz Item-ek halmaza

LZW kódolás stuktogramja*





LZW dekódolás stuktogramja*



LZW kódolás C# kódja*

```
public class LZW
{
    public static List<int> Compress(string input)
    {
        // Szótár inicializálása (ASCII karakterek)
        Dictionary<string, int> dictionary = new Dictionary<string, int>();
        for (int i = 0; i < 256; i++)
        {
            dictionary[((char)i).ToString()] = i;
        }

        string w = "";
        List<int> result = new List<int>();
        int dictSize = 256;

        foreach (char c in input)
        {
            string wc = w + c;
            if (dictionary.ContainsKey(wc))
            {
                w = wc;
            }
            else
            {
                result.Add(dictionary[w]);
                dictionary[wc] = dictSize++;
                w = c.ToString();
            }
        }

        // utolsó elem
        if (!string.IsNullOrEmpty(w))
        {
            result.Add(dictionary[w]);
        }

        return result;
    }
```

LZW dekódolás C# kódja

```
public static string Decompress(List<int> compressed)
{
    // Szótár inicializálása
    Dictionary<int, string> dictionary = new Dictionary<int, string>();
    for (int i = 0; i < 256; i++)
    {
        dictionary[i] = ((char)i).ToString();
    }

    int dictSize = 256;

    string w = dictionary[compressed[0]];
    StringBuilder result = new StringBuilder(w);

    for (int i = 1; i < compressed.Count; i++)
    {
        int k = compressed[i];
        string entry;

        if (dictionary.ContainsKey(k))
        {
            entry = dictionary[k];
        }
        else if (k == dictSize)
        {
            // speciális eset: s + első karaktere s-nek
            entry = w + w[0];
        }
        else
        {
            throw new Exception("Hibás tömörített adat!");
        }

        result.Append(entry);

        // új elem a szótárba
        dictionary[dictSize++] = w + entry[0];

        w = entry;
    }

    return result.ToString();
}
```

LZW főprogram C# kódja és kimenete*

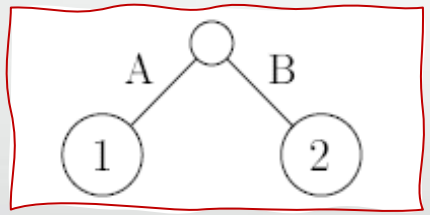
```
class Program
{
    static void Main()
    {
        string input = "ABABABA";

        var compressed = LZW.Compress(input);
        Console.WriteLine("Tömörített: " + string.Join(", ", compressed));

        var decompressed = LZW.Decompress(compressed);
        Console.WriteLine("Visszafejtett: " + decompressed);
    }
}
```

- 'A' → 65
- 'B' → 66
- "AB" → 256
- "ABA" → 258

```
Tömörített: 65, 66, 256, 258
Visszafejtett: ABABABA
```

Példa LZW kódolásra

- $\Sigma = \{A, B\}$
- Szöveg: *ABAB BABABAB ABAA*
  - Szóközök nem részei a bementnek
- Kezdeti szófa:
  - Ábécé betűinek kódjait tartalmazza
  - Kód: gyökérből a csúcshoz vezető út mentén az éleket címkéző betűkből összeolvasott szó kódja

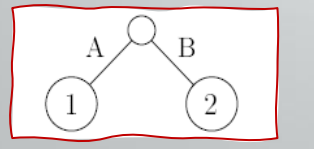


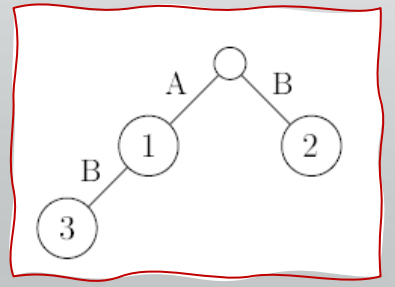
- A tömörítés lépései:
  - Minden lépésben a maradék szöveg leghosszabb olyan kezdőszeletét választjuk le, amely megtalálható a szótárban
- Szófa szinonimája: prefix-fa
  - A szótár új szavainak képzése:
    - Egy régi szó végéhez egy-egy betűt hozzátoldunk
  - ➢ Minden szótárbeli szónak minden prefixét a szótár tartalmazza

A tömörítés lépései

- 1. lépés:
  - *ABAB BABABAB ABAA* szöveg
  - *A* ~1 (szófa segítségével)
  - de *AB* még nincs a szótárban
  - ➢ *A* kódját kiírjuk,
    az *AB* kódját felvesszük a szótárba

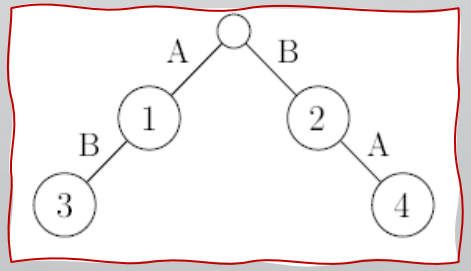
Bemenet:	*A*	*BAB BABA BAB ABA A*
Kimenet:	1	





- 2. lépés:
  - *BAB BABABAB ABAA* szöveg
  - *B* ~2 (szófa segítségével)
  - de *BA* még nincs a szótárban
  - ➢ *B* kódját kiírjuk,
    az *BA* kódját felvesszük a szótárba

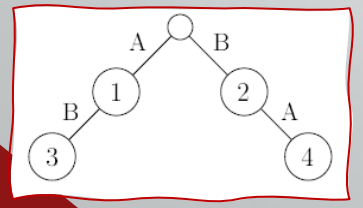
Bemenet:	*A*	*B*	*AB BABA BAB ABA A*
Kimenet:	1	2	

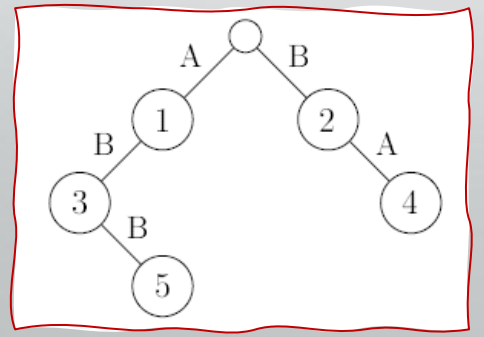


A tömörítés lépései

- 3. lépés:
  - *AB BABABAB ABAA* szöveg
  - *AB* ~3 (szófa segítségével)
  - de *ABB* még nincs a szótárban
  - ➢ *AB* kódját kiírjuk, az *ABB* kódját felvesszük a szótárba

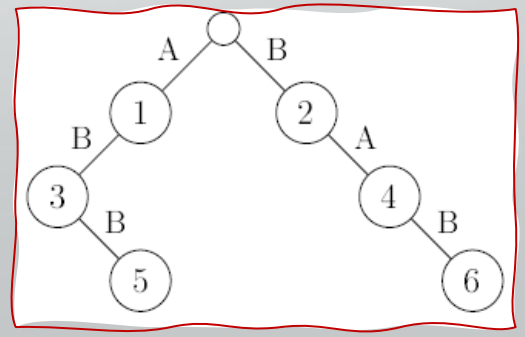
Bemenet:	*A*	*B*	*AB*	*BABA BAB ABA A*
Kimenet:	1	2	3	





- 4. lépés:
  - *BABABAB ABAA* szöveg
  - B*A* ~4 (szófa segítségével)
  - de B*AB* még nincs a szótárban
  - ➢ *BA* kódját kiírjuk, az *BAB* kódját felvesszük a szótárba

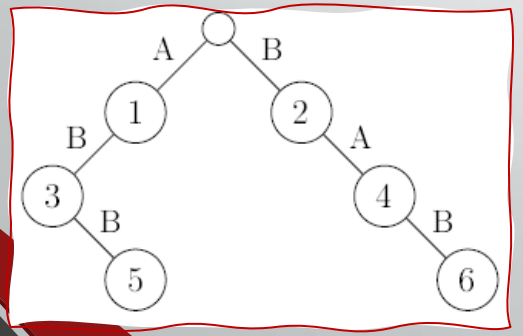
Bemenet:	*A*	*B*	*AB*	*BA*	*BA BAB ABA A*
Kimenet:	1	2	3	4	

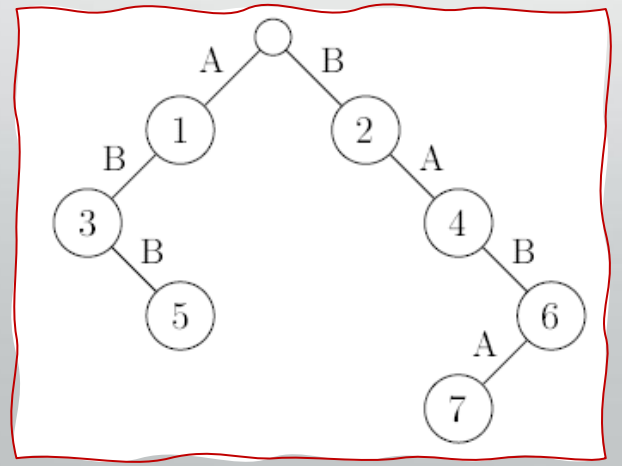


A tömörítés lépései

- 5. lépés:
  - *BABAB ABAA* szöveg
  - *BAB* ~6 (szófa segítségével)
  - de *BABA* még nincs a szótárban

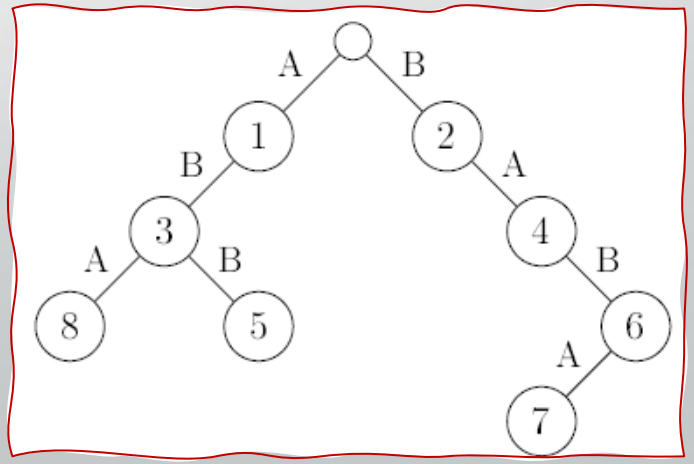
Bemenet:	*A*	*B*	*AB*	*BA*	*BAB*	*AB ABA A*
Kimenet:	1	2	3	4	6	





- 6. lépés:
  - *AB ABAA* szöveg
  - *AB* ~3 (szófa segítségével)
  - de *ABA* még nincs a szótárban

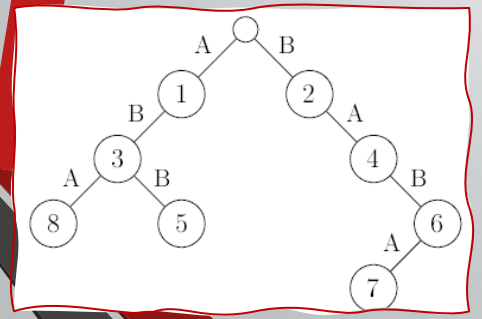
Bemenet:	*A*	*B*	*AB*	*BA*	*BAB*	*AB*	*ABA A*
Kimenet:	1	2	3	4	6	3	

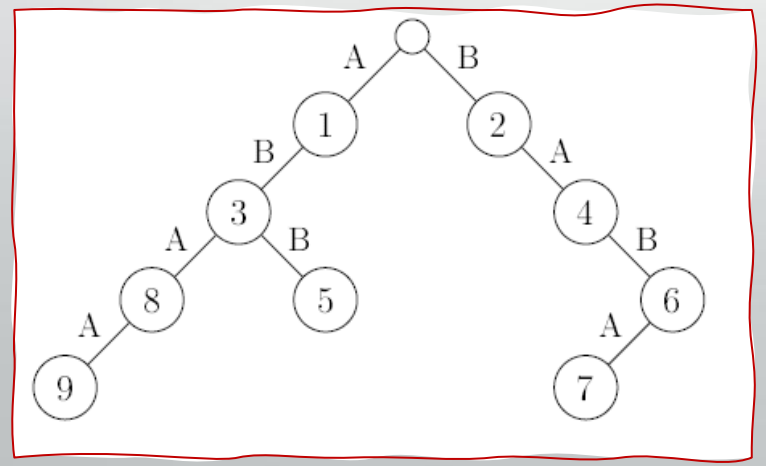


A tömörítés lépései

- 7. lépés:
  - *ABAA* szöveg
  - *ABA* ~8 (szófa segítségével)
  - de *ABAA* még nincs a szótárban

Bemenet:	A	B	AB	BA	BAB	AB	ABA	A
Kimenet:	1	2	3	4	6	3	8	





- 8. lépés:
  - *A* szöveg
  - *A* ~1 (szófa segítségével)
  - a maradék szöveg üres
  - ➢ *A* kódját kiírjuk, befejezzük a tömörítést

Bemenet:	A	B	AB	BA	BAB	AB	ABA	A
Kimenet:	1	2	3	4	6	3	8	1

A tömörített fájl mérete

- A legnagyobb kód: 9
  - Kódméret: 4 bit
    - 4 biten 15-ig ábrázolhatók
    - 3 biten csak 7-ig
  - Kimenet: 8*4=32 bit

+ Metaadatok
- Ábécére vonatkozó információ
  - (milyen karakterekből áll)
- A kódok fix hossza
  - (hány biten van tárolva)

Példa LZW dekódolásra

- Szótár kulcsai: kódok
- ➢ Szótár szerkezete:
  - Sztringek vektora
  - Kódokkal indexelve
- Kezdőállapot:

Bemenet:				1, 2, 3, 4, 6, 3, 8, 1
Kimenet:				
Szó-tár	kód	1	2	
	szó	*A*	*B*	

- Bemenet feldolgozása:
- Betömörítés
  - az *A*-t megtalálta a szótárban
  - de *AB*-t nem
- ➢ 1-es kódot írta ki és felvette a szótárba az *AB*-t 3-as kóddal

Bemenet:				1	2	3, 4, 6, 3, 8, 1
Kimenet:				*A*	*B*	
Szó-tár	kód	1	2	3		
	szó	*A*	*B*	AB		

Példa LZW dekódolásra

- Hasonlóan mehetünk tovább a 4-es kódhoz tartozó szó kiírásáig

Bemenet:		1	2	3	4	6, 3, 8, 1
Kimenet:		A	B	AB	BA	
Szó-tár	kód	1	2	3	4	5		
	szó	A	B	AB	BA	ABB		

- Probléma:
  - 6-os kód következik
  - De ez nem szerepel a szótárban
- Tudjuk, hogy a betömörítés
  - az *BA*-t megtalálta a szótárban
  - de *BAu*-t nem (*u*: unknown)
- A szótárban nem szerepló kód megfejtése
  - A kód az előző lépésben generálódott
  - a *BA*-ból generálódott

➢ Az *u* betű a *BA* kód első betűje: *B*

Bemenet:		1	2	3	4	6	3, 8, 1
Kimenet:		A	B	AB	BA	BAB	
Szó-tár	kód	1	2	3	4	5	6		
	szó	A	B	AB	BA	ABB	BAB		

Általánosságban:

A. A kitömörítés a *k* kódhoz tartozó szót megtalálja a szótárban

- A betömörítés a *k* kódhoz tartozó szót már korábban generálta

B. A kitömörítés a *k* kódhoz tartozó szót nem találja meg a szótárban

- A betömörítés a *k* kódhoz tartozó szót az előző lépésben generálta
- Az előző lépésben kiírt szó: *S*
  - A betömörítés előző lépésében generált *k* kódhoz tartozó szó *Su* alakú
  - Kitömörítés aktuális lépés idején, a *k* kódhoz tartozó szó: *Su* alakú
  - ➢ *u* betű éppen az *Su* szó 1. betűje
- ➢ *K* kódhoz tartozó szó: $S+S_1$ (S_1 az *S* szó 1. betűje)

Példa folytatása:

A.

Bemenet:		1	2	3	4	6	3	8, 1
Kimenet:		A	B	AB	BA	BAB	AB	

Szó-tár	kód	1	2	3	4	5	6	7		
	szó	A	B	AB	BA	ABB	BAB	$BABA$		

B.

Bemenet:		1	2	3	4	6	3	8	1
Kimenet:		A	B	AB	BA	BAB	AB	ABA	

Szó-tár	kód	1	2	3	4	5	6	7	8		
	szó	A	B	AB	BA	ABB	BAB	$BABA$	ABA		

A.

Bemenet:		1	2	3	4	6	3	8	1
Kimenet:		A	B	AB	BA	BAB	AB	ABA	A

Szó-tár	kód	1	2	3	4	5	6	7	8	9	
	szó	A	B	AB	BA	ABB	BAB	$BABA$	ABA	$ABAA$	

LZW szótár és maximális kódhossz

- A szótár a kitömörítés során is **ugyanannyi bejegyzést tartalmaz**, mint a betömörítésnél, azzal a különbséggel, hogy itt **a kódok a kulcsok**.
- Fix kódhossz esetén, ha a kódok ***b* bitesek**, akkor a legnagyobb használható kódérték:

$$MAXCODE = 2^b - 1$$

- Példa:
  - ha **b = 12**, akkor:

$$MAXCODE = 2^{12} - 1 = 4095$$

- Amikor a kódok elérik a **MAXCODE értéket**:
  - a szótár **nem bővül tovább**
  - a tömörítés és kitömörítés **ugyanazzal a rögzített szótárral folytatódik**

Ellenőrző kérdések: Huffman kód

1. Szemléltesse a Huffman kódolás működését az ÁBRÁBANÁBRA szövegen!
   - Adja meg a kódfát és a szótárat! Mekkora a Huffman kódolással tömörített kód hossza?
   - Mekkora lenne a tömörített kód x hosszú karakterkódok esetén?
   - Hogyan dekódolható egy Human kóddal tömörített szöveg?
   - Milyen értelemben optimális a Human kód? Azt jelenti-e ez, hogy a Huffman kódolás a lehető legjobb tömörítés? Miért?

Ellenőrző kérdések: Lempel-Ziv-Welch (LZW)

1. Adott az {A;B;C} ábécé. Szemléltesse az LZW tömörítő algoritmust
   - Tömörítő algoritmus működését a CBABABABCBABABAB szövegen
   - Majd a megfelelő kitömörítő algoritmusét az 1; 2; 3; 4; 6; 5; 9; 7; 11 kódon!
   - Mindkét esetben adja meg
     - A generált szótárat és
     - A tömörítetlen szövegen a részszavak és a kódok megfeleltetését!
   - Hogyan kezeli a kitömörítő algoritmus azt az esetet, amikor nem találja meg a szótárban az aktuális kódhoz tartozó sztringet?

Köszönöm a figyelmet!

Pusztai Kinga

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