

MéRNÖKI modellalkotás

Az elmélettől a gyakorlatig

**Prefix fák tömörítése:
a dinamikus programozás**

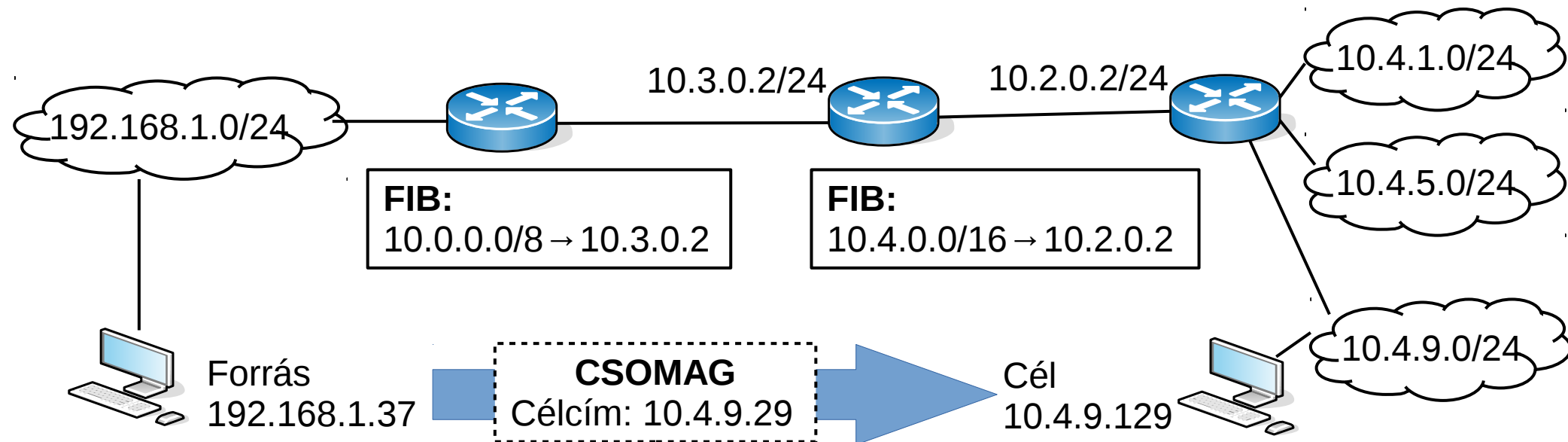
Tartalom

- Ismétlés:
 - IP forgalomtovábbítás és LPM
 - prefix fák és fabejárások
 - normalizálás: a minimális prefix-mentes forma
- FIB aggregáció:
 - FIBek leírása minimális számú prefixszel
 - Szint-tömörítés

Ismétlés

IPv4 forgalomtovábbítás

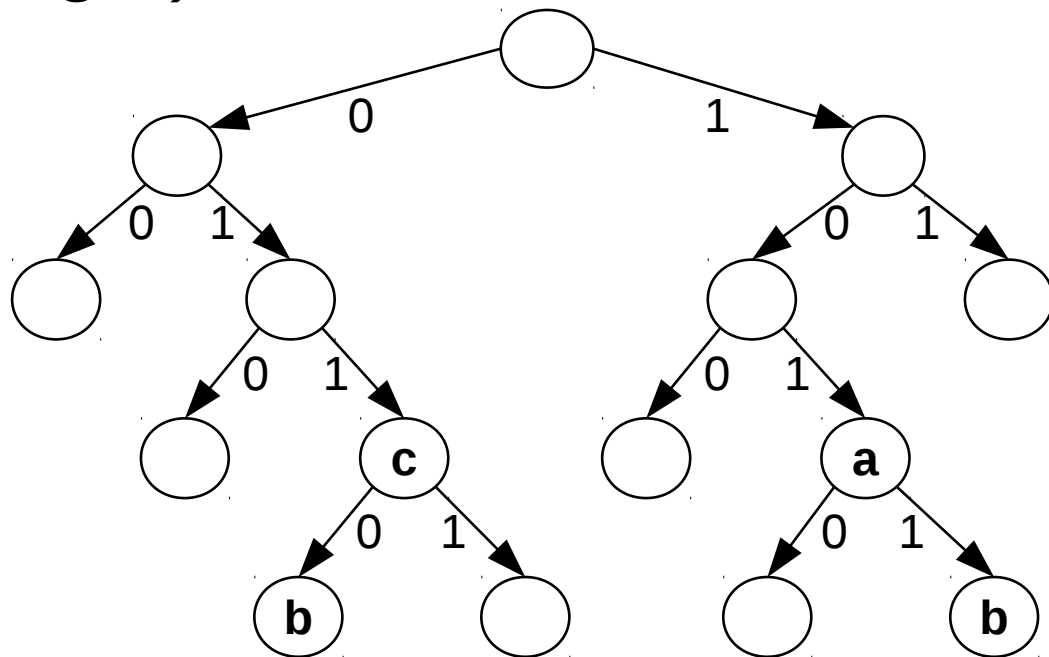
- Minden csomag tartalmazza a cél IP címét
- Keresés a forgalomtovábbítási táblában (FIB)
- Eredmény: az útvonalon következő router (next-hop) IP címe



A legspecifikusabb prefix

- **Longest Prefix Match (LPM):** ha egy IP címre több bejegyzés illeszkedik, akkor a legtöbb biten (leghosszabb prefixen) illeszkedő a preferált
- Táblázat: LPM komplexitása $O(n)$, n bejegyzésre
- **Bináris prefix fa:** $O(\log n)$ futási időben LPM

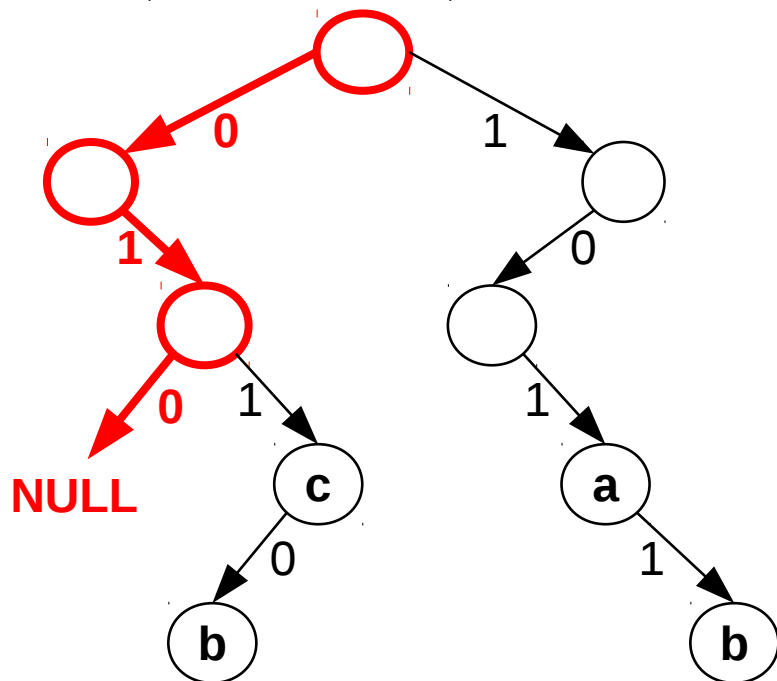
IP prefix	Prefix	NH
160.0.0.0/3	101	a
96.0.0.0/4	0110	b
96.0.0.0/3	011	c
176.0.0.0/4	1011	b



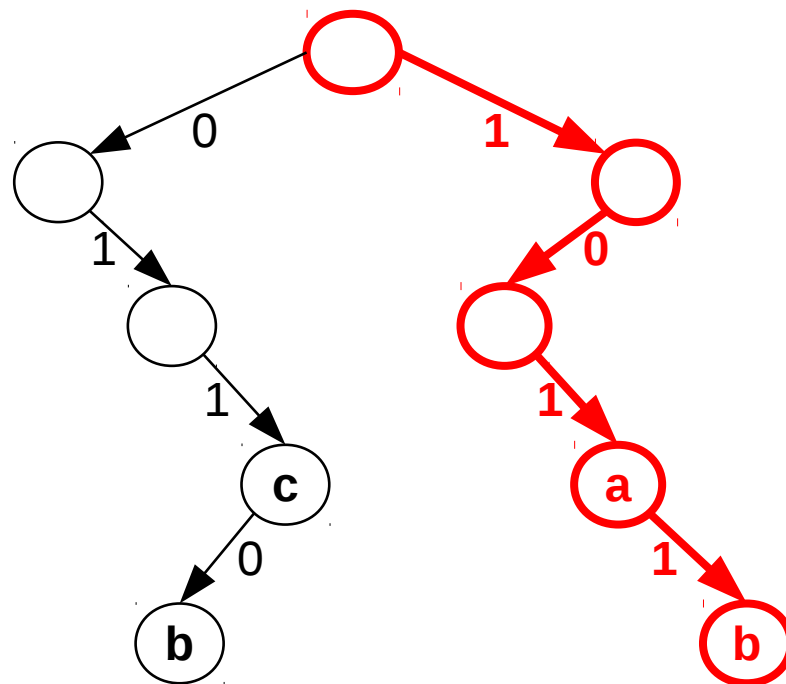
A prefix fa: keresés

- A keresett IP cím következő bitjének megfelelően a 0 vagy 1 élcímkejű élen lépünk tovább
- Tároljuk a legutoljára olvasott címkét

LPM(69.12.75.54) =
LPM(01000...) → NULL



LPM(178.4.66.19) =
LPM(1011...) → b



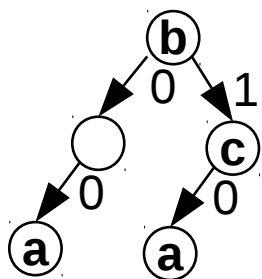
Prefix fák ekvivalenciája

- FIBek leírása nem egyedi: **FIB aggregáció**

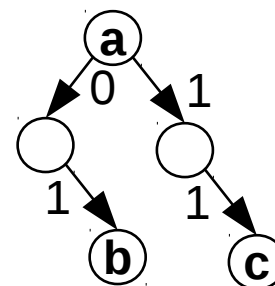
IP prefix	Prefix	Címke
0.0.0.0/0	-	b
128.0.0.0/1	1	c
0.0.0.0/2	00	a
128.0.0.0/2	10	a

≡

IP prefix	Prefix	Címke
0.0.0.0/0	-	a
64.0.0.0/1	01	b
192.0.0.0/2	11	c



≡



- Két FIB ekvivalens, ha minden 32 bites α IPv4 címre az LPM eredménye megegyezik

$$FIB_1 \equiv FIB_2, \text{ ha } \forall \alpha: LPM(FIB_1, \alpha) = LPM(FIB_2, \alpha)$$

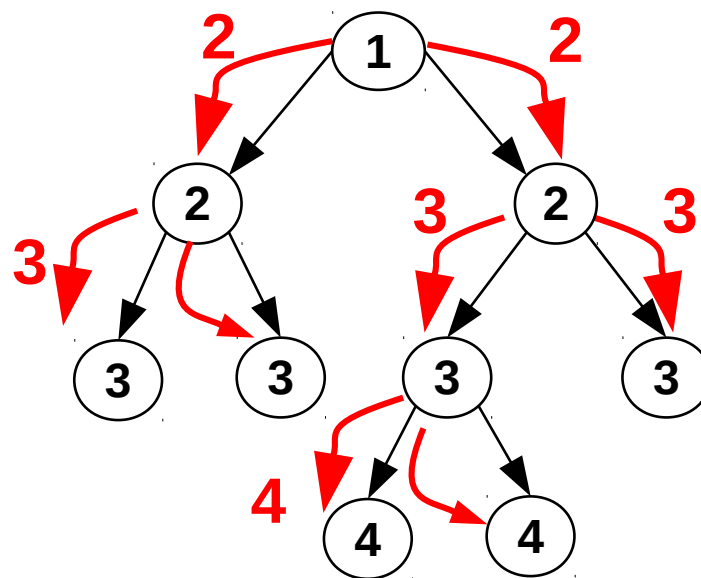
Fabejárások: preorder

- $preorder(F, f, i)$: alkalmazzuk f -et F gyökerére majd a bal és jobb oldali részfákra rekurzívan

```
preorder(F, f, i) :  
  x ← f(F, i)  
  preorder(left(F), f, x)  
  preorder(right(F), f, x)
```

- Írjuk be minden pontba a gyökértől vett távolságot:
 $preorder(F, f, 1)$

```
f(F, i) :  
  label(F) ← i  
  return (i+1)
```



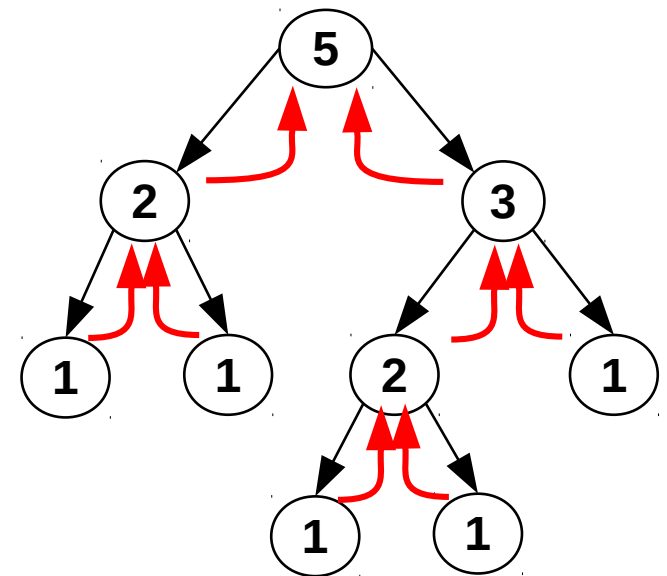
Fabejárások: postorder

- $postorder(F, f)$: f függvényt előbb alkalmazzuk a részfákon rekurzívan és csak ezután a gyökéren

```
postorder(F, f) :  
  postorder(left(F), f)  
  postorder(right(F), f)  
  return f(F)
```

- Írjuk be minden pontba a részfa **leveleinek** számát

```
f(F) :  
  if (F leaf) : label(F) ← 1  
  else :  
    label(F) ← label(left(F)) +  
      label(right(F))
```



Egy hasznos fa-transzformáció

- FIB ekvivalens transzformációja egyedi alakba
- Egymás utáni preorder és postorder bejárással

1. $preorder(F, f, d_0)$

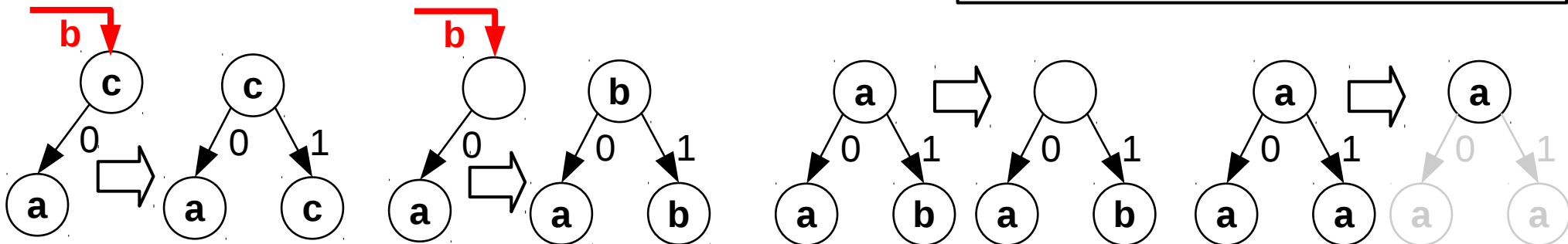
```

f(F, i):
  if F is interior:
    if  $\exists$  left(F): add_node(left(F))
    if  $\exists$  right(F): add_node(right(F))
  if (label(F) ==  $\emptyset$ ):
    label(F)  $\leftarrow$  i
  return label(F)
  
```

2. $postorder(F, g)$

```

g(F, i):
  if F is leaf: return
  if label(left(F)) ==
    label(right(F)) ==
    label(F):
    remove_node(left(F))
    remove_node(right(F))
  return
  label(F) =  $\emptyset$ 
  
```

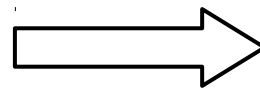


Normalizálás

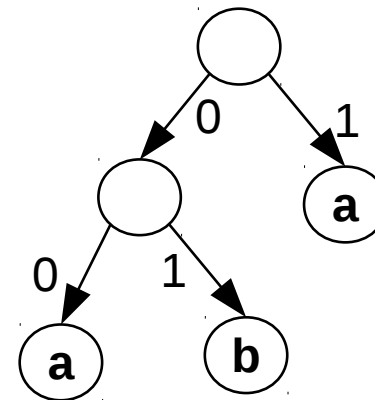
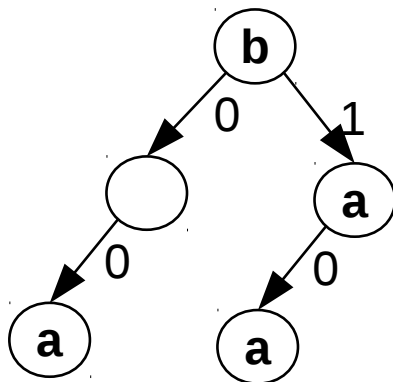
- d_0 : az első preorder-t kezdeti címkével indítjuk
- Ez kerül a címkézetlen pontokra: **default gateway!**

IP prefix	Prefix	Címke
0.0.0.0/0	-	b
128.0.0.0/1	1	a
0.0.0.0/2	00	a
128.0.0.0/2	10	a

normalizálás



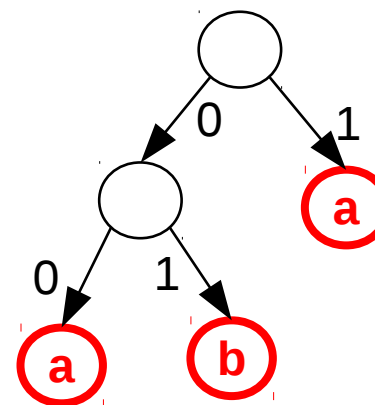
IP prefix	Prefix	Címke
0.0.0.0/2	00	a
64.0.0.0/2	01	b
128.0.0.0/1	1	a



Normalizálás: tulajdonságok

- A normalizált FIB táblázatos formában **prefix-mentes**: egyik bináris kulcs sem prefixe a másiknak
- Nincsenek kevésbé specifikus bejegyzések!

IP prefix	Prefix	Címke
0.0.0.0/2	00	a
64.0.0.0/2	01	b
128.0.0.0/1	1	a



- A prefix fa **levél-címkézett** (címke csak levélen) és **szabályos** (nincs benne „NULL pointer”)
- A továbbiakban a normalizált alakot használjuk

FIB aggregáció:

**FIB leírása a minimális számú
prefixszel**

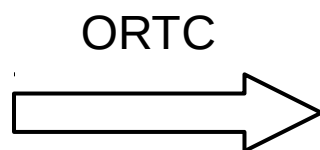
FIB aggregáció

- Egy IP router minden továbbítandó csomagra LPM keresést végez
- Kb. 500 ezer prefixen több millió LPM/sec
- A routerek kezdenek kifogyni a memóriából, ahogy az Internet növekszik
- **FIB aggregáció:** FIB ekvivalens transzformációja memóriatakarékos formába
- Régebbi routerek is használhatók maradnak
- Befér a FIB „gyors” memóriába (~ cache)

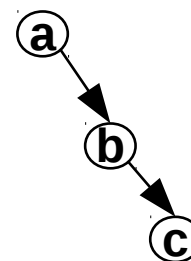
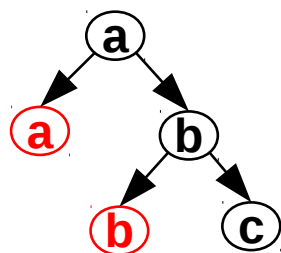
FIB „optimális” tömörítése

- Láthattuk, hogy egy FIB leírása nem egyedi
- Egyes bejegyzések feleslegesek lehetnek

IP prefix	Prefix	Címke
0.0.0.0/0	-	a
0.0.0.0/1	0	a
128.0.0.0/1	1	b
128.0.0.0/2	10	b
192.0.0.0/2	11	c



IP prefix	Prefix	Címke
0.0.0.0/0	-	a
128.0.0.0/1	1	b
192.0.0.0/2	11	c



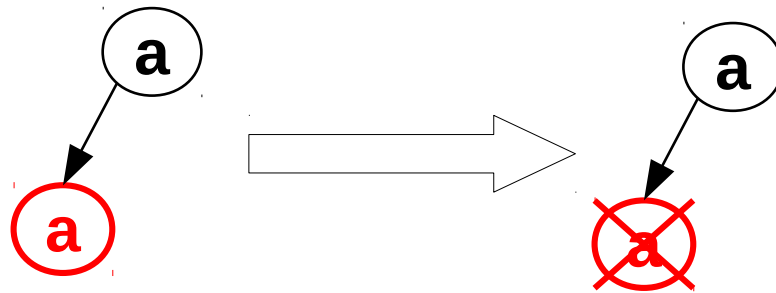
- **ORTC** (Optimal Routing Table Compression):
bejegyzések számának minimalizálása FIBben

ORTC: alapgondolat

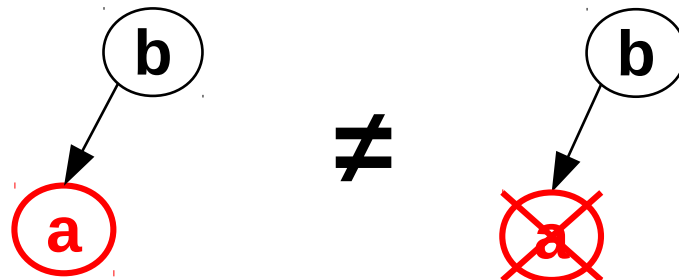
- Bejegyzések száma a FIBben = a prefix fában levő címkézett pontok száma
- Az ORTC célja olyan prefix fa előállítása, amelyben a lehető legkevesebb címke van
- **Ötlet:** ha egy pont a szülőjétől a címkéjével azonos címkét „örököl”, akkor nem kell bele címkét írunk
- Mivel az LPM eredménye a bejárás során utoljára látott címke, jó címkére „emlékszünk”
- Helyes választással megspórolható egy címke!

ORTC: alapgondolat

- Elhagyható a bejegyzés például egy levélben, ha a szülőtől azonos címkét „örököl”



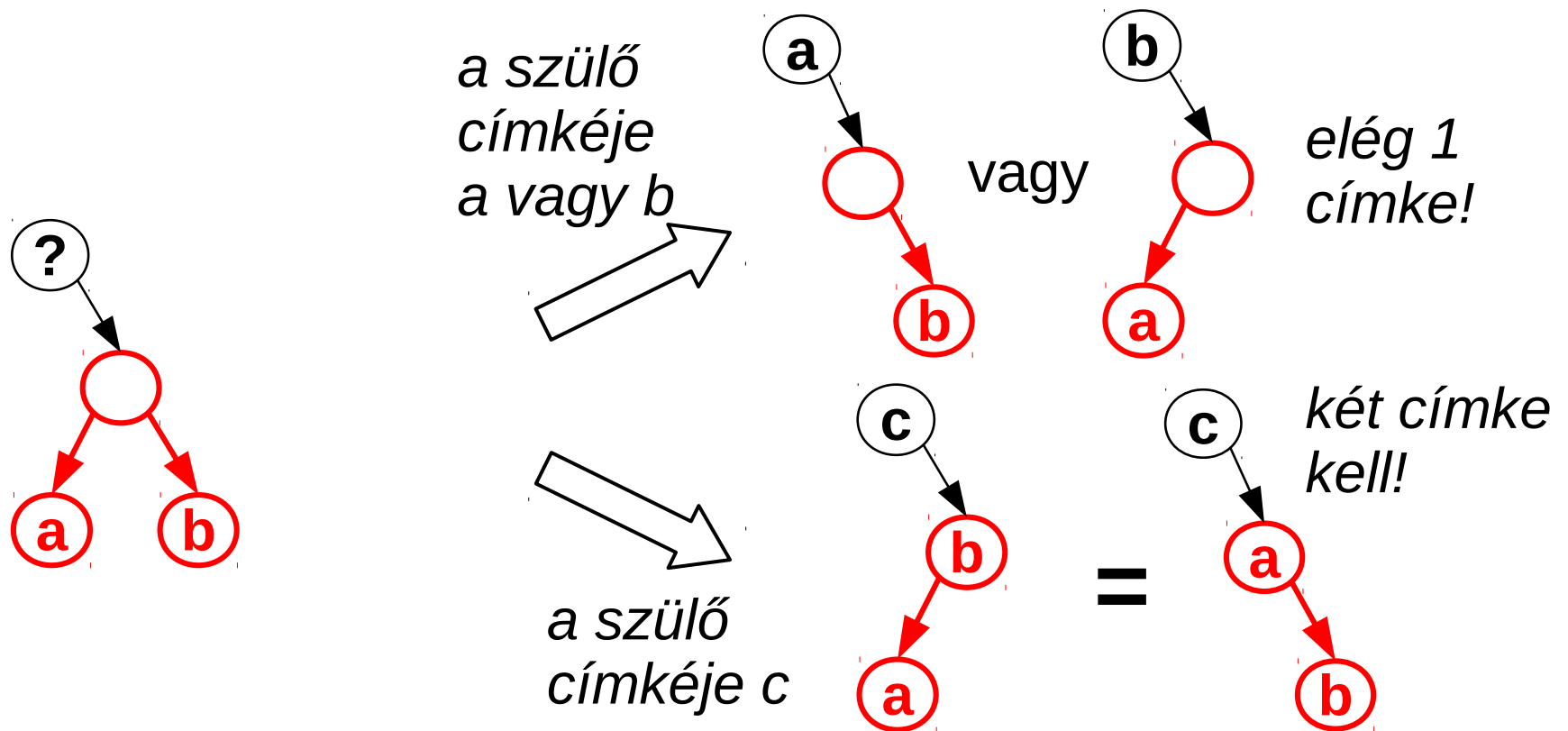
- Ha azonban a szülő más címkét tartalmaz, akkor ki kell írunk a levélbe a címkéjét



- Különben hibás csomagtovábbítás (a helyett b)

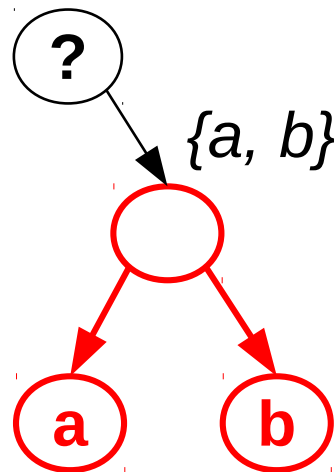
ORTC: alapgondolat

- Hasonlóan teljes részfákra: elhagyható a címke ha a szülőtől megfelelő címkét „örököl”
- Ha azonban a szülő más címkét választ, akkor ki kell írunk a pontba a címkéjét!



ORTC: alapgondolat

- Minden részfára definiálható azon címkék halmaza, amelyeket a részfa szülőjébe írva a részfa „megspórol” egy címkét

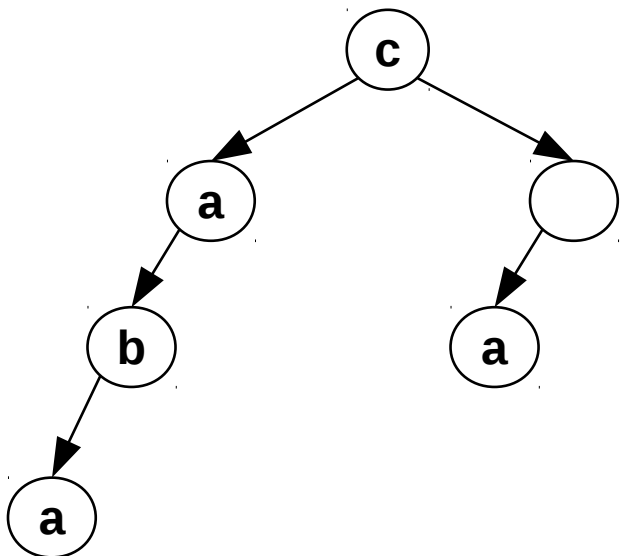


- Levélre triviális, belső pontra a részfái alapján
- A levelektől haladunk felfelé: postorder a normalizált alakról

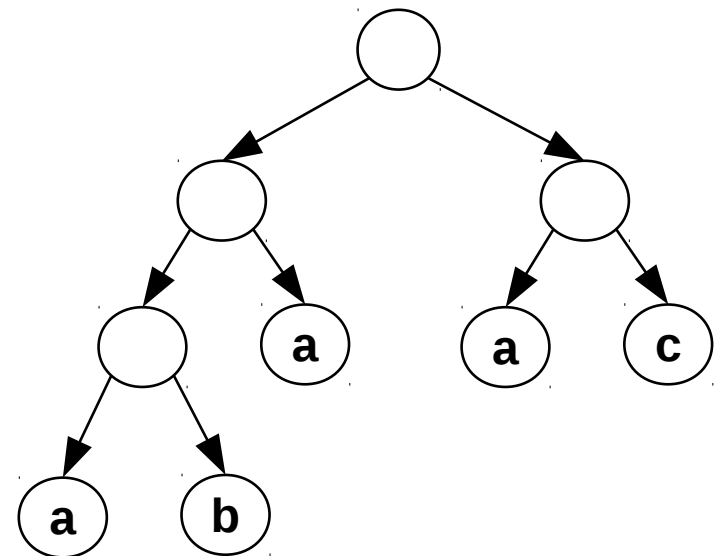
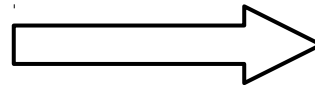
ORTC

- Induljunk a FIB normalizált alakjából

IP prefix	Prefix	Címke
0.0.0.0/0	-	c
0.0.0.0/1	0	a
0.0.0.0/2	00	b
128.0.0.0/2	10	a
0.0.0.0/3	000	a

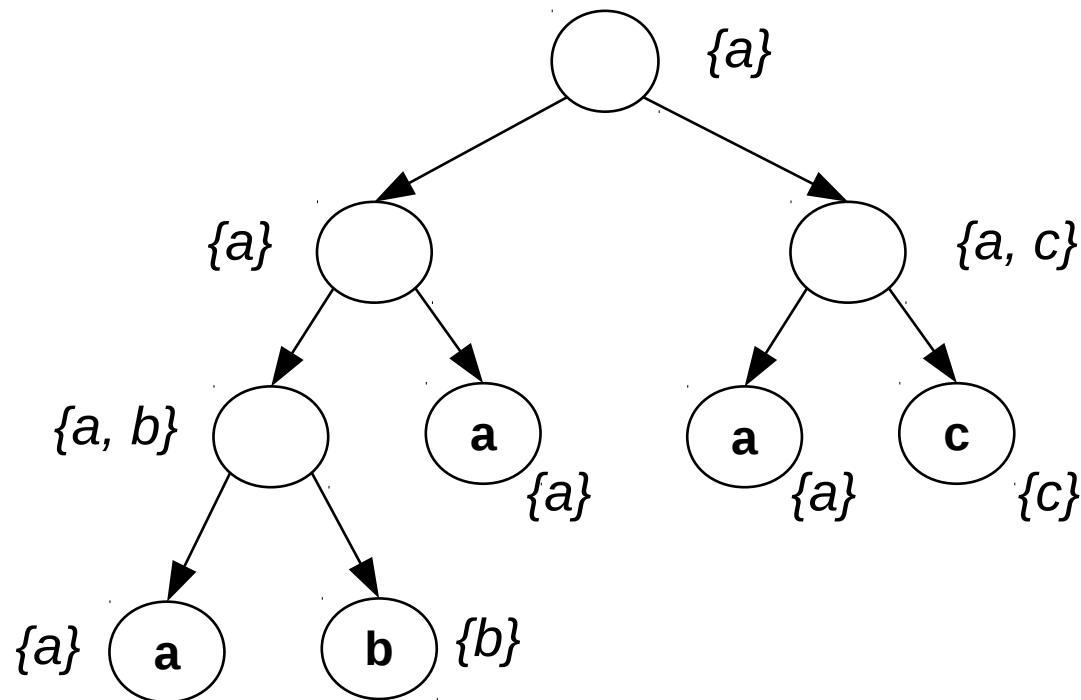


normalizálás



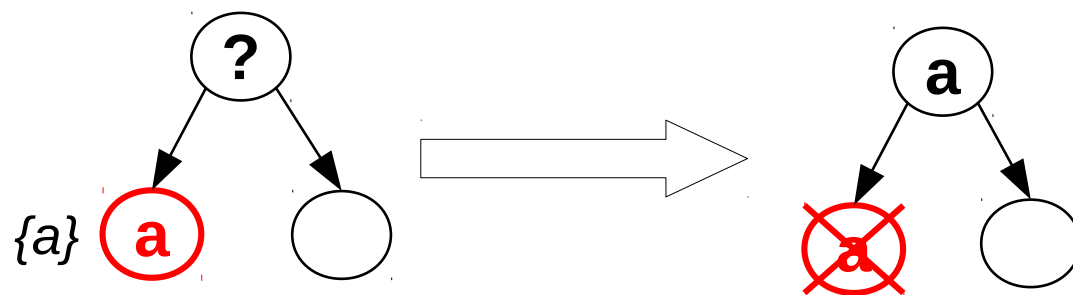
ORTC

- Jelölje D_F az F (rész)fa számára „preferált” címkék halmazát
- Ha F valamely D_F -beli címkét örököl a szülőtől, akkor F gyökérpontjába nem kell címkét írunk



Preferált címkék: levélpontok

- Ha F levélpont, akkor $D_F = \{ label(F) \}$



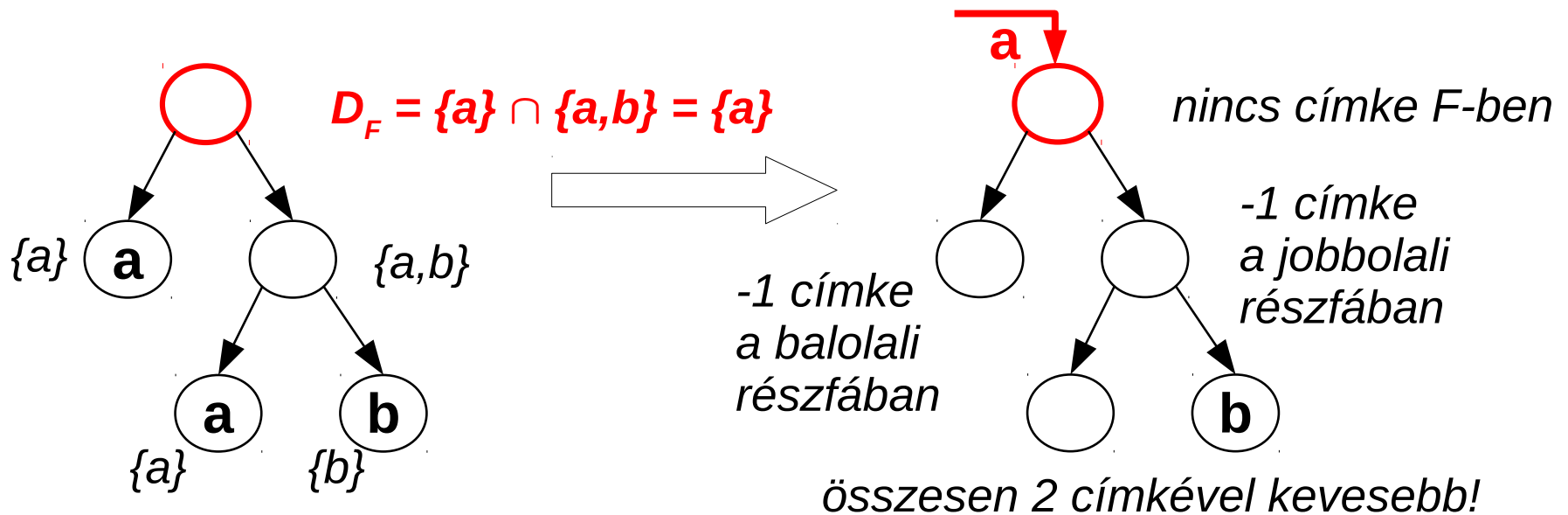
- Emlékeztető: a levélpontban mindig van címke, mert a normalizált alakból indultunk
- Hasonlóan: belső pontoknak mindig lesz két gyermeke a normalizálás miatt

Preferált címkék: belső pontok

- Ha F belső pont, két eset egyike lehetséges

1) Ha a gyermekeihez tartozó $D_{left(F)}$ illetve $D_{right(F)}$ halmazban van azonos címke, akkor ezt írva F -be a két részfában spórolunk egy-egy címkét

$$\text{ha } D_{left(F)} \cap D_{right(F)} \neq \emptyset, \text{ akkor } D_F = D_{left(F)} \cap D_{right(F)}$$



Preferált címkék: belső pontok

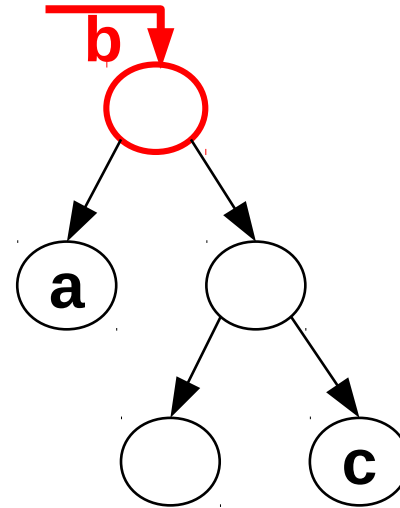
2) Ha azonban a $D_{left(F)}$ és $D_{right(F)}$ halmazokban nincs közös címke

- Ha F pont $D_{left(F)}$ -beli címkét örököl a szülőjétől, akkor a jobboldali részében lesz egy címkével kevesebb de a baloldaliban nem
- Ha $D_{right(F)}$ -beli címkét örököl, vice versa
- Ha egyiket sem, akkor nem tudunk spórolni
- Tehát F -ben $D_{left(F)}$ és $D_{right(F)}$ uniója preferált

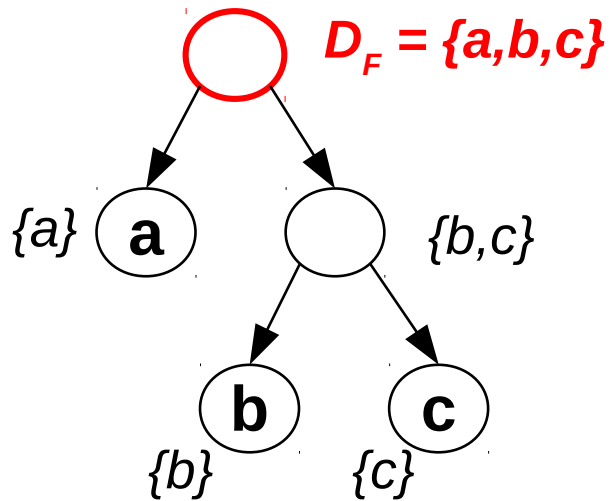
ha $D_{left(F)} \cap D_{right(F)} = \emptyset$, akkor $D_F = D_{left(F)} \cup D_{right(F)}$

Preferált címkék: belső pontok

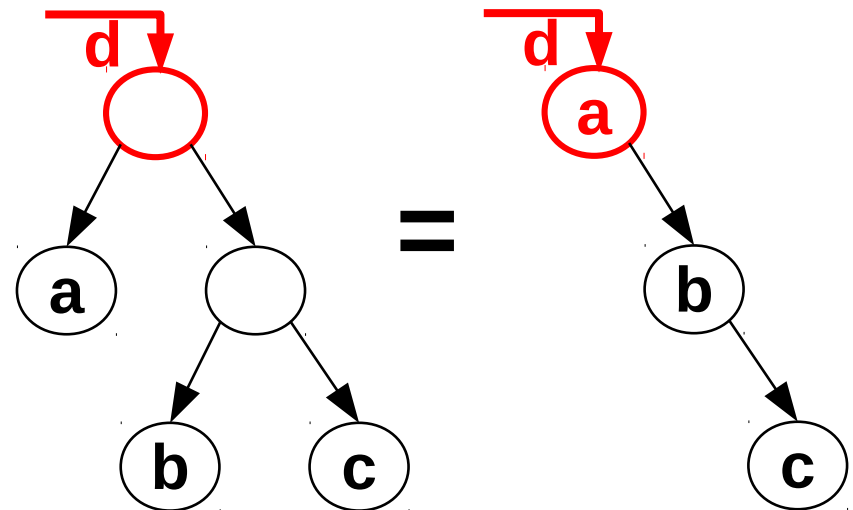
D_F -beli
címkét
örökölve,
például b -t



ha jó címkét kapunk, -1 címke



nem D_F -beli
címkét
örökölve,
például d -t



ha nem jó címkét kapunk, nem tudunk spórolni

ORTC algoritmus

- Input: normalizált prefix fa
- 1) Postorder bejárással **előállítjuk az egyes részfákra az D_F halmazokat**

postorder(F, f)

f(F) :

if F is leaf: $D_F \leftarrow label(F)$

else:

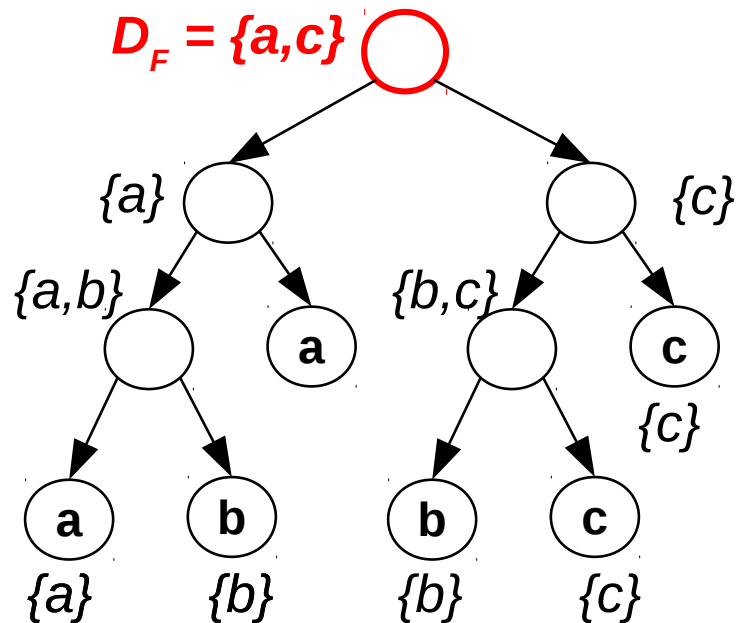
if $D_{left(F)} \cap D_{right(F)} \neq \emptyset$: $D_F \leftarrow D_{left(F)} \cap D_{right(F)}$

else: $D_F \leftarrow D_{left(F)} \cup D_{right(F)}$

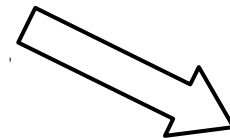
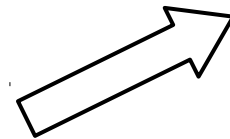
ORTC algoritmus

2) Előállítjuk az optimális d_0 címkét a gyökérben

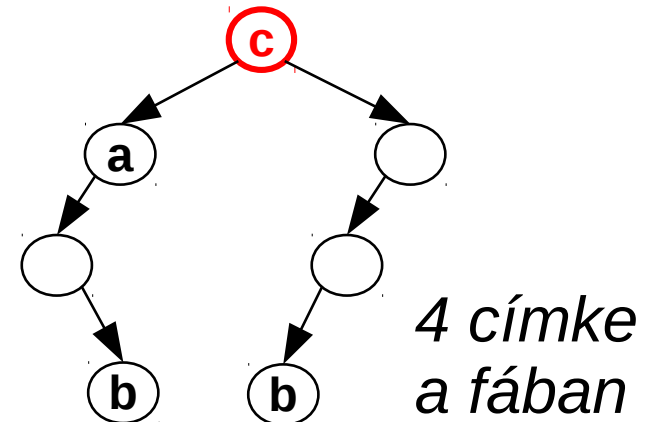
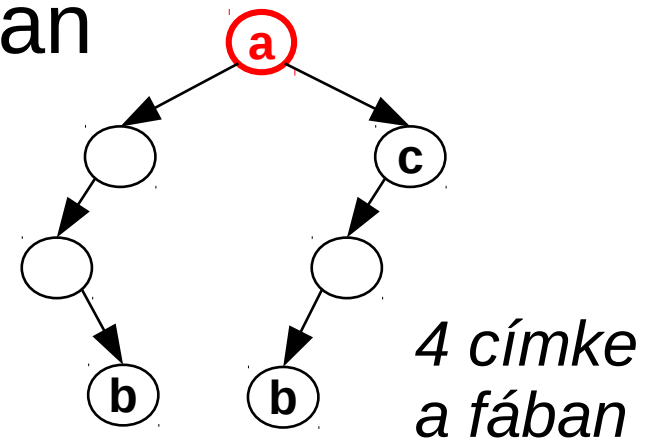
- bármely $d_0 \in D$ címkét választhatjuk, azonos számú címke lesz a kapott fában



*a gyökér
címkéje 'a'*



*a gyökér
címkéje 'c'*



ORTC algoritmus

3) Preorder bejárással **előállítjuk az optimális prefix fát**

- A gyökérbe kiírjuk az imént talált d_0 címkét
preorder(F, g, d₀), kezdeti érték: d_0

```
f(F, d) :
```

```
  if d ∈ DF :           // a szülő jó címkét választ
```

```
    label(F) ← ∅       // üres címke
```

```
    return d
```

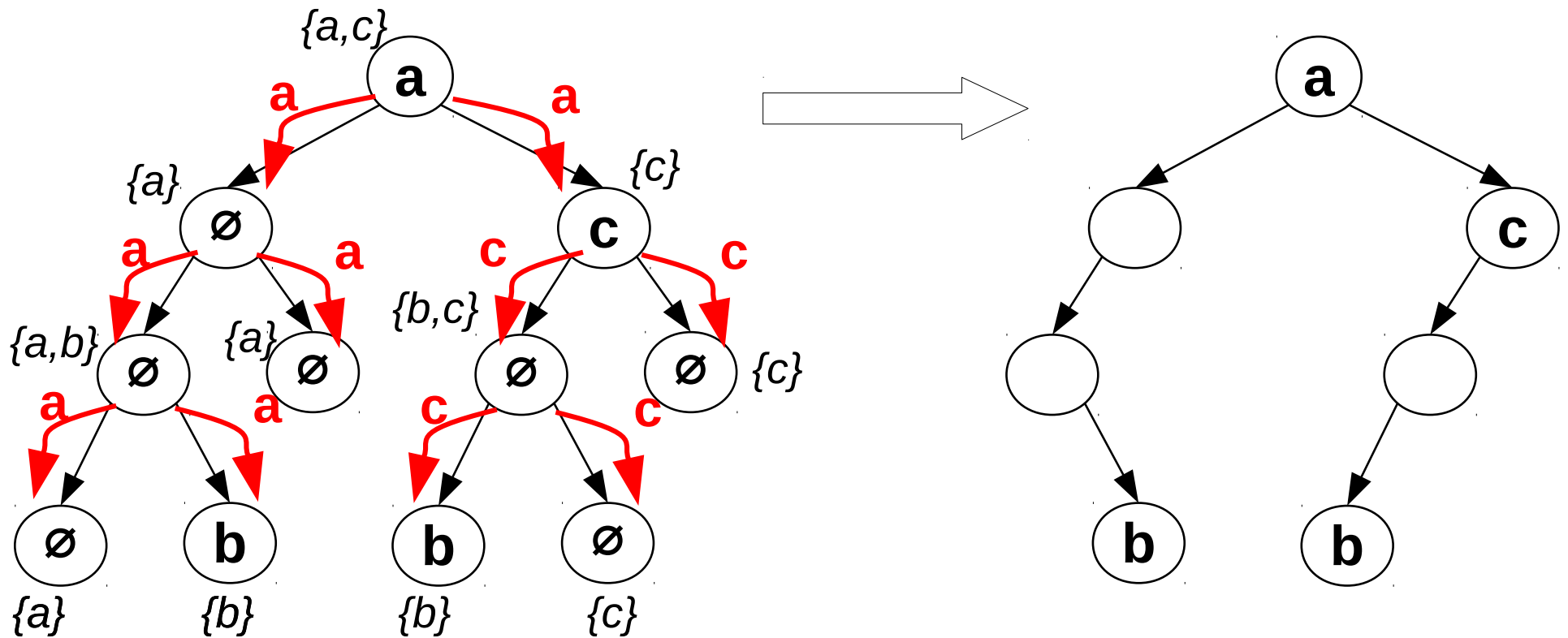
```
  else :                 // a szülő „rossz” címkén
```

```
    label(F) ← choose any d ∈ DF
```

```
    return label(F)
```

ORTC algoritmus

- A végén az üres leveleket eltávolíthatjuk

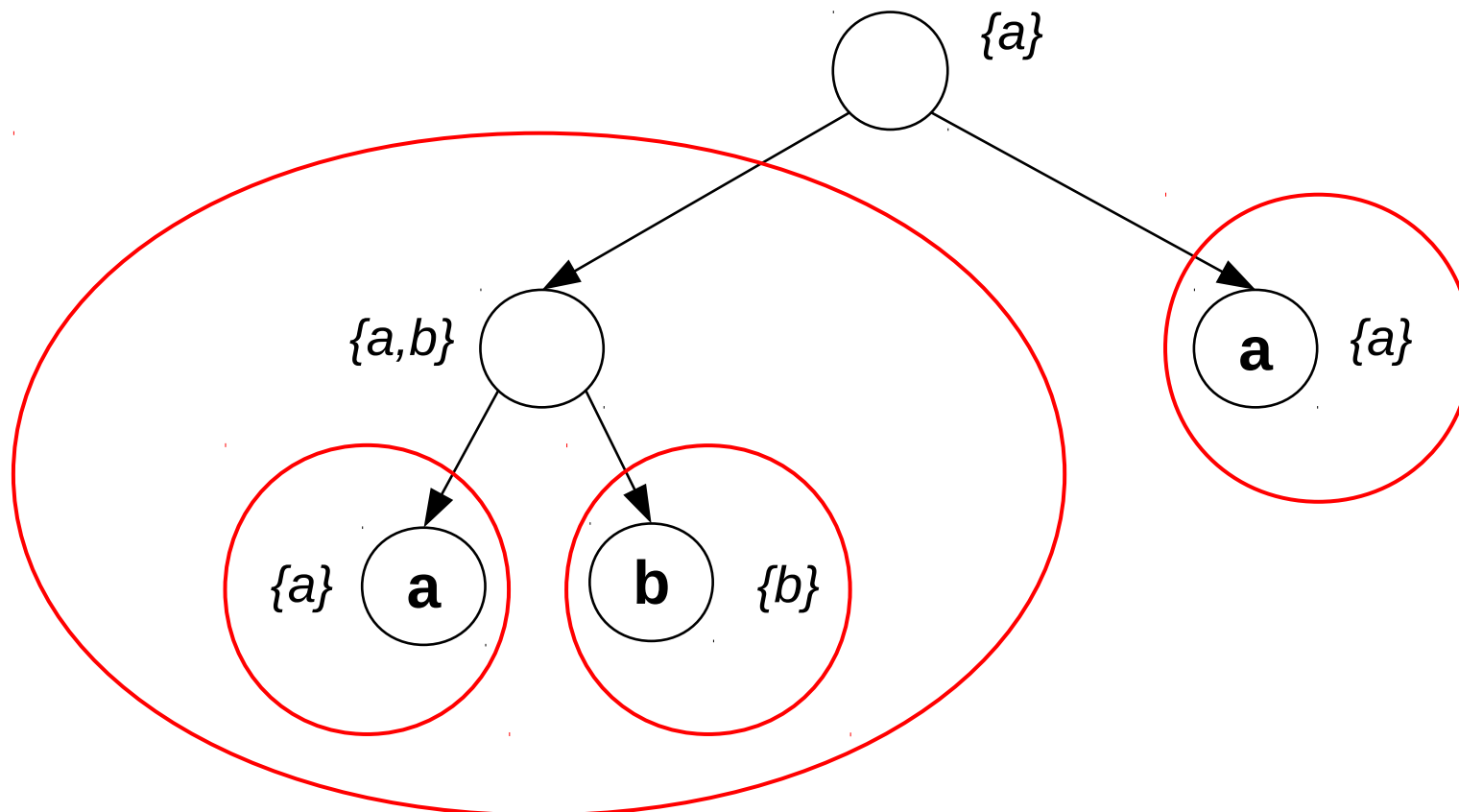


Dinamikus programozás

- Az ORTC algoritmus klasszikus stratégiát követ:
 - a problémát felosztjuk egymásba ágyazódó részproblémákra
 - a „legsűkebb” részproblémától indulva rekurzívan felírjuk a megoldást
- Most az „egymásba ágyazódó részproblémákat” a részfák definiálják
- A „legsűkebb” részprobléma az egyszerű levél
- Egy belső pontot mindig az alatta levő részfák megoldása ismeretében oldunk meg

Dinamikus programozás

- Ez az általános problémamegoldási stratégia a dinamikus programozás („*divide and conquer*”)



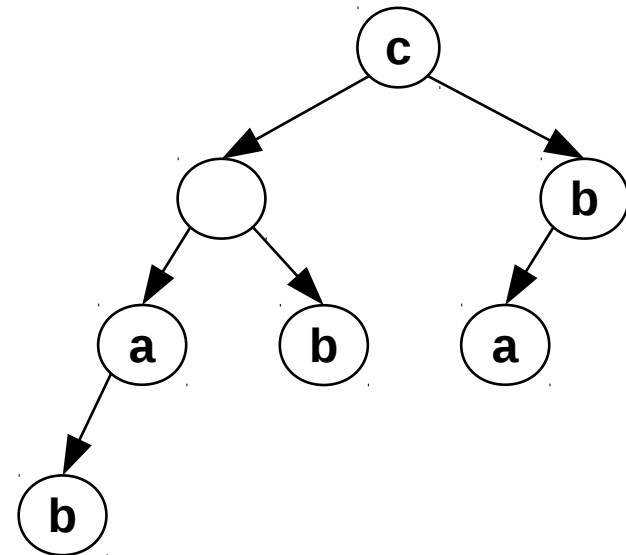
Dinamikus programozás

- Az ORTC algoritmus ekvivalensen felírható **dinamikus program** formájában is
- A „művészet” a részproblémák definiálása
 - fákon végzett optimalizálás esetén triviális
 - általános esetben a probléma struktúrájából
- Hatékony általános optimalizálási stratégia!

ORTC: példa

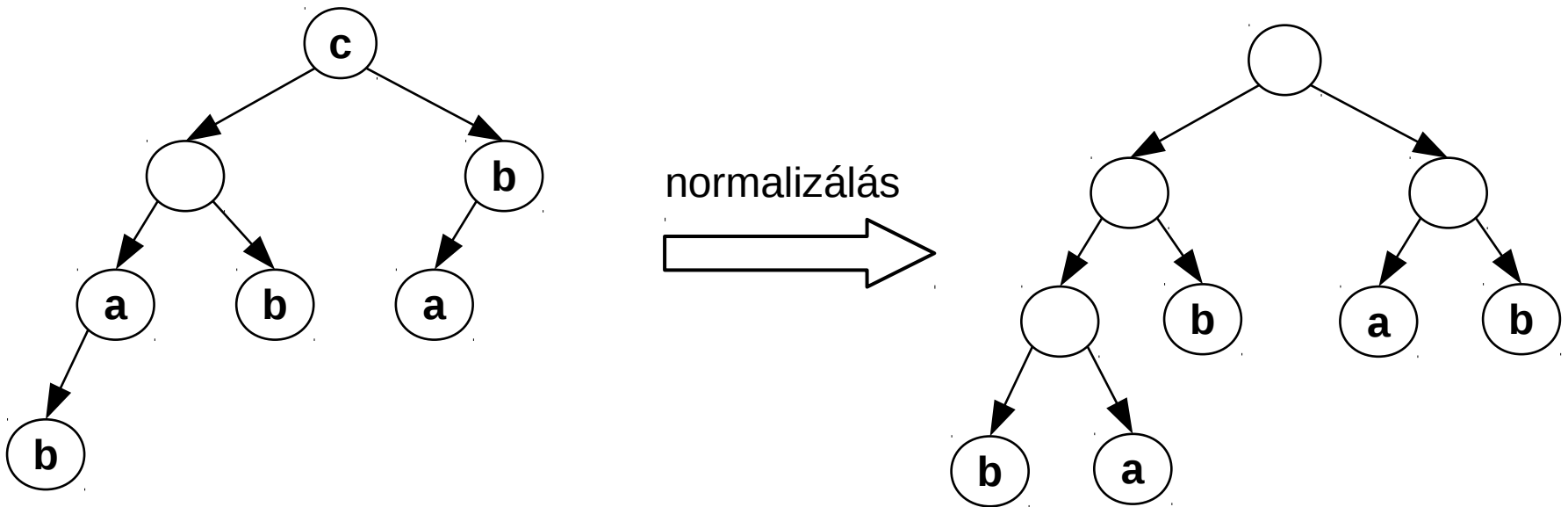
1) Prefix fa felírása

IP prefix	Prefix	Címke
0.0.0.0/0	-	c
128.0.0.0/1	1	b
0.0.0.0/2	00	a
64.0.0.0/2	01	b
128.0.0.0/2	10	a
0.0.0.0/3	000	b



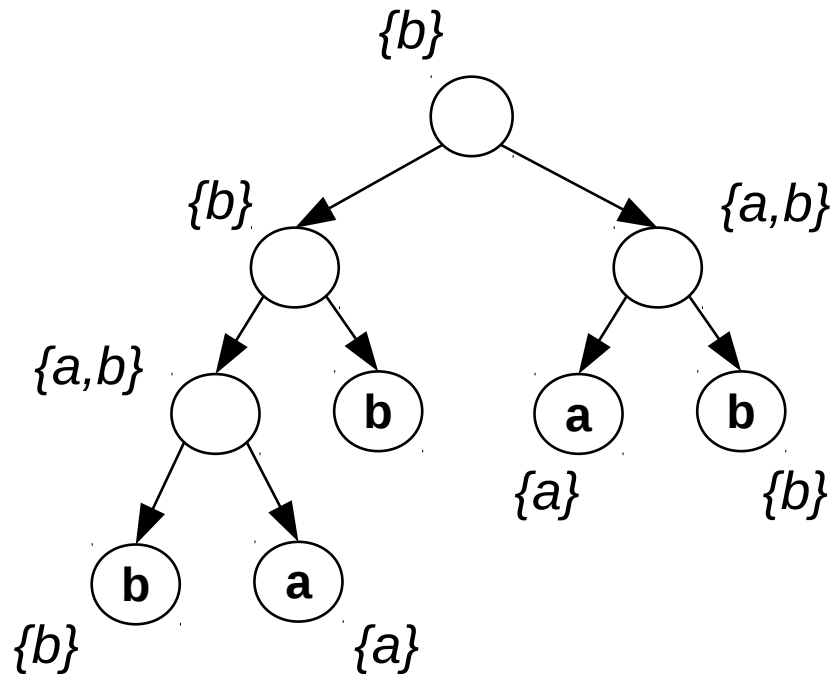
ORTC: példa

2) Normalizálás



ORTC: példa

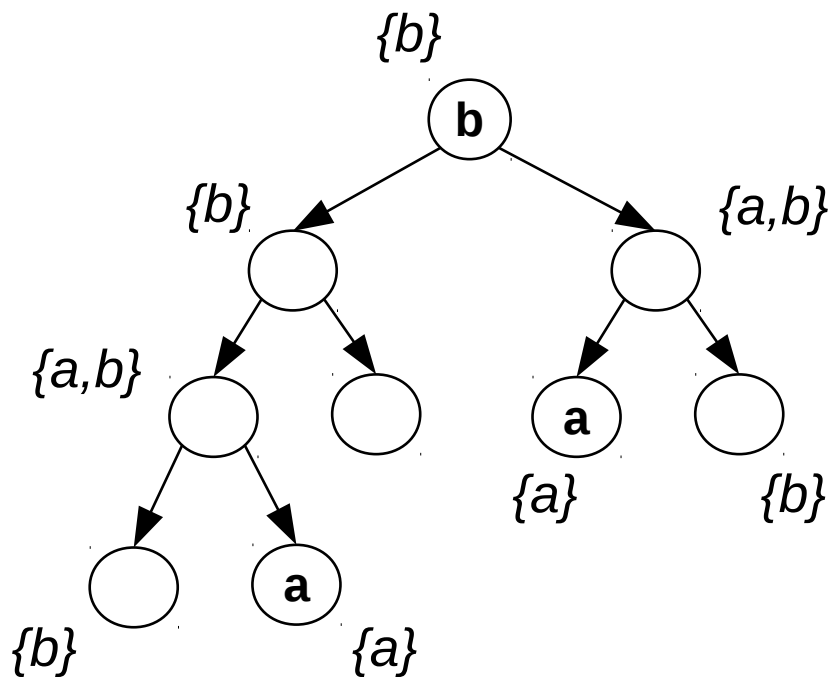
3) Preferált címkék keresése



ORTC: példa

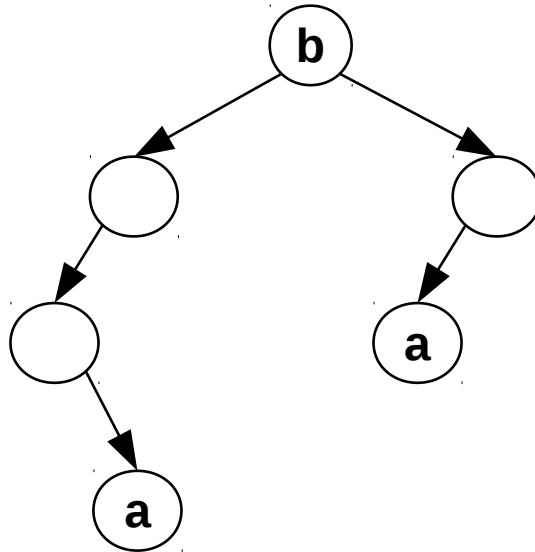
4) Gyökérpont címkéjének megválasztása

5) Módosított prefix fa felírása



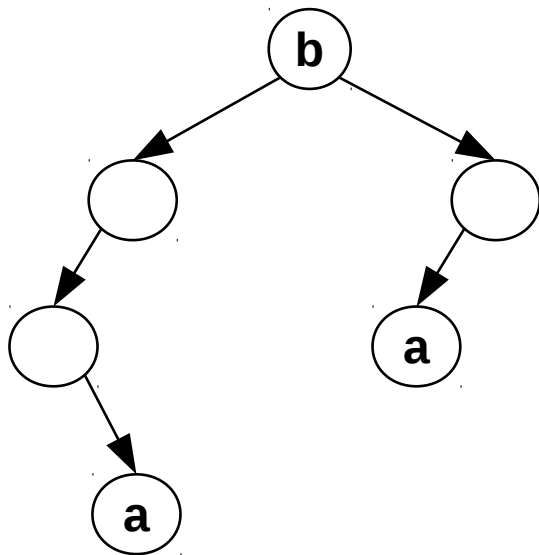
ORTC: példa

6) Üres levélpontok eltávolítása



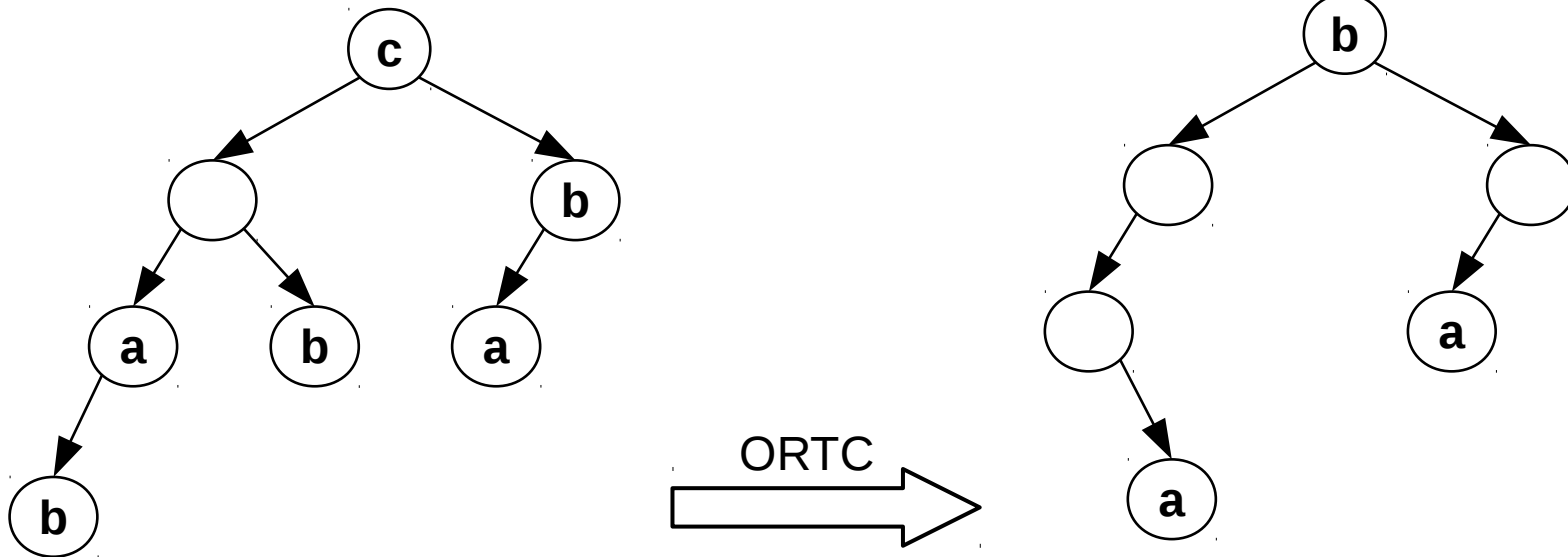
ORTC: példa

7) FIB felírása (6 bejegyzés helyett elég 3, ráadásul a c next-hop címke teljesen eltűnik)



IP prefix	Prefix	Címke
0.0.0.0/0	-	b
128.0.0.0/2	10	a
32.0.0.0/3	001	a

ORTC: példa



IP prefix	Prefix	Címke
0.0.0.0/0	-	c
128.0.0.0/1	1	b
0.0.0.0/2	00	a
64.0.0.0/2	01	b
128.0.0.0/2	10	a
0.0.0.0/3	000	b

IP prefix	Prefix	Címke
0.0.0.0/0	-	b
128.0.0.0/2	10	a
32.0.0.0/3	001	a

ORTC: összefoglaló

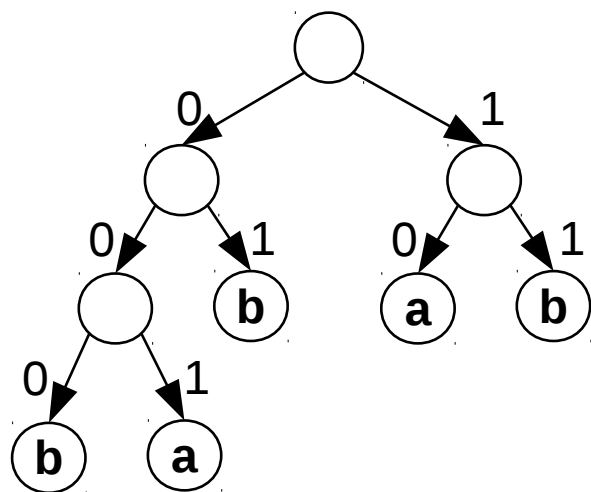
- Általában 30–40%-kal kisebb prefix fa
- A FIB táblázatos formája még kisebb lehet: TCAM-ek használatakor kulcsfontosságú!
- A módosított FIB teljesen ekvivalens az eredetivel (minden csomagot pont ugyanarra a next-hopra küld) és az LPM keresés is azonos
- ORTC futási idő: 3 fabejárás (optimalizálható kettőre) – lineáris idejű algoritmus
- De frissítés/módosítás nehéz – újra kell építeni az egész fát: $O(N)$ az optimális $O(\log N)$ helyett

FIB aggregáció:

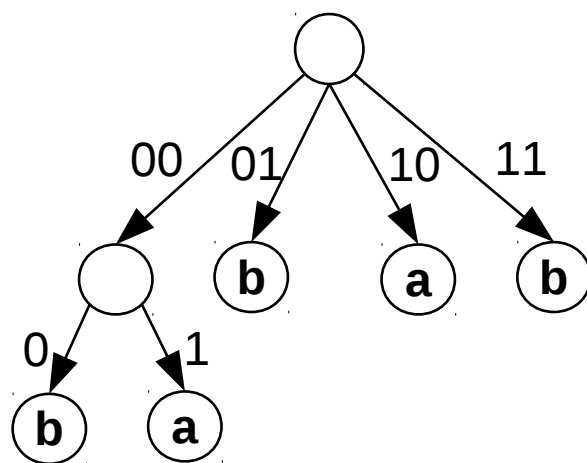
Prefix fák szinttömörítése

Szinttömörített prefix fák

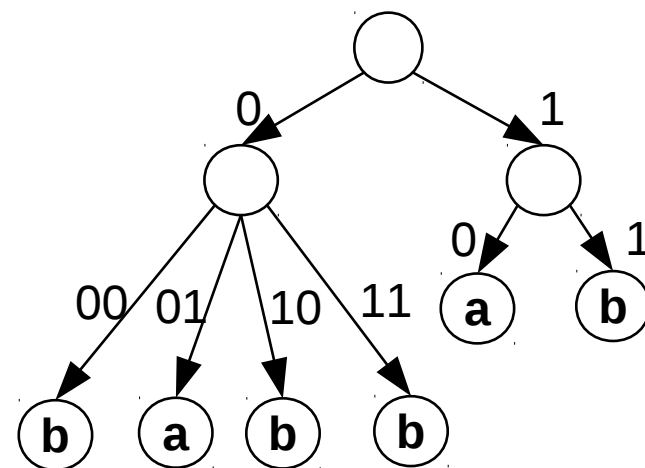
- Eddig bináris prefix fákkal dolgoztunk: minden belső pontnak 2 gyermeke van a fában
- **Szinttömörített fa:** minden belső pontnak 2^k gyermeke van valamely $k > 0$ egész számra



Bináris prefix fa
(normalizált)



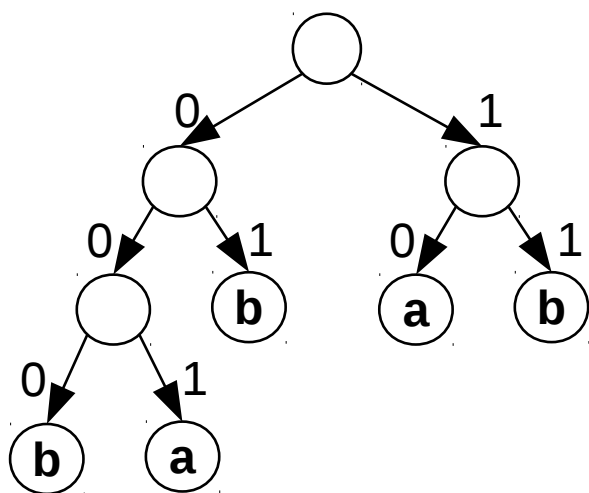
Szinttömötített prefix fa
(normalizált)



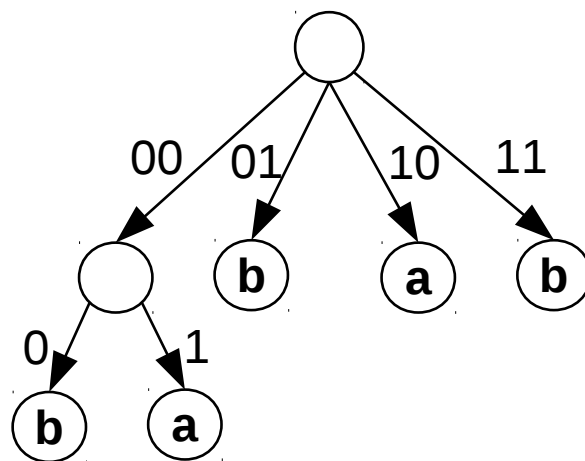
Alternatív
szinttömötített prefix fa
(normalizált)

Szinttömörített prefix fák

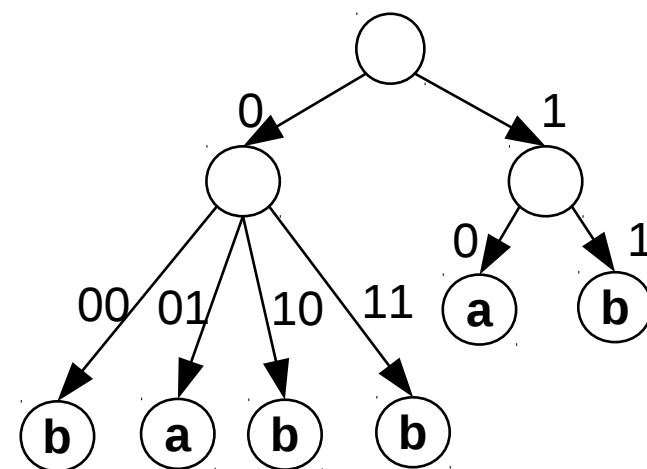
- LPM alapvetően ugyanaz, mint a bináris fán, de ha egy pontnak 2^k gyermeke van, akkor egyszerre k bitet olvasunk a kulcsból (IP cím)
- $2=2^1$ gyermek esetén 1 bitet, $4=2^2$ esetén kettőt



Bináris prefix fa
(normalizált)



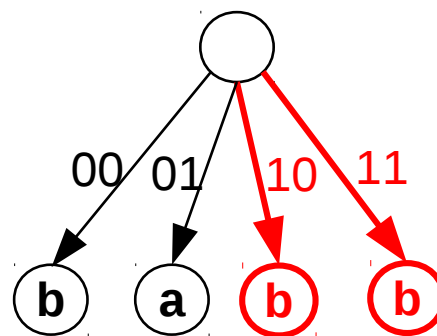
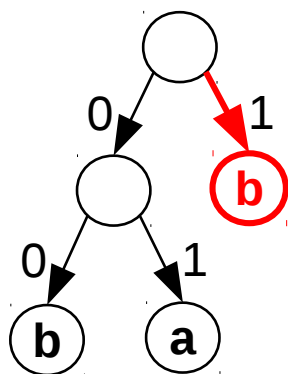
Szinttömötített prefix fa
(normalizált)



Alternatív
szinttömötített prefix fa
(normalizált)

Prefix kiterjesztése

- Maradjunk most a levélcímkézett fáknál
- Szinttömörítés: teljes mélységű fákat állítunk elő



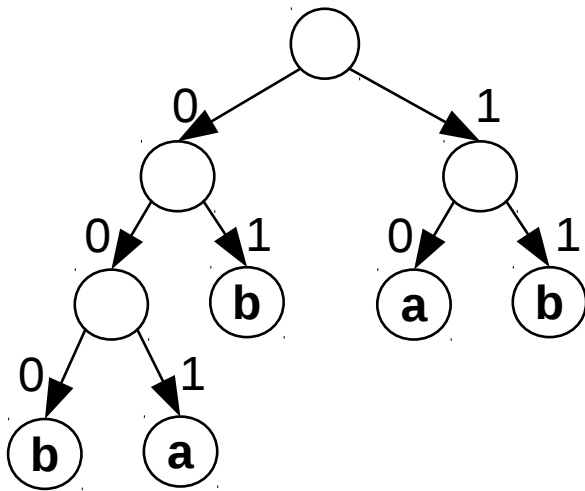
- Ekvivalens azzal, mintha a $128.0.0.0/1$ prefixet de-aggregáltuk volna a FIBben

IP prefix	Prefix	Címke
$0.0.0.0/2$	00	b
$64.0.0.0/2$	01	a
$128.0.0.0/1$	1	b

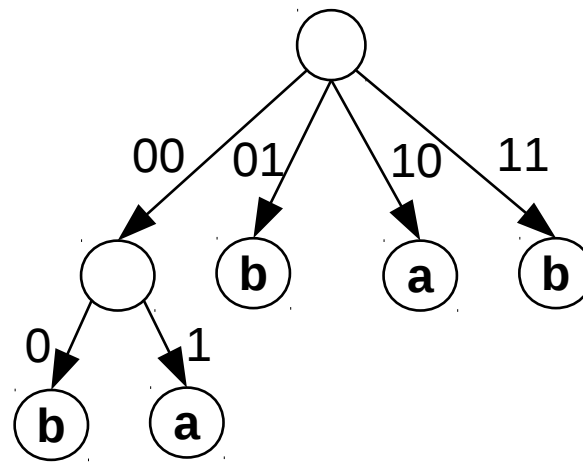
IP prefix	Prefix	Címke
$0.0.0.0/2$	00	b
$64.0.0.0/2$	01	a
$128.0.0.0/2$	10	b
$192.0.0.0/2$	11	b

Szinttömörített prefix fák

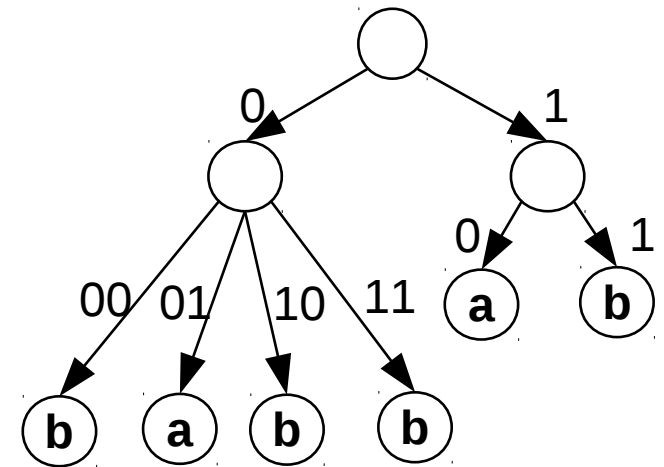
- **Előny:** kevesebb pointer = kisebb tárolási méret + gyorsabb LPM (kevesebb szintet kell bejárni)
- **Hátrány:** a pontokban tárolni kell a gyermekek 2^k számát, pontosabban k -t (ez az ún. *stride*)



Bináris prefix fa
(normalizált)



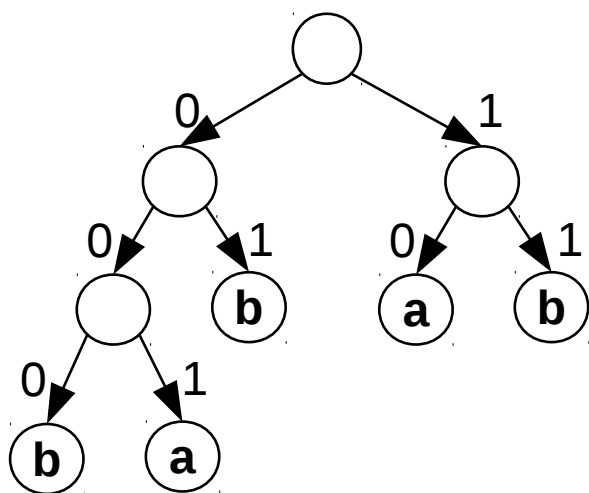
Szinttömötített prefix fa
(normalizált)



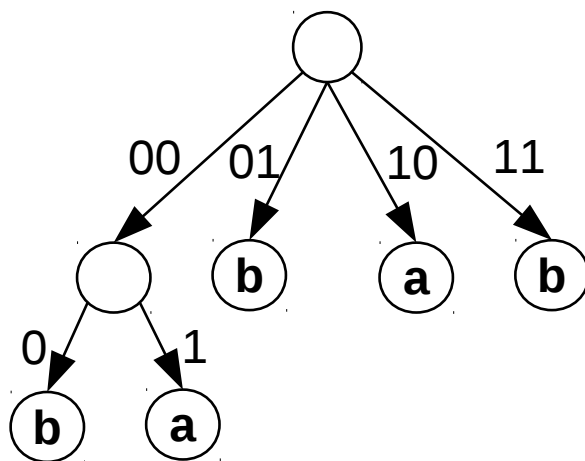
Alternatív
szinttömötített prefix fa
(normalizált)

Optimális színtömörítés

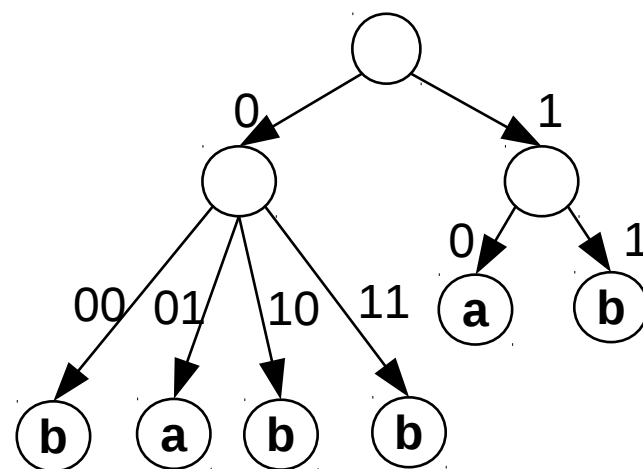
- Nem mindegy, hogy a fa egyes pontjait mekkora *stride*-ra írjuk ki → **optimalizálás**
- Az alábbi példán az első színtömörített fában csak 6 pointer van, a másodikban 8



Bináris prefix fa
(normalizált)



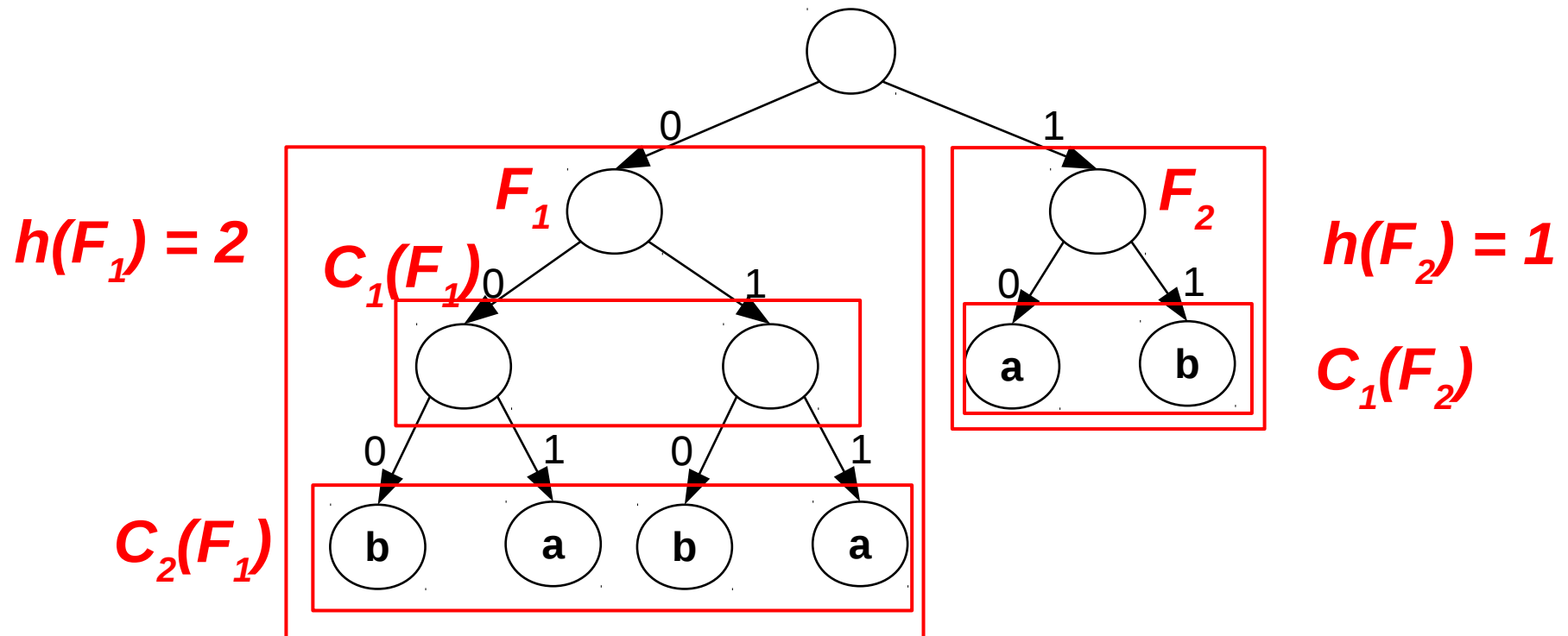
Színtömötített prefix fa
(normalizált)



Alternatív
színtömötített prefix fa
(normalizált)

Optimális szinttömörítés

- Induljunk ki a normalizált bináris prefix fából
- Legyen $h(F)$ egy F részfa **mélysége** és legyen $C_i(F)$ az F részfában az **i -edik szinten levő gyermekek halmaza** (a gyökértől számítva)



Optimális szinttömörítés

- Jelölje $n_k(F)$ az F részfában levő pointerek számát abban az esetben, ha F gyökerében a gyermekek száma pont 2^k ($stride=k$)
- Tegyük fel, hogy valahogy már meghatároztuk F összes részfájára az optimális $stride$ méretet és a hozzá tartozó optimális $n(F)$ faméretet
- Levélpontokra könnyű: levélpont alatti részfa mindig üres (nincs benne pointer), így:

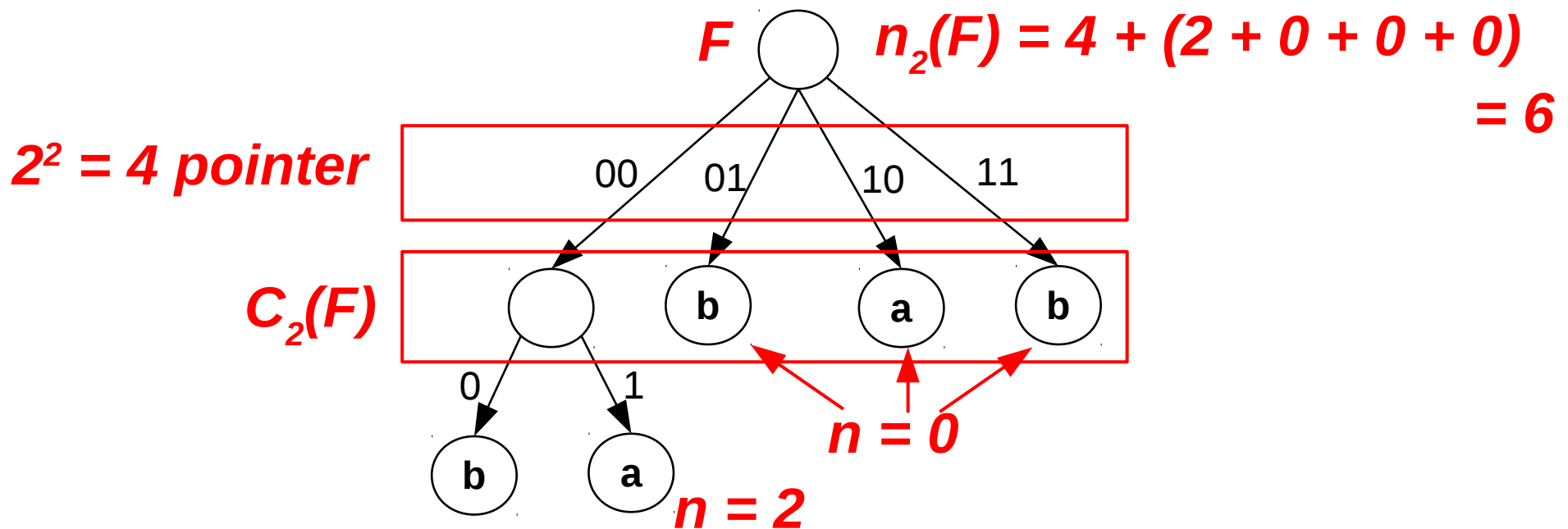
$$n(F) = 0 \quad \text{ha } F \text{ levélpont}$$

Optimális szinttömörítés

- **Ötlet:** ha egy F részfat k *stride*-ra írunk ki, akkor a benne levő pointerok száma

$$n_k(F) = 2^k + \sum_{G \in C_k(F)} n(G)$$

- Például az alábbi F fa 2 -es *stride*-on: 6 pointer



Optimális szinttömörítés

- Természetesen minden pontra érdemes a legkisebb fát eredményező *stride*-ot választani

$$n(F) = \min_{k=1..h(F)} n_k(F) = \min_{k=1..h(F)} (2^k + \sum_{G \in C_k(F)} n(G))$$

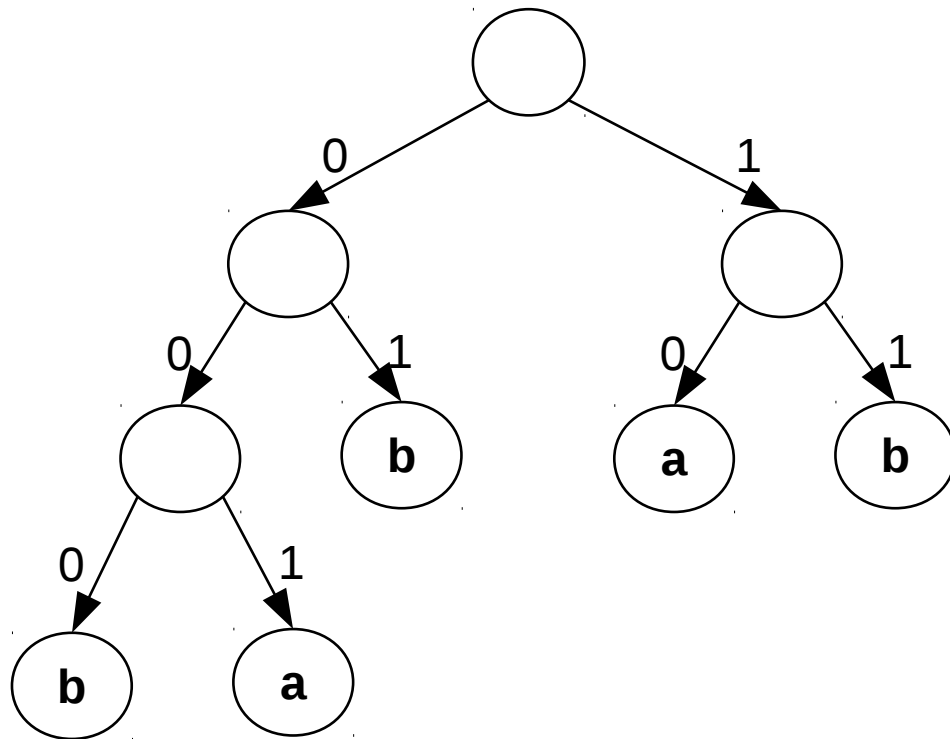
- **Algoritmus:** postorder sorrendben minden F részfára kiszámítjuk az optimális $n(F)$ értéket
- **Dinamikus program:** a részproblémák ismét a részfák, önmagukban optimálisan megoldhatók!

$$n(F) = \min_{k=1..h(F)} (2^k + \sum_{G \in C_k(F)} n(G)) \quad \forall F \text{ belső pont}$$

$$n(F) = 0 \quad \forall F \text{ levélpont}$$

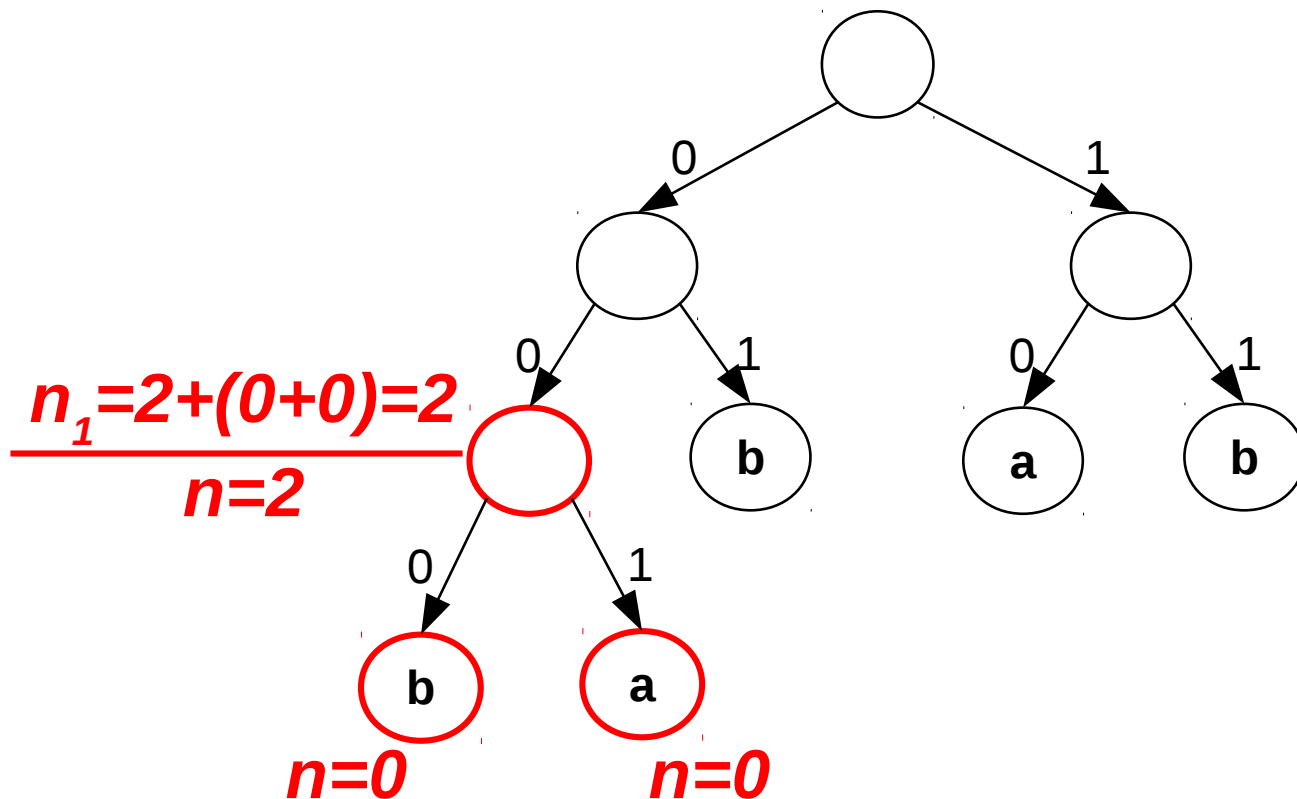
Optimális szinttömörítés: példa

1) Induljunk a normalizált bináris prefix fából



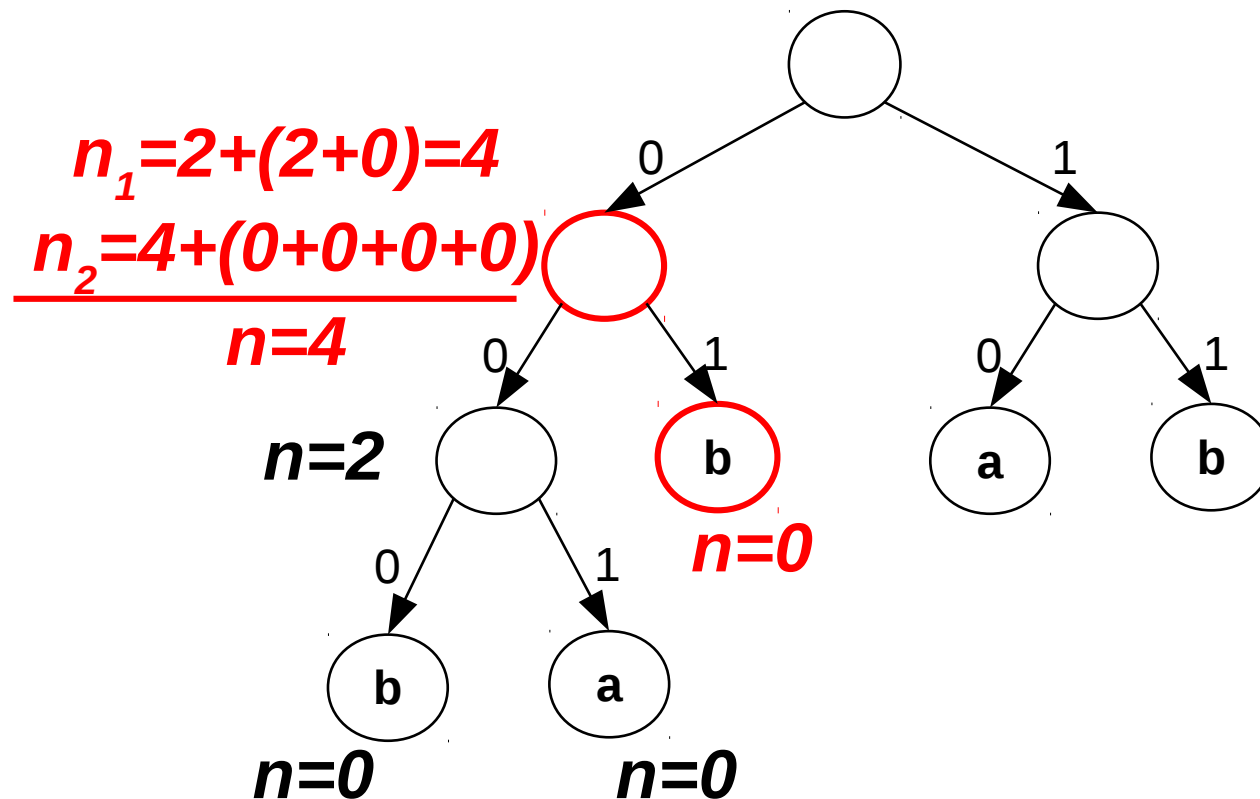
Optimális szinttömörítés: példa

2) Minden F részfára határozzuk meg az $n_k(F)$ és $n(F)$ értékeket postorder sorrendben



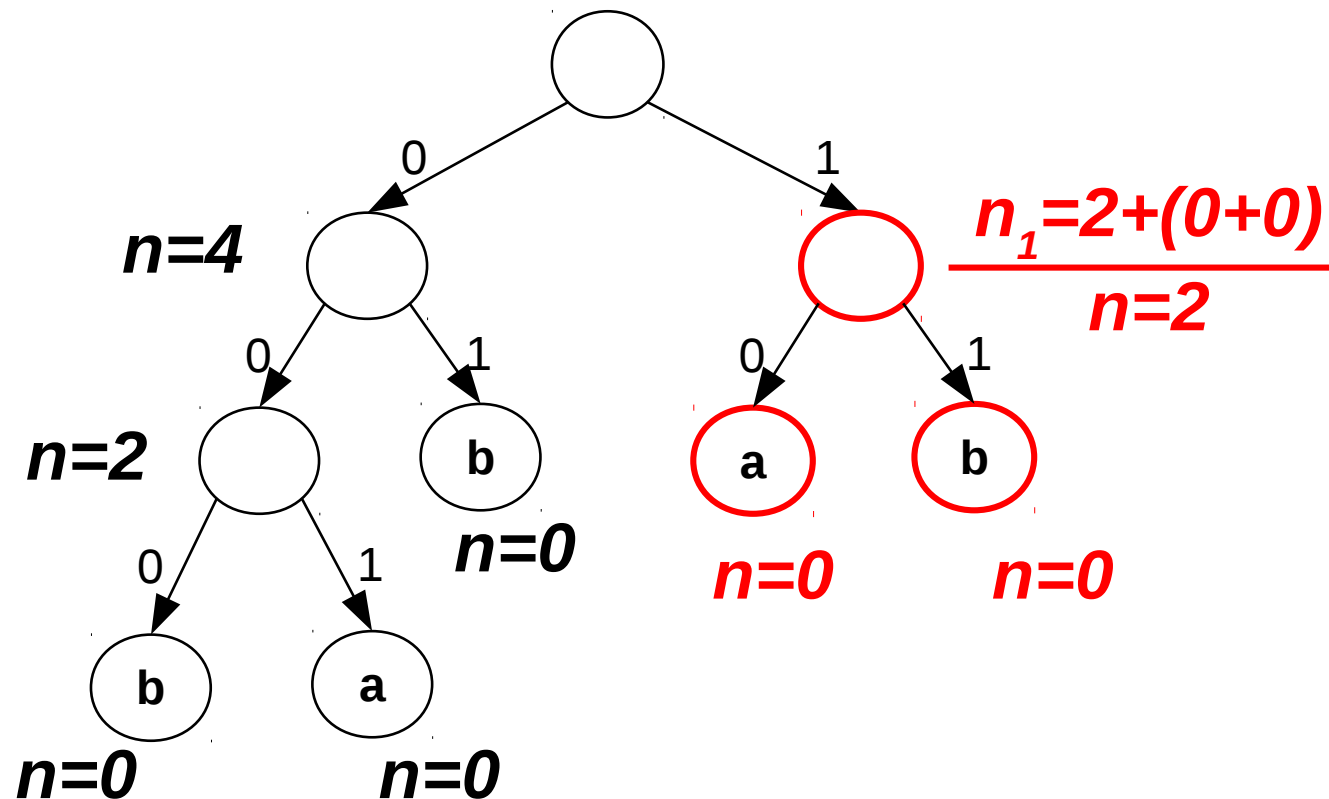
Optimális szinttömörítés: példa

2) Minden F részfára határozzuk meg az $n_k(F)$ és $n(F)$ értékeket postorder sorrendben



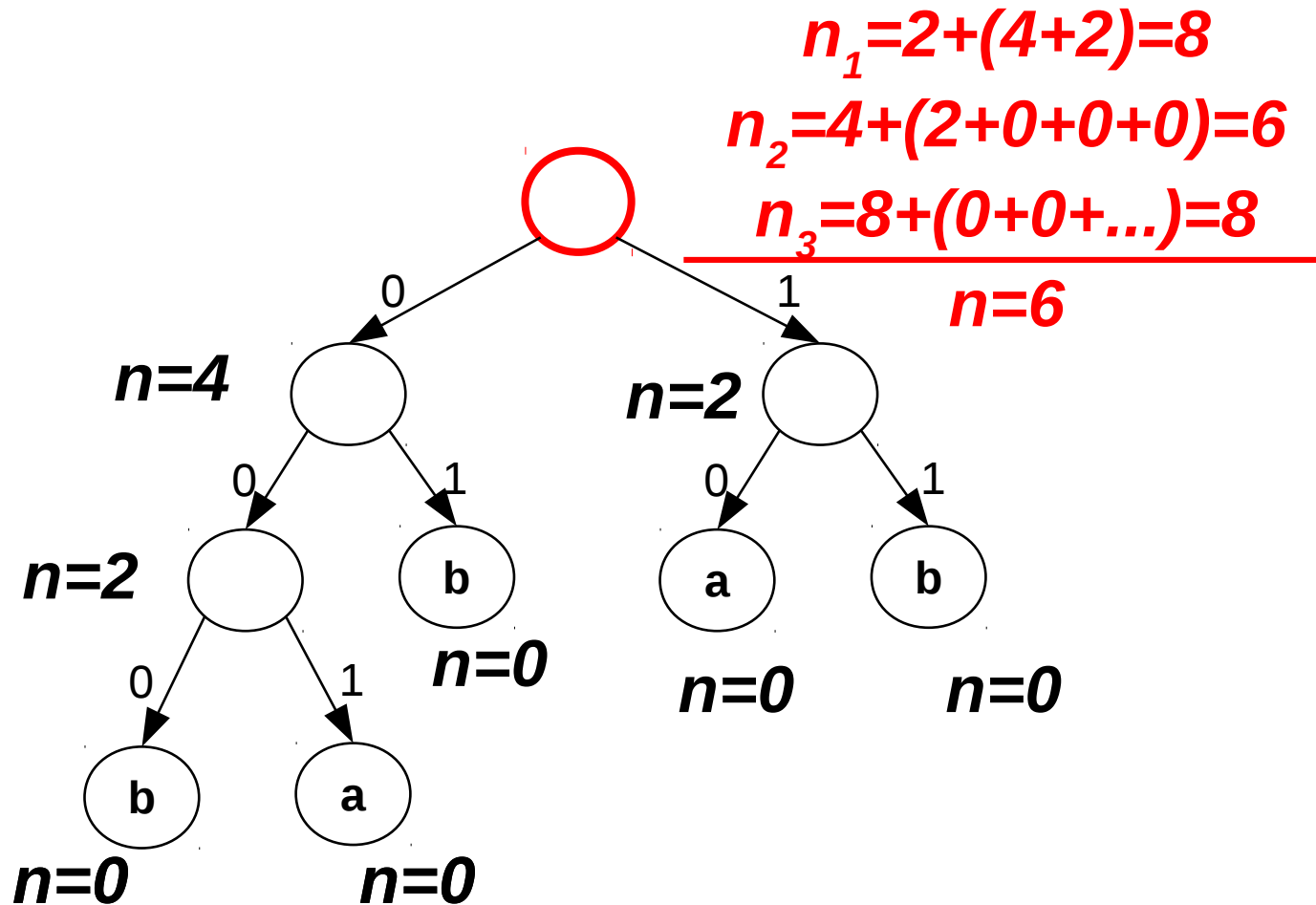
Optimális szinttömörítés: példa

2) Minden F részfára határozzuk meg az $n_k(F)$ és $n(F)$ értékeket postorder sorrendben



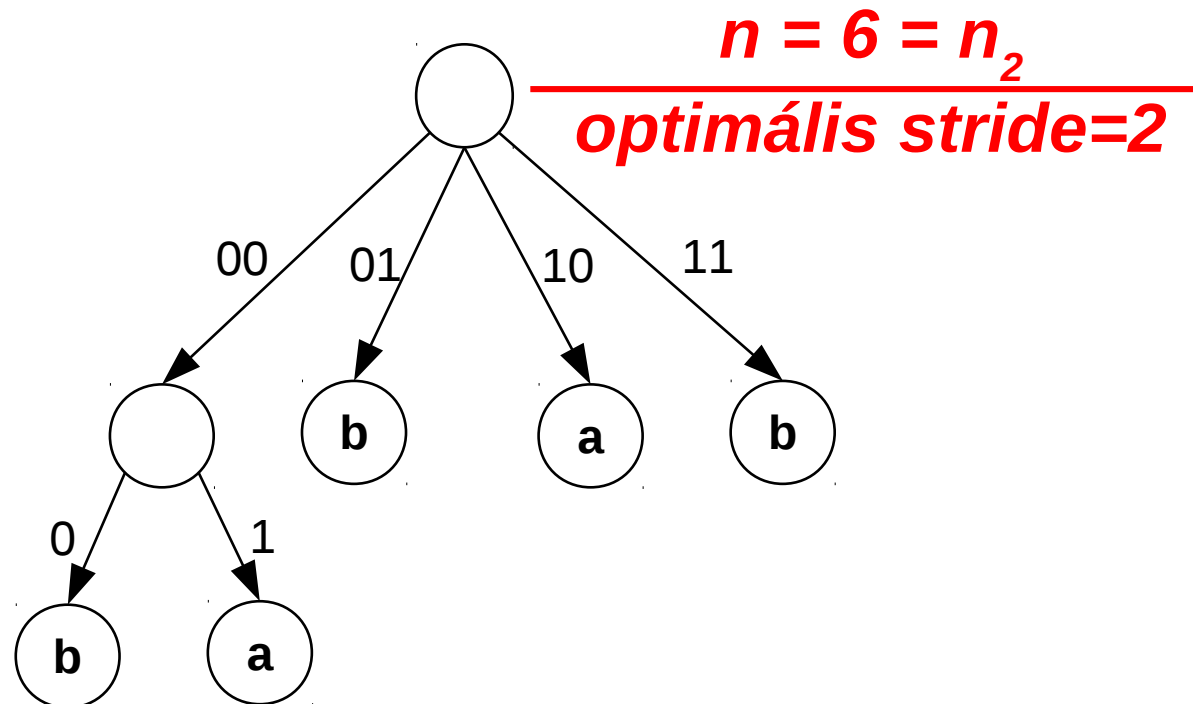
Optimális szinttömörítés: példa

2) Minden F részfára határozzuk meg az $n_k(F)$ és $n(F)$ értékeket postorder sorrendben



Optimális szinttömörítés: példa

- 3) Az optimális megoldás $n(F)=6$ birtokában előállítjuk az optimális szinttömörített prefix fát
- egy preorder bejárással megoldható, ha a pontokban tároljuk az optimális *stride*-méretet



Optimális szinttömörítés

- A szükséges preorder/postorder bejárásokat most nem részletezzük (gyakorlat)
- A szinttömörítés csökkenti a fa méretét és a szintjeinek számát (vagyis az LPM keresés idejét)
- A gyakorlatban elterjedten használt:
 - a Linux kernel `fib_trie` adatstruktúrája egy szint- (és útvonal)tömörített prefix fa
 - network processzorok mindegyike támogatja
 - Intel DPDK
- Kombinálható az ORTC-vel (PhD téma, valaki?)