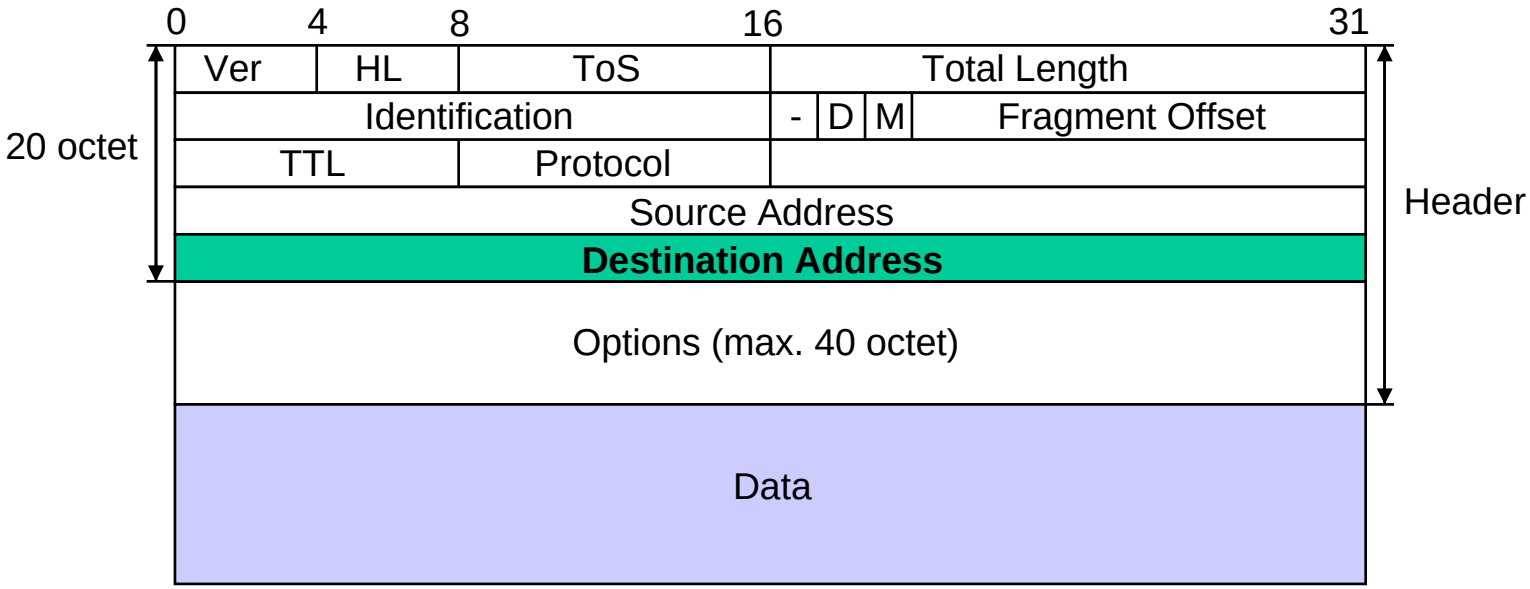
Hálózatok II
2006

2: Az IP Prefix Lookup Probléma I.

Internet Protocol IP

- Az adatok a küldőtől a cél-állomásig IP-csomagokban kerülnek átvitelre
- A csomagok fejléce tartalmazza a cél IP-címét
  - IPv4: 32 Bit-címek
  - IPv6: 128 Bit-címek

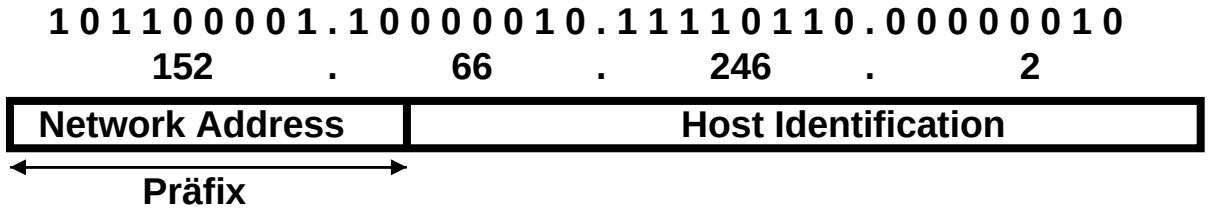


IPv4 Paket

Internet Protocol IP

- Minden csomópont (router) a cél címe alapján dönt, hogy melyik szomszédos csomópontnak kell továbbítania a csomagot
  - Az ehhez szükséges információt minden csomópontban egy routing-tábla (ill. forwarding-tábla) tárolja

Osztály alapú címzés IPv4-ben: A prefix 8, 16, vagy 24 bit hosszú lehet



Kiterjesztett hálózati cím (Extended Network Address) IPv4-ben:

- Hogyan lehet ezeket az információkat hatékonyan tárolni és megtalálni?

IP Prefix Lookup Probléma

- A címek *W* hosszúságú bináris sztringek.

Egy router routing-táblája *R*

- bejegyzéseket tartalmaz (*x,y*) formában
  - *x*: legfeljebb *W* hosszú bináris sztring, melynek a neve prefix.
    x lehet az üres sztring is, amit ε-nal jelölünk,
  - *y*: egy a router-ből kivezető link.
- Minden *x* bináris sztringhez *R* legfeljebb egy (*x,y*) bejegyzést tartalmaz.
- A bejegyzések száma a routing-táblában *N.*

Feladat:

- Egy a routerhez érkező csomaghoz, melynek cél-címe *k*,
- találjuk meg *R*-ben azt az (*x,y*) bejegyzést, melyre teljesül, hogy
  - *x* prefixe *k*-nak és
  - *x* maximális hosszúságú.
- Ezt az *x* sztringet a leghosszabb egyező prefixnek nevezzük
  (angol.: best matching prefix, röviden BMP)

IP Prefix Lookup Probléma

Példa:

- Legyen W=5.
- R bejegyzései: (0,y_1), (001,y_2), (10,y_1), (110,y_3), (111,y_2)
- Legyen a cél-cím egy csomagban k=(00100)
- A leghosszabb egyező prefix (BMP) ekkor (001,y_2)

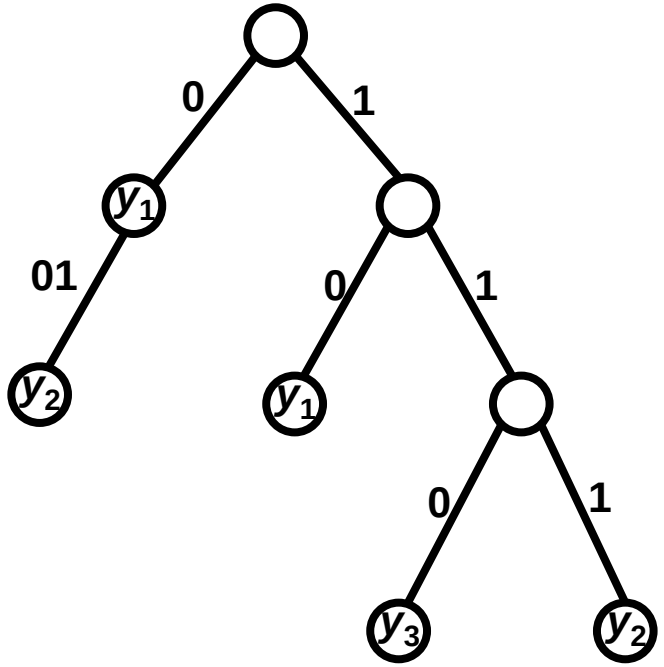
Kívánt tulajdonságok R adatstruktúrájához:

- Gyors BMP keresés (Lookup) minden csomaghoz
  - O(log W) tárhozzáférés, független N-től (a gyakorlatban 3-4 hozzáférés)
- Alacsony tárigény, azaz $O(N)$
- R aktualizálásának hatékony támogatása, azaz bejegyzések hozzáadásának, törlésének és megváltoztatásának a támogatása. Az ilyen adatstruktúrákat dinamikus adatstruktúráknak nevezzük.

IP Prefix Lookup Trie Adatstruktúrával

Egy Trie egy keresőfa, amiben

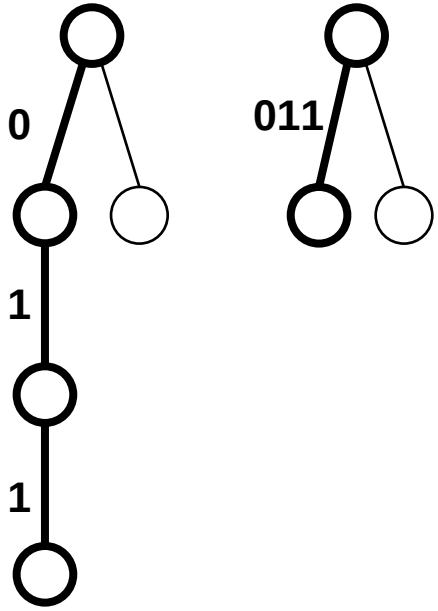
- az élek cimkézettek,
- minden *v* csomópontnál az élek cimkéi, amik *v*-t a gyermekeivel kötik össze, különböző jellel kezdődnek,
- minden *v* csomópont azt a sztringet reprezentálja, ami az élcimkék konkatenálásával az úton a gyökértől *v*-hez áll elő.



IP Prefix Lookup Trie Adatstruktúrával

Megjegyzés:

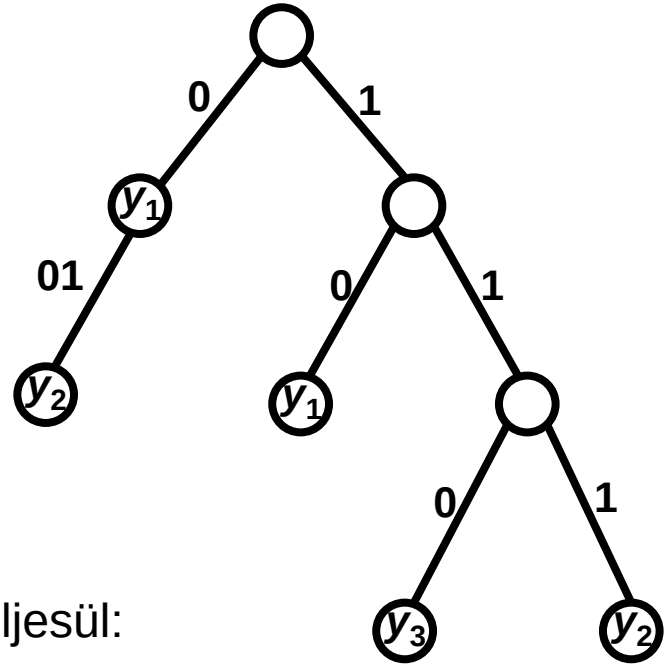
- Gyakran használnak olyan trie-t,melyben minden élcimke egyetlen jelből áll.
- Ha egy trie egy láncot tartalmaz, melyben a csomópontoknak csak egyetlen gyermeke van (és nem reprezentálnak bejegyzést a routing-táblában), akkor helyettesíthetjük a láncot egy egyetlen éllel, melynek cimkéje az eredeti élek cimkéinek konkatenációja.
- Ezt az operációt úttömörítésnek nevezzük. Az úttömörítés előnye: a trie csomópontok alacsonyabb száma és ezzel alacsonyabb tárigény.



IP Prefix Lookup Trie Adatstruktúrával

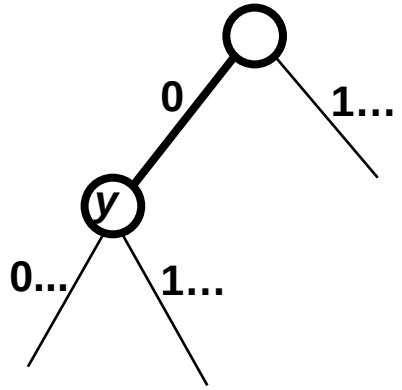
Trie mint adatstruktúra egy routing-tábla *R* tárolására:

- Az élcimkék bináris sztringek.
- Minden csomópontnak legfeljebb két gyermeke van.
- A cimke a baloldali gyermekhez vezető élen 0-val kezdődik, a jobboldalihoz 1-gyel.
- Minden (*x,y*) bejegyzéshez van a trie-ban egy csomópont, ami *x*-et reprezentálja. Ebben a csomópontban tároljuk az *y* linket.
- Minden legfeljebb *W* hosszú *x* bináris sztringhez, a trie pontosan egy csomópontot tartalmaz, amely *x*-et reprezentálja, ha a következő feltételek közül egy teljesül:
  - *x* az üres sztring, ekkor *x* a gyökérnek felel meg;
  - *R* tartalmaz egy (*x,y*) bejegyzést;
  - *R* tartalmaz két bejegyzést (x_0,y_0) és (x_1,y_1), úgy hogy *x*0 prefixe x_0-nek és *x*1 prefixe x_1-nek.



A Trie Konstrukciója

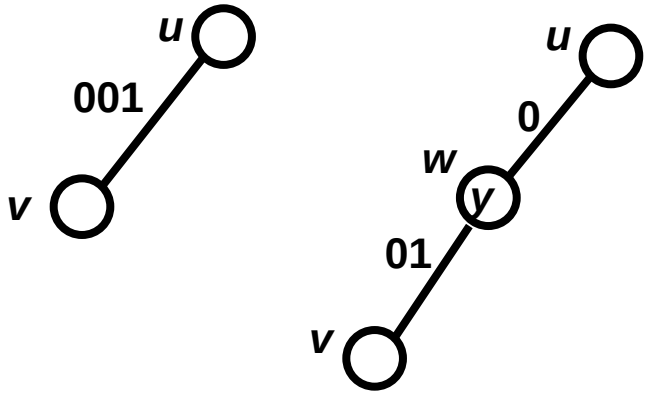
- Kezdetben a trie csak a gyökércsomópontból áll.
- Ezután *N* bejegyzést tetszőleges sorrendben befűzünk.
- Az (*x*,*y*) bejegyzés befűzésekor a gyökérnél indulunk és követjük az utat, amit *x* definiál:
  - Minden csomópontnál azt az élet követjük, melynek cimkéje ugyanazzal a jellel kezdődik, mint *x* fennmaradó része.
- Az út következőképp fejeződhet be:
  1. Elértük a trie egy *v* csomópontját, amikor *x*-et teljesen feldolgoztuk:
     Megjelöljük *v*-t mint csomópontot, ami *R* egy bejegyzésének felel meg és tároljuk az *y* linket *v*-ben. (Ilyen csomópontot röviden megjelöltnek nevezünk)



(*x*,*y*) befűzése *x*=0-val

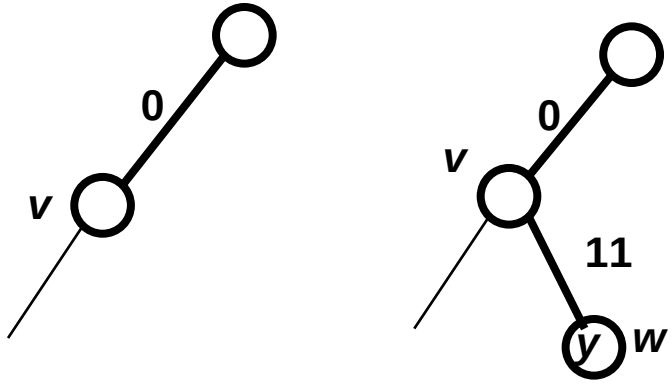
A Trie Konstrukciója

2. Az út egy (u,v) élen végződik, melynek cimkéje $\alpha\beta$, az α és β között, mikor x-et teljesen feldolgoztuk:
   Letrehozunk egy új w csomópontot (u,v) közepén és az (u,w) élhez az α cimkét rendeljük és a (w,v) élhez a β cimkét. Az y linket tároljuk w-ben és w-t megjelöljük.



(x,y) befűzése, x=0

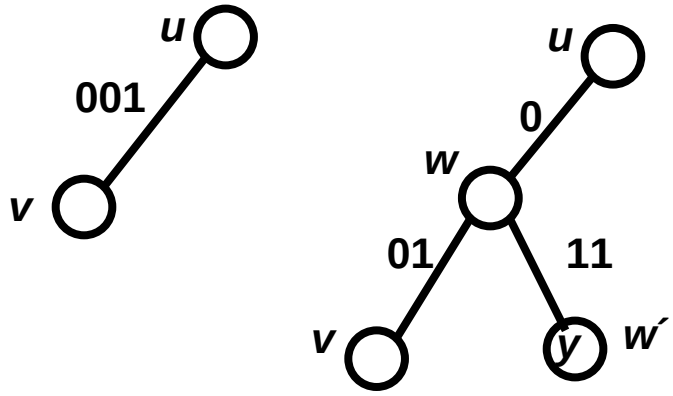
3. Az út egy v csomópontban végződik, mielőtt x-et teljesen feldolgoztuk:
   Legyen γ az x fennmaradó része. Létrehozunk v-hez egy új w gyermeket és az (v,w) élnek a γ cimkét adjuk. Az y linket tároljuk w-ben és w-t megjelöljük.



(x,y) befűzése, x=011

A Trie Konstrukciója

4. Az út egy (*u,v*) élen végződik, melynek cimkéje $\alpha\beta$, az α és β között, mielőtt x-et teljesen feldolgoztuk:
   Legyen γ az *x* fennmaradó része.
   A (*u,v*) élet kettéosztjuk α és β között egy új *w* csomópont befűzésével.
   Létrehozunk *w*-hez egy új *w´* gyermeket.
   Az (*u,w*) élnek az α cimkét, a (*w,v*) énekek a β-t és a (*w,w´*) élnek a γ-t adjuk.
   Az *y* linket tároljuk *w´*-ben és *w´*-t megjelöljük.



(*x,y*) befűzése, *x*=011

- Minden bejegyzést be tudunk a trie-ba fűzni $O(W)$ idő alatt
- A teljes konstrukció időigénye: $O(N \cdot W)$

A Trie Tárigénye

- Minden csomópont tárigénye konstans.
- Minden él tárigénye szintén konstans.
  (Feltesszük, hogy egy *W* hoszú bináris sztring konstans sok számítógép szóban tárolható.)
- Pontosan *N* csomópont reprezentál a trie-ban egy *R*-beli bejegyzést, a trie minden más csomópontja belső csomópont, melynek pontosan két gyermeke van (máskülönben, a trie definíciója miatt, ezeket a csomópontokat nem tartalmazná a trie).
- Egy fában, melynek legfeljebb *N* levele van, legfeljebb *N*-1 belső csomópont lehet melynek (legalább) két gyereke van.
- Ezért a trie legfeljebb 2*N*-1 csomópontot és 2*N*-2 élet tartalmaz.
- Így a tárigény *O*(*N*).

Keresés a Trie-ban

- Legyen *k* a cél címe, amit keresünk.
- A gyökérben kezdünk és követjük az utat, amit *k* határoz meg.
  Közben mindig megjegyezzük az utolsó megjelölt csomópontot, amit már meglátogattunk.
- Ha *k*-t teljesen feldolgoztuk, vagy ha az út a trie-ban nem hosszabbítható meg *k* következő bitjével, akkor az utolsó megjelölt csomópont, amit meglátogattunk, reprezentálja a leghosszabb egyező prefixet (BMP).
- Így a BMP keresése $O(W)$ időt igényel.

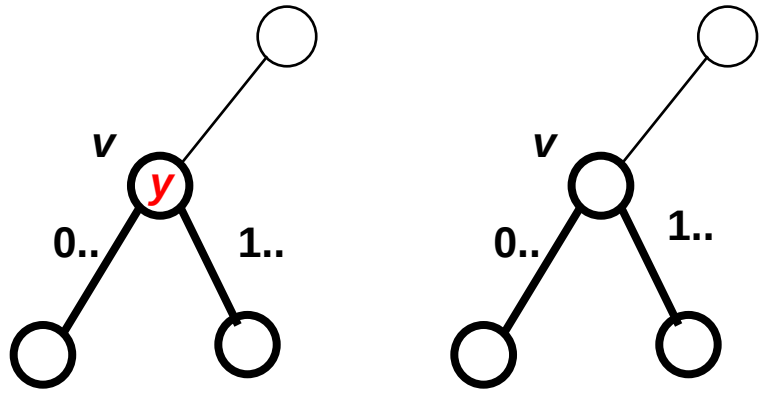
A Trie Aktualizálása

- Legyen (*x,y*) egy új bejegyzés, amit be kell fűzni.
  Ez, mint a trie konstrukciójánál, *O*(*W*) időt igényelt.

- Legyen (*x,y*) egy bejegyzés, amit törölni kell.

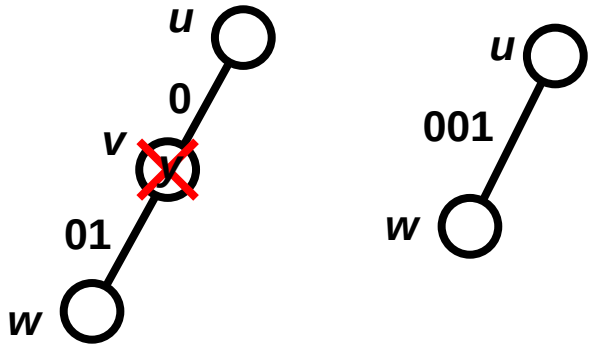
  - Megkeressük *O*(*W*) idő alatt azt a *v* csomópontot, ami (*x,y*) reprezentálja.

  - Ha *v*-nek két gyermeke van, a megjelölést töröljük, de *v* a trie-ban marad.

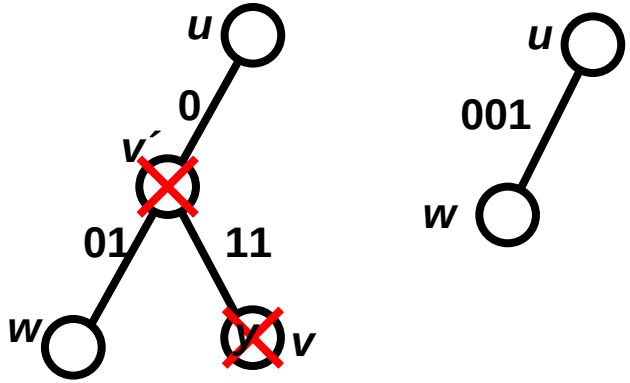


A Trie Aktualizálása

- Ha *v*-nek pontosan egy gyermeke van és *v* nem a gyökér, akkor töröljük *v*-t és az éleket, ami *v*-t a szülőjével és a gyermekével kötötte össze, egy éllé kombináljuk.



- Ha *v* egy levél, akkor töröljük *v*-t. Ha *v* szülője *u* nem a gyökér és nem megjelölt, akkor *u*-t töröljük és az *u*-hoz incidens két élet egy éllé kombináljuk..



A Trie Aktualizálása

- Egy bejegyzés törlése $O(W)$ időt igényel.

- Egy (x,y) bejegyzésben, ha csak a linket változtatjuk meg, akkor elég a trie-ban a bejegyzést lokalizálni és y-t megváltoztatni.
  Ez $O(W)$ időt igényel.
  (Egyébként egy bejegyzés $O(W)$ idő alatt úgy változtatható meg, hogy töröljük és újra befűzzük.)

Tétel 1: A trie adatstruktúrával a routing-táblát $O(N)$ tárterületen lehet tárolni. Minden keresés és aktualizálás $O(W)$ időt igényel. A trie felépítésének időigénye $O(N \cdot W)$. □

IP Prefix Lookup bináris kereséssel prefixhossz alapján

[Waldvogel et al. 97]

- Alapötlet: Az IP Prefix Lookup Probléma megoldható konstans idő alatt, ha minden bejegyzés hossza megegyezik (azaz ha az *x* sztring minden bejegyzésben ugyanolyan hosszú), ha elegendő tár áll rendelkezésre.

- A routing-tábla bejegyzéseit a hosszuk alapján *W* osztályba soroljuk.
- Egy osztályon belül a bejegyzéseket egy olyan adatstruktúrában tároljuk, amiben a keresés *k* cím szerint konstans idő alatt lehetséges.
- Bináris kereséssel $O(\log W)$ idő alatt megkeressük azt a hosszosztályt, amely *k*-hoz a leghosszabb egyező prefixét tartalmazza.

Keresés egy hosszosztályon belül

- Notáció:
  - Egy hosszosztály L_i tárolja az R routing-tábla mindazon (x,y) bejegyzéseit, melyben x hossza i.
  - Legyen $N_i=|L_i|$ az ilyen bejegyzések száma.

- Van olyan adatstruktúra, amiben
  - L_i-t $O(N_i)$ tárterületen tudjuk tárolni és
  - egy adott i hosszú k bináris sztringhez $O(1)$ idő alatt el tudjuk dönteni, hogy L_i tartalmaz-e egy (k,y) bejegyzést, vagy nem.

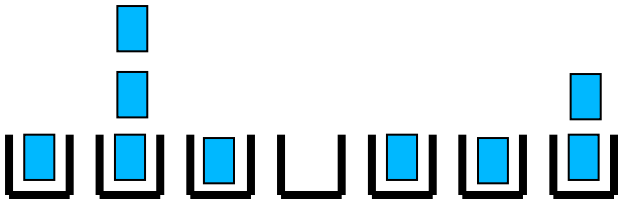
- Példa egy ilyen adatstruktúrára:
  Perfekt Hashing [Fredman, Komlós, Szemerédi 84]

Keresés egy hosszosztályon belül - Hashing

Hashing:

- A hash-táblában L_i tárolására a rendelkezésre álló indexek száma legalább $c \cdot N_i$ kell hogy legyen, ahol c>0 egy konstans.
- Egy *f* hash-függvény L_i minden (*x*,*y*) bejegyzéséhez hozzárendel egy *f*(*x*) indexet.

- Pl. $f(x) = (3*x \mod 23) \mod 7$



Keresés egy hosszosztályon belül - Hashing

Hashing:

- Egy *f* hash-függvény L_i minden (*x*,*y*) bejegyzéséhez hozzárendel egy *f*(*x*) indexet.
- Feltesszük, hogy *f*(*x*) konstans idő alatt kiszámolható.
- Ha több bejegyzés ugyanarra az indexre képeződik le, akkor kollízióról beszélünk.
- Kollíziók kezelésére egy lehetőség az összeláncolás: azaz minden *j* indexnél a prefixeket, melyek *j*-re képeződnek le, egy listába összeláncolva tároljuk.
- Ha a hash-táblában egy *k* prefixet keresünk, akkor kiszámoljuk *f*(*k*)-t és az *f*(*k*) indexnél az összeláncolt listában keresünk.
- Egy lista hosszának várható értéke *O*(1), így a keresés várhatóan *O*(1) időt igényel. (worst-case idő: $O(N_i)$)

Keresés egy hosszosztályon belül - Hashing

- Perfekt Hashing:
  - tárigénye $O(N_i)$ és
  - keresés időigénye $O(1)$ worst-case
- Alapötlet: Kétszintű hashing:
  - Az első hash-függvény f alkalmazása után kollízió léphet fel.
  - A kollíziók feloldására minden j indexnél egy második f_j hash-függvényt alkalmazunk, amely a j-re leképeződött bejegyzések számától függ, és amely címtartománya négyzetes, azaz, ha n_j a bejegyzések száma, amiket f a j -re képezett le, akkor $f_j(k) \in [1, c \cdot n_j^2]$.
  - Ha f és f_j véletlen lineáris függvények, akkor konstans valószínűséggel (pl. 1/2 valószínűséggel) nem lesz kollízió, és a szükséges tár $O(N_i)$.
  - Így a „függvények fele jó“. Ha egy véletlen funkcióról kiderül, hogy nem rendelkezik az elvárt tulajdonsággal, akkor választunk véletlenül egy másikat.

Perfekt Hashing

- Perfekt Hashing:
  - Minden hash függvényhez van worst-case input, ezért hash-függvények családját tekintjük:
    - Legyen s a hash-tábla mérete,
    - $U = \{1,\ldots,m\}$ (univerzum)
    - $S \subseteq U$, $|S| = n$, az input
    - $H_s = \{ h : U \rightarrow \{1,\ldots,s\} \mid h(x) = (kx \bmod p) \bmod s, 1 \leq k < p \}$ ahol p prím, $p > m$.
  - Az első hash-függvény s partícióra osztja az elemeket
  - A partíciókat egy második hash függvénnyel tároljuk kollízió nélkül
  - Válasszuk h-t véletlenül H_s-ből egyenletes eloszlás szerint
  - Legyen $W_j^h = \{ x \in S \mid h(x) = j \}$

Perfekt Hashing

Lemma 1: (bizonyítás nélkül) Ha *h*-t egyenletes eloszlás szerint véletlenül választjuk H_s-ből, akkor

$$E\left(\sum_{1\le j<s}\binom{|W_j^h|}{2}\right)\le\frac{n(n-1)}{s}$$

Ahol $E(X)$ az X véletlen változó várható értéke. □

Következésképpen:

$$\Pr\left(\sum_{1\le j<s}\binom{|W_j^h|}{2}<\frac{2n(n-1)}{s}\right)\ge\frac{1}{2} \qquad (1)$$

Ha $s = 2(n-1)$, akkor ebből következik, hogy legalább a $h\in H_s$ függvények felére:

$$\sum_{1\le j<s}\binom{|W_j^h|}{2}<n$$

Egy ilyen függvényt használunk S partícionálására W_j blokkokra, $1\le j < s$.

Perfekt Hashing

Minden W_j blokkhoz, $1 \leq j < s$, válasszuk $s_j = \max\{ 1, 2|W_j|(|W_j|-1) \}$,

Ekkor (1) szerint legalább a $h_j \in H_{sj}$ függvények felére érvényes:

$$\sum_{1 \leq j < s} \binom{|W_{j,\ell}^{h_j}|}{2} < 1$$

ahol $W_{j,\ell}^{h_j} = \{x \in W_j \mid h_j(x) = \ell\}$

Azaz legalább a $h_j \in H_{sj}$ függvények fele injektív.

Egy ilyen függvényt használunk W_j tárolására. Így nem fordul elő kollízió.

A teljes tárigény:

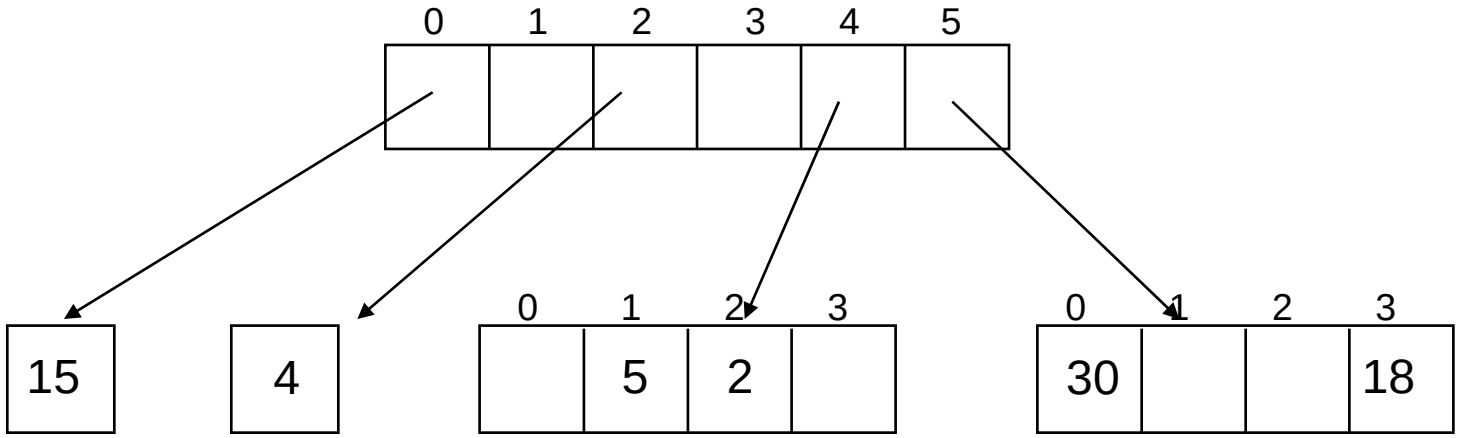
$$\sum_{1 \leq j < s} s_j \;\leq\; s + 4 \cdot \sum_{1 \leq j < s} \binom{|W_j|}{2} \;=\; O(n)$$

Perfekt Hashing Példa

m=30, p=31, s=6, S={2,4,5,15,18,30}

$h(x) = (2x \bmod 31) \bmod 6$

$h_4 = (1x \bmod 31) \bmod 4$, $h_5 = (3x \bmod 31) \bmod 4$,



Bináris keresés prefixhossz szerint

- Első ötlet:
  - Kezdjünk a „középső“ hosszosztállyal $L_{W/2}$-vel. $O(1)$ idő alatt teszteljük, hogy $L_{W/2}$ tartalmazza-e k prefixét.
  - A teszt eredményétől függően, keressünk tovább rekurzívan a L_1,.., $L_{W/2-1}$ (ha nem) vagy a $L_{W/2+1}$,…,L_W (ha igen) osztályokban.
  - Ehhez $O(\log W)$ teszt szükéges és így $O(\log W)$ idő.

- Sajnos ez az ötlet így még nem működik. Pl.:
  - Legyenek 0, 1, 00, 111 a prefixek R-ben.
    Ekkor L_1={0,1}, L_2={00}, L_3={111}.
  - Keressünk a 11101 célcím szerint.
    - L_2-vel kezdünk: a prefix 11 nincs benne L_2-ben, így a rövidebb prefixek között keresnénk L_1-ben, pedig a leghosszabbb egyező prefix 111 L_3-ban van.

Bináris keresés prefixhossz szerint

- Első módosítás:
  - Legyen x egy i hosszú prefix és jelölje $x[1..j]$ az x első j jelét, $1 \leq j \leq i$.
  - Ha x az L_i hosszoszályban van, akkor tárolunk minden L_j hosszosztályban, $1 \leq j < i$, amelyben nincs bejegyzés $x[1..j]$-hez, egy u.n. jelzést $x[1..j]$-hez.
    - A jelzés biztosítja, hogy egy keresésnél $x[1..j]$-nél felismerjük, hogy R-ben van egy hosszabb bejegyzés, ami $x[1..j]$-vel kezdődik. Így a bináris keresés a helyes döntést tud hozni, miszerint a hosszabb prefixek között keres tovább.
    - Az előző példában: $L_1=\{0,1\}$ $L_2=\{00,11\}$, $L_3=\{111\}$.

Bináris keresés prefixhossz szerint

- Van még egy probléma:
  - Ha a példánkban a 11010 célcím szerint keresünk, a 11 jelzés L_2-ben ahhoz vezetne, hogy a keresés L_3-ban folytatódik.
  - Mivel L_3-ban nem találunk 110 bejegyzést, a keresés visszatérne L_2-höz.
  - Mivel 11 csak egy jelzés, nem valódi bejegyzés, a keresést L_1-nél kellene folytatni. Ez a „backtracking“ $O(\log^2 W)$ futási időt eredményezne.

Bináris keresés prefixhossz szerint

- Második módosítás:
  - Minden *x* jelzésnél tárolunk egy *x.BMP* mutatót:
  - *x.BMP* arra az *R*-beli (*x´*,*y´*) bejegyzésre mutat,
    - ahol *x´* egy prefixe *x*-nek és
    - *x´* maximális hosszúságú.
  - Ha a bináris keresés egy *x* jelzést talál meg, mint leghosszabb egyező prefix, akkor *x.BMP* megadja a valódi leghosszabb egyező prefixet.
    - A példánkban 11.*BMP* az 1-re mutat L_1-ben.
    - Ha a 11010 célcím szerint keresünk, akkor 11.*BMP* megadja a leghosszabb egyező prefixet 1-et.

- Ezen kiegészítések után a bináris keresés helyesen működik:
  - A worst-case keresési idő $O(\log W)$.
  - Ha *R*-ben kevesebb mint *W* különböző hossz fordul elő, mondjuk *r*, akkor a worst-case keresési idő $O(\log r)$.

A routing-tábla tárigénye

- A jelzések bevezetése nélkül $O(N)$ tár.

- Ha egy L_i-beli x prefixhez minden L_j,$1 \le j < i$, hosszosztályban tárolunk egy jelzést, ha L_j még nem tartalmaz bejegyzést $x[1..j]$-hez, prefixenként $O(W)$ jelzésre lehet szükség.
- Így az adatstruktúra társzükségletére csak $O(W \cdot N)$ korlátot tudunk adni.
- Ha a jelzéseket csak azokban a hosszosztályokban tároljuk, ahol tényleg szükséges (amiket a bináris keresés tényleg tesztel), akkor a korlát $O(N \log W)$.

A routing-tábla konstrukciója

- Az adatstruktúra felépítéséhez szükséges idő lineáris a tárolt értékek (bejegyzések+jelzések) számával, feltéve, hogy a táblákat L_i-khez $O(|L_i|)$ idő alatt fel tudjuk építeni. Ez érvényes hash-táblák esetén.
  - R bejegyzéseit prefixhossz szerint növekvő sorredben dolgozzuk fel.
  - Egy i hosszú x prefixet befűzzük L_i táblájába.
  - Ezután befűzzük a jelzéseket az L_j táblákba, $j < i$, hossz szerint csökkenő sorrendben, amíg egy tábla nem tartalmaz egy igazi bejegyzést a megfelelő prefixhez.

A routing-tábla aktualizálása

- Egy (*x,y*) bejegyzés hozzáfűzése:
  - Először (*x,y*)-t befűzzük L_i-be, ahol *i* az *x* hossza.
  - Az L_j, $j < i$, táblákban be kell fűzni esetleg egy jelzést.
  - Minden *x´* jelzésnél a L_j, $j > i$, táblában, amihez *x* a leghosszabb egyező prefix, az *x´.BMP* mutatót (*x,y*)-ra kell állítani.
  - Az aktualizálás nagyon időigényes lehet, mivel lehetséges, hogy sok jelzést kell aktualizálni.
  - Egy bejegyzés törlésekor a szituáció hasonló.
  - Tanács: Egy darabig gyüjtsük az aktualizálásokat és utánna építsük fel újra a teljes adatstruktúrát.

Irodalom

- Marcel Waldvogel, George Varghese, Jon Turner, Bernhard Plattner: **Scalable High Speed IP Routing Lookups**. *Proc. ACM SIGCOMM '97,* 25-36, 1997.

- Michael L. Fredman, János Komlós, and Endre Szemerédi: **Storing a sparse table with O(1) worst case access time**. *Journal of the ACM*, Vol. 31(3), 538--544, 1984.

- Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan: **Dynamic perfect hashing: upper and lower bounds**, *SIAM Journal on Computing*, Vol. 23(4), 738-761, 1994.