Pokročilé dátové štruktúry a ich analýzy

- AVL stromy
- Analýza hašovania
- k – d stromy
- Reorganizácia - princíp dynamizácie
- **Dynamizácia hašovania**
  - dynamické hašovanie
  - rozšíriteľné hašovanie
  - lineárne hašovanie
- **Podpora dotazov**
  - čiastočná zhoda
  - intervalová zhoda
  - relačné operácie

AVL(1) stromy (Adelson, Velskij, Landis)

AVL(1) stromy sú binárne vyhľadávacie stromy, v ktorých výška ľavého podstromu a pravého podstromu v každom uzle sa líši najviac o jedna. *(Výška = najdlhšia cesta od koreňa k listu.)*

Úplný binárny strom o výške h má 2^h listov a $2^{h+1}-1$ uzlov.

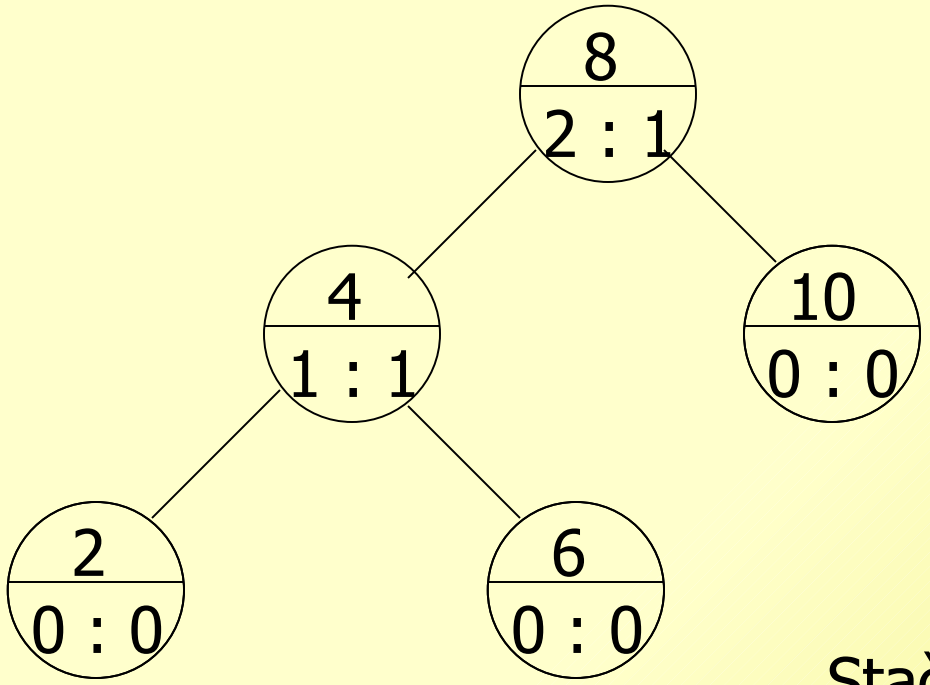
Veta: Pre výšku h AVL(1) stromu s n uzlami platí: $lg(n+1) \leq h < 1.4404\, lg(n+2) - 0.328$

Konvencia:
lg = $\log_2$
ln = $\log_e$

Dôkaz: Najmenšiu výšku dosiahneme ak AVL strom je úplný binárny strom. Označme $|T|$ počet uzlov stromu T. Aby AVL strom T_h výšky h mal minimálny počet uzlov, musí v každom platiť rekurentná rovnica: $|T_h| = |T_{h-1}| + |T_{h-2}| + 1$.

Jej riešením je $|T_h| = F_{h+2}-1 > \frac{1}{\sqrt{5}}\phi^{h+2} - 2$, kde $\phi = \frac{1}{2}\left(1+\sqrt{5}\right)$

Narábanie s AVL - stromami



Vyváženie sa poruší vložením uzlov 1, 3, 5, 7. Iné vloženia vyváženie napravia.

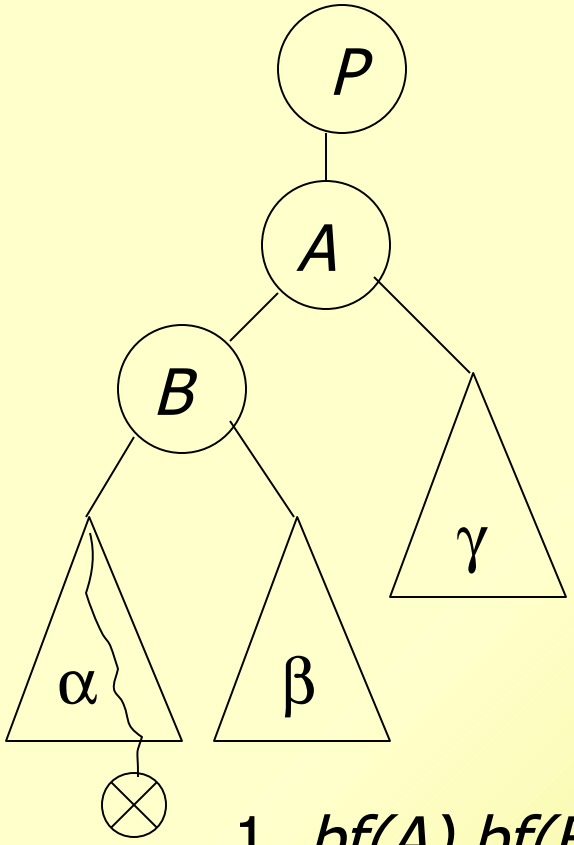
Faktor balansovania
$bf = h_R - h_L$; $bf \in \{-1, 0, 1\}$

Stačí v každom uzle pamätať *bf*.

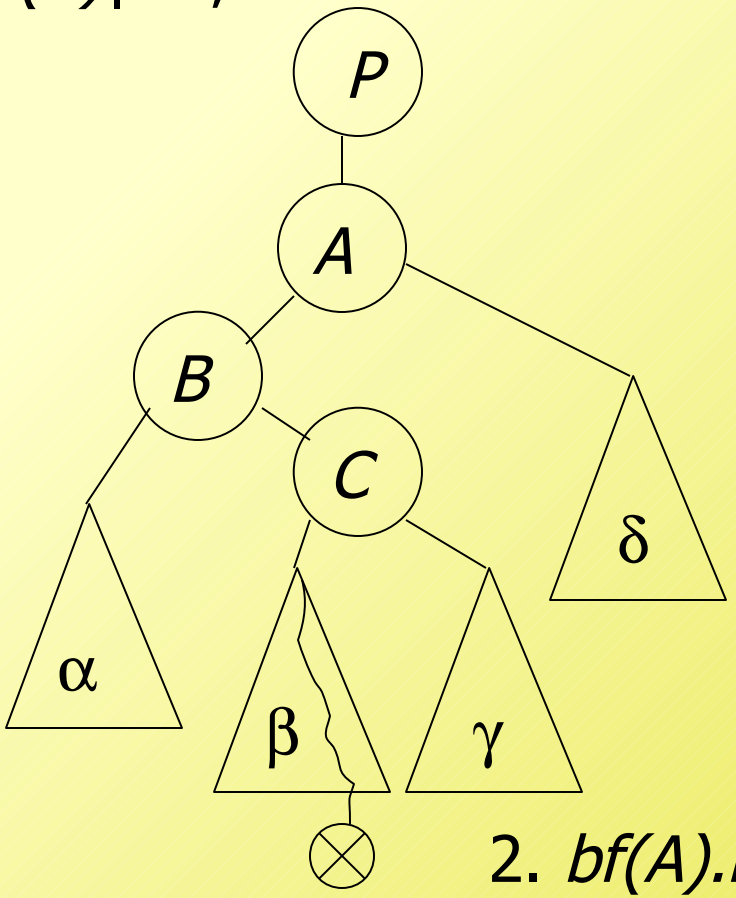
Vkladanie najprv realizujeme bez ohľadu na balansovanie. Pri hľadaní vhodného miesta si zapamätáme do premennej *a* smerník na posledný uzol s nenulovým *bf* alebo koreň, ak uzol s nenulovým *bf* neexistuje.

Dokončenie vloženia.

Spätne od vloženého listu po uzol *A*, na ktorý ukazuje *a* aktualizujeme *bf*. Ak v tomto uzle bf nadobudol prípustnú hodnotu skončíme. Ak |*bf(A)*|=2, môžu nastať dva prípady:

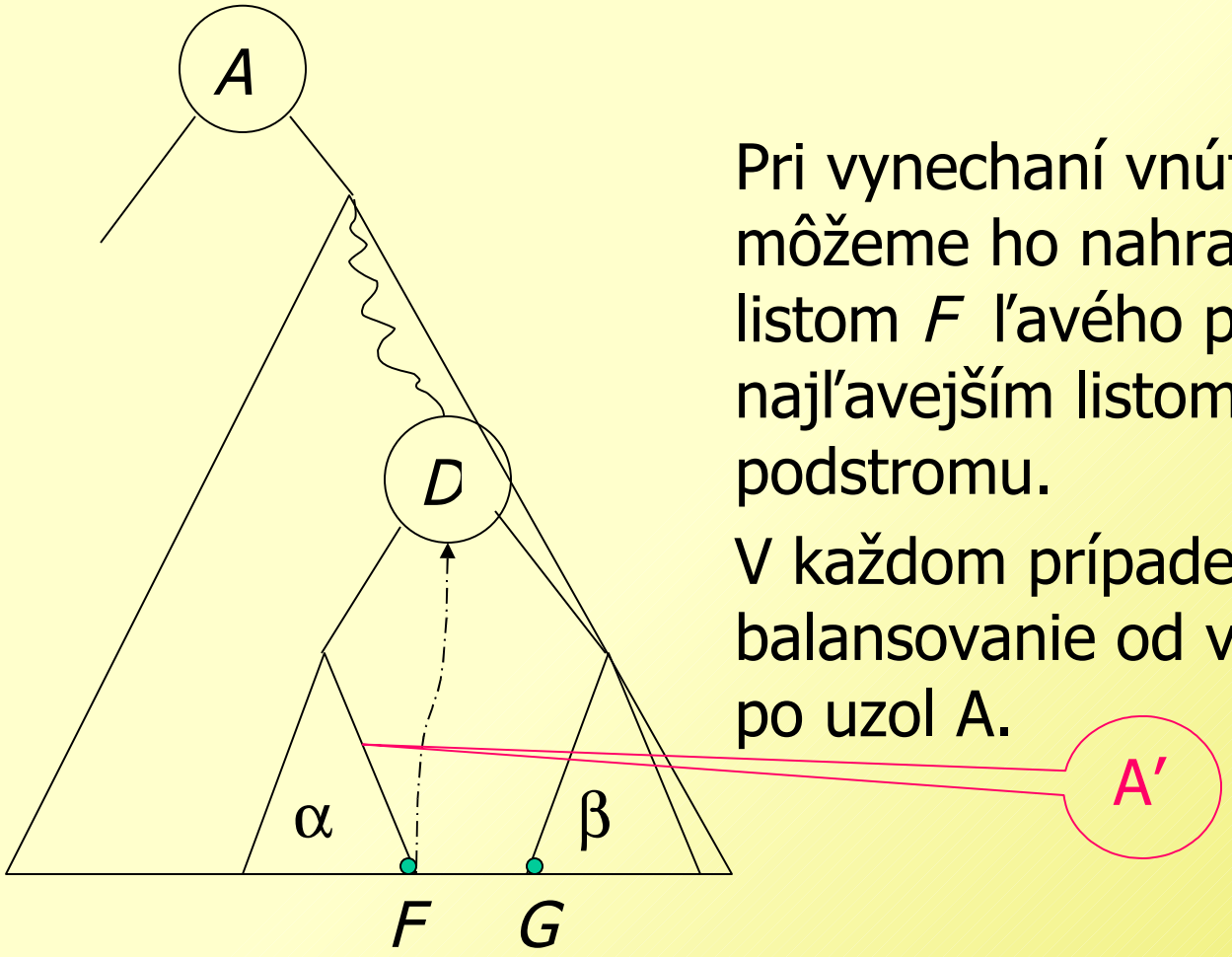


1. *bf(A).bf(B)>0*



2. *bf(A).bf(B)<0*

Vynechanie z AVL stromu



Pri vynechaní vnútorného uzla *D* môžeme ho nahradiť najpravejším listom *F* ľavého podstromu alebo najľavejším listom *G* pravého podstromu.

V každom prípade stačí opraviť balansovanie od vynechaného listu po uzol A.

Analýza hašovania

Predpokladajme, že hašujeme do blokov veľkosti 1. Budeme počítať pravdepodobnosť nájdenia voľného miesta na i-tý pokus (vkladanie, neúspešné vyhľadanie).

$$p_1 = \frac{M-n}{M}$$

$$p_2 = \frac{n}{M}\frac{M-n}{M-1}$$

$$p_3 = \frac{n}{M}\frac{n-1}{M-1}\frac{M-n}{M-2}$$

.

.

.

$$p_i = \frac{n}{M}\frac{n-1}{M-1}\frac{n-2}{M-2}\cdots\frac{n-i+2}{M-i+2}\frac{M-n}{M-i+1}$$

$$E_{n+1} = \sum_{i=1}^{n+1} i\, p_i = \frac{M+1}{M-n+1}$$

$$E = \frac{1}{N}\sum_{n=1}^{N} E_n = \frac{M+1}{N}\sum_{n=1}^{N}\frac{1}{M-n+2} = \frac{M+1}{N}\left(H_{M+1} - H_{M-N+1}\right)$$

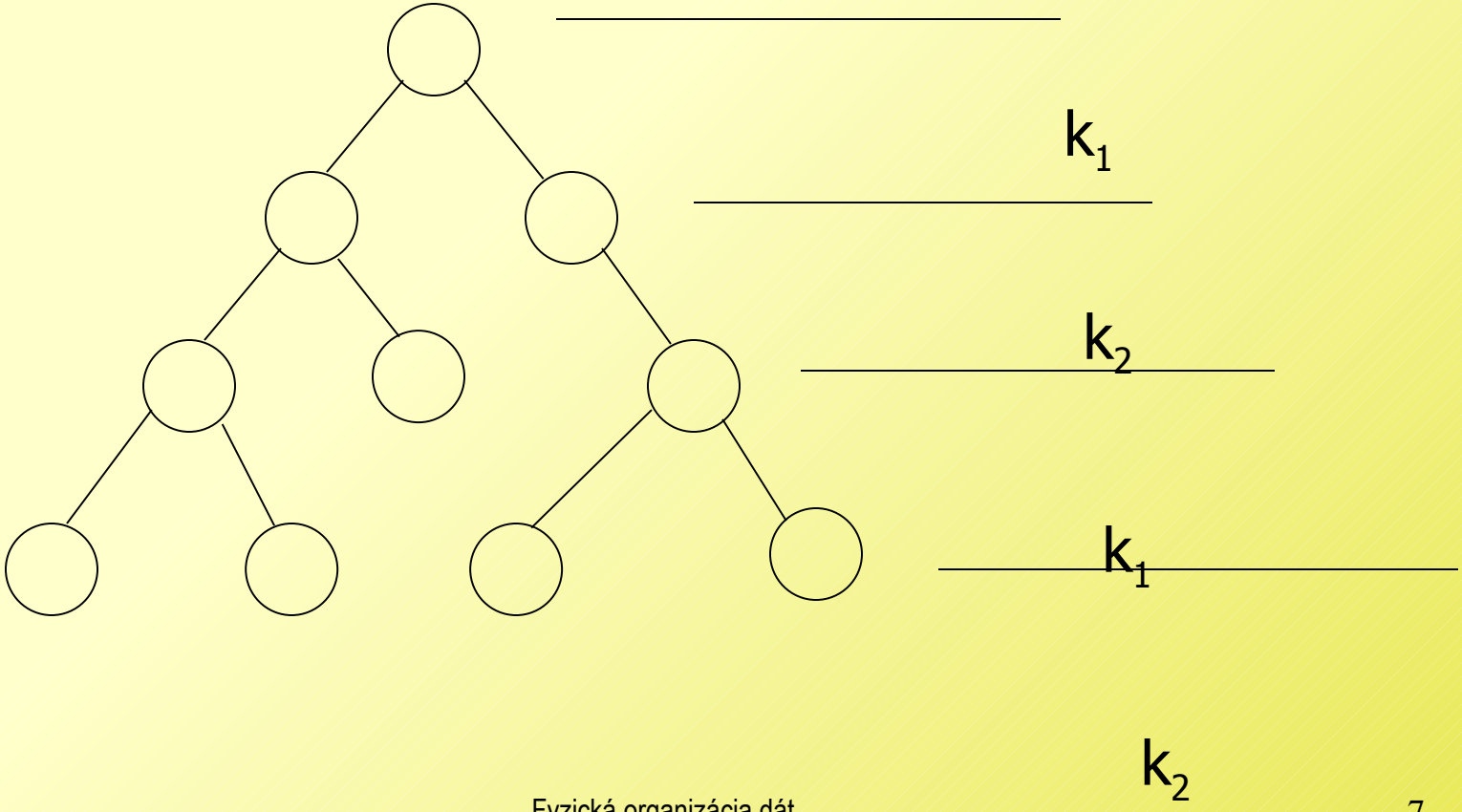
$$\alpha = \frac{N}{M+1} \Rightarrow E = -\frac{1}{\alpha}\ln(1-\alpha)$$

H_n je n-té harmonické číslo a platí

$H_n \approx \ln(n) + \gamma$

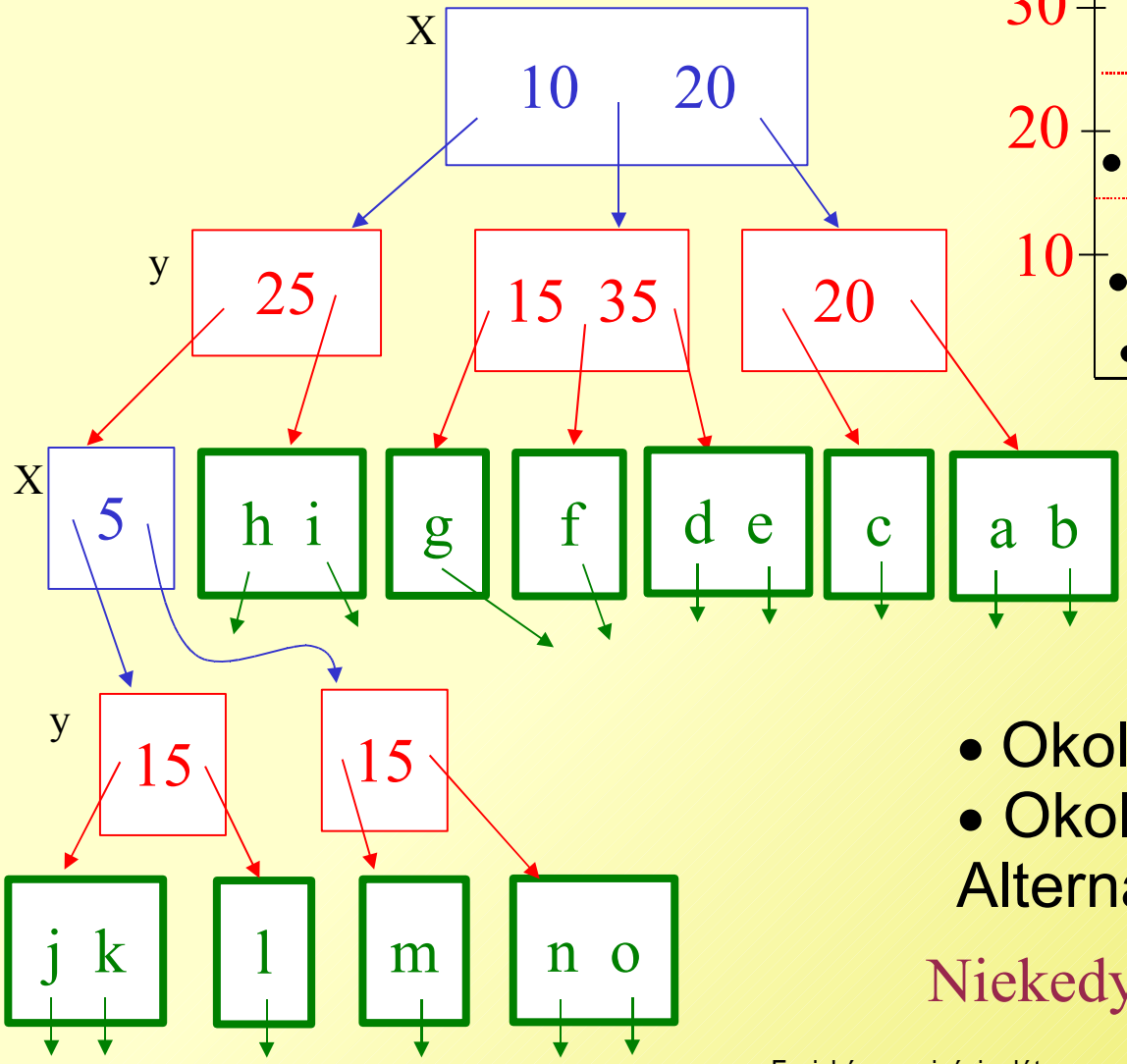
Mnohocestné vyhľadávacie stromy — k-d stromy

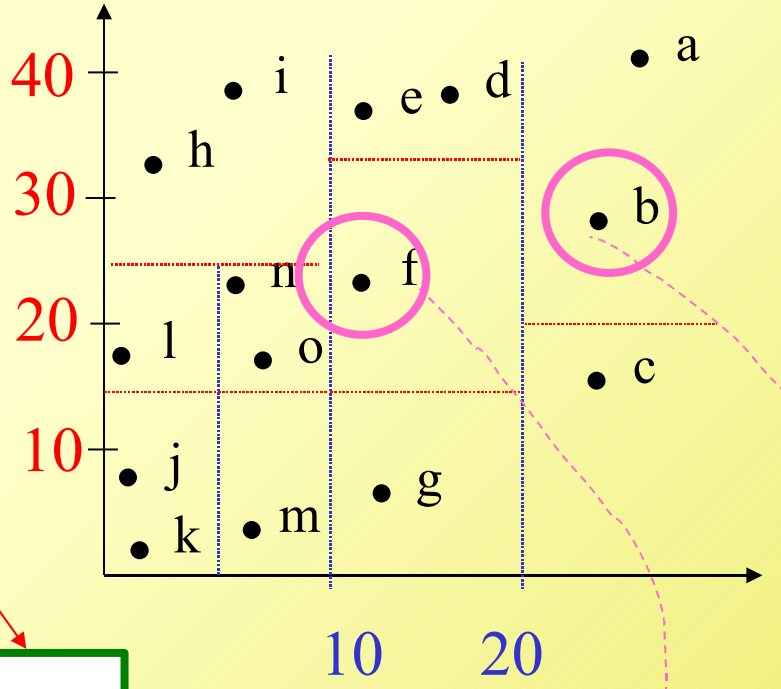
Pre vyhľadávanie podľa viacerých kľúčov môžeme uvažovať, že pri vyhľadávaní striedame cyklicky všetky použité kľúče.



Príklad:

„geografické dáta“





- Okolie (susedstvo) f
- Okolie b

Alternatíva: quadtrees

Niekedy sa tomu hovorí R-tree

Reorganizácia a dynamizácia

Uvažujeme, že dátová štruktúra veľkosti n sa dá vytvoriť v čase $t(n) \leq O(n \log n)$. Operácie v nej sa dajú vykonávať v čase $O(\log n)$ a počas jej životnosti sa vykoná $O(n)$ operácii. Potom ju treba reorganizovať.

Ak reorganizácia zväčší štruktúru na veľkosť qn. Potom si pri $q>1$ štruktúra zachová asymptotickú efektívnosť pri periodických reorganizáciách.

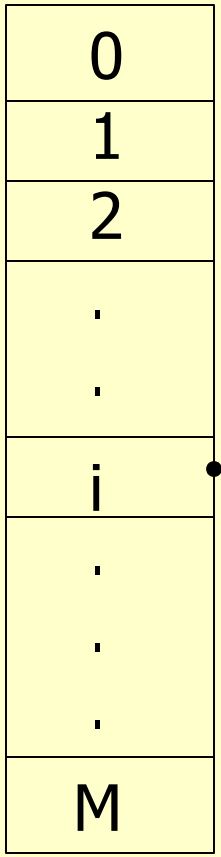
$$\lim_{n\to\infty} \frac{\sum_{i=0}^{n} q^i \log(q^i)}{nq^n} = konst$$

Amortizovaná cena operácií

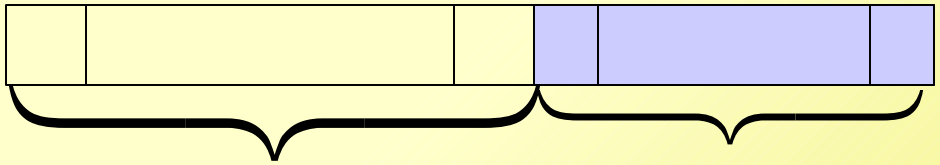
$$\sum_{i=0}^{n} i\,q^i = q\sum_{i=0}^{n} i\,q^{i-1} = q\,\frac{\partial}{\partial q}\left(\frac{q^n-1}{q-1}\right) = \frac{(n-1)\left(q^{n+1}-q^n\right)}{(q-1)^2}$$

$$\lim_{n\to\infty} \frac{(n-1)\left(q^{n+1}-q^n\right)\log q}{n(q-1)^2 q^n} = \lim_{n\to\infty} \frac{q^n(n-1)}{nq^n(q-1)} = \lim_{n\to\infty} \frac{(n-1)\log q}{n(q-1)} = \frac{\log q}{q-1}$$

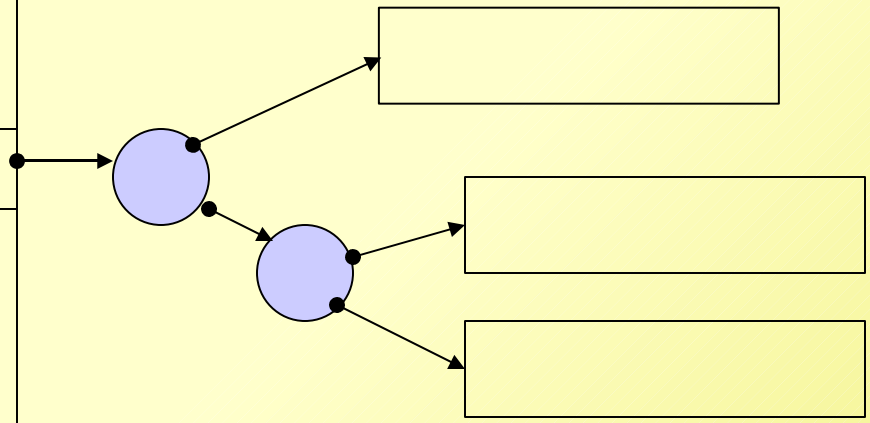
Dynamické hašovanie (P.A. Larson)



Idea:



hlavné bity h_1 náhradné bity h_2



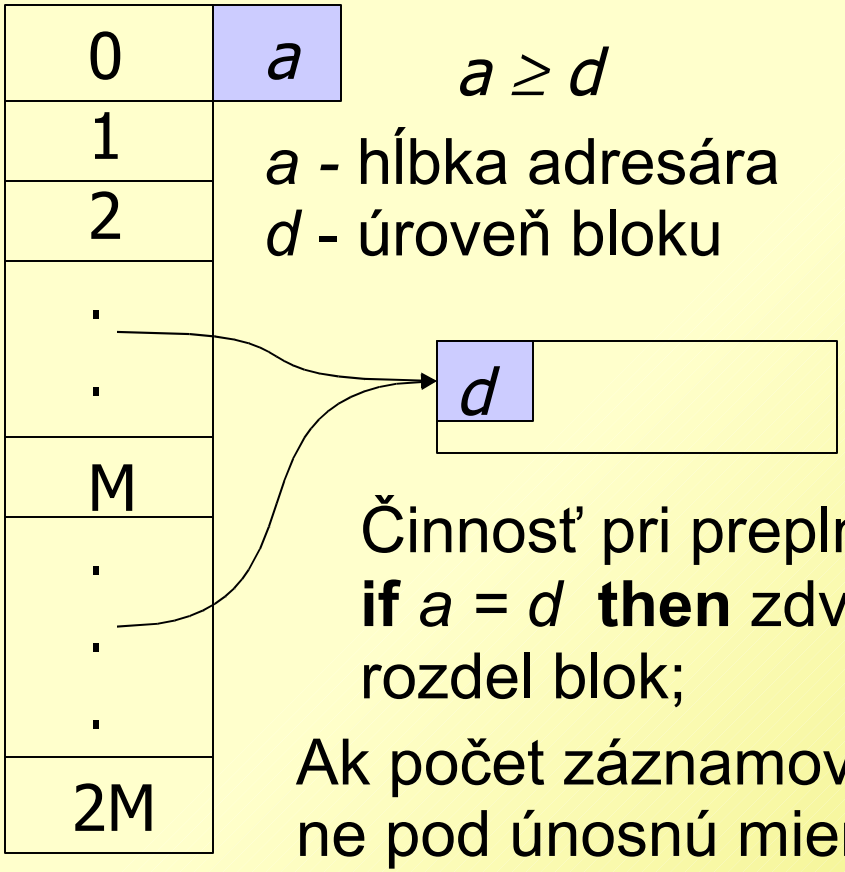
Príklad hašovacej funkcie:

$h(K) = K \bmod p;$ $h_1(K) = h(k) \text{ div } 2^q$

$h_2(K) = h(k) \bmod 2^q$

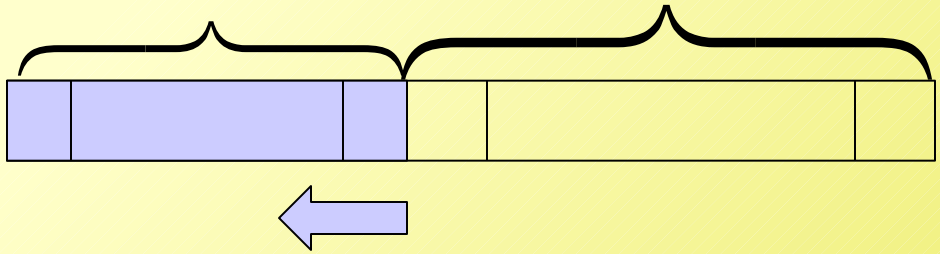
Rozšíriteľné hašovanie
(Fagin, Pipenger, Nievergelt a Strong)

Čo keby sme zmenili poradie hlavných a náhradných bitov?



$a \geq d$

a - hĺbka adresára
d - úroveň bloku



Počet záznamov
v dátovom bloku *b*.
Hysteréza

Činnosť pri preplnení:
if *a* = *d* **then** zdvojnásobiť adresár;
rozdel blok;

Ak počet záznamov v asociovaných blokoch poklesne pod únosnú mieru môžeme bloky opäť spojiť .

Lineárne hašovanie (W. Litwin)

Stále priveľa réžie s adresárom! Stačí jeden bit – existuje blok, neexistuje blok.

Nekonečná trieda hašovacích funkcií:

$$h(q, K) = K \bmod (2^q \times p)$$

Platí:

$$K \bmod 2n = \begin{cases} K \bmod n \\ (K \bmod n) + n \end{cases}$$

Nech $N = 2^q \times p$.
Adresár A = array[1:N] of bits.

Operácie v lineárne hašovanom súbore

Vyhľadanie bloku (kľúča, alebo miesta pre kľúč):
n: = N; k: = K mod N; /*hašovacia funkcia*/
while (k > p and not A[k]) do {n := n div 2; k := k – n}
Vloženie: Ak blok nie je plný vložíme.
Inak, ak n < N. k = k + n; A[k] := true; a vložíme.
Ak n = N a blok je plný zdvojnásobíme adresár A
for i = N downto 0 do A[N+i] = false;
A[k+N] = true; vloží sa záznam do bloku [k+N]
konečne sa modifikuje hašovacia funkcia N:= 2*N;

Zisk je zdanlivý, aby celá vec fungovala, musíme pre súbor vyhradiť dostatočný počet za sebou idúcich blokov, alebo si aj tak udržovať adresár blokov.

Vylepšenie

Ak uveríme sile štatistiky pri preplnení budeme rozdeľovať bloky v poradí. Potom si stačí pamätať len adresu posledného bloku. V takomto prípade treba však znovu ošetrovať preplnenia napr. reťazením v nezaplnených blokoch.

Štatistike možno trochu pomôcť, bloky budeme postupne zmenšovať.

Dotazy na čiastočnú zhodu

Problém: Záznamy obsahujú *n* položiek. Ako najlepšie nájsť všetky vety z danými hodnotami *k* položiek.

- indexy (riedky a hustý prienik)
- k-d stromy
- mnohorozmerné hašovanie

	0	1
0	(0,4) (4,6)	(4,3) (6,1) (6,7)
1	(5,0) (5,6) (7,2)	(1,1)

$h_1 = k_1 \bmod 2$

$h_2 = k_2 \bmod 2$

Zložitosť: $M^{1-k/n}$

Výhodné je použiť distribučné hašovacie funkcie.

Dôležité je udržovať hašovací (adresný) priestor približne v tvare n-rozmernej kocky.

Aj mnohorozmerné hašovanie sa dá dynamizovať.

Hašovacie funkcie zachovávajúce usporiadanie

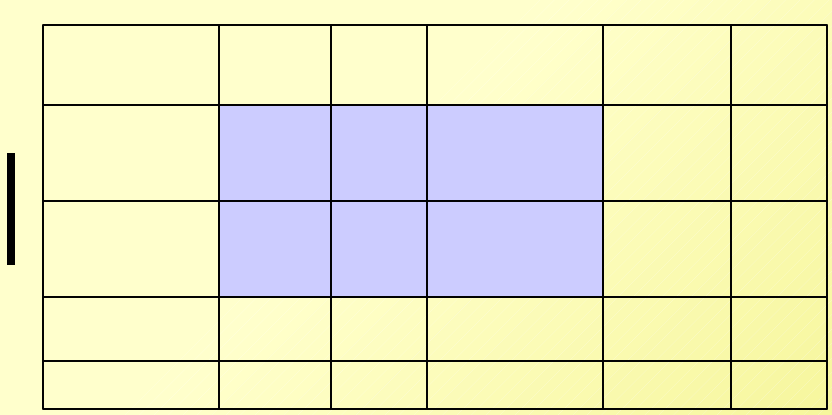
Potrebujeme hašovaciu funkciu zachovájúcu usporiadanie. Nech $K \in$ <*a,b)* a *M* nech je počet blokov, do ktorých budeme hašovať. Nech *f(x)* je rastúca funkcia na <a,b> taká, že *f(a)=0* a *f(b)=1*. Potom $h(K) = \lfloor M \times f(K) \rfloor$ je hašovacia funkcia zachovávajúca usporiadanie. Úplne ideálna je funkcia $\Phi(x) = p(K \leq x)$, pravdepodobnosť, že $K \leq x$. Túto obvykle nepoznáme a nahradzujeme ju lineárnou funkciou $(x - a)/(b - a)$, alebo po kusoch lineárnou funkciou.

Intervalová zhoda - Range queries

Použijeme hašovaciu funkciu zachovájúcu usporiadanie.

Predpokladajme: $K \in < a, b)$ a $h(K) = \frac{(K-a) \times M}{b-a}$

Táto funkcia je dobrá, ale nemusí rozdeľovať, záznamy do blokov rovnomerne. Ak poznáme kumulatívnu distribúciu Φ hodnôt kľúča na intervale $<a, b)$ môžeme vylepšiť hašovaciu funciu: $h(K) = M \times \Phi(K)$



Existujú aj stromy pre podporu intervalových dotazov (Willard).

V grafických dátach quad-tress a oct-trees.

Počet prenesených blokov pri výpočte kartézskeho súčinu.

Parametre:	R_1	R_2	výsledok
počet záznamov	T_1	T_2	$T_1 T_2$
dĺžka záznamu	ℓ_1	ℓ_2	$\ell_1 + \ell_2$
počet blokov	B_1	B_2	$B_1 T_2 + B_2 T_1$
veľkosť bloku	b	b	b

Sú dve možnosti:

1) Aspoň jedna relácia sa vôjde do operačnej pamäti a zostanú aspoň dva bloky, jeden pre druhú reláciu a druhý pre výsledok.
2) Ani jedna relácia sa nevôjde celá do pamäti .

Vlastný výpočet

V prvom prípade je všetko jednoduché: Prenesieme vhodnú reláciu do pamäti. Postupne prinášame bloky druhej relácie, kartézujeme a pamätáme výsledok. Počet prenesených blokov: $B_1(T_2+1) + B_2(T_1+1)$.

Druhý prípad: Nech do pamäti vôjde *M* blokov. Prvú reláciu segmentujeme na $B_1/(M-2)$ segmetov. Pre každý po blokoch prinášame celú druhú reláciu a kartézujeme.

Celkový počet blokov: $B_1(T_2+1)+B_2(T_1+\frac{B_1}{M-2})$

Segmentujeme reláciu s menším počtom blokov.

Triedenie zlučovaním

Algoritmus 1 (mnohocestné zlučovanie)

(1) Prinesieme do pamäti toľko blokov, koľko sa zmestí (M). Utriedíme zapamätame. Toto opakujeme pokiaľ nie je koniec súboru. Čítali sme B blokov, písali sme B blokov.

(2) Rezervujeme si M – 1 vstupných blokov a 1 výstupný blok. Pomocou nich zlučujeme „runy“ na dĺžku M(M – 1). Na prebehnutie celej fázy dve znovu B čítaní a B písaní.

(3) Fázu dve opakujeme, pokiaľ nie je súbor utriedený. Pri k-tom opakovaní vznikajú je dĺžka runu $M(M-1)^k$ a urobili sme $2(k+1)B$ prenesov blokov a máme utriedených $M(M-1)^k b$ viet.

Vzhľadom na to, že M je veľmi veľké, vystačíme obvykle s k = 1 alebo 2.

Triedenie hašovaním

Triedenie hašovaním je vlastne obrátenie algoritmu mnohocestného zlučovania. Na počiatku predpokladáme, že celý súbor je jeden bucket.

Algoritmus 2:

for each bucket **do**
while bucket > A **do** distribute into m buckets;
/* using order preserving hashing*/
for each bucket **do** sort bucket;

Ak zvolíme A = Mb a m = M – 1 je to inverzia algoritmu 1, za predpokladu, že použité hašovacie funkcie sú dokonalé (rovnomerne distribujú všetky buckety). To obvykle neplatí. Pre jednoprocesorový počítač je lepší algoritmus 1. Pre distribuované a viacprocesorové systémy môže byť druhý algoritmus vyhodnejší.

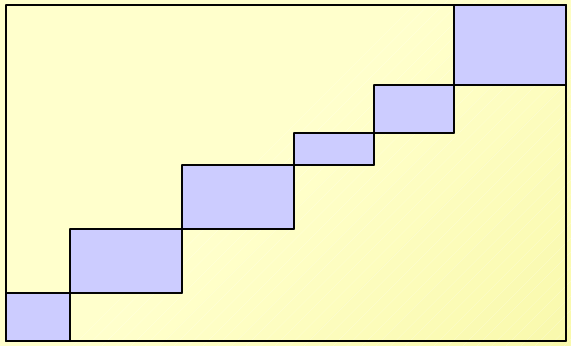
Relačné operácie

Zjednotenie rozdiel, projekcia: jednoduché, odstránenie opakovaných záznamov vyžaduje triedenie resp. je tak zložité ako triedenie.

Možnosti spájania dátových štruktúr (merge-able trees).

Prirodzené spojenie:

- Vložené cykly
- Utriedenie a zlučovanie (merge join)
- hašovaním (hash join)



Pri hašovaní podľa spajacích položiek spájať sa môžu iba záznamy s rovnakými hodnotami hašovacej funkcie. Na tie už môžeme použiť vložené cykly.

Viac o prirodzenom spojení

Pri vložených cykloch je počet vstupných blokov rovnaký ako pri kartézskom súčine. Veľkosť výsledku je však ťažko predikovateľná.
Okrem základných troch spôsobov výpočtu ešte môžeme využívať index. Riedky a hustý index sa chovajú rôzne.
Pr. Hustý index k obom súborom podľa spájacich atribútov. Prienik indexov.
Pr. Použitie jedneho indexu:

For each $r \in R_1$ **do** { X := index (R_2, C, r.C)

for each $s \in X$ **do** output rs }

Množstvo variacii pre spôsob výpočtu prirodzeného spojenia. Vyžaduje netriviálne optimalizácie.

SQL a indexy

Sql norma nepodporuje indexy.

Postgres:

CREATE [UNIQUE] INDEX *name* ON *table* [USING *method*]
({ *column* | (*expression*) } [*opclass*] [, ...])
[WHERE *predicate*]
DROP INDEX *name* [, ...] [CASCADE | RESTRICT]

PostgreSQL provides the index methods B-tree, R-tree, hash, and GiST (stands for Generalized Search Tree). The B-tree index method is an implementation of Lehman-Yao high-concurrency B-trees. The R-tree index method implements standard R-trees using Guttman's quadratic split algorithm. The hash index method is an implementation of Litwin's linear hashing. Users can also define their own index methods, but that is fairly complicated.

Varianty B^+ stromov a lineárne hašovanie.