STU FIIT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Dátové štruktúry a algoritmy

Hashovanie 2

31. 03. 2021

letný semester 2020/2021

prednášajúci: Lukáš Kohútka

Univerzálne hashovanie (universal hashing)

- Základné pozorovanie: Najlepšie ako rozptýliť prvky v tabuľke je **náhodne**, čo však nemôžeme použiť, lebo by sme ich tam nevedeli (opakovane) vyhľadať.
- Takže by sme chceli, aby h (hashovacia funkcia) bola **pseudonáhodná**
- Uvažujme hashovanie reťazením: N prvkov, M vedierok
- Ak **|U| ≥ (N-1)M + 1**, tak pre ľubovoľnú **h** existuje množina **N** kľúčov, ktoré hashujú do rovnakého vedierka
  - Uvažujme opačnú situáciu: Ak by každé vedierko malo najviac **(N-1)** prvkov, tak by kľúčov bolo najviac **(N-1)M**
- Ako teda môže byť to hashovanie vôbec na niečo dobré?
  - Nie je to také zle: pre rozličné typické množiny kľúčov v praxi poznáme dobré hashovacie funkcie

Univerzálne hashovanie (universal hashing)

- Chceli by sme čo najlepší najhorší prípad
- Skúsme randomizovať konštrukciu hashovacej funkcie
  - Funkcia h bude deterministická, ale
  - ak ju vyberieme takto „pravdepodobnostne“, tak pre ľubovoľnú postupnosť insert a search operácií bude očakávateľne dobrá
- Randomizovaný algoritmus H pre konštrukciu hashovacích funkcií h: **U → {1, ..., M}** je univerzálny ak pre každé x ≠ y v U platí:

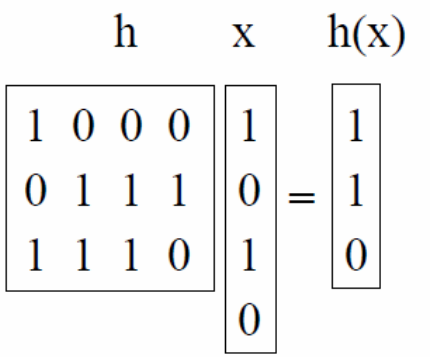
$$\Pr_{h \leftarrow H}[h(x) = h(y)] \leq \frac{1}{M}$$

- Očakávaný počet kolízií kľúča x s inými je N/M.
- H – **trieda univerzálna hashovacích funkcií**

Univerzálne hashovanie – maticová metóda

- Kľúče **u** bitov dlhé, **M = 2^b**
- Zvolíme h ako náhodnú maticu 0/1 veľkosti **b × u,** a definujeme **h(x) = h·x**

Napr.



- Tvrdíme, že pre x ≠ y platí $Pr[h(x) = h(y)] = \frac{1}{M} = \frac{1}{2^b}$
- Násobenie matice h·x: súčet (modulo 2) niektorých stĺpcov v riadku matici, podľa toho, ktoré bity sú 1 v kľúči
- Kľúče **x** a **y** sa líšia v i-tom bite (**x_i = 0, y_i = 1**), i-ty stĺpec vieme určiť **2^b** spôsobmi; každá zmena bitu spôsobí zmenu bitu v **h(y)**, a teda pravdepodobnosť, že **h(x) = h(y)** je $\frac{1}{2^b}$

Perfektné hashovanie (perfect hashing)

- Uvažujme, že množina kľúčov je statická
  - Tabuľka symbolov prekladača
  - Množina súborov na CD
- Vieme nájsť hashovaciu funkciu h, že všetky vyhľadania budú mať konštantný čas? Áno – tzv. **perfektné hashovanie**
- **Priestorová zložitosť O(N²)**
  Zvoľme veľkosť tabuľky $M = N^2$
- Uvažujme, triedu univerzálnu hashovacích funkcií H, náhodne vyberme $h \in H$, pravdepodobnosť kolízie:
  - Počet dvojíc $\binom{N}{2}$, pre dvojicu pravdepodobnosť kolízie je **≤ 1/M**, celková pravdepodobnosť kolízie **≤** $\binom{N}{2}$**/M < 1/2**
- Ak pre zvolenú $h \in H$ máme kolízie, vyberieme znovu :)

Perfektné hashovanie – O(n) metóda

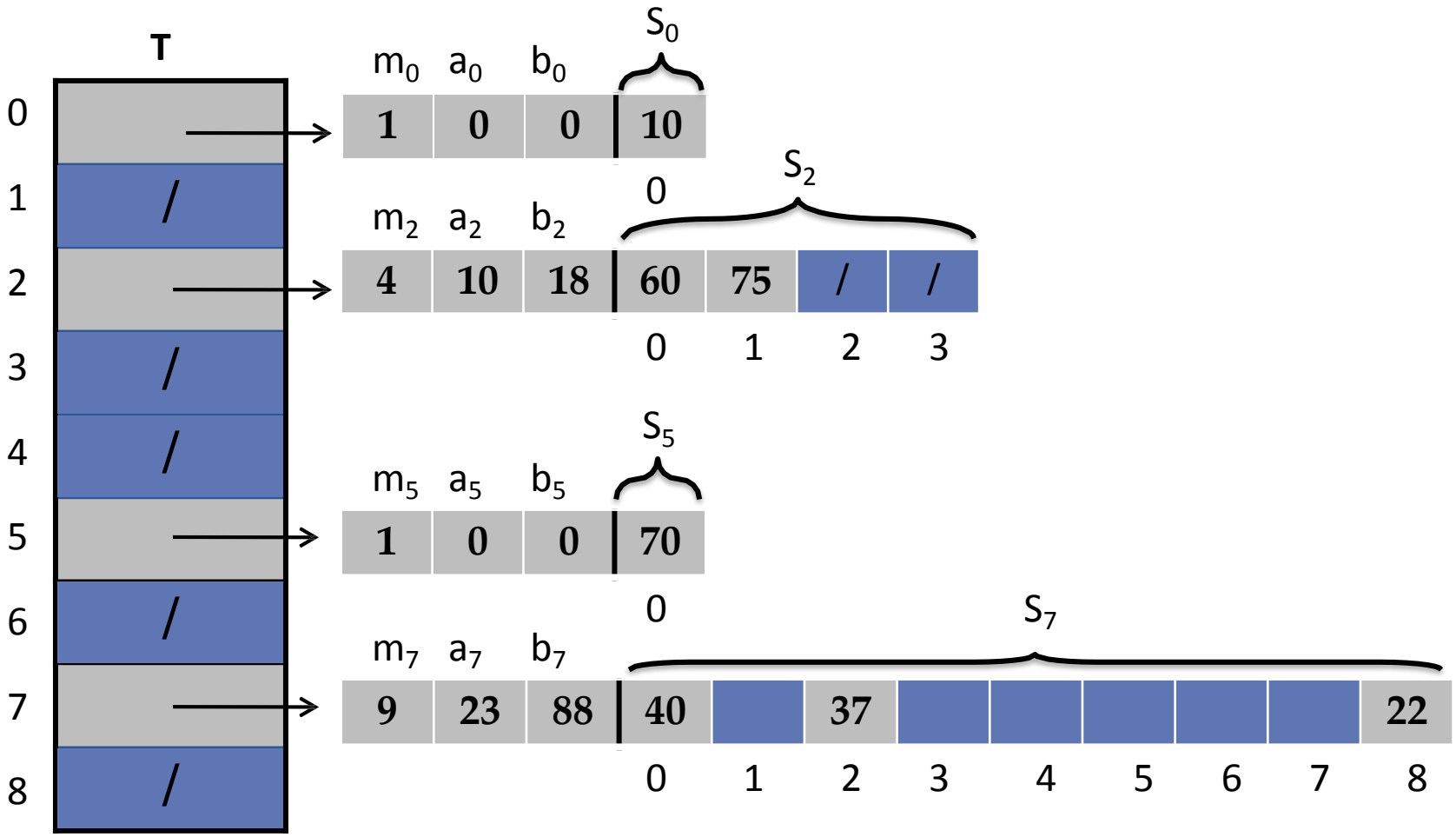
- Zlepšíme priestorovú zložitosť
- **Použijeme dve úrovne hashovacích tabuliek:**
  - Hashovanie na prvej úrovni rozptýli prvky na triedy (môže sa ešte vyskytnúť kolízia)
  - Druhá úroveň – v každom vedierku je ďalšia hashovacia tabuľka (i-te vedierko obsahuje n_i prvkov), ktorá využíva $O(N^2)$ metódu
  - Platí:

$$Pr\left[\sum_i (n_i)^2 > 4N\right] < \frac{1}{2}$$

Skúšame náhodné h, až kým nájdeme takú, že $\sigma_i(ni)^2 < 4N$, a potom nájdeme sekundárme hashovacie funkcie $h_1, h_2, \ldots, h_N$

Perfektné hashovanie - ukážka

- Množina kľúčov **K = {10, 22, 37, 40, 60, 70, 75}**



Zmena veľkosti hashovacej tabuľky

- Pri otvorenej adresácii sme videli, že **faktor naplnenia α značne ovplyvňuje výkonnosť**
- Dôležité je udržiavať α malé
  - Pre otvorenú adresáciu sa odporúča α najviac ½
- Kedy sa α môže zväčšiť?
  - keď sme vložili nejaký prvok
- Keď α presiahne prahovú hodnotu
  - Zväčšíme veľkosť tabuľky
  - **Dôležité: vždy zdvojnásobiť veľkosť** (ale stále prvočíslo)
  - Každý prvok zo starej tabuľky nanovo vložiť (insert) do tabuľky novej veľkosti
- Problém: takéto prehashovanie na väčšiu veľkosť trvá dlho

Zmena veľkosti hashovacej tabuľky (2)

- Jeden insert môže trvať veľmi dlho
  (ak je potrebné zväčšiť tabuľku)
- **Nepoužiteľné ak očakávame rýchle odozvy**
  napr. systémy reálneho času (letová prevádzka)
- Alternatíva, že si novú väčšiu tabuľku začneme vytvárať priebežne skôr ako dosiahneme prahovú hodnotu
  - Vyhľadanie aj v starej aj v novej tabuľke
  - Insert len do novej, pričom pri každom insert-e vložíme aj k prvkov zo starej tabuľky do tabuľky väčšej veľkosti
  - Po určitom čase budú všetky staré prvky v novej tabuľke a môžeme začať budovať ešte väčšiu tabuľku :)

Problém webového cachovania

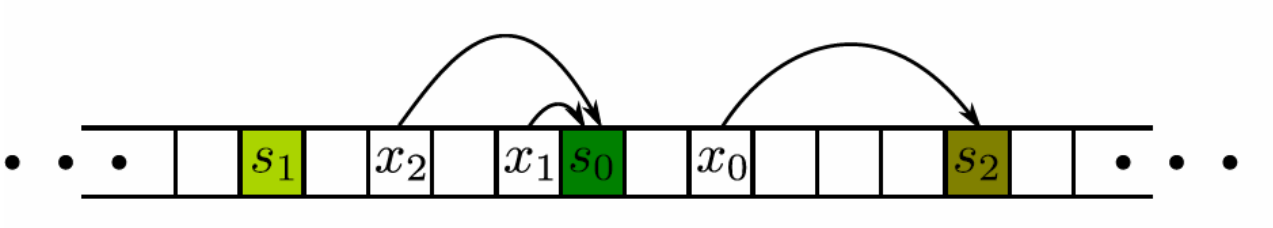
- Používateľ vyžiada stránku (napr. www.google.com)
  - Zbytočne sa opakovane sťahuje zo servera
  - Môžeme si ju odpamätať (do lokálnej cache prehliadača)
  - Pri ďalšom prístupe najskôr pozrieme do cache
  - Rýchlejšia odozva
- Ak bude takáto cache zdieľaná pre väčšiu skupinu používateľov (napr. celú FIIT), tak potom keď ku rovnakej stránke prístupi iný používateľ, môže ísť rovno z takejto zdieľanej cache.
- Takáto cache bude relatívne veľká, musí byť rozdelená na viacero uzlov – serverov
- Pri vyžiadaní stránky: ako zistíme, v ktorom uzle je uložená?

Konzistentné hashovanie (consistent hashing)

- Chceme zobrazenie **URL → cache**
- Máme n uzlov zdieľanej cache
- Použijeme hashovanie ☺
  - **h(url) mod n**
- Záťaž sa mení, a chceme počet uzlov cache meniť:
  - Rozšírenie cache – pridanie uzlu
  - Strata pripojenia servera – odstránenie uzlu
  - Obnovenie pripojenia – pridanie uzlu
- Ak sa n zmení, hodnoty **h(url) mod n** sa všetky zmenia!
- **Konzistentné hashovanie** – pri zmene veľkosti tabuľky zostane väčšina kľúčov na rovnakom mieste

Konzistentné hashovanie (consistent hashing)

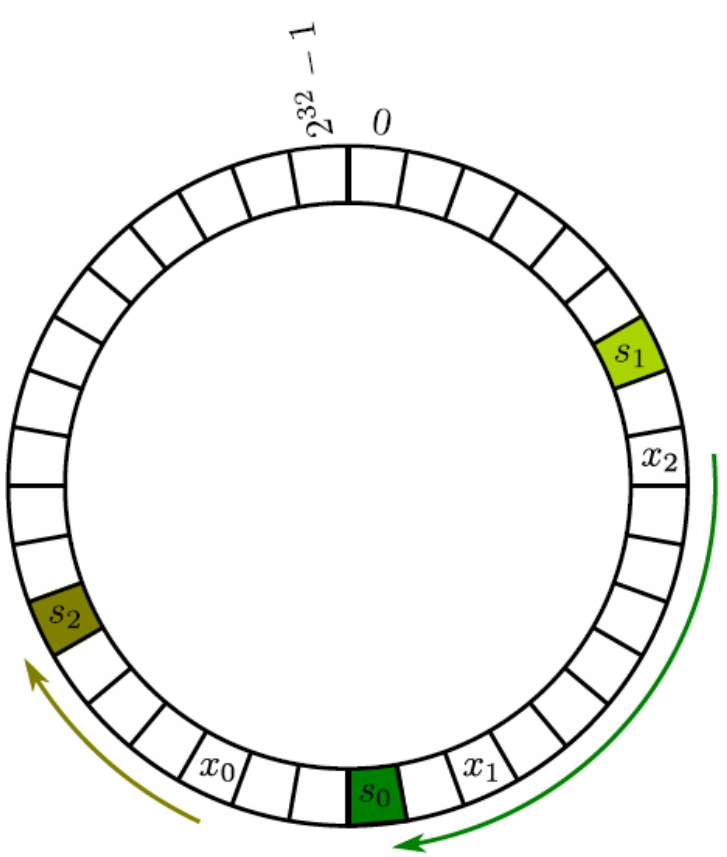
- Hlavná myšlienka: okrem hasovania stránok, budeme do rovnakej tabuľky hashovať aj uzly (servery) cache
- Prvok (stránku) x pridelíme tomu serveru, ktorý v tabuľke nasleduje najbližšie (doprava):



Napr. prvok x_0 je v s_2, a prvky x_1 a x_2 su v s_0

Konzistentné hashovanie (consistent hashing)

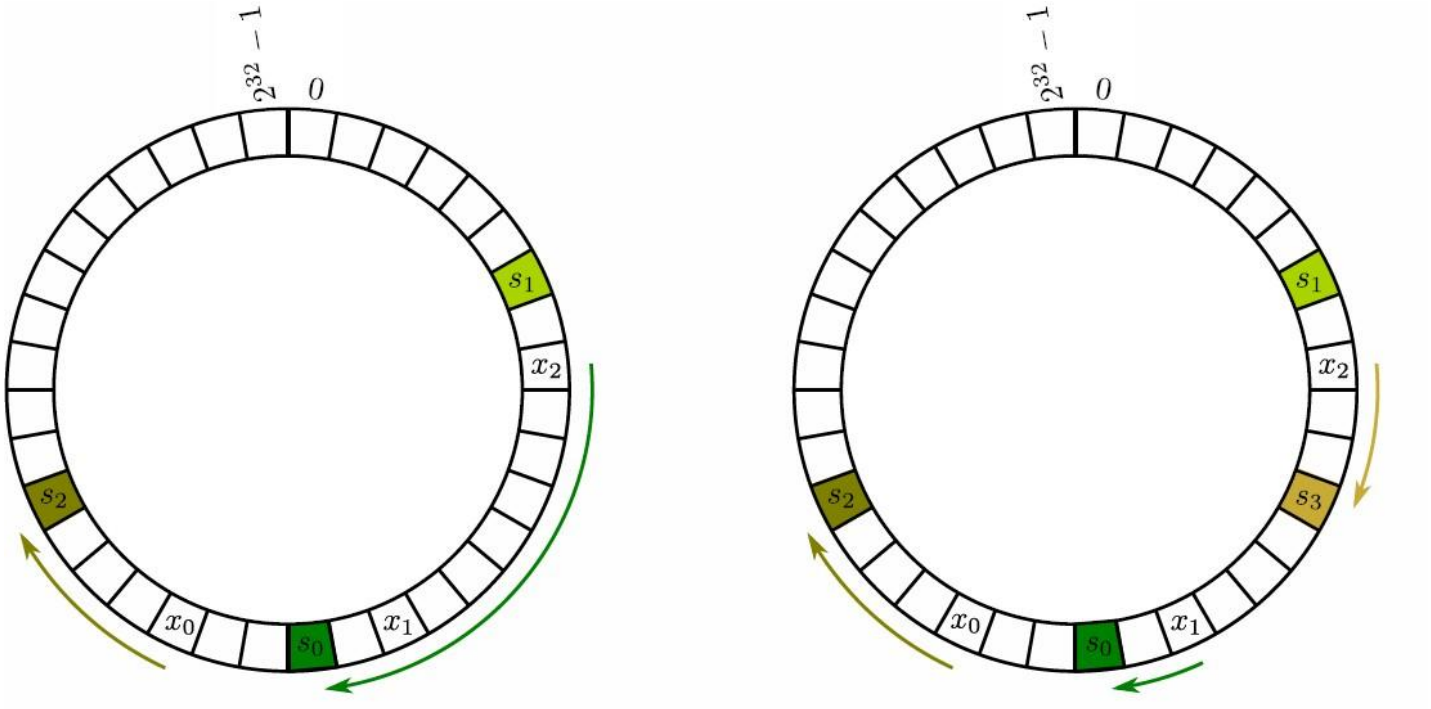
- Zobrazené na koliečku:



Stránku uložíme v uzle najbližšie v smere hodinových ručičiek

Konzistentné hashovanie (consistent hashing)

- Predpokladajúc rovnomerné rozptýlenie
  - Zaťaženie jedného uzlu = 1/n stránok
- Keď pridáme nový server s, **presunieme len prvky uložené v s:**



Konzistentné hashovanie (consistent hashing)

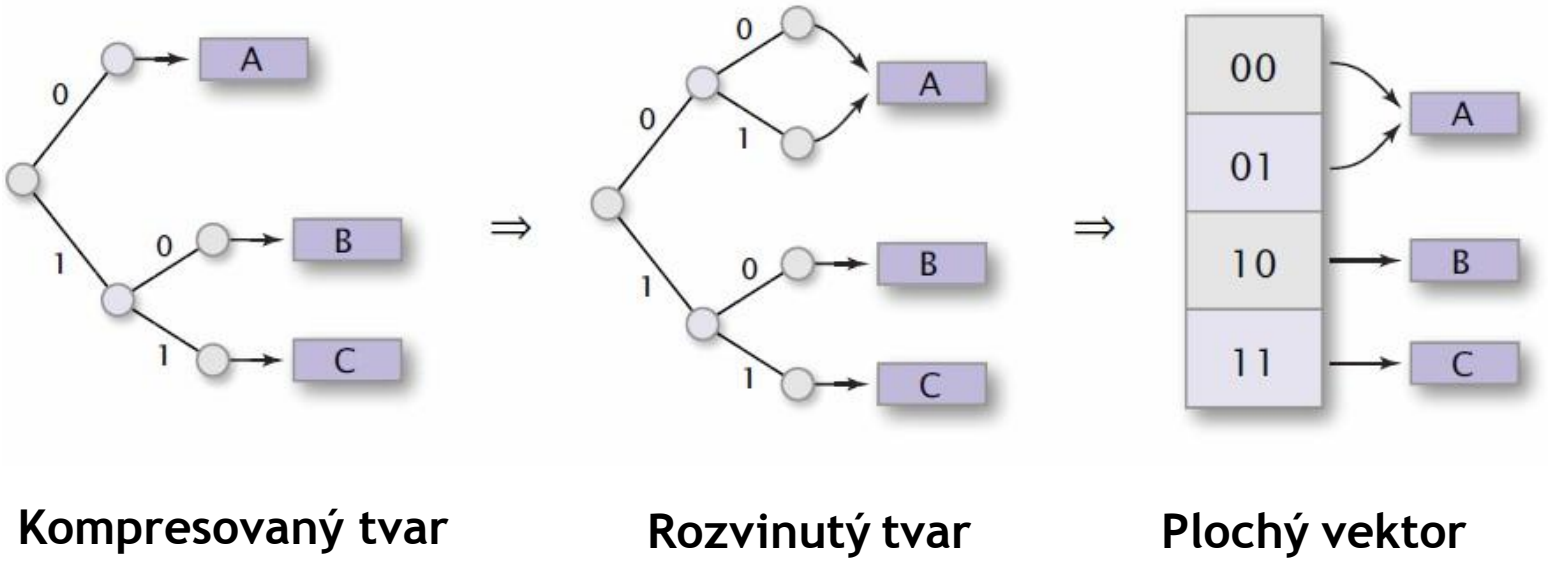
- Efektívna implementácia vyhľadania:
  - Pre daný h(x) potrebujeme zistiť uzol, do ktorého ho treba priradiť: **potrebujeme nájsť najbližší taký uzol s, ktorého h(s) ≥ h(x) …**
  - Operácia successor(x)
  - **Efektívna implementácia binárnym vyhľadávacím stromom (napr. červeno-čiernym alebo AVL)**
- Praktická poznámka
  - **Rozdiely v záťaži môžu kolísať** (ak by aj umiestnenie servera s bolo náhodné, tak nebude to vyvážené)
  - **Lepšie je každý uzol reprezentovať viacerými bodmi** (napr. k náhodných hashovacích funkcií, dobré k ≈ log n)

História konzistentného hashovania

- 1997 – vedecký článok o konzistentnom hashovaní
- 1998 – založenie Akamai
- 31. 3. 1999 – trailer **Star Wars: The Phantom Menace** je umiestnený online, pričom Apple je exkluzívny oficiálny distribútor, okamžite sa znefunkčnili stránky apple.com kvôli preťaženiu... Poväčšinu dňa jediný dostupný spôsob ako ho pozrieť je neautorizovaná kópia na web cache od Akamai!
- 1. 4. 1999 – Steve Job si všimol, ako to Akamai zvládlo, volá šéfovi Akamai: Paul Sagan, pričom Sagan okamžite zvesí telefón, lebo to považuje za prvo aprílový žartík ...
- 2001+ Konzistentné hashovanie sa začína používať v P2P sieťach tretej generácie (Napster bola prvá generácia :)

Rozšíriteľné hashovanie (extendible hashing)

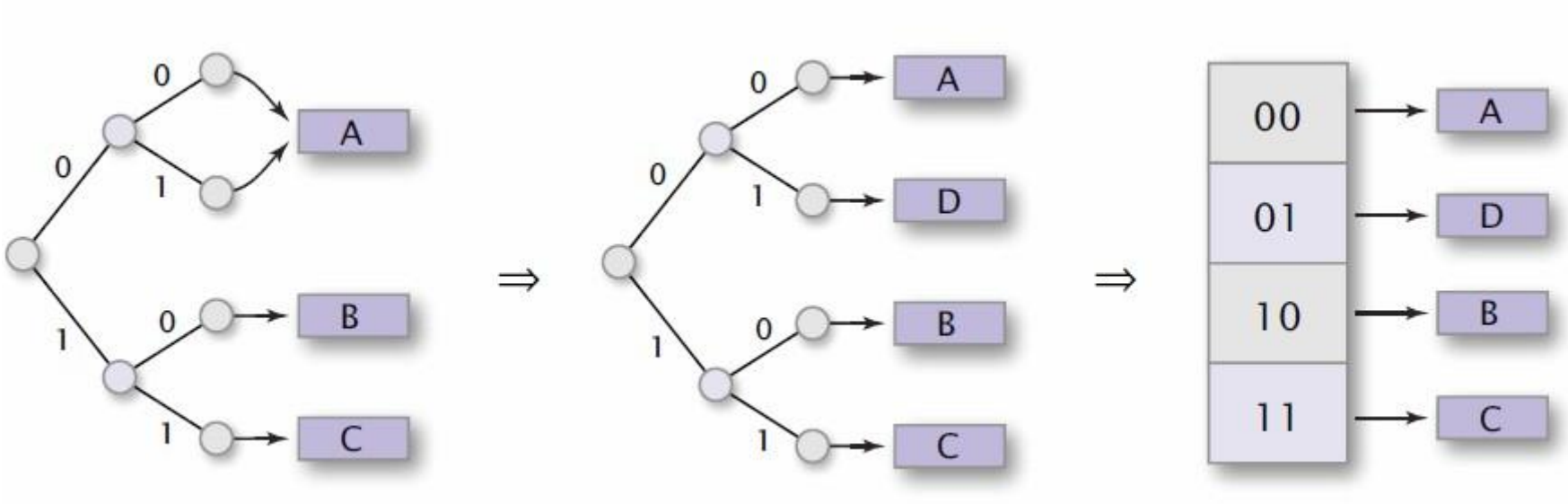
- Keď zväčšovanie tabuľky je výpočtovo náročné
  - Napr. súborový systém
- Binárny trie na vyhľadanie vedierka (kľúč je hash ako binárny reťazec), reprezentovaný v plochom poli



Kompresovaný tvar **Rozvinutý tvar** **Plochý vektor**

Rozšíriteľné hashovanie (extendible hashing)

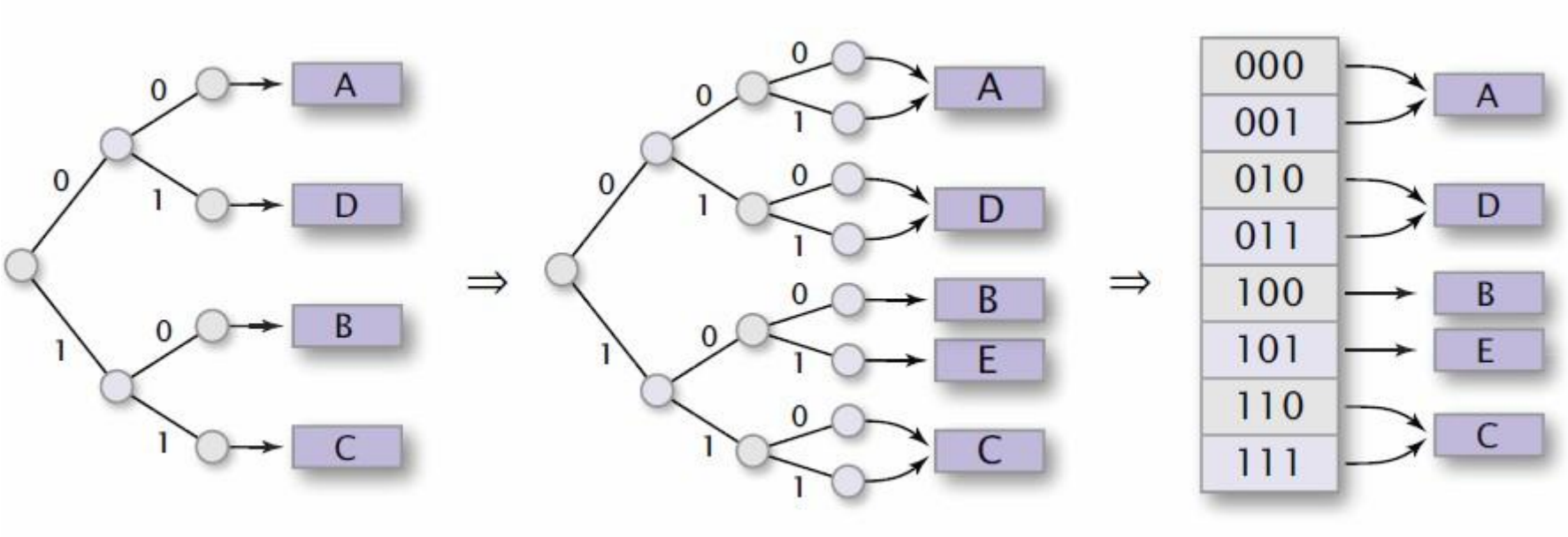
- Pridanie vedierka:



Kompresovaný tvar **Rozvinutý tvar** **Plochý vektor**

Rozšíriteľné hashovanie (extendible hashing)

- Rozšírenie stromu (zvýšenie výšky)



Kompresovaný tvar **Rozvinutý tvar** **Plochý vektor**

- Pridáme rozlišujúci bit – všetko zostane zachované
- Pridáme nové vedierko

Reťazenie – varianty

- Vo vedierku môže byť aj binárny vyhľadávací strom, ale voľbou dobrej hashovacej funkcie sú vedierka väčšinou prázdne, takže prakticky postačuje spájaný zoznam
- Obava: čo ak bude hashovanie pomalé ak sú „divné“ vzory prístupov: **dynamicky optimálne hashovanie**
  - Pri každom vyhľadaní vo vedierku hodnotu posunieme na začiatok zoznamu (podobne ako v prípade splay stromov)
  - Zložitosť: **2x viac operácií ako optimálne** (tzv. 2-competitive)

Reťazenie – varianty (2)

- **Dvojsmerné reťazenie** (two-way chaining)
  - Každý prvok prislúcha dvom vedierkam (hash funkcie h a g)
  - Pri inserte sa vloží do toho z nich, ktorá má menej prvkov
- Spôsoby ako sa vyhnúť použitiu smerníkov
  - Najskôr použijem otvorené adresovanie, a po prekročení (zvolenej) kapacity prejsť na reťazenie
  - každé vedierko má malú kapacitu (napr. 4)
  - Príp. postupnosť hashovacích tabuliek:
    najskôr skúsiť vložiť do prvej, ak je miesto obsadené, tak skúsiť vložiť do druhej, atď ...

Otvorená adresácia – varianty

- Ak vkladáme a nastane kolízia, tak máme dva prvky (starý, nový) a niektorý z nich môže zostať a ostatný musíme posunúť niekam inam
- Väčšinou sa posúva nový prvok
- Môžeme však posunúť radšej starý prvok
  - Kukučkové hashovanie (cuckoo hashing)
  - Last-come-first-served hashing
  - Robinhood hashing

- Split sequence hashing – posun zavisí od kľúča na aktuálnej pozícii, istým spôsobom to je ako BVS pri reťazení – rozhodovanie v strome

Aplikácia: Hashovanie stavu herného sveta

- Máme nejakú doskovú hru – figúrky / pozície
- Ako reprezentovať tento herný svet?
- Ako zahashovať takúto pozíciu, aby sme vedeli pri prehľadávaní rôznych pozícií rýchlo vyhodnotiť, či sme v pozícii už predtým boli (a kde sa nachádzajú predvypočítané údaje) alebo ešte neboli.
- **Zobristovo hashovanie**

Zobristovo hashovanie

- Náhodne vygenerujeme bitové reťazce pre každý možný prvok v hre
- Napr. v šachu:
  - figúrka × pozícia
  - Kráľ, ktorý ešte môže spraviť rošádu
  - Pešiak branie mimochodom (en passant)
  - …

- Hashovacia hodnota je XOR (exkluzívny logický súčet) bitových reťazcov prvkov na hracej ploche
- Dokážeme ľahko realizovať zmenu z hashovacej hodnoty pozície do hashovacích hodnôt pozícií, ktoré vzniknú jedným ťahom na hracej ploche
- Má dobré teoretické vlastnosti

Bezpečnostné aspekty hashovania

- Je predpoklad rovnomerného rozptýlenia dôležitý?
  - Áno, keď potrebujeme rýchlu odozvu – jadrové reaktory, riadenie letovej prevádzky (t.j. real-time systémy)
- Denial-of-service útoky založené na hashovaní
  - Keď útočník vie, akú používaš hashovaciu funkciu, môže ti podstrčiť veľké množstvo údajov, ktoré hashujú do rovnakého vedierka, a teda je veľa kolízií
  - zložitosť operácií $O(n)$ namiesto $O(1)$
- Pri hashovaní hesiel nechceme rýchlu hash funkciu
  - Rýchle funkcie, ktoré vyžadujú málo pamäti, sa dajú rýchlo počítať vektorovo na GPU – aj niekoľko **100M/s**
  - Chceme pomalú hash funkciu, ktorá vyžaduje veľa pamäti napr. **bcrypt**, **scrypt** – na GPU sa dá počítať do **10/s**

Aplikácia: Vyhľadávanie reťazcov

- Vyhľadávanie v dokumentoch (Word, grep, ...)
- Bioinformatika (DNA)

- Problém:
  - Daný je (dlhý) text N znakov **`text[0..N-1]`**
  - Hľadáme vzor dlhý M znakov **`pattern[0..M-1]`**

- Naivný prístup hrubou silou zložitosť O(NM)
  - Pre každý začiatok (N-M možných)
  - Vyskúšam po písmenách porovnať celú vzorku

Rabin-karpov algoritmus

- Čo keby sme porovnávali len na miestach, kde máme nejakú indíciu, že by výskyt vzorky mohol byť?
- Kolízia v hashovaní – relatívne veľká istota, či na nejakej pozícii **môže byť** výskyt vzorky
- Algoritmus:
  - Určíme hashovaciu hodnotu hľadanej vzorky P
  - Pre každý podreťazec dĺžky M v dlhom texte T si vypočítame hashovaciu hodnotu
  - Ak sa hodnoty rovnajú – máme „kolíziu“, resp. je veľká šanca, že sme našli výskyt vzorky v texte
  - Porovnáme znak po znaku

Rabin-karpov algoritmus (2)

- Kľúč k efektívnej implementácii:
  Pre každý podreťazec dĺžky M v dlhom texte T si vypočítame hashovaciu hodnotu
- Ako počítať efektívne hashovaciu hodnotu?
- Po písmenkách, pri posune o jeden znak ďalej, treba „odstrániť“ z hashovacej hodnoty ľavé písmeno a pridať do hashovacej hodnoty pravé písmenko – tzv. **rolujúci hash:**
  - Jednoduchý príklad rolujúceho hashu: súčet ASCII znakov
  - Úprava – posun o jedno písmenko
    `s[i+1..i+m] = s[i..i+m-1] - s[i] + s[i+m]`
  - Lepšia implementácia polynomiálnou akumuláciou

STU FIIT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Dátové štruktúry a algoritmy

Ďakujem za pozornosť