Regionálne kolo ACM v Záhrebe


PBZ
PARTNER BANKA

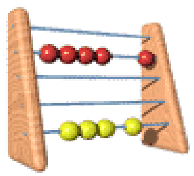
International Collegiate Programming Contest

Zúčastnite sa súťaže ACM ICPC

www.fiit.stuba.sk/acm

acm International Collegiate Programming Contest

Lokálne kolo programátorskej súťaže na STU v rámci



CTU Open Contest

27. - 28. 10. 2017

Pošli registračný e-mail na

acm.icpc@fiit.stuba.sk

Nadácia pre rozvoj informatiky

eset

www.fiit.stuba.sk/acm

GRATEX INTERNATIONAL

STU FIIT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Dátové štruktúry a algoritmy

Hashovanie

17. 10. 2017

zimný semester 2017/2018

Opakovanie – Problém vyhľadávania

- Vstup:
  - Postupnosť: $a_1, a_2, a_3 ... a_n$
    $k(a_i)$ označíme kľúč k_i prvku a_i
  - Hľadaný kľúč x
  - Čo sú kľúče?

    Definičný obor D – reťazce, reálne čísla, dvojice celých čísel, ...
  - Relácia = (rovnosti) – relácia ekvivalencie nad D
- Výstup:
  - Index res ∈ {1,2,…,n} takého prvku, že $k(a_{res}) = x$,
    alebo 0 ak taký prvok neexistuje.

Uvažujme špeciálny prípad slovníka

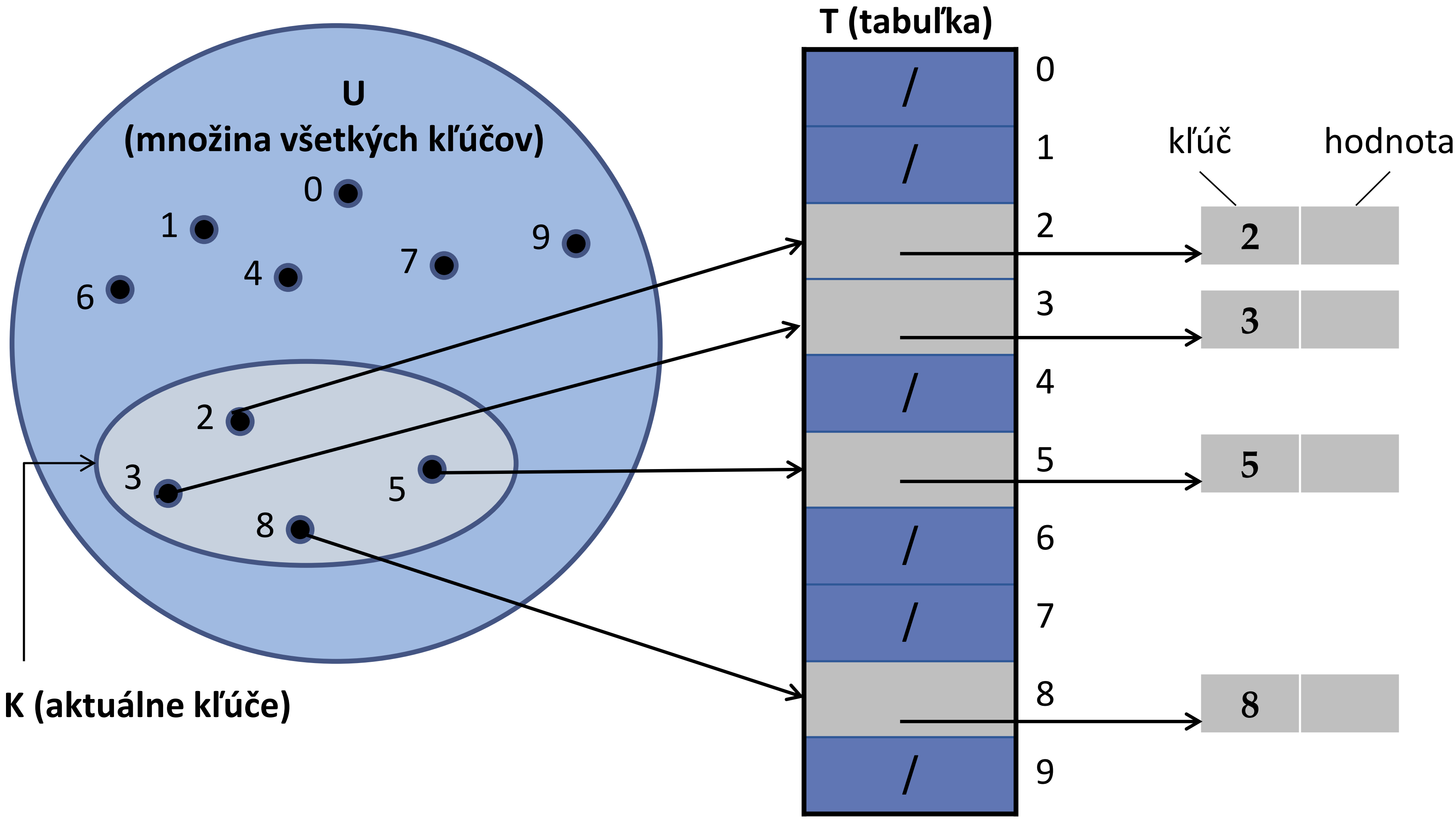
- Kľúče $k(a_i)$ nech sú rôzne čísla od 0 do n-1:
  Napr. (kľúč, hodnota):
  (0,"Peter"), (1,"Milan"), ..., (n-1,"Katka")

- Najefektívnejšia implementácia?
- Poľom/vektorom dĺžky n:

Peter	Milan			…		Katka
0	**1**	**2**		**…**		**n-1**

- Vyhľadanie kľúča x?
  - Pristúpiť k x-tému prvku $\mathbf{a_x}$, tam sa (ne)nachádza hľadaný prvok
- **Optimálna zložitosť všetkých operácií** (insert, search, delete): **O(1)**

Tabuľka s priamym prístupom



Hashovanie

- Zovšeobecnený prístup, keď:
  1. **Kľúče nemusia byť rôzne**
  2. **Rozsah (univerzum) kľúčov môže byť veľký** (v porovnaní s počtom prvkov)
  3. **Kľúče nemusia byť celé čísla**

- Očakávaná zložitosť všetkých operácií (insert, search, delete): **O(1)**
- Pamäťová zložitosť: **O(n)**

Základná myšlienka hashovania

- Hashovacou funkciou **h** zobraziť kľúč **x** do rozsahu indexov poľa:

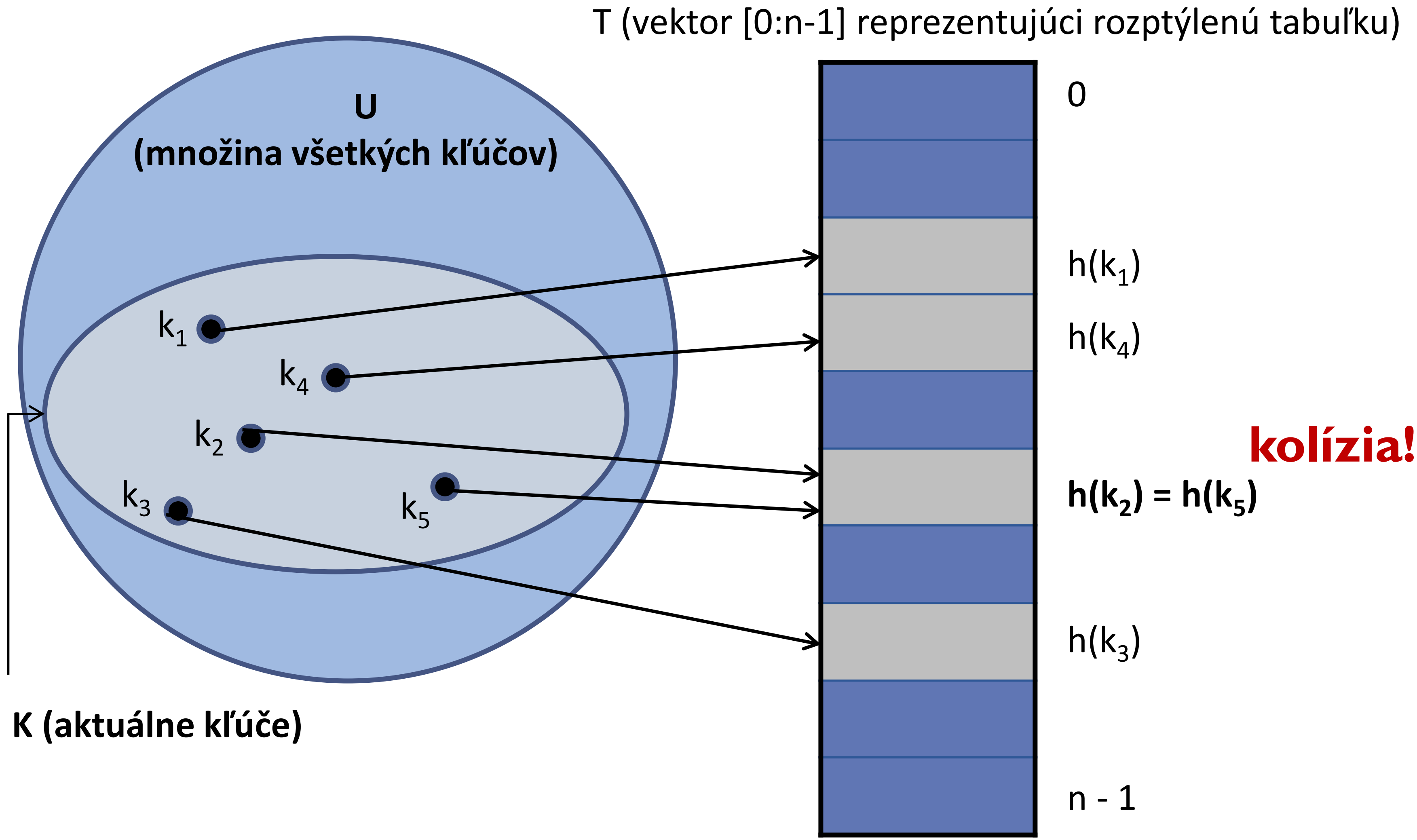
- Vložiť prvok s kľúčom **x** na index poľa **h(x)**:

  h(Milan) = 2

		Milan		…		
0	1	2		…		n-1

- Uvažujme, že chcem vložiť ďalší prvok Peter, ale h(Peter) = 2 je obsadené, tzv. **kolízia**

Hashovanie – kolízia



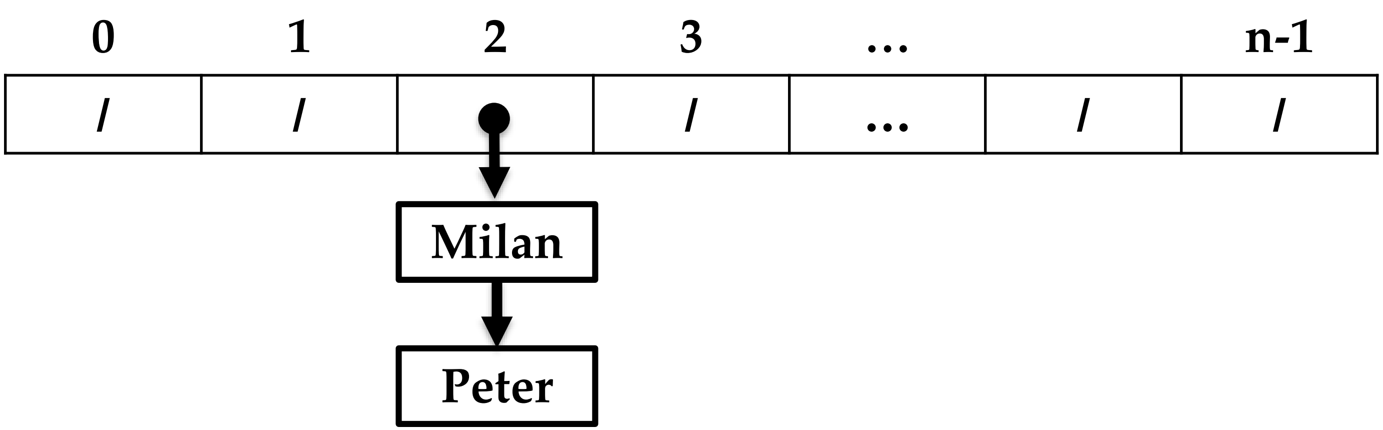
Spôsoby riešenia kolízií

- **Synonymá** – prvky, ktoré majú rôzne kľúče $\mathbf{x_1} \neq \mathbf{x_2}$, ale rovnaké hashovacie hodnoty $\mathbf{h(x_1) = h(x_2)}$
- Vkladám prvok **x**, ale miesto **h(x)** je už obsadené

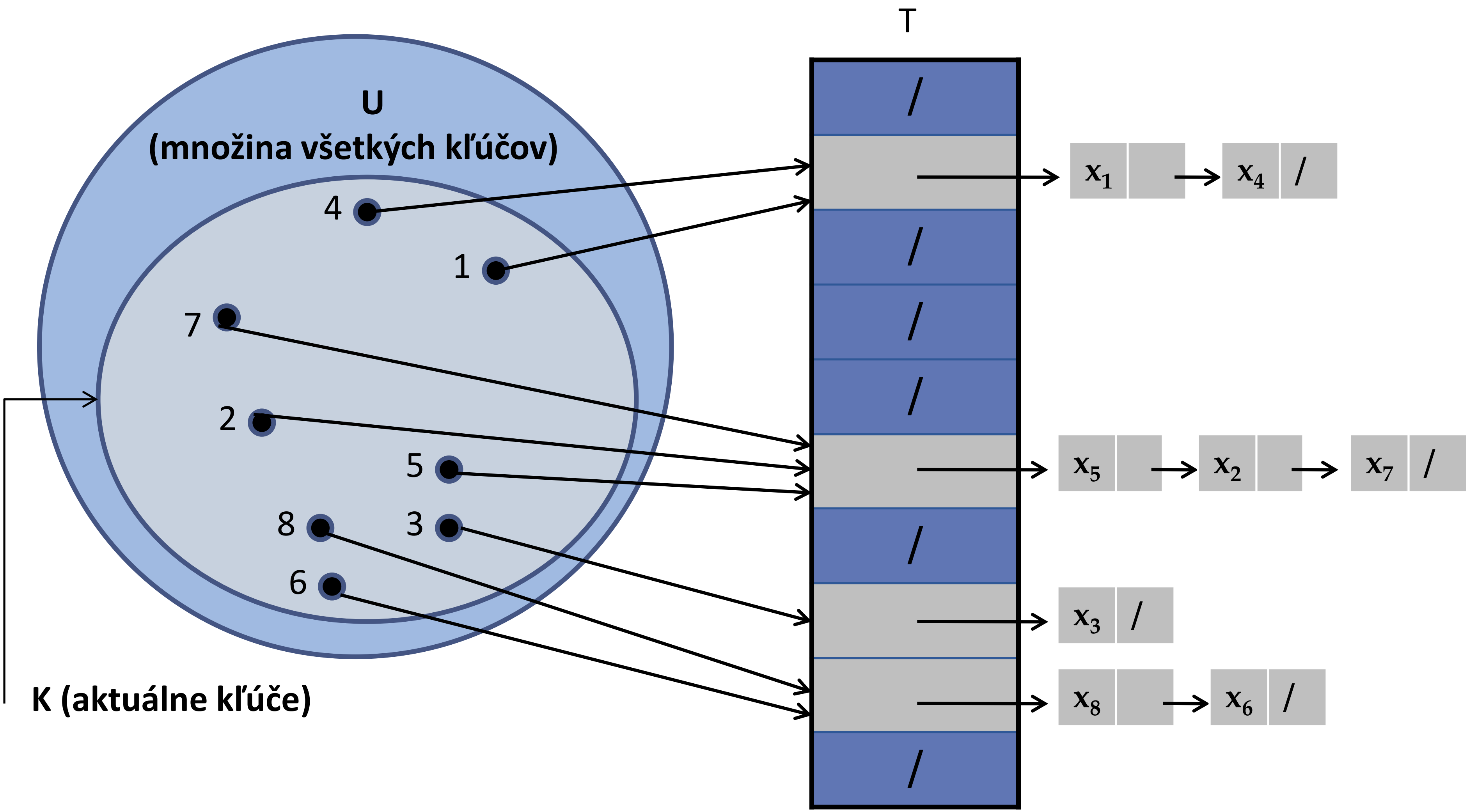
- Existujú dva spôsoby riešenia kolízií:
  - **Reťazenie (chaining)** – umožníme, aby v každom mieste tabuľky mohlo byť aj viacero prvkov (napr. v spájanom zozname)
  - **Otvorené adresovanie (open addressing)** – v každom mieste tabuľky môže byť nanajvýš jeden prvok (musíme nájsť nejaké alternatívne umiestnenie prvkov, ktoré majú rovnakú h(x) hodnotu)

Reťazenie (chaining)

- **V políčku tabuľky môže byť viac prvkov** (dynamická množina – sekundárna dátová štruktúra)
- Políčko tabuľky nazývame **vedierko (bucket)**
  - zvyčajne spájaný zoznam
  - môže byť aj iné, napr. binárny vyhľadávací strom (usporiadaný podľa nejakého sekundárneho kľúča)
- h(Milan) = 2, h(Peter) = 2



Hashovacia tabuľka (kolízie zreťazením)



Reťazenie – Implementácia a analýza

- insert(T, x): Vlož x do vedierka T[h(k(x))], ak tam nie je.
- delete(T, x): Odstráň x z vedierka T[h(k(x))]
- search (T, x): Vyhľadaj x vo vedierku T[h(k(x))]

- Odhad zložitosti:
- Uvažujme, že v tabuľke je N prvkov v M vedierkach
  faktor naplnenia $\alpha = N/M$
  (priemerný počet prvkov vo vedierku)
- Čas potrebný pre výpočet h(x): **O(1)**
- Očakávaný čas na vyhľadanie prvku vo vedierku: **O(α)**
  - Ak je počet vedierok úmerný počtu prvkov N=O(M):
    $\alpha = N/M = O(M)/M = O(1)$

Otvorené adresovanie (open adressing)

- **V políčku tabuľky môže byť najviac jeden prvok**
- Rôzne algoritmy sa líšia spôsobmi, ako prvky s rovnakou **h(x)** umiestnime tak, aby sme ich neskôr vedeli vyhľadať
- Najjednoduchší prístup je prvok umiestniť na najbližšie voľné miesto v tabuľke:
- h(Milan) = 2, h(Peter) = 2

		Milan	**Peter**	**…**		
0	**1**	**2**	**3**	**…**		**n-1**

Otvorené adresovanie (open adressing)

- Postupnosť skúšaných miest závisí od kľúča, t.j. rozptylová funkcia dostane ďalší parameter (i-ty pokus)

$$h: \text{kľúč} \times \{0, 1, ..., N-1\} \rightarrow \{0, 1, ..., N-1\}$$

- Pre kľúč **x** je postupnosť skúšaných indexov:
h(x,0), h(x,1), ..., h(x,N-1)
(mala by byť permutáciou 0,1,...,N-1)

- Dva hlavné prístupy k skúšaniu:
Lineárne skúšanie: **h(x,i) = (h(x) + i) mod N**
Dvojité rozptýlenie: **h(x,i) = (h(x) + i · g(x)) mod N**

Lineárne skúšanie (linear probing) – search

- Systematicky sa prehľadáva postupnosť indexov od **h(x)**:

```
search(T table, x key)
{
      i ← 0
      repeat j ← h(x,i)
            if T[j] = x
                  then return j
            else
                  i ← i + 1
      until T[j] = NIL or i = n
      return NIL
}
```

Lineárne skúšanie (linear probing) – insert

- Systematicky sa skúša nájsť prázdne miesto:

```
insert(T table, x key)
{
      i ← 0
      repeat j ← h(x,i)
            if T[j] = NIL
                  then T[j] ← x
                  return j
            else i ← i + 1
      until i = n

      error “overflow”
}
```

Lineárne skúšanie (linear probing) – delete

- **insert(Milan)** – h(Milan,0) = 2
- **insert(Peter)** – h(Peter,0) = 2, h(Peter,1) = 3

		Milan	**Peter**	**…**		
0	**1**	**2**	**3**	**…**		**n-1**

- **delete(Milan) ?** Obyčajné odstránenie nefunguje.

			Peter	**…**		
0	**1**	**2**	**3**	**…**		**n-1**

- **Zlyhá search(Peter)** – lebo skončí na indexe 2, ale Peter sa nachádza až na indexe 3.

Lineárne skúšanie (linear probing) – delete

- **insert(Milan)** – h(Milan,0) = 2
- **insert(Peter)** – h(Peter,0) = 2, h(Peter,1) = 3

		Milan	**Peter**	**…**		
0	**1**	**2**	**3**	**…**		**n-1**

- **delete(Milan) ?** Použijeme špeciálny symbol (deleted).

		deleted	**Peter**	**…**		
0	**1**	**2**	**3**	**…**		**n-1**

- **Nutné upraviť implementáciu insert a search!**
- **Nevýhoda lineárneho skúšania**: prvky sa zoskupujú do súvislých obsadených postupností – tzv. **strapcov / klastrov čo významne spomaľuje vykonávanie operácií...**

Lineárne skúšanie (linear probing) – analýza

- Jednoduchá implementácia
- **Nevýhoda lineárneho skúšania**: prvky sa zoskupujú do súvislých obsadených postupností – tzv. **strapcov (klastrov), čo významne spomaľuje vykonávanie operácií...**
- Vzniká tzv. **primárne klastrovanie** – dlhé strapce zvyšujú priemerný čas vyhľadania
  - Prázdne miesto, pred ktorým je i obsadených miest sa naplní pri ďalšom inserte s pravdepodobnosťou (i+1)/m
  - Dlhšie strapce sa ešte predlžujú a priemerný čas vyhľadania sa ešte zvyšuje

Ako zlepšiť situáciu s dlhými súvislými blokmi obsadených miest v hashovacej tabuľke?

Dvojité rozptýlenie (double hashing)

- **Posun vypočítame druhou hash funkciou g**

Dvojité rozptýlenie: $\mathbf{h(x,i) = (h(x) + i \cdot g(x)) \bmod N}$

Počiatočná pozícia h(x,0): $\mathbf{h(x) = x \bmod N}$

Posun: $\mathbf{g(x) = 1+(x \bmod N')}$

- Ak D=NSD(N,N') > 1, tak prejdeme len 1/D zo všetkých indexov, dobré voľby sú:

N prvočíslo **N' o kúsok menšie**	**alebo**	**N mocnina dvoch** **N' nepárne**

Ideálny prípad – rovnomerné rozptýlenie

- Chceli by sme, aby postupnosť skúšaných indexov **h(x,0), h(x,1), ..., h(x,N-1)** bola permutácia
- Ak chceme naozaj **rovnomerné rozptýlenie** (**uniform hashing**) potrebujeme, aby sme takto vedeli vytvoriť všetkých N! možných permutácií
- Pravé rovnomerné rozptýlenie je ťažko zrealizovať
  - existujúce prístupy sú aproximácie, pretože nedokážu vytvoriť požadovaných N! možných postupností skúšania indexov

Lineárne skúšanie vs. dvojité rozptýlenie

- Koľko rôznych postupností skúšania indexov dokáže pre N kľúčov vytvoriť:
  - **Lineárne skúšanie? N**
    (začiatok postupnosti index h(x,0) plne určuje celú postupnosť N indexov; je práve N rôznych začiatkov)
  - **Dvojité rozptýlenie? N^2**
    (postupnosť skúšaných indexov závisí dvoma spôsobmi od kľúču x, keďže h(x) a g(x) môžu byť rôzne; N×N' možností)
- Dvojité rozptýlenie je teda rovnomernejšie
  - V tabuľke budú viac „rozptýlenejšie prvky“
  - Kratší čas potrebný na vyhľadanie
  - V praxi sa blíži k ideálnemu rovnomernému rozptýleniu

Lineárne skúšanie vs. dvojité rozptýlenie

- Zložitosť insert a search závisí od veľkosti strapca
- Triviálna analýza: priemerná veľkosť strapca α = N/M
- Najhorší prípad: všetky prvky budú mať rovnakú hashovaciu hodnotu – budú v rovnakom strapci
- Podrobnejšia analýza:

Rovnomerné	**Lineárne skúšanie**	**Dvojité rozptýlenie**
insert: $\frac{1}{(1-\alpha)}$	**insert:** $\frac{1}{2}\left(1+\frac{1}{(1+\alpha)^2}\right)$	**insert:** $\frac{1}{(1-\alpha)}$
search: $\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$	**search:** $\frac{1}{2}\left(1+\frac{1}{(1+\alpha)}\right)$	**search:** $\frac{1}{\alpha}\ln(1+\alpha)$

- Pre veľké M ⇒ veľa prázdnych miest v tabuľke
- Pre malé M ⇒ strapce sa prelínajú
- Dobré je udržiavať **α < 0.5**

Lineárne skúšanie vs. dvojité rozptýlenie

- V praxi (experimentálne meranie)

Očakávaný počet pokusov		Faktor naplnenie α			
		50%	**66%**	**75%**	**90%**
Lineárne skúšanie	**search**	1.5	2.0	3.0	5.5
	insert	2.5	5.0	8.5	**55.5**
Dvojité rozptýlenie	**search**	1.4	1.6	1.8	2.6
	insert	1.5	2.0	3.0	**5.5**

- Dvojité rozptýlenie funguje dobre aj pri väčších α

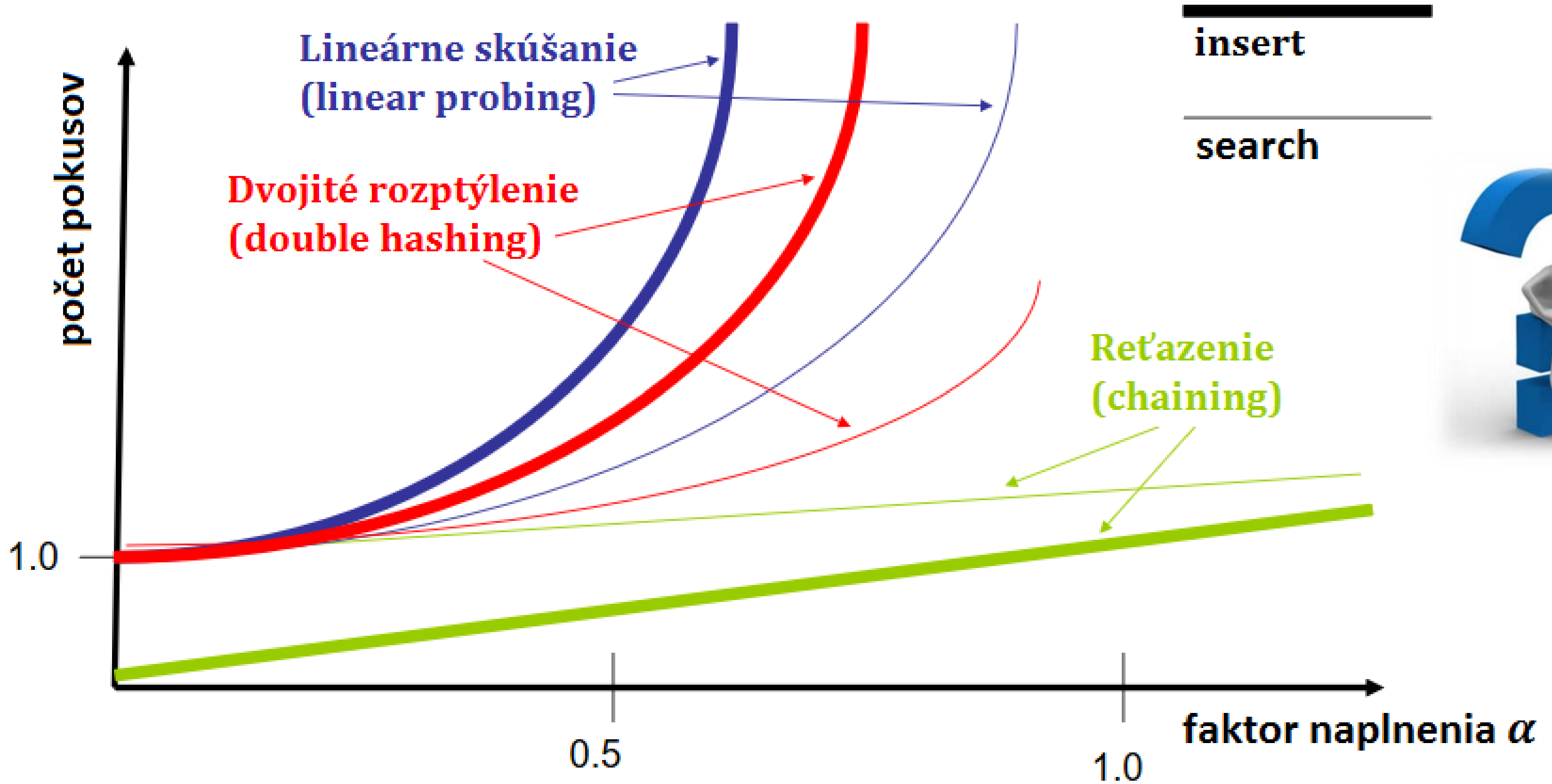
Reťazenie vs. otvorené adresovanie

- **Reťazenie:**
  - **Nutný priestor navyše pre smerníky – zlá lokalita v pamäti**
  - Počet prvkov v tabuľke nie je obmedzený ($\alpha > 1$)
  - Lineárne klesá výkonnosť pri zväčšujúcom sa α

- **Otvorené adresovanie:**
  - Prvky sú priamo v tabuľke – nie sú nutné smerníky navyše
    - **Obmedzený celkový počet prvkov ($\alpha \leq 1$)**
    - Rovnaké množstvo pamäti môže mať väčšiu tabuľku ako pri zreťazení
    - Dobrá lokalita v pamäti – výhodné pre cachovanie prístupov do pamäti
  - Faktor α **značne ovplyvňuje výkonnosť**
    - Pri nízkych $\alpha < 0.5$ rýchlejšie ako reťazenie (netreba prechádzať smerníky)
    - **Výrazné spomalenie pre α blízke 1**
  - **Nepodporuje delete (odstránenie)!**
    - Resp. použitie **deleted** symbolu výrazne spomaľuje vyhľadanie aj pri nízkych α

Reťazenie vs. otvorené adresovanie

insert = očakávaný prípad je najhorší prípad vyhľadania
search = očakávaný-priemerný prípad vyhľadania



Už máme celkom dobrú predstavu ako by sme riešili kolízie, pozrime sa teraz na hashovaciu funkciu ako zdroj rovnomernosti ...

Hashovacia (rozptylová) funkcia

- Výpočet by mal byť rýchly
- Dobrá hashovacia funkcia: minimalizuje počet kolízií
  - Rovnomerne rozptyľuje prvky do celej tabuľky
  - Pravdepodobnosť, že $h(x)=i$ je $1/n$ pre každé $i \in \{0, 1, ..., N-1\}$

- Zvyčajne ako kompozícia dvoch funkcií $h(x) = h_2(h_1(x))$:
  - Výpočet hashkódu **h_1: kľúč** $\rightarrow$ **celé číslo**
  - Kompresná funkcia **h_2: celé číslo** $\rightarrow \{0, 1, ..., N-1\}$

- Obe funkcie (h_1 a h_2) by mali byť navrhnuté pre celkovú minimalizáciu počtu kolízií

Výpočet hashkódu

- **h_1: kľúč → celé číslo**
  - Zobrazuje kľúč x na celé číslo
  - Nie nutne do intervalu [0,n-1]
  - Môže byť aj záporné
- Predpokladáme 32-bitové číslo (integer)
- Snažíme sa navrhnúť výpočet hashkódu tak, aby čo najlepšie predchádzal kolíziám
  - Kompresná funkcia nemá ako „opraviť“ kolíziu v hashkódoch

Výpočet hashkódu – prvý (chybný!) pokus

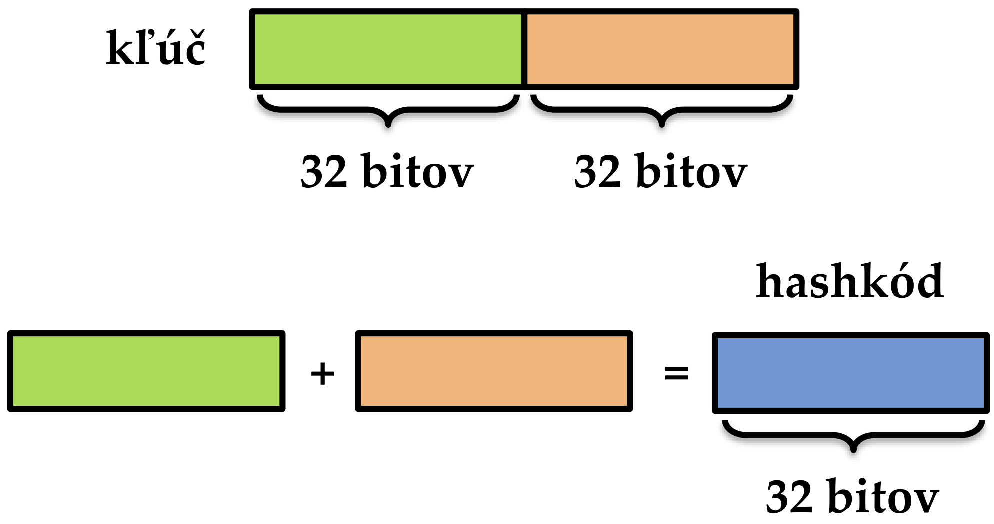
- Kľúč je v programe ako premenná
- **adresu premennej** kľúča môže byť hashkód kľúča
- V niektorých prípadoch to môže postačovať (možno napr. v prípade porovnávania objektov)
- Zvyčajne potrebujeme uvažovať **hodnotu** kľúča a nie jeho umiestnenie v pamäti

Výpočet hashkódu – druhý pokus

- Hashkód kľúča sú priamo bity kľúča (interpretované ako celé číslo)
- Vhodné ak dátový typ kľúča je menší alebo rovnaký ako dátový typ celého čísla (int)
  - char, byte, short, ...
- V prípade, že dátový typ kľúča je dlhší ako typ int, tak odstránime prebytočné bity
  - long, double, ...
  - kolízie nastanú, ak sa kľúče líšia v bitoch, ktoré sme odstránili

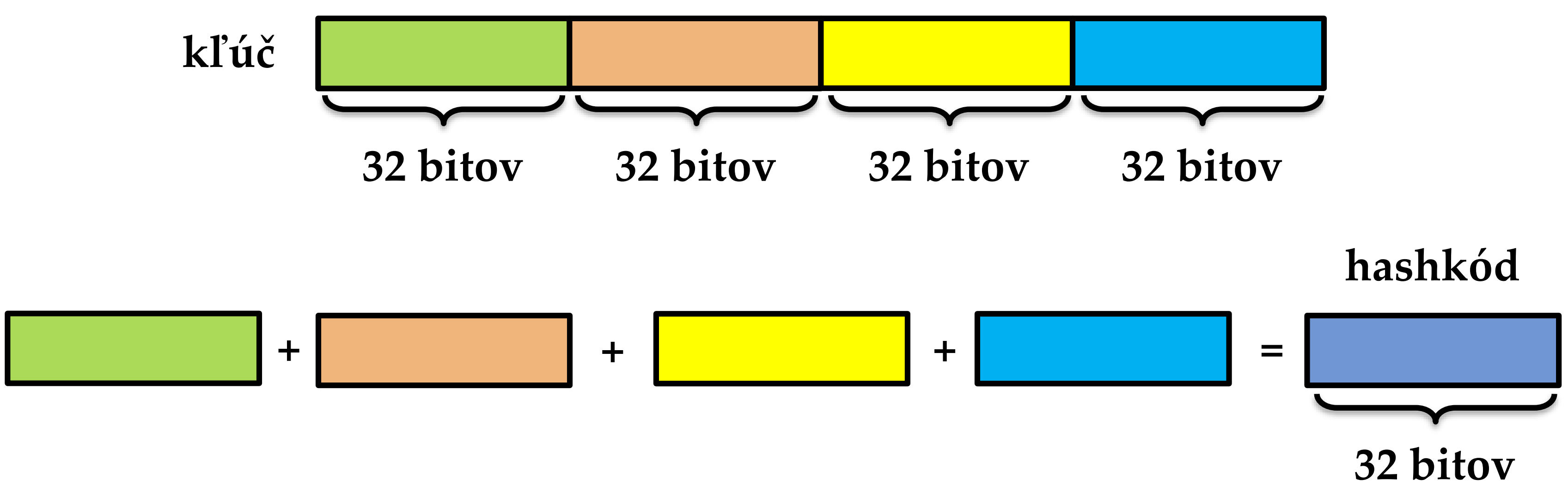
Výpočet hashkódu – tretí pokus

- Súčet komponentov
- Vhodné ak je dátový typ kľúča väčší ako celé číslo (int)
- Rozdelíme kľúč na 32-bitové časti, ktoré sčítame



Výpočet hashkódu – tretí pokus (2)

- Súčet komponentov
- Vhodné aj pre väčšie dátové typy – 128 bitové, aj dlhšie



Výpočet hashkódu – reťazce

- Súčet komponentov (8 bitové časti = 1 znak)
- Súčet ASCII kódov znakov
  - „abeceda“ = ‘a’ + ‘b’ + ‘e’ + ‘c’ + ‘e’ + ‘d’ + ‘a’
- Môže vznikať príliš veľa kolízií
  - Napr. anglické slová: „stop”, “spot”, “pots”, “tops”, “opts”, …
- Nedostatok obyčajného súčtu:
  nezohľadňuje pozíciu jednotlivých znakov

Výpočet hashkódu – polynomiálna akumulácia

- Kľúče sú k-tice (rozličnej dĺžky) $\mathbf{x_0, x_1, \ldots, x_{k-1}}$ pričom poradie komponentov (x_i) je dôležité
- Zvolíme konštantu c ≠ 0
- Hashkód pre kľúč x:

$$p(c) = x_0 + x_1 c + x_2 c^2 + \cdots + x_{k-1} c^{k-1}$$

- Pretečenia sa ignorujú
- Vhodné pre reťazce, dobrá voľba: c = 33, 37, 39, 41 (experimentálne: pre c = 33 je najviac 6 kolízií na množine 50 000 anglických slov)

Výpočet hashkódu – polynomiálna akumulácia

- Hashkód pre kľúč x=($\mathbf{x_0}$, $\mathbf{x_1}$, **...**, $\mathbf{x_{k-1}}$) c ≠ 0:

$$p(c) = x_0 + x_1 c + x_2 c^2 + \cdots + x_{k-1} c^{k-1}$$

- Ako to efektívne vypočítať?
- Hornerova schéma – špeciálna forma zápisu

$$p(c) = x_0 + c(x_1 + c(x_2 + \cdots + c(x_{k-2} + x_{k-1}c))$$

- Optimálny výpočet (čo do počtu sčítaní a násobení)
  - **k** operácií sčítania a **k** operácií násobenia

Polynomiálna akumulácia – implementácia

- Hornerova schéma – špeciálna forma zápisu

$$p(c) = x_0 + c(x_1 + c(x_2 + \cdots + c(x_{k-2} + x_{k-1}c))$$

- Implementácia pre c=31:

```
int hash(char *str)
{
    int i, len = strlen(str), h = 0;
    for (i = 0; i < len; i++)
        h = 31*h + str[i];
    return h;
}
```

Kompresné funkcie

- **h_2: celé číslo** $\rightarrow \{0, 1, ..., N-1\}$
  - máme celé číslo (nie nutne v rozsahu indexov tabuľky)
  - potrebujeme index tabuľky (v platnom rozsahu)
  - dobrá kompresná funkcia minimalizuje počet kolízií
- **mod N** – zvyšok po delení **$h_2(x) = |x| \bmod N$**
  - N by malo byť prvočíslo – rovnomernejšie rozptyľuje
    Uvažujme kľúče **200, 205, 210, 300, 305, 310, 400, 405, 410**
    Pre **N = 100**, výsledné hodnoty sú **0, 5, 10, 0, 5, 10, 0, 5, 10**
    Pre **N = 101**, výsledné hodnoty sú **99, 3, 8, 98, 2, 7, 97, 1, 6**
- **Multiply, Add, Divide – $h_2(x) = |ax + b| \bmod N$**
  - N by malo byť prvočíslo, $a > 0$, $b \geq 0$, $a \bmod N \neq 0$
  - Hodnoty a, b sa väčšinou zvolia náhodne
  - Lepšie rozptyľuje ako obyčajné mod N

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 <u>univerzálny</u> 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, $\mathbf{M = 2^b}$
- Zvolíme h ako náhodnú maticu 0/1 veľkosti $\mathbf{b \times u}$**,** a definujeme $\mathbf{h(x) = h \cdot x}$

Napr. $\begin{array}{ccc} h & x & h(x) \\ \boxed{\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{matrix}} & \boxed{\begin{matrix} 1 \\ 0 \\ 1 \\ 0 \end{matrix}} = & \boxed{\begin{matrix} 1 \\ 1 \\ 0 \end{matrix}} \end{array}$

- 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 ($\mathbf{x_i = 0, y_i = 1}$), i-ty stĺpec vieme určiť $\mathbf{2^b}$ 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(N^2)**
  Zvoľme veľkosť tabuľky M = N^2
- Uvažujme, triedu univerzálnu hashovacích funkcií H, náhodne vyberme h ∈ 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 ∈ H máme kolízie, vyberieme znovu :)

Perfektné hashovanie – O(n) metóda

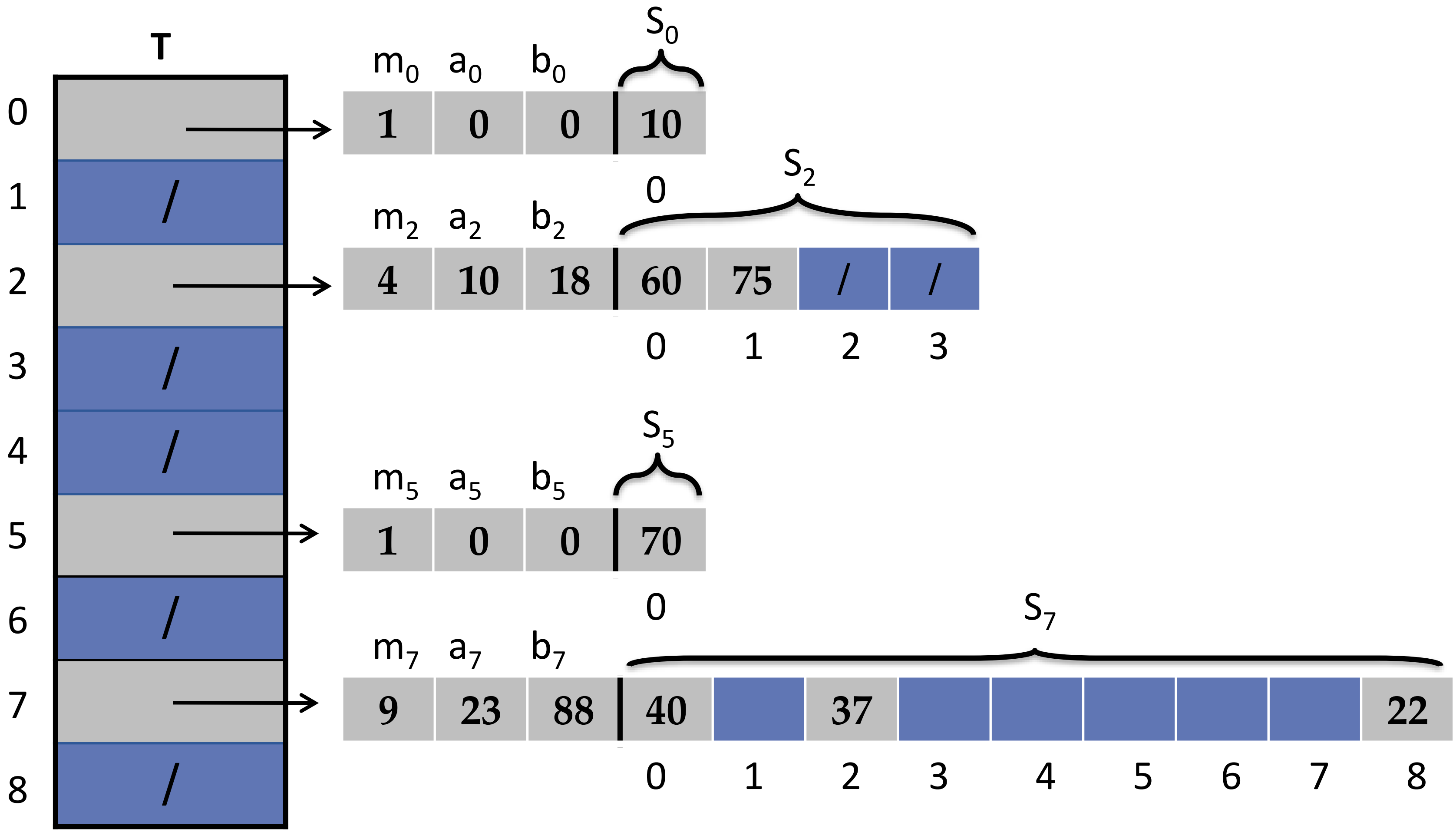
- Zlepšíme priestorovú zložitosť
- **Použijeme dve úrovne hashovacích tabuliek**:
  - Hashovanie na prvej úrovni rozptýli prvky na triedy (môže sa ešte vyskytnúť kolízia)
  - Druhá úroveň – v každom vedierku je ďalšia hashovacia tabuľka (i-te vedierko obsahuje $\boldsymbol{n_i}$ prvkov), ktorá využíva 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 $\sum_i (ni)^2 < 4N$, a potom nájdeme sekundárme hashovacie funkcie h_1, h_2, ..., 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 malý
  - Pre otvorenú adresáciu sa odporúča α okolo 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

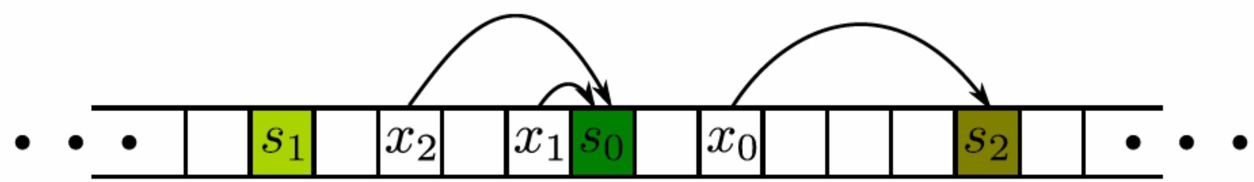
- Používateľ vyžiada stránku (napr. www.google.com)
  - Zbytočne sa opakovane sťahuje zo servera
  - Môžeme si ju odpamätať (do lokálnej cache prehliadača)
  - Pri ďalšom prístupe najskôr pozrieme do cache
  - Rýchlejšia odozva
- Ak bude takéto cache zdieľaná pre väčšiu skupinu používateľov (napr. celú FIIT), tak potom keď ku rovnakej stránke prístupy 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)

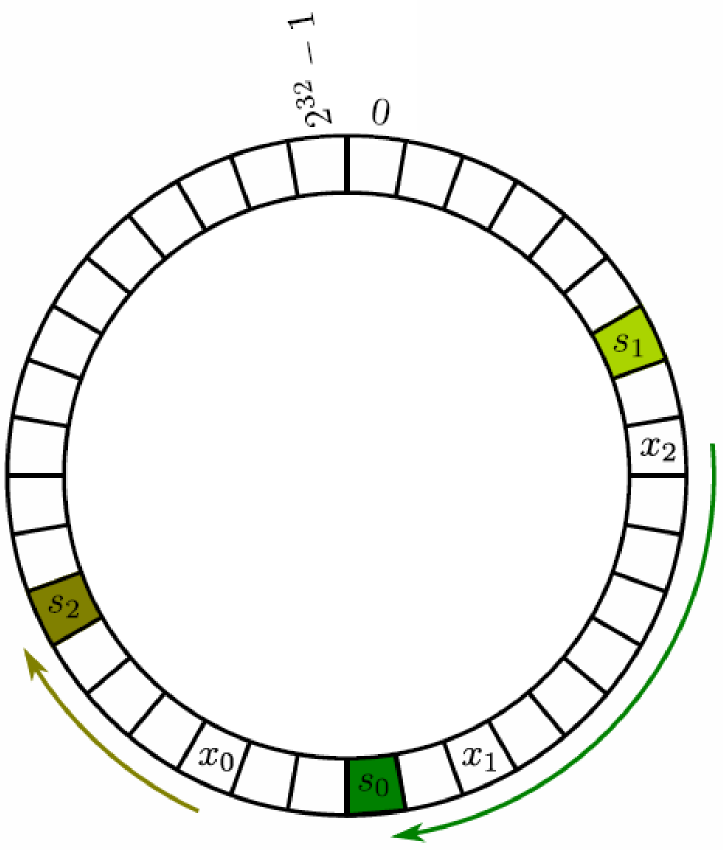
- Hlavná myšlienka: okrem hasovania stránok, budeme do rovnakej tabuľky hashovať aj uzly (servery) cache
- Prvok (stránku) x pridelíme tomu serveru, ktorý v tabuľke nasleduje najbližšie (doprava):



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

Konzistentné hashovanie (consistent hashing)

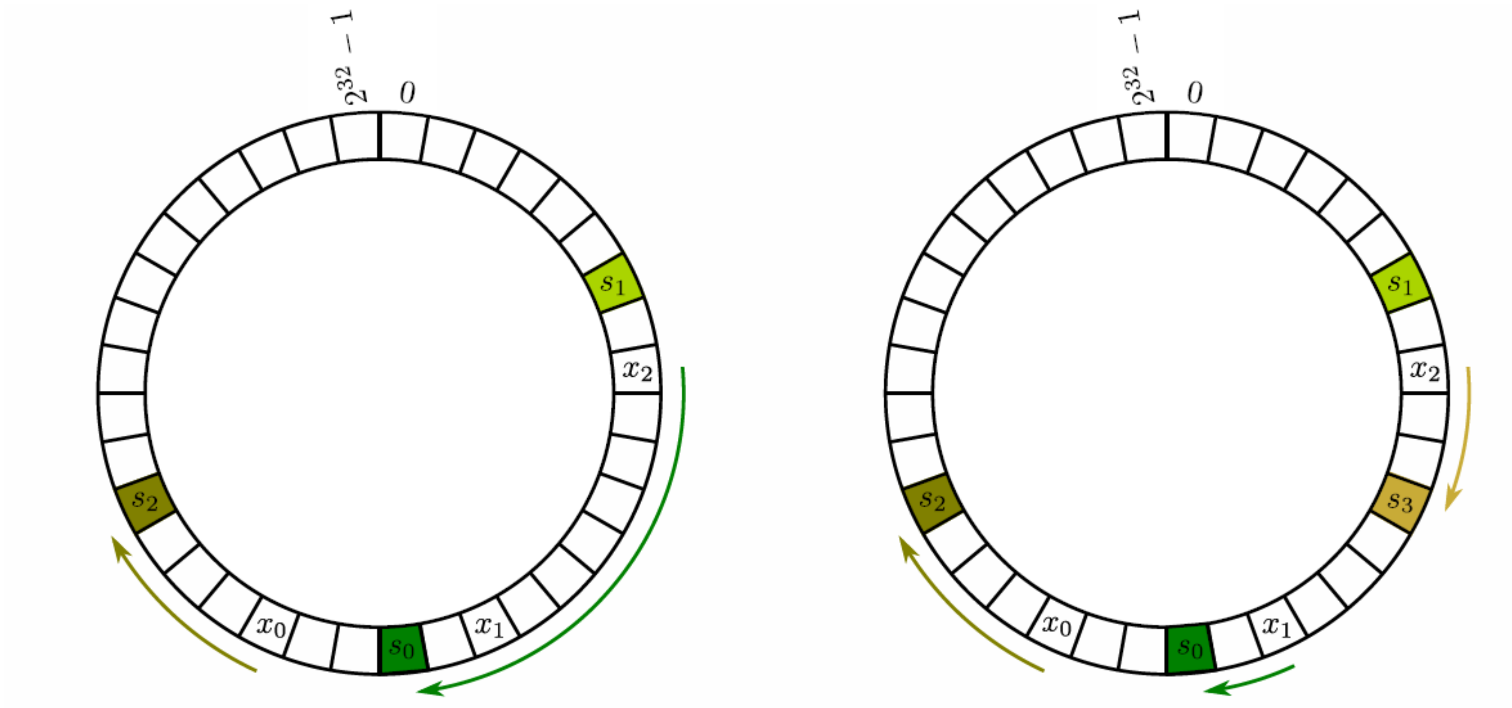
- Zobrazené na koliečku:



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)**
- 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)

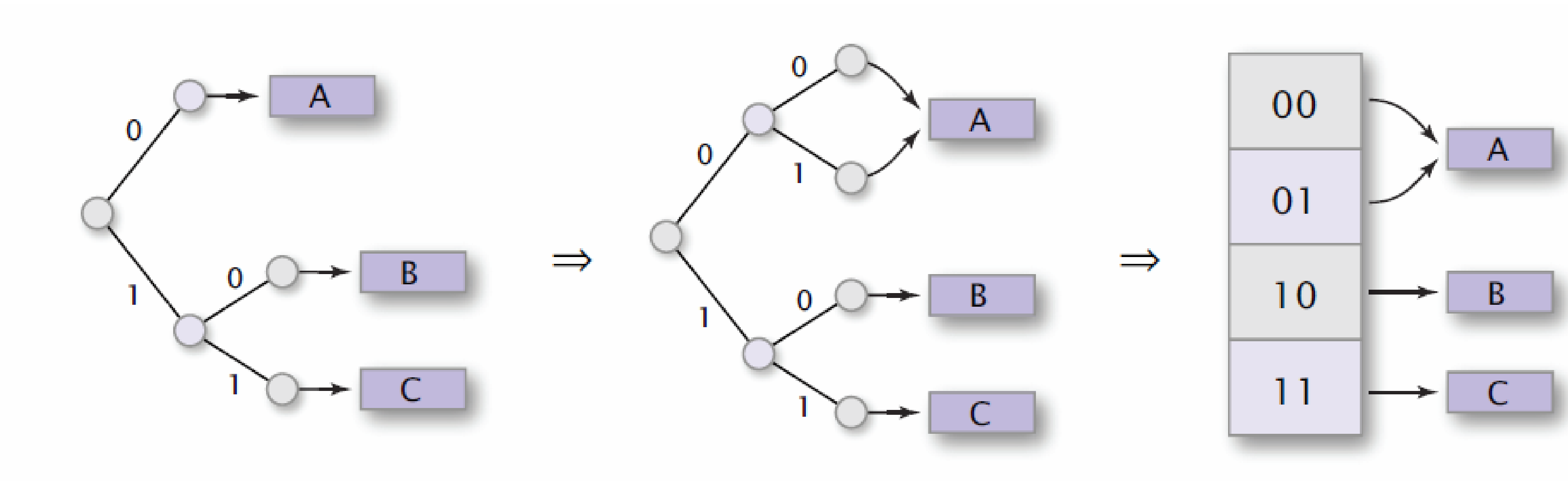

STAR WARS
EPISODE I
THE PHANTOM MENACE

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... Pováčšinu 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 používateľ v P2P sieťach tretej generácie (Napster bola prvá generácia :)

Rozšíriteľné hashovanie (extendible hashing)

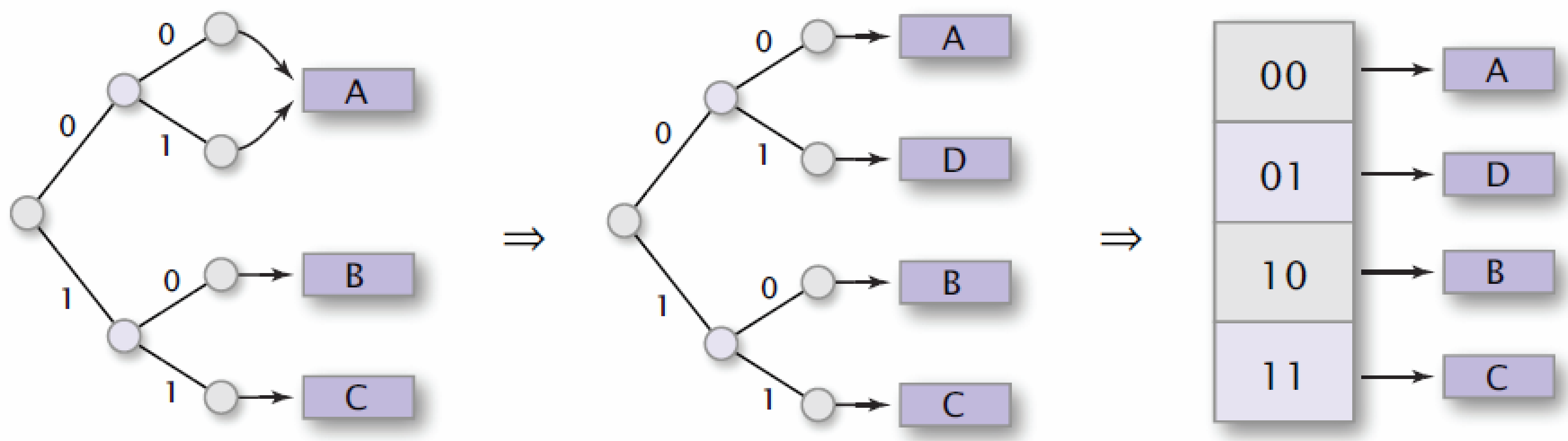
- Keď zväčšovanie tabuľky je výpočtovo náročné
  - Napr. súborový systém
- Binárny trie na vyhľadanie vedierka (kľúč je hash ako binárny reťazec), reprezentovaný v plochom poli



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

Rozšíriteľné hashovanie (extendible hashing)

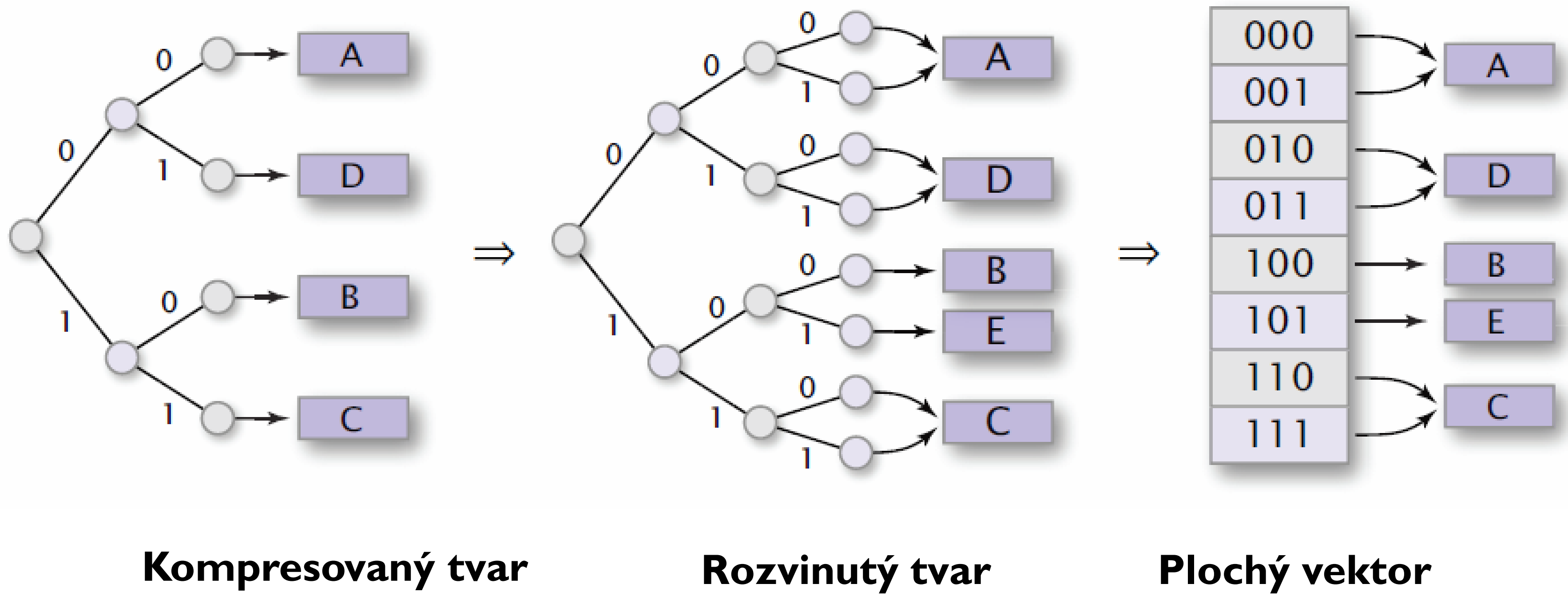
- Pridanie vedierka:



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

Rozšíriteľné hashovanie (extendible hashing)

- Rozšírenie stromu (zvýšenie výšky)



- 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: **najviac dva krát 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
- Spôsob 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
- Môže sa však posunúť aj starý prvok
  - Kukučkové hashovanie (cuckoo hashing)
  - Last-come-first-served hashing
  - Robinhood hashing

- Split sequence hashing – posun zavisí od prvku 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
- Denial-of-service útoky založená 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 pozícií **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 „odstránit"“ 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

Nabudúce ... Priebežný test

- Píšeme 60 minút.
- Prvý beh:
  10:50-11:50
- Druhý beh:
  11:55-12:55

- Rozpis pošlem mailom