SLOVENSKÁ TECHNICKÁ
UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY
A INFORMAČNÝCH TECHNOLÓGIÍ

Dátové štruktúry a algoritmy

Hashovanie

24. 03. 2020

letný semester
2020/2021

prednášajúci: Lukáš Kohútka

Opakovanie – Problém vyhľadávania

- **Vstup:**
  - Postupnosť: $a_1, a_2, a_3 \dots a_n$
    $k(a_i)$ 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 $\in \{1,2,\dots,n\}$ takého prvku, že $k(a_{res}) = x$,
    alebo 0 ak taký prvok neexistuje.

Uvažujme špeciálny prípad slovníka

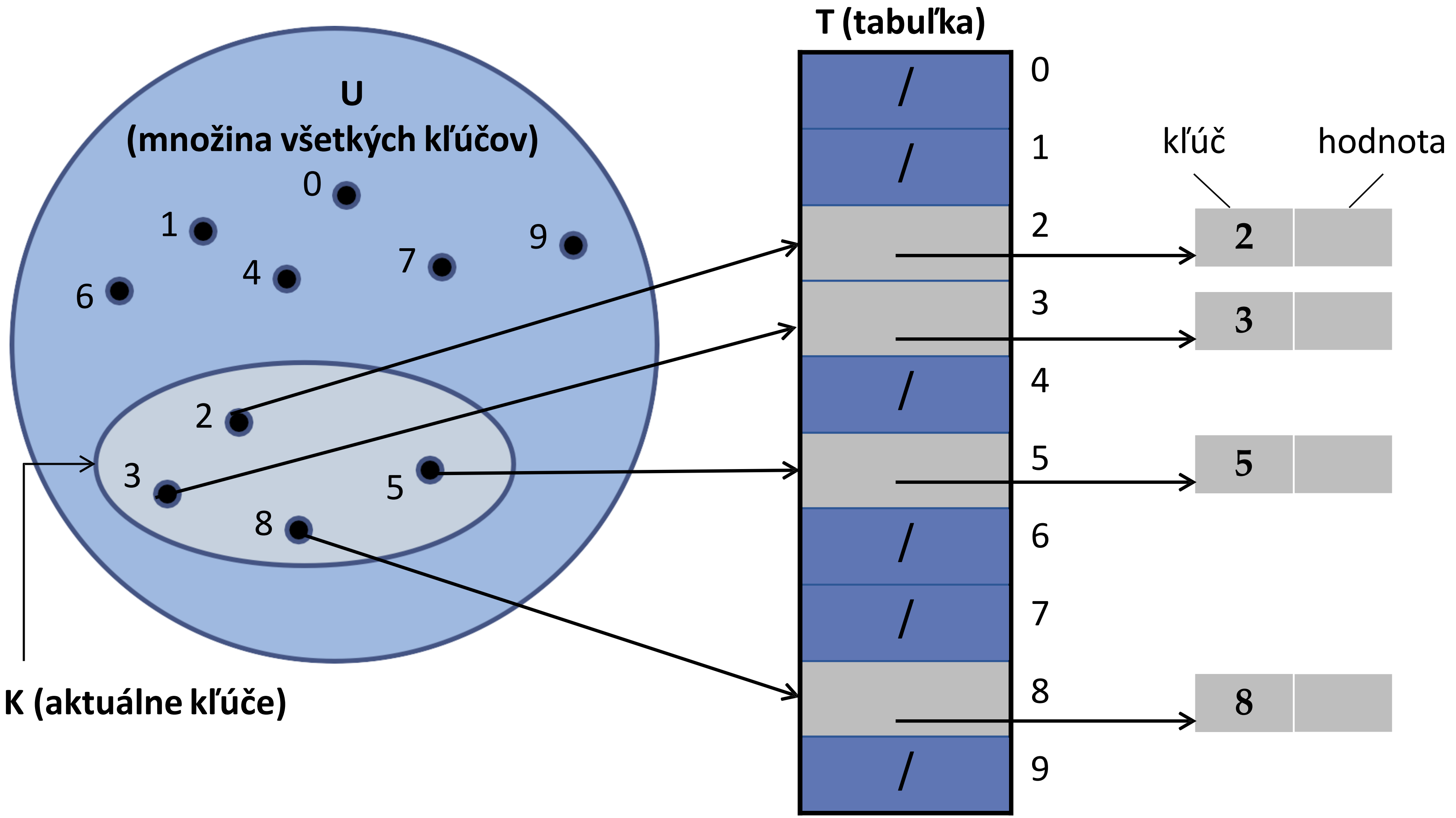
- Kľúče $k(a_i)$ nech sú rôzne čísla od 0 do n-1:
  Napr. (kľúč, hodnota):
  (0,”Peter”), (1,”Milan”), …, (n-1,“Katka“)

- 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

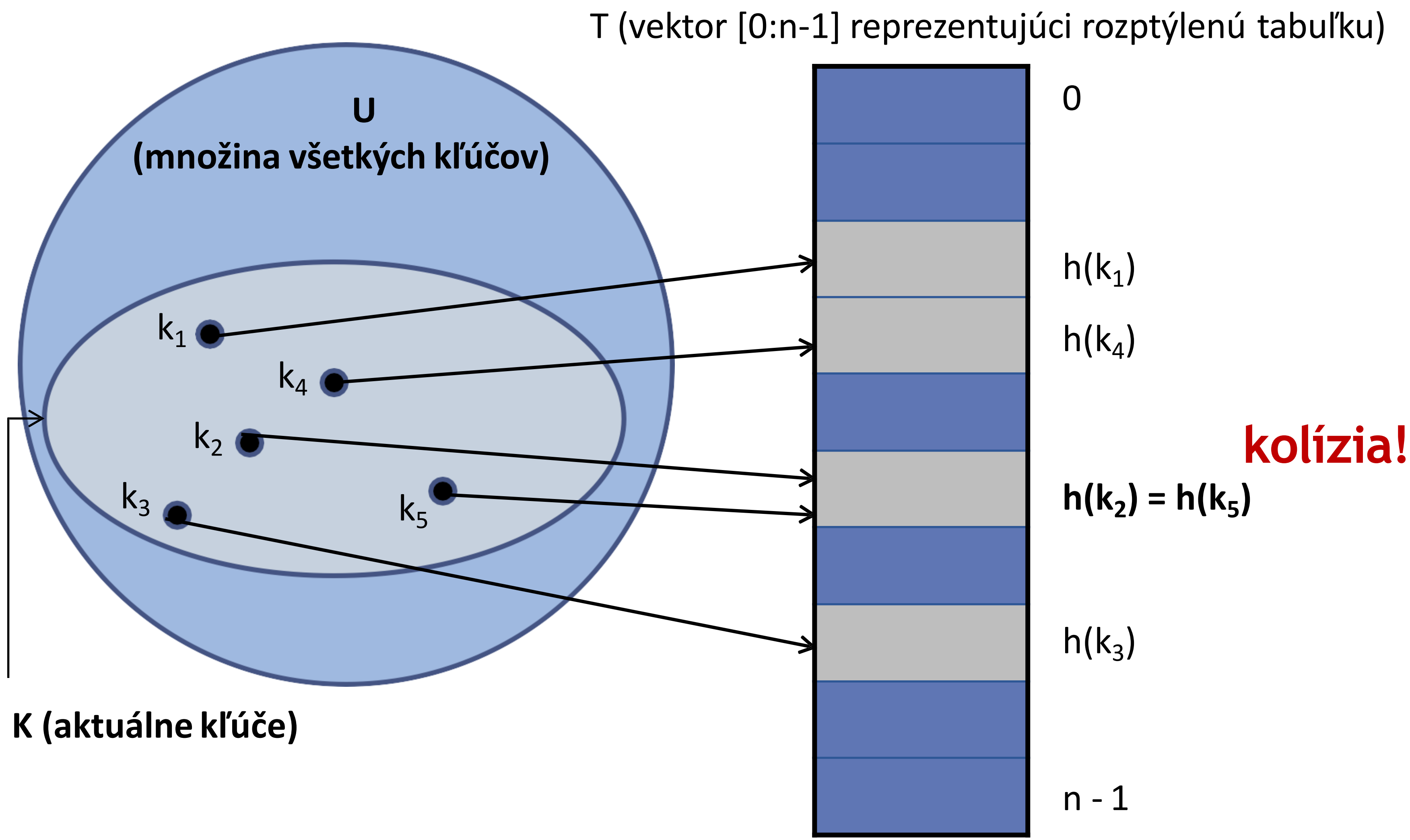
- Hashovacou funkciou **h** zobraziť kľúč **x** do rozsahu indexov poľa:

- 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



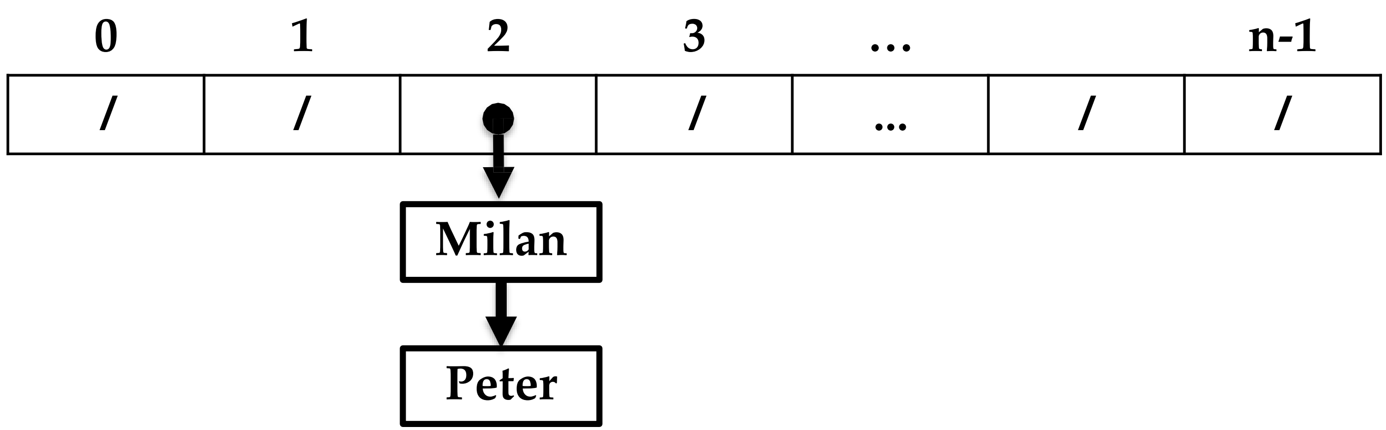
Spôsoby riešenia kolízií

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

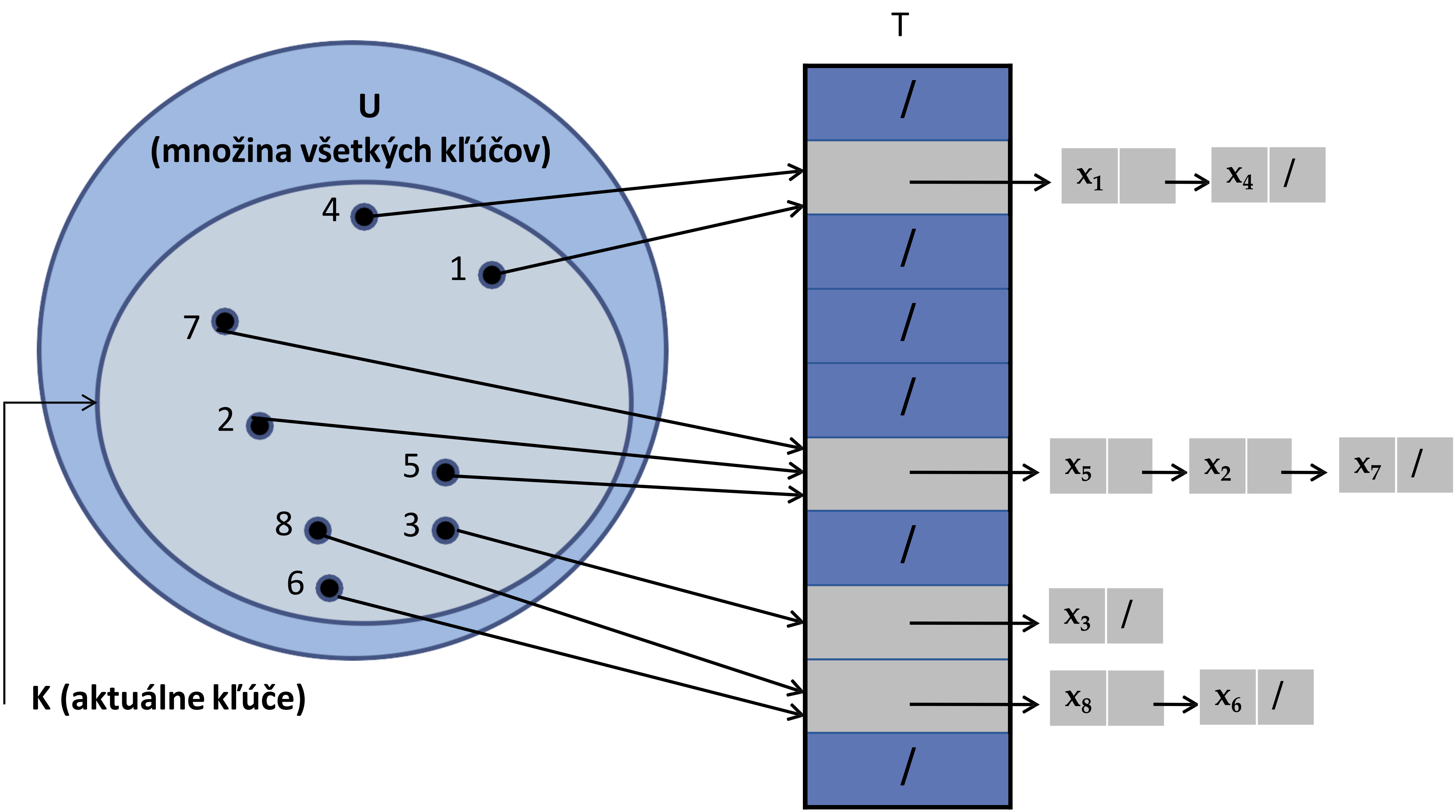
- Existujú dva spôsoby riešenia kolízií:
  - **Reťazenie (chaining)** – umožníme, aby v každom mieste tabuľky mohlo byť aj viacero prvkov (napr. v spájanom zozname)
  - **Otvorené adresovanie (open addressing)** – v každom mieste tabuľky môže byť nanajvýš jeden prvok (musíme nájsť nejaké alternatívne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

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

- 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
      retur  NIL
}     n
```

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	…				
		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 $\alpha = N/M$
- 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 $\Rightarrow$ veľa prázdnych miest v tabuľke
- Pre malé M $\Rightarrow$ 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

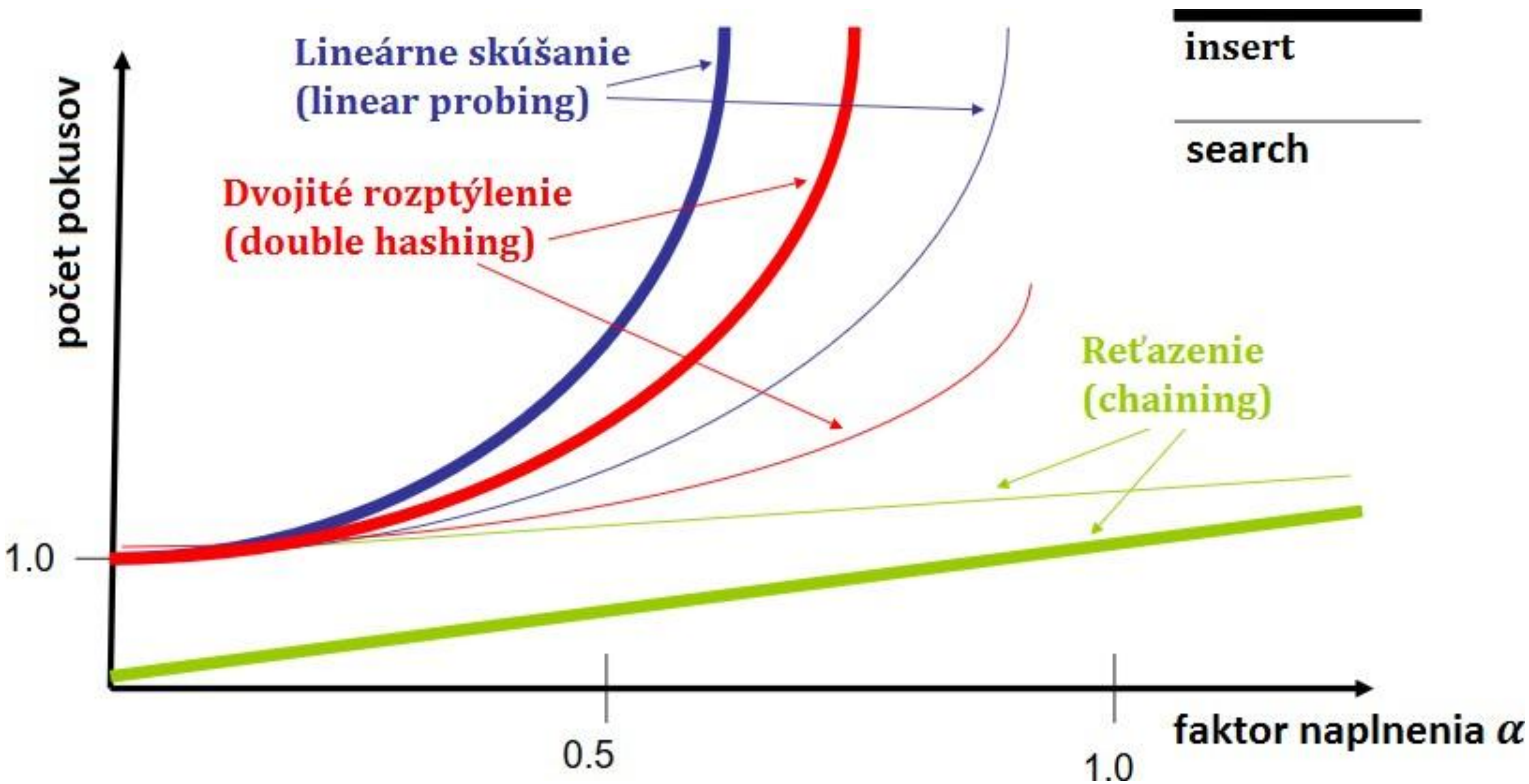
Reťazenie vs. otvorené adresovanie

- **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 $\longrightarrow$ 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

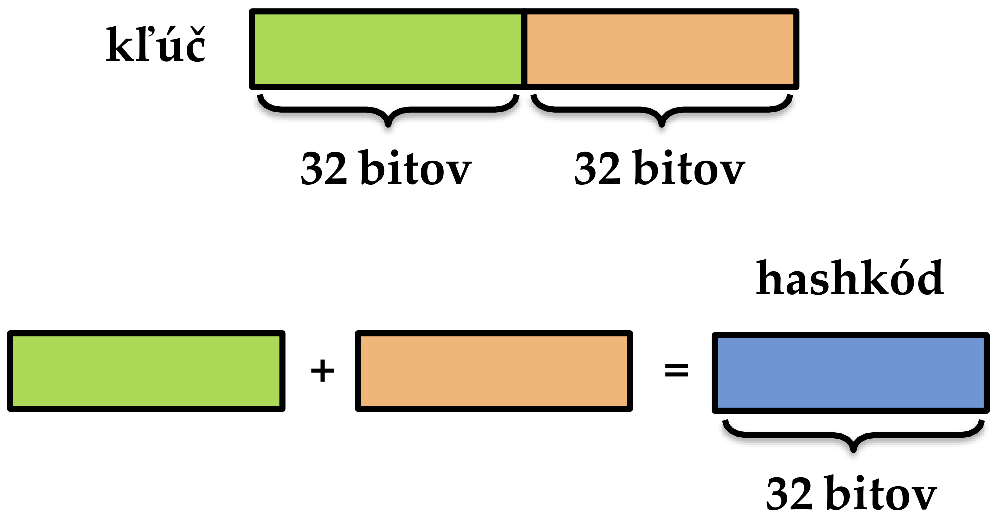
- Kľúč je v programe ako premenná
- **adresa premennej** kľúča môže byť hashkód kľúča
- V niektorých prípadoch to môže postačovať (možno napr. v prípade porovnávania objektov)
- Zvyčajne potrebujeme uvažovať **hodnotu** kľúča a nie jeho umiestnenie v pamäti

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

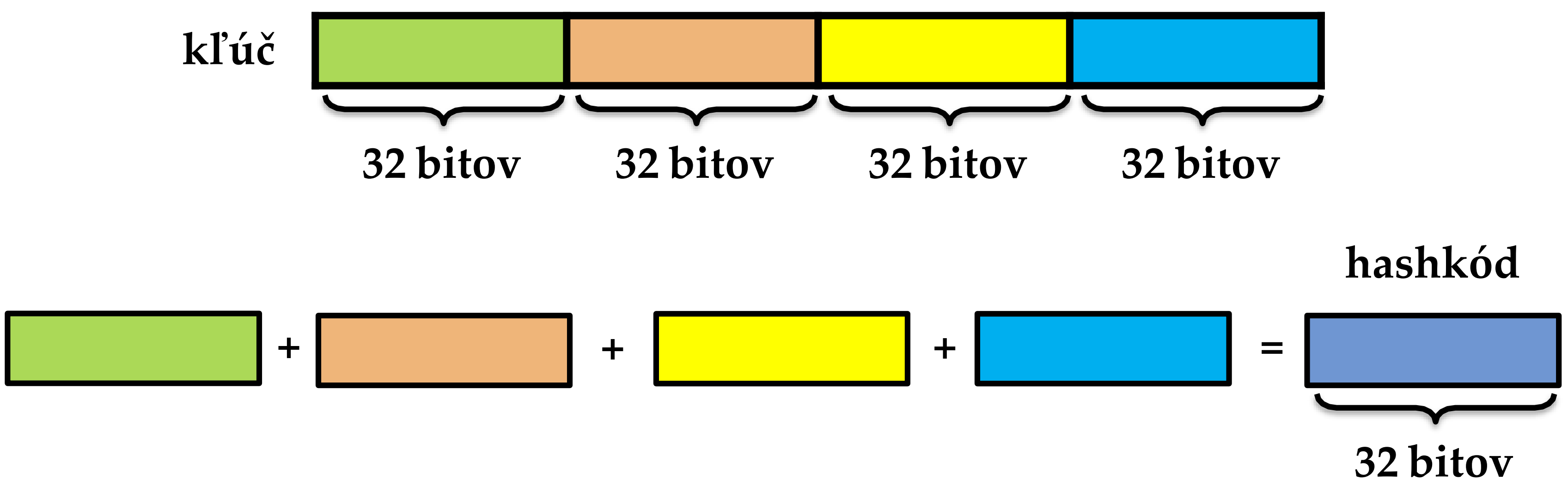
Výpočet hashkódu – tretí pokus

- 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 $c \neq 0$
- 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=($\mathbf{x_0, x_1, \ldots, x_{k-1}}$) 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** $\rightarrow \{0, 1, \ldots, N-1\}$
  - 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 $\mathbf{h_2(x) = |x| \bmod N}$
  - 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 –** $\mathbf{h_2(x) = |ax + b| \bmod N}$
  - N by malo byť prvočíslo, $a > 0$, $b \geq 0$, a N = veľkosť tabuľky
  - Hodnoty a, b sa väčšinou zvolia náhodne
  - Lepšie rozptyľuje ako obyčajné mod N

Ďakujem za pozornosť