Tabuľka

(Dynamická) množina

- abstraktný údajový typ množina
  - zvláštne vlastnosti:
    - počet prvkov v údajoch typu množina sa často mení
  - najčastejšie operácie:
    - insert, search, delete
- v takomto prípade ide o **slovník**
- napr:
  - tabuľka symbolov v prekladačoch

Tabuľka

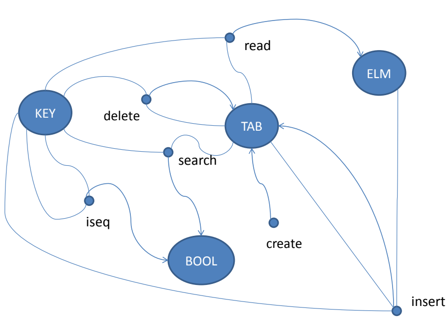
- skupina metód, ako implementovať slovník
- bežne aj synonymum pre slovník, najmä ak uvažujeme aj jeho implementáciu
- ale spravidla pomenovanie implementujúceho vektora

- určuje štruktúru pre jednotlivé údaje, ktorá združuje/asociuje hodnotu s kľúčom.
- V pamäti sa údaje uchovávajú ako dvojica kľúč – hodnota = položka, prvok tabuľky
- K hodnote sa pristupuje pomocou kľúča – kľúč položku jednoznačne určuje

Slovník/tabuľka – formálna špecifikácia

- Druhy: TAB, ELM, KEY, BOOL
- Operácie:
  - CREATE() → TAB vytvorenie prázdnej tabuľky
  - INSERT(tab, key, elem) → TAB vloženie prvku
  - READ(tab, key) → ELM výber prvku
  - DELETE(tab, key)→ TAB vymazanie prvku
  - ISEQ(key, key) → BOOL porovnanie 2 prvkov
  - SEARCH(key,tab) → BOOL test, či sa v tabuľke nachádza prvok

slovník/tabuľka



slovník/tabuľka

search(k, create) = false
search(k_1, insert(k_2,e,t)) =
if(iseq(k_1, k_2))
then true
else search(k_1, t)

slovník/tabuľka

```
delete(k, create) = create
delete(k1, insert(k2,e,t)) =
 if(iseq(k1, k2))
  then delete(k1, t)
  else insert(k2,e,delete(k1, t))
```


slovník/tabuľka

```
read(k, create) = error_elem
read(k1, insert(k2,e,t)) =
 if(iseq(k1, k2))
  then e
  else read(k1, t)
```


slovník/tabuľka

```
insert(k1,e1, insert(k2,e2,t)) =
  if( iseq(k1, k2))
     then insert(k1,e1,t)
     else insert(k2,e2,insert(k1,e1,t))
```


slovník/tabuľka – implementácia

reprezentácia pomocou vektora a prípadne aj zoznamov

- pre malý počet možných prvkov
  - sekvenčné sprístupňovanie … O(n)
- pre malý interval možných kľúčov
  - priamy prístup (kľúč = index) … O(1)
- pre veľkú množinu (univerzum) možných kľúčov
  - sprístupňovanie rozptýlením/hešovaním … od O(1) do O(n)

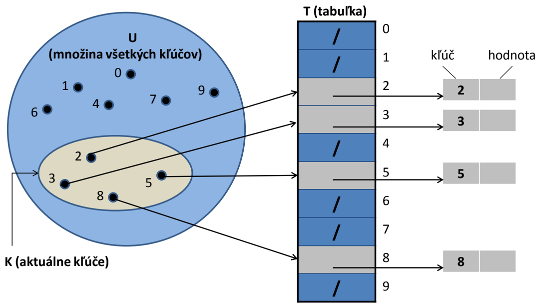
reprezentácia slovníka pomocou spájaného zoznamu

- insert – **`insert`**$_{SLL}$
- delete – **`delete`**$_{SLL}$
- search – **`find`**$_{SLL}$

- sekvenčné sprístupňovanie … O(n)
- príliš pomalé pre mohutnejšie množiny

reprezentácia slovníka pomocou tabuľky s priamym prístupom

- tabuľka s priamym prístupom:
  - vektor T[0 .. m-1]
  - každé miesto v tabuľke (každý prvok vektora) korešponduje s (má index) jediným kľúčom v univerze kľúčov U.
- insert(T, x)
  - T[kľúč[x]] <- x
- delete (T, x)
  - T[kľúč[x]] <- NIL
- search(T, k)
  - return T[k]



reprezentácia slovníka pomocou tabuľky s priamym prístupom
- každý kľúč z celej množiny kľúčov U = {0, 1, ..., 9} korešponduje s indexom v tabuľke T
- aktuálna množina prvkov s kľúčmi K = {2, 3, 5, 8} je zapísaná v T ako smerník na prvok (dáta), ostatné sú prázdne resp. sa rovnajú null

reprezentácia slovníka pomocou tabuľky s priamym prístupom

- tabuľka s priamym prístupom:
  - vektor T[0 .. m-1]
  - každé miesto v tabuľke (každý prvok vektora) korešponduje s (má index) jediným kľúčom v univerze kľúčov U.
- v mieste T[kľúč[x]] môže byť
  - smerník na prvok množiny (zapísanej v tabuľke) s kľúčom kľúč a hodnotou
  - samotná hodnota (úspora pamäti)

rozptýlená tabuľka

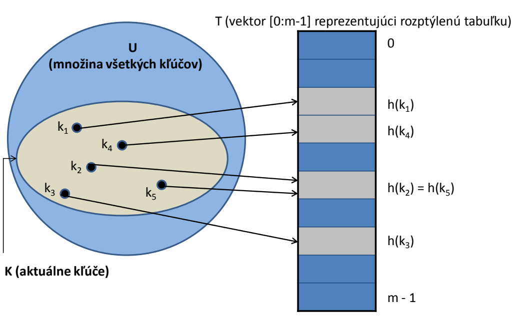
- tabuľka s priamym prístupom je fajn (t.j. ideálne riešenie, keďže časová zložitosť prístupu je O(1) , dokonca aj Θ(1)), ale...
- ak |U|>> size(T),
  - možno ani toľko pamäti v počítači nie je
  - ak by aj bolo, často |K| t.j. počet skutočne zapísaných prvkov s kľúčmi z K je tiež |U|>> |K| a teda by sa zbytočne mrhalo veľkou väčšinou pamäti pridelenej pre T
- uvažujme inú tabuľku, ktorá by mala
  - požiadavky na pamäť úmerné skutočnej potrebe aj v najhoršom prípade, t.j. Θ(K)
  - požiadavky na čas stále nezávislé od K, t.j. O(1) aspoň v priemernom čase, hoci nie aj v najhoršom čase, vtedy bohužiaľ Θ(K)

rozptýlená tabuľka

- Tabuľka, kde sa kľúč vypočíta pomocou špeciálnej rozptylovej (hešovacej) funkcie

- Príklad jednoduchej rozptylovej funkcie:
  - Kľúč = súčet ascii hodnôt jednotlivých znakov
  - Value = OKNO

OKNO → key = ascii(O) + ascii(K) + ascii(N) + ascii(O)
= 117 + 113 + 116 + 117
= 463



reprezentácia slovníka pomocou rozptýlenej tabuľky
- použitie rozptylovej funkcie h na zobrazenie/namapovanie kľúčov k do indexov rozptýlenej tabuľky
- kľúče k_2 a k_5 sa zobrazia na rovnaký index - kolízia

rozptylová funkcia

- Základné vlastnosti rozptylovej funkcie:
  - -Výpočet by nemal byť náročný
  - -Mala by byť navrhnutá tak, aby vznikalo čo najmenej kolízií
    - -úplne sa kolíziám nedá vyhnúť.
      - -Prečo? |U|>> m. Priestor možných kľúčov je omnoho väčší než adresový priestor. Adresový priestor sa navrhuje rozumne malý/veľký na základe odhadu, koľko môže byť najviac prvkov v slovníkoch.
    - -čím bude rozptylovanie náhodnejšie, tým bude pravdepodobnosť kolízií menšia.
- vždy treba spôsob rozriešenia kolízií
  - reťazenie
  - otvorené adresovanie

Aká má byť *h(k)* ?

- h(k) má byť rovnomerná t.j. má posielať kľúče rovnomerne na všetky adresy adresového priestoru *0..m-1* t.j. pravdepodobnosť, že pošle kľúč na hociktorú z *m* adries je *1/m*
- Pre tabuľku s *m* položkami:
- $\sum P(k) = 1/m$ pre j=0,1,...,m-1
  - suma cez k: $h(k)=j$
    t.j. pre všetky možné kľúče spočítame pravdepodobnosti, že ich rozptylová funkcia pošle na adresu j a súčet má byť 1/m pre všetky j
  - ale zvyčajne nepoznáme *P(k)*
- Ak poznáme *P(k)*, napr.:
  - Kľúče sú náhodné reálne čísla nezávisle rovnomerne rozdelené v intervale 0<=k<1,
  - *h(k)= floor(k.m)*

19

Ako navrhnúť *h(k)* ?

- Pomocou heuristík, opierajúc sa o kvalitatívne znalosti o P.
- Tabuľka symbolov v prekladači:
  - vieme, že často sa vyskytujú v programe symboly, ktoré sa len málo líšia, napr:
    refA, refB
  - Dobrá h(k) by mala minimalizovať prípady, že takéto symboly pošle na rovnaké miesto v tabuľke.

20

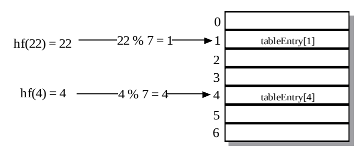
Návrh *h(k)* metódou delenia

- *h(k)* = *k* mod *m*
  - *m = 16, k = 36, h(36) = 4. (nie dobrý príklad)*
- *m* nemá byť mocnina 2. Prečo?
  - $m = 2^p$: vtedy *h(k)* závisí len od dolných *p* bitov kľúča
- *m* nemá byť mocnina 10. Prečo?
- m má byť prvočíslo nie blízke nejakej mocnine 2.

21

rozptylová funkcia - príklad

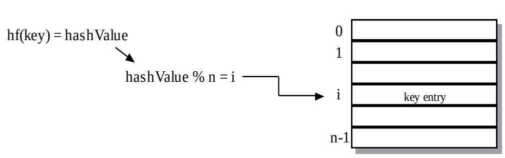
- h(x) = hf(x) mod m
- hf(x) = x, kde x je kladné číslo.
- Tabuľku predstavuje vektor s veľkosťou 7
  - rozptylové hodnoty sa musia zobraziť do [0:7-1]



22

rozptýlenie/hešovanie

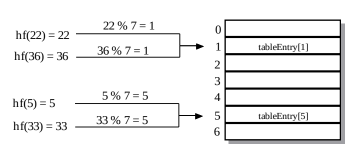
```
Hash Value:         hf(key) = hashValue
HashTable index:    hashValue % n
```



23

rozptylová funkcia - príklad

- Pri zvolenej rozptylovej funkcii nastáva kolízia pri každých 2 kľúčoch, ktoré sa navzájom líšia o násobok veľkosti implementujúceho vektora
- 36 – 22 = 14 = 2*7 → nastane kolízia



24

Jednoduchá rozptylová funkcia

- Častokrát je kľúčom reťazec (string)
  - Vo funkcii môžeme kombinovať postupnosť znakov z reťazca

```
public int hashCode()
{   int hash = 0;

    for (int i = 0; i < n; i++)
        hash = 31*hash + s[i];

    return hash;
}
```


Jednoduchá rozptylová funkcia

Výpočet rozptylovej hodnoty pre 3 rôzne reťazce:
Hodnota pre strB je záporná kvôli preplneniu

```
String strA = "and", strB = "uncharacteristically",
        strC = "algorithm";

hashValue = strA.hashCode();      // hashValue = 96727
hashValue = strB.hashCode();      // hashValue = -2112884372
hashValue = strC.hashCode();      // hashValue = 225490031
```

-ak nastane prípad, že rozptylová funkcia vráti záporné číslo, pribudne problém pri práci s vektorom/poľom (záporný index neexistuje).
-takýto výpočet však zabezpečí, že index bude vždy kladné číslo:
tableIndex = (hashValue & Integer.MAX_VALUE) % tableSize

Návrh *h(k)* metódou násobenia

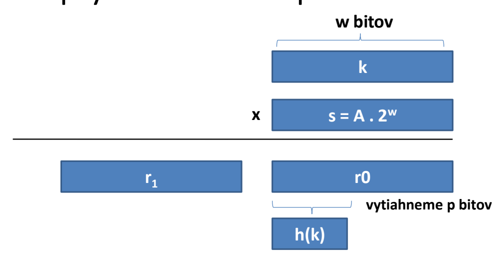
1. Vynásobiť kľúč *k* zvolenou konštantou *A*, 0<*A*<1 a vybrať zlomkovú časť z *k.A*.
2. Vynásobiť túto hodnotu *m* a vziať celú časť.

h(k) = floor(*m.* (*k.A* mod 1))

kde *k.A* mod 1 označuje zlomkovú časť z *k.A*, t.j. *k.A* – floor(*k.A*)

voľba *A*: podľa Knutha približne (sqrt(5) – 1)/2 = 0.6180339887

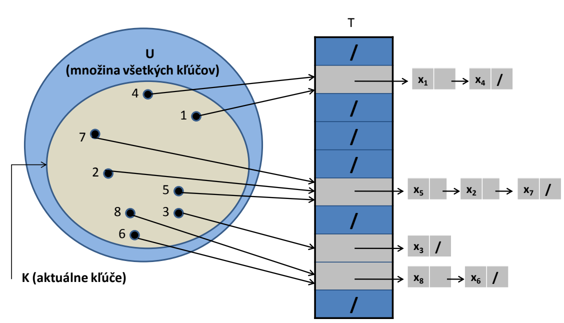
rozptylová funkcia - príklad



Násobenie ako metóda hešovania
-w-bitová reprezentácia kľúča sa vynásobí w-bitovou hodnotou s = A.2^w
- p najvyšších bitov nižšej w-bitovej časti výsledku zvolíme ako rozptylovú hodnotu funkcie h(k)

kolízia

- Ak rozptylová hodnota dvoch (alebo viacerých) prvkov ukazuje na rovnaké miesto vo vektore implementujúcom tabuľku (skrátene v tabuľke), nastáva kolízia. Dva prvky nemôžu byť uložené na rovnakom mieste v tabuľke.
- Možnosti riešenia problému:
  - zreťazenie: Navrhnutie takej štruktúry, ktorá bude schopná uchovávať viacero prvkov s rovnakou rozptylovou hodnotou (vektor ako postupnosť spájaných zoznamov)
  - otvorené adresovanie: Umiestnenie jedného z kolidujúcich prvkov na iné miesto v tabuľke

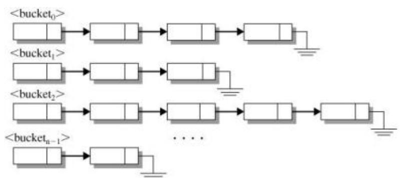


T = vektor jednosmerne spájaných zoznamov, implementujúci rozptýlenú tabuľku

Riešenie kolízií zreťazením
- každý prvok rozptylovej tabuľky T[*i*] obsahuje jednosmerne spájaný zoznam všetkých prvkov/kľúčov (bucket), ktorých rozptýlená hodnota je *i*,
napr. h(x_1) = h(x_4) a h(x_5) = h(x_2)= h(x_7)

postupnosť spájaných zoznamov

- V tomto prípade definujeme rozptýlenú tabuľku ako indexovanú postupnosť (vektor) jednosmerne spájaných zoznamov.
- Každý spájaný zoznam sa nazýva bucket (vedierko, košík, dátová oblasť, sektor) a uchováva prvky s rovnakým indexom (rovnakou rozptylovou hodnotou)



31

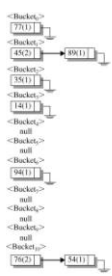
postupnosť spájaných zoznamov

- Vloženie prvku
  - rozptylovou funkciou sa zistí index bucketu v implementujúcom vektore, t.j. miesto vo vektore, kde je ukazovateľ na začiatok zoznamu všetkých prvkov, pre ktorých kľúče vracia rozptylová funkcia tú istú hodnotu
  - Ak je bucket prázdny, vloží sa prvok na prvú pozíciu
  - Ak nie je bucket prázdny, najskôr sa celý prehľadá, či sa v ňom prvok už nenachádza. Ak nie, tak sa vloží na začiatok.

32

postupnosť spájaných zoznamov

Vloženie prvkov: {54, 77, 94, 89, 14, 45, 35, 76}
Veľkosť tabuľky je 11.



33

reprezentácia slovníka pomocou rozptylovej tabuľky so zreťazením

- zreťazený-insert(T, x)
  - vlož x na začiatok zoznamu, na ktorý ukazuje T[h(kľúč[x])]
- zreťazený-delete (T, x)
  - odstráň x zo zoznamu, na ktorý ukazuje T[h(kľúč[x])]
- zreťazený-search(T, k)
  - hľadaj prvok s kľúčom k v zozname T[h(k)]

34

zložitosť operácií rozptylovej tabuľky so zreťazením

- zreťazený-insert(T, x)
  - čas v najhoršom prípade O(1)
- zreťazený-delete (T, x)
  - v najhoršom prípade čas úmerný dĺžke zoznamu v príslušnom vedierku
- zreťazený-search(T, k)
  - v najhoršom prípade čas úmerný dĺžke zoznamu v príslušnom vedierku
  - ak by bol obojstranne zreťazený zoznam: O(1)

35

zložitosť operácií rozptylovej tabuľky so zreťazením

- aký je odhad času úmerného dĺžke zoznamu v príslušnom vedierku?
  - nech má tabuľka T m miest a je v nej zapísaných n prvkov
  - faktor naplnenia α_T = n/m
  - α je priemerný počet prvkov vo vedierku
  - analýzu robíme tak, že čas vyjadríme v závislosti od α.
  - všimnime si, že α môže byť α<1, α=1, α>1.

36

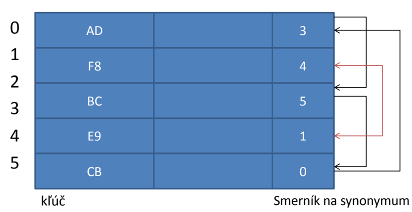
zložitosť operácií rozptylovej tabuľky so zreťazením

- najhorší prípad:
  - všetky prvky sa zreťazia v jedinom zozname/vedierku.
  - zreťazený-search: $\Theta(n)$ + čas potrebný na výpočet hodnoty h
  - horšie než pri reprezentácii slovníka pomocou SLL
- priemerný prípad:
  - závisí od toho, ako dobre h rozptyľuje kľúče na rôzne adresy.
  - predpoklad *jednoduchého rovnomerného rozptýlenia*: ľubovoľný prvok sa rovnako pravdepodobne môže dostať do ktoréhokoľvek vedierka.
  - predpokladajme čas potrebný na výpočet hodnoty h je O(1)
  - zreťazený-search: $\Theta(\alpha + 1)$
  - ak je počet vedierok úmerný počtu prvkov, tj n=O(m), je
    - α = n/m = O(m)/m = O(1)
  - zreťazený-search: O(1)

reprezentácia slovníka pomocou rozptylovej tabuľky so zreťazením

- Nie je obmedzená pevným maximálnym počtom prvkov v tabuľke
- zreťazuje sa vonku - mimo vektora v (ďalšej) zreťazenej voľnej pamäti
- čo tak zreťazovať vnútri vektora?

reprezentácia slovníka pomocou rozptylovej tabuľky s vnútorným reťazením



f(AD) = f(BC) = f(CB) f(F8) = f(E9)

Každé slovo obsahuje aj smerník na synonymum. Všetky synonymá potom môžeme získať postupným prechádzaním vzniknutého spájaného zoznamu

Otvorené adresovanie

- Všetky prvky sa ukladajú priamo vo vektore reprezentujúcom tabuľku (v tabuľke)
- Každá položka tabuľky obsahuje buď prvok reprezentovanej dynamickej množiny (slovníka) alebo NIL.
- Nič sa nezreťazuje, nič nie je mimo tabuľky.
- Miera naplnenia tabuľky α je vždy najviac 1.

Vkladanie pri otvorenom adresovaní

- systematicky sa skúša nájsť prázdne miesto
- postupnosť skúšaných miest nie je daná zreťazením, ale sa vypočítava
- výhody:
  - netreba pamäť na zreťazenie
  - do rovnako veľkého vektora sa zmestí viac prvkov, menej kolízií, rýchlejšie sprístupňovanie

Vkladanie pri otvorenom adresovaní

- Postupnosť skúšaných miest závisí od kľúča, t.j. rozptylová funkcia dostane ďalší parameter *h: U x {0,1,...,m-1} -> {0,1,...,m-1}*
- pre ľubovoľný kľúč *k:*
  - skúšobná postupnosť (m miest/adries) *h(k,0), h(k,1), ..., h(k,m-1)* musí byť permutáciou *0,1,...,m-1*

Hash insert(T, k)

```
insert(T table, k key)
{
        i ← 0

        repeat j ← h(k, i)

                if T[j] = NIL
                        then T[j] ← k
                                return j
                else i ← i + 1

        until i = m

        error "hash table overflow"
}
```


Hash search(T, k)

```
search(T table, k key)
{
        i ← 0

        repeat j ← h(k, i)

                if T[j] = k
                        then return j
                else
                        i ← i + 1

        until T[j] = NIL or i = m

        return NIL
}
```


Hash delete(T, k)

Zmazať prvok nie je také jednoduché. Prečo?

- jednoduché „riešenie", napr. označiť bunku *i* za prázdnu vpísaním NIL, by znemožnilo vyhľadanie prvkov s kľúčom k, pri vkladaní ktorých sa adresa *i* skúšala a nepoužila, lebo bola obsadená.
- možné riešenie:
  - označiť bunku i vpísaním zvláštnej hodnoty deleted (zmazaná)
  - zmeniť vhodne insert aj search

```
search(T table, k key)
{
        i ← 0
        repeat j ← h(k, i)
                if T[j] = k
// ak v T[j] aj hodnota: AND T[j].hodnota <> deleted
                        then return j
                else
                        i ← i + 1
        until T[j] = NIL or i = m
        return NIL
}
```

```
insert(T table, k key)
{
        i ← 0

        repeat j ← h(k, i)

                if T[j] = NIL
                   OR T[j] = deleted
                        then T[j] ← k
                                return j
                else i ← i + 1

        until i = m

        error "hash table overflow"
}
```


Lineárne skúšanie (linear probing)

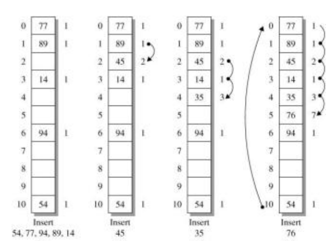
- Vloženie prvku:
  - Na začiatku sa všetky bunky tabuľky označia ako prázdne
  - Aplikuje sa rozptylová funkcia a zvyšok po delení tejto hodnoty veľkosťou tabuľky predstavuje index v tabuľke. Ak je bunka prázdna, vloží sa do nej prvok
  - Inak sa postupne prehľadávajú ďalšie bunky v poradí a prvok sa vloží do prvej voľnej.

Lineárne skúšanie (linear probing)

- Uvažujme bežnú rozptylovú funkciu
  h'(k): U → {0,1….,m-1}
  h(k,i)= (h'(k)+i) mod *m* pre *i= 0,1,...,m-1*
- výhoda: ľahko sa implementuje
- Problém: strapce

lineárne skúšanie

`tableIndex = x % 11`



a) vloženie prvkov na ich príslušné pozície bez posunov
b) prvok 45 sme museli posunúť o jedno miesto kvôli obsadenému miestu 1
c) posunutie 35 až o dve miesta (4 namiesto 2)
d) posunutie prvku 76 až na miesto 5 namiesto pôvodného miesta 10

lineárne skúšanie - algoritmus

```
//výpočet indexu tabuľky
int index = (item.hashCode()&Integer.MAX_VALUE)%n

// uloženie pôvodného indexu
int origIndex = index;

//cyklické prehľadávanie tabuľky a hľadanie voľnej pozície
// nájde miesto alebo sa tabuľka zaplní(origIndex == index).
do
{
    if table[index] is empty
       insert item in table at table[index] and return
    else if table[index] matches item
       return
    // posunutie v tabuľke
    index = (index+1) % n;
}
while (index != origIndex);
throw new BufferOverflowException();
```


lineárne skúšanie

- Táto metóda je vhodná v prípade, ak veľkosť vektora implementujúceho tabuľku je rádovo väčšia ako počet prvkov, ktoré sa do nej budú vkladať.
- Dobrá rozptylová funkcia minimalizuje kolízie a ak aj nastanú, tak v tabuľke bude dostatok voľných miest pre vyhľadanie náhradnej pozície

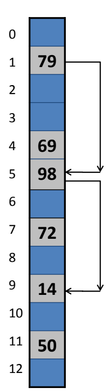
kvadratické skúšanie (quadratic probing)

- Uvažujme bežnú rozptylovú funkciu $h'(k): U \rightarrow \{0,1\ldots.,m-1\}$
- $h(k,i)= (h'(k)+c_1.i+c_2.i^2)$ mod m pre $i= 0,1,...,m-1$
- Problém: ako zvoliť c_1 a c_2
- Strapce vznikajú sekundárne, menej

dvojité rozptýlenie

- Uvažujme bežné rozptylové funkcie
  $h_1(k): U \rightarrow \{0,1\ldots.,m-1\}$
  $h_2(k): U \rightarrow \{0,1\ldots.,m-1\}$

- $h(k,i)= (h_1(k)+i.h_2(k))$ mod m pre $i= 0,1,...,m-1$



rozptylová funkcia - príklad

Vkladanie pomocou dvojitého rozptýlenia

- máme rozptýlenú tabuľku s veľkosťou 13 s $h_1(k) = k \bmod 13$ a $h_2(k) = 1 + (k \bmod 11)$

- ak 14 ≡ 1 (mod 13) a 14 ≡ 3 (mod 11), tak kľúč 14 sa vloží na prázdne miesto 9 po tom, ako sa zistilo, že miesta 1 a 5 sú obsadené

dvojité rozptýlenie

- postupnosť skúšaných miest závisí dvojako od kľúča k:
  - začiatočné miesto skúšania $T[h_1(k)]$
  - posunutie je vždy o $h_2(k)$, to celé modulo m
- hodnoty $h_2(k)$ nesmú byť súdeliteľné s m.
  - Prečo? ak by pre nejaký kľúč k mali najväčšieho spoločného deliteľa d>1, tak pri hľadaní kľúča k by sa prezerala iba 1/d tabuľky
  - ako to zabezpečiť?
    - návrh 1: $m=2^s$, pre nejaké s, h_2 nech vždy vracia nepárne číslo
    - návrh 2: m je prvočíslo, h_2 nech vždy vracia kladné číslo menšie než m

dvojité rozptýlenie

- návrh 2: m je prvočíslo, h_2 nech vždy vracia kladné číslo menšie než m
- napr
  - *$h_1(k)$ = k mod m*
    $h_2(k)$ = 1 + (k mod m'), kde *m'* je len o trochu menšie než *m*, povedzme *m-1* alebo *m-2*
  - *napr ak k = 123456 a m = 701, m' = 700:*
    - *$h_1(k)$ = 80, $h_2(k)$ = 257*
    - *takže ako prvé sa skúša 80. miesto a potom ďalšie vždy vzdialené o 257 miest (modulo 701).*

55

dvojité rozptýlenie

- je lepšie než lineárne alebo kvadratické rozptýlenie:
- pri sprístupňovaní sa skúša $\Theta(m^2)$ postupností na rozdiel od $\Theta(m)$
- prečo?
  - každá možná dvojica (*$h_1(k)$, $h_2(k)$*) generuje rôznu postupnosť adries na skúšanie
  - ako sa mení k, začiatok aj posunutie skúšania sa menia nezávisle.
- blíži sa ideálnemu predpokladu rovnomerného rozptýlenia

56

zložitosť operácií rozptylovej tabuľky s otvoreným adresovaním

- (veta) Ak je faktor naplnenia tabuľky α=n/m, α<1 a používa sa rovnomerné rozptýlenie, tak očakávaný počet skúšaní pri neúspešnom hľadaní je najviac 1/(1- α).
- ak α je konštanta, tak neúspešné hľadanie trvá konštantný čas, t.j. O(1).
  - napr. ak je tabuľka poloprázdna/poloplná, tak priemerný počet skúšaní je 1/(1-0.5)=2
  - ak je plná na 90%, tak 1/(1-0.9)=10

57

zložitosť operácií rozptylovej tabuľky s otvoreným adresovaním

- (dôsledok) Ak je faktor naplnenia tabuľky α=n/m, α<1 a používa sa rovnomerné rozptýlenie, tak vloženie prvku do tabuľky s otvoreným adresovaním si vyžaduje v priemere najviac 1/(1- α) skúšaní.

58

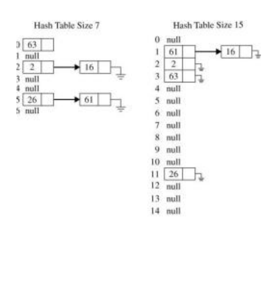
zložitosť operácií rozptylovej tabuľky s otvoreným adresovaním

- (veta) Ak je faktor naplnenia tabuľky α=n/m, α<1 a používa sa rovnomerné rozptýlenie a hľadanie každého kľúča je rovnako pravdepodobné, tak očakávaný počet skúšaní pri úspešnom hľadaní je najviac 1/α.ln(1-α)+1/ α.
  - napr. ak je tabuľka poloprázdna/poloplná, tak priemerný počet skúšaní je menej než 3.387
  - ak je plná na 90%, tak 3.670

59

prerozptýlenie/rehašovanie

- So zvyšujúcim sa počtom prvkov v rozptylovej tabuľke (a zároveň so zvyšujúcim sa počtom kolízií) sa efektívnosť vyhľadávania znižuje.
- Prerozptýlenie predstavuje zväčšenie veľkosti tabuľky, ak súčasná je zaplnená do určitej úrovne.

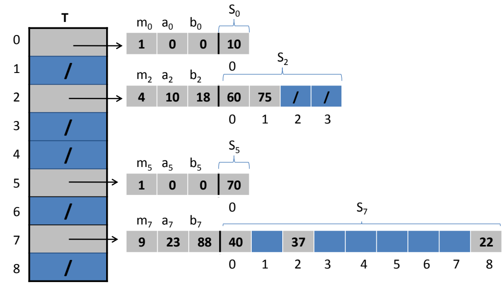


60

ďalšie čítanie...

perfektná rozptylová funkcia - príklad

Použitie perfektného rozptýlenia (perfect hashing) na uloženie množiny K = {10, 22, 37, 40, 60, 70, 75}



perfektná rozptylová funkcia - príklad

Použitie perfektného rozptýlenia (perfect hashing) na uloženie množiny K = {10, 22, 37, 40, 60, 70, 75}

-Základná rozptylová funkcia je *h(k) = ((ak + b) mod p) mod m*, kde *a* = 3, *b* = 42, *p* = 101 a *m* = 9

-Príklad: h(75) = 2, takže objekt 75 sa uloží na miesto s kľúčom 2

-Sekundárna hešovacia tabuľka S_j obsahuje všetky kľúče hešujúce index j

  -jej veľkosť je m_j

  -a hešovacia funkcia je $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$

-Keď $h_2(75) = 1$, kľúč 75 je uložený na miesto 1 v sekundárnej hešovacej tabuľke S2

-Takto nie sú žiadne kolízie v sekundárnych hešovacích tabuľkách a vyhľadávanie trvá v najhoršom prípade konštantný čas