Tomáš Oberhuber

Faculty of Nuclear Sciences and Physical Engineering
Czech Technical University in Prague

Hešování

- Budeme chtít ukládat data (=záznamy s klíčem) tak, abychom je dokázali vyhledávat se složitostí $O(1)$.
- Chceme najít tzv. **hešovací funkci** h, která bude mapovat klíče na indexy tabulky fixní velikosti.
- Je-li h **prosté zobrazení** z množiny klíčů do množiny indexů, pak h je **perfektní hešovací** funkce.
- Problém je v tom, že počet ukládáných záznamů často není známý dopředu, ale velikost tabulky musím být fixní.

Example 1

Hešování názvů proměnných během překladu zdrojových kódů:
Jak konstruovat hešovací funkci h?

- Např. podle prvních dvou znaků - to je pro ASCII kódování $31 \times 31 = 961$ indexů - to by třeba mohlo stačit.
- Pak ale platí

$$h(\text{"}abc\text{"}) = h(\text{"}abd\text{"}).$$

Hešování

- Pokud se dva klíče mapují na stejnou hondotu, jde o tzv. **kolizi**.
- Právě schopnost vypořádat se s kolizemi je to nejpodstatnější na hešování.

Modulo

- Modulo funkce je definována jako

$$h(K) = K \mod m,$$

  kde K je klíč a m je velikost tabulky.
- Nejlepší je volit m jako velké prvočíslo.

Example 2

k	4	8	12	16	20	24	28	32
$m = 8$	4	0	4	0	4	0	4	0
$m = 7$	4	1	5	9	6	3	0	4

- Pokud velikost tabulky m nelze volit jako prvočíslo, pak lze volit h jako

$$h(K) = (K \mod p) \mod m,$$

kde p je velké prvočíslo.

Folding

- Klíč rozdělíme na několik částí a ty pak kombinujeme dohromady:

$$\begin{aligned} 123456789 &\rightarrow 123 + 456 + 789 = 1368 \rightarrow 1368 \mod m, \\ 123456789 &\rightarrow 123 + 654 + 789 = 1566 \rightarrow 1566 \mod m. \end{aligned}$$

- Reálně se místo sčítání používá bitové XOR.
- To lze použít i na řetězce:

$$h(\text{"}abcd\text{"}) = \text{"}a\text{"} \ XOR \ \text{"}b\text{"} \ XOR \ \text{"}c\text{"} \ XOR \ \text{"}d\text{"},$$

nebo

$$h(\text{"}abcd\text{"}) = \text{"}ab\text{"} \ XOR \ \text{"}cd\text{"}.$$

Mid-square

- Např. (lze použít pro $m = 1000$)

$$h(3121) \to 3121^2 = 9740641 \to 406.$$

- Např. pro $m = 1024$ by šlo využít binární zápis.

Extrakce

-

$$123456789 \to 1289$$

Radixová transformace

- Index zapíšeme v soustavě s jiným základem

$$345 = 423_9 \rightarrow h(345) = 423 \mod m$$

Otevřené adresování (Open adressing)

- V případě kolize použijeme tzv. **probing function** p.
- Zkoušíme postupně indexy

$$\begin{aligned} (h(K) + p(1)) \mod m \quad &, \\ (h(K) + p(2)) \mod m \quad &, \\ &\vdots \\ (h(K) + p(i)) \mod m \end{aligned}$$

- i se označuje jako sonda (probe).

Řešení kolizí

- Při vkládání procházím celou poslounost indexů a hledám, kde je v tabulce volné místo - tam pak vložím záznam spolu s klíčem.
- Při hledání procházím opět celou posloupnost indexů a hledám, kde v tabulce je záznam s hledaným klíčem - končím, pokud narazím na prázdné políčko.
- Při mazání prvku označím políčko jako tzv. **tombstone**.
  - Na tombstone lze vkládat jiný záznam a klíč.
  - Pokud při hledání narazím na tombstone, pokračuju v hledání dál.

Lineární sondování (Linear probing)

- Pro lineární sondování je
$$p(i) = i.$$
- Lineární sondování má tendenci vytvářet shluky obsazených políček, což není žádoucí.

Example 3

	0	1	2	3	4	5	6	7	8	9	10
$a, b, c \rightarrow$			b	c		a					
$d, e, f \rightarrow$			b	c	f	a	d				e
$g, h \rightarrow$	g		b	c	f	a	d	h			e

Kvadratické sondování (Quadratic probing)

- Pro kvadratické sondování máme

$$p(i) = h(K) + (-1)^{i-1} \left\lfloor \frac{i+1}{2} \right\rfloor^2,$$

což vlastně generuje posloupnost

$$h(K) + i^2, h(K) - i^2, i = 1, 2, \ldots, \left\lfloor \frac{m-1}{2} \right\rfloor.$$

konkrétně tedy

$$h(K) + 1, h(K) - 1, h(K) + 4, h(K) - 4, \ldots$$

a vše $\mod m$.

Example 4

	0	1	2	3	4	5	6	7	8	9	10
$a, b, c \rightarrow$			b	c		a					
$d, e, f \rightarrow$			b	c		a	d		f		e
$g, h \rightarrow$	g		b	c		a	d		f	h	e

Kvadratické sondování také generuje shluky, ale už ne tak velké.

Řešení kolizí

- Kvadratická sonda nemůže mít pouze tvar $p(i) = i^2$.
- Např. pro $h(K) = 9$ a $m = 19$ dostáváme následující posloupnost

$$\underbrace{9, 10, 13, 18, 6, 15, 7, 1, 16, 14}_{I.}, \underbrace{14, 16, 1, 7, 15, 6, 18, 13, 10}_{II.}.$$

- Zde první polovina pokrývá jen půl políček celé tabulky a druhá polovina pokrývá ta stejná políčka.

- Sondy pokrývající stejná políčka odpovídají indexům

$$i = \frac{m}{2} + 1 \text{ a } j = \frac{m}{2} - 1,$$

a platí

$$\begin{aligned}(i^2 - j^2) &= \left(\frac{m}{2} + 1\right)^2 - \left(\frac{m}{2} - 1\right)^2 \\ &= \frac{m^2}{4} + m + 1 - \frac{m^2}{4} + m - 1 \\ &= 2m\end{aligned}$$

a tady

$$(i^2 - j^2) \mod m = 2m \mod m = 0.$$

Náhodně generovaná čísla

- Pro hešování lze také použít generátor pseudonáhodných čísel.
- Podle klíče K zvolíme inicializační seed a následně generujeme pseudonáhdnou posloupnost, tj.

$$p(i) = rand(i)$$

Dvojité hešování (double hashing)

- Použije se druhá hešovací funkce h_p.
- Následně se generuje posloupnost

$$(h(K) + ih_p(K)) \mod m \text{ pro } i = 0, 1, \ldots$$

- Pokud je pro nějaké $K_1 \neq K_2$

$$h(K_1) = h(K_2) \text{ a } h_p(K_1) \neq h_p(K_2),$$

pak je vznik shluků téměř eliminován.

- Vyčíslení h a h_p může být výpočetně náročné.
- Lze si pomoci volbou posloupnosti

$$h(K) + ih(K) + 1$$

Example 5

Pro $l = 1$ dostáváme

$$\begin{array}{l|llll} h(K_1) = j & j & 2j+1 & 3j+1 & \dots \\ h(K_2) = 2j+1 & 2j+1 & 4j+3 & 6j+4 & \dots \end{array}$$

Efektivita jednotlivých sondování

Neúspěšné hledání:

- Hledáme klíč, který v tabulce není.
- Pro $h(K) = j$ musíme projít indexy

$$j + p(i) \mod m \text{ pro } i = 0, 1 \ldots$$

  dokud nenarazíme na prázdné políčko v tabulce nebo opět na index j.
- To je úměrné délce shluků nebo počtu prvků v tabulce.

Úspěšné hledání:

- Hledáme klíč, který v tabulce nalezneme.

Pokud LF označuje tzv. *loading factor*, tj.

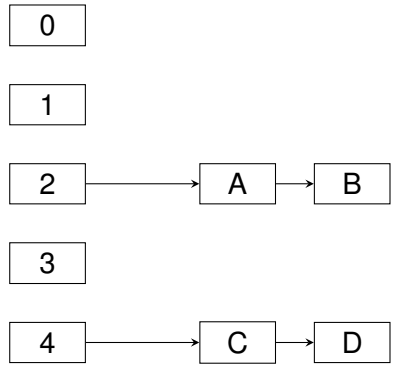
$$LF = \frac{\text{počet prvků v tabulce}}{m},$$

pak pro průměrný počet kroků při úspěšném a neúspěšném hledání platí:

	lineární	kvadratické	dvojité
Neúspěšné hledání	$\frac{1}{2}\left(1 + \frac{1}{(1-LF)^2}\right)$	$\frac{1}{1-LF} - LF - \ln(1-LF)$	$\frac{1}{1-LF}$
Úspěšně hledání	$\frac{1}{2}\left(1 + \frac{1}{1-LF}\right)$	$1 - \ln(1-LF) - \frac{LF}{2}$	$\frac{1}{LF} \ln \frac{1}{1-LF}$

Řetězení (chaining)

- Buňky tabulky jsou spojové seznamy.



Přihrádkové adresování (bucket adressing)

- Každé políčko hešovací tabulky tvoří celý blok buněk.
- V případě kolize, se záznamy a klíče ukládají do buněk následujících v daném bloku.
- Pokud se zaplní celý blok, lze použít otevřené adresování.

Kukaččí hešování (Cuckoo hashing)

- Podstatou je použít dvě hešovací funkce h_1 a h_2.
- Tím získáme pro každý klíč K dvě různé pozice $h_1(K)$ a $h_2(K)$.
- Při vkládání se používá hladový algoritmus:
  - Vkládaná prvek se vloží na jednu ze dvou pozic.
  - Pokud ta je již obsazena, vyhodí prvek na této pozici a vloží ho na jeho alternativní pozici.
  - Pokud ta je obsazena, proces se opakuje.
- Při hledání se stačí podívat pouze na obě pozice $h_1(K)$ a $h_2(K)$.
- To je hlavní výhoda oproti jiným hešovacím metodám.

Perfektní hešování

- Jsou-li hešovaná data známa předem, lze sestrojit **perfektní hešovací funkci**, se kterou nedochází ke kolizím.
- Pokud lze navíc splnit $m = n$ mluvíme o **minimální perfektní hešovací funkci**.
- Příklady fixně daných dat jsou:
  - Klíčová slova programovacího jazyka.
  - Seznam souborů na CD/DVD :-).

Cichelliho metoda

- Pro slovo $w = w_1 \dots w_l$ je hešovací funkce definována jako

$$h(w) = (l + g(w_1) + g(w_l)) \mod m.$$

- Předpokládáme tedy, že žádná dvě slova nemají stejnou délku a shodné první a poslední písmeno.
- Metodu budeme demonstrovat na klasickém příkladu 9 můz.
  - A. Drozdek, *Data Structures and Algorithms in C++*, Cengage Learning, 2012.

Perfektní hešování

- Máme následující slova:
  - Calliope, Clio, Erato, Euterpe, Melpomene, Polyhymnia, Terpsichore, Thalia, Urania.
- Spočítáme výskyty jednotlivých písmen na začátku a na konci slov:

E	*A*	*C*	*O*	*T*	*M*	*P*	*U*
6	3	2	2	2	1	1	1

- Klíčová slova seřadíme podle počtu výskytů prvního a posledního písmene:

Euterepe	$6 + 6 = 12$
Calliope	$2 + 6 = 8$
Erato	$6 + 2 = 8$
Terpsichore	$2 + 6 = 8$
Melpomene	$1 + 6 = 7$
Thalia	$2 + 3 = 5$
Clio	$2 + 2 = 4$
Polyhymnia	$1 + 3 = 4$
Urania	$1 + 3 = 4$

Perfektní hešování

- V následujícím kroku kostruujeme funkci g.
- Jednotlivým písmenům postupně přiřazujeme hodnoty $0, 1, 2 \ldots N_{max}$ a zkoušíme, jestli u výsledné hešovací funkce nedojde ke kolizi.
- Pokud ano, rekurzivně se vracíme zpět pomocí backtrackingu a zvyšujeme hodnoty již dříve namapované na jednotlivá písmena.
- V našem případě použijeme $N_{max} = 4$

Perfektní hešování

			rezervované heše
Euterpe	$g(E) = 0$	$h = 7$	$\{7\}$
Calliope	$g(C) = 0$	$h = 8$	$\{7, 8\}$
Erato	$g(O) = 0$	$h = 5$	$\{5, 7, 8\}$
Terpsichore	$g(O) = 0$	$h = 2$	$\{2, 5, 7, 8\}$
Melpomene	$g(M) = 0$	$h = 0$	$\{0, 2, 7, 8, 5\}$
Thalia	$g(A) = 0$	$h = 6$	$\{0, 2, 6, 7, 8, 5\}$
Clio		$h = 4$	$\{0, 2, 4, 6, 7, 8, 5\}$
Polyhymnia	$g(P) = 0$	$h = 1$	$\{0, 1, 2, 4, 6, 7, 8, 5\}$
Urania	$g(U) = 0$	$h = 6 \times$	$\{0, 1, 2, 4, 6, 7, 8, 5\}$
Urania	$g(U) = 1$	$h = 7 \times$	$\{0, 1, 2, 4, 6, 7, 8, 5\}$
Urania	$g(U) = 2$	$h = 8 \times$	$\{0, 1, 2, 4, 6, 7, 8, 5\}$
Urania	$g(U) = 3$	$h = 0 \times$	$\{0, 1, 2, 4, 6, 7, 8, 5\}$
Urania	$g(U) = 4$	$h = 1 \times$	$\{0, 1, 2, 4, 6, 7, 8, 5\}$
Polyhymnia	$g(P) = 1$	$h = 2 \times$	$\{0, 2, 4, 6, 7, 8, 5\}$
Polyhymnia	$g(P) = 2$	$h = 3 \times$	$\{0, 2, 3, 4, 6, 7, 8, 5\}$
Urania	$g(U) = 0$	$h = 6 \times$	$\{0, 2, 3, 4, 6, 7, 8, 5\}$
Urania	$g(U) = 1$	$h = 7 \times$	$\{0, 2, 3, 4, 6, 7, 8, 5\}$
Urania	$g(U) = 2$	$h = 8 \times$	$\{0, 2, 3, 4, 6, 7, 8, 5\}$
Urania	$g(U) = 3$	$h = 0 \times$	$\{0, 2, 3, 4, 6, 7, 8, 5\}$
Urania	$g(U) = 4$	$h = 1 \times$	$\{0, 1, 2, 3, 4, 6, 7, 8, 5\}$

Výsledkem je:

A	C	E	O	M	T	P	U
0	0	0	0	0	0	2	4

- Algoritmus má exponenciální složitost.
- Algoritmus nezaručuje úspěšnou konstrukci perfektní hešovací funkce.
- Pokud se konstrukce nevydaří, je potřeba zvýšit hodnotu N_{max}.

Hešování s proměnnou velikostí

Nyní se budeme zabývat úlohou, když m není fixní.

Extendible hashing (Fagin 1979)

- Tabulka se rozdělí na proměnný počet přihrádek (buckets) fixní velikosti.
- Přístup do tabulek se provádí nepřímo pomocí dynamicky upravovaného indexu.
- Hešovací funkce vrací tzv. pseudoklíč, což je pozice v indexu a ne rovnou v tabulce samotné.
- Pokud zvětšíme tabulku, změní se jen index a ne tabulka.

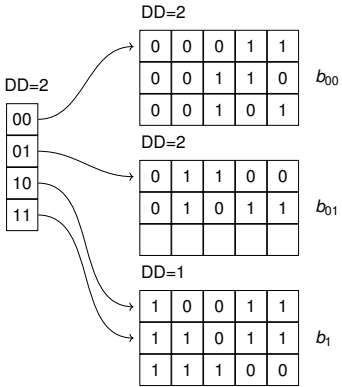
Hešování s proměnnou velikostí

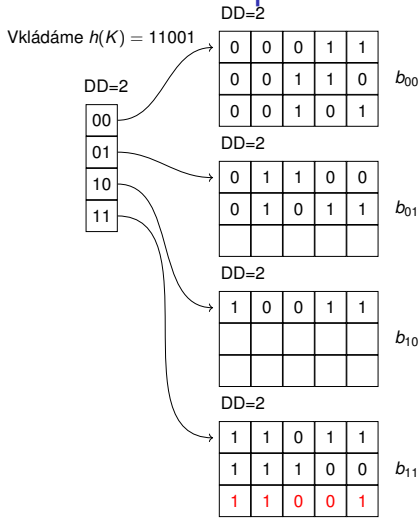
Příklad:

- Hešovací funkce $h(K)$ vrací 5 bitů dlouhý klíč.
- **Hloubka indexu (directory depth)**, DD určuje velikost indexu/adresáře.
- **Lokální hloubka, Local depth**, LD určuje, kolik bitů se skutečně použije pro indexování v této tabulce.
- Je-li $LD < DD$, pak k jedné přihrádce povede více různých odkazů z indexu/adresáře.
- Pokud se pak taková tabulka zaplní, rozdělí se na dvě části, část klíčů se překopíruje a lokální hloubka se změní.

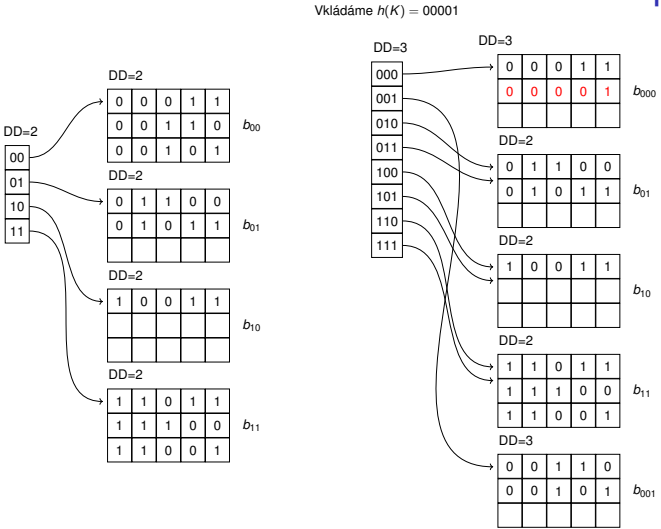
Na podobném principu funguje i systém dynamic **address translation, DAT** v operačních systémech.

Hešování s proměnnou velikostí





Hešování s proměnnou velikostí



Lineární hešování

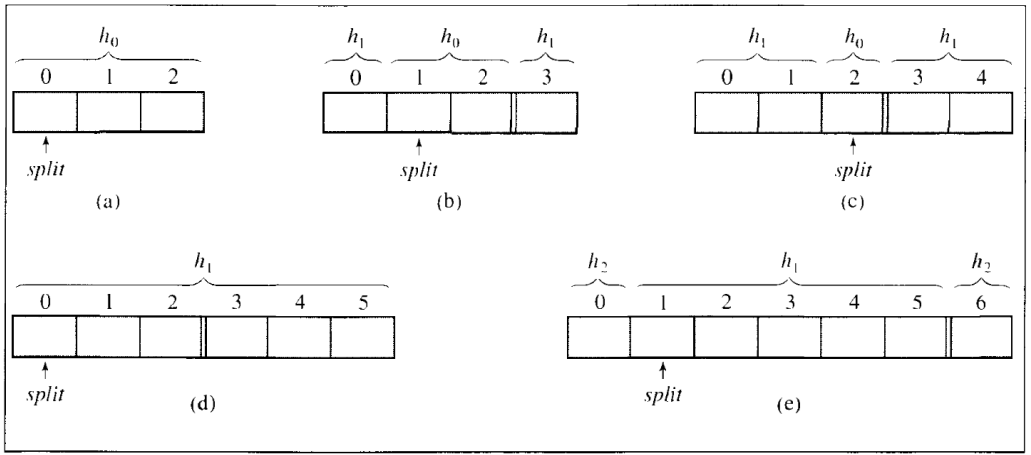
- Tato metoda nevyžaduje žádný index/adresář.
- Nové přihrádky, které vznikají rozdělením, se vkládají do lineárního pole.
- Na počátku máme m přihrádek fixní velikosti, do kterých vkládáme záznamy s klíči.
- Pokud se některá příhrádka zaplní, použije se tzv. *overflow buffer* (OFB) ve formě spojového seznamu.
- Celkové zaplnění se pak počítá jako

$$Lf = \frac{\text{počet vložených prvků}}{\text{počet slotů} + |OFB|}$$

Hešování s proměnnou velikostí

- Pokud zaplnění přesáhne určitou mez (např. 0.8), provede se rozdělení (split).
- Rozdělení přihrádek se ale provádí v předem daném pořadí a ne podle toho, která přihrádka je nejvíce zaplněná.
- Rozdělení provádíme postupně přiohrádku po přihrádce, někdy se dělí i nejméně zaplněná přhrádka.
- Pořadí určuje ukazatel *s* který se postupně posouvá od první přihrádky k dalším.
- Rozdělení vytvoří na konci pole novou přihrádku a záznamy jsou přerozděleny mezi původní a novou přihrádku.

Hešování s proměnnou velikostí



Hešování s proměnnou velikostí

- Rozdělením příhrádek dojde ke změně hešovací funkce.
- U každé přihrádky si pamatujeme tzv. úroveň rozdělení - l.
  - Při dělení přihrádky se l zvyšuje o jedna u púvodní příhrádky...
  - ... a na stejnou hodnotu se nastaví l nově vzniklé přihrádky.
- Index l pak udává, jaká hešovací funkce bude použita k hešovaní klíčů v dané přihrádce.
- Pro hešovací funkci platí

$$h_l(K) = K \mod (2^l m)$$

- V daný moment se mohou vyskytovat jen přihrádky na úrovni l a $l - 1$.
- Pokud je $h_l(K) < s$, pak $h(K) = h_{l+1}(K)$, jinak $h(K) = h_l(K)$

Hešování s proměnnou velikostí

- Nejprve máme $m = 3$ a $s = 0$ - obrázek (a).
- Pokud je u některé přihrádky zaplnění větší, než určitá úroveň:
  - vytvoří se nová přihrádka s indexem 3,
  - a klíče z přihrádky 0 jsou rozděleny mezi přihrádky 0 a 3,
  - ukazatel s se posune o jedna dále.
- Přihrádka 0 a 3 jsou nyní hešovány pomocí funkce h_1.
- Přihrádka 1 a 2 jsou nyní hešovány pomocí funkce h_0.
- Při dalším přeplnění kterékoliv přihrádky se stějně rozdělí příhrádka 1.

Hešování s proměnnou velikostí