Algoritmy 2 - hašovací tabulky

Petr Osička

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

- Stačí nám pouze operace vkládání a vyhledávání (případně mazání). (*tedy ne dotazy na mininimum, maximum atd.*)
- zobecnění pole: rozdíl je ve způsobu adresování. Místo množiny klíčů $\{0, 1, 2, \ldots, m-1\}$, které jsou vlastně indexy do pole, máme obecnou množinu klíčů U (které musíme umět porovnat na rovnost).
- Hlavní idea: hašovací funkce

  $$h : U \to \{0, 1, 2, \ldots, m-1\},$$

  pro přepočet obecného klíče na index v tabulce.
- Často se vyskytuje v programovacích jazycích `dict` v Pythonu, `hash-table` v CommonLispu, `map` v GoLangu.
- V nejhorším případě lineární složitost vyhledávání, v průměrném případě konstatní složitost.

```
struct Table
    data: []Key — pole klíčů
    hash-function: (Key) → Int — hašovací funkce
```

Kostra přístupu k místu, kde je v tabulce `T` klíč `k`

1. spočítáme index: `i ← T.hash-function(k)`
2. přistoupíme k prvku: `T.data[i]`

Složitost je konstantní (vzhledem k počtu klíčů v tabulce).

Problém: může být potřeba vložit klíč na již obsazené políčko

Kolize

Pro množinu klíčů U, velikost tabulky m a hašovací funkci h kolize nastane pokud

$$\text{existují } x, y \in U \text{ tak, že } x \neq y, \text{ ale } h(x) = h(y).$$

- Pokud $|U| > m$, pak nelze navrhnout h tak, aby kolize nikdy nenastala. (Dirichletův princip.)
- Pravděpodobnost toho, že nastane kolize v rámci podmnožiny klíčů (s méně než m prvky) lze někdy odhadnout.

Příklady

Předpokládejme konečnou množinu U. Označíme

$$b_i = |\{x \in U \mid h(x) = i\}|$$

Počet kolizí v rámci indexu i je

$$\frac{b_i \cdot (b_i - 1)}{2}.$$

Možných kolizí celkem je

$$\sum_{i=0}^{m-1} \frac{b_i \cdot (b_i - 1)}{2}$$

Položíme $|U| = 300$ a $m = 3$.

1. $b_0 = b_1 = b_2 = 100$.

   Počet kolizí je $3 \cdot \frac{100 \cdot 99}{2} = 14850$

2. $b_0 = 300, b_1 = b_2 = 0$

   Počet kolizí je $\frac{300 \cdot 299}{2} = 44850$

3. $b_0 = 200, b_1 = 80, b_3 = 20$

   Počet kolizí je $\frac{200 \cdot 199}{2} + \frac{80 \cdot 79}{2} + \frac{20 \cdot 19}{2} = 23250$

Minimum: pokud pro každé $i \neq j$ je $|b_i - b_j| \leqslant 1$.

Pravděpodobnostní pohled

Pro $x \in U$ je dána pravděpodobnost $p(x)$, že x vybereme.

Pokus: vzorkujeme $x \in U$ a vypočítáme $h(x)$.

Pravděpodobnost, že výsledek pokusu je i, je

$$P[h(x) = i] = \sum_{x:h(x)=i} p(x)$$

Pokud platí, že pro každé $0 \leqslant i < m$ je $P[h(x) = i] \approx 1/m$, pak h označujeme jako uniformní hašhovací funkci.

Za předpokladu $p(x) = \frac{1}{|U|}$ je na předchozím slajdu funkce z případu 1 uniformní.

Narozeninový paradox

Předpokládejme uniformní hashovací funkci h a dostatečně velkou množinu klíčů.

Jaká je pravděpodobnost, vybereme-li $n \leqslant m$ klíčů, že dojde ke kolizi (= existuje alespoň jedna dvojice vybraných klíčů tak, že dojde ke kolizi).

Pravděpodobnost, že nedojde k žádné kolizi je

$$\overline{p}(n) = 1 \cdot (1 - \frac{1}{m}) \cdot (1 - \frac{2}{m}) \cdot \cdots \cdot (1 - \frac{n-1}{m}).$$

Po úpravách dostaneme

$$\overline{p}(n) = \frac{m!}{m^n \cdot (m-n)!}$$

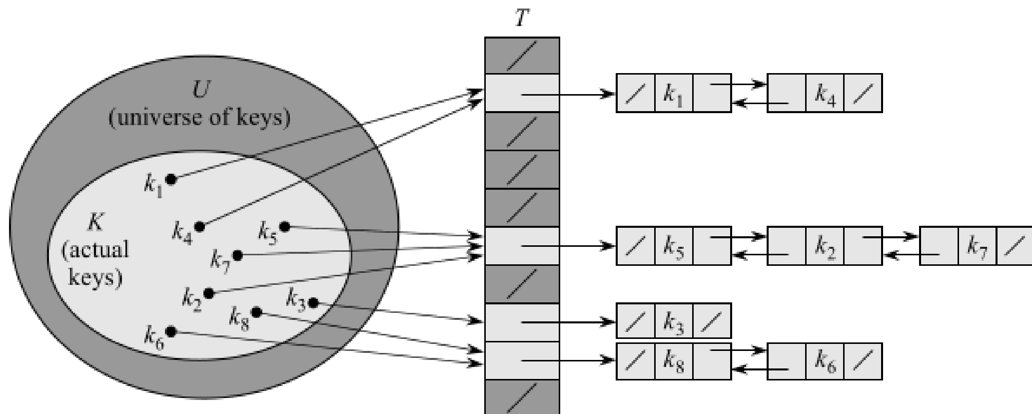
Odpověď na naši otázku je $1 - \overline{p}(n)$.

m / n	2	3	5	10	20	30
3	0.33	0.77				
100	0.01	0.02	0.09	0.37	0.86	0.99
200	0.00	0.01	0.04	0.20	0.62	0.89
365	0.00	0.01	0.03	0.11	0.41	0.71

pst / m	10	50	100	200	500
0.1	2	4	6	7	11
0.2	3	6	8	10	16
0.5	5	9	13	17	27
0.7	6	12	16	23	35
0.9	7	15	22	31	48

Řešení kolizí pomocí řetězení

Klíče, které se zahašují na stejný index ukládáme do seznamu (např. oboustranného se zarážkou, lze upravit i pro jiné seznamy).



```
insert
 ⟵ T: Table
 ⟵ k: Key
```

```
1  i ← T.hash-function(k)
2  list-insert(T.data[i], k)
```

```
search
 ⟵ T: Table
 ⟵ k: Key
```

```
1  i ← T.hash-function(k)
2  return list-search(T.data[i], k)
```

```
delete
 ⟵ T: Table
 ⟵ k: Key
```

```
1  i ← T.hash-function(k)
2  list-delete(T.data[i], k)
```

Složitost v nejhorším případě

Klíčová je operace `search`. Ostatní je v konstantním čase (za předpokladu, že `list-insert` přidává na začátek seznamu)

V nejhorším případě je složitost $O(n)$, kde n je počet klíčů v tabulce. Nejhorší případ:

- Všechny klíče v tabulce jsou v jednom seznamu.
- Vyhledávaný klíč vyhledáváme v tomto seznamu.
- Hledaný klíč se v tabulce nenachází (nebo je v seznamu poslední).

Složitost v průměrném případě

O tabulce velikosti m obsahující n klíčů říkáme, že má faktor zaplnění $\frac{n}{m}$.

Věta

Pokud je `T.hash` uniformní hašovací funkce, je průměrná složitost vyhledávání v tabulce `T` obsahující n klíčů $\Theta(1 + \frac{n}{m})$.

Kostra důkazu Délku seznamu `T.data[j]` označíme n_j.
Očekávaná délka tohoto seznamu je

$$E[n_j] = \sum_{i=1}^{n} 1 \cdot \frac{1}{m} = \frac{n}{m}$$

Pro všechny indexy j platí, že se do nich hledaný klíč zahašuje se stejnou pravděpodobností.
Při neúspěšném vyhledávání se musí prohledat celý seznam. Při úspěšném vyhledávání musíme v průměru prohledat půlku seznamu. □

Vidíme tedy, že pokud máme $n = O(m)$ (tj. ex. konstanta c tak, že $n < cm$), pak je průměrná složitost vyhledávání konstantní.

Pro návrh tabulky s řetězením to znamená následující:

- Vybereme konstantu c. To odpovídá délce seznamu, kterou jsme ještě ochotni snést.
- Tabulka pak má kapacitu cm a více prvků do ní nepřidáváme (je to podobné fixnímu poli).
- Alternativně při přidání klíče do tabulky, která obsahuje cm klíčů, tuto tabulku zvětšíme a všechny její klíče znovu přidáme (je to analogie dynamického pole). Amortizovaná složitost přidání bude pořád konstantní.

Konstrukce hašovacích funkcí

- Hašovací funkce závisí na typu klíče
- Typicky obsahuje process převodu klíče na číslo v nějakém rozsahu

Příklady

- Klíče jsou desetinná čísla z intervalu $[0, 1)$: $x \mapsto \lfloor x \cdot m \rfloor$ Například pro $m = 97$ máme $0.1 \mapsto \lfloor 0.1 \cdot 97 \rfloor = 9$.
- Klíče z intervalu $[s, t)$: $x \mapsto \lfloor \frac{x-s}{t-s} \cdot m \rfloor$
- w-bitová celá čísla: $x \mapsto \lfloor \frac{x}{2^w} \cdot m \rfloor$

Obecnější postup

1. Klíč k bereme jako posloupnost číslic v nějaké číselné soustavě (např. jako posloupnost bajtů = číslic v soustavě o základu 256)
2. Tuto posloupnost transformujeme na číslo v rozumném rozsahu (typicky dáno velikostí přirozeného slova v počítači).
3. Číslo získané v předchozím případě transformujeme na index z množiny $\{0, \ldots, m-1\}$.
   - *Dělící metoda:* $x \mapsto x \mod m$
   - *Metoda násobení:* $x \mapsto \lfloor m \cdot (xA - \lfloor xA \rfloor) \rfloor$ pro $0 < A < 1$.

Příklad

Klíče jsou třípísmenné ASCII řetězce, které interpretujeme jako číslo o základu 128.

$$\texttt{now} \sim \texttt{110 111 119} \mapsto 119 + (111 * 128) + (110 * 128^2) = 1816567$$

Použijeme dělící metodu a $m = 64$, $m = 31$.

	64	31
now	55	29
for	50	20
tip	48	1
ilk	43	18
dim	45	21
tag	39	22

	64	31
sob	34	26
nob	34	8
cab	34	24
sky	57	2
jay	57	4
joy	57	29

U dělící metody není dobré, když je velikost tabulky mocnina 2, protože pak výsledek záleží pouze na hodnotě spodních bitů (jejich počet odpovídá exponentu).

Chceme, aby hodnota závisela na celém klíči (ne jenom na vybraných bitech).

Proto je dobrá volba velikosti tabulky prvočíslo. Typicky se vybírá největší prvočíslo menší než mocnina 2.

2^6	53
2^7	97
2^8	193
2^9	389
2^{11}	1543
2^{16}	49157

Příklad první části (reálně používaná funkce, v jazyce C)

```c
unsigned long leeroy_jenkins(unsigned char *r, int n) {
    unsigned long h = 0;
    int cursor = 0;

    while (cursor < n) {
        h += r[cursor];     // prictu k h hodnotu bajtu
        h += h << 10;       // posunu h 0 10 bitu doleva
        h ^= h >> 6;        // prixorujici h posunute o 6 bitu doprava

        cursor += 1;

    }
    h += h << 3;            // prictu k h, h posunute o 3 bity doleva
    h ^= h >> 11;           // prixoruji h posunute o 11 bitu doleva
    h += h << 15;           // prictu h posunute o 15 bitu doleva

    return h;
}
```

```
1  unsigned int crc32b(unsigned char *message) {
2     int i, j;
3     unsigned int byte, crc, mask;
4  
5     i = 0;
6     crc = 0xFFFFFFFF;
7     while (message[i] != 0) {
8        byte = message[i];            // Get next byte.
9        crc = crc ^ byte;
10       for (j = 7; j >= 0; j--) {    // Do eight times.
11          mask = -(crc & 1);
12          crc = (crc >> 1) ^ (0xEDB88320 & mask);
13       }
14       i = i + 1;
15    }
16    return ~crc;
17 }
```

Otevřené adresování

Klíče jsou v samotné tabulce, prázdné políčko obsahuje hodnotu `nil`. Faktor zaplnění tabulky tak nemůže být větší než 1.

Pro účely vyhledávání a vkládání používáme průzkumné posloupnosti. Předp. průzkumnou funkci

$$g : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\},$$

Pro klíč $k \in U$ je průzkumná posloupnost

$$g(k, 0), g(k, 1), \ldots, g(k, m-1)$$

Řekneme, že průzkumná funkce projde pro k celou tabulku, pokud je průzkumná posloupnost klíče k permutací posloupnosti $0, 1, \ldots, m-1$.

Princip otevřeného adresování

Při vyhledávání indexu odpovídajícího klíči k prohledáváme tabulku v pořadí odpovídajícím průzkumné posloupnosti pro k.

- Pokud je na aktuálním indexu klíč různý od k, pokračujeme v průzkumné posloupnosti dál.
- Pokud je na aktuálním indexu `nil`, klíč se v tabulce nenachází.
- Pokud projdeme celou průzkumnou posloupnost, klíč se tabulce nenachází.

Analogický postup použijeme pro přidávání uzlu: uzel přidáme na první index z průzkumné posloupnosti, na kterém je `nil`.

```
struct table
   data:[]Key
   g: (Key, Int) → Int — průzkumná funkce
```

```
search
 ⟵ T: Table
 ⟵ k: Key
 ⟶ Int — index, na kterém se nachází k nebo nil
```

```
1  for i in 0 ..< len(T.data)
2     j ← T.g(k,i)
3     if (T.data[j] == k) then return j
4     if (T.data[j] == nil then return nil
5  vrať nil
```

Přidávání klíče je analogické

Mazání

Odstranění prvku z tabulky by způsobilo přerušení průzkumné posloupnosti při vyhledávání. Prvky v tabulce proto musíme ponechat, ale dát jim příznak, že jsou smazány. Potom musíme příslušným způsobem upravit procedury pro přidávání (přidáváme i na místo, kde je smazaný prvek) a prohledávání (smazané prvky přeskakujeme).

Složitost těchto operací pak ale nezávisí přímo na faktoru zaplnění tabulky. Pokud potřebujeme i operaci mazání, je lepší použít řetězení.

Lineární prozkoumávání

Předpokládejme, že máme hašovací funkci h. K ní vytvoříme průzkumnou funkci

$$g(k, i) = (h(k) + i) \mod m$$

Průzkumná posloupnost tedy začíná $h(k)$ a pak kontrolujeme políčkou s o 1 větším indexem (mod m). Například pro $h(k) = 3$ a $m = 7$ je průzkumná posloupnost

$$3, 4, 5, 6, 0, 1, 2$$

Primární shlukování

- vzniknou dlouhé úseky obsazených políček, které tak zvyšují průměrný čas nutný k vyhledávání.
- *Důvod vznikání shluků*: prázdné políčko, které je za úsekem i obsazených políček, bude jako další zaplněno s pravděpodobností $(i + 1)/m$. Dlouhé obsazené úseky tak mají tendenci se prodlužovat.

Existuje m různých průzkumných sekvencí. Pro každý klíč projde průzkumná funkce celou tabulku.

Kvadratické prozkoumávání

Předpokládejme, že máme hašovací funkci h. K ní vytvoříme průzkumnou funkci

$$g(k, i) = (h(k) + c_1 i + c_2 i^2) \mod m,$$

c_1, c_2 jsou vhodně zvolené konstanty (jsou-li neceločíslné, pak před operací $\mod$ zaokrouhlíme dolů.)

Pro některou kombinaci hodnot c_1, c_2 a m nemusí g prozkoumat celou tabulku (máme $i \neq j$ takové, že $g(k, i) = g(k, j)$). Pokud je např. m prvočíslo, pak většina voleb c_1, c_2 (např. 1,1 nebo 0,1). vede k tomu, že délka průzkumné posloupnosti před tím, než narazíme na opakující se hodnotu, je přibližně $m/2$.

Sekundární shlukování

- pokud máme $k_1 \neq k_2$, ale $h(k_1) = h(k_2)$, mají k_1 a k_2 stejné průzkumné sekvence.

Existuje m různých průzkumných sekvencí.

Příklady průzkumných posloupností

Předpokládejme, že $h(k) = 0$.

- $m = 7, c_1 = 0, c_2 = 1$

$$0, 1, 4, 2, 2, 4, 1$$

- $m = 11, c_1 = 1, c_2 = 1$

$$0, 2, 6, 1, 9, 8, 9, 1, 6, 2, 0$$

- $m = 7, c_1 = 13, c_2 = 15$

$$0, 0, 5, 1, 2, 1, 5$$

- $m = 16, c_1 = 0.5, c_2 = 0.5$

$$0, 1, 3, 6, 10, 15, 5, 12, 4, 13, 7, 2, 14, 11, 9, 8$$

Dvojité haš1ování

Pro hašovací funkce h_1, h_2 zavedeme průzkumnou funkci

$$g(k,i) = (h_1(k) + ih_2(k)) \mod m$$

Hašovací funkce h_1 určuje počáteční pozici, hašovací funkce h_2 určuje offset, o který se posouváme.

Aby g prohledala celou tabulku, musí být $h_2(k)$ a m nesoudělné. Například zvolíme m jako prvočíslo a

$$h_1(k) = k \mod m$$
$$h_2(k) = 1 + (k \mod m - 1)$$

Pro prvočíselné m existuje $\theta(m^2)$ průzkumných posloupností, protože každá možná dvojice $h_1(k), h_2(k)$ vede k rozdílné průzkumné posloupnosti.

Příklad pro $m = 5$

k	$h_1(k)$	$h_2(k)$	
0	0	1	0,1,2,3,4,
1	1	2	1,3,0,2,4,
2	2	3	2,0,3,1,4,
3	3	4	3,2,1,0,4,
4	4	1	4,0,1,2,3,
5	0	2	0,2,4,1,3,
6	1	3	1,4,2,0,3,
7	2	4	2,1,0,4,3,
8	3	1	3,4,0,1,2,
9	4	2	4,1,3,0,2,

k	$h_1(k)$	$h_2(k)$	
10	0	3	0,3,1,4,2,
11	1	4	1,0,4,3,2,
12	2	1	2,3,4,0,1,
13	3	2	3,0,2,4,1,
14	4	3	4,2,0,3,1,
15	0	4	0,4,3,2,1,
16	1	1	1,2,3,4,0,
17	2	2	2,4,1,3,0,
18	3	3	3,1,4,2,0,
19	4	4	4,3,2,1,0,

Idealizovaná analýza složitosti

Předpoklad: každá průzkumná sekvence je stejně pravděpodobná (tj. každá permutace posloupnosti $0, 1 \ldots, m - 1$ je stejně pravděpodobná, když náhodně vybíráme klíče). To je velmi silný předpoklad, který žádná z výše popsaných metod nezaručuje.

Věta

V tabulce s otevřeným adresováním s faktorem naplnění α je průměrná délka části průzkumné posloupnosti, kterou projdeme při neúspěšném prohledávání, nejvýše $\frac{1}{1-\alpha}$.

Empiricky se tomuto ideálním případu blíží dvojité hašování.