Hashovací tabulka

Hashovací tabulka (*hash table*, *rozptýlená tabulka*, *hešovací tabulka*) je datová struktura, která slouží k ukládání dvojic klíč-hodnota. Hashovací tabulka kombinuje výhody vyhledávání pomocí indexu (složitost $O(1)$) a procházení seznamu (nízké nároky na paměť).

Možnosti ukládání dvojic klíč-hodnota

Pole

Uvažujme, že chceme ukládat dvojice klíč-hodnota a vyhledávat v nich. Jednou z možností by pak bylo jako klíč zvolit celé číslo (nebo nějaké mapování klíče na celé číslo) a hodnoty ukládat do pole. Tímto bychom si sice zajistili, že prvek nalezneme s konstantní paměťovou složitostí (jednoduše bychom jej vyžádali pomocí indexu), ale na druhou stranu by docházelo k obrovskému mrhání pamětí.

Ukládejme tímto způsobem názory respondentů (hodnota) na jednotlivé odstíny barev (klíč). Vytvoříme proto pole o velikosti $165813375 = 255 \cdot 255 \cdot 255\ (RGB)$. Pokud se dotážeme tisícovky respondentů, a každý se vyjádří k pěti různým barvám (odstínům), tak využijeme pouze 0.03% všech klíčů, což znamená, že jsme promrhali 99.97% paměti, kterou spotřebovalo pole.

Seznam

Opačným přístupem by bylo využití seznamu, ve kterém bychom vyhledávali sekvenčně. Zcela zřejmě by tímto odpadl problém s nevyužitými klíči a pamětí (nevyužité klíče by vůbec neexistovaly). Nevýhoda je ovšem zcela zřejmá – výkonnost. V okamžiku, kdy bychom se neptali jen tisícovky respondentů (lokální výzkum), ale uspořádali bychom globální anketu s miliony dotazovaných, tak by v těchto datech již nešlo rozumně vyhledávat (pro nalezení jednoho záznamu bychom spotřebovali $O(n)$ kroků).

Binární vyhledávací strom

Binární vyhledávací strom by mohl být střední cestou. V něm, stejně jako v seznamu, neexistují zbytečné klíče. Významnou výhodou je pak stromová organizace stavového prostoru, která umožní najít zvolenou hodnotu v čase $O(\log_2(n))$. Ale jak si ukážeme, jsme schopni dosáhnout ještě lepšího výkonu.

Rozptýlená tabulka

Hashovací (rozptýlená) tabulka je struktura, jež je postavena nad polem omezené velikosti n (tzn. pole nepopisuje celý stavový prostor klíče), a která pro adresaci využívá *hashovací funkci*. Nalezení prvku pro daný klíč zabere průměrně $O(1)$ operací.

Hashovací funkce

Hashovací funkce má následující vlastnosti:

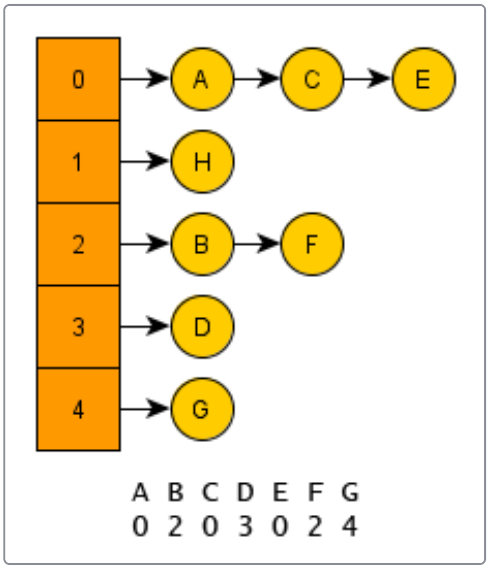
- Konzistentně vrací pro stejné objekty stejné adresy (slovo stejné typicky neznamená stejné instance, ale stejná data).
- Nezaručuje, že pro dva různé objekty vrátí různou adresu.
- Využívá celého prostoru adres se stejnou pravděpodobností.
- Výpočet adresy proběhne velmi rychle.

Velmi oblíbená je implementace hashovací funkce, která kombinuje násobení klíče multiplikativní konstantou a modulární aritmetiku (k je klíč, m je velikost pole).

$$H(k,\ m) = 97 \cdot k \mod m$$

Kolize

Jednou z uvedených vlastností hashovací funkce je, že nezaručuje, že dvěma různým objektům nepřiřadí stejné adresy. Situaci, kdy chceme uložit na stejnou adresu více objektů, se říká kolize.



Zřetězené hashování

Zřetězené rozptylování

Zřetězené hashování (*separate chaining*) je nejjednodušším způsobem, jakým se lze vypořádat s kolizemi – hodnoty totiž ukládáme do pole spojových seznamů. Nastane-li kolize, prvek se pouze přidá na konec adresovaného spojového seznamu. Nevýhoda tohoto přístupu je zřejmá – při velkém zaplnění tabulky dojde k postupné degradaci výkonu kvůli sekvenčnímu prohledávání příslušných seznamů.

Otevřené rozptylování

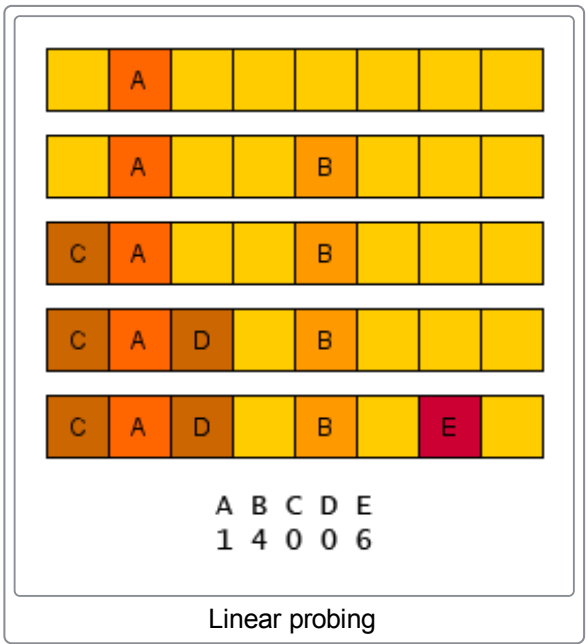
Při otevřeném rozptylování (*open addressing*, *closed hashing*) jsou hodnoty ukládány přímo do pole. Existují dvě základní strategie, pomocí níž se tabulka vypořádává s kolizemi – *linear probing* a *double hashing*.

Linear probing

V *linear probing* strategii nejprve vypočteme adresu, na kterou daný prvek uložíme. Je-li adresa obsazená, tak se posuneme o jedno místo dál a zkusíme prvek uložit znovu. Takto postupujeme tak dlouho, dokud se nám prvek nepodaří uložit.

Ukládací schéma lze charakterizovat funkcí ($i \in \{0,\ 1,\ \cdots,\ n-1\}$):

$$adresa = H(k,\ m) + i \mod m$$



Linear probing

Clustering

Velkou nevýhodou je linear probingu je *clustering* (shlukování). Vzhledem k principu ukládání objektů dochází ke vzniku shluků objektů, jež mají blízkou nebo totožnou adresu. Tyto shluky je pak při vyhledávání nutné sekvenčně procházet. Dopad na výkon je ještě vyšší než u zřetězeného rozptylování, protože shluky mohou obsahovat prvky odpovídající více klíčům.

Mazání prvků

Při mazání je zapotřebí nahradit mazaný prvek hlídkou (*sentinel*), což je speciální objekt, který značí, že je dané místo prázdné. Operace vyhledávání objekt hlídky přeskakuje a terminuje až v okamžiku, kdy hledaný prvek nalezne nebo narazí na skutečné prázné místo (*null*). Operace vkládání hlídku ignoruje (považuje adresu za práznou) a uloží na toto místo nový prvek.

Alternativně je při mazání možné odstranit a znovu uložit všechny prvky ve zbylé části shluku (tj. po nejbližší *null*). Naopak není možné prvky pouze setřepat, protože bychom mohli ztratit hodnoty pro další klíče, které se v daném shluku vyskytují (pokud bychom z tabulky na obrázku smazali prvek *C* a zbytek shluku setřepali, tak již nikdy nenalezneme prvek *A*).

Double hashing

Double hashing eliminuje shlukování díky využití dvojice rozptylovacích funkcí. První funkce vypočte iniciální adresu stejným způsobem jako u linear probingu nebo zřetězeného rozptylování. Pokud je dané místo již obsazené, nastoupí druhá funkce, která vypočte posun. Za předpokladu, že je i nové místo plné, dojde opět k posunu na základě druhé funkce.

$$adresa = H_1(k,\ m) + i \cdot H_2(k,\ m) \mod m$$

Mazání prvků

Při mazání prvků si tentokrát již nemůžeme pomoci novým uložením zbytku shluku, protože double hashing shluky netvoří. Musíme proto striktně využívat již zmíněného objektu hlídky, kterým nahradíme smazaný objekt.

Porovnání výkonu linear probingu a double hashingu

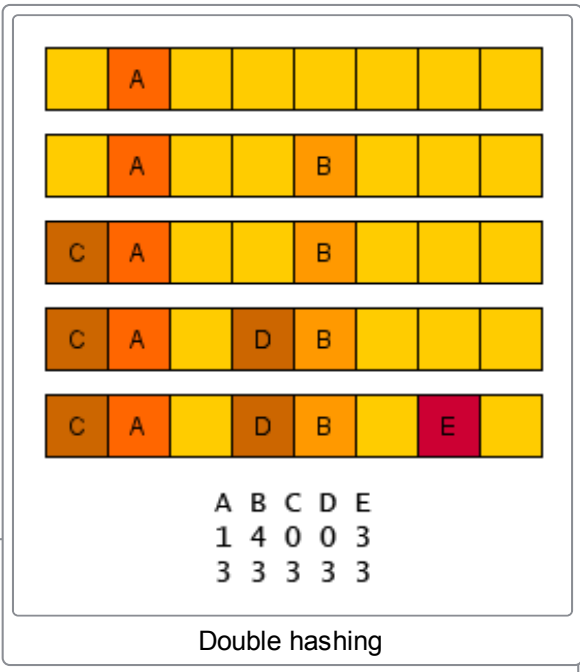
Pro porovnání výkonu linear probingu a double hashingu si zavedeme dvě metriky – *search hit* a *search miss*. Search hit značí, kolik operací musí průměrně algoritmus provést, aby našel prvek, který je v tabulce uložen. Search miss naopak vyjadřuje kolik operací musí algoritmus provést, aby zjistil, že tabulka prvek neobsahuje.

Search hit

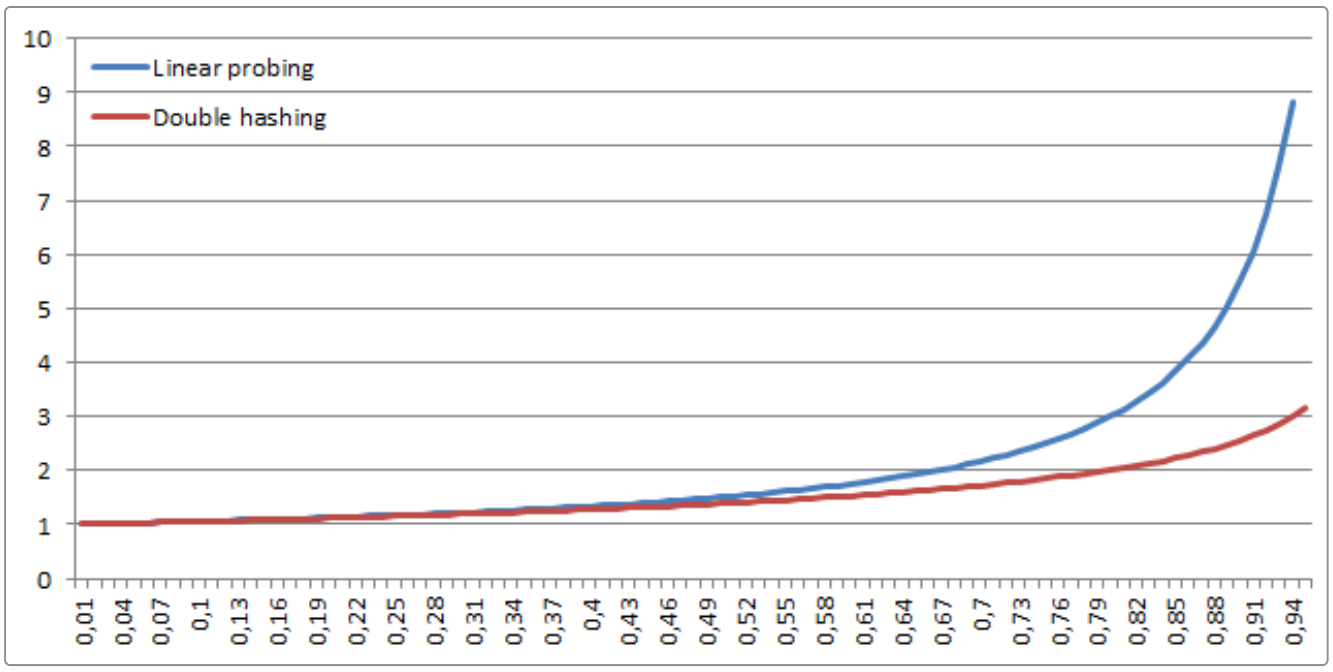
Pokud si zvolíme α jakožto poměr obsazení tabulky (0.9 znamená, že je tabulka obsazena z 90%), tak lze dle *Sedgewicka* spočítat průměrný počet operací pro *search miss* linear probingu (*lp*) a double hashingu (*dh*) jako:

$$search\_hit_{lp} = \frac{1}{2} \cdot (1 + \frac{1}{1-\alpha})$$

$$search\_hit_{dh} = \frac{1}{\alpha} \cdot \ln(\frac{1}{1-\alpha})$$



Double hashing



Počet operací nutných k nalezení obsaženého prvku při různém zaplnění tabulky (*search hit*)

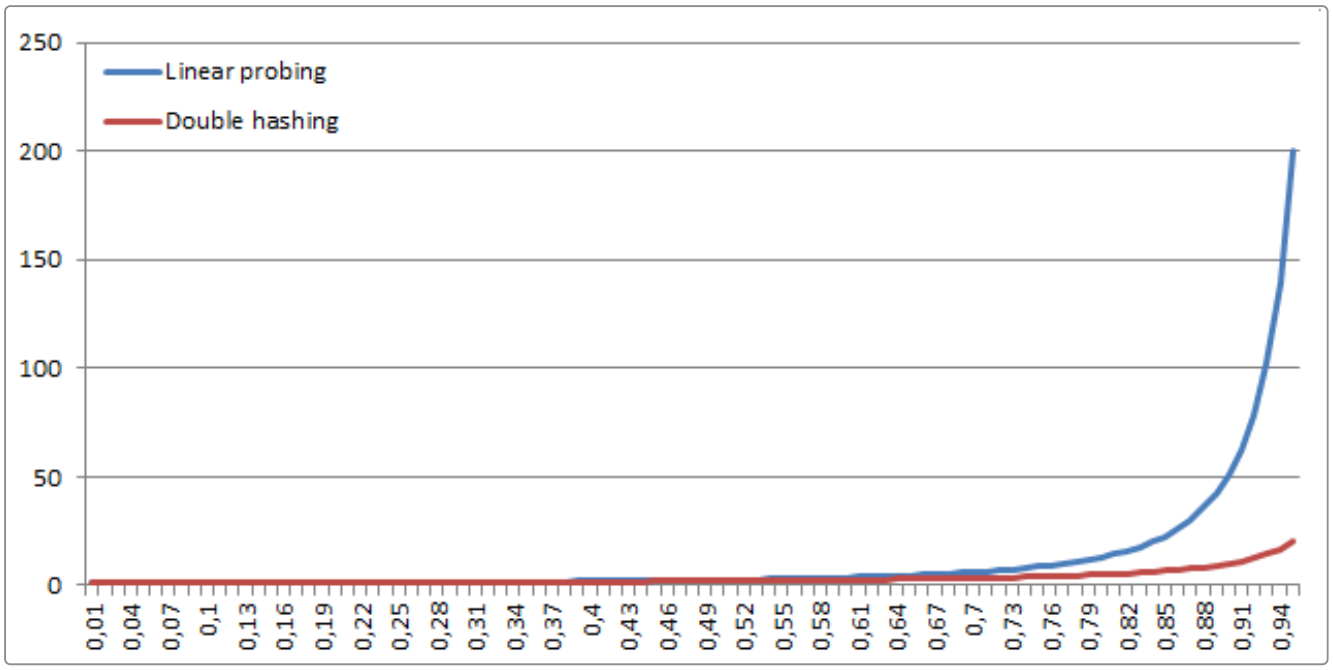
Osa x: α, osa y: počet operací

Search miss

Průměrný počet operací pro *search miss*:

$$search\_miss_{lp} = \frac{1}{2} \cdot (1 + \frac{1}{(1-\alpha)^2})$$

$$search\_miss_{dh} = \frac{1}{\alpha} \cdot \ln(\frac{1}{1-\alpha})$$



Počet operací nutných k zjištění, že tabulka hledaný prvek neobsahuje (*search miss*)
Osa x: α, osa y: počet operací

Vyhodnocení výkonnosti

Z uvedených vzorců a grafů jasně vyplývá, že u linear probingu dochází k velké degradaci výkonu, je-li tabulka obsazena z více než 75%, zatímco double hashing se chová poměrně přijatelně ještě při zaplnění 90%. Tyto poměry použijeme při konstrukci hashovací tabulky s dynamickou velikostí jako meze, při kterých zalokujeme novou větší tabulku, do níž přeneseme všechny prvky (prvky do nové tabulky znovu uložíme, prosté překopírování by vyústilo ve ztrátu dat).

Literatura

- SEDGEWICK, Robert. Algorithms in Java: Parts 1-4. Third Edition. [s.l.] : [s.n.], July 23, 2002. 768 s. ISBN 0-201-36120-5.

– Pavel Mička
Zakladatel a administrátor stránek Algoritmy.net