Kapitola 11

Státnice - Hašování

11.1 Hashování

Základní motivací pro hashování je slovníkový problém, kdy máme za úkol reprezentovat množinu S prvků z nějakého univerza U a provádět na ní následující operace:

- **MEMBER** (je třeba, aby tato operace probíhala velmi rychle)
- **INSERT**
- **DELETE**

Aby byl **MEMBER** rychlý, bylo by nejlepší mít v paměti pole bitů o velikosti U. V případě, že $|S| << |U|$ (a navíc U může být neúnosně velké), použiji hashovací funkci $h : U \to \{0, \ldots, m-1\}$ a množinu S reprezentuji polem s m políčky tak, že $x \in S$ je uložen na indexu $h(x)$. Předpokládejme, že funkce h se dá spočítat v čase $O(1)$ – jiné funkce vlastně nemají smysl, protože nepřináší dostatečné zrychlení.

Problém je, když nastane kolize: $x \neq y$, $h(x) = h(y)$. Jednotlivé druhy hashování, které následují, se liší strategiemi předcházení a řešení kolizí.

Pro následující analýzy si označíme:

- $|S| = n$
- $|U| = N$
- Load factor (faktor zaplnění) – $\alpha = \frac{n}{m}$.

Hashování se separovanými řetězci

V tomto typu hashování se kolize řeší řetězením ve spojácích: pro každé políčko založíme zvlášť spoják všech prvků, které se do něj hashují. Všechny algoritmy je musí projít. Předpokládejme, že řetězce jsou prosté – nic se v nich neopakuje. V nejhorším případě mají všechny prvky stejný hash a máme jen jeden seznam.

Paměťová náročnost je pro každý seznam $O(1 + l(i))$, kde $l(i)$ je délka toho seznamu.

Existují dvě varianty – neuspořádaná a s s uspořádanými prvky v řetězcích. Liší se jedině v očekávaném počtu testů pro neúspěšné hledání (když dojdu v řetězci za místo, kde by byl hledaný prvek, můžu skončit).

Pro odhad složitosti alogritmů předpokládáme, že:

- Hashovací funkce h rozděluje data rovnoměrně
- Sama reprezentovaná množina S je náhodný výběr z U s rovnoměrným rozdělením

Tyto předpoklady v praxi ale splněny být nemusí.

Očekávaná průměrná délka řetězců

Pro odhad složitosti se počítá **očekávaná délka řetězců**. Označme délku i-tého řetězce jako $\mathbf{l}(i)$. Potom pravděpodobnost, že tento řetězec má délku l, odpovídá binomickému rozdělení:

$$P(\mathbf{l}(i) = l) = p_{n,l} = \binom{n}{l} \left(\frac{1}{m}\right)^l \left(1 - \frac{1}{m}\right)^{n-l}$$

Toto je jen aproximace (pro nekonečnou velikost univerza i seznamů), pro případ, že $N >> n^2 m$, ale lze použít. Očekávaná délka řetězce pak vychází jako (rozepíšu faktoriál a vytknu $\frac{n}{m}$, pak změním rozsah sumace $1 \ldots n$ (protože násobení $l = 0$ mi nic nedá), pak můžu z $l - 1$ udělat l a sumovat $0 \ldots n - 1$):

$$E(l) = \sum_{l=0}^{n} l p_{n,l} = \frac{n}{m} \sum_{l=0}^{n-1} \binom{n-1}{l} (\frac{1}{m})^l (1-\frac{1}{m})^{n-1-l} = \frac{n}{m}(\frac{1}{m} + 1 - \frac{1}{m})^{n-1} = \frac{n}{m} = \alpha$$

Vlastně tu ale objevujeme Ameriku tím, že počítáme střední hodnotu binomicky rozdělené veličiny s parametrem $\frac{1}{m}$ – ze vzorce nám vyjde to samé. Stejně tak rozptyl ze vzorce vyjde $\frac{n}{m}\left(1-\frac{1}{m}\right)$.

Očekávaná délka nejdelšího řetězce

Tento údaj však sám o sobě nestačí, počítá se i očekávaný nejhorší případ (očekávaná **délka nejdelšího řetězce**). Ta se definuje následovně:

$$EMS = \sum_j j P(\max_i \mathbf{l}(i) = j) = \sum_j P(\max_i \mathbf{l}(i) \geq j)$$

Z toho (pravděpodobnost disjunkce jevů je $\leq$ součet jednotlivých pravděpodobností; vyčíslení: počet podmnožin správné velikosti a pravděpodobnost, že mají stejný hash):

$$P(\max_i \mathbf{l}(j) \geq j) \leq \sum_i P(\mathbf{l}(i) \geq j) \leq m\binom{n}{j}(\frac{1}{m})^j = \frac{\prod_{k=0}^{j-1}(n-k)}{j!}(\frac{1}{m})^{j-1} \leq n(\frac{n}{m})^{j-1}\frac{1}{j!}$$

Najdeme mezní hodnotu j_0, pro které $n(\frac{n}{m})^{j-1}\frac{1}{j!} \leq 1$. Označme $k_0 = \min\{k | n \leq k!\}$. Potom $j_0 \leq k_0$. Ze Stirlingovy formule plyne, že $\log x! = \Theta(x \log x)$. Z toho odvodíme (hodně neformálně, asymptoticky):

$$\log k_0! = k_0 \log k_0 = O(\log n)$$

$$\log k_0 + \log\log k_0 \approx \log k_0 = O(\log\log n)$$

$$k_0 = \frac{k_0 \log k_0}{\log k_0} = O(\frac{\log n}{\log\log n})$$

A $j_0 = O(k_0)$. Pro $\alpha \leq 1$ platí, že $EMS = O(j_0)$:

$$EMS = \sum_j P(\max_i \mathbf{l}(i) \geq j) \leq \sum_j \min\{1, n(\frac{n}{m})^{j-1}\frac{1}{j!}\} = \sum_{j=1}^{j_0} 1 + \sum_{j=j_0+1}^{\infty}(n(\frac{n}{m})^{j-1}\frac{1}{j!}) \leq j_0 + \sum_{j=j_0+1}^{\infty}\frac{n}{j!} = \cdots \leq j_0 + \frac{1}{j_0}$$

A tedy očekávaná délka nejdelšího řetězce je $O(\frac{\log n}{\log\log n})$.

Očekávaný počet testů

Testy jsou porovnání toho, co hledáme, s nějakým prvkem, nebo zjištění, že řetězec je prázdný. Jejich očekávaný počet je další odhad efektivity struktury. Rozlišujeme úspěšné a neúspěšné hledání.

Neúspěšné hledání (Je-li délka řetězce 0, jeden test stejně provedu, jinak otestuji celý řetězec):

$$E(T) = p_{n,0} + \sum_l l p_{n,l} = (1-\frac{1}{m})^n + \frac{n}{m} \approx e^{-\alpha} + \alpha$$

S uspořádanými řetězci končím dřív ($e^{-\alpha} + 1 + \frac{\alpha}{2} - \frac{1}{\alpha}(1-e^{-\alpha})$).

Počet testů pro úspěšné vyhledávání je roven průměru počtu testů provedených při vložení každého z prvků, tj. 1 + očekávaná délka řetězce při každém vkládání: $\frac{1}{n}\sum_{i=0}^{n-1}\left(1+\frac{i}{m}\right) = 1 + \frac{n-1}{2m} \approx 1 + \frac{\alpha}{2}$.

Hashování s přemísťováním

Nevýhodou separovaných řetězců je nutnost alokovat další paměť, to je neefektivní. Proto zavedeme do hashovací tabulky pomocné ukazatele a celé řetězce nacpeme přímo do ní (a zřetězené prvky prostě rozházíme na jiné adresy). Pro hashování s přemísťováním se v tabulce uchovává navíc jednoduše odkaz na předchozí a následující prvek řetězce. Pokud vkládáme na místo, kde už je nějaký prvek z jiného řetězce, přehodíme tento cizí prvek jinam.

Algoritmy jsou téměř stejné jako pro separované řetězce, jen při **DELETE** prvního prvku řetězce je nutné na jeho místo přesunout druhý (pokud existuje).

Očekávaný počet testů je stejný jako pro hashování se separovanými řetězci. Přemísťování v tabulce je ale náročnější než 1 test, proto jsou **INSERT** a **DELETE** pomalejší.

Hashování se dvěma ukazateli

Od předchozího se liší tím, že místo ukazatele na předchozí prvek používá odkaz na začátek řetězce BEGIN. Řetězec tak už nemusí začínat na indexu svého hashe.

Místo přesouvání prvků algoritmy mění BEGIN (ten je na j-tém políčku vyplněn, právě když existuje řetězec prvků s hashem j).

- **INSERT** všechno vkládá na konec řetězce, zakládá-li nový, do BEGIN (na místě určené hashem) píše, kde se ve skutečnosti nachází
- **DELETE** jen upravuje odkazy na následující, nebo BEGIN (pokud maže poslední prvek řetězce).

Kvůli tomu, že řetězce začínají jinde než na svém místě, je počet testů o něco větší:

- Úspěšné hedání: $1 + \frac{(n-1)(n-2)}{6m^2} + \frac{n-1}{2m} \approx 1 + \frac{\alpha^2}{6} + \frac{\alpha}{2}$
- Neúspěšné hledání přibližně $1 + \frac{\alpha^2}{2} + \alpha + e^{-\alpha}(2+\alpha) - 2$.

Srůstající (coalesced) hashování

Srůstající hashování používá jen jeden ukazatel v hashovací tabulce navíc – odkaz na další prvek NEXT. Řetězce tak obsahují hodnoty s různými hashi. Prvek s vkládáme vždy do řetězce, obsahujícího $h(s)$-té políčko v tabulce.

Existují různé varianty:

- Standardní (bez pomocné paměti, "late" a "early" insertion) – LISCH, EISCH
- Bezpřívlastkové (s pomocnou pamětí, "late", "early" a "varied" insertion) — LICH, VICH, EICH.

Bez pomocné paměti – LISCH a EISCH

LISCH je "late insertion", tedy přidává se za poslední prvek řetězce. EISCH ("early insertion") přidává za první prvek řetězce.

- Algoritmus **MEMBER** je stejný pro oba (jen projití řetězce po odkazech NEXT).
- Alg. **INSERT**:
  - U LISCH projití celého řetězce (v případě že není prázdný, jinak jednoduše vložím na správné políčko) s testy na přítomnost prvku, potom vložení na libovolné volné místo v tabulce a připojení na konec řetězce.
  - Pro EISCH vložení na nějaké volné místo v tabulce a jen přepojení ukazatelů NEXT – připojení do řetězce za první prvek (pokud je řetězec neprázdný).

Algoritmy **DELETE** nejsou známy, kromě primitivních. Problémem je u nich zachování náhodného uspořádání prvků v řetězcích, které se předpokládá pro dodržení očekávaných časů operací. Je ale možné také prvky jen označit jako odstraněné a jejich místa použít při vkládání dalších (to ale zpomaluje hledání).

EISCH je kupodivu o něco rychlejší na úspěšné vyhledání (je větší pravděpodobnost práce s novým prvkem), očekávaný počet testů je stejný.

Počet testů v neúspěšném případě: Spočteme průměr přes všechny posloupnosti délky $n+1$ (kde hledáme $n+1$. prvek v množině ostatních n). Označme $c_{n,l}$ počet řetězců délky l, které přispívají celkem $1 + 2 + \cdots + l = l + \binom{l}{2}$ porovnáními k sumě:

$$\frac{c_{n,0} + \sum_{l=1}^{n} l c_{n,l} + \sum_{l=1}^{n} \binom{l}{2} c_{n,l}}{m^{n+1}}$$

Tady $c_{n,0}$ představuje počet prázdných řádků, tedy $c_{n,0} = (m-n)m^n$, $lc_{n,l}$ je součet délek všech řetězců v reprezentacích n-prvk. množin, tedy $\sum_{l=1}^{n} l c_{n,l} = nm^n$.

Poslední člen označíme S_n. Při **INSERT**u do $n-1$-prvkové množiny jsou 2 možnosti vzniku řetězce délky l: buď přidávám do řetězce (délky $l-1$), nebo řetězec (pův. délky l) nezměním. Z toho vyjádříme rekurentní vztah pro S_n (úpravy: rozepsání rozdílů, vykrácení, rozpis l^2 jako $l^2 - l + l = \binom{l}{2} + l$):

$$S_n = \sum_l \binom{l}{2}(m-l)c_{n-1,l} + \sum_l \binom{l}{2}(l-1)c_{n-1,l-1} = mS_{n-1} - \sum_l l^2 c_{n-1,l} = (m+2)S_{n-1} + (n-1)m^{n-1})$$

Pak pomocí vztahu $T_n^c = \sum_{i=1}^{n} ic^i = \frac{nc^{n+2} - (n+1)c^{n+1} + c}{(c-1)^2}$ spočítaného z $(c-1)T_n^c = nc^{n+1} + (\sum_{i=2}^{n} - c^i) - c$ získáme nerekurentní vztah ($S_0 = 0$, obrácení sumace a vytknutí $m + 2^{n-1}$):

$$S_n = (m+2)^{n-1}S_0 + \sum_{i=0}^{n-1}(m+2)^i(n-1-i)m^{n-1-i} = (m+2)^{n-1}\sum_{i=1}^{n-1} i\left(\frac{m}{m+2}\right)^i = \frac{1}{4}(m(m+2)^n - m^{n+1} - 2nm^n$$

A tedy očekávaný počet testů vyjde:

$$1 + \frac{1}{4}\left(\left(1+\frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) \approx 1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$$

Počet testů pro úspěšný případ spočteme pro LISCH jako počet testů při vkládání prvku. Metoda EISCH pro tento postup nesplňuje předpoklady. Porovnání klíčů při neúspěšném vyhledávání je stejně při přístupu na obsazené políčko, neporovnávám ale nic při přístupu na neobsazené políčko, takže dostávám:

$$\frac{n}{m} + \frac{1}{4}\left(\left(1+\frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) = \frac{1}{4}\left(\left(1+\frac{2}{m}\right)^n - 1 + \frac{2n}{m}\right)$$

Průměr pro postupné vkládání všech prvků pak dává:

$$1 + \sum_{i=0}^{n-1}\frac{1}{4}\left(\left(1+\frac{2}{m}\right)^i - 1 + \frac{2i}{m}\right) = 1 + \frac{m}{8n}\left(\left(1+\frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) + \frac{n-1}{4m} \approx 1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}$$

Pro metodu EISCH vychází (bez důkazu):

$$\frac{m}{n}\left(\left(1+\frac{1}{m}\right)n - 1\right) \approx \frac{1}{\alpha}(e^{\alpha} - 1)$$

Všechny odhady mají odchylku $O(\frac{1}{m})$.

S pomocnou pamětí – LICH, VICH, EICH

V této variantě rozdělíme paměť na dvě části:

- (hash-funkcí) přímo adresovatelná
- pomocná část (bez přístupu hash-funkcí)

Při kolizích nejdříve ukládáme do řádků z pomocné části, pak teprve do přímo adresovatelné, tedy oddalujeme srůstání řetězců. Chování se tak až do určitého okamžiku podobá separovaným.

Existují tři varianty podle chování algoritmu **INSERT**:

- LICH vždy přidává na konec řetězce
- EICH v případě neprázdného řetězce vždy za 1. prvek
- VICH vždy za poslední prvek v pomocné paměti nebo (pokud žádné v pomocné paměti nejsou) za 1. prvek řetězce (tj. chová se na pomocné paměti jako LICH a v přímo adresovatelné části jako EICH).

Algoritmy až na VICH se chovají stejně jako ve standardním srůstajícím hashování, rozhodující je výběr volného řádku pro vložení: např. "vždy vyber z nejvyšší adresy" může zaručit používání pomocné paměti. Také tu není přirozené efektivní **DELETE**.

Odhad složitosti: definujeme si následující hodnoty:

- n – počet uložených prvků
- m – velikost přímo adresovatelné paměti
- m' – celková velikost paměti
- $\alpha = \frac{n}{m'}$ – faktor zaplnění
- $\beta = \frac{m}{m'}$ – adresovací faktor
- λ – jediné nezáp. řešení rovnice $e^{-\lambda} + \lambda = \frac{1}{\beta}$

Pokud je $\alpha \leq \lambda\beta$, pak pro všechny verze vychází očekávaný počet testů $e^{-\frac{\alpha}{\beta}} + \frac{\alpha}{\beta}$ v neúspěšném případě a $1 + \frac{\alpha}{2\beta}$ v úspěšném (chyba: $O(\log \frac{m'}{\sqrt{m'}})$.

V případě, že $\alpha \geq \lambda\beta$ (začínají srůstat řetězce), se metody liší a vychází divnosti. V neúspěšném případě je VICH a LICH lepší než EICH, v úspěšném vede VICH před EICH a LICH (vždy o jednotky procent). Doporučená hodnota $\beta = 0.86$. Na hledání volného řádku se v praxi hodí např. spojový seznam volných řádků.

Lineární přidávání

Tato a následující metoda nepoužívá žádné dodatečné položky v hashovací tabulce a zároveň řetězce kolidujících prvků ukládá přímo do ní. Nalezení dalšího prvku z řetězce je přímo v algoritmu.

Lineární přidávání je nejjednodušším řešením takové situace: v případě kolize při **INSERT**u nalezne nejbližší vyšší volné políčko a vloží nový prvek tam. Předpokládáme "cyklickou" paměť, tj. když dojdeme na konec, vkládáme od začátku.

Problémem je tvoření shluků – při velkém zaplnění se operace dost zpomalují. Také nepodporuje efektivní DELETE (a ani primitivní způsoby nejsou moc rychlé). V praxi je dobré uchovávat počet uložených prvků nebo mít zarážku (nikdy neobsazované pole), abychom věděli, kdy dojde k přeplnění.

Očekávaný počet testů je $\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)$ v neúspěšném a $\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)\right)$ v úspěšném případě (bez důkazu).

Dvojité hashování

Dvojité hashování je vylepšení předchozí metody tak, aby nevznikaly shluky. Výběr následujícího řádku bude závislý na předchozím, ale s rovnoměrným rozložením. Na to použiji druhou hashovací funkci h_2.

Při operacích **INSERT** pak hledám nejmenší i od 0, že $(h_1(x)+i\cdot h_2(x)) \mod m$ je volné políčko, tj. postupně přičítám $h_2(x)$ a modulím. Stejný postup je i pro operaci **MEMBER**.

Je nutné, aby $h_2(x) \not| m$, tj. abych měl prosté posloupnosti (a z každého políčka tak mohl vést řetězec po celé tabulce). Idea, že iterace h_2 tvoří pro každé x náhodnou permutaci paměťových míst, není úplně přesná, ale v praxi stačí, aby z $h_1(x)=h_1(y)$ plynulo, že $h_2(x)$ a $h_2(y)$ budou odlišné.

Funkce navíc musíme volit "chytře" (i lineární přidávání je spec. příp. dvojitého hashování, kdy $h_2\equiv 1$). Pak tato metoda je znatelně rychlejší než lin. přidávání. Předpoklad náhodnosti použitý v teoretické analýze sice splnit nelze, ale přiblížit se mu ano.

Očekávaný počet testů

Předpokládáme, že iterování funkce h_2 tvoří náhodné permutace (což, jak bylo řečeno, není úplně přesné).

Pro neúspěšný případ: označme $q_i(n,m)$ pravděpodobnost, že při zaplnění $\frac{n}{m}$ je pro nějaké x prvních $i-1$ políček, kam bych ho mohl vložit, plných. Potom $q_i(n,m)=\frac{\prod_{j=0}^{i-1}(n-j)}{\prod_{j=0}^{i-1}(m-j)}$ a tedy $q_i(n,m)=\frac{n}{m}q_{i-1}(n-1,m-1)$.

Očekávaný počet testů je (předposlední rovnost plyne z rekurentního vztahu pro q_j, poslední krok dokázat indukcí):

$$C(n,m)=\sum_{j=0}^{n}(j+1)(q_j(n,m)-q_{j+1}(n,m))=\sum_{j=0}^{n}(q_j(n,m))=1+\frac{n}{m}C(n-1,m-1)=\frac{m+1}{m-n+1}$$

Počet testů v úspěšném případě – stejná metoda jako u dřívějších analýz, takže vychází:

$$\frac{1}{n}\sum_{i=0}^{n-1}C(i,m)=\frac{1}{n}\sum_{i=0}^{n}n-1\frac{m+1}{m-i+1}\approx\frac{1}{\alpha}\ln\left(\frac{m+1}{m-n+1}\right)\approx\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

Srovnání

Podle počtu testů:

	neúspěšné	**úspěšné**
1.	separované uspořádané řetězce	separované (usp. i neusp.) řetězce, přemísťování
2.	separované řetězce, přemísťování	dva ukazatele
3.	dva ukazatele	VICH
4.	VICH, LICH	LICH
5.	EICH	EICH
6.	LISCH, EISCH	EISCH
7.	dvojité hashování	LISCH
8.	lineární přidávání	dvojité hashování
9.		lineární přidávání

- VICH je při vhodném α lepší než hashování se dvěma ukazateli.
- Lineární přidávání se nedá použít pro $\alpha>0.7$, dvojité hashování pro $\alpha>0.9$.
- Separované řetězce a obecné srůstající hashování používají víc paměti, přemísťování a dvojité hashování zas víc času, tj. nelze říct, které je jednoznačně lepší.

Implementační dodatky

- Pro hledání volných řádků se většinou používá seznam (zásobník).
- Přeplnění se většinou řeší držením α v rozumném intervalu ($\langle 1/4, 1\rangle$) a přehashováním do jinak velké tabulky ($2^i \cdot m$) při pře- nebo podtečení
- V praxi se doporučuje přehashování odkládat (např. pomocnými tabulkami) a provádět při nečinnosti systému.

DELETE se ve strukturách, které ho nepodporují, řeší označením políčka jako smazaného s možností využití při vkládání. V případě, že polovina polí je blokovaná tímto způsobem, se vše přehashuje. Pro srůstající hashování se toto používat nemusí, máme metody na zachování náhodnosti rozdělení dat.

V praxi je výhodné, známe-li něco o rozdělení vstupních dat, aby ho hashovací funkce kopírovala (většinou to ale nejde), jinak musíme předpokládat rovnoměrnost, což zaručeno zdaleka není. Nutnost rovnoměrného rozdělení vstupních dat lze obejít (viz níže).

11.2 Univerzální hashování

Abychom nemuseli předpokládat rovnoměrné rozložení vstupních hashovaných dat (což zdaleka není vždy zaručeno), budeme mít místo pevné hash-funkce nějakou množinu H, z níž funkci náhodně rovnoměrně vyberu. Analýza složitostí se pak dělá přes všechny $h \in H$ a platí pro všechny $S \subset U$ – i nerovnoměrné (S je daná pevně a h se k ní volí; $|U| = N$).

Pro formalizaci analýz je nutné mít analytické zadání funkcí h a znát přesnou velikost množiny $|H|$. To obejdeme očíslováním funkcí $H = \{h_i; i \in I\}$ a počítáním s indexovou množinou (očekávaná hodnota je průměr přes I). Při použití skutečné velikosti H v odhadech bychom dostali horší výsledky, protože některé funkce s různými indexy se můžou ve skutečnosti ukázat jako identické, a to tu zanedbáváme.

Definice: Systém funkcí $H = \{h_i; i \in I\} : U \to \{0, \ldots, m-1\}$ je c-univerzální, pokud:

$$\forall x, y \in U, x \neq y : |\{i \in I; h_i(x) = h_i(y)\}| \leq \tfrac{c|I|}{m}.$$

Tj. zaručuje se, že pro každé dva různé prvky má maximálně c funkcí kolizi.

Existence c-univerzálních systémů

Předpokládejme, že universum má tvar $U = \{0, 1, \ldots, N-1\}$ kde N je nějaké prvočíslo a vezmeme funkce typu

$$h_{a,b}(x) = ((ax + b) \mod N) \mod m$$

Jsou dobře použitelné, protože se dají počítat rychle. Protože N je prvočíslo, můžeme pracovat v $\mathbb{Z}_N$, coz je těleso. Rovnice $h_{a,b}(x) = h_{a,b}(y)$ je ekvivalentní s:

$$\exists i \in \{0, \ldots, m-1\} \wedge \exists r, s \in \{0, \ldots, \lceil \frac{N}{m} \rceil - 1\} : (ax + b \equiv i + rm) \mod N \wedge (ay + b \equiv i + sm) \mod N$$

Z Frobeniovy věty o jednoznačnosti řešení lineárních rovnic plyne, že pro každé r, s, i existuje jen jedna dvojice a, b, které vyhovuje. Počet řešení soustavy je tedy omezený číslem $m \cdot \lceil \frac{N}{m} \rceil^2$ (i – m hodnot, r, s – $\lceil \frac{N}{m} \rceil$ hodnot pro daná x, y).

Pak je systém c-univerzální pro $c = \left(\lceil \frac{N}{m} \rceil\right)^2 / \left(\frac{N}{m}\right)^2$ a jeho velikost opdovídá N^2.

Vlastnosti

Vyrobíme si pomocnou funkci

$$\delta_i(x, y) = \begin{cases} 1 \text{ pro } h_i(x) = h_i(y), x \neq y \\ 0 \text{ jinak} \end{cases}$$

Chceme potom spočítat součet $\delta_i(x, S) = \sum_{y \in S} \delta_i(x, y)$. Z výsledku vidíme očekávanou délku řetězce pro libovolnou (jednu) množinu dat. Tohle pak sečtu přes všechny mé hash-funkce a z c-univerzality dostanu

$$\sum_{i \in I} \delta_i(x, S) = \sum_{y \in S} \sum_{i \in I} \delta_i(x, y) \leq \sum_{y \in S, x \neq y} c\frac{|I|}{m} = \begin{cases} (|S| - 1)c\frac{|I|}{m} & \text{pro } x \in S \\ |S|c\frac{|I|}{m} & \text{jinak} \end{cases}$$

Z toho dopočítám (podělením $|I|$) horní odhad očekávaného $\delta_i(x, S)$.

Výsledek: očekávaný čas operací **MEMBER**, **INSERT** a **DELETE** v c-univerzálním hashování je $O(1 + c\alpha)$ (kde faktor naplnění $\alpha = \frac{|S|}{m}$). Čas n po sobě jdoucích operací na původně prázdné tabulce je $O(n(1 + \frac{c}{2}\alpha))$. To není lepší hodnota než mají separované řetězce ($O(1 + \alpha)$), ale u nich předpokládám rovnoměrné rozdělení dat.

Výběr vhodné funkce není úplně jednoduchý, protože funkcí může celkem být např až N^2, tj. nelze ho provést jednoduchým zavoláním generátoru náhodných čísel, nýbrž např. náhodným vybráním každého bitu indexu funkce. Proto je výhodné najít co nejmenší c-univerzální systémy (viz dále).

Dolní odhady velikosti univ. systémů

Očíslujme hash-funkce z I a induktivně definujme množiny U_i jako největší podmnožiny U_{i-1} takové, že $h_{i-1}(U_i)$ je jednoprvková. Platí $|U_i| \geq \lceil \frac{U_{i-1}}{m} \rceil$, tedy $|U_i| \geq \lceil \frac{N}{m^i} \rceil$ – velikost těchto množin klesá s logaritmem a $|I| \geq \frac{m}{c}(\lceil \log_m N \rceil - 1)$. Takže velikost univ. systému roste alespoň úměrně logaritmu velikosti univerza.

Dolní odhad c

5-univerzální systém: Zvolme $t \in \mathbb{N}$ a k němu vezměme t-té prvočíslo p_t tak, že $t \ln p_t \geq m \ln N$. Definujme systém funkcí $H = \{g_{c,d,l}(x) | t < l \leq 2t, c, d \in \{0, 1, \ldots, p_{2t}-1\}\}$ kde $((c(x \mod p_l)+d) \mod p_{2t}) \mod m$. Zřejmě $|H| = tp_{2t}^2$.

Odhadem $|G = \{(c,d,l); h_{c,d,l}(x) = h_{c,d,l}(y)\}|$, když si množinu rozdělíme na $G_1 = \{(c,d,l) \in G; x \mod p_l \neq y \mod p_l\}$ a $G_2 = \{(c,d,l) \in G; x \mod p_l = y \mod p_l\}$, se dá dokázat, že systém je 5-univerzální, za dalších podmínek i 3.25-univerzální.

Dolní odhad c: Platí: $c > 1 - \frac{m}{N}$. Spočítáme $\sum_{h \in H} \sum_{x,y \in U} \delta_h(x,y)$ – pro pevnou h máme (z Cauchy-Schwarzovy nerovnosti, $u_{i,h} = |\{x \in U, h(x) = i\}|$):

$$\sum_{x,y \in U} \delta_h(x,y) = \sum_{i=0}^{m-1} u_{i,h}(u_{i,h} - 1) \geq \frac{(\sum_{i=0}^{m-1} u_{i,h})^2}{m} - N = \frac{N^2}{m} - N$$

Tedy $\sum_{h \in H} \sum_{x,y \in U} \delta(x,y) \geq \frac{|H|N(N-m)}{m}$. Zároveň platí $\sum_{h \in H} \sum_{x,y \in U} \delta(x,y) \leq \sum_{x,y \in U} c\frac{|H|}{m} = N^2 c\frac{|H|}{m}$, což mi dává výsledek.

11.3 Perfektní hashování

Základní ideou perfektního hashování je nalézt hash-funkci, která pro danou množinu S nedělá žádné kolize, takže operace **MEMBER** bude velice rychlá. Potom je nevýhoda, že nelze přirozeným způsobem realizovat operaci **INSERT**, tj. v praxi se nesmí moc často vyskytovat.

Tabulka by neměla být o mnoho větší než množina S a funkce h rychle spočitatelná a její realizace nezabírat moc paměti (tedy žádná zadávání tabulkou).

Definice

- Hashovací funkce h je perfektní pro množinu S, pokud pro $\forall x, y \in S, x \neq y : h(x) \neq h(y)$
- Soubor funkcí $H : U \to \{0, \ldots, m-1\}$ je (N, m, n)-perfektní, pokud $\forall S \subseteq U$ takové, že $|S| = n$ existuje $h \in H$ perfektní pro S.

Odhady velikosti

Každá funkce h je perfektní pro $\sum\{\prod_{j=0}^{n-1} |h^{-1}(i_j)|; 0 \leq i_0 < \cdots < i_{n-1} < m\}$ množin (sčítáme přes všechny množiny hashů $h(S)$ a pro každou z nich uvažujeme všechny možnosti, jak mohla vzniknout). Z Cauchy-Schwarzovy nerovnosti plyne, že tento výraz nabývá maxima, když $\forall i : |h^{-1}(i)| = \frac{N}{m}$. Každá funkce je tedy perfektní pro max. $\binom{m}{n}(\frac{N}{m})^n$ množin. Z toho plyne:

$$|H| \geq \frac{\binom{N}{n}}{\binom{m}{n}\left(\frac{N}{m}\right)^n}$$

Jiný odhad lze provést jako u c-univ. systémů s očíslovanými funkcemi $|H| = \{h_1, \ldots, h_t\}$. Používám induktivně definované množiny U_i, kde $U_0 = U$ a U_i je největší podmnožina U_{i-1}, kde je zrovna funkce h_i konstantní. Dostáváme $|U_i| \geq \frac{|U_{i-1}|}{m}$, tj. $|U_t| \geq \frac{N}{m^t}$, ale z perfektnosti plyne $|U_t| \leq 1$. Dostáváme $t \geq \frac{logN}{logm}$.

Existence

Reprezentujme soubor funkcí $H = \{h_1, \ldots, h_t\}$ na univerzu velikosti N pomocí matice $M(H)$ typu $N \times t$, takže $M(H)_{x,i} = h_i(x)$, tj. v jednom sloupci jsou výsledky jedné hashovací funkce pro všechny prvky univerza.

Pak žádná funkce z H není perfektní pro množinu $S \subset U$, když podmatice $M(H)$ tvořená řádky příslušejícími prvkům S nemá prostý sloupec. Takových matic je maximálně (počet všech funkcí minus počet prostých, to celé krát libovolné doplnění na N řádek):

$$\left(m^n - \prod_{i=0}^{n-1}(m-i)\right)^t \cdot m^{(N-n)t}$$

Podmnožin U velikosti n je pak $\binom{N}{n}$, čímž vynásobeno mám počet matic neodpovídajících (N,m,n)-perfektnímu systému. Všech matic je m^{Nt}. Potom existuje (N,m,n)-perfektní systém, když:

$$\binom{N}{n}\left(m^n - \prod_{i=0}^{n-1}(m-i)\right)^t \cdot m^{(N-n)t} < m^{Nt}$$

Přišernými kejklemi dostaneme podmínku existence $t \geq n(\ln N)e^{\frac{n^2}{m}}$.

Konstrukce funkce

Chceme splnit rychlou spočitatelnost a paměťovou nenáročnost. Předpokládáme univerzum prvočíselné velikosti a funkce typu:

$$h_k(x) = (kx \mod N) \mod m$$

Označme $b_i^k = |\{x \in S; (kx \mod N) \mod m = i\}|$. Potom pokud $h_k(x)$ není perfektní, pak nějaké $b_i^k = 2$ a mám $\sum_{i=0}^{m-1}(b_i^k)^2 \geq n+2$.

Odhadnu výraz $\sum_{k=1}^{N-1}((\sum_{i=0}^{m-1}(b_i^k)^2) - n) = \sum_{x\neq y\in S}\{k; 1 \leq k < N, h_k(x) = h_k(y)\}$. Z vlastností modula mám takových k pro daná x,y nejvýše $2\lfloor\frac{N}{m}\rfloor = 2\lfloor\frac{N-1}{m}\rfloor$. Dostávám tedy odhad $2(N-1)\frac{n(n-1)}{m}$ a z něj vidím, že existuje takové k, že $\sum_{i=0}^{m-1}(b_i^k)^2 \leq \frac{2n(n-1)}{m} + n$, tedy pro tabulku velikosti $m > n(n-1)$ mám perfektní funkci.

Dá se dokázat trochu slabší předpoklad, že $P(k; \sum_{i=0}^{m-1}(b_i^k)^2 < \frac{3n(n-1)}{m} + n) \geq 1/4$, který je základem pravděpodobnostního algoritmu.

Pak mám deterministický algoritmus, který pro $m = n(n-1)+1$ nalezne perfektní h_k v čase $O(nN)$ a pravděpodobnostní, který pro $m = 2n(n-1)$ najde perfektní h_k v čase $O(n)$. Mám tedy konstrukci perfektní hash-funkce, ta ale nesplňuje požadavek na malou tabulku ($m = \Theta(n^2)$).

Menší tabulka

Zmenšíme-li velikost tabulky na $m = n$, bude výše uvedený algoritmus schopný nalézt funkci, pro kterou platí $\sum(b_i^k)^2 < 3n$ ($\sum(b_i^k)^2 < 4n$ v pravděpodobnostní variantě). Každou kolizi pak můžeme "rozstrkat" perfektní funkcí nad miniaturní tabulkou a celková velikost všech tabulek bude mnohem menší:

- Vezmeme nalezenou funkci a najdeme všechny neprázdné množiny $S_i = \{s \in S; h_k(s) = i\}$
- Pro jim odpovídající $c_i = |S_i|(|S_i|-1)+1$ (dvojnásobek v pravděp. metodě) najdeme k_i takové, že h_{k_i} je perfektní funkce pro S_i do tabulky velikosti c_i.
- Definujme $d_i = \sum_{j=0}^{i-1} c_j$, potom pokud $h_k(x) = l$, pak $g(x) = d_l + h_{kl}(x)$ je perfektní, její hodnota spočitatelná v čase $O(1)$ a hashuje do tabulky velikosti $O(3n)$ ($O(6n)$ s pravděp. případě), je naleznutelná v čase $O(nN)$ ($O(n)$) a pro její uložení do paměti jsou potřeba hodnoty k a k_i, vyžadující $O(n \log N)$ paměti.

Pro výpočet $g(x)$ potřebuji 2 násobení, 2 modulo a 1 sčítání (pro d_i v paměti), tabulka má velikost $\sum c_i \leq \sum(b_i^k)^2 < 3n$. Taková funkce ale stále nesplňuje požadavek na málo paměti pro uložení.

"Malá" funkce

Víme, že pro $m \in \mathbb{N}$ je počet prvočísel, která ho dělí $O(\frac{\log m}{\log\log m})$. Z toho úvahou o dělitelích čísla $D = \prod_{1\leq i<j\leq n}(s_j - s_i) \leq N^{n^2}$ na n-prvkové S a vzorce hustoty prvočísel $p_t \leq 2t\ln t$ dostanu, že existuje p o velikosti $O(\ln D) = O(n^2 \ln N)$ takové, že $\phi_p(x) = x \mod p$ je perfektní pro S.

Deterministické nalezení trvá $O(n^3 \log n \log N)$ (test perfektnosti každého systému je $O(n\log n)$). Proto použijeme pravděpodobnostní algoritmus (mezi $4cn^2 \ln N$ přir. čísly je aspoň 1/2 prvočísel, která vyhovují): nejdřív najde prvočíslo a pak testuje perfektnost. Očekávaný počet testů je $O(\ln(4cn^2 \ln N))$, celková složitost algoritmu je pak $O(n \log n(\log n + \log\log N))$. Najde zhruba až $2\times$ větší prvočíslo než deterministický.

Tuto funkci použijeme ke konstrukci výsledné hash-funkce:

1. Nalezneme prvočíslo q_0, aby $\phi_{q_0}(x) = x \mod q_0$ byla perfektní pro S
2. Položíme $S_1 = \{\phi_{q_0}(s) | s \in S\}$, pak najdeme prvočíslo $q_1 \in \langle n(n-1)+1, 2n(n-1)+2\rangle$
3. K němu existuje $l \in \langle 1, q_0 - 1\rangle$ takové, že $h_l(x) = ((lx \mod q_0) \mod q_1)$ je perfektní pro S_1
4. Položíme $S_2 = \{h_l(s) | s \in S_1\}$ a najdeme perfektní g pro S_2 do tabulky s méně než $3n$ řádky (viz výše, počítá se ale do univerza o velikosti q_1).
5. Pak $f(x) = g(h_l(\phi_{q_0}(x)))$ je perfektní.

Funkce q_0 je určena 1 číslem o velikosti $O(n^2 \log N)$, h_l 2 čísly o velikosti $O(n^2)$ a $O(q_0)$, g je určená $n+1$ čísly o $O(q_1)$, tj. celkem zadání vyžaduje $O(n\log n + \log n + \log\log N)$ paměti.