Datové struktury – hašování

I. Úvod

Základním tématem této přednášky je reprezentace množin a operací s nimi. V řadě úloh a algoritmů je tento typ podproblémů rozhodující pro složitost řešení, protože tyto operace se mnohokrát opakují. Proto je třeba navrhnout pro tyto úlohy co nejefektivnější algoritmy (každý ušetřený čas mnohonásobným opakováním začne hrát důležitou roli) a detailně analyzovat jejich složitost v závislosti na vnějších okolnostech. Ukazuje se, že teoretická analýza je sice pracnější, ale často efektivnější než experimenty s danou úlohou (leckdy ani nejsou možné). To motivuje detailní studium reprezentací a jejich porovnání v konkrétních situacích. Nelze ale jednoznačně říct, že některý algoritmus je nejlepší, protože za určitých okolností může být 'méně efektivní' algoritmus výhodnější.

Tato přednáška se těmto konkrétním situacím nebude věnovat, naším cílem je seznámit se se základními datovými strukturami a s metodami odhadu jejich složitosti. Jejich rozvíjení v konkrétních případech bude na vás, buď si ho najdete v literatuře nebo si ho uděláte sami. Porovnání datových struktur vyžaduje velké znalosti z kombinatoriky, pravděpodobnosti, numeriky a statistiky a jejich použití je probíráno na výběrových přednáškách. V této přednášce předpokládáme jen základní znalosti z pravděpodobnosti, numeriky a kombinatoriky, což omezuje detailnost prezentovaných metod.

Kromě popisu datových struktur a algoritmů realizujících operace s daty se alespoň na základní úrovni budeme zabývat složitostí těchto algoritmů. Každý algoritmus bude doplněn svou časovou složitostí – buď časovou složitostí v nejhorším případě nebo očekávanou časovou složitostí. Je třeba si uvědomit, že z praktického pohledu je užitečnější očekávaná složitost, ale abychom získali použitelné výsledky, je třeba znát rozložení vstupních dat. Když ho neznáme, tak získané výsledky mohou být zavádějící. Pro úlohy v dávkovém režimu má větší vypovídající hodnotu amortizovaná složitost než cena jednotlivé operace. Proto u řady algoritmů uvedeme amortizovanou složitost. Také v některých případech uvedeme paměťovou náročnost popisované datové struktury. V konkrétní úloze na konkrétním počítači nás zajímá přesná složitost. Bohužel, pokud máme popisovat abstraktně algoritmus, tak se jednotlivé implementace liší – záleží na konkrátní architektuře počítače, na použití cache-paměti, ale i na vstupních datech. To vede k tomu, že obecně se udává jen asymptotická složitost modulo O. Připomínáme, že $g = O(f)$, když existuje $c > 0$ takové, že $g(n) \leq cf(n)$ pro každé n až na konečně mnoho výjimek, $g = o(f)$, když $\lim_{n \mapsto \infty} \frac{g(n)}{f(n)} = 0$.

Základním problémem v této přednášce je slovníkový problém: Máme univerzum U a naším úkolem je reprezentovat množinu $S \subseteq U$ a navrhnout algoritmy pro následující operace:
MEMBER(x) – zjistí, zda $x \in S$, a nalezne jeho uložení,
INSERT(x) – když $x \notin S$, pak vloží x do struktury reprezentující S,
DELETE(x) – když $x \in S$, pak odstraní x ze struktury reprezentující S.

Literatura:
K. Mehlhorn: Data Structures and Algorithms 1: Sorting and Searching, Springer - Verlag, 1984
http://www.mpi-sb.mpg.de/mehlhorn/DatAlgbooks.html

J. S. Vitter, W.-Ch. Chen: Design and Analysis of Coalesced Hashing, Oxford Univ. Press, 1987

M. A. Weiss: Data Structures and Algorithm Analysis, The Benjamin/Cummings Publishing Comp. Inc., 1992

II. Hašování se separovanými řetězci

Když reprezentujeme množinu pomocí charakteristické funkce uložené v poli s přímým přístupem, pak implementace operací **MEMBER**, **INSERT** a **DELETE** vyžaduje $O(1)$ času. Pro velká univerza je okamžitě vidět nevýhoda této reprezentace, neboť vyžaduje velké množství paměti a v některých případech je nerealizovatelná – pole pro tuto velikost nelze zadat do počítače. Hašování chce zachovat rychlost operací, ale odstranit paměťovou náročnost.
První publikovaný článek o hašování je od W. W. Petersona: Addressing for Random Access Storage, publikovaný v roce 1957 v časopise IBM Journal of Research and Development, ale existuje starší technická zpráva od IBM o hašování z roku 1953.

Základní idea hašování je následující: Mějme univerzum U a množinu $S \subseteq U$ takovou, že $|S| << |U|$. Dále mějme funkci $h : U \longrightarrow \{0, 1, \ldots, m-1\}$. Množinu S reprezentujeme tabulkou (polem) s m řádky tak, že prvek $s \in S$ je uložen na řádku $h(s)$.
Nevýhodou je, že mohou existovat různá $s, t \in S$ taková, že $h(s) = h(t)$ – tento jev se nazývá kolize. Základní způsob řešení kolizí, spočívá v tom, že použijeme pole o rozsahu $[0..m-1]$, jehož i-tá položka bude spojový seznam S_i obsahující všechny prvky $s \in S$ takové, že $h(s) = i$. Toto řešení se nazývá hašování se separovanými řetězci.

Příklad: $U = \{1, 2, \ldots, 1000\}$, $S = \{1, 7, 11, 53, 73, 141, 161\}$ a hašovací funkce je $h(x) = x \bmod 10$. Pak $P(0) = P(2) = P(4) = P(5) = P(6) = P(8) = P(9) = NIL$ jsou prázdné seznamy,
$P(7) = 7 \mapsto NIL$, $P(3) = 53 \mapsto 73 \mapsto NIL$, $P(1) = 1 \mapsto 141 \mapsto 11 \mapsto 161 \mapsto NIL$.
Seznamy nemusí být uspořádané – jsou výsledkem konkrétní posloupnosti operací s prvky z dané množiny.

Algoritmy

Neformální popis algoritmů: Nejprve vypočteme hodnotu $h(x)$ hašovací funkce h v argumentu operace x a pak prohledáním řetězce začínajícího v místě $h(x)$ zjistíme, zda x je či není prvkem tohoto řetězce, protože pokud x patří do reprezentované množiny, pak nutně musí v tomto řetězci ležet. Tím dostaneme výsledek operace **MEMBER**. Operace **INSERT** nejprve zjistí, zda x je v řetězci, a pokud není, přidá ho na konec řetězce (v opačném případě neděláme už nic). Rovněž operace **DELETE** vyhledá prvek x a pokud je v řetězci, odstraní ho (v opačném případě nedělá nic). Podstatné je, že řetězce jsou prosté, tj. žádný prvek se v žádném řetězci se nevyskytuje dvakrát.

Formální zápis algoritmů:

MEMBER(x):
$i := h(x)$, $t := NIL$

if $S_i \neq NIL$ **then**
  $t :=$ první prvek S_i
  while $t.key \neq x$ a $t \neq$ poslední prvek S_i **do**
    $t :=$ následující prvek S_i
  enddo
endif
if $t.key = x$ **then Výstup:** $x \in S$ **else Výstup:** $x \notin S$ **endif**

INSERT(x):
$i := h(x)$, $t := NIL$
if $S_i \neq NIL$ **then**
  $t :=$ první prvek S_i
  while $t.key \neq x$ a $t \neq$ poslední prvek S_i **do**
    $t :=$ následující prvek S_i
  enddo
endif
if $t.key \neq x$ **then** vytvoř prvek reprezentující x a vlož ho do S_i **endif**

DELETE(x):
$i := h(x)$, $t := NIL$
if $S_i \neq NIL$ **then**
  $t :=$ první prvek S_i
  while $t.key \neq x$ a $t \neq$ poslední prvek S_i **do**
    $t :=$ následující prvek S_i
  enddo
endif
if $t.key = x$ **then** odstraň prvek reprezentující x z S_i **endif**

Analýza složitosti

V následující analýze předpokládáme, že hodnota funkce $h(x)$ je spočitatelná v čase $O(1)$. Zřejmě časová složitost operací v nejhorším případě je $O(|S|)$ (když všechny prvky jsou v jednom seznamu) a paměťová složitost datové struktury je $O(m + |S|)$ (předpokládáme, že uložení každého $s \in S$ vyžaduje paměť $O(1)$).
Paměť tedy není efektivně využita. Může se stát, že některý seznam je prázdný (přesto vyžaduje paměť na svou inicializaci) a některý seznam obsahuje více prvků (vyžaduje $O(1+$ délka seznamu) paměťových buněk).

Algoritmy zřejmě vyžadují čas přímo úměrný délce prohledávaného řetězce. Proto spočítáme odhad očekávané délky řetězců, a to za následujících předpokladů:
1) h je rychle spočitatelná – budeme předpokládat, že výpočet $h(x)$ vyžaduje $O(1)$ času a požadovaný čas je nezávislý na vstupu,
2) h rozděluje univerzum U rovnoměrně (tj. $-1 \leq |h^{-1}(i)| - |h^{-1}(j)| \leq 1$ pro $i, j \in \{0, 1, \ldots, m-1\}$),
3) S je náhodně vybraná z univerza U, tj. pro každé n platí, že každá množina S velikosti n je vybrána s pravděpodobností $\frac{1}{\binom{|U|}{n}}$,
4) každý prvek z U má stejnou pravděpodobnost být argumentem operace.

Velikost S označme n, velikost tabulky (pole seznamů) označme m, délku i-tého řetězce S_i označme $\ell(i)$ a faktor naplnění (load factor) označme $\alpha = \frac{n}{m}$.

Za těchto předpokladů pro náhodně zvolený prvek $x \in U$ a každé $i \in \{0, 1, \ldots, m-1\}$ platí, že $\text{Prob}(h(x) = i) = \frac{1}{m}$. Pro množinu S platí, že i-tý seznam má délku l, když existuje l-prvková podmnožina A množiny S (takových podmnožin je $\binom{n}{l}$), pro všechna $x \in A$ platí $h(x) = i$ (pravděpodobnost tohoto jevu je $(\frac{1}{m})^l$) a pro všechna $x \in S \setminus A$ platí $h(x) \neq i$ (pravděpodobnost tohoto jevu je $(1 - \frac{1}{m})^{n-l}$). Tedy $\text{Prob}(\ell(i) = l) = p_{n,l} = \binom{n}{l}(\frac{1}{m})^l(1 - \frac{1}{m})^{n-l}$.

Poznámka. Všimněme si, že tuto pravděpodobnost jsme odvodili za poněkud zjednodušeného předpokladu, který platí pouze v případě, že univerzum U je nekonečné a délka jednotlivých seznamů neomezená. Ve skutečnosti, když velikost univerza je N, pak z požadavku 2) plyne, že v něm pro každé pevně dané i existuje pouze $\frac{N}{m}$ prvků, pro které platí $h(x) = i$. Tedy pouze první náhodně vybraný prvek bude patřit do i-tého seznamu S_i s pravděpodobností $\frac{\frac{N}{m}}{N} = \frac{1}{m}$. Z požadavku 3) plyne, že pro $l \leq \min\{n, \frac{N}{m}\}$ platí

$$\begin{aligned}
\text{Prob}(\ell(i) = l) = &\frac{\binom{\frac{N}{m}}{l}\binom{N-\frac{N}{m}}{n-l}}{\binom{N}{n}} = \frac{\frac{\prod_{i=0}^{l-1}(\frac{N}{m}-i)}{l!}\frac{\prod_{i=0}^{n-l-1}(N-\frac{N}{m}-i)}{(n-l)!}}{\frac{\prod_{i=0}^{n-1}(N-i)}{n!}} = \\
&\frac{n!}{l!(n-l)!}\frac{(\frac{N}{m})^l(N-\frac{N}{m})^{n-l}}{N^n}\frac{\prod_{i=0}^{l-1}(1-\frac{im}{N})\prod_{i=0}^{n-l-1}(1-\frac{im}{N(m-1)})}{\prod_{i=0}^{n-1}(1-\frac{i}{N})} = \\
&\binom{n}{l}(\frac{1}{m})^l(1-\frac{1}{m})^{n-l}\frac{\prod_{i=0}^{l-1}(1-\frac{im}{N})\prod_{i=0}^{n-l-1}(1-\frac{im}{N(m-1)})}{\prod_{i=0}^{n-1}(1-\frac{i}{N})}.
\end{aligned}$$

Když N je podstatně větší než n^2m, pak

$$\frac{\prod_{i=0}^{l-1}(1-\frac{im}{N})\prod_{i=0}^{n-l-1}(1-\frac{im}{N(m-1)})}{\prod_{i=0}^{n-1}(1-\frac{i}{N})}$$

je přibližně 1, a tedy můžeme pro velikost pravděpodobnosti použít naši aproximaci. Navíc v limitním přechodu pro $N \mapsto \infty$ (pro pevné n, m a l) se skutečná pravděpodobnost shoduje s touto aproximací.

Očekávaná délka řetězců

Připomeňme, že očekaváná hodnota náhodné proměnné je součet všech jejích hodnot vynásobených pravděpodobnostmi, že náhodná proměnná nabývá právě této hodnoty. Abychom spočítali očekávanou délku řetězce, dosadíme vypočtené pravděpodobnosti do tohoto vztahu

a upravíme výraz použitím definice kombinačních čísel a binomické věty

$$\begin{aligned}
E(l) =& \sum_{l=0}^{n} l p_{n,l} = \sum_{l=0}^{n} l \binom{n}{l} (\frac{1}{m})^l (1-\frac{1}{m})^{n-l} = \sum_{l=0}^{n} l \frac{n!}{l!(n-l)!} (\frac{1}{m})^l (1-\frac{1}{m})^{n-l} = \\
& \frac{n}{m} \sum_{l=1}^{n} \frac{(n-1)!}{(l-1)!(n-l)!} (\frac{1}{m})^{l-1} (1-\frac{1}{m})^{n-l} = \\
& \frac{n}{m} \sum_{l=1}^{n} \binom{n-1}{l-1} (\frac{1}{m})^{l-1} (1-\frac{1}{m})^{(n-1)-(l-1)} = \\
& \frac{n}{m} \sum_{l=0}^{n-1} \binom{n-1}{l} (\frac{1}{m})^{l} (1-\frac{1}{m})^{n-1-l} = \frac{n}{m} (\frac{1}{m} + 1 - \frac{1}{m})^{n-1} = \frac{n}{m}.
\end{aligned}$$

Výpočet rozptylu

Očekávaná hodnota je důležitou charakteristikou náhodné proměnné, ale nevystihuje úplně její chování. Jednou z dalších důležitých charakteristik je vzdálenost hodnot náhodné proměnné od její očekávané hodnoty, kterou udává její rozptyl. Rozptyl je definován jako očekávaná hodnota druhé mocniny z rozdílu náhodné proměnné a její očekávané hodnoty. Abychom ho spočítali, tak vypočteme nejprve druhý moment, tj. očekávanou hodnotu druhé mocniny samotné náhodné proměnné – délky řetězce. Nejprve použijeme elementární úpravu a fakt, že součet očekávaných hodnot náhodných proměnných je očekávaná hodnota jejich součtu. Pak dosadíme spočítané pravděpodobnosti a provedeme analogické úpravy jako v předchozím výpočtu

$$\begin{aligned}
E(l^2) =& E(l(l-1)) + E(l), \\
E(l(l-1)) =& \sum_{l=0}^{n} l(l-1) \binom{n}{l} (\frac{1}{m})^l (1-\frac{1}{m})^{n-l} = \\
& \frac{n(n-1)}{m^2} \sum_{l=2}^{n} \binom{n-2}{l-2} (\frac{1}{m})^{l-2} (1-\frac{1}{m})^{(n-2)-(l-2)} = \\
& \frac{n(n-1)}{m^2} \sum_{l=0}^{n-2} \binom{n-2}{l} (\frac{1}{m})^{l} (1-\frac{1}{m})^{n-2-l} = \frac{n(n-1)}{m^2}, \\
E(l^2) =& \frac{n(n-1)}{m^2} + \frac{n}{m} = \frac{n}{m}(1 + \frac{n-1}{m}).
\end{aligned}$$

Nyní už dasazením vypočteme rozptyl délky řetězce

$$\mathrm{var}(l) = E(l - E(l))^2 = E(l^2) - (E(l))^2 = \frac{n}{m}(1 + \frac{n-1}{m}) - (\frac{n}{m})^2 = \frac{n}{m}(1 - \frac{1}{m}).$$

Shrneme výsledky:

Věta. *V modelu hašování se separovanými řetězci je očekávaná délka řetězců rovna* $\frac{n}{m}$ *a rozptyl délky řetězců je* $\frac{n}{m}(1 - \frac{1}{m})$.

Očekávaný nejhorší případ

Vypočítáme ENP očekávanou délku maximálního řetězce. K tomu použijeme následující vztah:

$$\mathrm{Prob}(\max_i \ell(i) = j) = \mathrm{Prob}(\max_i \ell(i) \geq j) - \mathrm{Prob}(\max_i \ell(i) \geq j+1).$$

Pak můžeme počítat

$$\begin{aligned}
ENP = &\sum_j j\,\mathrm{Prob}(\max_i \ell(i) = j) = \sum_j j(\mathrm{Prob}(\max_i \ell(i) \geq j) - \mathrm{Prob}(\max_i \ell(i) \geq j+1)) = \\
&\sum_j j\,\mathrm{Prob}(\max_i \ell(i) \geq j) - \sum_j j\,\mathrm{Prob}(\max_i \ell(i) \geq j+1) = \\
&\sum_j j\,\mathrm{Prob}(\max_i \ell(i) \geq j) - \sum_j (j-1)\,\mathrm{Prob}(\max_i \ell(i) \geq j) = \\
&\sum_j (j-j+1)\,\mathrm{Prob}(\max_i \ell(i) \geq j) = \sum_j \mathrm{Prob}(\max_i \ell(i) \geq j).
\end{aligned}$$

Vysvětlení: Při čtvrté rovnosti se ve druhé sumě zvětšil index, přes který sčítáme, o 1, v páté rovnosti se k sobě daly koeficienty při stejných pravděpodobnostech ve dvou sumách.

Dále

$$\begin{aligned}
\mathrm{Prob}(\max_i \ell(i) \geq j) = &\mathrm{Prob}(\ell(1) \geq j \vee \ell(2) \geq j \vee \cdots \vee \ell(m-1) \geq j) \leq \\
&\sum_i \mathrm{Prob}(\ell(i) \geq j) \leq m\binom{n}{j}(\frac{1}{m})^j = \\
&\frac{\prod_{k=0}^{j-1}(n-k)}{j!}(\frac{1}{m})^{j-1} \leq n(\frac{n}{m})^{j-1}\frac{1}{j!}.
\end{aligned}$$

Vysvětlení: První nerovnost plyne z toho, že pravděpodobnost disjunkce jevů je menší nebo rovna součtu pravděpodobností jevů, druhá nerovnost plyne z toho, že i-tý řetězec má délku alespoň j, jakmile existuje podmnožina $A \subseteq S$ taková, že $|A| = j$ (těchto možností je $\binom{n}{j}$) a pro každé $x \in A$ platí $h(x) = i$ (pravděpodobnost tohoto jevu je $(\frac{1}{m})^j$).

Důsledek. $\mathrm{Prob}(\max_i \ell(i) \geq j) \leq \min\{1, n(\frac{n}{m})^{j-1}\frac{1}{j!}\}$.

Nyní za předpokladu, že $\alpha = \frac{n}{m} \leq 1$, odhadneme hodnotu $j_0 = \min\{j \mid n(\frac{n}{m})^{j-1}\frac{1}{j!} \leq 1\}$. Ukážeme, že pro dostatečně velká n platí $j_0 \leq \frac{8\log n}{\log\log n}$. Z $\frac{n}{m} \leq 1$ a z $(\frac{j}{2})^{\frac{j}{2}} \leq j!$ plyne

$$\begin{aligned}
\min\{j \mid n(\frac{n}{m})^{j-1}\frac{1}{j!} \leq 1\} \leq &\min\{j \mid \frac{n}{j!} \leq 1\} \leq \\
&\min\{j \mid n \leq (\frac{j}{2})^{\frac{j}{2}}\}
\end{aligned}$$

pro každé $n \geq 1$, kde j probíhá celá čísla. Pro pevné n označme $k+1 = \min\{j \mid n \leq (\frac{j}{2})^{\frac{j}{2}}\}$, pak

$$(\frac{k}{2})^{\frac{k}{2}} < n \leq (\frac{k+1}{2})^{\frac{k+1}{2}}.$$

Nyní toto dvakrát zlogaritmujeme a protože logaritmus je rostoucí funkce (používáme jen logaritmy o základu větším než 1), dostáváme

$$(\frac{k}{2})\log(\frac{k}{2}) < \log n \leq \frac{k+1}{2}\log(\frac{k+1}{2})$$
$$\log(\frac{k}{2}) + \log\log(\frac{k}{2}) < \log\log n \leq \log(\frac{k+1}{2}) + \log\log(\frac{k+1}{2}).$$

Tedy $\log\log n < 2\log(\frac{k+1}{2})$ a za předpokladu, že $k \geq 3$, dostaneme

$$\frac{k}{8} < \frac{k}{4}\frac{\log(\frac{k}{2})}{\log(\frac{k+1}{2})} < \frac{\log n}{\log\log n},$$

protože pro $k \geq 3$ je $\frac{1}{2} < \frac{\log(\frac{k}{2})}{\log(\frac{k+1}{2})}$.
Doplňující výpočet: Ukážeme sofistikovanější metodu, která přímo odhaduje $k_0 = \min\{j \mid n \leq j!\}$ a dává lepší odhad. Ze Stirlingovy aproximace

$$m! = \sqrt{2\pi m}(\frac{m}{e})^m(1 + \frac{1}{12m} + O(m^2))$$

plyne, že $\log m! = (1 + o(1))m\log m$. Odtud pro každé $c > 1$ existuje m_c takové, že pro každé $m \geq m_c$ platí

$$\frac{1}{c}m\log m \leq \log m! \leq cm\log m.$$

Zafixujme libovolné $c > 1$. Protože z definice k_0 plyne, že $(k_0 - 1)! < n \leq k_0!$, dostaneme pro dostatečně velká n, že

$$\frac{1}{c}(k_0 - 1)\log(k_0 - 1) \leq \log(k_0 - 1)! < \log n \leq \log k_0! \leq ck_0\log k_0.$$

Dalším zlogaritmováním získáme

$$\log(k_0 - 1) - \log c + \log\log(k_0 - 1) < \log\log n \leq \log k_0 + \log c + \log\log k_0.$$

Uvědomíme-li si, že c je konstanta a k_0 roste nade všechny meze s rostoucím n, pak bude existovat n_c takové, že pro všechna $n \geq n_c$ můžeme tuto nerovnost zjednodušit na tvar

$$\log(k_0 - 1) < \log\log n \leq c\log k_0,$$

protože platí $\log c < \log\log k_0$ pro dostatečně velká n. Použijeme, že

$$\lim_{k_0 \mapsto \infty} \frac{\log k_0}{\log(k_0 - 1)} = \lim_{k_0 \mapsto \infty} \frac{\log(k_0 - 1)}{\log k_0} = 1$$

a pomocí odvozených nerovností získáme následující odhad pro $\frac{\log n}{\log\log n}$:

$$\frac{1}{c}(k_0-1) < \frac{1}{c^2}(k_0-1)\frac{\log(k_0-1)}{\log k_0} < \frac{\log n}{\log\log n} < ck_0\frac{\log k_0}{\log(k_0-1)} < c^2k_0.$$

Protože pro každé $c>1$ existuje n_c takové, že toto platí pro všechna $n \geq n_c$, dostáváme, že $k_0 = (1+o(1))\frac{\log n}{\log\log n}$. Tedy platí $j_0 = O(\frac{\log n}{\log\log n})$, protože $j_0 \leq k_0$.

To použijeme k dokončení odhadu ENP:

$$\begin{aligned} ENP = &\sum_j \mathrm{Prob}(\max_i \ell(i) \geq j) \leq \sum_j \min\{1, n(\frac{n}{m})^{j-1}\frac{1}{j!}\} = \\ &\sum_{j=1}^{j_0} 1 + \sum_{j=j_0+1}^{\infty}\left(n(\frac{n}{m})^{j-1}\frac{1}{j!}\right) \leq j_0 + \sum_{j=j_0+1}^{\infty}\frac{n}{j!} = j_0 + \frac{n}{j_0!}\sum_{j=j_0+1}^{\infty}\frac{j_0!}{j!} \leq \\ &j_0 + \sum_{j=j_0+1}(\frac{1}{j_0+1})^{j-j_0} = j_0 + \frac{\frac{1}{j_0+1}}{-\frac{1}{j_0+1}+1} = j_0 + \frac{1}{j_0} = O(j_0). \end{aligned}$$

Vysvětlení: Při druhé nerovnosti jsme použili, že $\frac{n}{m} \leq 1$, při třetí, že $\frac{n}{j_0!} \leq 1$ a

$$\frac{j_0!}{j!} = \frac{1}{\prod_{k=j_0+1}^{j} k} \leq (\frac{1}{j_0+1})^{j-j_0}.$$

Zformulujeme získaný výsledek:

Věta. *Za předpokladu, že $\alpha = \frac{n}{m} \leq 1$, je při hašování se separovanými řetězci horní odhad očekávané délky maximálního řetězce $O(\frac{\log n}{\log\log n})$.*
Když $0.5 \leq \alpha \leq 1$, je to zároveň i dolní odhad (bez důkazu).

Očekávaný počet testů

Testem budeme rozumět porovnání argumentu operace s prvkem na daném místě řetězce nebo zjištění, že vyšetřovaný řetězec je prázdný.

Počet testů, které potřebuje algoritmus pro provedení operace, můžeme chápat jako jiný kvantitativní odhad efektivity dané metody.
Při vyhledávání budeme rozlišovat dva případy:
úspěšné vyhledávání, když hledaný prvek je mezi prvky reprezentované množiny,
neúspěšné vyhledávání, když hledaný prvek není mezi prvky reprezentované množiny. Ukážeme obecnou ideu, jak za jistých předpokladů lze z očekávaného počtu testů při neúspěšném vyhledávání spočítat očekávaný počet testů při úspěšném vyhledávání. Nejprve ale budeme oba případy analyzovat přímo.

Neúspěšné vyhledávání

Očekávaný počet testů v tomto případě je

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

Vysvětlení: Zjištění, zda řetězec je prázdný, vyžaduje jeden test, tj. $\text{Prob}(\ell(i) = 0)$ není v součtu s koeficientem 0, ale 1. Protože z našich předpokladů plyne, že pravděpodobnost, že délka daného řetězce je k, je stejná pro všechny řetězce, nemusíme specifikovat řetězec, který vyšetřujeme, stačí psát obecně i. Součet $\sum_l l p_{n,l}$ jsme spočítali při výpočtu očekávané délky řetězce. Uvědomme si, že $p_{n,0} = (1 - \frac{1}{m})^n$, protože je to pravděpodobnost, že žádný prvek z reprezentované množiny nepatří do daného řetězce.

Úspěšné vyhledávání

Zvolme jeden řetězec prvků o délce l. Počet testů při vyhledání všech prvků v tomto řetězci je

$$1 + 2 + \cdots + l = \binom{l+1}{2}.$$

Očekávaný počet testů při vyhledání všech prvků v daném řetězci je $\sum_l \binom{l+1}{2} \text{Prob}(\ell(i) = l) = \sum_l \binom{l+1}{2} p_{n,l}$. Proto očekáný počet testů při vyhledání všech prvků v tabulce je $m \sum_l \binom{l+1}{2} p_{n,l}$, a tedy očekávaný počet testů při vyhledání jednoho prvku je

$$\begin{aligned}
\frac{m}{n} \sum_{l=0}^{n} \binom{l+1}{2} p_{n,l} =& \frac{m}{2n} \Big(\sum_{l=0}^{n} l^2 p_{n,l} + \sum_{l=0}^{n} l p_{n,l} \Big) = \\
& \frac{m}{2n} \Big(\sum_{l=1}^{n} l(l-1) p_{n,l} + 2 \sum_{l=1}^{n} l p_{n,l} \Big) = \\
& \frac{m}{2n} \Big(\frac{n(n-1)}{m^2} + \frac{2n}{m} \Big) = \frac{n-1}{2m} + 1 \approx \\
& 1 + \frac{\alpha}{2}.
\end{aligned}$$

Nyní uvedeme jinou, obecnější metodu pro výpočet očekávaného počtu testů při úspěšném vyhledávání. Tuto metodu lze použít, když je splněn následující předpoklad:

> Testy při úspěšném vyhledávání prvku $s \in S$ jsou (až na ten poslední) totožné s testy, které byly provedeny při neúspěšném vyhledávání s v okamžiku, kdy se tento prvek vkládal do tabulky. (Jediný rozdíl je, že místo vložení s se provede ještě jedno porovnání, které zjistí, že s se v tabulce vyskytuje.)

Když je splněn tento předpoklad, pak lze touto metodou spočítat očekávaný počet testů při úspěšném vyhledávání na základě znalosti očekávaného počtu testů při neúspěšném vyhledávání.

Počet testů při úspěšném vyhledávání prvku $x \in S$ je 1+počet porovnání klíčů při neúspěšném vyhledávání x v operaci **INSERT**(x) (nepočítá se test, který zjišťuje, že řetězec byl prázdný). Počet porovnání klíčů je délka řetězce, a proto očekávaný počet porovnání klíčů je očekávaná délka řetězce. Tedy očekávaný počet testů při úspěšném vyhledávání x je 1+očekávaná délka řetězce v okamžiku vkládání x, neboli

$$\frac{1}{n}\sum_{i=0}^{n-1}(1+\frac{i}{m}) = 1+\frac{n-1}{2m}.$$

Věta. *V hašování se separovanými řetězci je očekávaný počet testů při neúspešném vyhledávaní přibližně* $e^{-\alpha}+\alpha$ *a při úspěšném vyhledávání přibližně* $1+\frac{\alpha}{2}$.

Následující tabulka úvádí přehled očekávaného počtu testů pro různé hodnoty α:

α	0	0.1	0.2	0.3	0.4	0.5	0.6
neúsp. vyh.	1	1.005	1.019	1.041	1.07	1.107	1.149
úspěš. vyh.	1	1.05	1.1	1.15	1.2	1.25	1.3

α	0.7	0.8	0.9	1	2	3
neúsp. vyh.	1.196	1.249	1.307	1.368	2.135	3.05
úspěš. vyh.	1.35	1.4	1.45	1.5	2	2.5

Všimněme si, že očekávaný počet testů při neúspěšném vyhledávání je menší než očekávaný počet testů při úspěšném vyhledávání, když $\alpha \leq 1$. Na první pohled vypadá tento výsledek nesmyslně, ale důvodem je, že počet testů při úspěšném vyhledávání počítáme jako průměr přes n, kdežto při neúspěšném vyhledávání přes m. Ilustrujeme to na následujícím příkladu: Nechť $n = \frac{m}{2}$ a nechť polovina neprázdných řetězců má délku 1 a polovina má délku 2.
Očekávaný počet testů při neúspěšném vyhledávaní:
1 test pro prázdné řetězce a řetězce délky 1 – těchto případů je $\frac{5m}{6}$,
2 testy pro řetězce délky 2 – těchto případů je $\frac{m}{6}$.
Očekávaný počet testů je $\frac{1}{m}(1\frac{5m}{6}+2\frac{m}{6}) = \frac{7}{6}$.
Očekávaný počet testů při úspěšném vyhledávání:
1 test pro prvky na prvním místě řetězce – těchto případů je $\frac{2n}{3}$,
2 testy pro prvky, které jsou na druhém místě řetězce – těchto případů je $\frac{n}{3}$.
Očekávaný počet testů je $\frac{1}{n}(1\frac{2n}{3}+2\frac{n}{3}) = \frac{4}{3}$.

Pro efektivní vyhledávání je doporučována velikost faktoru naplnění α menší než 1. Důvodem je, že za tohoto předpokladu je pravděpodobnost kolizí malá. Ale hodnota α nemá být příliš malá, protože by paměť nebyla efektivně využita.

III. Hašování s uspořádanými řetězci

Jedinný rozdíl proti předchozí metodě je, že prvky v řetězcích S_i jsou uspořádány ve vzrůstajícím pořadí. Protože řetězce obsahují tytéž prvky, je počet očekávaných testů při úspěšném vyhledávání stejný jako v případě neuspořádaných řetězců. Při neúspěšném vyhledávání končíme, když argument operace je menší než vyšetřovaný prvek v řetězci, tedy končíme dřív. Následující věta (bez důkazu) uvádí očekavaný počet testů v neúspěšném případě.

Věta. *Očekávaný počet testů při neúspěšném vyhledávání pro hašování s uspořádanými separovanými řetězci je přibližně* $e^{-\alpha} + 1 + \frac{\alpha}{2} - \frac{1}{\alpha}(1 - e^{-\alpha})$. *Očekávaný počet testů při úspěšném vyhledávání je přibližně* $1 + \frac{\alpha}{2}$.

Pro úplnost uvedeme ještě algoritmy pro operace v datové struktuře s uspořádanými separovanými řetězci.

Algoritmy

Neformální popis algoritmů: Algoritmy pro práci s uspořádanými řetězci se liší od algoritmů pro separující řetězce hlavně při vyhledávání. V tomto případě skončíme vyhledávaní, když nalezneme konec řetězce anebo nalezneme prvek větší než hledaný prvek. Tato změna se projevuje ještě v operaci **INSERT**. Nový prvek vkládáme před místo, kde jsme skončili vyhledávání (tedy před prvek, který ukončil vyhledávání).

Formální popis algoritmů.

MEMBER(x):
$i := h(x)$, $t := NIL$
if $S_i \neq NIL$ **then**
  $t :=$ první prvek S_i
  while $t.key < x$ a $t \neq$ poslední prvek S_i **do**
    $t :=$ následující prvek S_i
  enddo
endif
if $t.key = x$ **then Výstup:** $x \in S$ **else Výstup:** $x \notin S$ **endif**

INSERT(x):
$i := h(x)$, $t := NIL$
if $S_i \neq NIL$ **then**
  $t :=$ první prvek S_i
  while $t.key < x$ a $t \neq$ poslední prvek S_i **do**
    $t :=$ následující prvek S_i
  enddo
endif
if $t.key \neq x$ **then**
  if $x < t.key$ **then**
    vytvoř prvek reprezentující x a vlož ho do S_i před prvek t
  else
    vytvoř prvek reprezentující x a vlož ho do S_i za prvek t
  endif
endif

DELETE(x):
$i := h(x)$, $t := NIL$
if $S_i \neq NIL$ **then**
  $t :=$ první prvek S_i

while $t.key < x$ a $t \neq$ poslední prvek S_i **do**
$t :=$ následující prvek S_i
enddo
endif
if $t.key = x$ **then** odstraň prvek reprezentující x z S_i **endif**

Nevýhodou metody hašovaní se separovanými řetězci je neefektivní využití paměti – stává se, že zatímco některá místa původního pole zůstávají prázdná, musíme zároveň pro kolidující prvky dodatečně alokovat další paměť. Řešením je najít vhodný způsob uložení řetězců kolidujících prvků do původní tabulky. Nevýhodou je, že pak každý řádek musí mít navíc položky umožňující práci s tabulkami. Položky řádku tabulky jsou rozděleny do tří skupin: klíč (key), položky pro odkaz na uložená data a položky pro práci s tabulkou. Předpokládáme, že data jsou velká, a v tom případě není výhodné, aby se ukládala přímo do tabulky, protože jejich přenos by vyžadoval hodně času. Proto se ukládají mimo tabulku a v tabulce je jen odkaz na ně. Protože tyto odkazy nemají vliv na vlastní práci s tabulkou, budeme je při popisu algoritmů vynechávat (tj. data budou reprezentována pouze klíčem).

Následující metody ilustrují fakt, že čím je sofistikovanější strategie, tím více položek pro práci s tabulkou vyžaduje a tím má větší nároky na paměť. Naším úkolem při návrhu hašování je nalézt vhodný kompromis mezi použitou strategií a paměťovou náročností. Pokud nechceme mít žádné položky pro práci s tabulkou, pak použité strategie jsou hodně omezené a tím je omezeno i použití těchto metod. Proto u každé popsané metody uvedeme její základní vlastnosti určující složitost a možnost jejího použití.

Uvedeme následující metody řešení kolizí:
hašování s přemísťováním, hašování s dvěma ukazateli,
srůstající hašování,
dvojité hašování a hašování s lineárním přidáváním.
Toto nejsou všechny existující metody řešení kolizí. Objevují se stále nové metody a uvést vyčerpávající přehled není možné kvůli zachování rozumné délky textu. Lze však říci, že následující metody jsou asi nejdůležitější a nejvíce prostudované.

IV. Hašování s přemísťováním

Idea této metody je přímočará, řetězce jsou implementovány jako dvousměrné seznamy a ukazatelé na předchozí a následující prvek jsou určeny položkami pro práci s tabulkou. Když vznikne kolize, tj. když chceme vložit prvek a jeho místo v tabulce je obsazeno prvkem z jiného řetězce, pak tento prvek z jiného řetězce přemístíme na jiný volný řádek tabulky (proto se této metodě říká hašování s přemísťováním).

Položky pro práci s tabulkou jsou next a previous a jejich význam je následující:
next – číslo řádku tabulky obsahující následující prvek řetězce,
previous – číslo řádku tabulky obsahující předchozí prvek řetězce.

Příklad: $U = \{1, 2, \ldots, 1000\}$, $S = \{1, 7, 11, 53, 73, 141, 161\}$ a $h(x) = x \bmod 10$,
Neprázdné seznamy jsou $P(1) = 1 \mapsto 141 \mapsto 11 \mapsto 161 \mapsto NIL$, $P(3) = 73 \mapsto 53 \mapsto NIL$,

$P(7) = 7 \mapsto NIL$.

řádek	key	next	previous
P(0)			
P(1)	1	9	
P(2)			
P(3)	73	6	
P(4)			
P(5)	161		8
P(6)	53		3
P(7)	7		
P(8)	11	5	9
P(9)	141	8	1

Tabulka vznikla následující posloupností operací:
INSERT(1), **INSERT**(141), **INSERT**(11), **INSERT**(73), **INSERT**(53), **INSERT**(7), **INSERT**(161).

Algoritmy

Neformální popis algoritmů: Algoritmy jsou založeny na praci s dvousměrnými spojovými seznamy. Algoritmus pro operaci **MEMBER** se neliší od algoritmu pro separované řetězce, jen místo ukazatele na další prvek použijeme hodnotu v položce next. Při popisu ideje metody jsme popsali operaci **INSERT**. Při operaci **DELETE**, když se odstraňuje první prvek řetězce, je třeba přemístit druhý prvek řetězce (pokud existuje) na jeho místo.

Formální popis algoritmů:

MEMBER(x):
$i := h(x)$
if $i.previous \neq$ prázdné nebo $i.key =$ prázdné **then**
  Výstup: $x \notin S$, stop
endif
while $i.next \neq$prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
if $i.key = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

INSERT(x):
$i := h(x)$
if $i.previous \neq$prázdné nebo $i.key =$prázdné **then**
  if $i.previous \neq$prázdné **then**
    if neexistuje volný řádek tabulky **then**
      Výstup: přeplnění, stop
    else
      nechť j je volný řádek tabulky
      $j.key := i.key$, $j.previous := i.previous$
      $j.next := i.next$, $(j.previous).next := j$

```
      (j.next).previous := j, i.next := i.previous :=prázdné
    endif
  endif
  i.key := x, stop
endif
while i.next ≠prázdné a i.key ≠ x do i := i.next enddo
if i.key ≠ x then
  if neexistuje prázdný řádek tabulky then
    Výstup: přeplnění
  else
    nechť j je volný řádek tabulky
    i.next := j, j.key := x, j.previous := i, stop
  endif
endif
```

```
  DELETE(x):
i := h(x)
if i.previous ≠prázdné nebo i.key =prázdné then stop endif
while i.next ≠prázdné a i.key ≠ x do i := i.next enddo
if i.key = x then
  if i.previous ≠prázdné then
    (i.previous).next := i.next
    if i.next ≠prázdné then
      (i.next).previous := i.previous
    endif
    i.key := i.next := i.previous :=prázdné
  else
    if i.next ≠prázdné then
      i.key := (i.next).key, i.next := (i.next).next
      if (i.next).next ≠prázdné then
        ((i.next).next).previous := i
      endif
      (i.next).key := (i.next).next := (i.next).previous :=prázdné
    else
      i.key :=prázdné
    endif
  endif
endif
```

Pro ilustraci provedeme **INSERT**(28) do předchozí tabulky. Kolidující prvek vložíme na 4. řádek tabulky. Výsledkem této operace je následující tabulka.

Očekávaný počet testů

Očekávaný počet testů je stejný jako pro hašování se separovanými řetězci, tj.:
$\frac{n-1}{2m} + 1 \approx 1 + \frac{\alpha}{2}$ pro úspěšné vyhledávání,

$(1 - \frac{1}{m})^n + \frac{n}{m} \approx e^{-\alpha} + \alpha$ pro neúspěšné vyhledávání,
kde m je velikost tabulky, n je velikost reprezentované množiny S a $\alpha = \frac{n}{m}$ je faktor naplnění.

řádek	key	next	previous
P(0)			
P(1)	1	9	
P(2)			
P(3)	73	6	
P(4)	11	5	9
P(5)	161		4
P(6)	53		3
P(7)	7		
P(8)	28		
P(9)	141	4	1

V. Hašování se dvěma ukazateli

Nevýhoda hašování s přemísťováním je v operacích **INSERT** a **DELETE**, protože přemístění prvků v tabulce je časově náročnější než provedení testu. To vedlo k hledání jiných implementací separovaných řetězců. V následující metodě implementujeme řetězec jako jednosměrný seznam, který ale nemusí začínat na svém místě. Přesněji, řetězec S_j obsahující prvky $s \in S$ takové, že $h(s) = j$, nemusí začínat na j-tém řádku. Místo ukazatele na předchozí prvek zde budeme používat položku, která udává, kde začíná řetězec příslušný danému řádku.

Položky pro práci s tabulkou tedy budou next a begin, kde
next – číslo řádku tabulky obsahujícího následující prvek řetězce,
begin – číslo řádku tabulky obsahujícího první prvek řetězce příslušného tomuto místu.

Následující hašovací tabulka pro metodu hašování se dvěma ukazateli reprezentuje stejnou množinu jako v příkladu metody hašování s přemísťováním a používá stejnou hašovací funkci.

řádek	key	next	begin
P(0)			
P(1)	1	9	1
P(2)			
P(3)	73	7	3
P(4)			
P(5)	161		
P(6)	7		
P(7)	53		6
P(8)	11	5	
P(9)	141	8	

Tabulka vznikla posloupností operací:
INSERT(1), **INSERT**(141), **INSERT**(11), **INSERT**(73), **INSERT**(53), **INSERT**(7), **INSERT**(161).

Algoritmy

Neformální popis algoritmů: Je třeba si uvědomit, že položka begin v j-tém řádku je vyplněna právě tehdy, když reprezentovaná množina S obsahuje prvek $s \in S$ takový, že $h(s) = j$. Algoritmy jsou podobné algoritmům pro hašování s přemísťováním, jen přemísťování prvků je nahrazeno odpovídajícími změnami v položce begin daných řádků.

Formální popis algoritmů:

MEMBER(x):
$i := h(x)$
if $i.begin =$ prázdné **then Výstup:** $x \notin S$**, stop else** $i := i.begin$ **endif**
while $i.next \neq$ prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
if $i.key = x$ **then Výstup:** $x \in S$ **else Výstup:** $x \notin S$ **endif**

INSERT(x)**:**
$i := h(x)$
if $i.begin =$prázdné **then**
  if $i.key =$prázdné **then**
    $i.key := x$, $i.begin := i$
  else
    if neexistuje prázdný řádek tabulky **then**
      Výstup: přeplnění
    else
      nechť j je volný řádek tabulky
      $j.key = x$, $i.begin := j$
    endif
  endif
  stop
else
  $i := i.begin$
endif
while $i.next \neq$prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
if $i.key \neq x$ **then**
  if neexistuje prázdný řádek tabulky **then**
    Výstup: přeplnění, stop
  else
    nechť j je volný řádek tabulky
    $i.next := j$, $j.key := x$
  endif,
endif

DELETE(x):
$i := h(x)$
if $i.begin$ =prázdné **then** stop **else** $j := i$, $i := i.begin$ **endif**
while $i.next \neq$prázdné a $i.key \neq x$ **do** $j := i$, $i := i.next$ **enddo**
if $i.key = x$ **then**
  if $i = j.begin$ **then**
    if $i.next \neq$prázdné **then**
      $j.begin := i.next$
    else
      $j.begin$:=prázdné
    endif
  else
    $j.next := i.next$
  endif
$i.key := i.next$:=prázdné
endif

Pro ilustraci v předchozí hašovací tabulce provedeme příkaz **INSERT**(28) a jako nový řádek zvolíme 4. řádek tabulky. Následující obrázek představuje výslednou hašovací tabulku. Operace **DELETE** pracuje velmi podobně jako v jednosměrných seznamech.

řádek	key	next	begin
P(0)			
P(1)	1	9	1
P(2)			
P(3)	73	7	3
P(4)	28		
P(5)	161		
P(6)	7		
P(7)	53		6
P(8)	11	5	4
P(9)	141	8	

Očekávaný počet testů

Algoritmus je díky práci s položkami rychlejší než hašování s přemísťováním, ale začátek řetězce v jiném místě tabulky přidává navíc jeden test. To mění složitost algoritmů. Výsledky uvedeme bez odvozování:
Očekávaný počet testů v hašování se dvěma ukazateli je
$1 + \frac{(n-1)(n-2)}{6m^2} + \frac{n-1}{2m} \approx 1 + \frac{\alpha^2}{6} + \frac{\alpha}{2}$ v úspěšném případě,
$\approx 1 + \frac{\alpha^2}{2} + \alpha + e^{-\alpha}(2+\alpha) - 2$ v neúspěšném případě.

VI. Srůstající hašování

Dále uvedeme několik verzí metody, která se nazývá srůstající hašování. Tato metoda používá jedinou položku pro práci s tabulkou, a to ukazatel jednosměrného spojového seznamu. Na rozdíl od předchozích metod nejsou řetězce separované, v jednom řetězci mohou být prvky s různými hodnotami hašovací funkce. Když máme přidat prvek s, pak ho zařadíme do řetězce, který obsahuje $h(s)$-tý řádek tabulky. Jinými slovy, řetězce v této metodě srůstají (odtud název srůstající hašování). Různé verze této metody se liší podle místa v řetězci, kam přidáváme nový prvek, a podle práce s pamětí – dělí se na standardní hašování bez pomocné paměti a na hašování používající pomocnou paměť (kuriózně tato druhá metoda nemá žádný přívlastek, nazývá se jen srůstající hašování).

Popíšeme následující metody:
standardní srůstající hašování LISCH a EISCH
a srůstající hašováníLICH, VICH a EICH.

Všechny metody pro práci s tabulkou používají jen položku next – číslo řádku tabulky obsahujícího následující prvek řetězce.

Metody LISCH a EISCH

Metody LISCH a EISCH se liší pouze místem, kam se přidává nový prvek, jak ukazuje jejich název:
LISCH – late-insertion standard coalesced hashing (přidává se za poslední prvek řetězce),
EISCH – early-insertion standard coalesced hashing (přidává se za první prvek řetězce).

Řádek tabulky má dvě položky – key a next a všechny řádky jsou přístupné pomocí hašovací funkce. Prvky $s \in S$ takové, že $h(s) = i$, jsou umístěny v řetězci od i-tého řádku, i když řetězec je delší (může začínat dřív).

Příklad: $U = \{1, 2, \ldots, 1000\}$, $S = \{1, 7, 11, 53, 73, 141, 171\}$ a $h(x) = x \bmod 10$
Následující hašovací tabulka může sloužit jako příklad pro obě uvedené metody.

řádek	key	next
P(0)		
P(1)	1	9
P(2)		
P(3)	73	6
P(4)		
P(5)	7	
P(6)	53	
P(7)	161	5
P(8)	11	7
P(9)	141	8

Pro metodu LISCH vznikla posloupností operací:
INSERT(1), **INSERT**(141), **INSERT**(11), **INSERT**(73), **INSERT**(53),

INSERT(161), **INSERT**(7).
Pro metodu EISCH tabulka vznikla posloupností operací:
INSERT(1), **INSERT**(161), **INSERT**(11), **INSERT**(73), **INSERT**(53), **INSERT**(7), **INSERT**(141).
Provedeme **INSERT**(28), nový prvek přidáváme do čvrtého řádku. Vlevo je výsledná tabulka pro metodu LISCH a vpravo pro metodu EISCH.

řádek	key	next
P(0)		
P(1)	1	9
P(2)		
P(3)	73	6
P(4)	28	
P(5)	7	4
P(6)	53	
P(7)	161	5
P(8)	11	7
P(9)	141	8

řádek	key	next
P(0)		
P(1)	1	9
P(2)		
P(3)	73	6
P(4)	28	7
P(5)	7	
P(6)	53	
P(7)	161	5
P(8)	11	4
P(9)	141	8

Algoritmy

Neformální popis algoritmů: Algoritmus pro operaci **MEMBER** je pro obě metody stejný a je prakticky stejný jako algoritmus pro předchozí metody – hašování s přemísťováním a hašování se dvěma ukazateli. Také algoritmus pro operaci **INSERT** je jednoduchý a přirozený a pro obě metody se liší teprve v okamžiku zařazení nového řádku do řetězce. V metodě LISCH se zařadí nový řádek přirozeně na konec řetězce. V metodě EISCH, když vkládáme prvek x a $h(x)$-tý řádek je prázdný, pak vložíme prvek x do tohoto řádku. Když $h(x)$-tý řádek je obsazen, pak nalezneme volný řádek, který bude reprezentovat x, a tento řádek vložíme do řetězce za $h(x)$-tý řádek (to znamená, že $h(x)$-tý řádek v položce next bude mít číslo nového řádku a položka next nového řádku bude obsahovat původní hodnotu položky next $h(x)$-tého řádku).

Formální popis algoritmů:

MEMBER(x):
$i := h(x)$
while $i.next \neq$prázdné a $i.key \neq x$ **do**
$i := i.next$
enddo
if $i.key = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

Metoda LISCH – **INSERT**(x):
$i := h(x)$
if $i.key =$ prázdné **then** $i.key := x$ **stop endif**

while $i.next \neq$ prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
if $i.key \neq x$ **then**
  if neexistuje prázdný řádek tabulky **then**
    Výstup: přeplnění
  else
    nechť j je volný řádek tabulky
    $j.key := x$, $i.next := j$
  endif
endif

Metoda EISCH – **INSERT**(x):
$k := i := h(x)$
if $i.key =$ prázdné **then** $i.key := x$ **stop endif**
while $i.next \neq$ prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
if $i.key \neq x$ **then**
  if neexistuje prázdný řádek tabulky **then**
    Výstup: přeplnění
  else
    nechť j je volný řádek tabulky
    $j.next := k.next$, $k.next := j$, $j.key := x$
  endif
endif

Přirozená efektivní operace **DELETE** pro standardní srůstající hašování není známa. Na druhou stranu i primitivní algoritmy pro operaci **DELETE** mají rozumnou očekávanou časovou složitost.
Náhodné uspořádání prvků v řetězcích (s rovnoměrným rozdělením) zajišťuje efektivní chování algoritmů pro srůstající hašovaní. Hlavní problém, který omezuje algoritmy pro operaci **DELETE**, je, aby zachovaly náhodné uspořádání prvků v řetězcích. Porušení náhodnosti může zapřičinit velké zhoršení očekávaného času operací. Této problematice je věnována celá kapitola v monografii Vittera a Chena. Uvádějí jednak algoritmus, který zachovává náhodnost za cenu přesunů prvků a částečného přehašování řetězců. Dále prezentují jednodušší algoritmus, který náhodnost nezachovává. Konečně se zabývají možností prvky z tabulky fyzicky neodstraňovat, ale pouze je označit jako smazané, s tím, že jejich pozice se mohou později využít pro vkládání nových prvků. Výhodou tohoto přístupu (s kterým se setkáme i u dalších metod) je jednoduchý algoritmus, nevýhodou je naopak to, že takto "smazané" prvky zpomalují vyhledávání.

Další otázka je, proč používat metodu EISCH, když programy pro metodu LISCH jsou jednodušší. Odpověď je na první pohled dost překvapující. Při úspěšném vyhledávání je metoda EISCH rychlejší než metoda LISCH. Zdůvodnění tohoto jevu spočívá v tom, že je pravděpodobnější práce s novým prvkem (neformálně řečeno pravděpodobnost akcí s původními prvky se zmenší a u nového prvku se stane nenulovou, to znamená výrazně naroste.) Očekávaný počet testů pro neúspěšné vyhledávání je pro obě metody stejný. Další fakta spojená s tímto jevem se studují v samoupravujících strukturách v letním semestru.

Analýza složitosti

Popíšeme analyzovanou situaci: Máme uloženou množinu $S = \{s_1, s_2, \ldots, s_n\} \subseteq U$ do tabulky velikosti m a je dán prvek s_{n+1}. Naším úkolem je zjistit, zda $s_{n+1} \in S$. Označme $a_i = h(s_i)$ pro $i = 1, 2, \ldots, n+1$, kde h je použitá hašovací funkce.
Analýzu provedeme za předpokladu, že všechny posloupnosti adres $a_1, a_2, \ldots, a_{n+1}$ jsou **stejně pravděpodobné**. Dále předpokládáme deterministický způsob výběru prázdných řádků při operaci **INSERT** nezávislý na reprezentované množině.

Neúspěšné vyhledávání

Počet testů při neúspěšném vyhledávání prvku s_{n+1} v S označíme $C(a_1, a_2, \ldots, a_n; a_{n+1})$. Na základě předpokladů platí, že očekávaný počet testů při neúspěšném vyhledávání v množině S je

$$\frac{\sum C(a_1, a_2, \ldots, a_n; a_{n+1})}{m^{n+1}},$$

kde se sčítá přes všechny posloupnosti $a_1, a_2, \ldots, a_{n+1}$, kterých je m^{n+1}.

Řetězec délky l v množině S je maximální posloupnost adres $(b_1, b_2, \ldots, b_l)$ taková, že $b_i.next = b_{i+1}$ pro $i = 1, 2, \ldots, l-1$. Když adresa a_{n+1} je i-tý prvek v řetězci, pak počet provedených testů je $l-i+1$. Řetězec délky l tedy přispívá k součtu $\sum C(a_1, a_2, \ldots, a_n; a_{n+1})$ počtem $1 + 2 + \cdots + l = l + \binom{l}{2}$ testů.

Označme $c_n(l)$ počet všech řetězců délky l ve všech reprezentacích n-prvkových množin (ztotožňujeme dvě množiny, které měly stejnou posloupnost adres při ukládání prvků). Pak

$$\begin{aligned}\sum C(a_1, a_2, \ldots, a_n; a_{n+1}) &= c_n(0) + \sum_{l=1}^{n} (l + \binom{l}{2}) c_n(l) \\ &= c_n(0) + \sum_{l=1}^{n} l c_n(l) + \sum_{l=1}^{n} \binom{l}{2} c_n(l),\end{aligned}$$

kde $c_n(0)$ je počet prázdných řádků ve všech reprezentacích.
Reprezentace S má $m - n$ prázdných řádků, posloupností všech možných adres vložených prvků je m^n, proto

$$c_n(0) = (m-n)m^n.$$

Součet délek všech řetězců ve všech tabulkách reprezentujících všechny n-prvkové množiny je $\sum_{l=1}^{n} l c_n(l)$, a proto

$$\sum_{l=1}^{n} l c_n(l) = n m^n.$$

Vypočteme $S_n = \sum_{l=1}^{n} \binom{l}{2} c_n(l)$. Nejprve odvodíme rekurentní vztah pro $c_n(l)$. Když k množině S přidáme nový prvek s adresou a_{n+1}, pak řetězec délky l v reprezentaci S se nezvětší, když adresa a_{n+1} neleží v tomto řetězci, v opačném případě se délka řetězce zvětší na $l+1$. Proto přidání jednoho prvku vytvoří z řetězce délky l celkem $m - l$ řetězců délky l a l řetězců délky $l+1$. Vysčítáním přes všechny n-prvkové posloupnosti adres dostaneme

$$c_{n+1}(l) = (m-l)c_n(l) + (l-1)c_n(l-1).$$

Odtud

$$\begin{aligned}
S_n =& \sum_{l=1}^{n} \binom{l}{2} c_n(l) = \sum_{l=1}^{n} \big(\binom{l}{2}(m-l)c_{n-1}(l) + \binom{l}{2}(l-1)c_{n-1}(l-1) \big) = \\
& \big(\sum_{l=1}^{n} \binom{l}{2}(m-l)c_{n-1}(l) \big) + \big(\sum_{l=0}^{n-1} \binom{l+1}{2} l c_{n-1}(l) \big) = \binom{n}{2}(m-n)c_{n-1}(n) + \\
& \big(\sum_{l=1}^{n-1} (\binom{l}{2}(m-l) + \binom{l+1}{2} l) c_{n-1}(l) \big) + \binom{1}{2} 0 c_{n-1}(0) = \\
& \sum_{l=1}^{n-1} \binom{l}{2}(m+2)c_{n-1}(l) + \sum_{l=1}^{n-1} l c_{n-1}(l) \\
& = (m+2)S_{n-1} + (n-1)m^{n-1},
\end{aligned}$$

kde jsme použili, že $c_{n-1}(n) = 0$, a identitu

$$\begin{aligned}
(m-l)\binom{l}{2} + l\binom{l+1}{2} =& \frac{1}{2}(l^2 m - lm - l^3 + l^2 + l^3 + l^2) = \\
& \frac{1}{2}(l^2 m - lm + 2l^2) = \frac{1}{2}(l^2 m - lm + 2(l^2 - l)) + l = \\
& (m+2)\binom{l}{2} + l.
\end{aligned}$$

Rekurence pro S_n dává

$$\begin{aligned}
S_n =& (m+2)S_{n-1} + (n-1)m^{n-1} = \\
& (m+2)^2 S_{n-2} + (m+2)(n-2)m^{n-2} + (n-1)m^{n-1} = \\
& (m+2)^3 S_{n-3} + (m+2)^2(n-3)m^{n-3} + (m+2)(n-2)m^{n-2} + (n-1)m^{n-1} \\
& = (m+2)^{n-1} S_0 + \sum_{i=0}^{n-1} (m+2)^i (n-1-i) m^{n-1-i} = \\
& (m+2)^{n-1} \sum_{i=0}^{n-1} (n-1-i) \big(\frac{m}{m+2} \big)^{n-1-i} \\
& = (m+2)^{n-1} \sum_{i=1}^{n-1} i \big(\frac{m}{m+2} \big)^{i},
\end{aligned}$$

kde jsme využili, že $S_0 = 0$.
Nyní provedeme pomocný výpočet. Spočítáme součet $T_c^n = \sum_{i=1}^{n} i c^i$ pro $n = 1, 2, \ldots$ a

$c \neq 1$. Z $cT_c^n = \sum_{i=1}^{n} ic^{i+1}$ plyne

$$\begin{aligned}(c-1)T_c^n =& cT_c^n - T_c^n = \sum_{i=2}^{n+1}(i-1)c^i - \sum_{i=1}^{n} ic^i = nc^{n+1} + \big(\sum_{i=2}^{n}((i-1)c^i - ic^i)\big) - c = \\ & nc^{n+1} + \big(\sum_{i=2}^{n} -c^i\big) - c = \\ & nc^{n+1} - \sum_{i=1}^{n} c^i = nc^{n+1} - \frac{c^{n+1}-c}{c-1} = \frac{nc^{n+2} - (n+1)c^{n+1} + c}{c-1}.\end{aligned}$$

Tedy platí

$$T_c^n = \frac{nc^{n+2} - (n+1)c^{n+1} + c}{(c-1)^2}.$$

Protože $\frac{m}{m+2} \neq 1$, dostáváme, že

$$\begin{aligned}S_n =& (m+2)^{n-1}\frac{(n-1)\big(\frac{m}{m+2}\big)^{n+1} - n\big(\frac{m}{m+2}\big)^n + \frac{m}{m+2}}{\big(\frac{m}{m+2}-1\big)^2} = \\ & \frac{1}{4}(m+2)^{n+1}\big[(n-1)\big(\frac{m}{m+2}\big)^{n+1} - n\big(\frac{m}{m+2}\big)^n + \frac{m}{m+2}\big] = \\ & \frac{1}{4}\big[(n-1)m^{n+1} - n(m+2)m^n + m(m+2)^n\big] = \frac{1}{4}\big(m(m+2)^n - m^{n+1} - 2nm^n\big).\end{aligned}$$

Očekávaný počet testů při neúspěšném vyhledávání je

$$\begin{aligned}& \frac{(m-n)m^n + nm^n + \frac{1}{4}\big(m(m+2)^n - m^{n+1} - 2nm^n\big)}{m^{n+1}} = \\ & \frac{m^{n+1} + \frac{1}{4}\big(m(m+2)^n - m^{n+1} - 2nm^n\big)}{m^{n+1}} = \\ & 1 + \frac{1}{4}\big((1+\frac{2}{m})^n - 1 - \frac{2n}{m}\big) \sim 1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha).\end{aligned}$$

Tento odhad je stejný pro obě metody – LISCH i EISCH, protože mají stejné posloupnosti adres (liší se jen pořadím prvků v jednotlivých řetězcích).

Úspěšný případ

Očekávaný počet testů při úspěšném vyhledávání v modelu LISCH vypočteme stejnou metodou jako pro hašování se separovanými řetězci. Všimněme si, že tam uvedený předpoklad je pro metodu LISCH splněn (metoda EISCH ho nesplňuje). Pro vyhledání prvku $s_{n+1} \in S$ je počet testů roven 1+počet porovnání klíčů při operaci **INSERT**(s_{n+1})**.** Když s_{n+1} je vložen na místo $h(s_{n+1})$, nebyl porovnáván žádný klíč a test bude jeden. Když $h(s_{n+1})$ je na na i-tém místě v řetězci délky l, pak bylo při operaci **INSERT**(s_{n+1}) použito $l - i + 1$

porovnání klíčů a testů tedy bude $l-i+2$. Očekávaný počet porovnání klíčů při neúspěšném vyhledávání je

$$\frac{1}{m^{n+1}}(\sum_{l=1}^{n}(l+\binom{l}{2})c_n(l)) = \frac{1}{m^{n+1}}(nm^n + \frac{1}{4}\big(m(m+2)^n - m^{n+1} - 2nm^n\big)) =$$

$$\frac{1}{4}\big((1+\frac{2}{m})^n - 1 + \frac{2n}{m}\big).$$

Tedy očekávaný počet testů při úspěšném vyhledávání v n-prvkové množině je podle předchozí analýzy roven $1+$ n-tina součtu očekávaného počtu porovnání klíčů při neúspěšném vyhledávání v i-prvkové množině, kde i probíhá čísla $0, 1, \ldots, n-1$. Podle předchozích výsledků je hledaný součet

$$\sum_{i=0}^{n-1}\frac{1}{4}\big[(1+\frac{2}{m})^i - 1 + \frac{2i}{m}\big] = \frac{1}{4}\frac{(1+\frac{2}{m})^n - 1}{1+\frac{2}{m}-1} - \frac{n}{4} + \frac{\binom{n}{2}}{2m} =$$

$$\frac{m}{8}\big((1+\frac{2}{m})^n - 1 - \frac{2n}{m}\big) + \frac{n^2-n}{4m}$$

a očekávaný počet testů v úspěšném případě pro n-prvkovou množinu je

$$1 + \frac{m}{8n}\big((1+\frac{2}{m})^n - 1 - \frac{2n}{m}\big) + \frac{n-1}{4m} \sim 1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}.$$

Pro metodu EISCH je očekáváný počet testů v úspěšném případě

$$\frac{m}{n}\big((1+\frac{1}{m})^n - 1\big) \sim \frac{1}{\alpha}(e^{\alpha} - 1),$$

k odvození se ale musí použít složitější metoda. Chyba aproximace pro všechny tyto odhady je $O(\frac{1}{m})$.

Shrneme získané výsledky:

Věta. *Za předpokladu rovnoměrného rozdělenídat platí:*

(1) *v metodě LISCH je očekávaný počet testů při neúspěšném vyhledávání v n-prvkové množině*

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

a při úspěšném vyhledávání v n-prvkové množině je

$$1 + \frac{m}{8n}((1+\frac{2}{m})^n - 1 - \frac{2n}{m}) + \frac{n-1}{4m} \approx 1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4},$$

(2) *v metodě EISCH je očekávaný počet testů při neúspěšném vyhledávání v n-prvkové množině*

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

a při úspěšném vyhledávání v n-prvkové množině je

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

Chyba aproximace pro tyto odhady je $O(\frac{1}{m})$.

Metody LICH, EICH, VICH

Nyní se budeme zabývat srůstajícím hašováním s pomocnou pamětí. V této metodě je používaná paměť rozdělena na dvě části, na část přímo adresovatelnou a na část pomocnou. V paměti je uložena tabulka tak, že v adresovací části jsou řádky, které jsou přístupné pomocí hašovací funkce (má m řádků, když hašovací funkce má hodnoty z oboru $\{0, 1, \ldots, m-1\}$), v pomocné části jsou řádky, ke kterým nemáme přístup přes hašovací funkci. Když při přidávání nového prvku vznikne kolize, tak se nejprve vybere volný řádek z pomocné části, a teprve, když jsou řádky z pomocné části zaplněny, použijí se k ukládání kolidujících prvků řádky z adresovací části. Tato strategie oddaluje srůstání řetězců. Proto chování srůstajícího hašování, dokud není naplněna pomocná část, se podobá chování hašování se separovanými řetězci. Budeme prezentovat tři varianty, které se liší místem v řetězci, kam se přidává nový prvek. Jsou to:
LICH – late-insertion coalesced hashing,
EICH – early-insertion coalesced hashing,
VICH – varied-insertion coalesced hashing.

Při úspěšné operaci **INSERT** metoda LICH vkládá nový prvek vždy na konec řetězce, metoda EICH vkládá nový prvek x buď do řádku $h(x)$, pokud byl prázdný, nebo do řetězce vždy hned za prvek na řádku $h(x)$, a metoda VICH vkládá nový prvek x za poslední prvek řetězce, který je ještě v pomocné části, a když řetězec neobsahuje žádný řádek z pomocné části, je nový prvek vložen do řetězce za prvek na řádku $h(x)$. Dá se říct, že VICH je kombinací obou předchozích metod – v pomocné části se chová jako LICH, v adresové části jako EICH.

Tyto metody nepodporují přirozené efektivní algoritmy pro operaci **DELETE**.

Příklad: $U = \{1, 2, \ldots, 1000\}$, $S = \{1, 7, 11, 53, 73, 141, 161\}$ a $h(x) = x \bmod 10$.
Hašovací tabulka má 12 řádků a má tvar

řádek	key	next
P(0)		
P(1)	1	10
P(2)		
P(3)	73	11
P(4)		
P(5)	7	
P(6)		
P(7)	161	5
P(8)	11	7
P(9)		
P(10)	141	8
P(11)	53	

Pro metodu LICH vznikla posloupností operací:
INSERT(1), **INSERT**(73), **INSERT**(141), **INSERT**(53), **INSERT**(11), **INSERT**(161), **INSERT**(7).

Pro metodu EICH vznikla posloupností operací:
INSERT(1), **INSERT**(73), **INSERT**(161), **INSERT**(53), **INSERT**(11), **INSERT**(141), **INSERT**(7).
(Zde však nebyl dodržen požadavek, že se nejprve zaplňují řádky z pomocné části. Při dodržení tohoto pravidla by tato tabulka metodou EICH nemohla vzniknout.)

Pro metodu VICH vznikla posloupností operací:
INSERT(1), **INSERT**(73), **INSERT**(141), **INSERT**(53), **INSERT**(161), **INSERT**(11), **INSERT**(7).

Aplikujme operace **INSERT**(28) a **INSERT**(31). Řádky, kam budeme přidávat nové prvky, budou 4. a 9. řádek tabulky. Tabulka vytvořená pomocí metody LICH je vlevo, pomocí metody VICH je uprostřed a pomocí metody EICH je vpravo

řádek	key	next
P(0)		
P(1)	1	10
P(2)		
P(3)	73	11
P(4)	28	9
P(5)	7	4
P(6)		
P(7)	161	5
P(8)	11	7
P(9)	31	
P(10)	141	8
P(11)	53	

řádek	key	next
P(0)		
P(1)	1	10
P(2)		
P(3)	73	11
P(4)	28	7
P(5)	7	
P(6)		
P(7)	161	5
P(8)	11	4
P(9)	31	8
P(10)	141	9
P(11)	53	

řádek	key	next
P(0)		
P(1)	1	9
P(2)		
P(3)	73	11
P(4)	28	7
P(5)	7	
P(6)		
P(7)	161	5
P(8)	11	4
P(9)	31	10
P(10)	141	8
P(11)	53	

Všimněme si, že strategie EICH je rozporuplná, vkládání prvku na druhé místo řetězce jde proti strategii pomocné paměti. Jak však uvidíme, tento rozpor se neprojevuje, protože řetězce jsou krátké.

Algoritmy

Neformální popis algoritmů: Algoritmus operace **MEMBER** je pro tyto metody stejný jako pro metody LISCH a EISCH. Algoritmus operace **INSERT** je pro metody LICH a EICH stejný jako pro metody LISCH a EISCH s jediným doplňkem, a to, že pokud existuje prázdný řádek v pomocné části, pak nový řádek je vybrán z pomocné části. Tento předpoklad platí i v algoritmu **INSERT** pro metodu VICH. V této metodě je však jiná strategie při zařazování nového řádku do řetězce. Pokud prvek x vkládáme na $h(x)$-tý řádek nebo pokud je nový řádek z pomocné části, tak postupujeme stejně jako v metodě LICH. Jiná situace je, když nový řádek je z adresovací části, ale je různý od $h(x)$-tého řádku, pak ho vložíme do řetězce před první řádek z adresovací části, který následuje za $h(x)$-tým řádkem. To znamená, že za $h(x)$-tým řádkem přeskočíme všechny řádky z pomocné části.

Formální popis algoritmů:

MEMBER(x):
$i := h(x)$
while $i.next \neq$prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
if $i.key = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

Metoda LICH – **INSERT**(x):
$i := h(x)$
if $i.key =$ prázdné **then** $i.key := x$**, stop endif**
while $i.next \neq$ prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
if $i.key \neq x$ **then**
  if neexistuje prázdný řádek tabulky **then**
    Výstup: přeplnění, stop
  else
    nechť j je volný řádek tabulky
    $j.key := x$, $i.next := j$
  endif
endif

Metoda EICH – **INSERT**(x):
$k := i := h(x)$
if $i.key =$ prázdné **then** $i.key := x$**, stop endif**
while $i.next \neq$ prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
if $i.key \neq x$ **then**
  if neexistuje prázdný řádek tabulky **then**
    Výstup: přeplnění, stop
  else
    nechť j je volný řádek tabulky
    $j.next := k.next$, $k.next := j$, $j.key := x$
  endif
endif

Metoda VICH – **INSERT**(x):
$i := h(x)$, $k := 0$
if $i.key =$ prázdné **then** $i.key := x$**, stop endif**
while $i.next \neq$ prázdné a $i.key \neq x$ **do**
  if ($i \geq m$ nebo $i = h(x)$) a $i.next < m$ **then** $k := i$ **endif**
  $i := i.next$
enddo
Komentář: $k = 0$, když nenastane kolize nebo řetězec zůstává v pomocné části tabulky, jinak k je číslo řádku posledního prvku v řetězci, který je ještě v pomocné části, a pokud řetězec neobsahuje žádný řádek v pomocné části, pak je to hodnota $h(x)$.
if $i.key \neq x$ **then**
  if neexistuje prázdný řádek **then**
    Výstup: přeplnění
  else
    nechť j je volný řádek tabulky

if $j \geq m$ **then**
Komentář: Tedy j je v pomocné části
$i.next := j$
else
$j.next := k.next$, $k.next := j$
endif
$j.key := x$
endif
endif

Mechanismus výběru volných řádků v těchto algoritmech sám zařizuje, že pokud pomocná část tabulky obsahuje volný řádek, pak j-tý řádek je vybrán z této pomocné části paměti (např. se vybírá vždy volný řádek s nejvyšším pořadovým číslem).

Složitost algoritmů

V následující větě používáme toto značení:
n – velikost uložené množiny,
m – velikost adresovací části tabulky,
m' – velikost celé tabulky,
$\alpha = \frac{n}{m'}$ – faktor zaplnění,
$\beta = \frac{m}{m'}$ – adresovací faktor,
λ – jediné nezáporné řešení rovnice $e^{-\lambda} + \lambda = \frac{1}{\beta}$.

Věta. *Očekávaný počet testů pro metodu LICH*
neúspěšný případ: $e^{-\frac{\alpha}{\beta}} + \frac{\alpha}{\beta}$, *když* $\alpha \leq \lambda\beta$,
$\frac{1}{\beta} + \frac{1}{4}(e^{2(\frac{\alpha}{\beta}-\lambda)} - 1)(3 - \frac{2}{\beta} + 2\lambda) - \frac{1}{2}(\frac{\alpha}{\beta} - \lambda)$, *když* $\alpha \geq \lambda\beta$
úspěšný případ: $1 + \frac{\alpha}{2\beta}$, *když* $\alpha \leq \lambda\beta$,
$1 + \frac{\beta}{8\alpha}(e^{2(\frac{\alpha}{\beta}-\lambda)} - 1 - 2(\frac{\alpha}{\beta} - \lambda))(3 - \frac{2}{\beta} + 2\lambda) + \frac{1}{4}(\frac{\alpha}{\beta} + \lambda) + \frac{\lambda}{4}(1 - \frac{\lambda\beta}{\alpha})$, *když* $\alpha \geq \lambda\beta$.

Očekávaný počet testů pro metodu EICH
neúspěšný případ: $e^{-\frac{\alpha}{\beta}} + \frac{\alpha}{\beta}$, *když* $\alpha \leq \lambda\beta$,
$e^{2(\frac{\alpha}{\beta}-\lambda)}(\frac{3}{4} + \frac{\lambda}{2} - \frac{1}{2\beta}) + e^{\frac{\alpha}{\beta}-\lambda}(\frac{1}{\beta} - 1) + (\frac{1}{4} - \frac{\alpha}{2\beta} + \frac{1}{2\beta})$, *když* $\alpha \geq \lambda\beta$
úspěšný případ: $1 + \frac{\alpha}{2\beta}$, *když* $\alpha \leq \lambda\beta$,
$1 + \frac{\alpha}{2\beta} + \frac{\beta}{\alpha}((e^{\frac{\alpha}{\beta}-\lambda} - 1)(1 + \lambda) - (\frac{\alpha}{\beta} - \lambda))(1 + \frac{\lambda}{2} + \frac{\alpha}{2\beta}))$, *když* $\alpha \geq \lambda\beta$.

Očekávaný počet testů pro metodu VICH
neúspěšný případ: $e^{-\frac{\alpha}{\beta}} + \frac{\alpha}{\beta}$, *když* $\alpha \leq \lambda\beta$,
$\frac{1}{\beta} + \frac{1}{4}(e^{2(\frac{\alpha}{\beta}-\lambda)} - 1)(3 - \frac{2}{\beta} + 2\lambda) - \frac{1}{2}(\frac{\alpha}{\beta} - \lambda)$, *když* $\alpha \geq \lambda\beta$
úspěšný případ: $1 + \frac{\alpha}{2\beta}$, *když* $\alpha \leq \lambda\beta$,
$1 + \frac{\alpha}{2\beta} + \frac{\beta}{\alpha}((e^{\frac{\alpha}{\beta}-\lambda} - 1)(1 + \lambda) - (\frac{\alpha}{\beta} - \lambda))(1 + \frac{\lambda}{2} + \frac{\alpha}{2\beta})) + \frac{1-\beta}{\alpha}(\frac{\alpha}{\beta} - \lambda - e^{\frac{\alpha}{\beta}-\lambda} + 1)$, *když* $\alpha \geq \lambda\beta$.

Chyba aproximace pro všechny tyto odhady je $O(\log \frac{m'}{\sqrt{m'}})$.

Všimněme si, že každý výraz má dvě části. První část, když $\alpha \leq \lambda\beta$, odpovídá tomu, že není zaplněna pomocná část paměti, a analýza a i její výsledky jsou blízké analýze hašování

se separovanými řetězci. Druhá část odpovídá situaci, kdy už začínají řetězce srůstat, a její analýza je velmi složitá (jak dokumentují výsledky). Proto ji vynecháváme.

VII. Hašování s lineárním přidáváním

Nyní popíšeme dvě metody, které nepoužívají žádné položky pro práci s tabulkou. To znamená, že způsob nalezení dalšího řádku řetězce je zabudován přímo do metody a určuje hledání volných řádků. První a nejjednodušší metoda je hašování s lineárním přidáváním. Idea je, že když máme přidávat prvek x operací **INSERT**(x) a vznikne kolize, pak nalezneme první následující volný řádek a tam prvek x uložíme. Předpokládáme, že řádky jsou číslovány modulo m, tj. tvoří cyklus délky m.

Řádek tabulky má tedy jedinou položku – key.

Tato metoda vyžaduje minimální velikost paměti. V tabulce se však vytvářejí shluky obsazených řádků, a proto při velkém zaplnění vyžaduje velké množství času (musí se projít všechny řádky ve shluku). Metoda nepodporuje efektivní implementaci operace **DELETE**. Zde neplatí podobná poznámka jako pro srůstající hašování, metody jsou skutečně neefektivní, jejich provedení odpovídá velikosti shluků.
Pro zjištění přeplnění je vhodné mít uložen počet vyplněných řádků a nebo při vyhledávání testovat, zda nevyšetřujeme podruhé první vyšetřovaný řádek. Jiná možnost je, že v tabulce je vždy alespoň jeden prázdný řádek, který slouží při vyhledáváníjako zarážka.

Algoritmy

Neformální popis algoritmů je uveden při popisu metody.

Formální popis algoritmů:

MEMBER(x):
$i := h(x)$, $h := i$
if $i.key = x$ **then Výstup**: $x \in S$, stop **endif**
if $i.key =$ prázdné **then Výstup**: $x \notin S$, stop **endif**
$i := i + 1$
while $i.key \neq$ prázdné a $i.key \neq x$ a $i \neq h$ **do**
$i := i + 1 \bmod m$
enddo
if $i.key = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

INSERT(x):
$i := h(x)$, $j := 0$
while $i.key \neq$ prázdné a $i.key \neq x$ a $j < m$ **do**
$i := i + 1 \bmod m$, $j := j + 1$
enddo,
if $j = m$ **then Výstup**: přeplnění, stop **endif**
if $i.key =$prázdný **then**

$i.key := x$
endif

Příklad: $U = \{1, 2, \ldots, 1000\}$, $S = \{1, 7, 11, 53, 73, 141, 161\}$ a $h(x) = x \bmod 10$.
Množina je uložena v tabulce vlevo. Tabulka vznikla posloupností operací:
INSERT(1), **INSERT**(11), **INSERT**(73), **INSERT**(141), **INSERT**(161),
INSERT(53), **INSERT**(7). Tabulka vpravo vznikne z tabulky vlevo provedením operace **INSERT**(35).

řádek	key
P(0)	
P(1)	1
P(2)	11
P(3)	73
P(4)	141
P(5)	161
P(6)	53
P(7)	7
P(8)	
P(9)	

řádek	key
P(0)	
P(1)	1
P(2)	11
P(3)	73
P(4)	141
P(5)	161
P(6)	53
P(7)	7
P(8)	35
P(9)	

Očekávaný počet testů

Na závěr uvedeme složitost této metody.

Věta. *Očekávaný počet testů v metodě hašování s lineárním přidáváním je v neúspěšném případě přibližně* $\frac{1}{2}(1 + \left(\frac{1}{1-\alpha}\right)^2)$ *a v úspěšném případě přibližně* $\frac{1}{2}(1 + \frac{1}{1-\alpha})$.

Analýza viz D. E. Knuth: The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison Wesley, 1973 (je částí letní přednášky).

VIII. Dvojité hašování

Základní nevýhodou předchozí metody je způsob výběru volného řádku. Je stejný pro všechny řádky a důsledkem toho je velmi brzký vznik shluků řádků, který vede brzy k výraznému zpomalení algoritmů. Následující metoda se snaží tento nedostatek odstranit tím, že výběr dalších řádků je závislý na vkládaném prvku (ideálem je, aby pro různé prvky při kolizi byly posloupnosti adres, na nichž se hledá volný řádek, různé) a je rovnoměrně rozprostřen po celé hašovací tabulce. Toho dosáhne použitím dvou hašovacích funkcí h_1 a h_2. Při operaci **INSERT**(x), když x není ještě uložen v tabulce, se nalezne nejmenší $i = 0, 1, \ldots$ takové, že $(h_1(x) + ih_2(x)) \bmod m$ je prázdný řádek, a tam se prvek x vloží.
Požadavek na korektnost je, že pro každé x musí být $h_2(x)$ a m nesoudělné (jinak prvek x nemůže být uložen na libovolném řádku tabulky).
Realizace ideje této metody je založena na předpokladu: pro každé $x \in U$ je posloupnost $\{h_1(x)+ih_2(x)\}_{i=0}^{m-1}$ náhodná permutace množiny řádků tabulky. To sice v praxi nelze úplně přesně realizovat, ale v každém případě je alespoň nutné, aby z $h_1(x) = h_1(y)$ plynulo, že

posloupnosti $\{h_1(x) + ih_2(x)\}_{i=0}^{m-1}$ a $\{h_1(y) + ih_2(y)\}_{i=0}^{m-1}$ budou dosti odlišné.
Nevýhodou metody je fakt, že nepodporuje operaci **DELETE**. Důvody jsou analogické jako v hašování s lineárním přidáváním.
Přeplnění se řeší stejným způsobem jako v hašování s lineárním přidáváním.

Záludnost metody dvojitého hašování nejlépe ukazuje fakt, že hašování s lineárním přidáváním je speciálním případem dvojitého hašování, kde funkce h_2 je konstantní s hodnotou 1. To jen ukazuje na nutnost prověřování předpokladů. Následující analýza ukazuje, že dvojité hašování je za určitých předpokladů podstatně výhodnější než hašování s lineárním přidáváním. Vychází přitom z teoretického předpokladu, že pro každé $x \in U$ je posloupnost $\{h_1(x) + ih_2(x)\}_{i=0}^{m-1}$ náhodnou permutací řádků tabulky. Experimenty ukazují, že i když teoretický předpoklad nelze splnit, tak při šikovné volbě funkce h_2 (viz předchozí poznámky) se praktické výsledky blíží výsledkům teoretické analýzy. Problém je však v zadání funkce h_2 a v času potřebném pro výpočet hodnoty $h_2(x)$.

Řádek tabulky má jedinou položku – key.

Algoritmy

Neformální popis algoritmů: Algoritmy jsou prakticky stejné jako pro hašování s lineárním přidáváním. Jediná změna je při hledání dalšího řádku, v hašování s lineárním přidáváním jdeme na následující v rádek a v této metodě jdeme na řádek s číslem o $h_2(x)$ větším (pochopitelně modulo m). Pochopitelně hodnota $h_2(x)$ se počítá jen jednou, po spočítání se uloží a přičítá se uložené číslo.

Formální popis algoritmů:

MEMBER(x):
$i := h_1(x)$, $h := h_2(x)$
$j := 0$
while $i.key \neq$prázdný a $i.key \neq x$ a $j < m$ **do**
  $i := i + h \bmod m$, $j := j + 1$
enddo
if $i.key = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

INSERT(x):
$i := h_1(x)$, $h := h_2(x)$
$j := 0$
while $i.key \neq$prázdný a $i.key \neq x$ a $j < m$ **do**
  $i := i + h \bmod m$, $j := j + 1$
enddo
if $j = m$ **then Výstup**: přeplnění, stop **endif**
if $i.key =$prázdný **then** $i.key := x$ **endif**

Příklad: $U = \{1, 2, \ldots, 1000\}$, $S = \{1, 7, 11, 53, 73, 141, 161\}$. Hašovací funkce jsou $h_1(x) = x \bmod 10$ a $h_2(x) = 1 + 2(x \bmod 4)$, když $x \bmod 4 \in \{0, 1\}$, $h_2(x) = 3 + 2(x \bmod 4)$, když $x \bmod 4 \in \{2, 3\}$. Množina je uložena v tabulce vlevo, která vznikla posloupností operací:

INSERRT(1), **INSERT**(73), **INSERT**(53), **INSERT**(141), **INSERT**(161), **INSERT**(11), **INSERT**(7). Aplikujme operaci **INSERT**(35). Pak $h_2(35) = 9$, tedy posloupnost možných adres pro uložení $x = 35$ je

$$(5, 4, 3, 2, 1, 0, 9, 8, 7, 6).$$

Výsledek je uložen v tabulce vpravo.

řádek	key
P(0)	11
P(1)	1
P(2)	
P(3)	73
P(4)	141
P(5)	7
P(6)	53
P(7)	161
P(8)	
P(9)	

řádek	key
P(0)	11
P(1)	1
P(2)	35
P(3)	73
P(4)	141
P(5)	7
P(6)	53
P(7)	161
P(8)	
P(9)	

Analýza

Nejprve provedeme analýzu vyhledávání v neúspěšném případě. Označme $q_i(n, m)$ pravděpodobnost toho, že když tabulka má m řádků, z nichž n je obsazeno, pak řádky $h_1(x)+jh_2(x)$ pro $j = 0, 1, \ldots, i-1$ jsou obsazené. Pak $q_0(n, m) = 1$ a na základě předpokladu platí $q_1(n, m) = \frac{n}{m}$, $q_2(n, m) = \frac{n(n-1)}{m(m-1)}$ a obecně

$$q_i(n, m) = \frac{\prod_{j=0}^{i-1}(n-j)}{\prod_{j=0}^{i-1}(m-j)}.$$

Odtud dostáváme rekurentní vztah:
$q_j(n, m) = \frac{n}{m} q_{j-1}(n-1, m-1)$ pro všechna $j, n > 0$ a $m > 1$.
Dále pro tabulku, která má m řádků a n z nich je obsazeno, označme $C(n, m)$ očekávaný počet testů při neúspěšném vyhledávání. Podle definice platí

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

Zřejmě pro každé m platí $C(0, m) = 1$ a dosazením rekurentního vztahu pro pravděpodobnosti dostaneme

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

Indukcí ukážeme, že $C(n,m) = \frac{m+1}{m-n+1}$. Když $n = 0$, pak $C(0,m) = \frac{m+1}{m-0+1} = 1$ a tvrzení platí. Předpokládejme, že platí pro $n-1 \geq 0$ a pro každé $m \geq n-1$, a dokažme, že pak platí pro n. Platí

$$C(n,m) = 1 + \frac{n}{m}C(n-1,m-1) = 1 + \frac{n((m-1)+1)}{m((m-1)-(n-1)+1)} = 1 + \frac{n}{m-n+1} = \frac{m+1}{m-n+1}.$$

Očekávaný počet testů při neúspěšném vyhledávání v tabulce s m řádky, z nichž n je obsazeno, je tedy $\frac{m+1}{m-n+1}$.

Nyní vypočteme očekávaný počet testů v úspěšném případě. Použijeme metodu z analýzy separovaných řetězců. Počet testů při vyhledávání $x \in S$ je stejný, jako byl počet testů při vkládání x do tabulky. Tedy očekávaný počet testů při úspěšném vyhledávání v tabulce s m řádky, z nichž n je obsazeno, je

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

Následující tabulka ukazuje tyto hodnoty v závislosti na velikosti α.

hodnota α	0.5	0.7	0.9	0.95	0.99	0.999
$\frac{1}{1-\alpha}$	2	3.3	10	20	100	1000
$\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$	1.38	1.70	2.55	3.15	4.65	6.9

IX. Shrnutí

Porovnání efektivity

Uvedeme pořadí metod hašování podle očekávaného počtu testů.

Neúspěšné vyhledávání:
Hašování s uspořádanými separovanými řetězci,
Hašovaní se separovanými řetězci=Hašování s přemísťováním,
Hašování se dvěma ukazateli,
VICH=LICH,
EICH,
LISCH=EISCH,
Dvojité hašování,
Hašování s lineárním přidáváním.

Úspěšné vyhledávání:
Hašování s uspořádanými sep. řetězci=Hašování se sep. řetězci=Hašování s přemísťováním,
Hašování se dvěma ukazateli,
VICH,
LICH,
EICH,
EISCH,
LISCH,
Dvojité hašování,
Hašování s lineárním přidáváním.

Poznámka: Metoda VICH při neúspěšném vyhledávání pro $\alpha < 0.72$ a při úspěšném vyhledávání pro $\alpha < 0.92$ vyžaduje menší očekávaný počet testů než metoda s dvěma ukazateli. Při neúspěšném vyhledávání má VICH a LICH stejnou složitost a je o 8% lepší než EICH a o 15% než LISCH a EISCH, které mají stejnou složitost. Při úspěšném vyhledávání je VICH nepatrně lepší než LICH a EICH, o 3% lepší než EISCH a o 7% lepší než LISCH.

Očekávaný počet testů při zcela zaplněné tabulce:
Metoda s přemísťováním: neúspěšné vyhledávání 1.5, úspěšné vyhledávání 1.4.
Metoda se dvěma ukazateli: úspěšné i neúspěšné vyhledávání 1.6.
VICH: neúspěšné vyhledávání 1.79, úspěšné vyhledávání 1.67,
LICH: neúspěšné vyhledávání 1.79, úspěšné vyhledávání 1.69,
EICH: neúspěšné vyhledávání 1.93, úspěšné vyhledávání 1.69,
EISCH: neúspěšné vyhledávání 2.1, úspěšné vyhledávání 1.72,
LISCH: neúspěšné vyhledávání 2.1, úspěšné vyhledávání 1.8.

Hašování s lineárním přidáváním je dobré použít jen pro $\alpha < 0.7$, dvojité hašování pro $\alpha < 0.9$. Pak již čas pro neúspěšné vyhledávání velmi rychle narůstá.

Vliv $\beta = \frac{m}{m'}$ při srůstajícím hašování.
Při úspěšném vyhledávání je optimální hodnota $\beta = 0.85$, při neúspěšném vyhledávání je optimální hodnota $\beta = 0.78$. V praxi se doporučuje použít hodnotu $\beta = 0.86$ (výše uvedené výsledky byly počítány právě pro tuto hodnotu β).

Komentář: Metody se separovanými řetězci a srůstající hašování používají více paměti (při srůstajícím hašování součet adresovací a pomocné části tabulky). Metoda s přemísťováním a metoda dvojitého hašování vyžadují více času – na přemístění prvku a na výpočet druhé hašovací funkce. Všechna tato fakta je třeba vzít v úvahu při rozhodování, kterou metodu je vhodné použít. Např. při velkém zaplnění tabulky se může ukázat výhodnější a i rychlejší použít standardní srůstající hašování než srůstající hašování s pomocnou pamětí. Větší paměťové nároky mohou zapříčinit použití pomalejší oblasti paměti a tím se metoda stane nevýhodnou.

Další otázky

Jak nalézt volný řádek?
Za nejlepší metodu se považuje mít seznam (zásobník) volných řádků, při operaci **INSERT** z jeho kraje (vrcholu) vzít volný řádek a po úspěšné operaci **DELETE** tam zase řádek vložit (pozor při operaci **DELETE** ve strukturách, které nepodporují **DELETE**).

Jak řešit přeplnění?
Standardní model: Při základní velikosti tabulky m se pracuje v závislosti na počtu vložených prvků s tabulkami s $2^i m$ řádky pro vhodné $i = 0, 1, \ldots$. Vhodné i znamená, že faktor naplnění α je v intervalu $< \frac{1}{4}, 1 >$ (s výjimkou $i = 0$, kde se uvažuje pouze horní mez). Při překročení meze se zvětší nebo zmenší i a všechna data se přehašují do nové tabulky.
Po přehašování do nové tabulky je počet operací, které vedou k novému přehašovávání, vždy roven alespoň polovině velikosti uložené množiny.
V praxi se nedoporučuje striktně se držet mezí. Při přeplnění je např. vhodné používat pro nové prvky malé pomocné tabulky a posunout velké přehašování na dobu klidu (aby systém nenechal uživatele v době normálního provozu čekat).

Jak řešit operaci **DELETE** v metodách, které **DELETE** nepodporují?
Jednou z možností je použití tzv. 'falešného **DELETE**'. Tato idea navrhuje odstranit prvek, ale řádek neuvolnit (v klíči nechat nějakou hodnotu, která bude znamenat, že řádek je prázdný, a položky podporující práci s tabulkami neměnit). Řádek nebude v seznamu volných řádků, ale operace **INSERT**, když tento řádek testuje, tam může vložit nový prvek. Když je alespoň polovina použitých řádků takto blokována, je vhodné celou strukturu přehašovat. Pravděpodobnostní analýzu tohoto modelu neznáme. Pro srůstající hašování není potřeba tento postup použít (v takto přímočaré podobě). Tam existují metody zaručující náhodnost řetězců.

Podle našich znalostí je otevřeným problémem, jak využít ideu hašování s uspořádanými separovanými řetězci pro ostatní metody řešení kolizí (jmenovitě pro srůstající hašování).

Jak se vyrovnat s teoretickými předpoklady?
Připomeňme si předpoklady pro předchozí analýzy hašování:
1) Hašovací funkce se rychle spočítá (v čase $O(1)$).
2) Hašovací funkce rovnoměrně rozděluje univerzum (to znamená, že pro dvě různé hodnoty i a j hašovací funkce platí $-1 \leq |h^{-1}(i)| - |h^{-1}(j)| \leq 1$).
3) Vstupní data jsou rovnoměrně rozložená.

Předpoklad 1) je jasný a dá se jednoduše splnit.
Předpoklad 2) není tak striktní. Dokonce je výhodné, když rozdělení univerza hašovací funkcí kopíruje známé rozložení vstupních dat. Tato idea byla použita při návrhu překladače pro FORTRAN (Lum 1971). Fortran byl vyvinut pro vědecké výpočty a byl a je používán numerickými matematiky. O nich je známo, že obvykle používají identifikátory ve tvaru i1, i2, j1, x1 atd., kdežto identifikátory ve tvaru xyz, ijk se prakticky nevyskytují. Toto bylo využito v návrhu hašovací funkce pro ukládání identifikátorů při překladu programů. Tato funkce nerozdělovala univerzum rovnoměrně, zato se snažila, aby pro běžné tvary identifikátorů nedocházelo ke kolizím. Takto navržený překladač byl testován a získané výsledky byly porovnány s teoretickými výpočty. Porovnání výsledků (byla aplikována metoda separovaných řetězců) je v následující tabulce:

hodnota α	0.5	0.6	0.7	0.8	0.9
experiment	1.19	1.25	1.28	1.34	1.38
teorie	1.25	1.30	1.35	1.40	1.45

Ukázalo se, že prakticky dosažené výsledky jsou lepší. Protože však obvykle není předem známo rozložení vstupních dat, nelze uvedený fakt použít. Proto je požadavek 2) formulován pro rovnoměrné rozdělení (vychází se z předpokladu, že obecně bude nejvíce vyhovovat) a lze jej dobře splnit.

Závěr: Podmínky 1) a 2) můžeme splnit a když známe rozložení vstupních dat, můžeme dosáhnout ještě lepších výsledků, než dává analýza.

Hlavní problém při použití těchto metod tedy je, že neznáme rozložení vstupních dat a nemůžeme ho ani ovlivnit. Přitom tento předpoklad je pro uvedené metody a jejich analýzu nejcitlivější. Je reálné, že rozdělení vstupních dat bude nevhodné pro konkrétní použitou hašovací funkci. Důsledkem tohoto faktu bylo, že na počátku 70. let se začalo od hašování ustupovat.

V následujícím odstavci uvedeme metodu navrženou Carterem a Wegmanem v roce 1977, která obchází předpoklad 3) a nahrazuje ho předpokladem, který můžeme ovlivnit. To vedlo k novému zájmu o hašování a důsledkem bylo a je jeho hojné používání v praxi.

X. Univerzální hašování

Tato metoda je založena na následující ideji. Místo pevné hašovací funkce mějme množinu H funkcí z univerza do tabulky velikosti m takových, že pro každou množinu $S \subseteq U$, $|S| \leq m$ se většina funkcí chová dobře vůči S v tom smyslu, že je málo kolizí. Požadavek 3) je pak nahrazen tím, že se náhodně zvolí funkce $h \in H$ (s rovnoměrným rozdělením) a pomocí ní se pak hašuje. Protože funkci volíme my, tak můžeme požadavek rovnoměrného rozdělení zajistit. Následující analýza je platná pro všechny množiny $S \subseteq U$. Je dělána nikoliv přes všechny množiny $S \subseteq U$, ale přes všechny funkce $h \in H$ (tj. množina S je vždy pevně daná, funkce $h \in H$ se volí).

Při návrhu formalizace se však objevily technické potíže. Požadavek, aby funkce $h \in H$ byly rychle spočitatelné, vede k tomu, aby byly zadané analyticky. Jak uvidíme dále, formalizace vyžaduje znát velikost H, její stanovení je ale obecně pro takto zadané funkce problematické. Jednoduché řešení tohoto problému je následující. Předpokládáme, že máme množinu indexovaných funkcí a místo s funkcemi budeme pracovat s jejich indexy. Dvě funkce budeme považovat za různé, pokud mají různé indexy (i když jsou to stejné funkce). To znamená, že $H = \{h_i \mid i \in I\}$, kde pro každé $i \in I$ je h_i funkce z univerza U do množiny $\{0, 1, \ldots, m-1\}$. Za velikost této množiny budeme považovat mohutnost množiny I. Při aplikaci metody budeme vybírat prvek $i \in I$ (s rovnoměrným rozdělením) a jako hašovací funkci pak použijeme funkci h_i. V tomto případě bude očekávaná hodnota náhodné proměnné průměr přes I, přesněji, když ϕ je náhodná proměnná z množiny I do reálných čísel, pak její očekávaná hodnota je $\frac{\sum_{i \in I} \phi(i)}{|I|}$.

Formálně: Nechť U je univerzum. Systédm funkcí $H = \{h_i \mid i \in I\}$ z univerza U do množiny $\{0, 1, \ldots, m-1\}$ se nazývá c-<u>univerzální</u> (c je kladné reálné číslo), když

$$\forall x, y \in U,\ x \neq y \text{ platí } |\{i \in I \mid h_i(x) = h_i(y)\}| \leq \frac{c|I|}{m}.$$

Naším cílem bude napřed ukázat existenci c-univerzálních systémů, pak nalézt jejich vlastnosti a zjistit, zda pro každou množinu $S \subseteq U$ je očekávaná délka řetězců skutečně úměrná faktoru naplnění (použijeme metodu separovaných řetězců).

Existence univerzálních systémů

Bez újmy na obecnosti budeme předpokládat, že univerzum je tvaru $U = \{0, 1, \ldots, N-1\}$ pro nějaké prvočíslo N. To lze, protože víme, že pro každé přirozené číslo n existuje prvočíslo p takové, že $n \leq p \leq 2n$. Vyšetříme množinu funkcí $H = \{h_{a,b} \mid (a,b) \in U \times U\}$, kde $h_{a,b}(x) = ((ax+b) \bmod N) \bmod m$.

Protože soubor H obsahuje jen lineární funkce, lze každou funkci z H rychle vyčíslit. Tedy požadavek na rychlou vyčíslitelnost hašovací funkce je splněn.

Zvolme $x, y \in U$ taková, že $x \neq y$. Chceme nalézt všechna a a b z U taková, že $h_{a,b}(x) = h_{a,b}(y)$.

Když $h_{a,b}(x) = h_{a,b}(y)$, pak musí existovat $i \in \{0, 1, \ldots, m-1\}$ a $r, s \in \{0, 1, \ldots, \lceil \frac{N}{m} \rceil - 1\}$ tak, že platí

$$(ax + b \equiv i + rm) \bmod N$$
$$(ay + b \equiv i + sm) \bmod N.$$

Považujme x, y, i, r a s za konstanty a a a b za neznámé. Pak řešíme systém lineárních rovnic v tělese $\mathbb{Z}/ \bmod N$, kde $\mathbb{Z}$ jsou celá čísla. Matice soustavy

$$\begin{pmatrix} x & 1 \\ y & 1 \end{pmatrix}$$

je regulární, protože $x \neq y$. Tedy pro fixovaná x, y, i, r a s existuje jediné řešení této soustavy. Přitom pro daná x a y může i nabývat m hodnot, r a s nabývají $\lceil \frac{N}{m} \rceil$ hodnot.

Lze tedy shrnout, že pro každá $x, y \in U$ taková, že $x \neq y$, existuje $m\big(\lceil \frac{N}{m} \rceil\big)^2$ dvojic (a,b) z $U \times U$ takových, že $h_{a,b}(x) = h_{a,b}(y)$.

Věta. *Systém H je c-univerzální pro*

$$c = \frac{\big(\lceil \frac{N}{m} \rceil\big)^2}{\big(\frac{N}{m}\big)^2}.$$

Důkaz. Velikost $U \times U$ je N^2, protože velikost U je N. Pro různá $x, y \in U$ je počet $(a,b) \in U \times U$ takových, že $h_{a,b}(x) = h_{a,b}(y)$, roven

$$m\big(\lceil \frac{N}{m} \rceil\big)^2 = \frac{\big(\lceil \frac{N}{m} \rceil\big)^2}{\big(\frac{N}{m}\big)^2} \frac{N^2}{m} = \frac{\big(\lceil \frac{N}{m} \rceil\big)^2}{\big(\frac{N}{m}\big)^2} \frac{|U \times U|}{m}.$$

Když položíme

$$c = \frac{\big(\lceil \frac{N}{m} \rceil\big)^2}{\big(\frac{N}{m}\big)^2},$$

pak je systém c-univerzální.

Závěr: Dokázali jsme existenci c-univerzálních systémů pro c blízké 1 pro všechna univerza (viz komentář k předpokladu na univerza).
Všimněme si, že když $b \equiv b' \bmod m$, pak $h_{0,b} = h_{0,b'}$, a proto počet různých funkcí v systému H je menší než N^2. Na druhé straně nelze vylepšit odhad počtu funkcí, pro které platí, že $h_{a,b}(x) = h_{a,b}(y)$ pro libovolné $x, y \in U$, $x \neq y$. Proto, když bychom v definici c-univerzálního systému použili skutečnou velikost H místo velikosti množiny indexů, dostali bychom horší výsledky, tj. větší hodnotu c.

Vlastnosti univerzálního hašování

Nyní budeme předpokládat, že máme c-univerzální systém funkcí $H = \{h_i \mid i \in I\}$.
Pro každé $i \in I$ a pro prvky $x, y \in U$ definujme

$$\delta_i(x,y) = \begin{cases} 1, & \text{když } x \neq y \text{ a } h_i(x) = h_i(y), \\ 0, & \text{když } x = y \text{ nebo } h_i(x) \neq h_i(y). \end{cases}$$

Pro množinu $S \subseteq U$, $x \in U$ a $i \in I$ definujme

$$\delta_i(x,S) = \sum_{y \in S} \delta_i(x,y).$$

Pro fixovanou množinu $S \subseteq U$ a pro fixované $x \in U$ sečteme $\delta_i(x,S)$ přes všechna $i \in I$:

$$\sum_{i \in I} \delta_i(x,S) = \sum_{i \in I} \sum_{y \in S} \delta_i(x,y) = \sum_{y \in S} \sum_{i \in I} \delta_i(x,y) = \sum_{y \in S, y \neq x} |\{i \in I \mid h_i(x) = h_i(y)\}| \leq$$

$$\sum_{y \in S, y \neq x} c\frac{|I|}{m} = \begin{cases} (|S|-1)c\frac{|I|}{m} & \text{když } x \in S, \\ |S|c\frac{|I|}{m} & \text{když } x \notin S. \end{cases}$$

Z toho plyne, že horní odhad očekávané hodnoty $\delta_i(x,S)$ (která je horním odhadem očekávané délky řetězce na adrese $h_i(x)$) počítaný pro fixovanou množinu $S \subseteq U$ a fixované $x \in U$ přes všechny indexy funkcí $i \in I$ s rovnoměrným rozdělením, je

$$\frac{1}{|I|} \sum_{i \in I} \delta_i(x,S) = \begin{cases} c\frac{|S|-1}{m} & \text{když} \quad x \in S, \\ c\frac{|S|}{m} & \text{když} \quad x \notin S. \end{cases}$$

Věta. *Očekávaný čas operací* **MEMBER**, **INSERT** *a* **DELETE** *v c-univerzálním hašování je $O(1 + c\alpha)$, kde α je faktor naplnění (tj. $\alpha = \frac{|S|}{m}$).*
Očekávaný čas posloupnosti n operací **MEMBER**, **INSERT** *a* **DELETE** *aplikovaných na prázdnou tabulku v c-univerzálním hašování je $O(n(1 + \frac{c}{2}\alpha))$.*

Význam výsledku: Vzorec se liší jen o multiplikativní konstantu c od vzorce pro hašování se separovanými řetězci. Přitom c může být jen o málo menší než 1 a ve všech známých příkladech je $c \geq 1$. Takže, čeho jsme dosáhli? Rozdíl je v předpokladech. Zde je předpoklad 3) nahrazen předpokladem, že index i je vybírán z množiny I s rovnoměrným rozdělením, a není kladen žádný předpoklad na vstupní data. **Výběr indexu i můžeme ovlivnit, ale výběr vstupních dat nikoliv.** Můžeme zajistit rovnoměrné rozdělení výběru i z I nebo se k tomuto rozdělení hodně přiblížit.

Markovova nerovnost

Ukážeme další vlastnosti c-univerzálních systémů. Předpokládejme, že je dána množina $S \subseteq U$ a prvek $x \in U$. Označme μ očekávanou hodnotu proměnné $\delta_i(x, S)$ (tedy očekávané délky řetězce v S, který obsahuje x), a mějme $t \geq 1$.

Vyšetříme, jaká je pravděpodobnost, že pro $i \in I$ je hodnota $\delta_i(x, S)$ alespoň $t\mu$. Naším cílem je shora odhadnout tuto pravděpodobnost číslem $\frac{1}{t}$ (předpokladáme opět, že i je z I vybrán s rovnoměrným rozdělením).

Označme $I' = \{i \in I \mid \delta_i(x, S) \geq t\mu\}$. Pak platí

$$\mu = \frac{\sum_{i \in I} \delta_i(x, S)}{|I|} > \frac{\sum_{i \in I'} \delta_i(x, S)}{|I|} \geq \frac{\sum_{i \in I'} t\mu}{|I|} = \frac{|I'|}{|I|} t\mu.$$

Odtud

$$|I'| < \frac{|I|}{t},$$

a tedy

$$\mathrm{Prob}(\delta_i(x, S) \geq t\mu) = \frac{|I'|}{|I|} < \frac{1}{t}.$$

Věta. *Pro každý c-univerzální systém H, pro každou množinu $S \subseteq U$ a každý prvek $x \in U$ platí: Když očekávaná velikost $\delta_i(x, S)$ je μ, pak pravděpodobnost, že $\delta_i(x, S) \geq t\mu$, je menší než $\frac{1}{t}$.*

Poznámka: Toto tvrzení platí v teorii pravděpodobnosti obecně a nazývá se Markovova nerovnost. Uvedený důkaz ilustruje jeho jednoduchou ideu pro konečný případ.

Problémy

Hlavní problém pro praktické užití univerzálního hašování je zajištění rovnoměrného rozdělení při výběru indexu i z I.

Postup, jak provést tento výběr, je jednoduchý. Zakódujeme indexy z množiny I do množiny čísel $0, 1, \ldots, |I| - 1$. Zvolíme náhodně číslo i z intervalu $\{0, 1, \ldots, |I| - 1\}$ s rovnoměrným rozdělením a pak použijeme funkci s indexem, jehož kód je i. Protože ale velikost I může být obrovská (jak jsme viděli, může být i N^2, kde N je velikost univerza), nelze tento výběr obvykle jednoduše provést jedním zavoláním běžného náhodného generátoru. Lze ho např. realizovat takto:
Nalezneme nejmenší j splňující $2^j - 1 \geq |I| - 1$. Pak čísla v intervalu $\{0, 1, \ldots, 2^j - 1\}$ jednoznačně korespondují s posloupnostmi 0 a 1 délky j. Takže stačí vybrat náhodně posloupnost 0 a 1 délky j. K tomu použijeme náhodný generátor 0 a 1 s rovnoměrným rozdělením.

Uvedený postup je jednoduchý, má však jednu závadu. Skutečný náhodný generátor pro rovnoměrné rozdělení je prakticky nedosažitelný (zdá se, že to jsou některé fyzikální procesy). K dispozici je pouze pseudonáhodný generátor. Jeho nevýhoda je v tom, že čím je

generovaná posloupnost delší, tím je pravidelnější a tedy vzdálenější od skutečně náhodné posloupnosti.

Tato fakta motivují naše další cíle. Chceme zkonstruovat co nejmenší c-univerzální systémy (bez nějakých požadavků na velikost c) a zároveň nalézt dolní odhady velikosti c-univerzálních systémů.

Dolní odhady na velikost

Předpokládejme, že U je univerzum velikosti N a že $H = \{h_i \mid i \in I\}$ je c-univerzální systém funkcí hašujících do tabulky velikosti m. Bez újmy na obecnosti můžeme předpokládat, že

$$I = \{0, 1, \ldots, |I| - 1\}.$$

Indukcí definujme množiny $U_0, U_1, \ldots$ tak, že $U_0 = U$ a dále:
Nechť U_1 je největší podmnožina U_0 vzhledem k počtu prvků taková, že $h_0(U_1)$ je jednoprvková množina.
Nechť U_2 je největší podmnožina U_1 vzhledem k počtu prvků taková, že $h_1(U_2)$ je jednoprvková množina.
Nechť U_3 je největší podmnožina U_2 vzhledem k počtu prvků taková, že $h_2(U_3)$ je jednoprvková množina.
Obecně, nechť U_i je největší podmnožina U_{i-1} vzhledem k počtu prvků taková, že $h_{i-1}(U_i)$ je jednoprvková množina.

Protože hašujeme do tabulky velikosti m, platí $|U_i| \geq \lceil \frac{|U_{i-1}|}{m} \rceil$. Jelikož $|U_0| = N$, dostáváme indukcí, že $|U_i| \geq \lceil \frac{N}{m^i} \rceil$ pro každé i. Zvolme $i = \lceil \log_m N \rceil - 1$. Pak i je největší přirozené číslo takové, že $\frac{N}{m^i} > 1$. Tedy U_i má alespoň dva prvky. Zvolme tedy $x, y \in U_i$ tak, že $x \neq y$. Pak $h_j(x) = h_j(y)$ pro $j = 0, 1, \ldots, i-1$. Takže dostáváme

$$i \leq |\{j \in I \mid h_j(x) = h_j(y)\}| \leq \frac{c|I|}{m}.$$

Věta. *Když $H = \{h_i \mid i \in I\}$ je c-univerzální systém pro univerzum U o velikosti N hašující do tabulky s m řádky, pak*

$$|I| \geq \frac{m}{c}(\lceil \log_m N \rceil - 1).$$

Posloupnosti 0 a 1 při náhodné volbě i z I musí mít proto délku $\lceil (\log m - \log c + \log\log N - \log\log m) \rceil$.

Ve výše uvedené větě jsou všechny logaritmy o základu 2, poslední člen jsme získali z převodu $\log_m N$ na $\log_2 N$.

Tato věta tedy říká, že čím větší je univerzum, tím větší musí být c-univerzální systém. Podle našeho odhadu velikost c-univerzálního systému roste úměrně alespoň s logaritmem velikosti univerza (následující konstrukce ukáže, že můžeme této velikosti dosáhnout až na multiplikativní konstantu).

Použijeme-li tuto metodu, tak nám náhodná volba indexu hašovací funkce nemusí dát dobrý výsledek, ale v takovém případě můžeme volbu indexu hašovací funkce opakovat a splnění předpokladu, že indexy jsou vybírány s rovnoměrným rozdělením, nám zaručuje dobrý výsledek při malém počtu voleb. To však neplatí pro obrovská univerza, protože nemáme náhodný, ale jen pseudonáhodný generátor. To nás varuje, abychom byli při obrovských univerzech opatrní.

Malý univerzální systém

Zkonstruujeme c-univerzální systém takový, že logaritmus velikosti jeho indexové množiny pro velká univerza je až na aditivní konstantu menší než $4(\log m + \log\log N)$, kde N je velikost univerza a m je počet řádků v tabulce. To znamená, že délka posloupnosti 0 a 1 pro náhodnou volbu funkce $i \in I$ je až na multiplikativní konstantu rovna jejímu dolnímu odhadu.

Nechť $p_1, p_2, \ldots$ je rostoucí posloupnost všech prvočísel.
Předpokládáme, že univerzum je tvaru $U = \{0, 1, \ldots, N-1\}$, kde N je přirozené číslo, a že je dáno číslo m. Nechť t je nejmenší přirozené číslo takové, že $t \ln p_t \geq m \ln N$. Definujme

$$H_1 = \{g_{c,d}(h_\ell) \mid t < \ell \leq 2t,\ c, d \in \{0, 1, \ldots, p_{2t} - 1\}\},$$

kde $h_\ell(x) = x \bmod p_\ell$ a $g_{c,d}(x) = ((cx + d) \bmod p_{2t}) \bmod m$.
Ukážeme, že H_1 je 3.25-univerzální systém.

Indexová množina systému H_1 je množina $I = \{(c, d, l) \mid c, d \in \{0, 1, \ldots, p_{2t}-1\},\ t < l \leq 2t\}$ a její velikost je $|I| = tp_{2t}^2$.

Připomeneme si odhady velikosti prvočísel.

Věta. *Pro $i \geq 6$ platí*

$$i(\ln i + \ln\ln i) > p_i > i \ln i.$$

Zde $\ln i$ znamená přirozený logaritmus, tj. logaritmus o základu e. Tuto větu lze nalézt v základních monografiích o teorii čísel, např. Algorithmic number theory od E. Bacha a J. Shallita, která vyšla v MIT Press v roce 1996.

Z předchozí věty plyne, že $p_i \leq 2i \ln i$ pro každé i, tedy $p_{2t} \leq 4t \ln 2t$. Odtud $|I| \leq 16t^3 \ln^2 2t$. Z toho dostáváme

$$\log(|I|) \leq 4 + 3\log t + 2\log\log t.$$

Pro dostatečně velké t (takové, že $\log t \geq 2\log\log t$) platí, že $\log(|I|) \leq 4(1 + \log t)$. Z definice t plyne, že $t \leq m \ln N$ (když $p_t \geq 3$, pak $\ln p_t \geq 1$).
Závěr: $\log(|I|) \leq 4(1 + \log m + \log\log N)$.

Tedy je splněn požadavek na odhad velikosti I a dále budeme dokazovat 3.25-univerzalitu.

Univerzalita malého systému

Zvolme různá x a y z univerza U. Označíme

$$G_1 = \{(c,d,\ell) \mid g_{c,d}(h_\ell(x)) = g_{c,d}(h_\ell(y)),\ h_\ell(x) \neq h_\ell(y)\},$$
$$G_2 = \{(c,d,\ell) \mid g_{c,d}(h_\ell(x)) = g_{c,d}(h_\ell(y)),\ h_\ell(x) = h_\ell(y)\}$$

a odhadneme velikost G_1 a G_2.

Odhad velikosti G_1. Když $(c,d,\ell) \in G_1$, pak existují $i \in \{0,1,\ldots,m-1\}$ a $r,s \in \{0,1,\ldots,\lceil \frac{p_{2t}}{m} \rceil - 1\}$ taková, že

$$(c(x \bmod p_\ell) + d \equiv i + rm) \bmod p_{2t}$$
$$(c(y \bmod p_\ell) + d \equiv i + sm) \bmod p_{2t}.$$

Když c a d považujeme za neznámé, pak je to soustava lineárních rovnic s regulární maticí (protože $x \bmod p_\ell \neq y \bmod p_\ell$) v tělese $\mathbb{Z}/ \bmod p_{2t}$, a tedy pro každé ℓ, i, r a s existuje právě jedna dvojice (c,d), která je jejím řešením. Proto

$$|G_1| \leq tm(\lceil \frac{p_{2t}}{m} \rceil)^2 \leq \frac{tp_{2t}^2}{m}(1 + \frac{m}{p_{2t}})^2 = \frac{|I|}{m}(1 + \frac{m}{p_{2t}})^2.$$

Odhad velikosti G_2. Označme $L = \{\ell \mid t < \ell \leq 2t,\ x \bmod p_\ell = y \bmod p_\ell\}$ a položme $P = \prod_{\ell \in L} p_\ell$. Protože $x \bmod p_\ell = y \bmod p_\ell$ je ekvivalentní s tvrzením, že p_ℓ dělí $|x-y|$, tak P dělí $|x-y|$. Odtud plyne, že $P < N$, protože $|x-y| < N$. Z faktu, že $p_t < p_\ell$ pro každé $\ell \in L$, dostáváme, že $P > p_t^{|L|}$. Z definice t dále plyne, že $|L| \leq \frac{\ln N}{\ln p_t} \leq \frac{t}{m}$. Protože $(c,d,\ell) \in G_2$, právě když $\ell \in L$ a $c,d \in \{0,1,\ldots,p_{2t}-1\}$, shrneme, že

$$|G_2| \leq |L| p_{2t}^2 \leq \frac{tp_{2t}^2}{m} = \frac{|I|}{m}.$$

Nyní spočítáme odhad výrazu $(1 + \frac{m}{p_{2t}})^2$. Zřejmě, bez jakýchkoliv předpokladů, platí, že $(1 + \frac{m}{p_{2t}})^2 < 4$, protože $m \geq p_{2t}$ implikuje

$$t \ln p_t \leq t \ln p_{2t} \leq m \ln m \leq m \ln N$$

a to je spor. Nejprve za předpokladů, že $t \geq 6$ a $m \ln m \ln\ln m < N$ ukážeme, že $m < \frac{p_t}{\ln t}$. Skutečně, z předpokladů plyne

$$\ln m + \ln\ln m + \ln\ln\ln m < \ln N.$$

Kdyby $m \geq \frac{p_t}{\ln t}$, pak z Věty o velikosti prvočísel dostaneme $m \geq \frac{p_t}{\ln t} > \frac{t \ln t}{\ln t} = t$ a odtud plyne, že

$$t \ln p_t < t(\ln t + \ln\ln t + \ln\ln\ln t) < m(\ln m + \ln\ln m + \ln\ln\ln m) < m \ln N$$

a to je spor s definicí t. Tedy $m < \frac{p_t}{\ln t}$. Odtud plyne

$$\frac{m}{p_{2t}} < \frac{\frac{p_t}{\ln t}}{2t \ln 2t} < \frac{t(\ln t \ln\ln t)}{2t \ln t \ln 2t} < \frac{1}{2}$$

pomocí Věty o odhadu velikosti prvočísel a faktu, že

$$\ln 2t > \ln t > \ln\ln t \quad \text{pro všechna } t \geq 1.$$

Když toto shrneme, dostaneme, že $(1 + \frac{m}{p_{2t}})^2 < 1.5^2 = 2.25$, a odtud plyne

$$\{i \in I \mid h(x) = h(y)\}| = |G_1| + |G_2| \leq \frac{|I|}{m}(1 + \frac{m}{p_{2t}})^2 + \frac{|I|}{m} \leq \frac{|I|}{m}(1 + 2.25) = 3.25\frac{|I|}{m}.$$

Shrnutí:

Věta. *Když $m \ln m \ln\ln m < N$ a $\ln N \geq 13$, pak H_1 je 3.25-univerzální systém hašovacích funkcí a bez jakýchkoliv předpokladů H_1 je 5-univerzální systém hašovacích funkcí. Velikost jeho indexové množiny je*

$$|I| = O(m^3 \ln^3 N(\ln m + \ln\ln N)^2).$$

Na závěr této části odvodíme dolní odhad na velikost c.

Věta. *Mějme univerzum U o velikosti N a nechť H je c-univerzální systém hašovacích funkcí do tabulky o velikosti m. Pak*

$$c \geq \frac{N - m}{N - 1} > 1 - \frac{m}{N}.$$

Nejprve dokážeme technické lemma.

Lemma. *Mějme reálná čísla b_i pro $i = 0, 1, \ldots, m-1$ a nechť $b = \sum_{i=0}^{m-1} b_i$. Pak platí*

$$\sum_{i=0}^{m-1} b_i(b_i - 1) \geq b(\frac{b}{m} - 1).$$

Důkaz lemmatu. Podle Cauchyho-Schwarzovy nerovnosti platí

$$(\sum_{i=0}^{m-1} x_i y_i)^2 \leq (\sum_{i=0}^{m-1} x_i^2)(\sum_{i=0}^{m-1} y_i^2)$$

pro každé dvě m-tice reálných čísel $x_0, x_1, \ldots, x_{m-1}$ a $y_0, y_1, \ldots, y_{m-1}$. Položíme-li $x_i = b_i$ a $y_i = 1$, pak dostaneme

$$(\sum_{i=0}^{m-1} b_i)^2 = b^2 \leq m(\sum_{i=0}^{m-1} b_i^2),$$

a tedy $\frac{b^2}{m} \leq \sum_{i=0}^{m-1} b_i^2$. Odtud

$$\sum_{i=0}^{m-1} b_i(b_i-1) = \sum_{i=0}^{m-1} b_i^2 - \sum_{i=0}^{m-1} b_i = \sum_{i=0}^{m-1} b_i^2 - b \geq \frac{b^2}{m} - b = b(\frac{b}{m} - 1). \quad \square$$

Důkaz Věty. Mějme funkci $f : U \to S$, kde U má velikost N a S má velikost m. Označme A množinu uspořádaných dvojic $u, v \in U$ takových, že $u \neq v$ a $f(u) = f(v)$. Když pro $s \in S$ označíme $k_s = |f^{-1}(s)|$, pak $|A| = \sum_{s \in S} k_s(k_s - 1)$. Z lemmatu plyne, že

$$|A| = \sum_{s \in S} k_s(k_s - 1) \geq N(\frac{N}{m} - 1) = N(\frac{N-m}{m}),$$

protože $\sum_{s \in S} k_s = N$.
Když $H = \{h_i \mid i \in I\}$ je c-univerzální systém funkcí z univerza U o velikosti N do tabulky o velikosti m, pak pomocí lemmatu dostáváme

$$\begin{aligned} |I|N(\frac{N-m}{m}) &\leq \sum_{i \in I} |\{(x,y) \in U \times U \mid h(x) = h(y),\, x \neq y\}| = \\ &\sum_{(x,y) \in U \times U,\, x \neq y} |\{i \in I \mid h(x) = h(y)\}| \\ &\leq \sum_{(x,y) \in U \times U, x \neq y} c\frac{|I|}{m} = N(N-1)c\frac{|I|}{m}. \end{aligned}$$

Odtud plyne, že $N - m \leq c(N-1)$, a tedy

$$c \geq \frac{N-m}{N-1} > \frac{N-m}{N} = 1 - \frac{m}{N}. \quad \square$$

Problémy univerzálního hašování

Lze použít i jiné metody na řešení kolizí než separované řetězce. Jak to ovlivní použitelnost univerzálního hašování? Platí podobné vztahy jako pro pevně danou hašovací funkci? Jaký vliv na efektivnost má nepřítomnost operace **DELETE**?

Existuje c-univerzální hašovací systém pro $c < 1$? Jaký je vztah mezi velikostí c-univerzálního hašovacího systému a velikostí c? Lze zkonstruovat malý c-univerzální systém pro $c < 3.25$? Zde hraje roli fakt, že při $c = 3.25$ se očekávaná délka řetězce může pohybovat až kolem hodnoty 7.

Použitím Čebyševovy nerovnosti místo Markovovy dostaneme horní odhad $\frac{\sigma^2}{t^2}$ pro pravděpodobnost, že délka řetězce je o t větší než její očekávaná hodnota (σ^2 je rozptyl). Za jakých okolností je to lepší odhad? Lze použít i momenty vyšších řádů?

Uvažujme následující model:
Je dána základní velikost tabulky m a dále pro $j = 0, 1, \ldots$ čísla (parametry) l_j a c-univerzální hašovací systémy $H_j = \{h_i \mid i \in I_j\}$ z univerza do tabulky s $m2^j$ řádky.
Množina $S \subseteq U$ je reprezentována následovně: je dáno j takové, že když $j > 0$, pak $m2^{j-2} \leq |S| \leq m2^j$, když $j = 0$, pak $|S| \leq m$, a je zvolen index $i \in I_j$. Dále máme prosté řetězce $r_0, r_1, \ldots, r_{m2^j-1}$, jejichž délky jsou nejvýše l_j, a řetězec r_k obsahuje prvky $\{s \in S \mid h_i(s) = k\}$.
Operace **INSERT**(x) prohledá řetězec $r_{h_i(x)}$ a když tento řetězec neobsahuje prvek x, pak ho přidá. Když $m2^{j-2} \leq |S| \leq m2^j$ a délka řetězce $r_{h_i}(x)$ je nejvýše l_j, pak operace končí. Když $|S| > m2^j$, tak se nejdříve zvětší j o 1. Pak se náhodně zvolí $i \in I_j$ a zkonstruují se řetězce reprezentující S. Když některý z nich má délku větší než l_j, tak se volba a konstrukce řetězců opakuje tak dlouho, dokud se nepovede zvolit $i \in I_j$ takové, že všechny zkonstruované řetězce mají délku nejvýše l_j. Operace **DELETE** se řeší analogicky.
Problém: Jak volit parametry l_i?

Jak použít Markovovu nerovnost a očekávanou délku maximálního řetězce pro odhad očekávaného počtu voleb hašovací funkce?

V případě řešení kolizí dvojitým hašováním nebo hašováním s lineárním přidáváním jsou zapotřebí silnější podmínky na velikost S. Jaké?
S jakou pravděpodobností se přepočítává uložení S s novou hašovací funkcí?

XI. Perfektní hašování

Jiný způsob řešení kolizí představuje perfektní hašování. Idea je nalézt pro danou množinu hašovací funkci, která nepřipouští kolize.

Nevýhodou této metody je, že při ní nelze přirozeným způsobem realizovat operaci **INSERT** (pro nový vstup nemůžeme zaručit, že nevznikne kolize). Metodu lze prakticky použít pro úlohy, kde lze očekávat hodně operací **MEMBER**, zatímco operace **INSERT** se téměř nevyskytuje (pak se kolize může řešit pomocí malé pomocné tabulky, kam se ukládají kolidující data).

Zadání úlohy: Pro danou množinu $S \subseteq U$ chceme nalézt hašovací funkci h takovou, že
1) pro $s, t \in S$ takové, že $s \neq t$, platí $h(s) \neq h(t)$ (tj. h je perfektní hašovací funkce pro S);
2) h hašuje do tabulky s m řádky, kde m je přibližně stejně velké jako $|S|$ (není praktické hašovat do příliš velkých tabulek – ztrácí se jeden ze základních důvodů pro použití hašování);
3) h musí být rychle spočitatelná – jinak hašování není rychlé;
4) uložení h nesmí vyžadovat moc paměti, nejvýhodnější je analytické zadání (když zadání h bude vyžadovat příliš mnoho paměti, např. když by byla dána tabulkou, pak se ztrácí důvod k použití hašování stejně jako v bodě 2).

Jedna z výhod této metody je, že nalezení perfektní hašovací funkce nevyžaduje velkou rychlost. Na její nalezení můžeme spotřebovat více času, protože se provádí jen na začátku úlohy.

Uvedené požadavky motivují zavedení následujícího pojmu:
Mějme univerzum $U = \{0, 1, \ldots, N-1\}$. Soubor funkcí H z U do množiny $\{0, 1, \ldots, m-1)$

se nazývá (N, m, n)-perfektní, když pro každou množinu $S \subseteq U$ takovou, že $|S| = n$, existuje $h \in H$ perfektní pro S (tj. $h(s) \neq h(t)$ pro každá dvě různá $s, t \in S$).

Tento pojem je sice zjednodušením naší úlohy, ale dává nám dobrou představu o její obtížnosti. Nevíme totiž, zda vůbec pro každou množinu existuje její perfektní hašovací funkce splňující další naše požadavky. Proto nejprve vyšetříme vlastnosti (N, m, n)-perfektních souborů funkcí.

Dolní odhady na velikost

Mějme funkci h z U do množiny $\{0, 1, \ldots, m-1\}$. Nejprve odhadneme počet množin $S \subseteq U$ takových, že h je perfektní hašovací funkce pro S a $|S| = n$. Funkce h je perfektní pro množinu $S \subseteq U$, právě když pro každé $i = 0, 1, \ldots, m-1$ je $|h^{-1}(i) \cap S| \leq 1$. Odtud plyne, že počet těchto množin je

$$\sum \{ \prod_{j=0}^{n-1} |h^{-1}(i_j)| \mid 0 \leq i_0 < i_1 < \cdots < i_{n-1} < m \}.$$

Vysvětlení: Když h je perfektní pro množinu S, pak $h(S) = \{i_j \mid j = 0, 1, \ldots, n-1\}$, kde $0 \leq i_0 < i_1 < \cdots < i_{n-1} < m$. Protože $|S \cap h^{-1}(i_j)| = 1$ pro každé $j = 0, 1, \ldots, n-1$, tak těchto množin je $\prod_{j=0}^{n-1} |h^{-1}(i_j)|$. Vysčítáním přes $h(S)$ dostaneme náš výraz.

Z Cauchyho-Schwarzovy nerovnosti plyne, že velikost tohoto výrazu je maximální, když pro každé i platí $|h^{-1}(i)| = \frac{N}{m}$. Tedy h může být perfektní nejvýše pro $\binom{m}{n}(\frac{N}{m})^n$ množin (číslo $\binom{m}{n}$ určuje počet posloupností $0 \leq i_0 < i_1 < \cdots < i_{n-1} < m$). Protože n-prvkových podmnožin univerza je $\binom{N}{n}$, dostáváme, že

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

Použijeme ještě jinou metodu k dolnímu odhadu velikosti H. Tato metoda je založená na stejné ideji jako je dolní odhad velikosti c-univerzálních systémů. Předpokládejme, že $H = \{h_1, \ldots, h_t\}$ je (N, m, n)-perfektní soubor funkcí. Indukcí definujme množiny U_i:
$U_0 = U$ a pro $i > 0$ je U_i největší (co do počtu prvků) podmnožina U_{i-1} taková, že h_i je konstantní na U_i. Pak $|U_i| \geq \frac{|U_{i-1}|}{m}$ pro všechna $i > 0$. Z $|U_0| = N$ plyne, že $|U_i| \geq \frac{N}{m^i}$. Pro každé $i = 1, 2, \ldots, t$ a každé $j \leq i$ je $h_j(U_i)$ jednobodová množina. Proto žádná h_j, kde $j \leq i$, není perfektní pro množinu $S \subseteq U$ takovou, že $|S \cap U_i| \geq 2$. Protože H je (N, m, n)-perfektní, musí být $|U_t| \leq 1$, a tedy $\frac{N}{m^t} \leq 1$. Odtud plyne, že $t \geq \frac{\log N}{\log m}$. Shrneme uvedené odhady:

Věta. *Když H je (N, m, n)-perfektní soubor funkcí, pak*

$$|H| \geq \max\{\frac{\binom{N}{n}}{\binom{m}{n}(\frac{N}{m})^n}, \frac{\log N}{\log m}\}.$$

Existence (N, m, n)-perfektního souboru

Mějme univerzum $U = \{0, 1, \ldots, N-1\}$ a soubor funkcí $H = \{h_1, h_2, \ldots, h_t\}$ z univerza U do množiny $\{0, 1, \ldots, m-1\}$. Reprezentujeme tento soubor pomocí matice $M(H)$ typu $N \times t$ s hodnotami $\{0, 1, \ldots, m-1\}$ tak, že pro $x \in U$ a $i = 1, 2, \ldots, t$ je v x-tém řádku a i-tém sloupci této matice hodnota $h_i(x)$. Pak pro množinu $S = \{s_1, s_2, \ldots, s_n\} \subseteq U$ platí, že žádná funkce $h \in H$ není perfektní pro množinu S, právě když podmatice $M(H)$ tvořená řádky $s_1, s_2, \ldots, s_n$ a všemi sloupci nemá prostý sloupec. Počet matic typu $N \times t$ takových, že žádná z reprezentovaných funkcí není perfektní pro fixovanou množinu $S \subseteq U$ takovou, že $|S| = n$, je nejvýše

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

Vysvětlení: m^n je počet všech funkcí z S do $\{0, 1, \ldots, m-1\}$, $\prod_{i=0}^{n-1}(m-i)$ je počet prostých funkcí z S do $\{0, 1, \ldots, m-1\}$, a tedy počet všech podmatic s n řádky takových, že žádný jejich sloupec není prostý, je $(m^n - \prod_{i=0}^{n-1}(m-i))^t$. Tyto podmatice můžeme libovolně doplnit na matici typu $N \times n$ a pro každou matici je těchto doplnění $m^{(N-n)t}$.

Podmnožin U velikosti n je $\binom{N}{n}$, tedy počet všech matic, které nereprezentují (N, m, n)-perfektní systém, je menší než

$$\binom{N}{n}(m^n - \prod_{i=0}^{n-1}(m-i))^t m^{(N-n)t}.$$

Všech matic je m^{Nt} a když

$$(*) \qquad \binom{N}{n}(m^n - \prod_{i=0}^{n-1}(m-i))^t m^{(N-n)t} < m^{Nt},$$

pak nutně existuje (N, m, n)-perfektní systém. Vydělíme nerovnost číslem m^{Nt}, pak ji zlogaritmujeme a vyčíslíme odhad velikosti t. Tím dostaneme, že následující výrazy jsou ekvivalentní s nerovností $(*)$:

$$\binom{N}{n}(1 - \frac{\prod_{i=0}^{n-1}(m-i)}{m^n})^t < 1 \quad \Leftrightarrow \quad t \geq \frac{\ln \binom{N}{n}}{-\ln(1 - \frac{\prod_{i=0}^{n-1}(m-i)}{m^n})}.$$

Protože $\ln \binom{N}{n} \leq n \ln N$ a protože

$$-\ln(1 - \frac{\prod_{i=0}^{n-1}(m-i)}{m^n}) \geq \prod_{i=0}^{n-1}(1 - \frac{i}{m}) = e^{\sum_{i=0}^{n-1} \ln(1-\frac{i}{m})} \geq e^{\int_0^n \ln(1-\frac{x}{m})dx},$$

kde integrál je roven

$$m[(1 - \frac{n}{m})(1 - \ln(1 - \frac{n}{m})) - 1] \geq m[(1 - \frac{n}{m})(1 + \frac{n}{m}) - 1] = -\frac{n^2}{m},$$

dostáváme, že když $t \geq n(\ln N)e^{\frac{n^2}{m}}$, pak (*) platí, a tedy existuje (N, m, n)-perfektní soubor funkcí.

Existence (N, m, n)-perfektního souboru funkcí ale nezaručuje splnění požadavků 3) a 4). Abychom uspěli i v ostatních požadavcích, použijeme ideu z metody univerzálního hašování ke konstrukci perfektní hašovací funkce.

Konstrukce perfektní hašovací funce

Předpokládejme, že univerzum je tvaru $U = \{0, 1, \ldots, N-1\}$, kde N je prvočíslo. Mějme množinu $S \subseteq U$ o velikosti n. Budeme uvažovat funkce

$$h_k(x) = (kx \bmod N) \bmod m \qquad \text{pro } k = 1, 2, \ldots, N-1.$$

Pro $i = 0, 1, \ldots, m-1$ a $k = 1, 2 \ldots, N-1$ označme

$$b_i^k = |\{x \in S \mid (kx \bmod N) \bmod m = i\}|.$$

Význam b_i^k: Hodnoty b_i^k lze považovat za veličiny, které ukazují odchylku funkce h_k od perfektnosti (udávají počet prvků množiny S, které se zobrazily na i-tý řádek tabulky). Všimněme si, že

$$\text{když } b_i^k \geq 2, \text{ pak } (b_i^k)^2 - b_i^k \geq 2,$$

protože platí $a^2 - a \geq 2$, když $a \geq 2$. Na druhou stranu

$$b_i^k \leq 1 \text{ implikuje } (b_i^k)^2 - b_i^k = 0.$$

Tedy dostáváme

Věta. *Funkce h_k je perfektní, právě když $\sum_{i=0}^{m-1}(b_i^k)^2 - n < 2$.*

Důkaz. Když h_k je perfektní funkce, pak $b_i^k \leq 1$ pro každé $i = 0, 1, \ldots, m-1$, a tedy

$$\sum_{i=0}^{m-1}(b_i^k)^2 = \sum_{i=0}^{m-1} b_i^k = n < n + 2.$$

Když h_k není perfektní, pak existuje $i_0 \in \{0, 1, \ldots, m-1\}$ takové, že $b_{i_0}^k \geq 2$. Protože $(b_i^k)^2 \geq b_i^k$ pro každé $i = 0, 1, \ldots, m-1$ a protože $(b_{i_0}^k)^2 \geq b_{i_0}^k + 2$, dostáváme

$$\sum_{i=0}^{m-1}(b_i^k)^2 \geq \sum_{i=0}^{m-1} b_i^k + 2 = n + 2. \qquad \square$$

Nyní odhadneme výraz $\sum_{k=1}^{N-1}\big((\sum_{i=0}^{m-1}(b_i^k)^2) - n\big)$. Metoda použitá k odhadu je modifikací důkazu z předchozí sekce, že systém je c-univerzální:

$$\sum_{k=1}^{N-1}\big((\sum_{i=0}^{m-1}(b_i^k)^2) - n\big) = \sum_{k=1}^{N-1}\big((\sum_{i=0}^{m-1}|\{x \in S \mid h_k(x) = i\}|^2) - n\big) =$$

$$\sum_{k=1}^{N-1}|\{(x, y) \mid x, y \in S,\, x \neq y,\, h_k(x) = h_k(y)\}| =$$

$$\sum_{x,y \in S, x \neq y}|\{k \mid 1 \leq k < N,\, h_k(x) = h_k(y)\}|.$$

Zvolme $x, y \in S$ taková, že $x \neq y$, a předpokládejme, že pro $k \in \{1, 2, \ldots, N-1\}$ platí $h_k(x) = h_k(y)$. To znamená, že existují $i \in \{0, 1, \ldots, m-1\}$ a $r, s \in \{0, 1, \ldots, \lfloor \frac{N}{m} \rfloor\}$ tak, že $kx \equiv i + rm \bmod N$ a $ky \equiv i + sm \bmod N$. Z toho plyne, že $k(y-x) \equiv ky - kx \equiv (s-r)m \bmod N$. Rozdíl proti metodě z předchozí sekce spočívá ve faktu, že může být $s - r < 0$. Je však zřejmé, že $-\lfloor \frac{N}{m} \rfloor \leq (s-r) \leq \lfloor \frac{N}{m} \rfloor$, a protože $x \neq y$, je $s - r \neq 0$. Dále, protože celá čísla modulo N tvoří těleso (předpokládáme, že N je prvočíslo), tak pro každé celé číslo $t \in \{-\lfloor \frac{N}{m} \rfloor, -\lfloor \frac{N}{m} \rfloor + 1, \ldots, -1, 1, 2, \ldots, \lfloor \frac{N}{m} \rfloor\}$ existuje právě jedno $k \in \{1, 2, \ldots, N-1\}$ takové, že $k(y-x) \equiv tm \bmod N$ (když $t < 0$, pak to odpovídá rovnici $k(x-y) \equiv -t(N-m) \bmod N$). Odtud dostáváme, že existuje nejvýše $2\lfloor \frac{N}{m} \rfloor = 2\lfloor \frac{N-1}{m} \rfloor$ čísel $k \in \{1, 2, \ldots, n-1\}$ takových, že $h_k(x) = h_k(y)$ (rovnost $2\lfloor \frac{N}{m} \rfloor = 2\lfloor \frac{N-1}{m} \rfloor$ platí, protože N je prvočíslo a $m < N$).

Efektivita tohoto postupu závisí na počtu k takových, že $h_k(x) = h_k(y)$, a náš postup dává, oproti výsledkům v předchozí sekci, jen horní odhad. Vzniká otázka, jestli jsme nebyli příliš opatrní a jestli ve skutečnosti neplatí rovnost, tj. jestli jsme nespočítali všechna taková k, že $h_k(x) = h_k(y)$. Ukážeme, že ne. Zvolme t, x a y a nalezněme k, že $k(y-x) \equiv tm \bmod N$. Předpokládejme dále, že $k(x) \equiv i + rm \bmod N$. Pak se lehce spočítá, že $h_k(x) = h_k(y)$, právě když $0 \leq r + t \leq \lfloor \frac{N}{m} \rfloor$. Budeme to ilustrovat na příkladech. Předpokládejme, že $N = 13$, $m = 9$ a $y - x = 1$. Pak t může nabývat jen hodnot $t = -1$ a $t = 1$, a tedy v prvním případě $k = 4$ a v druhém případě $k = 9$. Když $x = 1$ a $y = 2$, pak rutinním výpočtem zjistíme, že $h_k(1) \neq h_k(2)$ pro všechna k, a když $x = 6$ a $y = 7$, pak $h_4(6) = h_4(7) = 2$ a $h_9(6) = h_9(7) = 2$. Když změníme hodnotu m na 5, pak t může nabývat hodnot -2, -1, 1 a 2 a tomu odpovídají hodnoty $k = 3$, $k = 8$, $k = 5$ a $k = 10$. Pro $x = 1$ a $y = 2$ platí $h_5(1) = h_5(2) = 0$, $h_8(1) = h_8(2) = 3$ a pro $k \neq 5, 8$ platí $h_k(1) \neq h_k(2)$. Tedy se jedná skutečně jen o odhad a nelze ho jednoduchým způsobem vylepšit.

Jelikož pro různá x a y existuje nejvýše $2\lfloor \frac{N}{m} \rfloor = 2\lfloor \frac{N-1}{m} \rfloor$ různých $k \in \{1, 2, \ldots, N-1\}$, pro něž platí $h_k(x) = h_k(y)$, dostáváme

$$\sum_{k=1}^{N-1} \big((\sum_{i=0}^{m-1} (b_i^k)^2) - n \big) = \sum_{x,y \in S,\, x \neq y} |\{k \in \{1, 2, \ldots, N-1\} \mid h_k(x) = h_k(y)\}| \leq$$
$$\sum_{x,y \in S, x \neq y} 2\lfloor \frac{N-1}{m} \rfloor \leq 2(N-1)\frac{n(n-1)}{m}.$$

Tedy existuje k takové, že $\sum_{i=0}^{m-1} (b_i^k)^2 \leq \frac{2n(n-1)}{m} + n$.

Nyní ukážeme, že existuje více než $\frac{(N-1)}{4}$ takových k, že $\sum_{i=0}^{m-1} (b_i^k)^2 < \frac{3n(n-1)}{m} + n$, a tento fakt bude základem pro pravděpodobnostní algoritmus. Skutečně, když existuje alespoň $\frac{3(N-1)}{4}$ takových k, že $\sum_{i=0}^{m-1} (b_i^k)^2 \geq \frac{3n(n-1)}{m} + n$, pak

$$\sum_{k=1}^{N-1} \big((\sum_{i=0}^{m-1} (b_i^k)^2) - n) \big) \geq \frac{3n(n-1)}{m} \frac{3(N-1)}{4} = \frac{9}{4} \frac{n(n-1)}{m} (N-1) > 2(N-1)\frac{n(n-1)}{m},$$

a to je spor. Tedy při rovnoměrném výběru k platí

$$\text{Prob}\{k \in \{1, 2, \ldots, N-1\} \mid \sum_{i=0}^{m-1} (b_i^k)^2 < \frac{3n(n-1)}{m} + n\} \geq \frac{1}{4}.$$

Pro nalezení perfektní hašovací funkce použijeme pravděpodobnostní algoritmus. Jeho výpočet kromě vstupních dat závisí i na náhodně zvolených hodnotách jistých proměnných (v našem případě na volbě k). Algoritmus buď skončí s požadovaným výsledkem (v našem případě s hašovací funkcí splňující dané podmínky) nebo opakuje výpočet pro jinou volbu náhodných hodnot (v našem případě opakuje volbu k). Jeho složitost je popsána dobou výpočtu a očekávaným počtem opakování, než se získá požadovaný výsledek.

Tvrzení. *Když $n = m$, pak*

(a) *existuje deterministický algoritmus, který nalezne k takové, že $\sum_{i=0}^{m-1}(b_i^k)^2 < 3n$, v čase $O(nN)$;*

(b) *existuje pravděpodobnostní algoritmus, který nalezne k takové, že $\sum_{i=0}^{m-1}(b_i^k)^2 < 4n$, v očekávaném čase $O(n)$. Očekávaný počet iterací výpočtu k je nejvýše 4.*

Dále

(c) *existuje deterministický algoritmus, který pro $m = n(n-1)+1$ v čase $O(nN)$ nalezne k takové, že h_k je perfektní;*

(d) *existuje pravděpodobnostní algoritmus, který pro $m = 2n(n-1)$ v očekávaném čase $O(n)$ nalezne k takové, že h_k je perfektní. Očekávaný počet iterací výpočtu k je nejvýše 4.*

Důkaz. Mějme $n = m$. Protože výpočet $\sum_{i=0}^{m-1}(b_i^k)^2$ pro pevné k vyžaduje čas $O(n)$, pak systematickým prohledáním nalezneme k takové, že $\sum_{i=0}^{m-1}(b_i^k)^2 \leq \frac{2n(n-1)}{n}+n = 3n-2 < 3n$, v čase $O(nN)$. Tím je dokázáno a). Pro náhodně zvolené k víme, že platí

$$\mathrm{Prob}(\sum_{i=0}^{m-1}(b_i^k)^2 \leq \frac{3n(n-1)}{n} + n = 4n-3 < 4n) \geq \frac{1}{4}.$$

Tedy algoritmus volí k náhodně s rovnoměrným rozdělením a volbu opakuje, dokud nedostane požadovanou funkci. Ověření, že pro zvolenou funkci h_k platí $\sum_{i=0}^{m-1}(b_i^k)^2 < 4n$, vyžaduje čas $O(n)$, a protože pravděpodobnost, že tato nerovnost není splněna, je menší než $\frac{3}{4}$, tak očekávaný počet iterací je omezen shora odhadem

$$\sum_{i=1}^{\infty} i(\frac{3}{4})^{i-1}\frac{1}{4} = \frac{1}{4}(\sum_{i=1}^{\infty} i(\frac{3}{4})^{i-1}) = \frac{1}{4}\frac{1}{(1-\frac{3}{4})^2} = 4.$$

Zde jsme použili známý vztah pro mocniné řady, že pro $|x| < 1$ platí

$$\sum_{i=1}^{\infty} ix^{i-1} = (\sum_{i=1}^{\infty} x^i)' = (\frac{x}{1-x})' = \frac{1}{(1-x)^2}.$$

Protože očekávaný čas algoritmu je součin času, který potřebuje iterace, s očekávaným počtem iterací, je tvrzení b) dokázáno.

Když $m = n(n-1)+1$, pak systematickým prohledáním všech možností nalezneme v čase $O(nN)$ takové k, že

$$\sum_{i=0}^{m-1}(b_i^k)^2 \leq \frac{2n(n-1)}{n(n-1)+1} + n < n+2,$$

a c) plyne z předchozí věty. Když $m = 2n(n-1)$, pak pro náhodně zvolené k s pravděpodobností alespoň $\frac{1}{4}$ platí, že $\sum_{i=0}^{m-1}(b_i^k)^2 \leq \frac{3n(n-1)}{2n(n-1)} + n < n + 2$. Tedy algoritmus postupuje stejně jako v případě b). Volí k náhodně s rovnoměrným rozdělením, dokud nezíská perfektní hašovací funkci. Odhad počtu iterací je stejný jako v případě b). Nyní d) plyne z předchozí věty. □

Takto zkonstruované perfektní hašovací funkce nesplňují požadavek 2), protože $m = \Theta(n^2)$. Tento nedostatek odstraníme následujícím postupem:

1) Nalezneme k takové, že pro $m = n$ platí $\sum_{i=0}^{m-1}(b_i^k)^2 < 3n$ (v pravděpodobnostním případě $\sum_{i=0}^{m-1}(b_i^k)^2 < 4n$). Pro $i = 0, 1, \ldots, m-1$ nalezneme množiny $S_i = \{s \in S \mid h_k(s) = i\}$.
2) Pro každé $i = 0, 1 \ldots, m-1$ takové, že $S_i \neq \emptyset$, nalezneme pro $c_i = |S_i|(|S_i| - 1) + 1$ (v pravděpodobnostním případě $c_i = 2|S_i|(|S_i| - 1)$) takové k_i, že h_{k_i} je perfektní funkce pro S_i do tabulky velikosti c_i. Položme $c_i = 0$, když $S_i = \emptyset$.
3) Pro $i = 0, 1, \ldots, m$ definujeme $d_i = \sum_{j=0}^{i-1} c_j$ a pro $x \in U$ označíme $h_k(x) = l$. Položíme $g(x) = d_l + h_{k_l}(x)$.

Věta. *Zkonstruovaná funkce g je perfektní a hodnota $g(x)$ se pro každé $x \in U$ spočítá v čase $O(1)$. V deterministickém případě funkce hašuje do tabulky velikosti $< 3n$ a je nalezena v čase $O(nN)$, v pravděpodobnostním případě hašuje do tabulky velikosti $< 6n$ a je nalezena v čase $O(n)$. Pro její zakódování jsou potřeba hodnoty k a k_i pro $i = 0, 1, \ldots, m-1$. Ty jsou v rozmezí $1, 2, \ldots, N-1$, a tedy vyžadují $O(n \log N)$ paměti.*

Důkaz. Protože $g(S_i)$ jsou pro $i = 0, 1, \ldots, m-1$ navzájem disjunktní a h_{k_i} je perfektní na S_i, dostáváme, že g je perfektní. Pro výpočet hodnoty $g(x)$ jsou zapotřebí dvě násobení, dvojí výpočet zbytku při dělení a jedno sčítání (hodnoty d_i jsou uloženy v paměti). Proto výpočet $g(x)$ vyžaduje čas $O(1)$. Dále d_m je horní odhad na počet řádků v tabulce. Protože pro $S_i \neq \emptyset$ máme $|S_i|(|S_i| - 1) + 1 \leq |S_i|^2 = (b_i^k)^2$, dostáváme v deterministickém případě $d_m = \sum_{i=0}^{m-1} c_i \leq \sum_{i=0}^{m-1}(b_i^k)^2 < 3n$ a k nalezneme v čase $O(nN)$. Protože každé k_i nalezneme v čase $O(|S_i|N)$, lze g zkonstruovat v čase

$$O(nN + \sum_{i=0}^{m-1} |S_i|N) = O(nN + N \sum_{i=0}^{m-1} |S_i|) = O(2nN) = O(nN).$$

V pravděpodobnostním případě je

$$d_m = \sum_{i=0}^{m-1} c_i = 2(\sum_{i=0}^{m-1}(b_i^k)^2 - \sum_{i=0}^{m-1}(b_i^k)) < 2(4n - n) = 6n$$

(zde jsme použili, že $|S_i| = b_i^k$, $c_i = 2(|S_i|^2 - |S_i|)$ a $\sum_{i=0}^{m-1} b_i^k = n$), k nalezneme v čase $O(n)$ a k_i v čase $O(|S_i|)$. Protože počet iterací je nejvýše 4 a protože očekávaná hodnota součtu je součet očekávaných hodnot, dostáváme, že g nalezneme v čase $O(n)$. Zbytek je jasný. □

Poznámka: Pravděpodobnostní verze této metody byla použita při hledání 'perfektního hašování', které by umožňovalo operace **INSERT** a **DELETE**. Tato metoda bude orientačně popsána v dalším textu.

Zkonstruovaná hašovací funkce tedy splňuje požadavky 1), 2) a 3), ale nikoli požadavek 4). Tento nedostatek se nyní budeme snažit odstranit.

Mějme přirozené číslo m a nechť q je počet všech prvočísel, která dělí m ($p_1, p_2, \ldots$ je rostoucí posloupnost všech prvočísel). Pak

$$m \geq \prod_{i=1}^{q} p_i > q! = e^{\sum_{i=1}^{q} \ln i} \geq e^{\int_1^q \ln x dx} = e^{q\ln(\frac{q}{e})+1} \geq (\frac{q}{e})^q.$$

Zlogaritmováním dostaneme $\ln m > q\ln(q) - q$, a tedy pro vhodné $c > 0$ platí $\ln m \geq cq \ln q$. Protože $x \ln x$ je spojitá rostoucí funkce, existuje $q_1 \geq q$ takové, že $\ln m = cq_1 \ln q_1 \geq cq \ln q$. Nyní $\ln\ln m = \ln c + \ln q_1 + \ln\ln q_1 \leq \ln c + 2\ln q_1$, a tedy $q \leq q_1 = \frac{\ln m}{c \ln q_1} \leq \frac{2\ln m}{c(\ln\ln m - \ln c)}$. Proto existuje konstanta c taková, že $q \leq c\frac{\ln m}{\ln\ln m}$. Platí tedy:

Věta. *Nechť $\delta(m)$ =počet prvočísel, která dělí m. Pak $\delta(m) = O(\frac{\log m}{\log\log m})$.*

Mějme $S = \{s_1 < s_2 < \cdots < s_n\} \subseteq U$. Označme $d_{i,j} = s_j - s_i$ pro $1 \leq i < j \leq n$. Pak $s_i \bmod p \neq s_j \bmod p$, právě když $d_{i,j} \neq 0 \bmod p$. Dále označme $D = \prod_{1\leq i<j\leq n} d_{i,j} \leq N^{(n^2)}$. Pak počet prvočíselných dělitelů čísla D je nejvýše $c\frac{\ln D}{\ln\ln D}$, a tedy mezi prvními $1 + c\frac{\ln D}{\ln\ln D}$ prvočísly existuje prvočíslo p takové, že $s_i \bmod p \neq s_j \bmod p$ pro každé $1 \leq i < j \leq n$ a mezi prvními $2c\frac{\ln D}{\ln\ln D}$ přirozenými čísly má alespoň polovina prvočísel tuto vlastnost. To znamená, že funkce $\phi_p(x) = x \bmod p$ je perfektní pro S. Podle věty o rozložení prvočísel platí $p_t \leq 2t\ln t$ pro každé $t > 2$ (víme, že $p_t \leq t(\ln t + \ln\ln t)$ pro $t \geq 6$, protože $\ln t \geq \ln\ln t$ pro $t \geq 6$, tak vztah platí pro $t \geq 6$, pro $p_3 = 5$, $p_4 = 7$ a $p_5 = 11$ se požadovaný vztah ověří přímým výpočtem, pro $p_1 = 2$ a $p_2 = 3$ vztah neplatí). Tedy

$$\begin{aligned} p \leq & 2(1 + c\frac{\ln D}{\ln\ln D})\ln(1 + c\frac{\ln D}{\ln\ln D}) \leq 4c\frac{\ln D}{\ln\ln D}\ln(2c\frac{\ln D}{\ln\ln D}) \leq \\ & 4c\ln 2c\frac{\ln D}{\ln\ln D} + 4c\frac{\ln D}{\ln\ln D}\ln(\frac{\ln D}{\ln\ln D}) = O(\ln D) = O(n^2 \ln N). \end{aligned}$$

Shrňme získaná fakta.

Věta. *Pro každou n-prvkovou množinu $S \subseteq U$ existuje prvočíslo p o velikosti $O(n^2 \ln N)$ takové, že funkce $\phi_p(x) = x \bmod p$ je perfektní pro S.*

Test, zda funkce $\phi_p(x) = x \bmod p$ je perfektní pro S, vyžaduje čas $O(n \log n)$ (setříděním všech $x_i \bmod p$, kde $x_i \in S$). Tedy systematické hledání nejmenšího p, pro které ϕ_p je perfektní pro S, vyžaduje čas $O(n^3 \log n \log N)$. Toto nejmenší p je prvočíslo. Navrhneme pravděpodobnostní algoritmus pro jeho nalezení.

Mezi prvními $4cn^2 \ln N$ přirozenými čísly je alespoň polovina takových prvočísel p, že ϕ_p je perfektní pro S. Algoritmus pak opakuje následující krok, dokud nenalezne perfektní funkci:

> vyberme náhodně číslo p o velikosti nejvýše $4cn^2 \ln N$ a otestujme, zda p je prvočíslo a zda ϕ_p je perfektní.

Odhadneme očekávaný počet neúspěšných kroků.
Náhodně zvolené číslo $p \leq 4cn^2 \ln N$ je prvočíslo s pravděpodobností $\Theta(\frac{1}{\ln(4cn^2 \ln N)})$ a

pro prvočíslo p je ϕ_p perfektní s pravděpodobností $\geq \frac{1}{2}$. Tedy náhodně zvolené číslo $p \leq 4cn^2 \ln N$ splňuje test s pravděpodobností $\Theta(\frac{1}{\ln(4cn^2 \ln N)})$, a proto očekávaný počet neúspěšných testů je $O(\ln(4cn^2 \ln N)$. Pro test, zda p je prvočíslo, můžeme použít buď Eratostenovo síto nebo pravděpodobnostní Rabin-Millerův algoritmus (jeho očekávaný čas je $O(\log^3 p)$). Tedy očekávaný čas algoritmu je $O(n \log n(\log n + \log\log N))$.

Věta. *Pro danou množinu $S \subseteq U$ takovou, že $|S| = n$, nalezne deterministický algoritmus prvočíslo $p = O(n^2 \log N)$ takové, že $\phi_p(x) = x \bmod p$ je perfektní pro S, v čase $O(n^3 \log n \log N)$. Pravděpodobnostní algoritmus nalezne prvočíslo $p = O(n^2 \log N)$ takové, že ϕ_p je perfektní, v očekávaném čase $O(n \log n(\log n + \log\log N)$.*

Deterministický algoritmus nalezne nejmenší prvočíslo s požadovanou vlastností. Pravděpodobnostní algoritmus nalezne prvočíslo, které může být podstatně větší, ale jehož velikost je omezena $4cn^2 \log N$ (pro deterministický algoritmus by byla konstanta přibližně poloviční).

Nyní navrhneme postup na konstrukci perfektní hašovací funkce pro množinu $S \subseteq U$.

1) Nalezneme prvočíslo $q_0 \in O(n^2 \log N)$ takové, že $\phi_{q_0}(x) = x \bmod q_0$ je perfektní funkce pro S. Položíme $S_1 = \{\phi_{q_0}(s) \mid s \in S\}$.
2) Nalezneme prvočíslo q_1 takové, že $n(n-1) < q_1 \leq 2n(n-1) + 2$. Pak existuje $l \in \{1, 2, \ldots, q_0 - 1\}$ tak, že $h_l(x) = ((lx) \bmod q_0) \bmod q_1$ je perfektní pro $S_1 \subseteq \{0, 1, \ldots, q_0 - 1\}$. Položíme $S_2 = \{h_l(s) \mid s \in S_1\}$.
3) Podle předchozí věty zkonstruujeme perfektní hašovací funkci g pro podmnožinu S_2 univerza $\{0, 1, \ldots, q_1 - 1\}$ do tabulky s méně než $3n$ řádky. Položíme $f(x) = g(h_l(\phi_{q_0}(x)))$. Výsledná perfektní hašovací funkce je f.

Ověření: f je perfektní hašovací funkce pro S, protože složení perfektních hašovacích funkcí je opět perfektní funkce, a tedy požadavek 1) je splněn.
f hašuje S do tabulky s méně než $3n$ řádky, a tedy splňuje požadavek 2).
Protože každá z funkcí g, h_l, ϕ_{q_0} se vyčíslí v čase $O(1)$, i vyčíslení funkce f vyžaduje čas $O(1)$ a požadavek 3) je splněn.
Funkce ϕ_{q_0} je jednoznačně určena číslem $q_0 \in O(n^2 \log N)$. Funkce h_l je určena čísly $q_1 \in O(n^2)$ a $l \in O(q_0)$. Funkce g je určena $n + 1$ čísly velikosti $O(q_1)$. Tedy zadání f vyžaduje paměť o velikosti

$$O(\log n + \log\log N + n \log n) = O(n \log n + \log\log N).$$

Lze říci, že požadavek 4) je také splněn.

Nalezení ϕ_{q_0} vyžaduje čas $O(n^3 \log n \log N)$. Nalezení h_l vyžaduje čas $O(n(n^2 \log N)) = O(n^3 \log N)$ (použité univerzum je $\{0, 1, \ldots, q_0\}$). Nalezení g vyžaduje čas $O(nn^2) = O(n^3)$ (zde univerzum je $\{0, 1, \ldots, q_1\}$). Celkově výpočet f vyžaduje čas $O(n^3 \log n \log N)$.

Lze použít i pravděpodobnostní algoritmy pro nalezení g, h_l a ϕ_{q_0}. Pak hašujeme do tabulky s méně než $6n$ řádky, ale očekávaný čas pro nalezení f je $O(n \log n(\log n + \log\log N))$.

Tuto metodu navrhli Fredman, Komlós a Szemerédi.

Dynamické perfektní hašování

Jedna z velkých nevýhod perfektního hašovaní je nemožnost efektivních aktualizačních operací. Existují sice obecné metody na dynamizaci deterministických operací (viz pokračování přednášky v letním semrstru), ale ty v tomto případě neposkytují efektivní dynamizační operace, protože deterministický algoritmus pro řešení perfektního hašování je pro aktualizační operace příliš pomalý. To vedlo k návrhu, který kombinuje pravděpodobnostní algoritmus pro perfektní hašování s obecnou metodou dynamizace a tyto metody jsou upraveny pro konkrétní situaci perfektního hašování. Výsledkem je metoda, která se pro operaci **MEMBER** chová jako perfektní hašování a umožňuje operace **INSERT** a **DELETE** s malou očekávanou amortizovanou složitostí.

Nejprve popíšeme reprezentaci, která je založena na modifikaci předchozích výsledků. Předpokládáme, že univerzum má tvar $U = \{0, 1, \ldots, N-1\}$, kde N je prvočíslo. Pro libovolné číslo $t = 1, 2, \ldots, N-1$, označme $\mathcal{H}_t$ množinu funkcí z univerza U do množiny $\{0, 1, \ldots, t-1\}$ tvaru $h_k(x) = (kx \bmod N) \bmod t$ pro $k = 1, 2, \ldots, N-1$. Reprezentace je určena číslem s, které budeme specifikovat později. Předpokládejme, že máme reprezentovat n-prvkovou množinu $S \subseteq U$. Jednoduchá modifikace předchozích výsledků říká, že když volíme náhodně $k = 1, 2, \ldots, N-1$ s rovnoměrným rozdělením, pak funkce $h_k \in \mathcal{H}_s$ s pravděpodobností alespoň $\frac{1}{2}$ splňuje

$$\sum_{i=0}^{s-1} (b_i^k)^2 < \frac{8n^2}{s} + 2n. \tag{1}$$

Budeme předpokládat, že máme takové k, že (1) platí. Pro každé $i = 0, 1, \ldots, s-1$ označme $S_i = \{t \in S \mid h_k(t) = i\}$. Když volíme náhodně $j(i)$ z množiny $\{1, 2, \ldots, N-1\}$, pak s pravděpodobností alespoň $\frac{1}{2}$ je funkce $h_{j(i)} \in \mathcal{H}_{2(b_i^k)^2}$ perfektní na množině S_i (viz předchozí postup). Předpokládejme, že pro každé $i = 0, 1, \ldots, s-1$ takové $j(i)$ máme. Narozdíl od předchozí metody bude nyní každá množina S_i reprezentována pomocí funkce $h_{j(i)}$ ve své tabulce T_i pro $i = 0, 1, \ldots, s-1$ a tabulky budou uloženy v seznamu T. Důvod je, že když si aktualizační operace **INSERT** nebo **DELETE** vynutí opravu tabulky T_i, tak se nemusí pracovat s hodnotami v jiných tabulkách. Proto jsou tabulky nezávisle uloženy. Když vybrané číslo splňuje $s = O(|S|)$, pak tato metoda vyžaduje $O(|S|)$ paměti. Zbývá určit číslo s. Zafixujeme $c > 1$ (jeho velikost by se měla určit podle zkušeností s řešenou úlohou) a pak položíme $s = \sigma(|S|)$, kde $\sigma(n) = \frac{4}{3}\sqrt{6}(1+c)n$ pro každé n. Nyní popíšeme algoritmy.

Algoritmy

Neformální popis algoritmů:
Algoritmus pro operaci **MEMBER** je jasný. Algoritmus pro operaci **INSERT**(x) nejprve zjistí, zda $x \in S$. Když $x \notin S$, pak pomocí daného k určí, do které tabulky T_i se x má uložit (má být v tabulce T_i, kde $i = (kx \bmod N) \bmod s$), a zkusí ho uložit. Když po vložení x je hašovací funkce stále perfektní, algoritmus skončí. V opačném případě zkontroluje, zda k splňuje podmínku (1). Pokud ano, spočítá novou perfektní hašovací funkci pro novou množinu S_i a vytvoří tabulku T_i. Když k nesplňuje podmínku (1), tak pomocná podprocedura **RehashAll** zkonstruuje celou novou reprezentaci množiny $S \cup \{x\}$.

Tato podprocedura nejprve spočítá novou hodnotu s, pak nalezne k splňující podmínku (1). Pomocí $h_k \in \mathcal{H}_s$ rozdělí reprezentovanou množinu do jednotlivých množin S_i. Pro každé $i = 0, 1, \ldots, s-1$ nalezne $j(i)$ takové, že $h_{j(i)} \in \mathcal{H}_{2(|S_i|^2)}$ je perfektní na množině S_i, a pomocí této hašovací funkce vytvoří tabulku T_i pro množinu S_i. Operace **DELETE**(x) po případném odstranění prvku x zjistí, zda je splněna podmínka (1). Pokud ne, zavolá podproceduru **RehashAll**. Hledání každé hašovací funkce používá pravděpodobnostní postup. To znamená, že jsou náhodně voleny hodnoty k a $j(i)$, a tato volba se opakuje do té doby, než jsou splněny požadavky na k nebo $j(i)$. V následujícím algoritmu proměnné m, s a $m(i)$ pro $i = 0, 1, \ldots, s$ nastavuje podprocedura **RehashAll**. Platí, že $m = (1+c)|S|$, $s = \sigma(m)$, $m(i) = |S_i|$, kde hodnoty $|S|$ a $|S_i|$ jsou aktuální v okamžiku volaní **RehashAll**. Aktuální velikost S je v proměnné n.

Formální popis algoritmů.

MEMBER(x):
$i := h_k(x)$
if x je na $h_{j(i)}(x)$-té pozici v tabulce T_i **then**
  Výstup: $x \in S$
else
  Výstup: $x \notin S$
endif

INSERT(x):
if $x \in S$ **then** stop **endif**
$n := n + 1$
if $n \leq m$ **then**
  $i := h_k(x)$, $|S_i| := |S_i| + 1$
  if $|S_i| \leq m(i)$ a pozice $h_{j(i)}(x)$ v T_i je prázdná **then**
    vlož x do tabulky T_i na pozici $h_{j(i)}(x)$
  else
    if $|S_i| \leq m(i)$ a pozice $h_{j(i)}(x)$ v T_i je obsazená **then**
      vytvoř seznam S_i prvků z tabulky T_i spolu s x
      vyprázdni tabulku T_i
      repeat zvol náhodně funkci $h_{j(i)} \in \mathcal{H}_{2m(i)^2}$
      until $h_{j(i)}$ není prostá na množině S_i
      for every $y \in S_i$ **do** vlož y do T_i na pozici $h_{j(i)}(y)$ **enddo**
    else
      $m(i) := 2m(i)$
      if není dost prostoru pro tabulku T_i nebo

$$\sum_{i=0}^{\sigma(m)-1} 2(m(i))^2 \geq \frac{8m^2}{\sigma(m)} + 2m$$

      then
        RehashAll
      else

alokuj prostor pro novou prázdnou tabulku T_i
vytvoř seznam S_i prvků ze staré tabulky T_i a zruš ji
repeat zvol náhodně funkci $h_{j(i)} \in \mathcal{H}_{2m(i)^2}$
until $h_{j(i)}$ není prostá na množině S_i
for every $y \in S_i$ **do** vlož y do T_i na pozici $h_{j(i)}(y)$ **enddo**
endif
endif
endif
else
RehashAll
endif

RehashAll:
projdi tabulku T a tabulky T_i a vytvoř seznam prvků z množiny S
$m := (1+c)|S|$
repeat zvol náhodně $h_k \in \mathcal{H}_{\sigma(m)}$
for every $i = 0, 1, \ldots, \sigma(m)-1$ **do**
$S_i := \{x \in S \mid h_k(x) = i\}$
enddo
until $\sum_{i=0}^{\sigma(m)-1} 2(b_i^k)^2 < \frac{8m^2}{\sigma(m)} + 2m$
Komentář: zde b_i^k jsou hodnoty vzhledem k náhodně zvolené funkci h_k
$n := 0$
for every $i = 0, 1, \ldots, \sigma(m)-1$ **do**
$m(i) := |S_i|$, $S_i := \emptyset$,
$h_{j(i)} \in \mathcal{H}_{2m(i)^2}$ je náhodně zvolená funkce
enddo
for every $x \in S$ **do INSERT**(x) **enddo**

DELETE(x):
if $\mathbf{x} \notin S$ **then** stop **endif**
$i := h_k(x)$, $n := n-1$, $|S_j| := |S_j| - 1$
odstraň x z pozice $h_{j(i)}(x)$ v tabulce T_i, pozice bude prázdná
if $n < \frac{m}{1+2c}$ **then RehashAll endif**

V prezentované verzi algoritmů **INSERT** a **DELETE** znamená test, zda $x \in S$, provedení operace **MEMBER**(x), i když to tak není zapsáno. Upravit algoritmy na standardní tvar je jednoduchá technická záležitost.

Složitost

Uvedeme složitost této metody bez důkazu.

Věta. *Popsaná metoda vyžaduje lineární velikost paměti (neuvažujeme paměť pro zakódování hašovacích funkcí), operace* **MEMBER** *v nejhorším případě vyžaduje čas* $O(1)$ *a očekávaná amortizovaná složitost operací* **INSERT** *a* **DELETE** *je také* $O(1)$.

Toto zobecnění metody Fredmana, Komlóse a Szemerédiho navrhli Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert a Tarjan.

Další nevýhoda metody Fredmana, Komlóse a Szemerédiho je v tom, že se hašuje do tabulky velikosti $m < 3n$, nikoli do tabulky velikosti n. Vznikla otázka, zda pro množiny o velikosti n nelze najít přirozené hašovací funkce, které by hašovaly do tabulky velikosti n. Z výsledků pro (N, m, n)-perfektní soubory funkcí plyne existence (N, n, n)-perfektního souboru pro $n^N > e^{n+ln(n)} \ln(N)$. Zmíníme se jen orientačně o parametrizované metodě, která navrhuje perfektní hašovací funkci pro množinu $S \subseteq U$ do tabulky velikosti $m = |S|$. Parametrem bude přirozené číslo r, které určuje, jaké hypergrafy jsou užity při konstrukci funkce. Proto nejprve připomeneme několik definic.

Dvojice (X, E), kde X je množina a E je systém r-prvkových podmnožin X, se nazývá r-hypergraf. Prvky v E se nazývají hrany r-hypergrafu. Cyklus je hypergraf (X, E), kde každý vrchol leží alespoň ve dvou různých hranách. Naopak r-hypergraf (X, E) se nazývá acyklický, když žádný jeho podhypergraf není cyklus.

Nyní popíšeme metodu, která je rozdělena do dvou kroků. Je dáno $S \subseteq U$ takové, že $|S| = n$.

Krok 1):
Mějme r-hypergraf (V, E), kde $|E| = n$. Nalezneme zobrazení $g : V \to \{0, 1, \ldots, m-1\}$ takové, že funkce $h : E \to \{0, 1, \ldots, m-1\}$ definovaná pro $e = \{v_1, v_2, \ldots, v_r\}$ vztahem $h(e) = \sum_{i=1}^{r} g(v_i) \bmod m$, je prostá (místo sčítání modulo m můžeme použít libovolnou grupovou operaci na množině $\{0, 1, \ldots, m-1\}$). Pro acyklický r-hypergraf lze funkci g zkonstruovat následujícím postupem. Zvolíme bijekci $h : E \to \{0, 1, \ldots, m-1\}$ a pak definujeme g následovně: když $e = \{v_1, v_2, \ldots, v_r\}$ a $g(v_i)$ je definováno pro $i = 2, 3, \ldots, r$, pak $g(v_1) = h(e) - \sum_{i=2}^{r} g(v_i) \bmod m$. Protože pro každý acyklický r-hypergraf existuje vrchol, který leží v jediné hraně, lze tento postup použít ke konstrukci g pomocí indukce (a tedy máme algoritmus pro konstrukci g).
Krok 2):
Nalezneme r funkcí $f_1, f_2, \ldots, f_r : U \to V$ takových, že (V, E), kde

$$E = \{\{f_1(x), f_2(x), \ldots, f_r(x)\} \mid x \in S\},$$

je acyklický r-hypergraf. Pak hašovací funkce f je definována jako $f(x) = \sum_{i=1}^{r} g f_i(x)$ pro každé $x \in U$. Z konstrukce vyplývá, že je perfektní na množině S.

Autoři dokázali, že nejvhodnější alternativa je, když zobrazení $f_1, f_2, \ldots, f_r$ jsou náhodná zobrazení náhodně zvolená. Bohužel taková zobrazení neumíme zkonstruovat, ale autoři ukázali, že pro tyto účely lze použít náhodný výběr funkcí z nějakého c-univerzálního souboru funkcí.

Dále dokázali, že jejich algoritmus vyžaduje $O(rn + |V|)$ času a $O(n \log n + r \log |V|)$ paměti.

Tento metapostup navrhli Majewski, Wormald, Havas a Czech v roce 1996.

Pro praktické použití je problematická reprezentace r-hypergrafu a i náhodná volba funkcí $f_1, f_2, \ldots, f_r$ (viz předchozí diskuze o c-univerzalitě). Z požadavků na perfektní hašovací funkci je opět problémem splnění požadavku 4). Nevím, jak je uvedená metoda prakticky použitelná a zda se někde používá.

XII. Externí hašování

Poslední problém spojený s hašováním je odlišného charakteru. Chceme uložit data na externí medium a protože přístup k externím mediím je o několik řádů pomalejší než práce v interní paměti, bude naším problémem minimalizovat počet komunikací s externí pamětí.

Popíšeme naši úlohu podrobněji. Externí paměť je rozdělena na stránky, každá stránka obsahuje b prvků (předpokládáme, že $b > 1$, jinak to nemá smysl). Vždy v jednom kroku načteme celou stránku do interní paměti nebo celou stránku z interní paměti v jednom kroku zapíšeme na externí medium. Tyto operace jsou řádově pomalejší než operace v interní paměti, a proto je chceme minimalizovat. Jinými slovy, náš cíl je nalézt datovou strukturu na externí paměti a algoritmy pro operace **MEMBER**, **INSERT** a **DELETE**, které by použily pokud možno co nejmenší počet komunikací mezi interní a externí pamětí.

Předpokládejme, že $h : U \to \{0,1\}^*$ je prostá funkce taková, že délka $h(u)$ je stejná pro všechny prvky univerza U. Označme k délku $h(u)$ pro $u \in U$. Pak h je hašovací funkce pro naši úlohu, ale bude hrát jinou roli než v klasickém hašování. Předpokládejme, že $S \subseteq U$ je množina, kterou chceme uložit v externí paměti. Pro slovo α délky menší než k definujme $h_S^{-1}(\alpha) = \{s \in S \mid \alpha \text{ je prefix } h(s)\}$. Řekneme, že α je kritické slovo, když $0 < |h_S^{-1}(\alpha)| \leq b$ a pro každý vlastní prefix α' slova α platí $|h_S^{-1}(\alpha')| > b$. Pro každé $s \in S$ existuje právě jedno kritické slovo α, které je prefixem $h(s)$. Definujme $d(s)$ pro $s \in S$ jako délku kritického slova, které je prefixem $h(s)$, a

$$d(S) = \max\{\text{délka}(\alpha) \mid \alpha \text{ je kritické slovo}\} = \max\{d(s) \mid s \in S\}.$$

Množinu S reprezentujeme tak, že je jednoznačná korespondence mezi kritickými slovy a stránkami externí paměti sloužícími k reprezentaci S. Na stránce příslušející kritickému slovu α je reprezentován soubor $h_S^{-1}(\alpha)$.

První problém je, jak nalézt stránku odpovídající kritickému slovu α. K tomu použijeme pomocnou strukturu nazývanou adresář. Adresář je funkce, která každému slovu α o délce $d(S)$ přiřadí adresu stránky předpisem:

> když kritické slovo β je prefixem α, pak k α je přiřazena stránka korespondující s β, jinak je k α přiřazena stránka NIL – název prázdné stránky.

Abychom se přesvědčili o korektnosti tohoto přiřazení, všimněme si, že pro různá kritická slova β a γ platí $h_S^{-1}(\beta) \cap h_S^{-1}(\gamma) = \emptyset$. Tedy pro každé slovo α délky $d(S)$ existuje nejvýše jedno kritické slovo, které je prefixem α. Když α je slovo délky $d(S)$, pak nastane jeden z těchto tří případů:

(1) $h_S^{-1}(\alpha) \neq \emptyset$, pak $0 < |h_S^{-1}(\alpha)| \leq b$ a existuje právě jedno kritické slovo β, které je prefixem α;
(2) $h_S^{-1}(\alpha) = \emptyset$ a existuje prefix α' slova α takový, že $0 < |h_S^{-1}(\alpha')| \leq b$, pak existuje právě jedno kritické slovo, které je prefixem α' (a tedy také prefixem α);
(3) $h_S^{-1}(\alpha) = \emptyset$ a pro každý prefix α' slova α platí buď $h_S^{-1}(\alpha') = \emptyset$ nebo $|h_S^{-1}(\alpha')| > b$, pak k α je přiřazena stránka NIL.

Mějme slovo α o délce $d(S)$. Označme $c(\alpha)$ nejkratší prefix α' slova α takový, že stránka přiřazená slovu β o délce $d(S)$, které má α' za prefix, je stejná jako stránka přiřazená k α. Všimněme si, že když $h_S^{-1}(\alpha) \neq \emptyset$, pak $c(\alpha)$ je kritické slovo. Platí silnější tvrzení, které tvrdí, že následující podmínky jsou ekvivalentní:

(1) stránka přiřazená slovu α je různá od NIL;

(2) $c(\alpha)$ je kritické slovo;
(3) nějaký prefix α je kritické slovo.

Dále si všimněme, že znalost adresáře umožňuje nalézt slovo $c(\alpha)$ pro každé slovo o délce $d(S)$.

Lineární uspořádání na slovech délky n nazveme lexikografické, když $\alpha < \beta$, právě když $\alpha = \gamma 0 \alpha'$ a $\beta = \gamma 1 \beta'$ pro nějaká slova γ, α' a β'. Lexikografické uspořádání vždy existuje a je jednoznačné.

Dalším problémem je, jak reprezentovat adresář. Zřejmě si ho můžeme představit (a i reprezentovat) jako seznam adres stránek o délce $2^{d(S)}$ takový, že adresa na i-tém místě je adresa stránky odpovídající i-tému slovu délky $d(S)$ v lexikografickém uspořádání. NIL považujeme za adresu fiktivní prázdné stránky.

Příklad: U je množina všech slov nad $\{0,1\}$ o délce 5, h je identická funkce a $b = 2$. Reprezentujme množinu $S = \{00000, 00010, 01000, 10000\}$. Pak $d(00000) = d(00010) = d(01000) = 2$, $d(10000) = 1$, kritická slova jsou 00, 01 a 1 a adresář je

$$00 \mapsto \{00000, 00010\}, \quad 01 \mapsto \{01000\}, \quad 10 \mapsto 11 \mapsto \{10000\}$$

(místo adresy stránky uvádíme množinu, která je na této stránce uložena). Tedy $c(00) = 00$, $c(01) = 01$ a $c(10) = c(11) = 1$. Když odstraníme prvek 10000, pak 1 přestane být kritickým slovem a adresář bude mít tvar

$$00 \mapsto \{00000, 00010\}, \quad 01 \mapsto \{01000\}, \quad 10 \mapsto 11 \mapsto \text{NIL}\,.$$

Opět platí $c(00) = 00$, $c(01) = 01$ a $c(10) = c(11) = 1$. V adresáři je také uloženo $d(S)$.

Algoritmy

Nyní slovně popíšeme algoritmy realizující operace **MEMBER**, **INSERT** a **DELETE**. Předpokládáme, že adresář je uložen také v externí paměti na jedné stránce. V algoritmech je každá akce spojená s externí pamětí vytištěna tučným písmem.

MEMBER(x):
1) Vypočteme $h(x)$ a **načteme** adresář do interní paměti. Vezmeme prefix α slova $h(x)$ o délce $d(S)$ a nalezneme adresu stránky příslušející k α. Když je to stránka NIL, pak $x \notin S$ a konec, jinak pokračujeme krokem 2).
2) **Načteme** stránku příslušející k α do interní paměti. Prohledáme ji, a pokud neobsahuje x, pak $x \notin S$ a konec. Když obsahuje x, pak provedeme požadované změny a stránku **uložíme** do externí paměti na její původní místo. Konec.

INSERT(x):
1) Vypočteme $h(x)$ a **načteme** adresář do interní paměti. Vezmeme prefix α slova $h(x)$ o délce $d(S)$ a nalezneme adresu stránky příslušející k α a slovo $c(\alpha)$. Když stránka přiřazená k α je NIL, pokračujeme krokem 3), v opačném případě pokračujeme krokem 2).
2) **Načteme** stránku přiřazenou slovu α. Když x je uloženo na této stránce, pak skončíme. Když x není na této stránce, pak ho tam přidáme. Pokud je na stránce nejvýše b prvků,

uložíme stránku na její původní místo a skončíme. Když je na stránce více než b prvků, pak nalezneme nová kritická slova, která nám stránku rozdělí, a vytvoříme dvě stránky – jednu **uložíme** na místo původní stránky a druhou **uložíme** na novou stránku. Pokračujeme krokem 4).
3) Vytvoříme v interní paměti novou stránku, která obsahuje x, nalezneme novou stránku v externí paměti a tam **uložíme** vytvořenou stránku (všem slovům, která mají $c(\alpha)$ za prefix, bude přiřazena tato stránka) a pokračujeme krokem 4).
4) **Načteme** opět adresář do interní paměti, aktualizujeme adresy přiřazených stránek a případně adresář zvětšíme. (To nastane, když nějaké nové kritické slovo má délku větší než $d(S)$. Pak nové $d(S)$ je právě délka tohoto slova – obě kritická slova vzniklá v kroku 2) mají stejnou délku.) Aktualizovaný adresář **uložíme** do externí paměti. Konec.

DELETE(x):
1) Vypočteme $h(x)$ a **načteme** adresář do interní paměti. Vezmeme prefix α slova $h(x)$ o délce $d(S)$ a nalezneme adresu stránky příslušející k α a slovo $c(\alpha)$. Když stránka přiřazená k α je NIL, pak skončíme. Označme β' slovo, které má stejnou délku jako $c(\alpha)$ a liší se od $c(\alpha)$ pouze v posledním bitu. Když existuje slovo β délky $d(S)$ takové, že $c(\beta) = \beta'$, pak stránka přiřazená k β je kandidát na spojení stránek.
2) **Načteme** stránku příslušnou k slovu α do interní paměti. Když tato stránka neobsahuje x, pak skončíme. Když tato stránka obsahuje x, pak ho z této stránky odstraníme. Když neexistuje kandidát nebo když nová stránka a stránka kandidáta dohromady obsahují více než b prvků, pak načtenou stránku **uložíme** na její původní místo a skončíme.
3) Když nová stránka a stránka kandidáta mají dohromady b prvků, pak **načteme** stránku kandidáta do interní paměti. V interní paměti obě stránky spojíme do jedné a tuto stránku pak **uložíme** do externí paměti.
4) **Načteme** adresář, kde zaktualizujeme adresy stránek. Pokud jsme sloučili dvě stránky, musíme nalézt nové $c(\alpha)$ (je to nejkratší prefix α' slova α takový, že ke každému slovu β o délce $d(S)$, které má α' za prefix, je přiřazena jedna z těchto adres: adresa stránky přiřazená k α, adresa stránky kandidáta, NIL) a každému slovu o délce $d(S)$, které má nové $c(\alpha)$ za prefix, bude přiřazena adresa nové (spojené) stránky. Otestujeme, zda se adresář nemůže zkrátit. (To nastane, když adresy stránek přiřazené $(2i+1)$-ímu slovu a $(2i+2)$-ému slovu o délce $d(S)$ jsou stejné pro všechna i. Pak se tato slova spojí a $d(S)$ se zmenší o 1. Tento krok se může několikrát opakovat.) Upravený adresář **uložíme**. Konec.

Složitost

Následující věta ukazuje, že jsme náš hlavní cíl splnili.

Věta. *Operace* **MEMBER** *vyžaduje nejvýše tři operace s externí pamětí. Operace* **INSERT** *a* **DELETE** *vyžadují nejvýše šest operací s externí pamětí.*

V našem příkladu provedeme operaci **INSERT**(00001). Po přidání prvku stránka původně přiřazená k slovu 00 vypadá takto: $\{00000, 00001, 00010\}$. Tuto stránku rozdělíme na stránky $\{00000, 00001\}$ a $\{00010\}$. Přitom kritické slovo první stránky je 0000 a druhé stránky

0001. Takže $d(S) = 4$ a adresář vypadá následovně:

$$0000 \mapsto \{00000, 00001\}, 0001 \mapsto \{00010\}, 0010 \mapsto 0011 \mapsto \text{NIL},$$
$$0100 \mapsto 0101 \mapsto 0110 \mapsto 0111 \mapsto \{0100\},$$
$$1000 \mapsto 1001 \mapsto 1010 \mapsto 1011 \mapsto 1100 \mapsto 1101 \mapsto 1110 \mapsto 1111 \mapsto \{10000\}.$$

To znamená, že kromě adresy 00 se ostatní slova rozdělila na čtyři slova, ale adresy zůstaly stejné. Jen u slova 00 vzniklá slova dostala různé adresy.

V původním příkladu provedeme operaci **DELETE**(01000). Pak kandidát je 00 a po odstranění prvku 01000 nastane spojení těchto dvou stránek. Po aktualizaci adres dostane adresář tvar

$$00 \mapsto 01 \mapsto \{00000, 00010\}, 10 \mapsto 11 \mapsto \{10000\},$$

tj. k prvnímu a druhému slovu je přiřazena stejná stránka a stejně tak k třetímu a čtvrtému slovu. Takže můžeme adresář zmenšit. Pak bude $d(S) = 1$ a adresář bude mít podobu

$$0 \mapsto \{00000, 00010\}, 1 \mapsto \{10000\}.$$

Vzniká otázka, jak je tato metoda efektivní, hlavně jak efektivně využívá paměť. To popisuje následující věta, kterou uvádíme bez důkazu.

Věta. *Při reprezentaci množiny S o velikosti n je očekávaný počet použitých stránek $\frac{n}{b \ln 2}$ a očekávaná velikost adresáře je $\frac{e}{b \ln 2} n^{1+\frac{1}{b}}$.*

První část tvrzení říká, že očekávaný počet prvků na stránce je $b \ln 2 \approx 0.69b$. Tedy zaplněno je asi 69% míst. Tento výsledek není překvapující a je akceptovatelný. Horší je to s velikostí adresáře, jak ukazuje následující tabulka

velikost S	10^5	10^6	10^8	10^{10}
2	$6.2 \cdot 10^7$	$1.96 \cdot 10^8$	$1.96 \cdot 10^{11}$	$1.96 \cdot 10^{14}$
10	$1.2 \cdot 10^5$	$1.5 \cdot 10^6$	$2.4 \cdot 10^8$	$3.9 \cdot 10^{10}$
50	$9.8 \cdot 10^3$	$1.0 \cdot 10^6$	$1.1 \cdot 10^8$	$1.2 \cdot 10^{10}$
100	$4.4 \cdot 10^3$	$4.5 \cdot 10^4$	$4.7 \cdot 10^6$	$4.9 \cdot 10^8$

kde jednotlivé řádky odpovídají hodnotám b uvedeným v prvním sloupci. Protože očekávaná velikost adresáře se zvětšuje rychleji než lineárně (exponent u n je $1+\frac{1}{b}$), tak nelze očekávat, že tuto metodu lze vždy použít. Výpočty i experimenty ukazují, že použitelná je do velikosti $|S| = 10^{10}$, když $b \approx 100$. V tomto rozmezí je nárůst adresáře jen kolem 5%. Pro větší n je třeba, aby b bylo ještě větší.