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

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

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

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

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:

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

Pak můžeme počítat

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

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

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

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

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

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

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

$$(c-1)T_c^n = cT_c^n - T_c^n = \sum_{i=2}^{n+1}(i-1)c^i - \sum_{i=1}^{n} ic^i = nc^{n+1} + \big(\sum_{i=2}^{n}((i-1)c^i - ic^i)\big) - c =$$

$$nc^{n+1} + \big(\sum_{i=2}^{n} -c^i\big) - c =$$

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

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

$$S_n = (m+2)^{n-1} \frac{(n-1)\big(\frac{m}{m+2}\big)^{n+1} - n\big(\frac{m}{m+2}\big)^n + \frac{m}{m+2}}{\big(\frac{m}{m+2} - 1\big)^2} =$$

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

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

Očekávaný počet testů při neúspěšném vyhledávání je

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

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

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

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

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

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 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.

Datové struktury – stromy

Úvod

Hašování je velmi efektivní nástroj pro reprezentaci množin umožňující řešení klasického slovníkového problému. Nevýhodou této metody je ignorování případného uspořádání univerza. Tím se stávají problematickými operace, které jsou na tomto uspořádání založeny. Existují sice modifikace hašování, které využívají uspořádání univerza a umožňují provádět operace na něm založené, ale za cenu ztráty jednoduchosti hašování. Toto spolu se zjištěním, že hašování může velmi zklamat, když neznáme rozložení vstupních dat, vedlo v letech 1970–1975 k velkému zájmu o datové struktury založené na stromech. Zatímco hašování používá hlavně metody z teorie čísel, stromové struktury používají kombinatorické metody. Zde už není základní úlohou slovníkový problém, ale jeho rozšíření o několik operací založených na uspořádání univerza. Tuto úlohu nazýváme uspořádaný slovníkový problém. Popíšeme ho formálně.

Uspořádaný slovníkový problém

Je dáno totálně uspořádané univerzum U (tj. pro každé dva různé prvky $u, v \in U$ platí buď $u < v$ nebo $v < u$). Naším úkolem je reprezentovat množinu $S \subseteq U$ a navrhnout algoritmy pro následující operace:
MEMBER(x) – určí, zda $x \in S$, a pokud ano, vrátí adresu dat spojených s klíčem x;
INSERT(x) – zjistí, zda $x \in S$, a pokud $x \notin S$, rozšíří reprezentovanou množinu S o prvek x a vymezí prostor, kam uloží data spojená s x;
DELETE – zjistí, zda $x \in S$, a pokud ano, odstraní x z reprezentované množiny S a uvolní prostor, kde byla uložena data spojená s x;
MIN – nalezne nejmenší prvek v S;
MAX – nalezne nevětší prvek v S;
SPLIT(x) – zkonstruuje reprezentace dvou množin $S_1 = \{s \in S \mid s < x\}$ a $S_2 = \{s \in S \mid s > x\}$ a oznámí, zda $x \in S$;
JOIN – používají se dvě verze této operace:
JOIN2(S_1, S_2) – jsou-li dány reprezentace množin S_1 a S_2, které splňují $\max S_1 < \min S_2$, vytvoří reprezentaci množiny $S = S_1 \cup S_2$;
JOIN3(S_1, x, S_2) – jsou-li dány reprezentace množin S_1 a S_2 a prvek $x \in U$ tak, že je splněno $\max S_1 < x < \min S_2$, vytvoří reprezentaci množiny $S = S_1 \cup \{x\} \cup S_2$.

Je vidět, že operace **JOIN2** a **JOIN3** lze pomocí operací **INSERT** a **DELETE** převést jednu na druhou. Proto často budeme popisovat pro danou strukturu jen jednu z nich. Občas se také používá operace
ord(k) – nalezne k-tý nejmenší prvek v S (předpokádáme, že $k \leq |S|$).
Zřejmě operace **MIN** a **MAX** jsou speciálním případem operace **ord**(k), konkrétně **MIN** je operace **ord**(1) a **MAX** je operace **ord**$(|S|)$.

Při návrhu datové struktury a při popisu algoritmů se, stejně jako při hašování, omezíme na způsob uložení klíče a na práci s ním. Předpokládáme, že data jsou uložena v jiné části paměti a zde je na ně pouze ukazatel. Protože práce s daty spojenými s klíčem neovlivňuje navrženou datovou strukturu ani algoritmy s ní pracující, nezahrneme je do popisu struktury.

II. (a,b)-stromy

Důležitou datovou strukturou vhodnou pro řešení uspořádaného slovníkového problému, kterou lze použít jak pro interní tak pro externí paměť, jsou (a, b)-stromy.

Nejobecnější grafová definice (a, b)-stromu je:
Nechť $a \leq b$ jsou kladná přirozená čísla. Pak kořenový strom (T, t) se nazývá (a, b)-strom, když

(1) každý vnitřní vrchol v stromu T různý od kořene t má alespoň a a nejvýše b synů;
(2) všechny cesty z kořene do libovolného listu mají stejnou délku.

Tato definice je příliš obecná a pro naše účely se nehodí. Idea stromových datových struktur je totiž založena na paradigmatu 'rozděl a panuj' – podstrom určený nějakým vrcholem reprezentuje množinu a jeho synové reprezentují její disjunktní části. S vrcholem je spojen mechanismus, který nám umožní správně pokračovat (vybrat si správného syna). Z tohoto pohledu není vhodné, když vrchol má jen jediného syna. To vede k omezení na velikost parametru a. Technické požadavky algoritmů pro operace **INSERT** a **DELETE** pak určují omezení na velikost b. Dále chceme, abychom mohli reprezentovat množiny všech velikostí. Proto používáme jen speciální případ (a, b)-stromů, ktrý je definovaný takto:
Nechť a a b jsou přirozená čísla taková, že $2 \leq a$ a $2a - 1 \leq b$. Pak kořenový strom (T, t) se nazývá (a, b)-strom, když

(1) každý vnitřní vrchol v stromu T různý od kořene t má alespoň a a nejvýše b synů;
(2) kořen t je buď list nebo má alespoň dva syny a nejvýše b synů;
(3) všechny cesty z kořene do libovolného listu mají stejnou délku.

Základní vlastnost (a, b)-stromů, která motivovala jejich používání, je vyjádřena vzorcem spojujícím výšku (a, b)-stromu s počtem jeho listů. Připomínáme, že výška (a, b)-stromu je délka cesty z kořene do listu.

Tvrzení. *Mějme přirozená čísla a a b taková, že $a \geq 2$ a $b \geq 2a - 1$. Pak pro každé kladné přirozené číslo n existuje (a, b)-strom, který má přesně n listů. Když (a, b)-strom má přesně n listů, pak jeho výška je alespoň $\log_b n$ a nejvýše $1 + \log_a(\frac{n}{2})$. Tedy výška (a, b)-stromu je $O(\log n)$.*

Důkaz. Zřejmě strom o výšce 0 má jediný list. Strom o výšce 1 má alespoň dva a nejvýše b listů. Strom o výšce 2 má alespoň 2a listů a nejvýše b^2 listů. Indukcí podle výšky h dostaneme, že pro každé n takové, že $2a^{h-1} \leq n \leq b^h$, existuje (a, b)-strom o výšce h s n listy. Přesněji, když (a, b)-strom o výšce $h - 1$ má k listů, tak přidáním další (h-té) hladiny dostaneme (a, b)-strom o výšce h s n listy pro každé n takové, že $ka \leq n \leq kb$. Pro $k \geq 2$ a pro $b \geq 2a - 1$, $a \geq 2$ platí $kb \geq (k + 1)a$ a odtud dostáváme požadované tvrzení. Tím je první část věty dokázána. Druhou část dostaneme zlogaritmováním nerovnosti mezi počtem listů a výškou stromu. □

Nyní připomeneme několik pojmů, které jsou dále použity při definování reprezentace.

Mějme kořenový strom (T, t) takový, že pro každý vnitřní vrchol v platí:
Když v má $\rho(v)$ synů, pak jsou očíslovány od 1 do $\rho(v)$. Řekneme, že vrchol v je v hloubce h, když cesta z kořene t do v má délku h. Množina všech vrcholů v hloubce h se nazývá h-tá hladina. Lexikografické uspořádání na h-té hladině je definováno rekurzivně:

$v \leq w$, právě když buď otec$(v) <$ otec(w), nebo otec$(v) =$ otec(w) a když v je i-tý syn otec(v) a w je j-tý syn otec(v), pak $i \leq j$.

Protože synové každého vnitřního uzlu jsou uspořádány a listy tvoří hladinu h, kde h je hloubka (a, b)-stromu, je i na nich definováno lexikografické uspořádání.

Mějme lineárně uspořádané univerzum U a množinu $S \subseteq U$. (a, b)-strom (T, t) reprezentuje množinu S, když má přesně $|S|$ listů a je dán izomorfismus mezi lexikografickým uspořádáním listů stromu T a uspořádanou množinou S (tj. bijekce key : list$(T) \rightarrow S$, která pro $s, t \in S$ splňuje $s \leq t$ v U, právě když key$^{-1}(s) \leq$ key$^{-1}(t)$ v lexikografickém uspořádání na množině listů stromu T).

Struktura vnitřních vrcholů a listů (a, b)-stromu reprezentujícího množinu $S \subseteq U$ je následující.

Deklarace vnitřních vrcholů (a, b)-stromu (T, t):
$\rho(v)$ – počet synů vrcholu v,
$S_v(1..\rho(v))$ – pole ukazatelů na syny vrcholu v, kde $S_v(i)$ je i-tý syn v pro $i = 1, \ldots, \rho(v)$,
$H_v(1..\rho(v) - 1)$ – pole prvků z U takové, že $H_v(i)$ je největší prvek z S reprezentovaný v podstromu i-tého syna vrcholu v (alternativa: $H_v(i)$ je prvek z U takový, že největší prvek reprezentovaný v podstromu i-tého syna vrcholu v je menší nebo roven $H_v(i)$ a to je menší než nejmenší prvek reprezentovaný v podstromu $(i+1)$-ního syna vrcholu v. Naše algoritmy ale tuto alternativu nepřipouštějí.).

Deklarace listů:
listu v je přiřazen prvek key$(v) \in S$.

Někdy se deklaruje pro každý vrchol v (a, b)-stromu ještě ukazatel otec(v) na otce vrcholu v. Když v je kořen, pak otec$(v) = NIL$, kde NIL je prázdný ukazatel.

Když $H_v(i)$ jsou přímo prvky z reprezentované množiny, pak pro každý prvek $s \in S$ kromě největšího existuje právě jeden vnitřní vrchol v (a, b)-stromu a jedno i tak, že $H_v(i) = s$, a největší prvek v S není prvkem pole H_v pro žádný vrchol v. Tento fakt se používá při implementaci, kde se vynechávají listy. Prvky z S jsou reprezentovány v polích H_v vnitřních vrcholů stromu a největší prvek je uložen zvlášť. Je to prostorově efektivnější reprezentace množiny S, ale je technicky nepřehledná. Proto při práci s (a, b)-stromy budeme používat verzi s listy.

Algoritmy

Nyní uvedeme algoritmy pro (a, b)-stromy. Nejprve pomocný algoritmus

Vyhledej(x):
$t :=$kořen stromu T, $w := NIL$
while t není list **do**
  $i := 1$
  while $H_t(i) < x$ a $i < \rho(t)$ **do**
    $i := i + 1$
    if $H_t(i) = x$ **then** $w := t$ **endif**

enddo
$t := S_t(i)$
enddo
Výstup: t a w

MEMBER(x):
Vyhledej(x)
if key$(t) = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

INSERT(x):
Vyhledej(x)
if key$(t) \neq x$ a t není kořen **then**
vytvoř nový list t', key$(t') := x$, $u :=$ otec(t)
if key$(t) < x$ **then** (komentář: $x > \max S$)
$S_u(\rho(u)+1) := t'$, $H_u(\rho(u)) :=$ key(t), $\rho(u) := \rho(u)+1$
else
najdi i, že $S_u(i) = t$
$S_u(\rho(u)+1) := S(\rho(u))$, $j := \rho(u)-1$
while $j \geq i$ **do**
$S_u(j+1) := S_u(j)$, $H_u(j+1) := H_u(j)$, $j := j-1$
enddo
$S_u(i) := t'$, $H_u(i) := x$, $\rho(u) := \rho(u)+1$
endif
$t := u$
while $\rho(t) > b$ **do Štěpení**(t) **enddo**
else
if key$(t) \neq x$ **then**
vytvoř nový list t', key$(t') := x$, vytvoř nový kořen r
if $x <$ key(t) **then**
$H_r(1) := x$, $S_r(1) := t'$, $S_r(2) := t$
else
$H_r(1) :=$ key(t), $S_r(1) := t$, $S_r(2) := t'$
endif
$\rho(r) = 2$
endif
endif

Štěpení(t):
if t je kořen stromu **then**
vytvoř nový kořen u, $S_u(1) := t$, $\rho(u) := 1$
endif
$u :=$ otec(t), najdi i, že $S_u(i) = t$,
vytvoř nový vnitřní vrchol t', $j := 1$
while $j < \lfloor \frac{b+1}{2} \rfloor$ **do**
$S_{t'}(j) := S_t(j + \lceil \frac{b+1}{2} \rceil)$, $H_{t'}(j) := H_t(j + \lceil \frac{b+1}{2} \rceil)$, $j := j+1$
zruš $S_t(j + \lceil \frac{b+1}{2} \rceil)$ a $H_t(j + \lceil \frac{b+1}{2} \rceil)$

enddo
$S_{t'}(\lfloor \frac{b+1}{2} \rfloor) := S_t(b+1)$, zruš $S_t(b+1)$, $\rho(t) := \lceil \frac{b+1}{2} \rceil$, $\rho(t') := \lfloor \frac{b+1}{2} \rfloor$,
if $i < \rho(u)$ **then**
  $S_u(\rho(u)+1) := S_u(\rho(u))$, $j := \rho(u) - 1$
  while $j > i$ **do**
    $S_u(j+1) := S_u(j)$, $H_u(j+1) := H_u(j)$, $j := j - 1$
  enddo
endif
$S_u(i+1) := t'$, $H_u(i+1) := H_u(i)$, $H_u(i) := H_t(\rho(t))$
zruš $H_t(\rho(t))$, $t := u$, $\rho(u) := \rho(u) + 1$

DELETE(x):
Vyhledej(x)
if key$(t) = x$ a t není kořen **then**
  $u :=$ otec(t)
  if $u \neq w$ a $w \neq NIL$ **then**
    najdi j takové, že $H_w(j) = x$, $H_w(j) := H_u(\rho(u) - 1)$
  endif
  najdi i takové, že $S_u(i) = t$, $k := i$
  while $k < \rho(u) - 1$ **do**
    $H_u(k) := H_u(k+1)$, $S_u(k) := S_u(k+1)$, $k := k + 1$
  enddo
  if $i \neq \rho(u)$ **then** $S_u(\rho(u) - 1) := S_u(\rho(u))$ **endif**
  $\rho(u) := \rho(u) - 1$, zruš $H_u(\rho(u))$ a $S_u(\rho(u) + 1))$
  odstraň t, $t := u$
  while $\rho(t) < a$ a t není kořen **do**
    y je bezprostřední bratr t
    if $\rho(y) = a$ **then Spojení**(t, y) **else Přesun**(t, y) **endif**
  enddo
  if t je kořen a $\rho(t) = 1$ **then** odstraň t **endif**
endif
if key$(t) = x$ a t je kořen **then** odstraň t **endif**

Spojení(t, y):
$u :=$ otec(t), najdi i, že $S_u(i) = t$
if $S_u(i-1) = y$ **then** vyměň jména vrcholů t a y, $i := i - 1$ **endif**
$j := 1$
while $j < \rho(y)$ **do**
  $S_t(\rho(t) + j) := S_y(j)$, $H_t(\rho(t) + j) := H_y(j)$, $j := j + 1$
enddo
$S_t(\rho(t) + \rho(y)) := S_y(\rho(y))$, $H_t(\rho(t)) := H_u(i)$, $\rho(t) := \rho(t) + \rho(y)$, odstraň y
while $i < \rho(u) - 1$ **do**
  $S_u(i+1) := S_u(i+2)$, $H_u(i) := H_u(i+1)$, $i := i + 1$
enddo
zruš $S_u(\rho(u))$ a $H_u(\rho(u) - 1)$, $\rho(u) := \rho(u) - 1$
if u je kořen a $\rho(u) = 1$ **then** odstraň u **else** $t := u$ **endif**

Přesun(t, y):
$u :=$ otec(t), najdi i takové, že $S_u(i) = t$
if $S_u(i+1) = y$ **then**
  $S_t(\rho(t)+1) := S_y(1)$, $H_t(\rho(t)) := H_u(i)$, $H_u(i) := H_y(1)$, $j := 1$
  while $j < \rho(y) - 1$ **do**
    $S_y(j) := S_y(j+1)$, $H_y(j) := H_y(j+1)$, $j := j+1$
  enddo
  $S_y(\rho(y)-1) := S_y(\rho(y))$, $\rho(t) := \rho(t)+1$, $\rho(y) := \rho(y)-1$
else
  $S_t(\rho(t)+1) := S_t(\rho(t))$, $j := \rho(t) - 1$
  while $j > 0$ **do**
    $S_t(j+1) := S_t(j)$, $H_t(j+1) := H_t(j)$, $j := j-1$
  enddo
  $\rho(t) := \rho(t)+1$, $S_t(1) := S_y(\rho(y))$, $H_t(1) := H_u(i-1)$,
  $H_u(i-1) := H_y(\rho(y)-1)$, $\rho(y) := \rho(y)-1$
endif

MIN:
$t :=$kořen stromu
while t není list **do** $t := S_t(1)$ **enddo**
Výstup: key(t) je nejmenší prvek S

MAX:
$t :=$kořen stromu
while t není list **do** $t := S_t(\rho(t))$ **enddo**
Výstup: key(t) je největší prvek S

JOIN2(T_1, T_2):
komentář: předpokládáme, že T_1 a T_2 jsou (a,b)-stromy reprezentující množiny S_1 a S_2 a že $\max S_1 < \min S_2$ (algorimtus nekontroluje platnost tohoto předpokladu, který je silnější než požadavek, že S_1 a S_2 jsou disjunktní)
if výška $T_1 \geq$ výška T_2 **then**
  $t :=$kořen T_1, $k :=$výška T_1- výška T_2
  while $k > 0$ **do** $t := S_t(\rho(t))$, $k := k-1$ **enddo**
  if výška $T_1 =$ výška T_2 **then**
    vytvoř nový kořen u, $S_u(1) :=$ kořen T_1, $\rho(u) := 1$
  endif
  $v :=$ kořen T_2, $u :=$ otec(t), $H_u(\rho(u)) := \max S_1$
  $S_u(\rho(u)+1) := v$, $\rho(u) := \rho(u)+1$, **Spojení**(t, v), $t := u$
  while $\rho(t) > b$ **do Štěpení**(t) **enddo**
else
  $t :=$kořen T_2, $k :=$výška T_2- výška T_1
  while $k > 0$ **do** $t := S_t(1)$, $k := k-1$ **enddo**
  $v :=$ kořen T_1, $u :=$ otec(t), $S_u(\rho(u)+1) := S_u(\rho(u))$, $i := \rho(u) - 1$
  while $i > 0$ **do**
    $H_u(i+1) := H_u(i)$, $S_u(i+1) := S_u(i)$, $i := i-1$
  enddo

$H_u(1) := \max S_1$, $S_u(1) := v$,
$\rho(u) := \rho(u) + 1$, **Spojení**(t, v), $t := u$
while $\rho(t) > b$ **do Štěpení**(t) **enddo**
endif

SPLIT(T, x):
Z_1, Z_2 prázdné zásobníky, t :=kořen T
while t není list **do**
$i := 1$
while $H_t(i) < x$ a $i < \rho(t)$ **do** $i := i + 1$ **enddo**
if $i = 2$ **then** vlož podstrom vrcholu $S_t(1)$ do Z_1 **endif**
if $i > 2$ **then**
vytvoř nový vrchol t_1, $\rho(t_1) = i - 1$,
for every $j = 1, 2, \ldots, i - 2$ **do**
$S_{t_1}(j) := S_t(j)$, $H_{t_1}(j) := H_t(j)$
enddo
$S_{t_1}(i - 1) := S_t(i - 1)$, vlož podstrom vrcholu t_1 do Z_1
endif
if $i = \rho(t) - 1$ **then** vlož podstrom $S_t(\rho(t))$ do Z_2 **endif**
if $i < \rho(t) - 1$ **then**
vytvoř nový vrchol t_2, $\rho(t_2) := \rho(t) - i$
for every $j = 1, 2, \ldots, \rho(t) - i - 1$ **do**
$S_{t_2}(j) := S_t(i + j)$, $H_{t_2}(j) := H_t(i + j)$
enddo
$S_{t_2}(\rho(t) - i) := S_t(\rho(t))$, vlož podstrom t_2 do Z_2
endif
$t := S_t(i)$
enddo
if key$(t) = x$ **then**
Výstup: $x \in S$
else
Výstup: $x \notin S$
if key$(t) < x$ **then**
vlož podstrom vrcholu t do Z_1
else
vlož podstrom vrcholu t do Z_2
endif
endif
T_1 :=vrchol Z_1, odstraň T_1 ze Z_1
while $Z_1 \neq \emptyset$ **do**
T' :=vrchol Z_1, odstraň T' ze Z_1, T_1 :=**JOIN**(T', T_1)
enddo
T_2 :=vrchol Z_2, odstraň T_2 ze Z_2
while $Z_2 \neq \emptyset$ **do**
T' :=vrchol Z_2, odstraň T' ze Z_2, T_2 :=**JOIN**(T_2, T')
enddo

Poznámky k algoritmům. Odkaz na otce vrcholu lze realizovat tak, že buď v každém vrcholu v stromu T je deklarován ukazatel otec(v) nebo se v proceduře **Vyhledej** vkládají vrcholy do pomocného zásobníku a pak otec(v) je vrchol v zásobníku bezprostředně před vrcholem v. Všimněme si, že pro jiné vrcholy (tj. vrcholy, které neprošly procedurou **Vyhledej**) příkaz otec(v) není použit.

Korektnost všech algoritmů závisí na korektnosti podprocedury **Vyhledej**. Všimněme si, že podprocedura **Vyhledej**(x) splňuje před každým během cyklu následující invariant:

(1) největší prvek reprezentovaný v podstromu určeném předchůdcem vrcholu t je menší než x a to je menší než nejmenší prvek reprezentovaný v podstromu určeném následníkem vrcholu t – zde se předchůdce a následník bere v hladině vrcholu t vzhledem k lexikografickému uspořádání (tedy nikoli vzhledem ke vztahu otec – syn). Pro korektnost je třeba definovat dolní omezení, když t je první prvek hladiny (v tom případě je jím prvek menší než všechny prvky v univerzu U) a horní omezení, když t je poslední prvek v hladině (v tom případě je to prvek větší než všechny prvky v univerzu U).

Použití pole H k určení syna vrcholu t, ve kterém bude podprocedura pokračovat, zajišťuje platnost tohoto invariantu a odtud plyne korektnost podprocedury **Vyhledej** a všech ostatních algoritmů kromě algoritmu operace **SPLIT**. V algoritmu pro operaci **INSERT** je třeba si uvědomit, že po skončení podprocedury **Vyhledej**(x) platí buď $x > \max S$ nebo $x \leq \mathrm{key}(t)$ – to plyne z vlastnosti pole H a z invariantu podprocedury **Vyhledej**.

Při operaci **SPLIT** se zásobníky používají jednoprůchodově – nejprve se naplní (v této části algoritmu se nepoužije operace **pop**), pak se vyprazdňují (v této fázi se nepoužívá operace **push**). V okamžiku, kdy jsou zásobníky naplněné, platí:

v zásobnících jsou uloženy (a,b)-stromy reprezentující podmnožiny S tak, že v Z_1 jsou stromy s prvky menšími než x, v Z_2 stromy s prvky většími než x;
když (a,b)-strom T_i reprezentující množinu S_i je v zásobníku Z_1 a následující (a,b)-strom T_{i+1} v Z_1 reprezentuje množinu S_{i+1}, pak platí $\max S_i < \min S_{i+1} < x$ a výška T_i je větší nebo rovna výšce T_{i+1};
když (a,b)-strom T_i reprezentující množinu S_i je v zásobníku Z_2 a následující (a,b)-strom T_{i+1} v Z_2 reprezentuje množinu S_{i+1}, pak platí $x < \max S_{i+1} < \min S_i$ a výška T_i je větší nebo rovna výšce T_{i+1};
když T_i a T_{i+1} jsou dva po sobě následující (a,b)-stromy v zásobníku Z_j pro $j = 1, 2$, které mají stejnou výšku, pak následující strom v zásobníku Z_j má výšku ostře menší.

Toto plyne z první fáze algoritmu operace **SPLIT** a zajišťuje korektnost druhé fáze algoritmu.

Složitost operací

Podprocedury **Štěpení**, **Spojení** a **Přesun** vyžadují čas $O(1)$, a proto algoritmy pro operace **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN2** a pro první fázi algoritmu **SPLIT** vyžadují čas $O(1)$ na práci v dané hladině. Protože hladin je nejvýše $\log_a |S|$, můžeme shrnout:

Věta. *Algoritmy pro operace* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN2** *a* **SPLIT** *v (a,b)-stromech vyžadují v nejhorším případě čas $O(\log_a |S|)$, kde S je reprezentovaná množina.*

Důkaz. Je třeba ještě odhadnout spotřebovaný čas druhé fáze algoritmu pro operaci **SPLIT**. Nejprve si všimněme, že algoritmus **JOIN2**(T_1, T_2) vyžaduje ve skutečnosti jen čas úměrný rozdílu výšek stromů T_1 a T_2 (v absolutní hodnotě). Když po naplnění zásobník Z_j pro každé $j = 1, 2$ obsahuje stromy $U_1, U_2, \ldots, U_k$ v tomto pořadí, pak $k \leq 2 \log_a |S|$ a podle předchozích úvah vyprázdnění zásobníku Z_j vyžaduje čas $O(\sum_{i=1}^{k-1}(u_i - u_{i+1} + 1) = O(u_1 + k)$, kde u_i je výška stromu U_i pro $i = 1, 2, \ldots, k$. Protože výška stromu U_1 je nejvýše rovna výšce stromu T, dostáváme, že druhá fáze algoritmu **SPLIT** vyžaduje čas $O(\log_a |S|)$, a důkaz je kompletní. □

Pořádkové statistiky

Nyní popíšeme algoritmus pro operaci **ord**(k). Tato operace se často nazývá k-tá pořádková statistika Bohužel navržená struktura nepodporuje efektivní algoritmus pro tuto operaci. Abychom tento nedostatek odstranili, musíme rozšířit strukturu vnitřního vrcholu v o pole $P_v(1..\rho(v))$, kde $P_v(i)$ je počet prvků S reprezentovaných v podstromu i-tého syna vrcholu v. Udržovat pole P_v v aktuálním stavu znamená při úspěšném provedení aktualizační operace projít cestu z listu, kterého se aktualizační operace týkala, do kořene, a aktualizovat pole P u vrcholů na této cestě. Nyní uvedeme algoritmus pro nalezení k-té pořádkové statistiky.

ord(k):
if $k > |S|$ **then** neexistuje k-tý nejmenší prvek, konec **endif**
t :=kořen stromu
while t není list **do**
  $i := 1$
  while $k > P_t(i)$ a $i < \rho(t)$ **do**
    $k := k - P_t(i)$, $i := i + 1$
  enddo
  $t := S_t(i)$
enddo
Výstup: hledaný prvek je key(t)

Korektnost algoritmu pro operaci **ord**(k) plyne z invariantu:

(1) Původní hodnota k se rovná součtu aktuálního k a počtu prvků z S, které jsou reprezentovány v podstromech určených vrcholy, které v lexikografickém uspořádání předcházejí i-tému synu vrcholu t a jsou na stejné hladině.

Platnost tohoto invariantu zaručuje výběr syna vrcholu t, ve kterém algoritmus pokračuje. Korektnost algoritmu je přímým důsledkem tohoto invariantu. Protože čas, který algoritmus potřebuje na práci v dané hladině, je $O(1)$, dostáváme:

Věta. *Algoritmy pro operace* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **SPLIT**, **JOIN2** *a* **ord**(k) *pro všechna k v rozšířené struktuře (a, b)-stromů vyžadují v nejhorším případě čas $O(\log |S|)$, kde S je reprezentovaná množina.*

Stromy s paralelním přístupem

Pro řešení uspořádaného slovníkového problému jak v interní tak v externí paměti se používají (a, b)-stromy. Jejich použití se liší v parametrech a a b. Pro interní paměť se používají a jsou doporučovány hodnoty $a = 2$, $b = 4$ nebo $a = 3$ a $b = 6$.
Pro externí paměť jsou to hodnoty $a \approx 100$, $b = 2a$. Důvodem je snaha minimalizovat počet komunikací mezi vnitřní a externí pamětí. Snaha je, aby uzel zaplnil celou stránku.

Když je množina reprezentovaná (a, b)-stromem uložena na serveru a má k ní přístup více uživatelů, vzniká problém s aktualizačními operacemi. Tyto operace mění strukturu (a, b)-stromu a ve svých důsledcích způsobují, že se v něm jiný uživatel může ztratit. Klasickým řešením tohoto problému je uzavření celého stromu při zavolání aktualizační operace. Nevýhoda je, že ostatní uživatelé nemohou v tomto případě pracovat, protože nemají přístup k datům. Ukážeme jiné řešení tohoto problému. Je založeno na tzv. paralelní implementaci operací **INSERT** a **DELETE**. Tento postup se v literatuře také nazývá vyvažování shora dolů.

Předpoklad pro tyto algoritmy je, že $b \geq 2a$.
Při operaci **INSERT** jsou ve vyhledávací fázi vždy uzavřeny vrcholy t, otec(t) a synové vrcholu t. Algoritmus zjistí, ve kterém synu vrcholu t má pokračovat, a pak, když $\rho(t) = b$, provede **Štěpení** (proto je nutně $b \geq 2a$, abychom po této operaci měli stále (a, b)-strom). V algoritmu pak odpadne vyvažovací část (tj. **Štěpení** při cestě vzhůru ke kořeni).
Při operaci **DELETE** jsou ve vyhledávací fázi uzavřeny vrcholy t, otec(t), bezprostřední bratr y vrcholu t a jejich synové. Když $\rho(t) = a$, pak po nalezení vrcholu, kde se bude pokračovat, se provede buď **Přesun** (když $\rho(y) > a$) nebo **Spojení** (když $\rho(y) = a$) a vynechá se vyvažovací část zakončující původní algoritmus.

Tato modifikace vyžaduje více **Štěpení**, **Spojení** a **Přesunů**, ale asymptoticky má stejný čas (větší multiplikativní konstantou). Doporučené hodnoty parametrů a a b pro tyto algoritmy jsou $a \approx 100$ a $b = 2a + 2$ při uložení na serveru v externí paměti.

A-sort

Důležitou aplikací (a, b)-stromů jsou třídicí algoritmy. Použití (a, b)-stromů pro setřídění náhodné posloupnosti není vhodné, protože režie na udržování struktury (a, b)-stromu vede k větší multiplikativní konstantě než je u klasických třídicích algoritmů a také uložení (a, b)-stromu vyžaduje více paměti než klasické algoritmy. Situace se podstatně změní, když vstupní posloupnost je předtříděná a je ji třeba jen dotřídit. Klasické algoritmy nejsou schopné využít faktu, že posloupnost je předtříděná. Jejich časová náročnost je prakticky stejná (někdy i horší) jako u náhodné posloupnosti. Na rozdíl od nich algoritmus A-**sort** založený na (a, b)-stromech je schopen předtříděnosti využít a má na předtříděných posloupnostech lepší výsledky než klasické algoritmy.

Abychom získali efektivní algoritmus pro třídění, modifikujeme datovou strukturu (a, b)-stromů pro A-**sort**. V (a, b)-stromu reprezentujícím vstupní posloupnost je dán ukazatel Prv na první list, je dána cesta z tohoto listu do kořene (t.j. pro každý vrchol na této cestě je znám jeho otec) a listy (a, b)-stromu jsou propojeny do seznamu v rostoucím lexikografickém pořadí (ukazatel na následující prvek je Nasl). Nyní uvedeme algoritmus A-**sort**.

A-**sort**$(x_1, x_2, \ldots, x_n)$:
$i := n - 1$
vytvoř jednoprvkový strom s vrcholem t, key$(t) := x_n$, Prv $:= t$
while $i \geq 1$ **do**
  A-**Insert**(x_i), $i := i - 1$
enddo
$y_1 :=$ key(Prv), $t :=$ Nasl(Prv), $i := 2$
while $i \leq n$ **do**
  $y_i :=$ key(t), $i := i + 1$, $t :=$ Nasl(t)
enddo
Výstup: $(y_1, y_2, \ldots, y_n)$ je setříděná posloupnost $(x_1, x_2, \ldots, x_n)$

A-**Insert**(x):
$t :=$ Prv
while $t \neq$kořen T a $H_t(1) < x$ **do**
  $t :=$ otec(t)
enddo
while $t \neq$list **do**
  $i := 1$
  while $H_t(i) < x$ a $i < \rho(t)$ **do** $i := i + 1$ **enddo**
  if $i > 1$ **then**
    $v := S_t(i - 1)$
  else
    $v := S_v(\rho(v))$
  endif
  $t := S_t(i)$
enddo
if key$(t) \neq x$ **then**
  vytvoř nový list t', key$(t') := x$
  if t je kořen stromu **then**
    vytvoř nový kořen r stromu, $\rho(r) := 2$
    if key$(t) < x$ **then**
      $H_r(1) :=$ key(t), $S_r(1) := t$, $S_r(2) := t'$, Prv $:= t$, Nasl$(t) := t'$, Nasl$(t') := NIL$
    else
      $H_r(1) := x$, $S_r(1) := t'$, $S_r(2) := t$, Prv $:= t'$, Nasl$(t') := t$, Nasl$(t) := NIL$
    endif
    otec(Prv) $:= r$
  else
    $u :=$ otec(t)
    if key$(t) < x$ **then** (komentář: $x > \max S$)
      $S_u(\rho(u) + 1) := t'$, $H_u(\rho(u)) :=$ key(t), $\rho(u) := \rho(u) + 1$,
      Nasl$(t) := t'$, Nasl$(t') := NIL$
    else
      Nasl$(v) := t'$**,** Nasl$(t') := t$
      najdi i, že $S_u(i) = t$, $S_u(\rho(u) + 1) := S(\rho(u))$, $j := \rho(u) - 1$
      while $j \geq i$ **do**

$S_u(j+1) := S_u(j)$, $H_u(j+1) := H_u(j)$, $j := j-1$
enddo
$S_u(i) := t'$, $H_u(i) := x$, $\rho(u) := \rho(u)+1$,
if $t = \mathrm{Prv}$ **then** $\mathrm{Prv} := t'$ **endif**
endif
$t := u$
while $\rho(t) > b$ **do Štěpení**(t) **enddo**
endif
endif

Korektnost algoritmu plyne z faktu, že key je izomorfismus zachovávající uspořádání a seznam listů je v rostoucím pořadí. Protože v je vždy bezprostřední předchůdce t, je seznam korektně definován. Ukazatel otec(v) je znám na cestě z vrcholu Prv do kořene, pro ostatní vrcholy se řeší stejným způsobem jako pro (a,b)-stromy.

Složitost algoritmu A-**sort**

Algoritmus A-**sort** pro většinu posloupností vyžaduje více času i více paměti než klasické třídicí algoritmy, ale jejich asymptotická složitost je stejná. Jeho výhoda se projeví při použití na předtříděné posloupnosti. Mějme posloupnost $(x_1, x_2, \ldots, x_n)$ prvků z totálně uspořádaného univerza U. Definujme

$$f_i = |\{j \mid i < j,\, x_j < x_i\}| \quad \text{a} \quad F = |\{(i,j) \mid i < j,\, x_j < x_i\}| = \sum_{i=1}^{n} f_i.$$

(F je počet inverzí v posloupnosti $(x_1, x_2, \ldots, x_n)$.)
Zřejmě $F = 0$, právě když posloupnost $(x_1, x_2, \ldots, x_n)$ je setříděná. Dále $0 \le F \le \binom{n}{2}$ a $F = \binom{n}{2}$, právě když posloupnost $(x_1, x_2, \ldots, x_n)$ je klesající. Proto vezmeme F jako míru předtříděnosti posloupnosti. Spočítáme složitost algoritmu A-**sort** v závislosti na n a F.

Algoritmus A-**sort** v nejhorším případě vyžaduje celkový čas, který potřebuje A-**Insert**, plus $O(n)$ (čas pro vypsání setříděné posloupnosti z (a,b)-stromu a čas na n zavolání podprocedury A-**Insert**). Algoritmus A-**Insert**(x) vyžaduje čas potřebný na nalezení místa, kam vložit prvek x, plus O(počet volání **Štěpení**). Protože každý běh procedury **Štěpení** vytvoří jeden vnitřní vrchol (a,b)-stromu a protože $a \ge 2$ a (a,b)-strom po skončení volání A-**Insert** má n listů, je počet vnitřních vrcholů (a,b)-stromu menší než n. Proto čas, který vyžadují všechny běhy podprocedury A-**Insert**, je čas potřebný na nalezení míst jednotlivých prvků posloupnosti plus $O(n)$. Když podprocedura A-**Insert**(x) při hledání místa pro prvek x skončí ve výšce h (tj. první cyklus se h-krát opakuje), pak nalezení místa pro prvek x vyžaduje čas $O(h)$. Všechny prvky reprezentované (a,b)-stromem pod prvním vrcholem ve výšce $h-1$ jsou menší než x a je jich nutně alespoň a^{h-2} (v tomto podstromu má každý vrchol alespoň a synů). Když $x = x_i$, pak počet prvků reprezentovaných (a,b)-stromem při běhu procedury A-**Insert**(x), které jsou menší než x, je $f_i = |\{j = i+1, i+2, \ldots, n \mid x_j < x_i\}|$. Pak platí

$$a^{h-2} \le f_i \implies h - 2 \le \log_a f_i \implies h \in O(\log f_i).$$

Odtud čas potřebný pro nalezení pozice x_i je v nejhorším případě $O(\log f_i)$. Celkový čas algoritmu A-**sort** je tedy $O(\sum_{i=1}^{n} \log f_i + n)$. Využijeme toho, že aritmetický průměr není nikdy menší než geometrický, a dostaneme

$$\sum_{i=1}^{n} \log f_i = \log \prod_{i=1}^{n} f_i = n \log(\prod_{i=1}^{n} f_i)^{\frac{1}{n}} \leq n \log \frac{\sum_{i=1}^{n} f_i}{n} = n \log \frac{F}{n}.$$

Shrneme získané odhady.

Věta. *Algoritmus A-**sort** vyžaduje na setřídění n-členné posloupnosti v nejhorším případě čas $O(n + n \log \frac{F}{n})$, kde F je míra setříděnosti vstupní posloupnosti.*

Zhodnocení: Protože A-**sort** nepoužívá operaci **DELETE**, doporučuje se použít $(2,3)$-stromy. Když se budou třídit posloupnosti s mírou $F \leq n \log n$, pak algoritmus A-**sort** bude potřebovat v nejhorším případě čas $O(n \log \log n)$. Mehlhorn a Tsakalidis dokázali, že když $F \leq 0.02n^{1.57}$, pak algoritmus A-**sort** je rychlejší než algoritmus **Quicksort**.

Hladinově propojené (a,b)-stromy s prstem

Hladinově propojený (a,b)-strom s prstem je (a,b)-strom, kde struktura vrcholu je rozšířena (oproti klasickému (a,b)-stromu) o ukazatele:
otec(v), levy(v), pravy(v), kde ukazatel levy(v) ukazuje na vrchol ve stejné hladině, který je bezprostředním předchůdcem v v lexikografickém uspořádání (má hodnotu NIL, když v je první vrchol ve své hladině),
pravy(v) ukazuje na vrchol ve stejné hladině, který je bezprostředním následníkem v v lexikografickém uspořádání (má hodnotu NIL, když v je poslední vrchol ve své hladině),
otec(v) ukazuje na otce vrcholu v (má hodnotu NIL, když v je kořen).
Navíc je dán ukazatel Prst na některý list.

V této struktuře se liší hlavně vyhledávání, které je zobecněním postupu A-**sortu**. Úlohu ukazatele Prv v používaného v A-**sortu** zde hraje ukazatel Prst. To znamená, že vyhledávání začíná od listu p, na který ukazuje Prst. Když x je menší než prvek reprezentovaný tímto listem, pak se pokračuje ve vrcholu v =otec(p), a když p byl i-tým synem v, tak se pomocí pole H_v zjišťuje, zda x nemá být reprezentován v podstromu jeho j-tého syna pro nějaké $j < i$. Když ne, pokračuje se ukazatelem levy(v) a testuje se, zda x nemá být v podstromu tohoto vrcholu. Když nenastane ani tento případ, celý postup opakuje o hladinu výš (zkoumá se otec vrcholu v atd.). Pokud x je větší než prvek reprezentovaný listem, na který ukazuje Prst, postupuje se zrcadlově (místo ukazatele levy se použije ukazatel pravy). Když se nalezne vrchol, v jehož podstromu má x ležet, aplikuje se podprocedura **Vyhledej**, která však začíná od tohoto vrcholu a ne od kořene jako klasická podprocedura **Vyhledej**.

Tato struktura kromě operací uspořádaného slovníkového problému **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN2** a **SPLIT** (operace **MIN**, **MAX**, **JOIN2** a **SPLIT** jsou stejné jako v klasickém (a,b)-stromu, tj. začínají od kořene) ještě používá operaci **PRST**(x), která nastaví ukazatel Prst na list, který reprezentuje nejmenší prvek větší nebo rovný x (pokud $x > \max S$, ukazatel Prst bude ukazovat na největší list). Operace **PRST**(x) provede vyhledání (výše popsaným způsobem) a pak nastaví ukazatel Prst na příslušný list.

Tyto stromy jsou výhodné pro úlohy, kde argumenty řady po sobě jdoucích operací jsou v blízkém okolí nějakého prvku $x \in U$. Pak vyhledávání je rychlejší než v klasickém (a,b)-stromu. Tento jev ilustruje právě algoritmus A-**sort**.

Amortizovaná složitost operací

Budeme se zabývat analýzou počtu vyvažovacích operací, protože každé **Štěpení, Spojení** a **Přesun** vyžaduje sice čas $O(1)$, ale ve skutečnosti tvoří nejpomalejší část algoritmů pro operace **INSERT** a **DELETE**. Navíc omezení jejich počtu v algoritmu A-**sort** vedlo k menší složitosti, než mají klasické algoritmy pro předtříděné posloupnosti. Víme, že libovolný běh algoritmu **INSERT** volá podproceduru **Štěpení** nejvýše $\log(|S|)$-krát a libovolný běh algoritmu **DELETE** může nejvýše $\log(|S|)$-krát zavolat podproceduru **Spojení** a nejvýše jednou podproceduru **Přesun**. Je vidět, že v obecném případě tyto odhady nejdou zlepšit. Pro vhodný typ (a,b)-stromu se však amortizovaný počet vyvažovacích operací, začínáme-li s původně prázdným stromem, drasticky změní. Je asymptoticky roven počtu operací ve vyšetřované posloupnosti, což znamená, že je konstantní pro jednotlivé dané operace. Dokážeme tento výsledek.

Pro pevné a a b označme

$$c = \min\{\min\{2a-1, \lceil \frac{b+1}{2} \rceil\} - a, b - \max\{2a-1, \lfloor \frac{b+1}{2} \rfloor\}\}.$$

Připomínáme, že výška vrcholu v kořenovém stromě je maximální délka cesty z tohoto vrcholu do některého listu v jeho podstromu. V (a,b)-stromech nezáleží na tom, který list budeme uvažovat, všechny cesty mají stejnou délku.

Věta. *Předpokládejme, že $b \geq 2a$ a $a \geq 2$. Nechť $\mathcal{P}$ je posloupnost n operací* **INSERT** *a* **DELETE***, kterou aplikujeme na původně prázdný (a,b)-strom. Označme*
St_h = počet **Štěpení** *ve výšce h při aplikaci $\mathcal{P}$, $St = \sum_h St_h$;*
Sp_h = počet **Spojení** *ve výšce h při aplikaci $\mathcal{P}$, $Sp = \sum_h Sp_h$;*
P_h = počet **Přesunů** *ve výšce h při aplikaci $\mathcal{P}$, $P = \sum_h P_h$.*
Pak platí

(1) $P \leq n$ *a* $(2c-1)St + cSp \leq n + c + \frac{c(n-2)}{a+c-1}$;
(2) $St_h + Sp_h + P_h \leq \frac{2(c+2)n}{(c+1)^h}$.

Z definice plyne, že $c \geq 1$, a tedy z (1) dostaneme, že $St + Sp \leq \frac{n}{c} + 1 + \frac{n-2}{a} \leq \frac{3n}{2} + 1$. Pak amortizovaný počet vyvažovacích operací je

$$\lim_{n \mapsto \infty} \frac{P + St + Sp}{n} \leq \frac{5}{2}.$$

To znamená, že sice může existovat operace, která vyžaduje $\log(|S|)$ vyvažovacích akcí, ale těchto operací je málo. Velká většina operací vyžaduje nejvýše dvě vyvažovací akce. Tedy vzhledem k asymptotickému počtu vyvažovacích operací je situace podobná jako v A-**sortu**.

Důkaz věty je založen na tzv. bankovním principu – navrhneme kvantitativní ohodnocení (a,b)-stromu, nalezneme jeho horní odhad a popíšeme, jak vyvažovací operace toto ohodnocení mění. Srovnání těchto odhadů dá požadovaný výsledek.

Mějme (a,b)-strom T. Pro vnitřní vrchol v různý od kořene definujme

$$b(v) = \min\{\rho(v) - a, b - \rho(v), c\}$$

a pro kořen r definujme

$$b(r) = \min\{\rho(r) - 2, b - \rho(r), c\}.$$

Pozorování 1. *Pro vnitřní vrchol v stromu různý od kořene platí*

(1) $b(v) \leq c$;
(2) *když* $\rho(v) = a$ *nebo* $\rho(v) = b$, *pak* $b(v) = 0$;
(3) *když* $\rho(v) = a - 1$ *nebo* $\rho(v) = b + 1$, *pak* $b(v) = -1$;
(4) *když* $\rho(v) = 2a - 1$, *pak* $b(v) = c$.

Když v' a v'' jsou dva různé vrcholy stromu různé od kořene takové, že $\rho(v') = \lceil \frac{b+1}{2} \rceil$ *a* $\rho(v'') = \lfloor \frac{b+1}{2} \rfloor$, *pak* $b(v') + b(v'') \geq 2c - 1$. *Pro kořen stromu platí* $b(\text{kořene}) \leq c$.

Důkaz. Platnost (1) plyne přímo z definice $b(v)$. Dále, když $\rho(v) = a$ nebo $\rho(v) = a - 1$, pak $\min\{\rho(v) - a, b - \rho(v), c\} = \rho(v) - a$, protože $b \geq 2a$ a $a \geq 2$, a tedy v prvním případě $b(v) = 0$ a v druhém případě $b(v) = -1$. Analogicky dostaneme, že $\rho(v) = b$ implikuje $b(v) = 0$ a $\rho(v) = b + 1$ implikuje $b(v) = -1$, takže (2) a (3) platí. Když $\rho(v) = 2a - 1$, pak $\rho(v) - a = (2a - 1) - a \geq \min\{2a - 1, \lceil \frac{b+1}{2} \rceil\} - a \geq c$ a také $b - \rho(v) = b - (2a - 1) \geq b - \max\{2a - 1, \lfloor \frac{b+1}{2} \rfloor\} \geq c$, a proto $b(v) = c$ a (4) platí. Předpokládejme, že $\rho(v') = \lceil \frac{b+1}{2} \rceil$ a $\rho(v'') = \lfloor \frac{b+1}{2} \rfloor$. Pak $\rho(v') - a \geq \min\{2a - 1, \lceil \frac{b+1}{2} \rceil\} - a \geq c$ a $b - \rho(v') \geq b - \lfloor \frac{b+1}{2} \rfloor - 1 \geq b - \max\{2a - 1, \lfloor \frac{b+1}{2} \rfloor\} - 1 \geq c - 1$. Navíc, když $b - \rho(v') < c$, pak $\lceil \frac{b+1}{2} \rceil \neq \lfloor \frac{b+1}{2} \rfloor$ a $2a - 1 \leq \lfloor \frac{b+1}{2} \rfloor$. Analogicky $b - \rho(v'') \geq b - \max\{2a - 1, \lfloor \frac{b+1}{2} \rfloor\} \geq c$ a $\rho(v'') - a \geq \lceil \frac{b+1}{2} \rceil - a - 1 \geq \min\{2a - 1, \lceil \frac{b+1}{2} \rceil\} - a - 1 \geq c - 1$. Navíc, když $\rho(v'') - a < c$, pak $\lceil \frac{b+1}{2} \rceil \neq \lfloor \frac{b+1}{2} \rfloor$ a $2a - 1 \geq \lceil \frac{b+1}{2} \rceil$. Tedy máme $b(v'), b(v'') \geq c - 1$. Protože $\lceil \frac{b+1}{2} \rceil \geq \lfloor \frac{b+1}{2} \rfloor$, dostáváme, že současně nemůže nastat $b - \rho(v') < c$ a $a - \rho(v'') < c$. Odtud plyne, že buď $b(v') \geq c$ nebo $b(v'') \geq c$, a proto $b(v') + b(v'') \geq 2c - 1$. □

Strom (T, r) ohodnotíme funkcí $b(T)$ definovanou následovně:

$$b_h(T) = \sum \{b(v) \mid v \neq r \text{ vnitřní vrchol stromu } T \text{ ve výšce } h\}, b(T) = \sum_{h=1}^{\infty} b_h(T) + b(r).$$

Řekneme, že (T, r, v) je parciální (a,b)-strom, když r je kořen stromu, v je vnitřní vrchol T různý od r a platí:

$a - 1 \leq \rho(v) \leq b + 1$ a $2 \leq \rho(r) \leq b$;
když t je vnitřní vrchol T různý od v a r, pak $a \leq \rho(t) \leq b$;
všechny cesty z kořene r do nějakého listu mají stejnou délku.

Nyní budeme vyšetřovat vliv vyvažovacích operací **Štěpení**, **Spojení** a **Přesun** na ohodnocení vyšetřovaného stromu. Navíc také zjistíme, jaký vliv na toto ohodnocení má přidání nebo ubrání listu. Idea je, že změny vyšetřovaného stromu v průběhu operací **INSERT** a **DELETE** lze rozložit právě do těchto akcí. Sečtením získaných odhadů dostaneme odhad změny ohodnocení během provedení posloupnosti operací $\mathcal{P}$. Odhady změn ohodnocení jsou založeny na následujícím pozorování.

Pozorování 2. *Když v a v' jsou vnitřní vrcholy stromů T a T' různé od jejich kořenů, pak platí:*

(1) *když $\rho(v) = \rho(v')$, pak $b_{T'}(v') = b_T(v)$;*
(2) *když $|\rho(v) - \rho(v')| = 1$, pak $b_{T'}(v') \geq b_T(v) - 1$.*

Lemma 3. *Když (T, r) je (a, b)-strom a když strom T' vznikne z T přidáním nebo ubráním jednoho syna vrcholu v ve výšce 1 (tj. přidaný nebo ubraný syn je list), pak (T', r, v) je parciální (a, b)-strom a platí*

$$b_1(T') \geq b_1(T) - 1, \quad b_h(T') = b_h(T) \text{ pro } h > 1 \text{ a } b(T') \geq b(T) - 1.$$

Důkaz. Plyne z definice parciálního (a, b)-stromu a z Pozorování 2. □

Lemma 4. *Nechť (T, r, v) je parciální (a, b)-strom, kde v je vrchol ve výšce $l \geq 1$ takový, že $\rho(v) = b + 1$. Když T' vznikne z T operací* **Štěpení**(v)*, pak $(T', r, \text{otec}(v))$ je parciální (a, b)-strom a platí:*

$$b_l(T') \geq b_l(T) + 2c, \; b_{l+1}(T') \geq b_{l+1}(T) - 1, \; b_h(T') = b_h(T) \text{ pro } h \neq l, l+1,$$
$$\text{a odtud} \qquad b(T') \geq b(T) + 2c - 1.$$

Důkaz. Z popisu operace **Štěpení** plyne, že $(T', r, \text{otec}(v))$ je parciální (a, b)-strom. Pro přehlednost zapisujme ohodnocení vrcholu x ve stromu T' jako $b'(x)$. Předpokládejme, že z vrcholu v vznikly vrcholu v' a v'' a že platí $\rho(v') = \lceil \frac{b+1}{2} \rceil$ a $\rho(v'') = \lfloor \frac{b+1}{2} \rfloor$. Vztahy pro hladiny $h \neq l$ okamžitě plynou Z Pozorování 2. Pro l-tou hladinu platí $b'_l(T') = b_l(T) - b(v) + b'(v') + b'(v'')$ a výslednou nerovnost dostaneme z Pozorování 1. □

Lemma 5. *Nechť (T, r, v) je parciální (a, b)-strom, kde v je vrchol ve výšce $l \geq 1$ takový, že $\rho(v) = a - 1$. Když y je bezprostřední bratr v takový, že $\rho(y) = a$, a když strom T' vznikne z T operací* **Spojení**(v, y)*, pak $(T', r, \text{otec}(v))$ je parciální (a, b)-strom a platí:*

$$b_l(T') \geq b_l(T) + c + 1, \; b_{l+1}(T') \geq b_{l+1}(T) - 1, \; b_h(T') = b_h(T) \text{ pro } h \neq l, l+1,$$
$$\textit{a odtud} \qquad b(T') \geq b(T) + c.$$

Důkaz. Z popisu operace **Spojení** plyne, že $(T', r, \text{otec}(v))$ je parciální (a, b)-strom. Pro přehlednost budeme stejně jako v důkazu Lemmatu 4 ohodnocení vrcholu x ve stromě T' zapisovat jako $b'(x)$. Předpokládejme, že spojením vrcholů v a y vznikl vrchol u. Pak $\rho(u) = 2a - 1$. Z Pozorování 2 okamžitě plynou vztahy pro hladiny $h \neq l$. Pro l-tou hladinu platí $b'_l(T') = b_l(T) - b(v) - b(y) + b'(u)$ a opět použijeme Pozorování 1. □

Lemma 6. *Nechť (T, r, v) je parciální (a, b)-strom, kde v je vrchol ve výšce $l \geq 1$ takový, že $\rho(v) = a - 1$. Když y je bezprostřední bratr v takový, že $\rho(y) > a$, a když strom T' vznikne z T operací* **Přesun**(v, y)*, pak (T', r) je (a, b)-strom a platí:*

$$b_l(T') \geq b_l(T),\ b_h(T') = b_h(T) \text{ pro } h \neq l \text{ a } b(T') \geq b(T).$$

Důkaz. Z popisu operace **Přesun** plyne, že T' je (a, b)-strom. Z Pozorování 2 okamžitě plynou vztahy pro hladiny $h \neq l$ a pro l-tou hladinu platí $b'_l(T') = b_l(T) - b(v) - b(y) + b'(v') + b'(v'')$, kde v a v' označují modifikované vrcholy v a y a symbol $b'(x)$ je opět zjednodušeným zápisem ohodnocení vrcholu x ve stromě T'. Výsledný vztah pro l-tou hladinu okamžitě plyne z Pozorování 1 a 2. □

Označme T_k (a, b)-strom vzniklý provedením posloupnosti $\mathcal{P}$ na původně prázdný (a, b)-strom. Uvědomíme-li si, že všechna ohodnocení prázdného stromu jsou rovna 0, pak sečtením předchozích výsledků z Lemmat 3–6 dostáváme:

Důsledek 7. *Položme $St_0 + Sp_0 =$* počet listů v $T_k \leq n$. *Pak*

$$b_h(T_k) \geq 2cSt_h + (c+1)Sp_h - St_{h-1} - Sp_{h-1} \text{ pro } h \geq 1.$$

Dále $b(T_k) \geq (2c-1)St + cSp - n$, kde n je délka posloupnosti $\mathcal{P}$. □

Z prvního výrazu vyjádříme $St_h + Sp_h$ (využijeme toho, že $c \geq 1$, a tedy $2c \geq c + 1$), pak do něho rekurzivně dosadíme a na závěr zlomek rozšíříme. Dostaneme

$$St_h + Sp_h \leq \frac{b_h(T_k)}{c+1} + \frac{St_{h-1} + Sp_{h-1}}{c+1} \leq \frac{b_h(T_k)}{c+1} + \frac{b_{h-1}(T_k)}{(c+1)^2} + \frac{St_{h-2} + Sp_{h-2}}{(c+1)^2} \leq \cdots \leq$$
$$\sum_{i=0}^{h-1} \frac{b_{h-i}(T_k)}{(c+1)^{i+1}} + \frac{n}{(c+1)^h} = \frac{n}{(c+1)^h} + \sum_{l=1}^{h} b_l(T_k) \frac{(c+1)^l}{(c+1)^{h+1}}.$$

Nyní odhadneme shora $b(T_k)$.

Lemma 8. *Když T je (a, b)-strom s m listy, pak $0 \leq b(T) \leq c + (m-2)\frac{c}{a+c-1}$.*

Důkaz. Pro $0 \leq j < c$ označme m_j počet vnitřních vrcholů různých od kořene, které mají přesně $a + j$ synů, a m_c počet vnitřních vrcholů různých od kořene, které mají alespoň $a + c$ synů. Když vrchol v má $a + j$ synů, pak $b_T(v) \leq j$ a pro každý vnitřní vrchol v platí $b_T(v) \leq c$. Tedy $b(T) \leq c + \sum_{j=0}^{c} j m_j$ (první c je odhad ohodnocení kořene). Z vlastností stromů plyne

$$2 + \sum_{j=0}^{c} (a+j) m_j \leq \sum \{\rho(v) \mid v \text{ je vnitřní vrchol } T\} = m + \sum_{j=0}^{c} m_j.$$

Odtud $\sum_{j=0}^{c} (a + j - 1) m_j \leq m - 2$.
Protože $\frac{j}{a+j-1} \leq \frac{c}{a+c-1}$ pro každé j takové, že $0 \leq j \leq c$, dostáváme

$$b(T) \leq c + \sum_{j=0}^{c} j m_j = c + \sum_{j=0}^{c} \frac{j}{a+j-1} (a+j-1) m_j \leq c + \frac{c}{a+c-1}(m-2)$$

a lemma je dokázáno. □

Nyní již dokážeme tvrzení (1) z Věty. Protože každá operace **DELETE** použije nejvýše jednu operaci **Přesun** (a operace **INSERT** operaci **Přesun** nepoužívá), dostáváme, že

$$P \leq \text{počet operací } \mathbf{DELETE} \leq n,$$

a první nerovnost platí. Abychom dokázali druhou nerovnost, spojíme druhé tvrzení v Důsledku 7 a Lemma 8 (T_k má nejvýše n listů):

$$(2c-1)St + cSp - n \leq b(T_k) \leq c + (n-2)\frac{c}{a+c-1}.$$

Odtud plyne požadovaná nerovnost a (1) je dokázáno.

Pro důkaz tvrzení (2) z Věty použijeme následující odhad.

Lemma 9. *Pro každé $h \geq 1$ a pro každý (a,b)-strom T s m listy platí*

$$\sum_{l=1}^{h} b_l(T)(c+1)^l \leq (c+1)m.$$

Důkaz. Pro $0 \leq j < c$ a pro libovolné h označme $m_j(h)$ počet vrcholů ve výšce h různých od kořene, které mají přesně $a+j$ synů, a $m_c(h)$ počet vrcholů ve výšce h různých od kořene, které mají alespoň $a+c$ synů. Pak z vlastností stromů dostáváme

$$b_h(T) \leq \sum_{j=0}^{c} j m_j(h) \text{ a } \sum_{j=0}^{c}(a+j)m_j(h) \leq \sum_{j=0}^{c} m_j(h-1) \text{ pro každé } h \geq 1,$$

kde jsme dodefinovali $\sum_{j=0}^{c} m_j(0) = m$. Tyto vztahy použijeme v následujícím odhadu. Platí

$$\begin{aligned}
\sum_{l=1}^{h} b_l(T)(c+1)^l \leq & \sum_{l=1}^{h} \big[(c+1)^l \big(\sum_{j=0}^{c} j m_j(l) \big)\big] \leq \\
& \sum_{l=1}^{h} \big[(c+1)^l \big(\sum_{j=0}^{c} m_j(l-1) - a \sum_{j=0}^{c} m_j(l) \big)\big] = \\
& (c+1)\sum_{j=0}^{c} m_j(0) - (c+1)^h a \sum_{j=0}^{c} m_j(h) + \\
& \sum_{l=1}^{h-1} (c+1)^{l+1} \big(\sum_{j=0}^{c} m_j(l) - \frac{a}{c+1} \sum_{j=0}^{c} m_j(l) \big) \leq (c+1)m,
\end{aligned}$$

kde první nerovnost dostáváme dosazením odhadu pro $b_l(T)$, druhou nerovnost dosazením odhadu počtu vrcholů s daným stupněm, rovnost jsme získali přerovnáním sčítanců tak, aby

výrazy $\sum_{j=0}^{c} m_j(l)$ pro stejná l byly u sebe, a poslední nerovnost plyne z toho, že $\frac{a}{c+1} \geq 1$, a tedy třetí sčítanec v předchozím výrazu není kladný. □

Zkombinujeme odhad $St_h + Sp_h$ s výsledkem Lemmatu 9 a dostaneme

$$St_h + Sp_h \leq \frac{n}{(c+1)^h} + \sum_{l=1}^{h} b_l(T_k)\frac{(c+1)^l}{(c+1)^{h+1}} \leq \frac{n}{(c+1)^h} + \frac{n(c+1)}{(c+1)^{h+1}} = \frac{2n}{(c+1)^h}.$$

Protože $P_h \leq Sp_{h-1} - Sp_h \leq St_{h-1} + Sp_{h-1} \leq \frac{2n}{(c+1)^{h-1}}$, platí

$$St_h + Sp_h + P_h \leq \frac{2n}{(c+1)^h} + \frac{2n}{(c+1)^{h-1}} = \frac{2n + 2n(c+1)}{(c+1)^h} = \frac{2n(c+2)}{(c+1)^h}$$

a důkaz (2) ve Větě je hotov.

Věta vysvětluje, proč jsou doporučovány hodnoty $b \geq 2a$ – pak je počet vyvažovacích operací během posloupnosti operací **INSERT** a **DELETE** lineární vzhledem k délce této posloupnosti. Pro $b = 2a - 1$ lze lehce nalézt posloupnost operací **INSERT** a **DELETE** o délce n takovou, že její aplikace na prázdný (a, b)-strom vyžaduje počet vyvažovacích operací úměrný $n \log n$ (pro každé dostatečně velké n). Podobná věta platí i pro paralelní implementaci (a, b)-stromů, ale zde je nutný předpoklad $b \geq 2a+2$ místo původního $b \geq 2a$. Pro $b = 2a$ nebo $b = 2a+1$ lze nalézt posloupnost, která je protipříkladem tvrzení Věty pro paralelní implementaci algoritmů.

Pro hladinově propojené (a, b)-stromy platí silnější verze.

Věta. *Předpokládejme, že $b \geq 2a$ a $a \geq 2$. Mějme hladinově propojený (a, b)-strom s prstem T, který reprezentuje n-prvkovou množinu. Pak posloupnost $\mathcal{P}$ operací* **MEMBER**, **INSERT**, **DELETE** *a* **PRST** *aplikovaná na T vyžaduje čas*

$$O(\log(n) + \text{ čas na vyhledání prvků}).$$

Vysvětlení: Začínáme v libovolném hladinově propojeném (a, b)-stromu T, takže jeho struktura může být nevýhodná pro danou posloupnost operací $\mathcal{P}$. Abychom se dostali do vhodného režimu, může být třeba až $\log(n)$ vyvažovacích operací. Čas na vyhledávání nemůžeme ovlivnit, ten musí ovlivnit uživatel.

Vyvažování při operaci **INSERT** lze provádět také tak, že operace **Štěpení**(t) se provede, jen když oba bratři vrcholu t mají b synů. Jinak se provádí operace **Přesun**. Nevím o žádném seriózním pokusu tyto alternativy porovnat.

III. Vyhledávání v uspořádaném poli

Jednou z nejstarších datových struktur je uspořádané pole. Tento způsob uložení dat se používal již v době před počítači – jako příklad lze uvést kartotéku se setříděnými záznamy. Lze říci, že použití vyhledávacích stromů je vlastně jakýmsi rozvinutím této ideje. Přesto

i uspořádané pole umožňuje řadu nových postupů. Seznámíme se s několika algoritmy pro tuto strukturu a ukážeme si jejich složitost. Některé postupy se zatím nepovedlo vhodným způsobem zobecnit a použít jinde.

Zadání úlohy: Máme podmnožinu S lineárně uspořádaného univerza, která je uložena v poli $A[1..|S|]$ tak, že pro $i < j$ je $A(i) < A(j)$. Pro dané $x \in U$ máme zjistit, zda $x \in S$ (operace **MEMBER**(x)).

Řešení: Pokud $x < A(1)$ nebo $A(|S|) < x$, pak x není prvkem S. V opačném případě buď $x = A(1)$ nebo $x = A(|S|)$ nebo existují dvě hodnoty d a h takové, že $1 \leq d < d+1 < h \leq |S|$ a $A(d) < x < A(h)$. Najdeme n takové, že $d < n < h$, a dotazem zjistíme, zda $x = A(n)$ (pak končíme a $x \in S$), nebo zda $x < A(n)$ (pak položíme $h = n$) nebo $x > A(n)$ (pak položíme $d = n$), a proces opakujeme. Končíme, když $d + 1 \geq h$, pak $x \notin S$. Na začátku po ověření, že $A(1) < x < A(|S|)$, položíme $d = 1$ a $h = |S|$. Formální zápis algoritmu:

MEMBER(x):
if $x = A(1)$ **then**
  Výstup: $x \in S$ stop
else
  if $x < A(1)$ **then Výstup**: $x \notin S$ stop **else** $d = 1$ **endif**
endif
if $x = A(|S|)$ **then**
  Výstup: $x \in S$ stop
else
  if $x > A(|S|)$ **then Výstup**: $x \notin S$ stop **else** $h = |S|$ **endif**
endif
while $d + 1 < h$ **do**
  $n :=$ **next**(d, h)
  if $x = A(n)$ **then**
    Výstup: $x \in S$ stop
  else
    if $x < A(n)$ **then**
      $h := n$
    else
      $d := n$
    endif
  endif
enddo
Výstup: $x \notin S$ stop

V tomto algoritmu je **next**(d, h) funkce, která nalezne hodnotu n takovou, že $d < n < h$. Jeho korektnost plyne z pozorování, že když $d + 1 = h$, pak $A(d) < x < A(h)$ implikuje, že neexistuje i takové, že $x = A(i)$, a tedy $x \notin S$. Efektivita závisí na funkci **next**. Zpracování dotazu vyžaduje čas $O(1)$ a když i vyhodnocení funkce **next** vyžaduje čas $O(1)$, pak čas celého algoritmu je úměrný počtu dotazů, a ten je roven počtu volání funkce *next*.

Unární vyhledávání: Zde **next**$(d, h) = d + 1$. Každý dotaz zvětší d o 1, a tedy maximální

počet dotazů je $|S|$. Algoritmus v nejhorším případě vyžaduje čas $O(|S|)$ a očekávaný počet dotazů při rovnoměrném rozložení vstupních dat je $\frac{|S|}{2}$ (tedy i očekávaný čas je $O(|S|)$).

Poznámka: Duální přístup je, když $\mathbf{next}(d,h) = h-1$, výsledky se nezmění. Obecně se mluví o unárním vyhledávání, když existuje konstanta $c \geq 1$ taková, že buď $d < \mathbf{next}(d,h) \leq d+c$ nebo $h - c \leq \mathbf{next}(d,h) < h$. Jak uvidíme později, tento pojem je výhodné použít při různých aplikacích. O konstantě c se pak mluví jako o délce kroku.

Binární vyhledávání: Zde $\mathbf{next}(d,h) = \lceil \frac{d+h}{2} \rceil$. Každý dotaz zmenší rozdíl $h-d$ přibližně na polovinu. Když algoritmus pro zjištění, že $x \notin S$, potřeboval k dotazů, pak nutně $|S| \leq 2^{k-3}$ (dva úvodní dotazy, které zajišťují, že x není mimo hranice, tj. je v rozmezí $\min S$ a $\max S$, a poslední dotaz se nepočítají). Proto počet dotazů je nejvýše $3 + \log(|S| - 2)$. Algoritmus tedy v nejhorším případě vyžaduje čas $O(\log |S|)$ a očekávaný čas při rovnoměrném rozložení vstupních dat je také $O(\log |S|)$.

Interpolační vyhledávání: V této metodě $\mathbf{next}(d,h) = d + \lceil \frac{x-A(d)}{A(h)-A(d)}(h-d) \rceil$. V nejhorším případě musíme položit více než $\frac{|S|}{2}$ dotazů, a proto čas v nejhorším případě je $O(|S|)$, ale při rovnoměrném rozložení vstupů je očekávaný čas $O(\log\log |S|)$. To je založeno na faktu, že hodnota bf next závisí i na velikosti x. Když x je velké, tak hodnota **next** je posunuta do větších hodnot, když x je malé, pak je posunuta do menších hodnot.

Poznámka: Když rozložení prvků není rovnoměrné, ale je známé, pak podle toho můžeme upravit funkci **next** a v tom případě se odhad očekávaného času algoritmu výrazně nezmění.

Pro funkci **next** definovanou následujícím způsobem bude jednodušší spočítat očekávaný počet dotazů než v případě interpolačního vyhledávání, ale výsledek je asymptoticky stejný.

Zobecněné kvadratické vyhledávání. Funkce **next** je zde definována složitější procedurou, jejíž výsledek závisí i na předchozích dotazech a odpovědích na ně. Procedura pracuje v blocích a dotazy zadané v rámci téhož bloku jsou mezi sebou korelované. První dotaz v bloku je interpolační a procedura přitom zjistí, zda x je menší nebo větší než hodnota dotazu, a stanoví velikost kroku pro další přiblížení. Pak střídá unární a binární dotazy do té doby, než nalezne hledaný prvek nebo než skončí práce v tomto bloku. Blok končí, když rozdíl mezi h a d je nejvýše roven velikosti kroku. Krok v každém následujícím bloku klesne vždy přibližně na odmocninu velikosti kroku v předchozím bloku. K provedení tohoto postupu používá procedura pro výpočet funkce **next** boolské proměnné *blok*, *typ*, *smer* a numerickou proměnou *krok*. Proměnná *blok* je inicializována hodnotou $false$ a určuje, zda se dotaz zadává v rámci bloku nebo zda blok už skončil (v tom případě má hodnotu $false$). Proměnná *typ* určuje, zda příští dotaz je unární (když $typ = true$) nebo binární. Proměnná *smer* určuje, zda další dotazy budou vlevo ($smer = true$) nebo vpravo od prvního dotazu v bloku. Celočíselná proměnná *krok* určuje velikost kroku pro unární dotazy v rámci jednoho bloku. Hodnoty těchto proměnných se předávají z jednoho volání procedury do dalšího (tj. jsou to globální proměnné, které se neinicializují voláním procedury **next**).

```
   next(d, h):
if blok then
  if typ then
    if smer then
```

$next(d,h) := h - krok$
if $A(next(d,h)) < x$ **then** $blok := false$ **endif**
else
$next(d,h) := d + krok$
if $A(next(d,h)) > x$ **then** $blok := false$ **endif**
endif
$typ := false$
else
if $krok < \min\{\lceil \frac{d+h}{2} \rceil - d, h - \lceil \frac{d+h}{2} \rceil\}$ **then** $blok := false$ **endif**
$next(d,h) := \lceil \frac{d+h}{2} \rceil$
$typ := true$
endif
else
$krok := \lfloor \sqrt{h-d} \rfloor$
$next(d,h) := d + \lceil \frac{x-A(d)}{A(h)-A(d)}(h-d) \rceil$
if $A(next(d,h)) > x$ **then**
$smer := true$
else
$smer := false$
endif
$blok := true$
$typ := true$
endif

Provedeme podrobnou analýzu zobecněného kvadratického vyhledávání. Nejprve si všimněme, že i přes svou komplikovanost vyžaduje každé vyčíslení funkce *next* čas $O(1)$. Proto algoritmus vyžaduje čas úměrný počtu dotazů. Vlastní analýza tohoto faktu využije a odhadne potřebný počet dotazů a tím i časovou složitost.

Nejprve vypočteme očekávaný počet dotazů v jednom bloku. Nechť p_i je pravděpodobnost, že v rámci bloku se položí alespoň i dotazů. Pak očekávaný počet dotazů C v rámci bloku je

$$C = \sum_{i \geq 1} i(p_i - p_{i+1}) = \sum_{i \geq 1} p_i.$$

Nyní odhadneme p_i. Na začátku bloku označme: $n + d$ argument prvního (interpolačního) dotazu, k velikost kroku a X náhodnou proměnnou definovanou jako $X = |\{i \mid i > d,\ A(i) \leq x\}|$. Když se v bloku položí alespoň i dotazů, kde $i > 2$, pak $|X - n| \geq \lfloor \frac{i-2}{2} k \rfloor$, protože každý unární dotaz, po jehož položení neskončil blok, nalezl dalších k hodnot v rozdílu $|X - n|$. Proto pro dané i platí

$$p_i \leq \text{Prob}(|X - n| \geq \lfloor \frac{i-2}{2} k \rfloor) (= \text{Prob}(|X - n| \geq k (\text{počet unárních dotazů v bloku}))).$$

K odhadu použijeme Čebyševovu nerovnost pro náhodnou proměnnou X. Připomeňme, že když Y je náhodná proměnná s očekávanou (střední) hodnotou μ a rozptylem σ^2, pak

Čebyševova nerovnost říká, že

$$\text{Prob}(|Y-\mu| \geq t) \leq \frac{\sigma^2}{t^2} \qquad \text{pro každé } t > 0.$$

Uvažujme okamžik, kdy jsme na začátku nějakého bloku. Protože S je vybraná s rovnoměrným rozdělením, je pravděpodobnost, že $A(i) < x$ pro $d < i < h$, rovna $p = \frac{x-A(d)}{A(h)-A(d)}$, a pak pravděpodobnost, že $X = j$, je $\binom{h-d}{j}p^j(1-p)^{h-d-j}$. To znamená, že X je náhodná veličina s binomickým rozdělením, a tedy její očekávaná hodnota je

$$\mu = \sum_{j=0}^{h-d} \binom{h-d}{j} p^j(1-p)^{h-d-j} = p(h-d)$$

a rozptyl má hodnotu

$$\sigma^2 = \sum_{j=0}^{h-d} (j-\mu)^2 \binom{h-d}{j} p^j(1-p)^{h-d-j} = p(1-p)(h-d).$$

Když si uvědomíme, že $k = \lfloor\sqrt{h-d}\rfloor$ a $n = p(h-d)$, dostáváme

$$p_i, p_{i+1} \leq \text{Prob}(|X-n| \geq \lfloor\frac{i-2}{2}k\rfloor) \leq \frac{4p(1-p)(h-d)}{(i-2)^2k^2} \leq \frac{4p(1-p)}{(i-2)^2} \leq \frac{1}{(i-2)^2},$$

protože $p(1-p) \leq \frac{1}{4}$. Tedy

$$C = \sum_{i\geq 1} p_i \leq 2 + 2\sum_{i\geq 3} \frac{1}{(i-2)^2} = 2 + 2\sum_{i\geq 1}\frac{1}{i^2} = 2 + 2\frac{\pi^2}{6} = 2 + \frac{\pi^2}{3} \approx 5.3$$

Závěr: očekávaný počet dotazů v bloku je menší než 6.

Nyní vypočteme horní odhad počtu bloků. Víme, že na počátku byl $krok \leq \lfloor\sqrt{|S|}\rfloor \leq \sqrt{|S|}$, a když $krok_i$ je hodnota proměnné $krok$ v i-tém bloku, pak $krok_{i+1} \leq \lfloor\sqrt{krok}\rfloor \leq \sqrt{krok_i}$. Odtud dostáváme, že $krok_i \leq \sqrt[2^i]{|S|}$. Předpokládejme, že $2 = \sqrt[2^j]{|S|}$ pro nějaké j. Pak $krok_{\lceil j\rceil} \leq 2$, a tedy algoritmus projde nejvýše $\lceil j\rceil + 1$ bloků. Rovnice $2 = \sqrt[2^j]{|S|}$ je ekvivalentní s rovnicí $1 = \frac{1}{2^j}\log_2|S|$, což je dále ekvivalentní s $j = \log_2\log_2|S|$. Tedy počet bloků je nejvýše $1 + \log\log|S|$ a pro očekávaný počet dotazů $T(n)$ musí platí

$$T(n) \leq C + C\log\log|S|.$$

Na závěr odhadneme počet dotazů v nejhorším případě. Všimněme si, že interpolační dotaz se položí v každém bloku jen jeden a po unárním dotazu buď skončí blok nebo následuje binární dotaz. Protože binárních dotazů je nejvýše tolik, kolik jich je položeno při binárním vyhledávání, dostáváme, že binárních dotazů je nejvýše $3 + \log(|S|-1)$, unárních dotazů je nejvýše $3 + \log(|S|-1) + \log\log|S| + 1$ a interpolačních dotazů je nejvýše $1 + \log\log|S|$. Tedy celkem je položeno nejvýše $8 + 2\log(|S|-1) + 2\log\log|S|$ dotazů. Shrneme dosažené výsledky:

Věta. *Operace* **MEMBER** *při zobecněném kvadratickém vyhledávání vyžaduje v nejhorším případě čas* $O(\log n)$*, kde* n *je velikost reprezentované množiny. Očekávaný čas za předpokladu rovnoměrného rozložení vstupních dat je* $O(\log \log n)$*.*

Nevýhodou datové struktury uspořádané pole je nemožnost efektivní implementace aktualizačních operací. Každé přidání nebo ubrání prvku nás nutí posunout v průměru $\frac{|S|}{2}$ prvků, aby se zachovala struktura pole. To vede k tomu, že čas pro operace **INSERT** a **DELETE** je $O(|S|)$, a to je nevyhovující. Snaha odstranit tento problém vedla k používání binárních vyhledávacích stromům. Na druhé straně tyto stromy zobecňují pouze binární vyhledávání. Rysy interpolačního vyhledávání se do nich nepovedlo uspokojivým způsobem zabudovat.

IV. Binární vyhledávací stromy

Binární vyhledávací strom je struktura pro binární vyhledávání v uspořádaném poli roztaženém do roviny a vyhledávání odpovídá cestě ve stromě. Formální definice:

Nechť U je lineárně uspořádané univerzum a $S \subseteq U$. Binární vyhledávací strom T reprezentující množinu S je úplný binární strom (tj. každý vrchol je buď listem nebo má dva syny, levého a pravého), kde existuje bijekce key mezi množinou S a vnitřními vrcholy stromu taková, že

> když v je vnitřní vrchol stromu T, kterému je přiřazen prvek $s \in S$, pak každému vnitřnímu vrcholu u v podstromu levého syna vrcholu v je přiřazen prvek z S menší než s a každému vnitřnímu vrcholu w v podstromu pravého syna vrcholu v je přiřazen prvek z S větší než s.

Struktura vnitřního vrcholu v:
ukazatel otec(v) na otce vrcholu v,
ukazatel levy(v) na levého syna vrcholu v,
ukazatel pravy(v) na pravého syna vrcholu v,
atribut key(v) – prvek z S přiřazený vrcholu v.
Když v je kořen stromu, pak hodnota ukazatele otec(v) je NIL. List má ukazatele pouze na otce.

Každý list reprezentuje interval mezi dvěma sousedními prvky z S, a to tak, že: Když list u je levým synem vrcholu v, nalezneme nejbližší vrchol na cestě z u do kořene, který je pravým synem svého otce w. Pak list u reprezentuje interval $(\text{key}(w), \text{key}(v))$. Když takový vrchol w neexistuje, pak u reprezentuje interval $(-\infty, \text{key}(v))$ a prvek key(v) je nejmenší prvek v S. Analogicky když list u je pravým synem vrcholu v, nalezneme nejbližší vrchol na cestě z u do kořene takový, že je levým synem svého otce w. Pak u reprezentuje interval $(\text{key}(v), \text{key}(w))$ a když takový vrchol w neexistuje, u reprezentuje interval $(\text{key}(v), +\infty)$ a prvek key(v) je největší prvek v S.

Při implementaci binárních vyhledávácích stromů je úspornější vynechat listy (místo nich bude ukazatel NIL), protože nenesou žádnou novou informaci. Při návrhu algoritmů je však výhodné s listy pracovat (je to logičtější). Proto budeme vždy předpokládat, že stromy mají listy reprezentující intervaly.

Algoritmy

Navrhneme algoritmy pro binární vyhledávací stromy realizující operace z uspořádaného slovníkového problému.

Vyhledej(x):
t :=kořen stromu
while t není list a key(t) $\neq x$ **do**
  if key(t) $> x$ **then** t := levy(t) **else** t := pravy(t) **endif**
enddo

MEMBER(x):
Vyhledej(x)
if t není list **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

INSERT(x):
Vyhledej(x)
if t je list **then**
  t změníme na vnitřní vrchol, key(t) := x
  levy(t),pravy(t) := nové listy, otec(levy(t)) := otec(pravy(t)) := t
endif

DELETE(x):
Vyhledej(x)
if t není list **then**
  if levy(t) je list **then**
    otec(pravy(t)) := otec(t)
    if t = levy(otec(t)) **then**
      levy(otec(t)) := pravy(t)
    else
      pravy(otec(t)) := pravy(t)
    endif
    odstraníme vrcholy t a levy(t)
  else
    u := levy(t)
    while pravy(u) není list **do** u := pravy(u) **enddo**
    key(t) := key(u), otec(levy(u)) := otec(u)
    if u = levy(otec(u)) **then**
      levy(otec(u)) := levy(u)
    else
      pravy(otec(u)) := levy(u)
    endif
    odstraníme vrcholy u a pravy(u)
  endif
endif

MIN:
t :=kořen stromu
while levý syn t není list **do** $t :=$ levy(t) **enddo**
Výstup: key(t)

MAX:
t :=kořen stromu
while pravý syn t není list **do** $t :=$ pravy(t) **enddo**
Výstup: key(t)

SPLIT(x):
T_1 a T_2 jsou jednoprvkové stromy
u_i je ukazatel na kořen stromu T_i pro $i = 1, 2$
t :=kořen stromu T
while t není list a key$(t) \neq x$ **do**
  vytvoříme list v
  if key$(t) > x$ **then**
    $u :=$ levy(t), levy$(t) := v$, otec$(u) := NIL$, otec$(v) := t$
    t nahradí list stromu T_2, na který ukazuje ukazatel u_2
    a list odstraníme, $u_2 := v$
  else
    $u :=$ pravy(t), pravy$(t) := v$, otec$(u) := NIL$, otec$(v) := t$
    t nahradí list stromu T_1, na který ukazuje ukazatel u_1
    a list odstraníme, $u_1 := v$
  endif
  $t := u$
enddo
if key$(t) = x$ **then**
  Výstup: $x \in S$
  otec(levy(t)) := otec(u_1), pravy(otec(u_1)) := levy(t)
  otec(pravy(t)) := otec(u_2), levy(otec(u_2) := pravy(t), otec$(u_1) := NIL$, otec$(u_2) := NIL$
else
  Výstup: $x \notin S$
endif

Komentář: T_1 je binární vyhledávací strom reprezentující množinu $\{s \in S \mid s < x\}$ a T_2 je binární vyhledávací strom reprezentující množinu $\{s \in S \mid s > x\}$. Předpokládáme, že $\max S_1 < x < \min S_2$.

JOIN3(T_1, x, T_2):
vytvoříme nový vrchol u, key$(u) = x$, otec$(u) := NIL$
otec(kořen T_1) := x, otec(kořen T_2) := x
levy(u) :=kořen T_1, pravy(u) :=kořen T_2.

Hlavním problémem je přesný (a přesvědčující) důkaz korektnosti podprocedury **Vyhledej**, která je modifikací vyhledávání v uspořádaném poli. Kvůli tomu nejprve zavedeme pomocné značení. Mějme binární vyhledávací strom T reprezentující množinu S a nechť t je vrchol T. Nejprve definujme $\mathrm{lhr}(t)$ a $\mathrm{phr}(t)$. Když t je kořen, pak $\mathrm{lhr}(t)$ a $\mathrm{phr}(t)$ není definováno. Když t není kořen a vrchol v je jeho otec, pak definujeme

> když t je levý syn v, pak $v = \mathrm{phr}(t)$ a $\mathrm{lhr}(t) = \mathrm{lhr}(v)$ (když $\mathrm{lhr}(v)$ není definováno, pak ani $\mathrm{lhr}(t)$ není definováno);
> když t je pravý syn v, pak $v = \mathrm{lhr}(t)$ a $\mathrm{phr}(t) = \mathrm{phr}(v)$ (když $\mathrm{phr}(v)$ není definováno, pak ani $\mathrm{phr}(t)$ není definováno).

Když $\mathrm{lhr}(t)$ není definováno, pak $\lambda(t) = -\infty$, když $\mathrm{lhr}(t)$ je definováno, pak $\lambda(t) = \mathrm{key}(\mathrm{lhr}(t))$ a analogicky, kdyz $\mathrm{phr}(t)$ není definováno, pak $\pi(t) = +\infty$, když $\mathrm{phr}(t)$ je definováno, pak $\pi(t) = \mathrm{key}(\mathrm{phr}(t))$. Nyní dokážeme

Lemma. *Je-li T' podstrom binárního vyhledávacího stromu T určený vrcholem t, pak T' reprezentuje množinu $S \cap (\lambda(t), \pi(t))$. Navíc interval $(\lambda(t), \pi(t))$ je největší interval, který neobsahuje žádný prvek z S, který je reprezentován vrcholem T, který neleží v T'.*

Důkaz. Tvrzení dokážeme indukcí. Zřejmě platí, když t je kořen stromu T. Předpokládejme, že platí pro vrchol t a dokážeme ho pro syny vrcholu t. Označme t_l levého syna vrcholu t, t_p pravého syna vrcholu t. Z definice binárního vyhledávacího stromu stromu plyne, že když u je vnitřní vrchol v podstromu T určeném vrcholem t_l a když v je vnitřní vrchol v podstromu T určeném vrcholem t_p, pak $\mathrm{key}(u) < \mathrm{key}(t) < \mathrm{key}(v)$. Nyní platnost tvrzení pro t implikuje platnost tvrzení i pro vrcholy t_l a t_p. □

Korektnost podprocedury **Vyhledej** plyne z následujícího invariantu:

> Když při vyhledávání x vyšetřujeme vrchol t, pak
>
> $$\lambda(t) < x < \pi(t).$$

Toto tvrzení se lehce dokáže indukcí z popisu algoritmu **Vyhledej**. Tedy operace **Vyhledej** je korektní a korektnost operací **MEMBER** a **INSERT** je teď zřejmá. V operaci **DELETE**, když $\mathrm{levy}(t)$ je list, pak korektnost je zřejmá. Když $\mathrm{levy}(t)$ není list, pak algoritmus nalezne list v takový, že $v = \mathrm{pravy}(u)$ pro $u = \mathrm{otec}(v)$ a $\mathrm{lhr}(v) = u$, $\mathrm{phr}(v) = t$. Když $\mathrm{key}(u) = y$, pak $(y, x) \cap S = \emptyset$, protože v je list. Odstranění vrcholů u a v dává binární vyhledávací strom reprezentující $S \setminus \{y\}$. Protože $(y, x) \cap S = \emptyset$, tak příkaz $\mathrm{key}(t) := y$ dává binární vyhledávací strom reprezentující $S \setminus \{x\}$ a proto operace **DELETE** je korektní.

Korektnost operací **MIN**, **MAX** a **JOIN3** plyne z definice binárního vyhledavacího stromu. Korektnost operace **SPLIT** plyne z korektnosti algoritmu **Vyhledej** a z faktu, že u_1 je nejpravější list stromu T_1 a u_2 je nejlevější list stromu T_2. Protože ke stromu T_1 se přidává část stromu T reprezentující prvky, které jsou větší než prvky reprezentované v T_1 (přidávaná část stromu T je ve stromu T napravo od částí, které reprezentovaly prvky, které už jsou v T_1), a ke stromu T_2 se přidává část stromu T reprezentující prvky, které jsou menší než prvky reprezentované v T_2 (přidávaná část stromu T je ve stromu T nalevo od částí, které reprezentovaly prvky, které už jsou v T_2), korektnost algoritmu pro operaci **SPLIT** je jasná.

Zpracování jednoho vrcholu vyžaduje čas $O(1)$ a algoritmus se pohybuje po jedné cestě z kořene do nějakého listu. Označme vyska(T) délku nejdelší cesty z kořene do nějakého listu. Pak dostáváme:

Věta. *Algoritmy pro operace* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN3** *a* **SPLIT** *v binárním vyhledávacím stromu T vyžadují čas O*(vyska(T)).

Pořádkové statistiky

Bohužel ani struktura binárních vyhledávacích stromů nepodporuje efektivní implementaci operace **ord**(k). K tomu je nutné rozšířit datovou strukturu podobně jako u (a,b)-stromů, tj. u každého vrcholu t deklarovat také údaj $p(t)$ – počet listů v podstromu určeném vrcholem t. Po provedení operací **INSERT**, **DELETE**, **JOIN3** a **SPLIT** je pak nutné aktualizovat tuto položku na cestě z vrcholu do kořene. Následující algoritmus realizuje tuto operaci.

ord(k):
t :=kořen stromu
if $k \geq p(t)$ **then** k-tý prvek neexistuje, stop **endif**
while true **do**
  if $k > p(\text{levy}(t))$ **then**
    $k := k - p(\text{levy}(t))$, $t :=$ pravy(t)
  else
    if $k < p(\text{levy}(t)$ **then**
      $t :=$ levy(t)
    else
      Výstup: key(t), **stop**
    endif
  endif
enddo

Korektnost algoritmu plyne z následujícího invariantu:

> Když je v daném okamžiku v proměnné t vrchol v a hodnota proměnné k je k', pak k-tý prvek v S se rovná k'-tému prvku v intervalu reprezentovaném podstromem T určeným vrcholem v.

Protože algoritmus je inicializován tak, že v je kořen stromu, interval je S a $k' = k$, dostáváme, že na počátku běhu algoritmu invariant platí. Předpokládejme, že platí v daném kroku. Nechť u je levý syn v, w je pravý syn v. Nejprve si připomeňme, že pro každý vrchol stromu y je velikost množiny prvků reprezentovaných v podstromu T určeném vrcholem y rovna $p(y)-1$. Z definice binárního vyhledávacího stromu plyne, že když u' je vnitřní vrchol v podstromu T určeném vrcholem u a w' je vnitřní vrchol v podstromu T určeném vrcholem w, pak platí key(u') < key(v) < key(w'). Odtud plyne

> když $k' < p(u)$, pak k'-tý prvek v množině reprezentované v podstromu T určeném vrcholem u je zároveň k'-tý prvek v množině reprezentované podstromem T určeným vrcholem v;
> když $k' = p(u)$, pak key(v) je k'-tý prvek v množině reprezentované podstromem T určeným vrcholem v;

když $k' > p(u)$, pak k'-tý prvek v množině reprezentované podstromem T určeným vrcholem v je $(k' - p(u))$-tý prvek v množině reprezentované podstromem T určeným vrcholem w.

Odtud plyne, že invariant platí po provedení tohoto kroku. Tím jsme dokázali platnost invariantu a korektnost algoritmu. Podle stejných argumentů jako v předchozím případě dostáváme, že časová složitost algoritmu je $O(\text{vyska}(T))$. Můžeme tedy tato fakta shrnout.

Věta. *Algoritmy pro operace* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN3**, **SPLIT** *a* **ord**(k) *pro všechna k v rozšířených binárních vyhledávacích stromech vyžadují čas $O(\text{vyska}(T))$, kde T je reprezentující strom.*

Vyvážené binární vyhledávací stromy

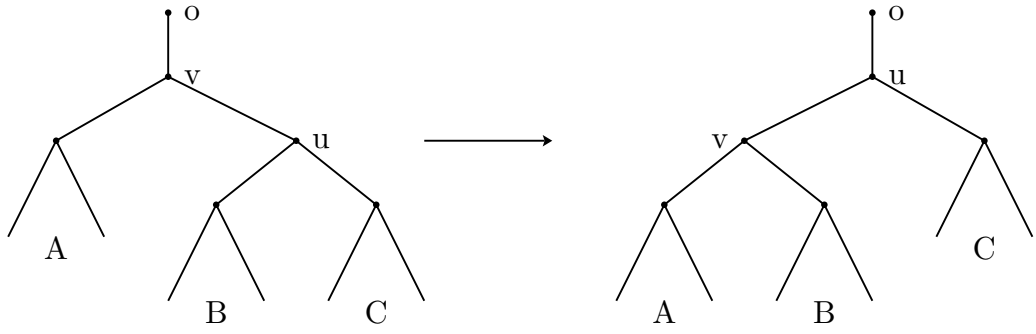
Předchozí výsledek motivuje používání binárních vyhledávacích stromů splňujících další podmínku, která má zajistit, že $\text{vyska}(T) = O(\log |S|)$. Pak mluvíme o vyvážených binárních vyhledávacích stromech. Je však nutné přinejmenším k operacím **INSERT**, **DELETE**, **JOIN3** a **SPLIT** přidat další kroky, které zaručí, že po jejich provedení strom opět splňuje požadované podmínky. To vede k požadavku, aby vyvažovací operace byly rychlé a provádělo se jich málo.

Při náhodné posloupnosti operací **INSERT** a **DELETE** a rovnoměrném rozložení jejich argumentů je velká pravděpodobnost, že dostaneme náhodný binární vyhledávací strom. Je známo, že očekávaná hodnota proměnné $\text{vyska}(T)$ je $O(\log |S|)$. Protože algoritmy pro obyčejné binární vyhledávací stromy nepoužívají žádné vyvažovací operace, můžeme dostat lepší (ve smyslu časové složitosti) výsledek než pro vyvážené binární vyhledávací stromy. Navíc vyvážené binární vyhledávací stromy vyžadují rozšíření datové struktury o položky, které umožní zjistit nevyváženost a tím spustí vyvažovací operace. To motivuje podrobně studovat porovnání obyčejných binárních vyvažovacích stromů a vyvážených binárních vyhledávacích stromů. Tento problém se v současné době intenzivně studuje. Velká pozornost je věnována také pravděpodobnostním algoritmům pro binární vyhledávací stromy, které nahrazují vyvažovací podmínky znáhodněním operací. Hledají se i další možnosti řešení tohoto problému.

Studují se například tzv. samoupravující struktury. Zde se pracuje s datovou strukturou bez dodatečných informací, ale operace nad ní provádějí vyvažování v závislosti na argumentu. Bylo dokázáno, že existuje strategie vyvažování, která zajišťuje dobré chování bez ohledu na vstupní data. Navíc se ukázalo, že při jistých aplikacích je tento přístup výhodnější než použití vyvážených binárních vyhledávacích stromů. Další navrhovaná strategie spočívá v tom, že se jen zjišťuje, zda datová struktura nemá výrazně špatné chování, a pokud ho má, nebo po dlouhé řadě úspěšných aktualizačních operací, se vybuduje nová datová struktura (s optimálním chováním). Třetí, poměrně stará, strategie je založena na znalosti rozdělení vstupních dat. Zde se datová struktura předem upravuje pro toto rozdělení. Ukazuje se, že v řadě situacích tyto strategie mají úspěch. Další podrobnosti budou v přednášce v letním semestru.

Nyní si ukážme dvě pomocné operace se stromy, na nichž je založeno vyvažování binárních vyhledávacích stromů. Obě operace se provedou v čase $O(1)$.

Mějme vrchol v binárního vyhledávacího stromu T a jeho syna u, který je vnitřním vrcholem. Pak **Rotace**(v, u) je znázorněna na obrázku (představuje jeden ze dvou zrcadlově symetrických případů) a provádí ji následující algoritmus.



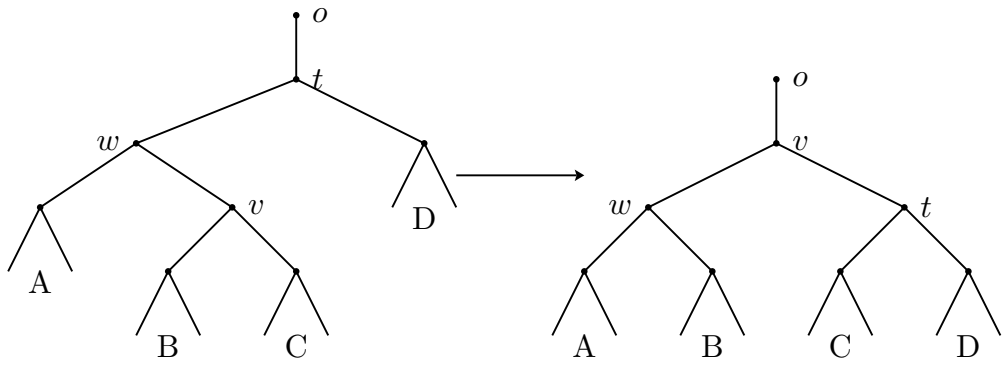
Obr. 1

Rotace(v, u):
otec(u) := otec(v),
if v = levy(otec(v) **then**
  levy(otec(v)) := u
else
  pravy(otec(v)) := u
endif
otec(v) := u
if u = levy(v) **then**
  otec(pravy(u) := v, levy(v) := pravy(u), pravy(u) := v
else
  otec(levy(u)) := v**,** pravy(v) := levy(u)**,** levy(u) := v
endif

Všimněme si, že při **Rotace** můžeme aktualizovat i funkci p. Pro vrchol $w \neq u, v$ se její hodnota nemění, nová hodnota $p(u)$ je rovna původní hodnotě $p(v)$ a novou hodnotu $p(v)$ dostaneme jako $p(\text{levy}(v)) + p(\text{pravy}(v))$.

Mějme dále vrchol w stromu T, jeho syna v a syna u vrcholu v takového, že u není list a je splněna podmínka, že v je pravý syn vrcholu w, právě když u je levý syn vrcholu v. Pak **Dvojita-rotace**(w, v, u) je znázorněna na obrázku (opět jeden ze dvou symetrických případů) a provádí ji následující algoritmus.

Dvojita-rotace(w, v, u):
otec(u) := otec(w)
if w = levy(otec(w) **then**
  levy(otec(w)) := u
else



Obr. 2

pravy(otec(w)) := u
endif
otec(v) := u, otec(w) := u
if v = levy(w) **then**
levy(w) := pravy(u), otec(pravy(u)) := w,
pravy(v) := levy(u), otec(levy(u)) := v,
levy(u) := v, pravy(u) := w
else
pravy(w) := levy(u), otec(levy(u)) := w,
levy(v) := pravy(u), otec(pravy(u)) := v,
levy(u) := w, pravy(u) := v
endif

Také zde můžeme v čase $O(1)$ spočítat nové hodnoty p. Pro vrchol $x \neq u, v, w$ se hodnota nemění, nová hodnota $p(u)$ je rovna původní hodnotě $p(w)$ a nové hodnoty $p(v)$ a $p(w)$ získáme podle stejného vzorce jako v **Rotace**.

V. AVL-stromy

Prvními vyváženými binárními vyhledávacími stromy byly AVL-stromy (AVL jsou začáteční písmena jmen autorů) a jejich obliba dodnes neklesá. Není se co divit, protože přes své stáří patří k nejefektivnějším vyváženým binárním vyhledávacím stromům a jejich definice je jednoduchá a průhledná, i když detailní provedení je technicky náročné. To může být hodně zrádné.

Binární vyhledávací strom je AVL-strom, když pro každý vnitřní vrchol v se délka nejdelší cesty z jeho levého syna do listu a délka nejdelší cesty z jeho pravého syna do listu liší nejvýše o 1 (ekvivalentně: výška jeho levého a pravého podstromu se liší nejvýše o 1).

Pro vnitřní vrchol v stromu T označme $\eta(v)$ délku nejdelší cesty z vrcholu v do listů.

Struktura vnitřních vrcholů v AVL-stromech je rozšířena o hodnotu ω definovanou jako:

$\omega(v) = -1$, když

$$\eta(\text{levý syn vrcholu } v) = \eta(\text{pravý syn vrcholu } v) + 1;$$

$\omega(v) = 0$, když

$$\eta(\text{levý syn vrcholu } v) = \eta(\text{pravý syn vrcholu } v);$$

$\omega(v) = +1$, když

$$\eta(\text{levý syn vrcholu } v) + 1 = \eta(\text{pravý syn vrcholu } v).$$

Všimněme si, že hodnota $\eta(v)$ pro vnitřní vrcholy v stromu T není nikde uložena. Hodnoty η jsme schopni spočítat z hodnot ω, ale není to třeba. Stačí, když budeme umět aktualizovat hodnoty ω a na základě toho dokážeme upravit binární vyhledávací strom tak, aby to byl opět AVL-strom.

Odhadneme velikost $\eta(\text{kořen } T)$ v závislosti na velikosti reprezentované množiny S.
Když T je AVL-strom a v je vnitřní vrchol T, pak podstrom T_v určený vrcholem v je opět AVL-strom. Označme
$mn(i)$ velikost nejmenší množiny reprezentované AVL-stromem T s kořenem t, pro který platí $\eta(t) = i$,
$mx(i)$ velikost největší množiny reprezentované AVL-stromem T s kořenem t, pro který platí $\eta(t) = i$.
Z definice AVL-stromu plynou následující rekurzivní vztahy

$$mn(i) = mn(i-1) + mn(i-2) + 1, \quad mx(i) = 2mx(i-1) + 1$$

s počátečními podmínkami

$$mn(1) = mx(1) = 1, \quad mn(2) = 2, \quad mx(2) = 3.$$

Nejprve spočítáme mx.

Dokážeme indukcí, že $mx(i) = 2^i - 1$. Tento vztah je zřejmě splněn pro $i = 1, 2$. Dále

$$mx(i+1) = 2mx(i) + 1 = 2(2^i - 1) + 1 = 2^{i+1} - 1.$$

Tím je vzorec dokázán.

Abychom spočítali mn, připomeneme si definici Fibonacciho čísel. Fibonacciho číslo F_i je definováno rekurencí

$$F_1 = F_2 = 1 \text{ a } F_{i+2} = F_i + F_{i+1} \text{ pro všechna } i \geq 3.$$

Pro každé $i = 1, 2, \ldots$ platí $F_i = \frac{(\frac{1+\sqrt{5}}{2})^i - (\frac{1-\sqrt{5}}{2})^i}{\sqrt{5}}$ (tento vztah bude dokázán v navazující části textu o haldách). Protože $-1 < \frac{1-\sqrt{5}}{2} < 0$ a $\frac{1+\sqrt{5}}{2} > 1$, dostáváme, že

$$\lim_{i \mapsto \infty} F_n \sqrt{5} (\frac{1+\sqrt{5}}{2})^{-n} = 1.$$

Proto existují konstanty $0 < c_1 < c_2$ takové, že

$$c_1(\frac{1+\sqrt{5}}{2})^i < \sqrt{5}F_i < c_2(\frac{1+\sqrt{5}}{2})^i.$$

Dokážeme, že $mn(i) = F_{i+2} - 1$. Protože $F_3 = 2$ a $F_4 = 3$, tvrzení platí pro $i = 1$ a $i = 2$. Dále

$$mn(i+2) = mn(i+1) + mn(i) + 1 = F_{i+3} - 1 + F_{i+2} - 1 + 1 = F_{i+4} - 1.$$

Z toho indukcí plyne požadovaný vztah.

Když AVL-strom T o výšce i reprezentuje množinu S o velikosti n, pak platí

$$\frac{c_1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{i+2} - 1 < F_{i+2} - 1 \leq n \leq 2^i - 1.$$

Po zlogaritmování z toho okamžitě dostáváme

$$\log(\frac{c_1}{\sqrt{5}}) + (i+2)\log(\frac{1+\sqrt{5}}{2}) < \log(n+1) < i.$$

Protože $\log(\frac{1+\sqrt{5}}{2}) \approx 0.69 \approx \frac{1}{1.44}$ dostáváme, že pro dostatečně velká n platí, že $0.69i < \log(n+1) \leq i$. Odtud plyne, že $\log(n+1) \leq i \leq 1.44\log(n+1)$, a tedy $i = \Theta(\log(n))$.

Algoritmy

Operace **MEMBER** pro AVL-stromy je stejná jako pro nevyvážené binární vyhledávací stromy. Aktualizační operace **INSERT** a **DELETE** v AVL-stromech nejprve provedou požadovanou operaci stejně jako v nevyvážených binárních vyhledávacích stromech a poté následuje jejich vyvažovací část.

Při úspěšně provedené operaci **INSERT**(x) v nevyvážených binárních vyhledávacích stromech změníme příslušný list t na vnitřní vrchol stromu reprezentující x a přidáme k t dva syny, kteří budou listy. Důsledkem toho je, že definujeme $\omega(t) = 0$. Protože se však zvětšila hodnota $\eta(t)$ (bylo $\eta(t) = 0$ a teď je $\eta(t) = 1$), zavoláme proceduru **Kontrola-INSERT**(t), která zajistí správnou hodnotu funkce ω pro otce t. Navíc, když zjistí, že se zvětšila hodnota η otce t, pak zavolá sama sebe na vrchol otec t. Nejprve neformálně popíšeme algoritmus pro **Kontrola-INSERT**(t).

Výchozí situace: máme vrchol t, do jehož podstromu jsme vložili nový prvek, jeho výšku $\eta(t) = a$ (ale a neznáme), která na začátku operace **INSERT** byla $\eta(t) = a - 1$. V podstromu určeném vrcholem t máme už správné hodnoty ω. Vrchol v je otcem t (bez újmy na obecnosti předpokládejme, že t je levý syn v) a $\omega(v)$ má ještě původní hodnotu. Platí

Lemma. *Když se hodnota $\eta(t)$ při operaci* **INSERT** *zvětšila a t nebyl listem před operací, pak po operaci je $\omega(t) \neq 0$.*

Důkaz. Předpokládejme, že $\eta(t) = b$ a t_1 a t_2 jsou synové vrcholu t. Když platí $\omega(t) = 0$, pak nutně $\eta(t_1) = \eta(t_2) = \eta(t) - 1 = b - 1$. Při operaci **INSERT** jsme mohli přidat vrchol jen do jednoho z podstromů určených vrcholy t_1 a t_2. Řekněme, že do podstromu vrcholu t_1. Před operací **INSERT** tedy muselo platit $\eta(t_1) + 1 = \eta(t_2) = \eta(t) - 1 = b - 1$, takže hodnota $\eta(t)$ se nemohla při operaci **INSERT** zvětšit. □

Označme u pravého syna vrcholu v. Pak nastávají následující případy:
A) Původní hodnota $\omega(v) = 1$ (tj. před operací **INSERT** bylo $\eta(t) = a - 1$, $\eta(u) = a$ a $\eta(v) = a + 1$). Po operaci **INSERT** je $\eta(t) = a$ a hodnota $\eta(v) = a + 1$ se nezměnila. Tedy stačí položit $\omega(v) = 0$ a můžeme skončit.
B) Původní hodnota $\omega(v) = 0$ (tj. před operací **INSERT** bylo $\eta(t) = a - 1$, $\eta(u) = a - 1$ a $\eta(v) = a$). Po operaci je $\eta(t) = a$ a hodnota $\eta(v) = a + 1$ se změnila. Tedy musíme položit $\omega(v) = -1$ a zavolat proceduru **Kontrola-INSERT** na vrchol v.
C) Původní hodnota $\omega(v) = -1$ (tj. před operací **INSERT** bylo $\eta(t) = a - 1$, $\eta(u) = a - 2$ a $\eta(v) = a$). Po operaci je $\eta(t) = a$ a hodnota $\eta(v) = a + 1$ se změnila. Nyní by mělo být $\omega(v) = -2$, ale to je zakázané. Předpokládejme, že t_1 je levý syn vrcholu t a t_2 je pravý syn vrcholu t. Podle lemmatu $\omega(t) = 0$ není možné (výška t se změnila). Máme tedy dvě možnosti:
C1) $\omega(t) = -1$. Pak platí $\eta(t_1) = a - 1$, $\eta(t_2) = a - 2$. Provedeme **Rotace**$(v, t,)$. Po ní t je kořen (dostane se na místo v), v je pravý syn t, t_2 je levý syn v a platí $\eta(t_2) = \eta(u) = a - 2$, $\eta(v) = \eta(t_1) = a - 1$ a $\eta(t) = a$ – tedy stejné jako bylo původní $\eta(v)$. Proto stačí položit $\omega(v) = \omega(t) = 0$ a můžeme skončit.
C2) $\omega(t) = 1$. Pak platí $\eta(t_1) = a - 2$, $\eta(t_2) = a - 1$. Označme t_3 levého syna t_2 a t_4 pravého syna t_2. Provedeme **Dvojita-rotace**(v, t, t_2), po které t_2 je kořen (dostane se na místo v), t je levý a v pravý syn t_2, t_3 je pravý syn t a t_4 levý syn v. Pak nastane jedna z možností:
C2i) Když $\omega(t_2) = 1$, pak platí $\eta(t_3) = a - 3$ a $\eta(t_4) = a - 2$. Po rotaci stačí položit $\omega(t) = -1$, $\omega(v) = \omega(t_2) = 0$ a můžeme skončit, protože $\eta(t_2) = a$ je stejné jako původní $\eta(v)$.
C2ii) Když $\omega(t_2) = 0$, pak platí $\eta(t_3) = \eta(t_4) = a - 2$. Po rotaci stačí položit $\omega(t_2) = \omega(v) = \omega(t) = 0$ a skončit, protože opět $\eta(t_2) = a$ je stejné jako původní $\eta(v)$.
C2iii) Když $\omega(t_2) = -1$, pak platí $\eta(t_3) = a - 2$ a $\eta(t_4) = a - 3$. Po rotaci stačí položit $\omega(v) = 1$, $\omega(t_2) = \omega(t) = 0$ a skončit, protože i v tomto případě je $\eta(t_2) = a$ stejné jako původní $\eta(v)$.
Když t je pravý syn v, pak situace je symetrická.
Toto je náplní podprocedury **Kontrola-INSERT**.

```
Kontrola-INSERT(t):
v := otec(t)
if t = levy(v) then
  if ω(v) = 1 then
    ω(v) := 0
  else
    if ω(v) = 0 then
      ω(v) := −1, t := v
```

```
      Kontrola-INSERT(t)
    else
      if ω(t) = −1 then
        Rotace(v, t), ω(v) := 0, ω(t) := 0
      else
        w := pravy(t), Dvojita-rotace(v, t, w),
        if ω(w) = 0 then
          ω(t) := 0, ω(v) := 0
        else
          if ω(w) = 1 then
            ω(v) := 0, ω(t) := −1
          else
            ω(v) := 1, ω(t) := 0
          endif
        endif
        ω(w) := 0
      endif
    endif
  endif
else
  if ω(v) = −1 then
    ω(v) := 0
  else
    if ω(v) = 0 then
      ω(v) := 1, t := v
      Kontrola-INSERT(t)
    else
      if ω(t) = 1 then
        Rotace(v, t), ω(v) := 0, ω(t) := 0
      else
        w := levy(t), Dvojita-rotace(v, t, w),
        if ω(w) = 0 then
          ω(t) := 0, ω(v) := 0
        else
          if ω(w) = 1 then
            ω(v) := 0, ω(t) := −1
          else
            ω(v) := 1, ω(t) := 0
          endif
        endif
        ω(w) := 0
      endif
    endif
  endif
endif
```

Všimněme si, že po provedením **Rotace** nebo **Dvojita-rotace** vyvažování v operaci **INSERT** končí. Tedy operace **INSERT** provádí nejvýše jednu proceduru **Rotace** nebo **Dvojita-rotace**. Korektnost vyvažovací operace je založena na Lemmatu, protože nenastane případ $\omega(t) = 0$. Dále si Všimněme, že kdyby mohlo být $\omega(t) = 0$, pak by algoritmus pro tento případ šel jen velmi obtížně doplnit.

Nyní popíšeme algoritmus pro operaci **DELETE**. Situace je podobná jako při **INSERT**. Předpokládejme, že t je vrchol, jehož otec byl odstraněn (tj. bratr t byl list) a hodnota $\eta(t)$ je menší než byla hodnota $\eta(\text{otec}(t))$. Proto zavoláme proceduru **Kontrola-DELETE**(t). Tato procedura zajistí správnou hodnotu funkce ω pro otce t. Navíc, když zjistí, že se zmenšila hodnota η otce t, pak zavolá sama sebe na vrchol otec t. Provedeme analogickou analýzu situace jako při operaci **INSERT**. Na ní je založena korektnost procedury **Kontrola-DELETE**. Pro operaci **DELETE** je však analýza komplikovanější.

Předpokládejme, že se provádí operace **DELETE** a t je vrchol, kde se hodnota $\eta(t)$ zmenšila. Přitom v podstromu určeném vrcholem t jsou hodnoty ω už aktualizovány, ale pro vrchol v, který je otcem t, je známá ještě původní hodnota $\omega(v)$. Předpokládejme, bez újmy na obecnosti, že t je levý syn vrcholu v a jeho nová hodnota $\eta(t) = a$ (přitom a je neznámé číslo). Označme u pravého syna vrcholu v. Pak nastávají následující případy:
A) Původní hodnota $\omega(v) = 1$ (tj. před operací **DELETE** platilo $\eta(t) = a+1$, $\eta(u) = a+2$ a $\eta(v) = a+3$). Označme u_1 levého syna vrcholu u a u_2 pravého syna vrcholu u. Jsou tři možnosti:
A1) $\omega(u) = 1$. Pak $\eta(u_1) = a$ a $\eta(u_2) = a+1$. Provedeme **Rotace**(v, u). Po ní u bude kořen (dostane se na místo v), v bude levý syn u, u_1 se stane pravým synem vrcholu v a platí $\eta(t) = \eta(u_1) = a$, $\eta(v) = \eta(u_2) = a+1$ a $\eta(u) = a+2$. Tedy položíme $\omega(v) = \omega(u) = 0$, ale protože $\eta(u)$ je menší než byla původní hodnota $\eta(v)$, musíme zavolat proceduru **Kontrola-DELETE** na vrchol u.
A2) $\omega(u) = 0$. Pak $\eta(u_1) = \eta(u_2) = a+1$. Provedeme opět **Rotace**(v, u) a po ní platí $\eta(t) = a$, $\eta(u_1) = a+1 = \eta(u_2)$, $\eta(v) = a+2$, $\eta(u) = a+3$. Položíme $\omega(v) = 1$, $\omega(u) = -1$ a protože hodnota $\eta(u)$ je stejná jako původní hodnota $\eta(v)$, končíme.
A3) $\omega(u) = -1$. Pak $\eta(u_1) = a+1$ a $\eta(u_2) = a$. Označme u_3 levého syna vrcholu u_1 a u_4 pravého syna vrcholu u_1. Provedeme **Dvojita-rotace**(v, u, u_1), po které u_1 bude kořen (dostane se na místo v), v bude levý a u pravý syn u_1, u_3 bude pravý syn v a u_4 levý syn u. Podle $\omega(u_1)$ (před rotací) nastanou případy:
A3i) $\omega(u_1) = -1$. Pak platí $\eta(u_3) = a$, $\eta(u_4) = a-1$ a tedy po rotaci $\eta(v) = a+1$, $\eta(u) = a+1$ a $\eta(u_1) = a+2$. Proto položíme $\omega(v) = \omega(u_1) = 0$, $\omega(u) = 1$ a zavoláme proceduru **Kontrola-DELETE** na vrchol u_1, protože $\eta(u_1)$ je menší než byla původní hodnota $\eta(v)$.
A3ii) $\omega(u_1) = 0$. Pak platí $\eta(u_3) = \eta(u_4) = a$ a tedy po rotaci $\eta(v) = a+1$, $\eta(u) = a+1$ a $\eta(u_1) = a+2$. Proto položíme $\omega(v) = \omega(u_1) = \omega(u) = 0$ a zavoláme proceduru **Kontrola-DELETE** na vrchol u_1, protože $\eta(u_1)$ je menší než byla původní hodnota $\eta(v)$.
A3iii) $\omega(u_1) = 1$. Pak platí $\eta(u_3) = a-1$, $\eta(u_4) = a$ a tedy po rotaci $\eta(v) = a+1$, $\eta(u) = a+1$ a $\eta(u_1) = a+2$. Proto položíme $\omega(u) = \omega(u_1) = 0$, $\omega(v) = -1$ a zavoláme proceduru **Kontrola-DELETE** na vrchol u_1, protože hodnota $\eta(u_1)$ je opět menší než byla původní hodnota $\eta(v)$.
B) Původní hodnota $\omega(v) = 0$ (tj. před operací **DELETE** platilo $\eta(t) = a+1$, $\eta(u) = a+1$ a $\eta(v) = a+2$). Nyní stačí položit $\omega(v) = 1$ a skončit, protože se hodnota $\eta(v)$ nezměnila.

C) Původní hodnota $\omega(v) = -1$ (tj. před operací **DELETE** platilo $\eta(t) = a + 1$, $\eta(u) = a$ a $\eta(v) = a + 2$). V tomto případě položíme $\omega(v) = 0$ a zavoláme proceduru **Kontrola-DELETE** na vrchol v, protože se jeho hodnota $\eta(v)$ zmenšila.

Všimněme si, že když procedura **Kontrola-DELETE** přesune vrchol x na místo vrcholu y, pak skutečná hodnota $\eta(x)$ je buď původní hodnota $\eta(y)$ nebo je přesně o 1 menší. Na tomto faktu je založena analýza situace a je důležitý pro korektnost operace **DELETE**. Nyní popíšeme algoritmus pro **Kontrola-DELETE**(t) formálně.

Kontrola-DELETE(t):
$v := \text{otec}(t)$
if $t = \text{levy}(v)$ **then**
  if $\omega(v) = 1$ **then**
    $u := \text{pravy}(v)$
    if $\omega(u) \geq 0$ **then**
      Rotace(v, u)
      if $\omega(v) = 0$ **then**
        $\omega(v) := 1$, $\omega(u) := -1$
      else
        $\omega(u) := \omega(v) := 0$, $t := u$, **Kontrola-DELETE**(t)
      endif
    else
      $w := \text{levy}(u)$, **Dvojita-rotace**(v, u, w)
      if $\omega(w) = 1$ **then**
        $\omega(u) := 0$, $\omega(v) := -1$
      else
        if $\omega(w) := 0$ **then**
          $\omega(u) := 0$, $\omega(v) := 0$
        else
          $\omega(u) := 1$, $\omega(v) := 0$
        endif
      endif
      $\omega(w) := 0$, $t := w$, **Kontrola-Delete**(t)
    endif
  else
    if $\omega(v) = 0$ **then**
      $\omega(v) := 1$
    else
      $\omega(v) := 0$, $t := v$, **Kontrola-DELETE**(t)
    endif
  endif
else
  if $\omega(v) = -1$ **then**
    $u := \text{levy}(v)$
    if $\omega(u) \leq 0$ **then**
      Rotace(v, u)

```
      if ω(u) = 0 then
        ω(v) := −1, ω(u) := 1
      else
        ω(u) := ω(v) := 0, t := u, Kontrola-DELETE(t)
      endif
    else
      w := pravy(u), Dvojita-rotace(v, u, w)
      if ω(w) = 1 then
        ω(u) := −1, ω(v) := 0
      else
        if ω(w) := 0 then
          ω(u) := 0, ω(v) := 0
        else
          ω(u) := 0, ω(v) := 1
        endif
      endif
      ω(w) := 0, t := w, Kontrola-Delete(t)
    endif
  else
    if ω(v) = 0 then
      ω(v) := −1
    else
      ω(v) := 0, t := v, Kontrola-DELETE(t)
    endif
  endif
endif
```

Složitost operací

V operaci **DELETE** se může stát, že procedury **Rotace** nebo **Dvojita-rotace** jsou volány až $\log(|S|)$-krát. To je výrazný rozdíl oproti operaci **INSERT**. Proto je operace **DELETE** pomalejší než operace **INSERT**, i když asymptoticky jsou stejně rychlé. Korektnost plyne z provedené analýzy situace při operaci **DELETE**. Shrneme uvedené výsledky.

Věta. *Implementace operací* **MEMBER**, **INSERT** *a* **DELETE** *pro datovou strukturu AVL-stromů vyžadují v nejhorším případě čas* $O(\log(|S|))$, *kde* S *je reprezentovaná množina. Operace* **INSERT** *zavolá nejvýše jednou proceduru* **Rotace** *nebo* **Dvojita-rotace**.

VI. Červeno-černé stromy

Všeobecně uznávanou nejefektivnější variantou vyvážených binárních vyhledávacích stromů jsou červeno-černé stromy. Například ve srovnání s AVL-stromy, které pro zadání pomocné struktury potřebují tříhodnotovou proměnnou (funkce ω), červeno-černým stromům stačí boolská dvouhodnotová proměnná (barva). Operace **DELETE** v červeno-černých stromech volá nejvýše dvakrát pomocnou proceduru **Rotace** nebo jednou pomocnou proceduru **Rotace** a jednou **Dvojita-rotace**, kdežto AVL-stromy mohou vyžadovat až $\log|S|$ volání

pomocných vyvažovacích procedur. Definice AVL-stromů je sice průzračnější, zato algoritmy pro červeno-černé stromy jdou pěkně popsat obrázky. Pro červeno-černé stromy lze vyvažování také provádět shora dolů, viz přednáška v letním semestru. Tam si také ukážeme, že přes tyto rozdíly jsou AVL stromy speciálním případem červeno-černých stromů. Nyní budeme definovat červeno-černé stromy formálně.

Binární vyhledávací strom T reprezentující množinu S, jehož vrcholy jsou obarveny červeně nebo černě (každý vrchol má právě jednu barvu) tak, že jsou splněny podmínky:

listy jsou obarveny černě,
když v je vrchol obarvený červeně, pak je buď kořen stromu nebo jeho otec je obarven černě,
všechny cesty z kořene do listů mají stejný počet černých vrcholů

se nazývá červeno-černý strom.

Nejprve ukážeme, že pro červeno-černý strom T reprezentující množinu S platí $\text{vyska}(T) = O(\log(|S|))$. Odtud plyne, že červeno-černé stromy patří mezi vyvážené binární vyhledávací stromy. Předpokládejme, že T je červeno-černý strom, který má na cestě z kořene do listu právě k černých vrcholů. Pak pro počet vrcholů $\#T$ stromu T platí

$$2^k - 1 \leq \#T \leq 2^{2k} - 1.$$

Nejmenší takový strom má všechny vrcholy obarvené černě a je to úplný pravidelný binární strom o vysce $k-1$, což dává dolní odhad. Největší takový strom má všechny vrcholy v sudých hladinách obarveny červeně a v lichých hladinách černě, je to úplný pravidelný binární strom o vysce $2k-1$ a tím je dán horní odhad. Tedy $k \leq 1 + \log \#T$. Protože velikost množiny S je počet vnitřních vrcholů, dostáváme, že $\#T = 2|S| + 1$. Z vlastností červeno-černých stromů plyne, že

$$k \leq \text{vyska}(T) \leq 2k.$$

Shrneme tato fakta.

Tvrzení. *Červeno-černý strom T reprezentující množinu S splňuje* $\text{vyska}(T) \leq 4 + 2\log|S|$.

Pro červeno-černé stromy navrhneme algoritmy realizující operace z uspořádaného slovníkového problému. Operace **MEMBER** je stejná jako pro nevyvážené binární vyhledávací stromy. Ooperace **INSERT** a **DELETE** (stejně jako pro AVL-stromy) mají dvě části: nejprve se provede operace **INSERT** nebo **DELETE** jako v nevyvážených binárních vyhledávacích stromech a pak následují vyvažovací operace, které zajistí, že výsledný strom splňuje podmínky pro červeno-černé stromy. Schéma operací **JOIN** a **SPLIT** bude vycházet z jejich realizací v (a,b)-stromech. V operaci **JOIN** prohledáváním nalezneme místo, kde se stromy dají spojit (tam aplikujeme operaci **JOIN** pro nevyvážené binární vyhledávací stromy), a pak použijeme vyvažovací operace. Algoritmus operace **SPLIT** rozdělí červeno-černý strom do několika menších červeno-černých stromů průchodem původního červeno-černého stromu podél cesty vyhledávající x (podobně jako v (a,b)-stromech) a na tyto stromy pak aplikuje operaci **JOIN** a zkonstruuje výsledné červeno-černé stromy.

Vyvažování

Nejprve popíšeme vyvažovací operaci pro **INSERT**. Dvojice (T, v) se nazývá 2-parciální červeno-černý strom, když T je binární vyhledávací strom, kde každý vrchol je obarven právě jednou z dvojice barev červená – černá, v je vnitřní vrchol stromu T obarvený červeně a platí:

listy jsou obarveny černě,
když t je vrchol obarvený červeně, pak je buď kořen stromu nebo $t = v$ nebo jeho otec je obarven černě,
všechny cesty z kořene do listů mají stejný počet černých vrcholů.

Předpokládejme, že T je červeno-černý strom reprezentující množinu S a provádíme operaci **INSERT**(x) pro $x \notin S$. Když operace **INSERT**(x) pro nevyvážené binární vyhledávací stromy vytvoří strom T', kde vrchol v reprezentuje x, pak v přebarvíme červeně a syny vrcholu v (jsou to listy) obarvíme černě. Je okamžitě vidět, že (T', v) je 2-parciální červeno-černý strom.

Na 2-parciální červeno-černý strom (T', v) zavoláme proceduru **Vyvaz-INSERT**(v). Po jejím provedení buď dostaneme červeno-černý strom nebo znovu zavoláme proceduru **Vyvaz-INSERT**(v') na vrchol v' takový, že (T', v') je 2-parciální červeno-černý strom a v' je děd v (tj. je o dvě hladiny blíž ke kořeni než vrchol v). Navíc, když procedura **Vyvaz-INSERT** volá sama sebe, tak do té doby neměnila strukturu stromu, nevolala ani pomocnou proceduru **Rotace** ani **Dvojita-rotace**, jen měnila obarvení vrcholů.

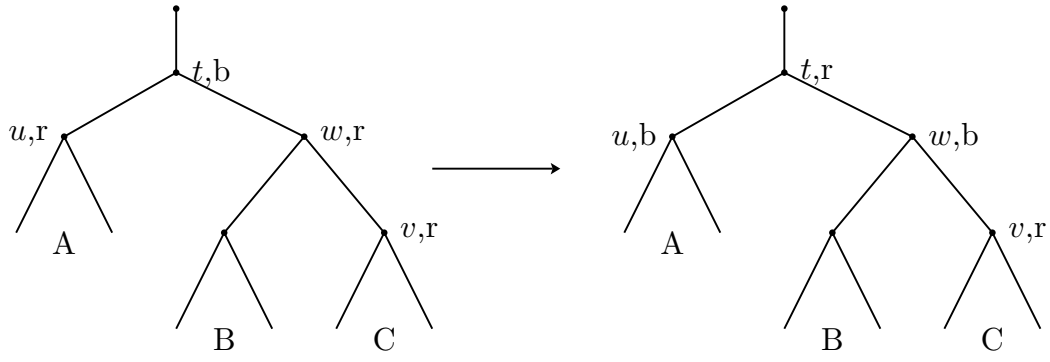
Červeno-černý strom zadáme tak, že rozšíříme strukturu vrcholu v o boolskou proměnnou $b(v)$, kde $b(v) = 0$ znamená, že v je obarven červeně, a $b(v) = 1$ znamená, že v je obarven černě.

Popíšeme proceduru **Vyvaz-INSERT** (předpokládáme, že v je obarven červeně). Pro jednoduchost označme $s(v) = levy$, když $v = \text{levy}(\text{otec}(v))$, a $s(v) = pravy$, když $v = \text{pravy}(\text{otec}(v))$.

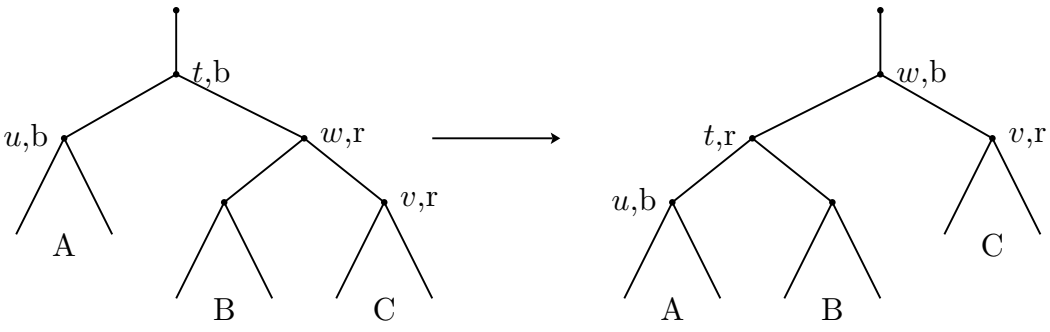
Vyvaz-INSERT(v):
if v není kořen T' a $b(\text{otec}(v)) = 0$ **then**
  if *otec*(v) je kořen **then**
    $b(\text{otec}(v)) := 1$
  else
    $w := \text{otec}(v)$, $u := \text{bratr}(w)$
    if $b(u) = 0$ **then**
      $v := \text{otec}(w)$, $b(w) := 1$, $b(u) := 1$, $b(v) := 0$
      Vyvaz-INSERT(v) (Komentář: Viz Obr. 1)
    else
      $t := \text{otec}(w)$
      if $s(w) = s(v)$ **then**
        Rotace(t, w), $b(t) := 0$, $b(w) := 1$ (Komentář: Viz Obr. 2)
      else
        Dvojita-rotace(t, w, v)

$b(t) := 0$, $b(v) := 1$ (Komentář: Viz Obr. 3)
endif
endif
endif
endif

Na obrázcích b značí černou barvu a r značí červenou barvu. Otec vrcholu w je označen t.



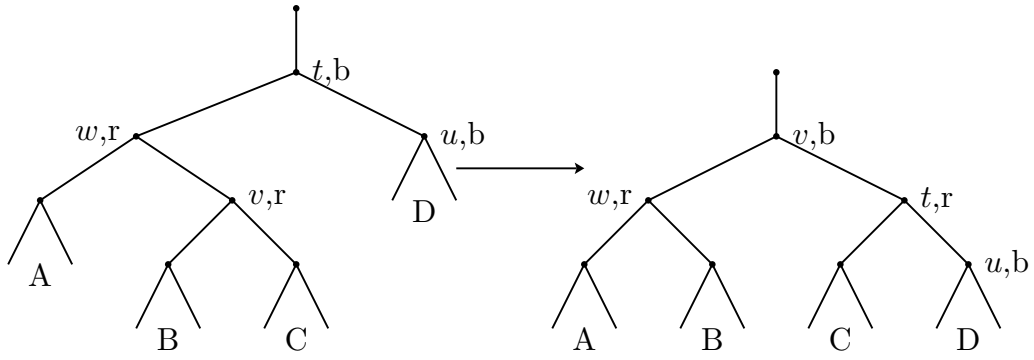
Obr. 1



Obr. 2

Operace **INSERT** v červeno-černých stromech volá nejvýše $2 + \log(|S|)$-krát proceduru **Vyvaz-INSERT** a provede nejvýše jednou **Rotace** nebo **Dvojita-rotace**. Podproceduru **Vyvaz-INSERT** používá při vyvažování také operace **JOIN**.

Operace **DELETE** je řešena podobným způsobem jako operace **INSERT**. Rozdíl je v tom, že při operaci **DELETE** je porušena třetí podmínka v definici červeno-černých stromů a vyvažování je technicky náročnější. Nejprve budeme definovat 3-parciální červeno-černý strom. Tato definice ukáže, jakým způsobem se poruší struktura červeno-černých stromů při operaci **DELETE**.



Obr. 3

Řekneme, že dvojice (T, v) je 3-parciální červeno-černý strom, když T je binární vyhledávací strom reprezentující množinu S, každému vrcholu je přiřazena právě jedna z dvojice barev červená – černá, v je vrchol ve stromu T a platí následující podmínky:

listy a vrchol v jsou obarveny černě,
když t je vrchol obarvený červeně, pak je buď kořen stromu nebo jeho otec je obarven černě,
existuje číslo k takové, že všechny cesty z kořene do listů, které neprocházejí vrcholem v, obsahují právě k černých vrcholů, a všechny cesty z kořene do listů procházející vrcholem v obsahují $k - 1$ černých vrcholů.

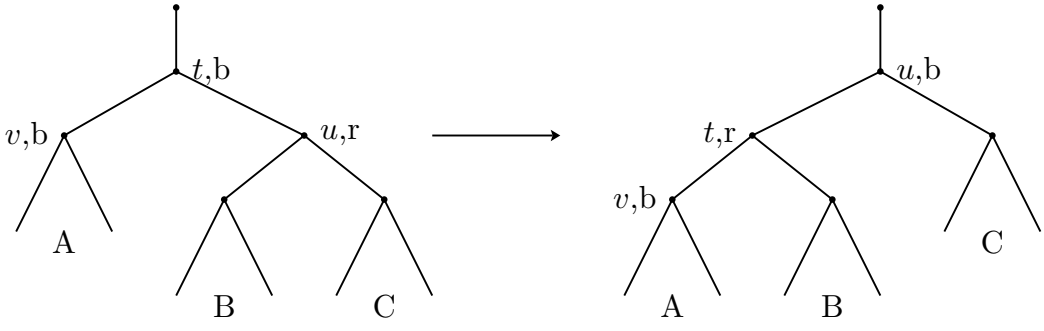
Předpokládejme, že provádíme operaci **DELETE**(x). V její první části použijeme algoritmus pro **DELETE** v nevyvážených binárních vyhledávacích stromech. Když tato část odstraní vrchol u a jeho syna w, který je list, pak na místo vrcholu u se dostane jeho druhý syn v. Pak nastane druhá část, jejímž cílem je vytvořit znovu červeno-černý strom. Na začátku této fáze obarvíme vrchol v černě. Pak jsou splněny první dvě podmínky v definici červeno-černých stromů a pokud vrchol u nebo vrchol v byl původně obarven červeně, je splněna i třetí podmínka. Pokud vrchol u i vrchol v byly obarveny černě, pak každá cesta z kořene do listu obsahující vrchol v má o jeden černý vrchol méně než cesta z kořene do listu neobsahující vrchol v (chybí černý vrchol u). Po provedení těchto akcí jsme buď obdrželi červeno-černý strom (a tím se úspěšně ukončila operace **DELETE**) nebo (T, v) je 3-parciální červeno-černý strom.

Předchozí analýza dává rychlý test na to, zda vznikne červeno-černý strom nebo 3-parciální červeno-černý strom (pak v je list). Na 3-parciální červeno-černý strom (T, v) pak aplikujeme podproceduru **Vyvaz-DELETE**(v), jejíž cílem je transformovat 3-parciální červeno-černý strom (T, v) do červeno-černého stromu tak, že se nezmění reprezentovaná množina.

Popíšeme proceduru **Vyvaz-DELETE**(v). Předpokládáme, že (T, v) je 3-parciální červeno-černý strom a že v není jeho kořen. Výsledkem procedury bude buď červeno-černý strom nebo 3-parciální červeno-černý strom (T', v'), kde v' je otcem vrcholu v, a zavolání procedury **Vyvaz-DELETE**(v') na strom (T', v'). Splnění požadavku, abychom po provedení procedury **Vyvaz-DELETE**(v) na 3-parciální červeno-černý strom (T, v) obdrželi červeno-černý strom, plyne z faktu, že když (T, v) je 3-parciální červeno-černý strom a v je jeho kořen, pak T je červeno-černý strom.

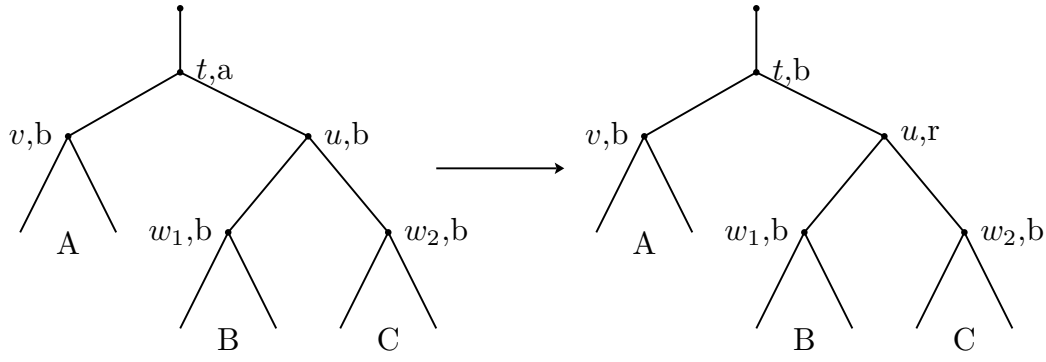
Vyvaz-DELETE(v):
$u :=$ bratr(v), $t :=$ otec(v)
if $b(u) = 0$ **then**
  Rotace(t, u), $b(u) := 1$, $b(t) := 0$, $u :=$ bratr(v)
endif
(Komentář: Viz Obr. 4, nyní $b(u) = 1$)
$w_1 :=$syn u takový, že $s(v) = s(w_1)$, $w_2 :=$ bratr(w_1)
if $b(w_1) = b(w_2) = 1$ **then**
  $b(u) := 0$
  if $b(t) = 0$ **then**
    $b(t) := 1$
  else
    if t není kořen stromu **then**
      $v := t$, **Vyvaz-DELETE**(v)
    endif
  endif (Komentář: Viz Obr. 5)
else
  if $b(w_1) = 1$ **then** (Komentář: $b(w_2) = 0$)
    Rotace(t, u), $b(w_2) := 1$, $b(u) := b(t)$, $b(t) := 1$ (Komentář: Viz Obr. 6)
  else
    Dvojita-rotace(t, u, w_1), $b(w_1) := b(t)$, $b(t) := 1$ (Komentář: Viz Obr. 7)
  endif
endif

Na následujících obrázcích jsou vrcholy, které nemají specifikovanou barvu (mohou být jak červené tak černé). Tyto barvy budeme označovat a, a'. Důvodem je, že se tato barva může přenést do cílového stromu buď tak, že si ji vrchol ponechá, nebo ji zdědí jiný vrchol, který se dostal na jeho místo. Na Obr. 5 se taková barva a v cílovém stromě neobjevuje.

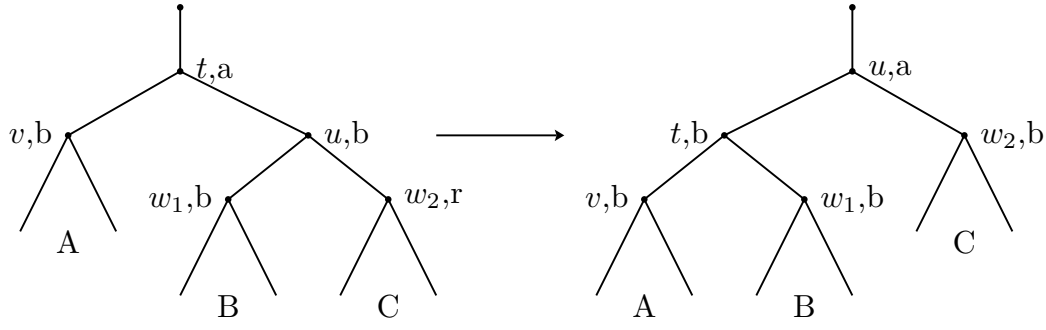


Obr. 4

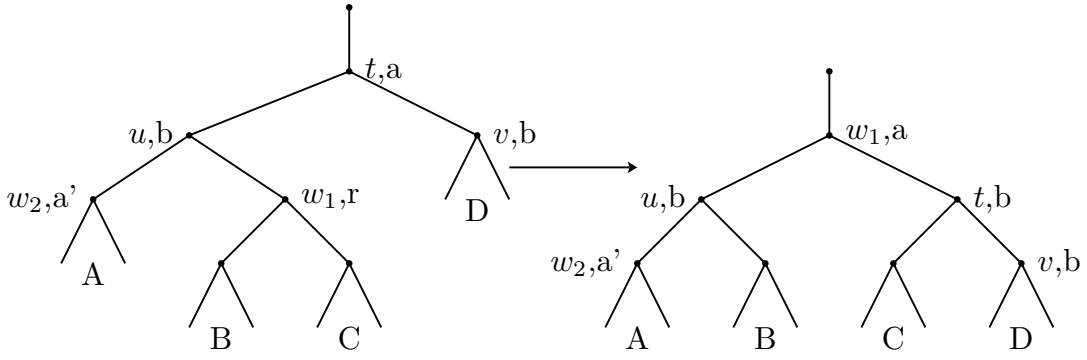
Korektnost algoritmů je zřejmá z obrázků. Všimněme si, že když u (bratr vrcholu v v původním stromu) je obarven červeně, pak po provedení **Rotace**(t, u) bude (T, v) opět 3-parciální červeno-černý strom a vrchol t (otec vrcholu v) bude obarven červeně. Pak z



Obr. 5



Obr. 6



Obr. 7

Obr. 5 je vidět, že pokračování procedury **Vyvaz-DELETE** může provést podproceduru **Rotace** nebo **Dvojita-rotace**, které skončí červeno-černým stromem.

Algoritmy operací

Nyní popíšeme algoritmy realizující operace z uspořádaného slovníkového problému. Neformální popis algoritmů operací **INSERT** a **DELETE** jsme už uvedli, nyní ho zformalizujeme.

INSERT(x):
Vyhledej(x)
if t je list **then**
  změníme t na vnitřní vrchol, key(t) := x
  vytvoříme syny levy(t) a pravy(t)
  $b(t) := 0$, b(levy(t)) := 1, b(pravy(t)) := 1
  Vyvaz-INSERT(t)
endif

DELETE(x):
Vyhledej(x)
if t není list **then**
  $vyv := false$
  if levy(t) je list **then**
    v := pravy(t)
    if $b(t) = 1$ a $b(v) = 1$ **then**
      $vyv := true$
    endif
    odstraníme vrchol levy(t), otec(v) := otec(t)
    if t = levy(otec(t)) **then**
      levy(otec(t)) := v
    else
      pravy(otec(t)) := v
    endif
    $b(v) := 1$, odstraníme vrchol t
  else
    u := levy(t)
    while pravy(u) není list **do**
      u := pravy(u)
    enddo
    key(t) := key(u), v := levy(u),
    if $b(u) = 1$ a $b(v) = 1$ **then**
      $vyv := true$
    endif
    $b(v) := 1$, odstraníme vrchol pravy(u),
    otec(v) := otec(u)
    if u = levy(otec(u)) **then**
      levy(otec(u)) := v
    else
      pravy(otec(u)) := v

endif
odstraníme vrchol u
endif
if vyv **then Vyvaz-DELETE**(v) **endif**
endif

Algoritmy pro operace **MIN** a **MAX** jsou stejné jako pro nevyvážené binární vyhledávací stromy, proto je vynecháme. Zbývá popsat operace **JOIN** (pro tuto strukturu vezmeme verzi **JOIN3**) a **SPLIT**.

Nejprve popíšeme algoritmus pro operaci **JOIN3**. Mějme červeno-černé stromy T_1 a T_2 reprezentující množiny S_1 a S_2 a dále prvek $x \in U$ takový, že $\max S_1 < x < \min S_2$. Nejprve zajistíme, že kořeny T_1 i T_2 jsou obarveny černě. Předpokládejme, že k_i je počet černých vrcholů na cestě z kořene do listů ve stromě T_i, kde $i = 1, 2$. Když $k_1 = k_2$, pak stačí provést **JOIN3**(T_1, x, T_2) pro nevyvážené binární vyhledávací stromy (kořen obarvíme červeně). Problém je, když $k_1 \neq k_2$. Například předpokládejme, že $k_1 > k_2$. Pak začneme v kořeni stromu T_1 a jdeme po pravých synech dolů tak dlouho, až nalezneme černý vrchol v takový, že všechny cesty z v do listů v T_1 obsahují právě k_2 černých vrcholů. Pak provedeme **JOIN3** pro nevyvážené binární vyhledávací stromy na podstrom T_1 určený vrcholem v a na T_2. Kořen w vzniklého stromu obarvíme červeně a tento strom vložíme do T_1 místo podstromu určeného vrcholem v. Pak (T_1, w) je 2-parciální červeno-černý strom a aplikujeme proceduru **Vyvaz-INSERT**. Případ $k_2 > k_1$ se řeší symetricky.

Algoritmus pro operaci **SPLIT** je velmi podobný algoritmu pro (a, b)-stromy. Vyhledáváme vrchol reprezentující x. Když jsme ve vrcholu t a máme pokračovat akcí $t :=$ levy(t), pak dvojici key(t) a podstrom T určený pravým synem t vložíme do zásobníku Z_2, když máme pokračovat akcí $t :=$ pravy(t), pak do zásobníku Z_1 vložíme dvojici podstrom T určený levým synem t a key(t). Když key(t) $= x$, pak do Z_1 vložíme podstrom určený levým synem t a do Z_2 podstrom určený pravým synem t. Když t je list, pak do Z_1 i Z_2 vložíme jednoprvkové stromy. Pomocí posloupnosti operací **JOIN3** vytvoříme ze zásobníku Z_1 strom T_1 a ze zásobníku Z_2 strom T_2.

JOIN3(T_1, x, T_2):
if b(kořen T_1) $= 0$ **then** b(kořen T_1) $:= 1$ **endif**
if b(kořen T_2) $= 0$ **then** b(kořen T_2) $:= 1$ **endif**
$k_1 :=$ počet černých vrcholů v T_1 na cestě z kořene do listů
$k_2 :=$ počet černých vrcholů v T_2 na cestě z kořene do listů
if $k_1 \geq k_2$ **then**
$t :=$kořen stromu T_1, $i := k_1 - k_2$
while $i > 0$ **do**
$t :=$ pravy(t)
if $b(t) = 1$ **then** $i := i - 1$ **endif**
enddo
vytvoř vrchol u, $b(u) := 0$, key(u) $:= x$
if t není kořen T_1 **then**
otec(u) $:=$ otec(t), pravy(otec(t)) $:= u$
endif

otec(t) := u, otec(kořen T_2) := u
pravy(u) :=kořen T_2, levy(u) := t
Vyvaz-INSERT(T_1, u)
else
t :=kořen T_2, $i := k_2 - k_1$
while $i > 0$ **do**
t := levy(t)
if $b(t) = 1$ **then** $i := i - 1$ **endif**
enddo
vytvoř vrchol u, $b(u) := 0$, key(u) := x
otec(u) := otec(t), levy(otec(t)) := u, otec(t) := u
otec(kořen T_1) := u, levy(u) :=kořen T_1
pravy(u) := t
Vyvaz-INSERT(T_2, u)
endif

SPLIT(x):
Z_1, Z_2 := prázdné zásobníky, t :=kořen T
while key(t) $\neq x$ a t není list **do**
if key(t) $> x$ **then**
vlož (key(t), pravy(t)) do Z_2, t := levy(t)
else
vlož (levy(t), key(t)) do Z_1, t := pravy(t)
endif
enddo
if key(t) $= x$ **then**
T_1 := podstrom T určený levy(t)
T_2 := podstrom T určený pravy(t)
Výstup: $x \in S$
else
T_1, T_2 := jednoprvkové stromy
Výstup: $x \notin S$
endif
while $Z_1 \neq \emptyset$ **do**
(t, x) := vrchol Z_1, odstraň (t, x) ze Z_1
T' := podstrom T určený t, T_1 :=**JOIN3**(T', x, T_1)
enddo
while $Z_2 \neq \emptyset$ **do**
(x, t) := vrchol Z_2, odstraň (x, t) ze Z_1
T' := podstrom T určený t, T_2 :=**JOIN3**(T_2, x, T')
enddo

Složitost operací

Všimněme si, že každý algoritmus pracuje s vrcholy jen na jedné cestě z kořene do listu (ten

je určen operací a argumentem operace) a že na práci s libovolným vrcholem vyžaduje čas $O(1)$. Pak můžeme shrnout:

Věta. *Implementace operací* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN3** *a* **SPLIT** *pro červeno-černé stromy vyžadují v nejhorším případě čas* $O(\log(|S|))$, *kde* S *je reprezentovaná množina. Operace* **INSERT** *a* **JOIN3** *zavolají nejvýše jednou buď* **Rotace** *nebo* **Dvojita-rotace** *a operace* **DELETE** *zavolá nejvýše dvakrát* **Rotace** *nebo jednou* **Rotace** *a* **Dvojita-rotace**.

Operace **JOIN3** ve skutečnosti vyžaduje čas $O(|k_1 - k_2| + 1)$. Protože Z_1 a Z_2 obsahují nejvýše $\log(|S|)$ položek, odhadne se časová složitost operace **SPLIT** stejným způsobem jako v (a, b)-stromech. V ostatních případech odhad časové složitosti plyne z toho, že $\text{vyska}(T) = O(\log(|S|))$.

Pokud chceme mít i algoritmus pro operaci **ord**(k), musíme rozšířit strukturu o funkci p stejně jako v nevyvážených binárních vyhledávacích stromech. Pak lze použít přímo algoritmus pro **ord**(k) pro nevyvážené binární vyhledávací stromy. Připomeňme si, že procedury **Rotace** a **Dvojita-rotace** mohou aktualizovat funkci p v čase $O(1)$. Proto dostáváme

Věta. *Algoritmy pro realizaci operací* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN3**, **SPLIT** *a* **ord**(k) *pro rozšířenou strukturu červeno-černých stromů vyžadují v nejhorším případě čas* $O(\log(|S|))$, *kde* S *je reprezentovaná množina. Operace* **INSERT** *a* **JOIN3** *zavolají nejvýše jednou buď* **Rotace** *nebo* **Dvojita-rotace** *a operace* **DELETE** *zavolá nejvýše dvakrát* **Rotace** *nebo jednou* **Rotace** *a* **Dvojita-rotace**.

Vzniká otázka, proč se tolik pozornosti věnuje procedurám **Rotace** a **Dvojita-rotace**. Sice vyžadují čas $O(1)$, ale jsou to nejsložitější akce a spotřebují nejvíce času. V mnoha aplikacích (používají se hlavně ve výpočetní geometrii) tvar stromu spolu s parametry nesou ještě další zakódované informace. Při změně tvaru stromu je třeba je přepočítat. **Rotace** a **Dvojita-rotace** mění tvar stromu, kdežto posun směrem ke kořeni pouze mění obarvení. V tomto případě pak **Rotace** nebo **Dvojita-rotace** vyžaduje čas $O(|S|)$ (obvykle je třeba prohlédnout celý strom a přepočítat údaje) a nikoliv $O(1)$.

VII. Váhově vyvážené stromy

V osmdesátých letech se ve výpočetní geometrii hodně používaly $BB(\alpha)$-stromy. S příchodem červeno-černých stromů jejich popularita poklesla, ale občas se objeví. Proto se o nich zmíníme alespoň orientačně. Mějme reálné číslo α takové, že $\frac{1}{4} < \alpha \leq \frac{\sqrt{2}}{2}$. Pro strom T označme $p(T)$ počet listů ve stromu T. Binární vyhledávací strom T reprezentující množinu S se nazývá $BB(\alpha)$-strom, když pro každý vnitřní vrchol v platí:

$$\alpha \leq \frac{p(T_{l,v})}{p(T_v)} = 1 - \frac{p(T_{r,v})}{p(T_v)} \leq 1 - \alpha,$$

kde T_v je podstrom T určený vrcholem v, $T_{l,v}$ je podstrom T určený levým synem vrcholu v a $T_{r,v}$ je podstrom T určený pravým synem vrcholu v. Platí

Tvrzení. *Když T je $BB(\alpha)$-strom reprezentující n-prvkovou množinu, pak*

$$\text{vyska}(T) \leq 1 + \frac{\log(n+1) - 1}{\log \frac{1}{1-\alpha}}.$$

Důsledkem je, že $BB(\alpha)$-stromy patří do skupiny vyvážených binárních vyhledávacích stromů. Vyvažování se provádí opět pomocí procedur **Rotace** a **Dvojita-rotace** a popisuje ho následující technické tvrzení.

Tvrzení. *Pro každé α existuje konstanta d taková, že $\alpha < d < 1-\alpha$ a že pro každý binární vyhledávací strom T s kořenem t splňující podmínky*

(1) *podstromy T_l a T_r stromu T určené levým a pravým synem t jsou $BB(\alpha)$-stromy;*
(2) $\frac{p(T_l)}{p(T)} < \alpha$ *a buď* $\alpha \leq \frac{p(T_l)}{p(T)-1} \leq 1-\alpha$ *nebo* $\alpha \leq \frac{p(T_l)+1}{p(T)+1} \leq 1-\alpha$,

platí:
Nechť $\rho = \frac{p(T')}{p(T_r)}$, *kde T' je určen levým synem pravého syna kořene t. Když $\rho \leq d$, pak provedeme* **Rotace**$(t, \text{pravy}(t))$
, a když $\rho > d$, provedeme **Dvojita-rotace**$(t, \text{pravy}(t), \text{levy}(\text{pravy}(t)))$.
V obou případech dostaneme $BB(\alpha)$-strom.

Toto tvrzení a jeho symetrické verze jednoznačně ukazují (až na specifikaci konstanty d, která však přesahuje rámec této přednášky), jak vyvažovat $BB(\alpha)$-stromy při aktualizačních operacích, protože podstrom $BB(\alpha)$-stromu určený nějakým jeho vrcholem je opět $BB(\alpha)$-strom. Pak dostáváme:

Věta. *Implementace operací* **MEMBER**, **INSERT** *a* **DELETE** *v $BB(\alpha)$-stromech vyžadují v nejhorším případě čas $O(\log(|S|))$, kde S je reprezentovaná množina.*

Obliba $BB(\alpha)$-stromů byla zapříčiněna vlastností uvedenou v následující větě, která je analogií věty o vyvažovacích operacích pro (a,b)-stromy.

Věta. *Nechť α je reálné číslo takové, že $\frac{1}{4} < \alpha < 1 - \frac{\sqrt{2}}{2}$. Pak existuje konstanta $c > 0$ závislá jen na α taková, že každá posloupnost operací* **INSERT** *a* **DELETE** *o délce m aplikovaná na původně prázdný $BB(\alpha)$-strom volá nejvýše cm procedur* **Rotace** *a* **Dvojita-rotace**.

VIII. Historický přehled

(a,b)-stromy zavedli Bayer a McGreght (1972).

Věty o počtu vyvažovacích operací pro (a,b)-stromy dokázali Huddleston a Mehlhorn (1982). Ti také ukázali možnost aplikace tohoto odhadu na paralelní implementace algoritmů v (a,b)-stromech.
A-sort navrhli a analyzovali Guibas, McGreight, Plass a Roberts (1977).

Analýza interpolačního vyhledávání pochází od Perla, Itaie a Avniho (1978).
Kvadratické vyhledávání analyzovali Perl a Reingold (1977).

Adelson-Velskij a Landis (1962) definovali AVL-stromy.
Červeno-černé stromy definovali Guibas a Sedgewick (1978), ale poslední verze algoritmu **DELETE** pochází od Tarjana (1983).
$BB(\alpha)$-stromy zavedli Nievergelt a Reingold (1973).
Věty o jejich vyvažování dokázali Blum a Mehlhorn (1980).

Priorita AVL-stromů se odráží v jejich hojném používání, i když červeno-černé stromy jsou efektivnější.

Datové struktury - haldy a třídicí algoritmy

I. Úvod

V praxi se často setkáváme s následujícím problémem, který vzniká na uspořádaném univerzu, jehož uspořádání se však v průběhu času mění. Úloha se liší od slovníkového problému v tom, že se nevyžaduje efektivní operace **MEMBER**. Dokonce se předpokládá, že operace dostane spolu s argumentem informaci o uložení zpracovávaného prvku. Hlavním požadavkem je rychlost provedední ostatních operací a malé paměťové nároky. Přitom v praxi obvykle nestačí znát jen asymptotickou složitost, důležitou roli hraje skutečná rychlost, kterou však neumíme obecně spočítat, protože je závislá na použitém systému a hardwaru. Přesto je při použití následujících struktur dobré mít realistickou představu o skutečných rychlostech operací a podle toho si vybrat vhodnou strukturu.

Zadání problému: Nechť U je univerzum. Je dána množina $S \subseteq U$ a funkce $f : S \to \mathbb{R}$, kde $\mathbb{R}$ jsou reálná čísla (tato funkce realizuje uspořádání na univerzu U – pro $u, v \in U$ platí $u \leq v$, právě když $f(u) \leq f(v)$; změna uspořádání se pak realizuje změnou funkce f). Máme navrhnout reprezentaci S a f, která umožňuje operace:
INSERT(s, a) – přidá k množině S prvek s tak, že $f(s) = a$,
MIN – nalezne prvek $s \in S$ s nejmenší hodnotou $f(s)$,
DELETEMIN – odstraní prvek $s \in S$ s nejmenší hodnotou $f(s)$,
DELETE(s) – odstraní prvek $s \in S$ z množiny S,
DECREASE(s, a) – zmenší hodnotu $f(s)$ o a (tj. $f(s) := f(s) - a$),
INCREASE(s, a) – zvětší hodnotu $f(s)$ o a (tj. $f(s) := f(s) + a$).
Při operaci **INSERT**(s, a) se předpokládá, že $s \notin S$, a tento předpoklad operace **INSERT** neověřuje. Při operacích **DELETE**(s), **DECREASE**(s, a) **a INCREASE**(s, a) se předpokládá, že $s \in S$, a operace navíc dostává informaci, jak najít prvek s v reprezentaci S a f. Haldy jsou typ struktury, která se používá pro řešení tohoto problému.

Halda je stromová struktura, kde vrcholy reprezentují prvky z S a splňují lokální podmínku na f. Obvykle se používá následující podmínka nebo její duální verze:

(usp) Pro každý vrchol v platí: když v reprezentuje prvek $s \in S$ a otec(v) reprezentuje $t \in S$, pak $f(t) \leq f(s)$.

Probereme několik verzí hald a budeme předpokládat, že vždy splňují tuto podmínku a že požadavek na provedení operací **DELETE**(s), **DECREASE**(s, a) a **INCREASE**(s, a) také zadává ukazatel na vrchol reprezentující $s \in S$. Navíc budeme uvažovat operace

MAKEHEAP(S, f) – operace vytvoří haldu reprezentující množinu S a funkci f,
MERGE(H_1, H_2) – předpokládá, že halda H_i reprezentuje množinu S_i a funkci f_i pro $i = 1, 2$ a $S_1 \cap S_2 = \emptyset$. Operace vytvoří haldu H reprezentující $S_1 \cup S_2$ a $f_1 \cup f_2$, přičemž neověřuje disjunktnost S_1 a S_2.

II. Regulární haldy

První použité haldy byly binární neboli 2-regulární haldy. Tyto haldy jsou velmi oblíbené pro svou jednoduchost a názornost a pro velmi efektivní implementaci.

Předpokládejme, že $d > 1$ je přirozené číslo. d-regulární strom je kořenový strom (T, r), pro který existuje pořadí synů jednotlivých vnitřních vrcholů takové, že očíslování vrcholů prohledáváním do šířky (kořen r je číslován 1) splňuje následující vlastnosti

(1) každý vrchol má nejvýše d synů,
(2) když vrchol není list, tak všechny vrcholy s menším číslem mají právě d synů,
(3) když vrchol má méně než d synů, pak všechny vrcholy s většími čísly jsou listy.

Toto očíslování se nazývá přirozené očíslování d-regulárního stromu.

Tvrzení. *Každý d-regulární strom má nejvýše jeden vrchol, který není list a má méně než d synů. Když d-regulární strom má n vrcholů, pak jeho výška je $\lceil \log_d(n(d-1)+1) \rceil$. Nechť o je přirozené očíslování vrcholů d-regulárního stromu. Když pro vrchol v je $o(v) = k$, pak vrchol w je syn vrcholu v, právě když $o(w) \in \{(k-1)d+2, (k-1)d+3, \ldots, kd+1\}$, a vrchol u je otcem vrcholu v, právě když $o(u) = 1 + \lfloor \frac{k-2}{d} \rfloor$.*

Důkaz. První část tvrzení plyne přímo z požadavku 2) na d-regulární strom. Má-li d-regulární strom výšku k, pak má alespoň $\Sigma_{i=0}^{k-1} d^i + 1$ a nejvýše $\Sigma_{i=0}^{k} d^i$ vrcholů. Proto

$$\frac{d^k - 1}{d-1} < n \leq \frac{d^{k+1}}{d-1} \quad , \quad d^k - 1 < n(d-1) \leq d^{k+1} - 1$$

a zlogaritmováním dostaneme

$$k < \log_d(n(d-1)+1) \leq k+1.$$

Odtud plyne druhá část tvrzení. Třetí část pro čísla synů dokážeme indukcí podle očíslování. Synové kořene mají čísla $2, 3, \ldots, d+1$, protože kořen má číslo 1. Když tvrzení platí pro vrchol s číslem k, pak synové vrcholu s číslem $k+1$ mají čísla $kd+2, kd+3, \ldots, kd+d+1$, což odpovídá číslům $(k+1-1)d+2, (k+1-1)d+3, \ldots, (k+1)d+1$, a tedy tvrzení platí. Poslední část pak plyne z toho, že když $i \in \{(k-1)d+2, (k-1)d+3, \ldots, kd+1\}$, pak $1 + \lfloor \frac{i-2}{d} \rfloor = k$. □

Všimněme si, že speciálně pro $d = 2$ mají synové vrcholu s číslem k čísla $2k$ a $2k+1$ a otec vrcholu s číslem k má číslo $\lfloor \frac{k}{2} \rfloor$. Tedy pro 2-regulární stromy je předpis pro nalezení synů a otce zvláště jednoduchý.

Řekneme, že množina S s funkcí f je reprezentována d-regulární haldou H, kde H je d-regulární strom (T, r), když přiřazení prvků množiny S vrcholům stromu T je bijekce splňující podmínku (usp). Toto přiřazení je realizováno funkcí key, která vrcholu přiřazuje jím reprezentovaný prvek.

Defince d-regulárního stromu umožňuje velmi efektivní implementace d-regulárních hald. Mějme množinu S reprezentovanou d-regulární haldou H s přirozeným očíslováním o d-regulárního stromu (T, r). Pak haldu H můžeme reprezentovat polem $H[1..|S|]$, kde pro

vrchol stromu v, pro který $o(v) = i$, je $H(i) = (\text{key}(v), f(\text{key}(v))$. Algoritmy budeme popisovat pro stromy, protože je to názornější. Přeformulovat je pro pole je snadné (viz očíslování synů a otce vrcholu v). Pro jednoduchost budeme pro vrchol v psát $f(v)$ místo $f(\text{key}(v))$, neboli $f(v)$ bude označovat $f(s)$, kde s je reprezentován vrcholem v. U d-regulárního stromu předpokládáme, že známe přirozené očíslování, a fráze 'poslední vrchol', 'předcházející vrchol' atd. se vztahují k tomuto očíslování.

Algoritmy

Pro d-regulární haldy není známa efektivní implementace operace **MERGE**. Efektivní implementace ostatních operací jsou založeny na pomocných operacích **UP**(v) a **DOWN**(v). Operace **UP**(v) posunuje prvek s reprezentovaný vrcholem v směrem ke kořeni, dokud vrchol reprezentující prvek s nesplňuje podmínku (usp). Operace **DOWN**(v) je symetrická.

UP(v):
while v není kořen a $f(v) < f(\text{otec}(v))$ **do**
  vyměň key(v) a key$(\text{otec}(v))$
  $v :=$ otec(v)
enddo

DOWN(v):
if v není list **then**
  $w :=$syn vrcholu v reprezentující prvek s nejmenší hodnotou $f(w)$
  while $f(w) < f(v)$ a v není list **do**
    vyměň key(v) a key(w), $v := w$
    $w :=$syn vrcholu v reprezentující prvek s nejmenší hodnotou $f(w)$
  enddo
endif

INSERT(s):
$v :=$nový poslední list, key$(v) := s$, **UP**(v)

MIN:
Výstup key(kořen(T))

DELETEMIN:
$v :=$poslední list, $r :=$kořen, key$(r) :=$ key(v)
odstraň v
DOWN(r)

DELETE(s):
$v :=$vrchol reprezentující s
$w :=$poslední list
$t :=$ key(w), key$(v) := t$, odstraň w
if $f(t) < f(s)$ **then UP**(v) **else DOWN**(v) **endif**

DECREASE(s, a):
v :=vrchol reprezentující s
$f(s) := f(s) - a$, **UP**(v)

INCREASE(s, a):
v :=vrchol reprezentující s
$f(s) := f(s) + a$, **DOWN**(v)

MAKEHEAP(S, f):
$T := d$-regulární strom s $|S|$ vrcholy
zvol libovolnou reprezentaci S vrcholy stromu T
v :=poslední vrchol, který není list
while v je vrchol T **do**
DOWN(v)
v :=vrchol předcházející vrcholu v
enddo

Ověříme korektnost algoritmů. Je zřejmé, že pomocné operace jsou korektní – skončí, když podmínku (usp) splňuje prvek s, který byl původně reprezentován vrcholem v. Korektnost operace **MIN** plyne přímo z podmínky (usp), protože kořen reprezentuje nejmenší prvek množiny S. U operace **INSERT** je podmínka (usp) splněna pro všechny vrcholy s výjimkou nově vytvořeného listu a operace **UP** zajistí její splnění. Při operaci **DELETEMIN** je podmínka (usp) splněna pro všechny vrcholy s výjimkou kořene a v tomto případě operace **DOWN** zajistí její splnění. Po provedení operací **DELETE**(s), **DECREASE**(s, a) a **INCREASE**(s, a) je podmínka (usp) splněna pro všechny vrcholy s výjimkou vrcholu v a její splnění opět zajistí operace **UP** resp. **DOWN**. Pro operaci **MAKEHEAP** budeme uvažovat duální formulaci podmínky (usp):

(d-usp) když s je prvek reprezentovaný vrcholem v, pak $f(s) \leq f(t)$ pro všechny prvky reprezentované syny vrcholu v.

Pokud každý vrchol splňuje podmínku (d-usp), pak splňuje i podmínku (usp). Zřejmě každý list splňuje podmínku (d-usp) a když operace **MAKEHEAP** provede proceduru **DOWN**(v), pak je podmínka (d-usp) splněna pro všechny vrcholy s čísly alespoň tak velkými jako je číslo v. Operace **MAKEHEAP** končí provedením operace **DOWN** na kořen a odtud plyne její korektnost.

Složitost operací

Vypočteme časovou složitost operací: Jeden běh cyklu v operaci **UP** vyžaduje čas $O(1)$ a v operaci **DOWN** čas $O(d)$. Proto operace **UP** v nejhorším případě vyžaduje čas $O(\log_d |S|)$ a operace **DOWN** čas $O(d \log_d |S|)$. Operace **MIN** vyžaduje čas $O(1)$, **INSERT** a **DECREASE** vyžadují čas $O(\log_d |S|)$ a **DELETEMIN**, **DELETE** a **INCREASE** čas $O(d \log_d |S|)$.

Haldu můžeme vytvořit iterací operace **INSERT**, což vyžaduje čas $O(|S| \log_d(|S|))$. Ukážeme, že složitost operace **MAKEHEAP** je menší, ale pro malé haldy je výhodnější provádět opakovaně operaci **INSERT**. Operace **DOWN**(v) na vrchol ve výšce h vyžaduje v nejhorším případě čas $O(hd)$. Vrcholů v hloubce i je nejvýše d^i. Předpokládejme, že strom

má výšku k, pak vrchol v hloubce i má výšku nejvýše $k-i$. Tedy operace **MAKEHEAP** vyžaduje čas $O(\sum_{i=0}^{k-1} d^i(k-i)d) = O(\sum_{i=0}^{k-1} d^{i+1}(k-i))$. Označme $A = \sum_{i=0}^{k-1} d^{i+1}(k-i)$, pak

$$dA - A = \sum_{i=0}^{k-1} d^{i+2}(k-i) - \sum_{i=0}^{k-1} d^{i+1}(k-i) = \sum_{i=2}^{k+1} d^i(k-i+2) - \sum_{i=1}^{k} d^i(k-i+1) =$$
$$d^{k+1} + \sum_{i=2}^{k} d^i(k-i+2-k+i-1) - dk = d^{k+1} + \sum_{i=2}^{k} d^i - dk =$$
$$d^{k+1} + d^2\frac{d^{k-1}-1}{d-1} - dk.$$

Tedy $A = \frac{d^{k+1}}{d-1} + \frac{d^{k+1}-d^2}{(d-1)^2} - \frac{dk}{d-1}$. Protože $k = \lceil \log_d(|S|(d-1)+1)\rceil$, dostáváme, že $d^{k+1} \leq d^2((d-1)|S|+1)$, a proto $A \leq 2d^2|S|$. Tedy **MAKEHEAP** vyžaduje v nejhorším případě jen čas $O(d^2|S|)$.

Aplikace

Třídění: prostou posloupnost čísel $x_1, x_2, \ldots, x_n$ lze setřídit následujícím algoritmem používajícím haldu (f bude v tomto případě identická funkce).

d-**HEAPSORT**$(x_1, x_2, \ldots, x_n)$:
MAKEHEAP$(\{x_i \mid i = 1, 2, \ldots, n\}, f)$
$i = 1$
while $i \leq n$ **do**
  $y_i :=$**MIN**, **DELETEMIN**, $i := i + 1$
enddo
Výstup: $y_1, y_2, \ldots, y_n$

Teoreticky lze ukázat, že použití d-regulárních hald v algoritmu **HEAPSORT** pro $d = 3$ a $d = 4$ je výhodnější než $d = 2$. Experimenty ukázaly, že optimální algoritmus pro posloupnosti délek do 1 000 000 by měl používat $d = 6$ nebo $d = 7$ (v experimentech byl měřen skutečně spotřebovaný čas, nikoli počet porovnání a výměn prvků). Pro delší posloupnosti se optimální hodnota d může zmenšit.

Dalším příkladem je nalezení nejkratších cest v grafu z daného bodu. Řešme následující úlohu:
Vstup: orientovaný ohodnocený graf (X, R, c), kde c je funkce z R do množiny kladných reálných čísel, a vrchol $z \in X$.
Úkol: nalézt pro každý bod $x \in X$ délku nejkratší cestu ze z do x, kde délka cesty je součet c-ohodnocení hran na cestě.

Dijkstrův algoritmus:
$d(z) := 0$, $U := \{z\}$
for every $x \in X \setminus \{z\}$ **do** $d(x) := +\infty$ **enddo**

while $U \neq \emptyset$ **do**
  najdi vrchol $u \in U$ s nejmenší hodnotou $d(u)$
  odstraň u z U
  for every $(u, v) \in R$ **do**
    if $d(u) + c(u, v) < d(v)$ **then**
      if $d(v) = +\infty$ **then** vlož v do U **endif**
      $d(v) := d(u) + c(u, v)$
    endif
  enddo
enddo

Korektnost algoritmu je založena na kombinatorickém lemmatu, které říká, že když odstraňujeme z U prvek x s nejmenší hodnotou $d(x)$, pak vzdálenost ze z do x je právě $d(x)$. Proto když $U = \emptyset$, pak $d(x)$ jsou délky nejkratších cest ze z do x pro všechna $x \in X$. Tedy práce s množinou U vyžaduje nejvýše $|X|$ operací **INSERT**, **MIN** a **DELETEMIN** a $|R|$ operací **DECREASE** a vždy platí $|U| \leq |X|$. Vypočteme časovou složitost **Dijkstrova algoritmu** za předpokladu, že U reprezentujeme jako d-regulární haldu. Když $d = 2$, pak dostáváme, že algoritmus vyžaduje čas $O(|X| \log(|X|) + |R| \log(|X|))$. Když $d = \max\{2, \lfloor \frac{|R|}{|X|} \rfloor\}$, pak algoritmus vyžaduje čas $O(|R| \log_d |X|)$. V případě, že (X, R) je hustý graf, tj. $|R| > |X|^{1+\varepsilon}$ pro $\varepsilon > 0$, pak $\log_d |X| = O(1)$ a algoritmus je lineární (tj. vyžaduje čas $O(|R|)$).

III. Leftist haldy

Dalším typem hald, se kterými se seznámíme, jsou lefist haldy (neznáme vhodný český překlad, proto zůstáváme u anglického názvu). Je to velmi elegantní a jednoduchý typ hald. Všechny operace jsou stejně jako u regulárních hald založeny na dvou základních operacích, z nichž v tomto případě hlavní je **MERGE** a druhou je **DECREASE**. Použití **MERGE** při návrhu jiných operací je běžné i v dalších haldách. Operace **MERGE** využívá speciálních vlastností leftist hald a idea operace **DECREASE** je stejná jako ve Fibonacciho haldách. Nejprve formálně popíšeme strukturu leftist hald.

Mějme binární kořenový strom (T, r) (to znamená, že r je kořen, každý vrchol má nejvýše dva syny a u každého syna víme, zda je to pravý nebo levý syn). Pro vrchol v označme $\mathrm{npl}(v)$ délku nejkratší cesty z v do vrcholu, který má nejvýše jednoho syna, takže např. pro list l platí $\mathrm{npl}(l) = 0$.

Mějme $S \subseteq U$ a funkci $f : S \to \mathbb{R}$. Pak binární strom (T, r) takový, že

(1) když vrchol v má jen jednoho syna, pak je to levý syn,
(2) když vrchol v má dva syny, pak

$$\mathrm{npl}(\text{pravý syn } v) \leq \mathrm{npl}(\text{levý syn } v),$$

(3) existuje jednoznačné přiřazení prvků S vrcholům T, které splňuje podmínku (usp) (toto přiřazení je reprezentováno funkcí key, která vrcholu v přiřadí prvek z množiny S reprezentovaný vrcholem v)

je leftist halda reprezentující množinu S a funkci f.

Struktura vrcholu v v leftist haldě:
S vrcholem v jsou spojeny ukazatelé otec(v), levy(v) a pravy(v) na otce a na levého a pravého syna vrcholu v. Když ukazatel není definován, pak píšeme, že jeho hodnota je NIL. Dále jsou s vrcholem spojeny funkce
npl(v) – proměnná s hodnotou npl(v),
key(v) – prvek reprezentovaný vrcholem v,
$f(v)$ – proměnná obsahující hodnotu f(key(v)).

Uvedeme základní vlastnost leftist haldy, která umožňuje efektivní implementace operací. Posloupnost vrcholů $v_0, v_1, \ldots, v_k$ se nazývá pravá cesta z vrcholu v, když $v = v_0$, v_{i+1} je pravý syn v_i pro každé $i = 0, 1, \ldots, k-1$ a v_k nemá pravého syna. Pak podstrom vrcholu v do hloubky k je úplný binární strom a má tedy alespoň $2^{k+1} - 1$ vrcholů. Proto platí

Tvrzení. *V leftist haldě je délka pravé cesty z každého vrcholu v nejvýše rovna*

$$\log(\text{velikost podstromu určeného vrcholem } v).$$

Algoritmy a složitost operací

Základní operací pro leftist haldy je **MERGE**. Tato operace je definována rekurzivně a hloubka rekurze je omezena právě délkami pravých cest.

MERGE(T_1, T_2):
if $T_1 = \emptyset$ **then Výstup**= T_2 konec **endif**
if $T_2 = \emptyset$ **then Výstup**= T_1 konec **endif**
if key(kořen T_1) > key(kořen T_2) **then**
  zaměň T_1 a T_2
endif
T' :=**MERGE**(podstrom pravého syna kořene T_1, T_2)
pravy(kořen T_1) := kořen T'
otec(kořen T') := kořen T_1
if npl(pravy(kořen T_1)) > npl(levy(kořen T_1)) **then**
  vyměň levého a pravého syna kořene T_1
endif
npl(kořen T_1) := npl(pravy(kořen T_1) + 1

INSERT(x):
Vytvoř haldu T_1 reprezentující $\{x\}$
MERGE(T, T_1)

MIN:
Výstup: key(kořen T)

DELETEMIN:
T_1 :=podstrom levého syna kořene T
T_2 :=podstrom pravého syna kořene T
MERGE(T_1, T_2)

MAKEHEAP(S, f):
Q :=prázdná fronta
for every $s \in S$ **do**
  vlož leftist haldu T_s reprezentující $\{s\}$ do Q
enddo
while $|Q| > 1$ **do**
  vezmi haldy T_1 a T_2 z vrcholu Q (odstraň je)
  MERGE(T_1, T_2) vlož do Q
enddo

Vypočteme časovou složitost předchozích algoritmů. Každý běh algoritmu **MERGE** (bez rekurzivního volání) vyžaduje čas $O(1)$. Počet rekurzivních volání je součet délek pravých cest, proto algoritmus **MERGE** vyžaduje čas $O(\log(|S_1|+|S_2|))$, kde S_i je množina reprezentovaná haldou T_i pro $i = 1, 2$. Odtud dále plyne, že čas algoritmů **INSERT** a **DELETE-MIN** je v nejhorším případě $O(\log(|S|))$. Operace **MIN** vyžaduje čas $O(1)$. Pro odhad složitosti **MAKEHEAP** budeme uvažovat, že na začátku algoritmu je na vrcholu fronty speciální znak, který se jen přenese na konec fronty. Odhadneme čas, který spotřebují **while-**cykly mezi dvěma přeneseními speciálního znaku. Předpokládejme, že se speciální znak přenesl po k-té. V tomto okamžiku mají všechny haldy ve frontě až na jednu velikost 2^{k-1}. Proto ve frontě Q je $\lceil \frac{|S|}{2^{k-1}} \rceil$ hald a jelikož jedna operace **MERGE** vyžaduje $O(k)$ času, tak **while**-cykly vyžadují čas $O(k\frac{|S|}{2^{k-1}})$. Můžeme tedy shrnout, že operace **MAKEHEAP** potřebuje čas

$$O(\sum_{k=1}^{\infty} k\frac{|S|}{2^{k-1}}) = O(|S|\sum_{k=1}^{\infty} \frac{k}{2^{k-1}}) = O(|S|).$$

Řada $\sum_{k=1}^{\infty} \frac{k}{2^{k-1}}$ konverguje např. podle podílového d'Alambertova kritéria a lze jednoduše spočítat (např. stejnou metodou jako pro regulární haldy), že součet je 4.

Implementace operací **DECREASE** a **INCREASE** pomocí operací **UP** a **DOWN** jako v d-regulárních haldách není efektivní, protože délka cesty z kořene do listu v leftist haldě může být až $|S|$. Proto navrhneme pro tyto operace efektivnější algoritmus založený na jiném principu. Tento princip je pak použit i pro Fibonacciho haldy.

Nejprve popíšeme pomocnou operaci **Oprav**(T, v), která vytvoří leftist haldu z binárního stromu T' vzniklého z leftist haldy T odtržením podstromu s kořenem ve vrcholu v.

Oprav(T, v):
$t :=$ otec(v), npl$(t) := 0$
if pravy$(t) \neq v$ **then** levy$(t) :=$ pravy(t) **endif**
pravy$(t) := NIL$
while se zmenšilo npl(t) a t není kořen **do**
  $t :=$ otec(t)
  if npl(pravy(t)) > npl(levy(t)) **then**
    vyměň levy(t) a pravy(t)
  endif
  npl$(t) :=$ npl(pravy(t)) + 1
enddo

Po provedení operace **Oprav** mají všechny vrcholy správné číslo npl a podmínky kladené na leftist haldu jsou splněny. Tedy po provedení **Oprav** je T opět leftist halda. Když t je poslední vrchol, u kterého se zmenšilo npl, pak všechny vrcholy, kde se zmenšilo npl, tvoří pravou cestu z vrcholu t. To znamená, že **while**-cyklus se prováděl nejvýše $\log(|S|)$-krát a každý běh **while**-cyklu vyžadoval čas $O(1)$. Proto algoritmus **Oprav** vyžaduje čas $O(\log(|S|))$.

Popíšeme ostatní algoritmy.

DECREASE(s, a):
v :=prvek reprezentující s
T_1:=podstrom T určený vrcholem v, $f(v) := f(v) - a$
T_2 :=**Oprav**(T, v), T :=**MERGE**(T_1, T_2)

INCREASE(s, a):
v :=prvek reprezentující s
T_1 :=podstrom T určený vrcholem levy(v)
T_2 :=podstrom T určený vrcholem pravy(v)
T_3 :=leftist halda reprezentující prvek s
$f(v) := f(v) + a$, T_4 :=**Oprav**(T, v), T_1 :=**MERGE**(T_1, T_3)
T_2 :=**MERGE**(T_2, T_4), T :=**MERGE**(T_1, T_2)

DELETE(s, a):
v :=prvek reprezentující s
T_1 :=podstrom T určený vrcholem levy(v)
T_2 :=podstrom T určený vrcholem pravy(v)
T_3 :=**MERGE**(T_1, T_2), T_4 :=**Oprav**(T, v)
T :=**MERGE**(T_3, T_4)

Protože algoritmy **MERGE** a **Oprav** vyžadují čas $O(\log(|S|)$ a protože zbylé části algoritmů pro operace **DECREASE**, **INCREASE** a **DELETE** vyžadují $O(1)$ času, můžeme shrnout výsledky:

Věta. *V leftist haldách existuje implementace operace* **MIN***, která v nejhorším případě vyžaduje čas* $O(1)$*, implementace operací* **INSERT**, **DELETEMIN**, **DELETE**, **MERGE**, **DECREASE** *a* **INCREASE***, které vyžadují v nejhorším případě čas* $O(\log(|S|))$*, a implementace operace* **MAKEHEAP***, která vyžaduje čas* $O(|S|)$*, kde* S *je reprezentovaná množina.*

IV. Amortizovaná složitost

Popíšeme bankovní paradigma pro počítání s amortizovanou složitostí. Předpokládejme, že máme funkci h, která ohodnucuje konfigurace a kvantitativně vystihuje jejich vhodnost pro provedení operace o. Když na konfiguraci D aplikujeme operaci o a dostaneme konfiguraci D', pak amortizovaná složitost $am(o)$ operace o má vystihovat nejen časovou náročnost operace, ale i to, jak se změnila vhodnost konfigurace pro tuto operaci. Proto ji definujme jako $am(o) = t(o) + h(D') - h(D)$, kde $t(o)$ je čas potřebný pro provedení operace o.

Předpokládejme, že chceme provést posloupnost operací $o_1, o_2, \ldots, o_n$ na konfiguraci D_0. Znázorníme si to takto:

$$D_0 \xrightarrow{o_1} D_1 \xrightarrow{o_2} D_2 \xrightarrow{o_3} \ldots \xrightarrow{o_n} D_n.$$

Předpokládejme, že pro každé $i = 1, 2, \ldots, n$ máme odhad $c(o_i)$ amortizované složitosti operace o_i, tj. $am(o_i) \leq c(o_i)$ pro všechna $i = 1, 2, \ldots, n$. Pak

$$\sum_{i=1}^{n} am(o_i) = \sum_{i=1}^{n} \big(t(o_i) + h(D_i) - h(D_{i-1})\big) = h(D_n) - h(D_0) + \sum_{i=1}^{n} t(o_i) \leq \sum_{i=1}^{n} c(o_i).$$

Z toho plyne, že

$$\sum_{i=1}^{n} t(o_i) \leq \sum_{i=1}^{n} c(o_i) - h(D_n) + h(D_0).$$

Obvykle je $h(D) \geq 0$ pro všechny konfigurace D nebo naopak $h(D) \leq 0$ pro všechny konfigurace D. Když $h(D) \geq 0$, pak

$$\sum_{i=1}^{n} t(o_i) \leq \sum_{i=1}^{n} c(o_i) + h(D_0),$$

když $h(D) \leq 0$, pak

$$\sum_{i=1}^{n} t(o_i) \leq \sum_{i=1}^{n} c(o_i) - h(D_n).$$

To znamená, že odhad amortizované složitosti dává také odhad na časovou složitost posloupnosti operací, který bývá lepší než odhad složitosti v nejhorším případě. Tato skutečnost vysvětluje řadu případů, kdy výsledky byly lepší než teoretický výpočet. Ukazuje se, že složitost posloupnosti operací v nejhorším případě je často podstatně menší než součet složitostí v nejhorším případě pro jednotlivé operace.

V. Binomiální haldy

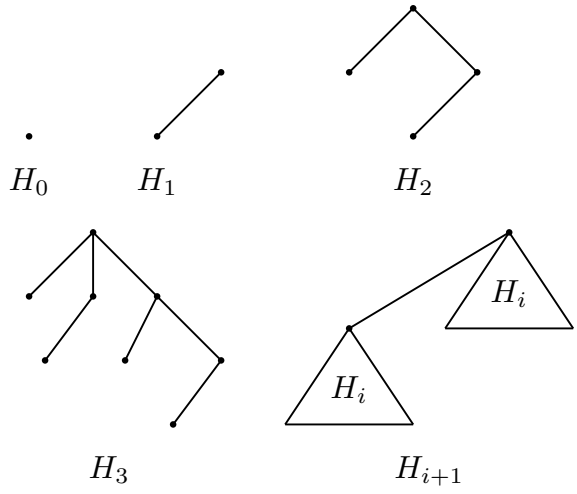
Další typ hald je motivován sčítaním přirozených čísel. Binomiální halda reprezentující $n-$prvkovou množinu se totiž chová podobně jako číslo n. Tento typ hald je také po zobecnění v jistém smyslu vzorem pro Fibonacciho haldy.

Pro $i = 0, 1, \ldots$ definujeme rekurentně binomiální stromy H_i. Jsou to kořenové stromy takové, že H_0 je jednoprvkový strom a strom H_{i+1} vznikne ze dvou disjunktních stromů H_i, kde kořen jednoho stromu se stane dalším synem (nejlevějším nebo nejpravějším) kořene druhého stromu. Viz Obr. 1.

Nejprve uvedeme základní vlastnosti těchto stromů.

Tvrzení. *Pro každé přirozené číslo $i = 0, 1, \ldots$ platí:*

(1) *strom H_i má 2^i vrcholů,*
(2) *kořen stromu H_i má i synů,*
(3) *délka nejdelší cesty z kořene do listu ve stromu H_i je i (tj. výška H_i je i),*
(4) *podstromy určené syny kořene stromu H_i jsou izomorfní se stromy $H_0, H_1, \ldots, H_{i-1}$.*



Obr. 1

Důkaz. Tvrzení platí pro strom H_0 a jednoduchou indukcí se dokáže i pro další stromy. Skutečně, když H_i má 2^i vrcholů, pak H_{i+1} má $2(2^i) = 2^{i+1}$ vrcholů. Kořen stromu H_{i+1} má o jednoho syna více než kořen stromu H_i a nejdelší cesta do listu je o 1 delší. Protože podstrom syna, který přibyl kořeni stromu H_{i+1}, je izomorfní s H_i a jinak se nic neměnilo, je důkaz kompletní. □

Binomiální halda $\mathcal{H}$ reprezentující množinu S je soubor (seznam) stromů $\{T_1, T_2, \ldots, T_k\}$ takový, že

> celkový počet vrcholů v těchto stromech je roven velikosti S a existuje a je dáno jednoznačné přiřazení prvků z S vrcholům stromů takové, že platí podmínka (usp) – toto přiřazení je realizováno funkcí key, která vrcholu stromu přiřazuje prvek jím reprezentovaný;
> každý strom T_i je izomorfní s nějakým stromem H_j;
> T_i není izomorfní s žádným T_j pro $i \neq j$.

Z binárního zápisu přirozených čísel plyne, že pro každé přirozené číslo $n > 0$ existuje prostá posloupnost $i_1, i_2, \ldots, i_k$ přirozených čísel taková, že $n = \sum_{j=1}^{k} 2^{i_j}$. Z toho plyne, že pro každou neprázdnou množinu S existuje binomiální halda reprezentující S. Tato halda obsahuje strom izomorfní s H_i, právě když v binárním zápise čísla $|S|$ je na i-tém místě zprava 1.

Algoritmy a složitost operací

Operace pro binomiální haldy jsou stejně jako pro leftist haldy založeny na operaci **MERGE**. Operace **MERGE** pro binomiální haldy je analogií sčítání přirozených čísel v binárním zápise.

MERGE$(\mathcal{H}_1, \mathcal{H}_2)$:
(komentář: $\mathcal{H}_i$ reprezentuje množinu S_i pro $i = 1, 2$ a $S_1 \cap S_2 = \emptyset$)

$i := 0$, $T :=$prázdný strom, $\mathcal{H} := \emptyset$
while $i < \log(|S_1| + |S_2|)$ **do**
  if existuje $U \in \mathcal{H}_1$ izomorfní s H_i **then**
    $U_1 := U$
  else
    $U_1 :=$prázdný strom
  endif
  if existuje $U \in \mathcal{H}_2$ izomorfní s H_i **then**
    $U_2 := U$
  else
    $U_2 :=$prázdný strom
  endif
  case
    (existuje právě jeden neprázdný strom $V \in \{T, U_1, U_2\}$) **do**:
      vlož V do $\mathcal{H}$, $T :=$prázdný strom
    (existují právě dva neprázdné stromy $V_1, V_2 \in \{T, U_1, U_2\}$) **do**:
      $T :=$**spoj**(V_1, V_2)
    (všechny stromy T, U_1 a U_2 jsou neprázdné) **do**:
      vlož T do $\mathcal{H}$, $T :=$**spoj**(U_1, U_2)
  endcase
  $i := i + 1$
enddo
if $T \neq$prázdný strom **then** vlož T do $\mathcal{H}$ **endif**
Výstup:$\mathcal{H}$

spoj(T_1, T_2):
if f(kořen T_1) $>$ f(kořen T_2) **then**
  vyměň stromy T_1 a T_2
endif
vytvoř nového syna v kořene T_1
$v :=$kořen T_2

Je vidět, že když oba stromy T_1 a T_2 jsou izomorfní s H_i, pak výsledný strom operace **spoj** je izomorfní s H_{i+1}. Korektnost operace **MERGE** plyne z tohoto pozorování a z faktu, že $\mathcal{H}_j$ obsahuje strom izomorfní s H_i, právě když v binárním zápise čísla $|S_j|$ je na i-tém místě zprava 1, a že T je neprázdný strom, když se provádí posun řádu při sčítání. Protože každý běh cyklu vyžaduje čas $O(1)$, algoritmus **MERGE** vyžaduje čas $O(\log(|S_1| + |S_2|))$. Implementace dalších algoritmů je podobná jako pro leftist haldy.

INSERT(x):
Vytvoř haldu $\mathcal{H}_1$ reprezentující $\{x\}$
MERGE$(\mathcal{H}, \mathcal{H}_1)$

MIN:
Prohledej prvky reprezentované kořeny všech stromů v $\mathcal{H}$
Výstup: nejmenší z těchto prvků

DELETEMIN:
Prohledej prvky reprezentované kořeny všech stromů v $\mathcal{H}$
$T :=$ strom, jehož kořen reprezentuje nejmenší prvek
$\mathcal{H}_1 := \mathcal{H} \setminus \{T\}$
$\mathcal{H}_2 :=$ halda tvořená podstromy T určenými syny kořene T
MERGE$(\mathcal{H}_1, \mathcal{H}_2)$

Z podmínky (usp) je zřejmé, že nejmenší prvek v S je reprezentován v kořeni nějakého stromu haldy. Tím je dána korektnost operace **MIN**. Z úvodního tvrzení plyne, že $\mathcal{H}_2$ v operaci **DELETEMIN** je binomiální halda, a odtud plyne korektnost operace **DELETEMIN**. Operace **DECREASE** se implementuje pomocí operace **UP** a operace **INCREASE** pomocí operace **DOWN** stejně jako v regulárních haldách. Struktura binomiální haldy nepodporuje přímo operaci **DELETE** – ta se dá realizovat jedině jako posloupnost operací **DECREASE**(s, ∞) a **DELETEMIN**. Operace **MAKEHEAP** se provádí opakováním operace **INSERT**.

Výpočet časové složitosti operací pro binomiální haldy využívá několik známých faktů. Operace **MERGE** simuluje sčítání přirozených čísel v binárním zápise a má tedy stejnou složitost. Odhad složitosti vytváření haldy využívá známého faktu, že amortizovaná složitost přičítání 1 k binárnímu číslu je $O(1)$. Odhad složitosti operací **MIN** a **DELETEMIN** je založen na pozorování, že binomiální halda reprezentující množinu S má tolik stromů, kolik je jedniček v binárním zápise $|S|$, a to je nejvýše $\log(|S|)$.

Z tvrzení také plyne, že výška všech stromů v binomiální haldě je $\leq \log(|S|)$ a počet synů kořene každého stromu je také $\leq \log(|S|)$, přičemž tento odhad se nedá zlepšit. Odtud dostáváme složitost operací **DECREASE** a **INCREASE** v nejhorším případě. Můžeme tedy shrnout:

Věta. *Pro binomiální haldy algoritmy operací* **INSERT**, **MIN**, **DELETEMIN**, **DECREASE** *a* **MERGE** *vyžadují čas* $O(\log(|S|))$, *algoritmus operace* **INCREASE** *vyžaduje čas* $O(\log^2(|S|))$ *a algoritmus operace* **MAKEHEAP** *čas* $O(|S|)$.

Z těchto výsledků je vidět, že předchozí typy hald mají efektivnější chování než binomiální haldy. Význam binomiálních hald tak spočívá především v tom, že se dají dále zobecnit (tímto zobecněním jsou Fibonacciho haldy) a že na nich lze krásně ilustrovat princip, že s řadou úprav je výhodné počkat a neprovádět je okamžitě.

Líná implentace operací

Následující algoritmy jsou založeny na ideji, že 'vyvažování' stačí provádět jen při operacích **MIN** a **DELETEMIN**, kdy je stejně zapotřebí prohledat všechny stromy. Z tohoto důvodu zeslabíme podmínky na binomiální haldy.

Líná binomiální halda $\mathcal{H}$ reprezentující množinu S je seznam stromů $\{T_1, T_2, \ldots, T_k\}$ takový, že

> celkový počet vrcholů v těchto stromech je roven velikosti S a existuje jednoznačné přiřazení prvků množiny S vrcholům stromů, které splňuje podmínku (usp) – toto přiřazení je jako obvykle realizováno funkcí key;
> každý strom T_i je izomorfní s nějakým stromem H_j.

V líné binomiální haldě je vynechán předpoklad neizomorfnosti stromů tvořících haldu. Tento fakt se projeví ve velmi jednoduchém algoritmu pro operaci **MERGE**.

MERGE($\mathcal{H}_1, \mathcal{H}_2$):
Proveď konkatenaci seznamů $\mathcal{H}_1$ a $\mathcal{H}_2$

Samotný algoritmus pro operaci **INSERT** se nezmění, jen provede tuto implementaci operace **MERGE**. Operace **MIN** a **DELETEMIN** použijí následující pomocnou proceduru **vyvaz**. Jejím vstupem je soubor seznamů $\{O_i \mid i = 0, 1, \ldots, k\}$, kde seznam O_i obsahuje jen stromy izomorfní se stromem H_i. Procedura **vyvaz** pak z těchto stromů vytvoří klasickou binomiální haldu.

vyvaz($\{O_i \mid i = 0, 1, \ldots, k\}$):
$i := 0$, $\mathcal{H} := \emptyset$
while existuje $O_i \neq \emptyset$ **do**
  while $|O_i| > 1$ **do**
    vezmi dva různé stromy T_1 a T_2 z O_i
    odstraň je z O_i
    spoj(T_1, T_2) vlož do O_{i+1}
  enddo
  if $O_i \neq \emptyset$ **then**
    strom $T \in O_i$ odstraň z O_i a vlož do $\mathcal{H}$
  endif,
  $i := i + 1$
enddo
Výstup: $\mathcal{H}$

MIN:
Prohledej prvky reprezentované kořeny všech stromů v $\mathcal{H}$
Výstup: nejmenší z těchto prvků
stromy rozděl do množin $O_i = \{$všechny stromy v $\mathcal{H}$ izomorfní s $H_i\}$
vyvaz($\{O_i \mid i = 0, 1, \ldots, \lfloor \log(|S|) \rfloor\}$)

DELETEMIN:
Prohledej prvky reprezentované kořeny všech stromů v $\mathcal{H}$
$T :=$ strom, jehož kořen reprezentuje nejmenší prvek
stromy rozděl do množin $O_i = \{$všechny stromy v $\mathcal{H}$ izomorfní s H_i různé od $T\} \cup \{$podstrom T určený nějakým synem kořene T izomorfní s $H_i\}$
vyvaz($\{O_i \mid i = 0, 1, \ldots, \lfloor \log(|S|) \rfloor\}$)

Časová složitost operací **INSERT** a **MERGE** při líné implementaci je $O(1)$, ale časová složitost operací **MIN** a **DELETEMIN** je v nejhorším případě $O(|S|)$. Tento odhad je velmi špatný, ale ukážeme, že amortizovaná složitost má rozumné hodnoty. Připomínáme, že amortizovaná složitost je čas operace plus ohodnocení výsledné struktury minus ohodnocení počáteční struktury. Konfiguraci ohodnotíme počtem stromů v haldě. Protože operace **MERGE** nemění počet stromů a protože operace **INSERT** přidá jen jeden strom, je

amortizovaná složitost operací **MERGE** a **INSERT** stále $O(1)$. Ukážeme, že amortizovaná složitost operací **MIN** a **DELETEMIN** při líné implementaci binomiálních hald je $O(\log(|S|)$. Protože každý běh vnitřního **while**-cyklu v operaci **vyvaz** vyžaduje čas $O(1)$ a zmenší počet stromů v seznamech O_i o 1, operace **vyvaz** vyžaduje čas $O(k + \sum_{i=0}^{k} |O_i|)$. Operace **MIN** bez podprocedury **vyvaz** vyžaduje čas $O(|\mathcal{H}|)$ a operace **DELETEMIN** bez podprocedury **vyvaz** čas $O(\mathcal{H} + i)$ pro takové i, že T je izomorfní s H_i. Podle tvrzení je $i \leq \log(|S|)$, a tedy operace **MIN** vyžaduje čas $O(|\mathcal{H}|)$ a operace **DELETEMIN** čas $O(|\mathcal{H}| + \log(|S|))$. Protože ohodnocení klasické binomiální haldy je nejvýše $\log(|S|)$ (obsahuje tolik stromů, kolik je 1 v binárním zápise čísla $|S|$), dostáváme, že amortizovaná složitost operace **MIN** je $O(|\mathcal{H}| - |\mathcal{H}| + \log(|S|)) = O(\log(|S|))$ a amortizovaná složitost operace **DELETEMIN** je $O(|\mathcal{H}| + \log(|S|) - |\mathcal{H}| + \log(|S|)) = O(\log(|S|))$.

Protože si funkci ohodnocení volíme, můžeme použít takové multiplikativní koeficienty, aby jednotka času odpovídala jednotce v amortizované složitosti. Proto lze $|\mathcal{H}|$ od sebe odečíst.

VI. Fibonacciho haldy

Význam Fibonacciho hald určuje fakt, že amortizovaná složitost operací **INSERT** a **DECREASE** v těchto haldách je $O(1)$ a amortizovaná složitost operace **DELETEMIN** je $O(\log(|S|)$. Proto se hodně používají v grafových algoritmech, kde umožňují v mnoha případech dosáhnout asymptoticky téměř lineární složitosti. Neznáme však žádné experimentální výsledky, které by porovnávaly použití Fibonacciho hald a např. d-regulárních hald v těchto grafových algoritmech v praxi. Takže neznáme podmínky, za kterých jsou Fibonacciho haldy lepší než třeba d-regulární haldy, ani nevíme, do jaké míry je to jen teoretický výsledek a do jaké míry jsou opravdu prakticky použitelné.

Neformálně řečeno, je Fibonacciho halda množina stromů, jejichž některé vrcholy různé od kořenů jsou označeny, a kde existuje jednoznačná korepondence mezi prvky S a vrcholy stromů (realizována funkcí key), která splňuje podmínku (usp). Toto je však jen přibližné vyjádření. Existují totiž struktury, na které se tento popis hodí, ale nevznikly z prázdné Fibonacciho haldy aplikací posloupnosti haldových operací. Přitom důkaz efektivity Fibonacciho hald se dosti výrazně opírá o fakt, že halda vznikla z prázdné haldy aplikací algoritmů pro Fibonacciho haldy. Proto nejprve popíšeme algoritmy pro tyto operace, a pak budeme definovat Fibonacciho haldy jako struktury vzniklé z prázdné haldy aplikací posloupnosti těchto algoritmů.

Algoritmy

V algoritmech předpokládáme, že Fibonacciho halda je seznam stromů, kde některé vrcholy různé od kořenů jsou označeny. Vrchol je označen, právě když není kořen a když mu byl někdy dříve odtržen některý jeho syn. Toto se nezaznamenává pro kořeny stromů. Proto když se vrchol stane kořenem (odtržením podstromu určeného tímto vrcholem), zapomene se tento údaj a začne se znovu zaznamenávat, až když vrchol přestane být kořenem. Řekneme, že strom má rank i, když jeho kořen má i synů. Tento fakt nahrazuje test používaný v binomiálních haldách, že strom je izomorfní se stromem H_i.

Algoritmy pro operace **MERGE**, **INSERT**, **MIN** a **DELETEMIN** jsou založeny na stejných idejích jako algoritmy pro línou implementaci v binomiálních haldách, pouze požadavek, aby strom byl izomorfní s H_i, je nahrazen požadavkem, že má rank i. Algoritmy pro

operace **DECREASE**, **INCREASE** a **DELETE** vycházejí z algoritmů pro tyto operace v leftist haldách. V algoritmech předpokládáme, že $c = \log^{-1}(\frac{3}{2})$.

MERGE$(\mathcal{H}_1, \mathcal{H}_2)$:
Proveď konkatenaci seznamů $\mathcal{H}_1$ a $\mathcal{H}_2$

INSERT(x):
Vytvoř haldu $\mathcal{H}_1$ reprezentující $\{x\}$
MERGE$(\mathcal{H}, \mathcal{H}_1)$

MIN:
Prohledej prvky reprezentované kořeny všech stromů v $\mathcal{H}$
Výstup: nejmenší z těchto prvků
stromy rozděl do množin $O_i = \{$všechny stromy v $\mathcal{H}$ s rankem $i\}$
vyvaz1$(\{O_i \mid i = 0, 1, \ldots, \lfloor c \log(\sqrt{5}|S| + 1)\rfloor\})$

DELETEMIN:
Prohledej prvky reprezentované kořeny všech stromů v $\mathcal{H}$
$T :=$ strom, jehož kořen reprezentuje nejmenší prvek
stromy rozděl do množin $O_i = \{$všechny stromy v $\mathcal{H}$ s rankem i různé od $T\} \cup \{$podstrom T určený některým synem kořene T s rankem $i\}$
vyvaz1$(\{O_i \mid i = 0, 1, \ldots, \lfloor c \log(\sqrt{5}|S| + 1)\rfloor\})$

vyvaz1$(\{O_i \mid i = 0, 1, \ldots, k\})$:
$i := 0$, $\mathcal{H} := \emptyset$
while existuje $O_i \neq \emptyset$ **do**
  while $|O_i| > 1$ **do**
    vezmi dva různé stromy T_1 a T_2 z O_i
    odstraň je z O_i
    spoj(T_1, T_2) vlož do O_{i+1}
  enddo
  if $O_i \neq \emptyset$ **then**
    strom $T \in O_i$ odstraň z O_i a vlož ho do $\mathcal{H}$
  endif
  $i := i + 1$
enddo
Výstup: $\mathcal{H}$

spoj(T_1, T_2):
if f(kořen T_1) $>$ f(kořen T_2) **then**
  vyměň stromy T_1 a T_2
endif
vytvoř nového syna v kořene T_1
$v :=$kořen T_2

DECREASE(s, a):
$T :=$strom v $\mathcal{H}$, který obsahuje vrchol reprezentující s

v :=vrchol stromu T reprezentující s
if v není kořen **then**
  odtrhni podstrom T' určený vrcholem v
  vyvaz2(T, v)
  if v byl označen **then** zruš označení v **endif**
  vlož T' do $\mathcal{H}$
endif
$f(v) := f(v) - a$

INCREASE(s, a):
T :=strom v $\mathcal{H}$, který obsahuje vrchol reprezentující s
v :=vrchol stromu T reprezentující s
if v není list **then**
  odtrhni podstrom T' určený vrcholem v
  if v není kořen **then vyvaz2**(T, v) **endif**
  if v byl označen **then** zruš označení v **endif**
  zruš označení všech synů vrcholu v
  odtrhni podstromy T' určené všemi syny v a vlož je do $\mathcal{H}$
  do $\mathcal{H}$ vlož strom mající jen vrchol v
endif
$f(v) := f(v) + a$

DELETE(s):
T :=strom v $\mathcal{H}$, který obsahuje vrchol reprezentující s
v :=vrchol stromu T reprezentující s
if v není list **then**
  zruš označení synů vrcholu v
  odtrhni podstromy určené všemi syny vrcholu v a vlož je do $\mathcal{H}$
endif
if v není kořen **then vyvaz2**(T, v) **endif**
zruš vrchol v

vyvaz2(T, v):
$u :=$ otec v
while u je označen **do**
  $u' :=$ otec(u), zruš označení u
  odtrhni podstrom T' určený vrcholem u
  vlož T' do $\mathcal{H}$, $u := u'$
enddo
if u není kořen T **then** označ u **endif**

Všimněme si, že když stromy T_1 a T_2 mají rank i, pak procedura **spoj**(T_1, T_2) vytvoří strom s rankem $i + 1$. Aby algoritmy pro operace **MIN** a **DELETEMIN** byly korektní, musíme ukázat, že všechny stromy ve Fibonacciho haldě $\mathcal{H}$ reprezentující množinu S mají rank nejvýše $c \log(\sqrt{5}|S| + 1)$. Jen tak zajistíme, aby výsledná halda reprezentovala S, respektive $S \setminus \{$prvek s nejmenší hodnotou $f\}$. Operace **vyvaz1** zajišťuje, že od každého

vrcholu stromu různého od kořene byl v tomto stromě odtržen podstrom nejvýše jednoho syna – v tom případě je tento prvek označen a když se mu odtrhává podstrom dalšího syna, bude odtržen i celý podstrom tohoto vrcholu (tím se tento vrchol stane kořenem stromu). Když se později stane tento vrchol zase vrcholem různým od kořene, celý proces se opakuje.

Složitost operací

Naším cílem bude odhadnout amortizovanou složitost těchto operací, protože složitost v nejhorším případě není použitelný výsledek. Abychom to mohli udělat, spočítáme parametry složitosti jednotlivých operací:

MERGE – časová složitost $O(1)$, nevzniká žádný nový strom, označené vrcholy se nemění;
INSERT – časová složitost $O(1)$, přibyl jeden strom, označené vrcholy se nemění;
MIN – časová složitost $O(|\mathcal{H}|)$, po provedení operace různé stromy v haldě mají různé ranky, označené vrcholy se nemění;
DELETEMIN – časová složitost $O(|\mathcal{H}| + \text{počet synů } v)$, kde v reprezentoval prvek s nejmenší hodnotou f. Po provedení operace různé stromy v haldě mají různé ranky, žádný nový vrchol nebyl označen, některé označené vrcholy přestaly být označené;
DECREASE – časová složitost $O(1+c)$, kde c je počet vrcholů, které přestaly být označené. Bylo přidáno $1 + c$ nových stromů a byl označen nejvýše jeden vrchol;
INCREASE – časová složitost $O(1 + c + d)$, kde c je počet vrcholů, které přestaly být označené, d je počet synů vrcholu v reprezentujícího prvek, jehož hodnota se zvyšuje. Bylo přidáno nejvýše $1 + c + d$ nových stromů a byl označen nejvýše jeden vrchol;
DELETE – časová složitost $O(1+c+d)$, kde c je počet vrcholů, které přestaly být označené, d je počet synů vrcholu v reprezentujícího prvek, který se má odstranit. Bylo přidáno nejvýše $c + d$ nových stromů a byl označen nejvýše jeden vrchol.

Pro výpočet amortizované složitosti musíme nejprve navrhnout funkci ohodnocující konfigurace. Při vyšetřování líné implementace binomiálních hald se ukázalo, že vhodným ohodnocením je počet stromů v haldě. Když si ale prohlédneme algoritmus pro operaci **DECREASE**, vidíme, že zde je vhodné brát do ohodnocení i počet označených vrcholů, a to dokonce tak, aby se pokryl nejen čas, ale i přírůstek stromů. To vede k následujícímu ohodnocení konfigurace: ohodnocení je počet stromů v konfiguraci plus dvojnásobek počtu označených vrcholů.

Nechť $\rho(n)$ je maximální počet synů vrcholu ve Fibonacciho haldě reprezentující n-prvkovou množinu. Pak amortizovaná složitost operací **MERGE**, **INSERT** a **DECREASE** je $O(1)$ a operací **MIN**, **DELETEMIN**, **INCREASE** a **DELETE** je $O(\rho(n))$.

Abychom spočítali odhad $\rho(n)$, využijeme toho, že Fibonacciho halda vznikla z prázdné haldy pomocí popsaných algoritmů. Nejprve uvedeme jedno technické lemma.

Lemma. *Nechť v je vrchol stromu ve Fibonacciho haldě a nechť u je i-tý nejstarší syn vrcholu v. Pak u má aspoň $i - 2$ synů.*

Důkaz. V momentě, kdy se u stával synem v, se aplikovala operace **spoj**, u a v byly kořeny stromů a měly stejný počet synů. Podle předpokladů měl vrchol v alespoň $i - 1$ synů (jinak by u nebyl i-tý nejstarší syn), a protože se od u mohl odtrhnout jen jeden syn, dostáváme, že u musí mít alespoň $i - 2$ synů. □

Tvrzení. *Nechť v je vrchol stromu ve Fibonacciho haldě, který má právě i synů. Pak podstrom určený vrcholem v má aspoň F_{i+2} vrcholů.*

Důkaz. Tvrzení dokážeme indukcí podle maximální délky cesty z vrcholu v do některého listu. Tato délka je 0, právě když v je list. V tom případě v nemá syna a podstrom určený vrcholem v má jediný vrchol. Protože $1 = F_2 = F_{0+2}$, tvrzení platí. Mějme nyní vrchol v, který má k synů, a nechť maximální délka cesty z vrcholu v do listů je j. Předpokládejme, že tvrzení platí pro všechny vrcholy, pro něž tato délka je menší než j, tedy platí i pro všechny syny vrcholu v. Pak pro $i > 1$ má i-tý nejstarší syn vrcholu v podle předchozího lemmatu alespoň $i-2$ synů a podle indukčního předpokladu podstrom určený tímto synem má alespoň F_i vrcholů. Odtud dostáváme, že podstrom určený vrcholem v má alespoň

$$1 + F_2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=1}^{k} F_i$$

vrcholů, protože $F_1 = F_2$ (na levé straně první 1 je za vrchol v a první F_2 je za nejstarší vrchol). Indukcí pak dostaneme, že

$$1 + \sum_{i=1}^{n} F_i = F_{n+2}$$

pro všechna $n \geq 0$. Skutečně, pro $n = 0$ platí

$$1 + \sum_{i=1}^{0} F_i = 1 = F_2 = F_{0+2},$$

pro $n = 1$ máme

$$1 + \sum_{i=1}^{1} F_i = 1 + F_1 = 2 = F_3 = F_{1+2}$$

a z indukčního předpokladu a z vlastností Fibonacciho čísel plyne, že

$$1 + \sum_{i=1}^{n} F_i = 1 + \sum_{i=1}^{n-1} F_i + F_n = F_{n+1} + F_n = F_{n+2}.$$

Když shrneme tato fakta, dostáváme, že podstrom určený vrcholem v má alespoň F_{i+2} vrcholů, a tvrzení je dokázáno. □

Vezměme nyní nejmenší i takové, že $n < F_i$. Protože posloupnost $\{F_i\}_{i=1}^{\infty}$ je rostoucí, plyne z předchozího tvrzení, že každý vrchol ve Fibonacciho haldě reprezentující $n-$prvkovou množinu má méně než $i-2$ synů (když vrchol v Fibonacciho haldy má $i-2$ synů, pak podstrom vrcholu v reprezentuje množinu alespoň s F_i prvky). Proto $\rho(n) < i-2$. K odhadu velikosti i použijeme explicitní vzorec pro i-té Fibonacciho číslo:

$$F_i = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^i.$$

Protože $0 > \frac{1-\sqrt{5}}{2} > -\frac{3}{4}$ a protože $\sqrt{5} > 2$, dostáváme, že $|\frac{1}{\sqrt{5}}\big(\frac{1-\sqrt{5}}{2}\big)^i| < \frac{3}{8}$ pro všechna $i = 1, 2, \ldots$, a tedy

$$\frac{1}{\sqrt{5}}\big(\frac{1+\sqrt{5}}{2}\big)^i - \frac{3}{8} < F_i < \frac{1}{\sqrt{5}}\big(\frac{1+\sqrt{5}}{2}\big)^i + \frac{3}{8}.$$

Odtud plyne, že když i splňuje

$$n \leq \frac{1}{\sqrt{5}}\big(\frac{1+\sqrt{5}}{2}\big)^i - \frac{3}{8},$$

pak $n < F_i$. Převedením $\frac{3}{8}$ na druhou stranu výrazu, jeho vynásobením $\sqrt{5}$ a zlogaritmováním dostaneme následující ekvivalenci:

$$\log_2(\sqrt{5}n + \frac{3\sqrt{5}}{8}) \leq i \log_2\big(\frac{1+\sqrt{5}}{2}\big) \quad \Leftrightarrow \quad n \leq \frac{1}{\sqrt{5}}\big(\frac{1+\sqrt{5}}{2}\big)^i - \frac{3}{8}.$$

Z $\frac{3\sqrt{5}}{8} < 1$ a z $\frac{3}{2} < \frac{1+\sqrt{5}}{2}$ plyne, že

$$\frac{\log_2(\sqrt{5}n + \frac{3\sqrt{5}}{8})}{\log_2 \frac{1+\sqrt{5}}{2}} < \frac{\log_2(\sqrt{5}n+1)}{\log_2 \frac{3}{2}}.$$

Tedy platí následující implikace

$$\frac{\log_2(\sqrt{5}n+1)}{\log_2 \frac{3}{2}} < i \quad \Longrightarrow \frac{\log_2(\sqrt{5}n + \frac{3\sqrt{5}}{8})}{\log_2\big(\frac{1+\sqrt{5}}{2}\big)} < i.$$

Proto když $\frac{\log_2(\sqrt{5}n+1)}{\log_2 3-1} < i$, pak $n < F_i$, a tedy $\rho(n) < i - 2$.

Výsledky shrneme do následující věty:

Věta. *Ve Fibonacciho haldě, která reprezentuje n-prvkovou množinu, má každý vrchol stupeň menší než*

$$\frac{\log_2(\sqrt{5}n+1)}{(\log_2 3)-1} - 2.$$

Amortizovaná složitost operací **INSERT**, **MERGE** *a* **DECREASE** *je* $O(1)$ *a amortizovaná složitost operací* **MIN**, **DELETEMIN**, **INCREASE** *a* **DELETE** *je* $O(\log n)$. *Operace* **MIN** *a* **DELETEMIN** *jsou korektní.*

Pro úplnost dokážeme, že $F_i = \frac{\big(\frac{1+\sqrt{5}}{2}\big)^i - \big(\frac{1-\sqrt{5}}{2}\big)^i}{\sqrt{5}}$.
Pro $i = 1$ platí

$$\frac{\big(\frac{1+\sqrt{5}}{2}\big)^1 - \big(\frac{1-\sqrt{5}}{2}\big)^1}{\sqrt{5}} = \frac{1+\sqrt{5}-1+\sqrt{5}}{2\sqrt{5}} = \frac{2\sqrt{5}}{2\sqrt{5}} = 1 = F_1.$$

Pro $i = 2$ platí

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^2 - \left(\frac{1-\sqrt{5}}{2}\right)^2}{\sqrt{5}} = \frac{1 + 2\sqrt{5} + 5 - 1 + 2\sqrt{5} - 5}{4\sqrt{5}} = \frac{4\sqrt{5}}{4\sqrt{5}} = 1 = F_2.$$

Indukční krok:

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i-2}\left(\frac{1+\sqrt{5}}{2}\right)^2 - \left(\frac{1-\sqrt{5}}{2}\right)^{i-2}\left(\frac{1-\sqrt{5}}{2}\right)^2}{\sqrt{5}} =$$
$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i-2}\left(\frac{3+\sqrt{5}}{2}\right) - \left(\frac{1-\sqrt{5}}{2}\right)^{i-2}\left(\frac{3-\sqrt{5}}{2}\right)}{\sqrt{5}} =$$
$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i-2}\left(1 + \frac{1+\sqrt{5}}{2}\right) - \left(\frac{1-\sqrt{5}}{2}\right)^{i-2}\left(1 + \frac{1-\sqrt{5}}{2}\right)}{\sqrt{5}} =$$
$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i-2} + \left(\frac{1+\sqrt{5}}{2}\right)^{i-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{i-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{i-1}}{\sqrt{5}} =$$
$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{i-2}}{\sqrt{5}} + \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{i-1}}{\sqrt{5}} = F_{i-2} + F_{i-1} = F_i.$$

Tedy indukcí dostáváme požadovaný vztah.

Aplikace

Vrátíme se k Dijkstrově algoritmu. Množinu U budeme reprezentovat pomocí Fibonacciho haldy. Protože ohodnocení je nezáporné a ohodnocení počáteční haldy je 0, dává odhad amortizované složitosti také odhad časové složitosti (viz odstavec IV.). Proto Dijkstrův algoritmus s použitím Fibonacciho haldy vyžaduje v nejhorším případě čas $O(|X|(1+\log|X|)+|R|) = O(|R| + |X|\log|X|)$. Stejný výsledek dostaneme i pro konstrukci nejmenší napnuté kostry grafu.

Otázka je, kdy v Dijkstrově algoritmu nebo v algoritmu konstruujícím nejmenší napnutou kostru použít Fibonacciho haldu a kdy např. d-regulární haldy. Lze říci, že Fibonacciho halda by měla být výrazně lepší pro větší, ale řídké grafy (tj. grafy s malým počtem hran). Dá se předpokládat, že d-regulární haldy budou lepší (díky svým jednodušším algoritmům) pro husté grafy (tj. grafy, kde počet hran je $|X|^{1+\varepsilon}$ pro vhodné $\varepsilon > 0$). Problém je, pro které hodnoty nastává zlom. Nevím o žádných experimentálních ani teoretických výsledcích tohoto typu.

Historický přehled

Binární neboli 2-regulární haldy zavedl Williams 1964. Jejich zobecnění na d-regulární haldy pochází od Johnsona 1975. Leftist haldy definoval Crane 1972 a detailně popsal Knuth 1975. Binomiální haldy navrhnl Vuillemin 1978, Brown 1978 je implementoval a prokázal jejich praktickou použitelnost. Fibonacciho haldy byly zavedeny Fredmanem a Tarjanem 1987.

VII. Třídicí algoritmy

Jednou z nejčastěji řešených úloh při práci s daty je setřídění posloupnosti prvků nějakého typu. Proto velká pozornost byla a je věnována třídicím algoritmům řešícím tuto úlohu, která svým charakterem a svými požadavky na algoritmy je řazena do datových struktur. Byla navržena řada algoritmů, které se stále ještě analyzují a optimalizují. Analýzy jsou velmi detailní a algoritmy se studují za různých vstupních předpokladů. Kromě toho třídění je jedna z mála úloh, pro kterou alespoň za jistých předpokladů umíme spočítat dolní odhad složitosti.

Formulace úlohy:
Nechť U je totálně uspořádané univerzum.
Vstup: Prostá posloupnost $\{a_1, a_2, \ldots, a_n\}$ prvků z univerza U.
Výstup: Rostoucí posloupnost $\{b_1, b_2, \ldots, b_n\}$ taková, že $\{a_i \mid i = 1, 2, \ldots, n\} = \{b_i \mid i = 1, 2, \ldots, n\}$.
Tento problém se nazývá třídění. V praxi se setkáváme s řadou jeho modifikací, a nichž asi nejběžnější je vynechání předpokladu, že vstupem je prostá posloupnost. Pak jsou dvě varianty řešení – buď se ve výstupní posloupnosti odstraní duplicity nebo výstupní posloupnost zachová četnost prvků ze vstupní posloupnosti.

Základní algoritmy, které řeší třídicí problém, jsou **QUICKSORT**, **MERGESORT** a **HEAPSORT**.

HEAPSORT

S algoritmem **HEAPSORT** jsme se seznámili při aplikacích hald. Byl to první algoritmus používající haldy (binární regulární haldy byly definovány právě při návrhu **HEAPSORTU**). Podíváme se detailněji na jednu z jeho implementací, která třídí takzvaně na místě.

Třídicí algoritmy se často používají jako podprocedura při řešení jiných úloh. V takovém případě je obvykle vstupní posloupnost uložena v poli v pracovní paměti programu a požadavkem je setřídit ji bez použití další paměti pouze s výjimkou omezeného (malého) počtu pomocných proměnných. Pro řešení tohoto problému se hodí **HEAPSORT**. Zvolíme implementaci **HEAPSORTU** pomocí d-regulárních hald, které jsou reprezentovány polem, v němž je uložena vstupní posloupnost (viz odstavec Aplikace v kapitole o d-regulárních haldách). Použijeme algoritmus s jedinou změnou – budeme požadovat duální podmínku na uspořádání (to znamená, že prvek reprezentovaný vrcholem bude menší než prvek reprezentovaný jeho otcem) a nahradíme operace **MIN** a **DELETEMIN** operacemi **MAX** a **DELETEMAX**. V algoritmu vždy umístíme odebrané maximum na místo prvku v posledním listu haldy (tj. prvku, který ho při operaci **DELETEMAX** nahradil) místo toho, abychom ho vložili do výstupní posloupnosti. Musíme si ale pamatovat, kde v poli končí reprezentovaná halda. Každá aplikace operace **DELETEMAX** zkrátí počáteční úsek pole reprezentujícího haldu o jedno místo a zároveň o toto místo zvětší druhou část, ve které je uložena již setříděná část posloupnosti.

HEAPSORTU je stále věnována velká pozornost a bylo navrženo několik jeho modifikací, snažících se např. minimalizovat počet porovnání prvků apod.

MERGESORT

Nejstarší z uvedených algoritmů je **MERGESORT**, který je starší než je počítačová éra, neboť některé jeho verze se používaly už při mechanickém třídění. Popíšeme jednu jeho iterační verzi, tzv. přirozený **MERGESORT**.

MERGESORT$(a_1, a_2, \ldots, a_n)$:
Q je prázdná fronta, $i = 1$
while $i \leq n$ **do**
  $j := i$
  while $i < n$ a $a_{i+1} > a_i$ **do** $i := i + 1$ **enddo**
  posloupnost $P = (a_j, a_{j+1}, \ldots, a_i)$ vlož do Q
  $i := i + 1$
enddo
while $|Q| > 1$ **do**
  vezmi P_1 a P_2 dvě posloupnosti z vrcholu Q
  odstraň P_1 a P_2 z Q
  MERGE(P_1, P_2) vlož na konec Q
enddo
Výstup: posloupnost z Q

MERGE$(P_1 = (a_1, a_2, \ldots, a_n), P_2 = (b_1, b_2, \ldots, b_m))$:
$P :=$ je prázdná posloupnost, $i := 1$, $j := 1$, $k := 1$
while $i \leq n$ a $j \leq m$ **do**
  if $a_i < b_j$ **then**
    $c_k := a_i$, $i := i + 1$, $k := k + 1$
  else
    $c_k := b_j$, $j := j + 1$, $k := k + 1$
  endif
enddo
while $i \leq n$ **do**
  $c_k := a_i$, $i := i + 1$, $k := k + 1$
enddo
while $j \leq m$ **do**
  $c_k := b_j$, $j := j + 1$, $k := k + 1$
enddo
Výstup: $P = (c_1, c_2, \ldots, c_{n+m})$

Všimněme si, že všechny posloupnosti v Q jsou rostoucí a že sjednocením všech jejich prvků je vždy na začátku běhu cyklu **while** $|Q| > 1$ **do** množina $\{a_i \mid i = 1, 2, \ldots, n\}$. Protože počet posloupností ve frontě Q je nejvýše roven délce vstupní posloupnosti a každý průběh tohoto cyklu zmenší jejich počet o 1, je algoritmus **MERGESORT** korektní.

Spočítáme časovou složitost **MERGESORTU**. Nejprve vyšetříme složitost podprocedury **MERGE**. Protože určení prvku c_k vyžaduje čas $O(1)$ (provede se nejvýše jedno porovnání) a protože maximální hodnota k je $n + m$, dostáváme, že podprocedura **MERGE** vyžaduje čas $O(n + m)$ (nejvýše $n + m$ porovnání), kde n a m jsou délky vstupních posloupností.

Nyní vypočteme složitost hlavní procedury. Zřejmě první cyklus vyžaduje lineární čas. Vyšetříme druhý cyklus probíhající přes frontu Q. Předpokládejme, že před prvním během tohoto cyklu je na vrcholu Q speciální znak $\natural$, který se vždy pouze přenese z vrcholu Q na její konec. Protože mezi dvěma přenosy $\natural$ projde každý prvek vstupní posloupnosti podprocedurou **MERGE** právě jednou, vyžadují jednotlivé běhy cyklu čas $O(n)$, kde n je délka vstupní posloupnosti (a zároveň součet všech délek posloupností v Q). Všechny posloupnosti v Q mají na počátku délku ≥ 1. Odtud jednoduchou indukcí dostaneme, že po i-tém přenosu znaku $\natural$ mají délku $\geq 2^{i-1}$. Proto počet přenosů je nejvýše $\lceil \log_2 n \rceil$, a tedy algoritmus **MERGESORT** vyžaduje čas $O(n \log n)$ (provede se nejvýše $n \log n$ porovnání).

Vzhledem k počtu porovnání je **MERGESORT** optimální třídicí algoritmus. Navíc v této verzi je adaptivní na předtříděné posloupnosti, které mají jen malý počet dlouhých setříděných úseků (běhů). Při konstantním počtu běhů má složitost $O(n)$. Jiná jeho verze, která začíná slévání vždy od jednoprvkových posloupností (tzv. přímý **MERGESORT**) tuto vlastnost nemá.

QUICKSORT

Nyní popíšeme patrně vůbec nejpoužívanější třídicí algoritmus, kterým je **QUICKSORT**. Důvodem je, že pro obecnou posloupnost je nejrychlejší, při rovnoměrném rozložení vstupních polsoupností má nejmenší očekávaný čas.

Quick$(a_i, a_{i+1}, \ldots, a_j)$:
if $i = j$ **then**
  Výstup: (a_i)
else
  zvol k takové, že $i \leq k \leq j$, $a := a_k$, vyměň a_i a a_k, $l := i + 1$, $q := j$
  while true **do**
    while $a_l < a$ **do** $l := l + 1$ **enddo**
    while $a_q > a$ **do** $q := q - 1$ **enddo**
    if $l \geq q$ **then**
      exit
    else
      vyměň a_l a a_q, $l := l + 1$, $q := q - 1$
    endif
  enddo
  if $i + 1 = l$ **then**
    Výstup$(a,$**Quick**$(a_{q+1}, a_{q+2}, \ldots, a_j))$
  else
    if $j = q$ **then**
      Výstup(**Quick**$(a_{i+1}, a_{i+2}, \ldots, a_{l-1}), a)$
    else
      Výstup(**Quick**$(a_{i+1}, a_{i+2}, \ldots, a_{l-1}), a,$**Quick**$(a_{q+1}, \ldots, a_j))$
    endif
  endif
endif

QUICKSORT$(a_1, a_2, \ldots, a_n)$:
Výstup(**Quick**$(a_1, a_2, \ldots, a_n)$)

Algoritmus **Quick** setřídí posloupnost $(a_i, a_{i+1}, \ldots, a_j)$ tak, že pro prvek $a = a_k$ vytvoří posloupnost $(a_i, a_{i+1}, \ldots, a_{l-1})$ všech prvků menších než a a posloupnost $(a_{q+1}, \ldots, a_j)$ všech prvků větších než a. Na tyto posloupnosti pak zavolá sám sebe a do výsledné posloupnosti uloží nejprve setříděnou první posloupnost, pak prvek a a nakonec setříděnou druhou posloupnost. Korektnost procedury **Quick** i algoritmu **QUICKSORT** je tedy zřejmá, protože $l \leq j$ a $i \leq q$.

Procedura **Quick** bez rekurzivního volání vyžaduje čas $O(j-i)$. Tedy kdyby a_k byl medián (tj. prostřední prvek) posloupnosti $(a_i, a_{i+1}, \ldots, a_j)$, pak by algoritmus **QUICKSORT** v nejhorším případě vyžadoval čas $O(n \log n)$. Jak uvidíme později, medián lze sice nalézt v lineárním čase, ale použití jakékoli procedury pro jeho nalezení má za následek, že algoritmy **MERGESORT** a **HEAPSORT** budou rychlejší (nikoliv asymptoticky, ale multiplikativní konstanta bude v tomto případě vysoká). Proto je třeba vybrat prvek a_k (tzv. pivot) co nejrychleji. Původně se bral první nebo poslední prvek posloupnosti. Při této volbě a při rovnoměrném rozdělení vstupů je očekávaný čas **QUICKSORTU** $O(n \log n)$ a algoritmus je obvykle rychlejší než **MERGESORT** a **HEAPSORT**. Avšak čas v nejhorším případě je kvadratický a dokonce pro určitá rozdělení vstupních dat je i očekávaný čas kvadratický. Proto tuto volbu pivota není vhodné používat pro úlohy, kdy neznáme rozdělení vstupních dat (mohlo by se stát, že je nevhodné). Jednoduše to lze napravit tak, že budeme volit k náhodně. Bohužel použití pseudonáhodného generátoru také vyžaduje jistý čas, a pak už by algoritmus zase nemusel být rychlejší než algoritmy **MERGESORT** a **HEAPSORT** (navíc takto náhodně zvolený prvek není skutečně náhodný, ale to v tomto případě nevadí). Důsledkem je návrh vybírat pivota jako medián ze tří nebo pěti pevně zvolených prvků posloupnosti. Praxe ukázala, že tento výběr pivota je nejpraktičtější, dá se provést rychle a zajišťuje dostatečnou náhodnost.

Protože při každém volání má **Quick** jako argument kratší vstupní posloupnost, lze ukázat, že:

(1) při každé volbě pivota je nejhorší čas algoritmu **QUICKSORT** $O(n^2)$,
(2) pokud je pivot vybrán jednoduchým a rychlým způsobem (to platí, i když se volí náhodně), pak existují vstupní posloupnosti, které vyžadují čas $O(n^2)$,
(3) očekávaný čas je $O(n \log n)$.

Následná analýza očekávaného případu je pro náhodně zvoleného pivota (bez dalšího předpokladu na vstupní data) nebo pro případ, kdy pivot je pevně zvolen a data jsou rovnoměrně rozdělena.

Ukážeme dva způsoby výpočty očekávaného času. Jeden je založen na několika jednoduchých pozorováních a není v něm mnoho počítání, druhý na rekurzivním výpočtu. Ten je početně náročnější, ale postup je standardní. Hlavní idea v obou případech spočívá v tom, že očekávaný čas algoritmu **QUICKSORT** je úměrný očekávanému počtu porovnání v algoritmu **QUICKSORT**. Tento fakt plyne přímo z popisu algoritmu. Budeme tedy počítat očekávaný počet porovnání pro algoritmus **QUICKSORT**.

První způsob výpočtu:
Každé dva prvky a_i a a_j algoritmus **QUICKSORT** porovná při třídění posloupnosti

$(a_1, a_2, \ldots, a_n)$ nejvýše jednou, přičemž když porovnává a_i a a_j, pak pro nějaký běh podprocedury **Quick** je a_i nebo a_j pivot, ale v předchozích bězích **Quick** a_i ani a_j nebyl pivotem (protože pivot je vždy vyřazen z následujících volání této podprocedury).

Nechť $(b_1, b_2, \ldots, b_n)$ je výsledná posloupnost. Označme $X_{i,j}$ boolskou proměnou, která má hodnotu 1, když **QUICKSORT** provedl porovnání mezi prvky b_i a b_j, a jinak má hodnotu 0. Předpokládejme, že je to náhodná veličina. Když $p_{i,j}$ je pravděpodobnost, že $X_{i,j} = 1$, pak očekávaná hodnota $X_{i,j}$ je

$$\mathbf{E}(X_{i,j}) = 0(1 - p_{i,j}) + 1p_{i,j} = p_{i,j}.$$

Protože počet porovnání při běhu algoritmu **QUICKSORT** je

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{i,j}$$

a protože očekávaná hodnota součtu náhodných proměnných je součtem očekávaných hodnot, dostáváme, že očekávaný počet porovnání v algoritmu **QUICKSORT** je

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}(X_{i,j}) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j}.$$

Abychom spočítali $p_{i,j}$, popíšeme chování algoritmu **QUICKSORT** pomocí modifikace stromu výpočtu. Bude to binární strom, v němž každý vrchol odpovídá jednomu běhu podprocedury **Quick**. Vrchol v bude vnitřním vrcholem, když odpovídající podprocedura volila pivota, a tento pivot bude ohodnocením v. V podstromu levého syna vrcholu v budou právě všechna následující rekurzivní volání podprocedury **Quick** nad částí posloupnosti, která předchází pivotu. Analogicky v podstromu pravého syna vrcholu v budou právě všechna následující rekurzivní volání procedury **Quick** nad částí posloupnosti, která následuje po pivotu. Listy stromu odpovídají volání procedury **Quick** nad jednoprvkovými posloupnostmi a každý takový jednotlivý prvek ohodnocuje příslušný list. Když vrchol v odpovídá volání **Quick** nad posloupností $(a_i, a_{i+1}, \ldots, a_j)$, pak vrcholy v podstromu levého syna v jsou ohodnoceny prvky z posloupnosti $(a_i, a_{i+1}, \ldots, a_{l-1})$ a vrcholy v podstromu pravého syna vrcholu v jsou ohodnoceny prvky z posloupnosti $(a_{q+1}, \ldots, a_j)$ (po přerovnání). Dále platí $\{a_l \mid i \leq l \leq j\} = \{b_l \mid i \leq l \leq j\}$.

Očíslujeme vrcholy tohoto stromu prohledáváním do šířky za předpokladu, že levý syn vrcholu předchází pravému synu. Nechť $(c_1, c_2, \ldots, c_n)$ je posloupnost prvků $\{a_i \mid 1 \leq i \leq n\}$ v pořadí daném tímto očíslováním. Pak platí, že $X_{i,j} = 1$, právě když první prvek v posloupnosti $(c_1, c_2, \ldots, c_n)$ z množiny $\{b_l \mid i \leq l \leq j\}$ je buď b_i nebo b_j. Pravděpodobnost tohoto jevu je $\frac{2}{j-i+1}$, tedy $p_{i,j} = \frac{2}{j-i+1}$ pro $1 \leq i < j \leq n$. Odtud očekávaný počet porovnání v algoritmu **QUICKSORT** je

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq 2n(\sum_{k=2}^{n} \frac{1}{k}) \leq 2n \int_1^n \frac{1}{x} dx = 2n \ln n.$$

Druhý způsob výpočtu:
Označme $QS(n)$ očekávaný počet porovnání provedených algoritmem **QUICKSORT** při třídění n-členné posloupnosti. Pak platí

$$QS(0) = QS(1) = 0 \text{ a}$$

$$QS(n) = \frac{1}{n}\big(\sum_{k=0}^{n-1} n - 1 + QS(k) + QS(n-k-1)\big) = n - 1 + \frac{2}{n}(\sum_{k=0}^{n-1} QS(k)).$$

Z toho dostáváme, že

$$nQS(n) = n(n-1) + 2\sum_{k=0}^{n-1} QS(k).$$

Přepíšeme ještě jednou tuto rovnici s $n+1$ místo n:

$$(n+1)QS(n+1) = (n+1)n + 2\sum_{k=0}^{n} QS(k).$$

Od této rovnice odečteme rovnici předchozí a po jednoduché úpravě získáme rekurentní vztah

$$QS(n+1) = \frac{2n}{n+1} + \frac{n+2}{n+1}QS(n).$$

Postupným dosazováním dostaneme řešení

$$QS(n) = \sum_{i=2}^{n} \frac{n+1}{i+1}\frac{2(i-1)}{i} \le 2(n+1)\big(\sum_{i=2}^{n}\frac{1}{i+1}\big) = 2(n+1)\big(\sum_{i=3}^{n+1}\frac{1}{i}\big) =$$
$$2(n+1)\big(\sum_{i=2}^{n+1}\frac{1}{i} - \frac{1}{2}\big) \le 2(n+1)\big((\int_{i=1}^{n+1}\frac{1}{x}dx) - \frac{1}{2}\big) =$$
$$2n\ln(n+1) + 2\ln(n+1) - n - 1.$$

Pro dostatečně velká n tedy platí

$$2n\ln(n+1) + 2\ln(n+1) - n \le 2n\ln n.$$

Porovnání třídicích algoritmů

Nyní porovnáme složitost algoritmů **HEAPSORT**, **MERGESORT**, **QUICKSORT**, **A-sort** (byl popsán v kapitole o $(a,b)-$stromech), **SELECTIONSORT** a **INSERTIONSORT**. Připomeňme si, že **SELECTIONSORT** třídí posloupnost tak, že jedním průchodem nalezne její nejmenší prvek, který vyřadí a vloží do výsledné posloupnosti (ve verzi, která třídí na místě, ho vymění s levým krajním prvkem pole). Tento proces pak opakuje se zbytkem původní posloupnosti. Tato idea byla základem algoritmu **HEAPSORT**. **INSERTIONSORT** třídí tak, že do již setříděného začátku posloupnosti vkládá další prvek,

který pomocí výměn zařadí na správné místo, a tento proces (začíná druhým prvkem zleva) opakuje.

QUICKSORT v nejhorším případě vyžaduje čas $\Theta(n^2)$, očekávaný čas je $9n \log n$, v nejhorším případě provádí $\frac{n^2}{2}$ porovnání, očekávaný počet porovnání je $1.44n \log n$. Potřebuje $n + \log n + konst$ paměti, používá přímý přístup k paměti a není adaptivní na předtříděné posloupnosti.
HEAPSORT v nejhorším případě vyžaduje čas $20n \log n$, očekávaný čas je $\leq 20n \log n$, v nejhorším i v očekávaném případě provádí $2n \log n$ porovnání. Potřebuje $n + konst$ paměti, používá přímý přístup k paměti a není adaptivní na předtříděné posloupnosti.
MERGESORT v nejhorším případě vyžaduje čas $12n \log n$, očekávaný čas je $\leq 12n \log n$, v nejhorším i v očekávaném případě provádí $n \log n$ porovnání (nejmenší možný počet). Potřebuje $2n + konst$ paměti, používá sekvenční přístup k paměti a má verzi, která je adaptivní na předtříděné posloupnosti s malým počtem běhů.
A-sort v nejhorším případě i v očekáném případě vyžaduje čas $O(n \log \frac{F}{n})$, kde F je počet inverzí ve vstupní posloupnosti, v nejhorším i v očekávaném případě provádí $O(n \log \frac{F}{n})$ porovnání. Potřebuje $5n + konst$ paměti, používá přímý přístup k paměti a je adaptivní na předtříděné posloupnosti s malým počtem inverzí.
SELECTIONSORT v nejhorším i v očekávaném případě vyžaduje čas $2n^2$, počet porovnání v nejhorším i v očekávaném případě je $\frac{n^2}{2}$. Potřebuje $n + konst$ paměti, používá přímý přístup k paměti a není adaptivní na předtříděné posloupnosti.
INSERTIONSORT v nejhorším i v očekávaném případě vyžaduje čas $O(n^2)$, počet porovnání v nejhorším případě je $\frac{n^2}{2}$, v očekávaném případě $\frac{n^2}{4}$. Potřebuje $n+konst$ paměti, používá sekvenční přístup k paměti a má verzi, která je adaptivní na předtříděné posloupnosti s malým počtem inverzí.

Prezentované výsledky byly spočítány pro model RAM (viz Mehlhorn 1984).

Očekávaný čas pro **HEAPSORT** je prakticky stejný jako jeho nejhorší čas. Byly navrženy verze, které optimalizují počet porovnání, ale většinou mají větší nároky na čas, a proto až na výjimky nejsou pro praktické použití vhodné. Situace pro **MERGESORT** je komplikovanější, hodně závisí na konkrétní verzi algoritmu. Algoritmus **MERGESORT** je nejvhodnější pro externí paměti se sekvenčním přístupem k datům, pro interní paměť kvůli velké prostorové náročnosti není doporučován (je např. dvojnásobná proti **HEAPSORTU** a téměř dvojnásobná proti **QUICKSORTU**). Také se hodí pro návrh paralelních algoritmů. Pro třídění krátkých posloupností je doporučováno místo **QUICKSORTU** pro posloupnosti délky ≤ 22 použít **SELECTIONSORT** a pro posloupnosti délky ≤ 15 **INSERTIONSORT.** To vede k návrhu optimalizovaného **QUICKSORTU**, který**,** když volá rekurzivně sám sebe na krátkou posloupnost, pak použije **SELECTIONSORT** nebo **INSERTIONSORT**. V algoritmu **A-sort** se doporučuje použít $(2, 3)$-strom. Poměr časů spotřebovaných algoritmy **QUICKSORT**, **MERGESORT** a **HEAPSORT** na klasických počítačích uvádí Mehlhorn (1984) jako $1 : 1.33 : 2.22$. To však nemusí být pravda pro současné procesory, paměti a operační systémy.

Slévání nestejně dlouhých posloupností

V algoritmu **MERGESORT** jsme použili frontu, která řídila proces slučování rostoucích posloupností. Tato metoda je uspokojující a dává optimální výsledek (ve smyslu časové

náročnosti), pokud posloupnosti ve frontě jsou stejně dlouhé. Pokud se ale jejich délky hodně liší, nedosáhneme tímto způsobem optimálního výsledku. Přitom různé verze tohoto problému se vyskytují v mnoha úlohách. Jednou z prvních úloh, kde jsme se s ním setkali, je konstrukce Huffmanova kódu – to je minimální redundantní kód, který byl nalezen v roce 1952. K optimálnímu řešení vede např. postup, který je kombinací 'mergeování' a optimalizace a používá metody dynamického programování. Nejprve formálně popíšeme abstraktní verzi tohoto problému.

Vstup: Množina rostoucích navzájem disjunktních posloupností.
Úkol: Pomocí operace **MERGE** co nejrychleji spojit všechny tyto posloupnosti do jediné rostoucí posloupnosti.

Předpokládejme, že máme postup, který z daných rostoucích posloupností vytvoří jedinou rostoucí posloupnost. Tento postup určuje úplný binární strom T, jehož listy jsou ohodnoceny vstupními posloupnostmi a každý vnitřní vrchol je ohodnocen posloupností, která je sloučením vstupních posloupností ohodnocujících listy v podstromu určeném tímto vrcholem. Tedy kořen je ohodnocen výstupní posloupností. Formálně pro každý vnitřní vrchol v platí:

> když v_1 a v_2 jsou synové v a $P(v)$ je posloupnost ohodnocující vrchol v, pak $P(v) =$**MERGE**$(P(v_1), P(v_2))$.

Označme $l(P)$ délku posloupnosti P. Pak součet časů, které v tomto procesu vyžaduje podprocedura **MERGE**, je $O(\sum\{l(P(v)) \mid v \text{ je vnitřní vrchol stromu } T\})$. Indukcí lehce dostaneme, že

$$\sum\{l(P(v)) \mid v \text{ vnitřní vrchol stromu } T\} = \sum_{\{t \text{ je list } T\}} d(t)l(P(t)),$$

kde $d(t)$ je hloubka listu t.

Když tedy T je úplný binární strom, jehož listy jsou ohodnoceny navzájem disjunktními rostoucími posloupnostmi, pak následující algoritmus **Slevani** spojí tyto posloupnosti do jediné rostoucí posloupnosti a procedury **MERGE** budou vyžadovat celkový čas

$$O(\sum_{\{t \text{ je list } T\}} d(t)l(P(t))).$$

Slevani$(T, \{P(l) \mid l \text{ je list } T\})$
while P(kořen T) není definováno **do**
  $v :=$ vrchol T takový, že $P(v)$ není definováno a
  pro oba syny v_1 a v_2 vrcholu v jsou $P(v_1)$ a $P(v_2)$ definovány
  $P(v) :=$**MERGE**$(P(v_1), P(v_2))$
enddo

Nyní můžeme přeformulovat původní problém:
Vstup: n čísel $x_1, x_2, \ldots, x_n$
Výstup: úplný binární strom T s n listy a bijekce ϕ z množiny $\{1, 2, \ldots, n\}$ do listů T

taková, že $\sum_{i=1}^{n} d(\phi(i))x_i$ je minimální (kde $d(\phi(i))$ je hloubka listu $\phi(i)$).
Řekneme, že dvojice (T, ϕ) je optimální strom vzhledem k $x_1, x_2, \ldots, x_n$.

V přeformulováné úloze už nepracujeme s posloupnostmi, ale jen s jejich délkami. To znamená, že když pro původní úlohu byly vstupem posloupnosti $P_1, P_2, \ldots, P_n$, pak pro přeformulovanou úlohu jsou vstupem jen délky $l(P_1), l(P_2), \ldots, l(P_n)$. Strom vytvořený pro přeformulovanou úlohu je použit v algoritmu **Slevani** tak, že posloupnost P_i ohodnocuje list, který byl v přeformulované úloze ohodnocen délkou $l(P_i)$, a hledaná posloupnost v původní úloze ohodnocuje kořen stromu.

Mějme množinu $\{x_i \mid i = 1, 2, \ldots, n\}$. Pro úplný binární strom T s n listy a bijekci ϕ z množiny $\{1, 2, \ldots, n\}$ do listů stromu T definujme

$$\mathrm{Cont}(T, \phi) = \sum_{i=1}^{n} d(\phi(i))x_i,$$

kde $d(\phi(i))$ je hloubka listu $\phi(i)$, tj. délka cesty z kořene do listu $\phi(i)$ pro $i = 1, 2, \ldots, n$. Chceme zkonstruovat úplný binární strom s n listy, který minimalizuje hodnotu Cont. K řešení použijeme následující algoritmus, který je upravenou verzí hladového algoritmu pro náš problém.

Optim$(x_1, x_2, \ldots x_n)$:
V je množina n jednoprvkových stromů
ϕ je bijekce mezi $\{1, 2, \ldots, n\}$ a množinou V
for every $v \in V$ **do** $c(v) := x_{\phi^{-1}(v)}$ **enddo**
while $|V| > 1$ **do**
  vezmi z V dva stromy v_1 a v_2 s nejmenším ohodnocením
  odstraň je z V
  vytvoř nový strom v spojením stromů v_1 a v_2
  $c(v) := c(v_1) + c(v_2)$, strom v vlož do V
enddo
Výstup: (T, ϕ), kde T je strom v množině V

Vytvoření nového stromu v spojením stromů v_1 a v_2 znamená vytvoření nového vrcholu, který bude kořenem stromu v a jehož synové budou kořeny stromů v_1 a v_2. To je analogické proceduře **spoj**.

Věta. *Pro danou posloupnost čísel $(x_1, x_2, \ldots, x_n)$ algoritmus* **Optim** *nalezne optimální strom pro množinu $x_1, x_2, \ldots, x_n$ a pokud je posloupnost $(x_1, x_2, \ldots, x_n)$ neklesající, pak vyžaduje čase $O(n)$.*

Důkaz. Důkaz má dvě části. V první dokážeme korektnost algoritmu a ve druhé popíšeme reprezentaci množiny V a vypočteme časovou složitost.

Nejprve připomeňme, že $\phi(i)$ je list T pro každé $i \in \{1, 2, \ldots, n\}$. Protože na začátku V obsahuje jen jednoprvkové stromy, tak tvrzení platí. Každý běh cyklu **while do** zmenší počet stromů V o jeden, ale nezmění množinu listů. Proto T je strom s n listy, ϕ je bijekce z $\{1, 2, \ldots, n\}$ do množiny listů T a algoritmus vždy končí. Dokážeme indukcí podle n, že zkonstruovaná dvojice (T, ϕ) je optimální strom vzhledem k $(x_1, x_2, \ldots, x_n)$.

Když $n = 2$, tvrzení zřejmě platí. Předpokládejme, že platí pro každou posloupnost čísel $(y_1, y_2, \ldots, y_{n-1})$, a nechť $x_1 \leq x_2 \leq \cdots \leq x_n$ je neklesající posloupnost čísel. Bez újmy na obecnosti můžeme předpokládat, že v prvním kroku algoritmus **Optim** zvolil stromy $\phi(1)$ a $\phi(2)$. Uvažujme množinu $(y_1, y_2, \ldots, y_{n-1})$, kde $y_i = x_{i+2}$ pro $i = 1, 2, \ldots, n-2$, $y_{n-1} = x_1 + x_2$. Nechť T' je strom získaný ze stromu T odstraněním listů $\phi(1)$ a $\phi(2)$ a nechť ψ je bijekce z množiny $\{1, 2, \ldots, n-1\}$ taková, že $\psi(i) = \phi(i+2)$ pro $i = 1, 2, \ldots, n-2$ a $\psi(n-1)$ je otec listu $\phi(1)$. Pak můžeme předpokládat, že algoritmus **Optim**$(y_1, y_2, \ldots, y_{n-1})$ zkonstruoval strom (T', ψ), a podle indukčního předpokladu je to optimální strom pro $(y_1, y_2, \ldots, y_{n-1})$. Nechť (U, θ) je optimální strom vzhledem k $(x_1, x_2, \ldots, x_n)$. Zvolme vnitřní vrchol u stromu U takový, že délka cesty z kořene do vrcholu u je největší mezi všemi vnitřními vrcholy stromu U. Nechť u_1 a u_2 jsou synové u, pak nutně u_1 a u_2 jsou listy stromu U. Nechť $i, j \in \{1, 2, \ldots, n\}$ takové, že $\theta(i) = u_1$, $\theta(j) = u_2$. Po eventuálním přejmenování můžeme předpokládat, že když $i, j \in \{1, 2\}$, pak $i = 1$ a $j = 2$. Definujme η z $\{1, 2, \ldots, n\}$ do listů U tak, že $\eta(1) = u_1$, $\eta(2) = u_2$, $\eta(i) = \theta(1)$, $\eta(j) = \theta(2)$ a $\eta(k) = \theta(k)$ pro všechna $k \in \{3, 4, \ldots, n\} \setminus \{i, j\}$. Pak η je bijekce a

$$\mathrm{Cont}(U, \eta) - \mathrm{Cont}(U, \theta) = (d(u_1) - d(\theta(1))(x_1 - x_i) + (d(u_2) - d(\theta(2))(x_2 - x_j).$$

Z volby u plyne, že $d(u_1) \geq d(\theta(1))$, $d(u_2) \geq d(\theta(2))$, $x_1 \leq x_i$ a $x_2 \leq x_j$. Odtud

$$(d(u_1) - d(\theta(1))(x_1 - x_i) + (d(u_2) - d(\theta(2))(x_2 - x_j) \leq 0$$

a protože (U, θ) je optimální strom pro $(x_1, x_2, \ldots, x_n)$, dostáváme, že (U, η) je také optimální strom pro $(x_1, x_2, \ldots, x_n)$. Odstraněním listů u_1 a u_2 ze stromu U dostaneme strom U'. Definujme τ z $\{1, 2, \ldots, n-1\}$ předpisem $\tau(i) = \eta(i+2)$ pro $i = 1, 2, \ldots, n-2$ a $\tau(n-1) = u$. Pak τ je bijekce z $\{1, 2, \ldots, n-1\}$ do množiny listů U' a protože (T', ψ) je optimální strom pro $(y_1, y_2, \ldots, y_{n-1})$, platí, že

$$\mathrm{Cont}(T', \psi) \leq \mathrm{Cont}(U', \tau).$$

Protože

$$\begin{aligned} \mathrm{Cont}(T, \phi) &= \mathrm{Cont}(T, \psi) + x_1 + x_2 \ , \\ \mathrm{Cont}(U, \eta) &= \mathrm{Cont}(U', \tau) + x_1 + x_2 \end{aligned}$$

pak závěr je, že (T, ϕ) je optimální strom pro $(x_1, x_2, \ldots, x_n)$.

Předpokládejme opět, že $x_1 \leq x_2 \leq \cdots \leq x_n$ a že v daném okamžiku jsou $v_1, v_2, \ldots, v_k$ postupně vytvořené víceprvkové stromy (tj. strom v_i byl vytvořen před stromem v_j, když $i < j$). V tomto okamžiku je množina V sjednocením množiny $\{v_1, v_2, \ldots, v_k\}$ a množiny jednoprvkových stromů, které nebyly ještě zpracovány. Nyní vytvoříme strom w spojením stromů t_1 a t_2 s nejmenším ohodnocením. Z popisu algoritmu plyne, že když strom v_i pro $i = 1, 2, \ldots, k$ vznikl spojením stromů u_1 a u_2, pak $\max\{c(u_1), c(u_2)\} \leq \min\{c(t_1), c(t_2)\}$, a proto $c(w) \geq c(v_i)$ pro každé $i = 1, 2, \ldots, k$. Pak indukcí okamžitě dostáváme, že $c(v_1) \leq c(v_2) \leq \cdots \leq c(v_k)$. Tedy stačí, abychom měli rostoucí posloupnost listů a v ní ukazatel na nejmenší list, který je ještě nezpracovaným jednoprvkovým stromem (tj.

před ukazatelem jsou listy, které už nejsou stromy v množině V, za ukazatelem jsou listy, které jsou ještě jednoprvkové stromy v množině V) a frontu víceprvkových stromů (z níž stromy ke zpracování odebíráme zpředu a nově vytvořené ukládáme na konec). Udržovat tyto struktury vyžaduje čas $O(1)$ stejně jako nalezení dvou stromů s nejmenším ohodnocením. Můžeme tedy shrnout, že algoritmus **Optim** konstruuje optimální stromy v čase $O(n)$, kde n je počet zadaných čísel x_i. □

Pro aplikaci na naši původní úlohu je třeba ještě setřídit vstupní posloupnost délek pro přeformulovanou úlohu. Tato posloupnost je tvořena přirozenými čísly a k jejímu setřídění můžeme použít algoritmus **BUCKETSORT** (bude popsán dále v textu), který vyžaduje čas $O(n+m)$, kde n je počet posloupností a m je maximální délka posloupnosti.

Věta. *Uvedený algoritmus množinu disjunktních rostoucích posloupností $P_1, P_2, \ldots, P_n$ o délkách $l(P_1), l(P_2), \ldots, l(P_n)$ spojí do jediné rostoucí posloupnosti v čase $O(\sum_{i=1}^{n} l(P_i))$.*

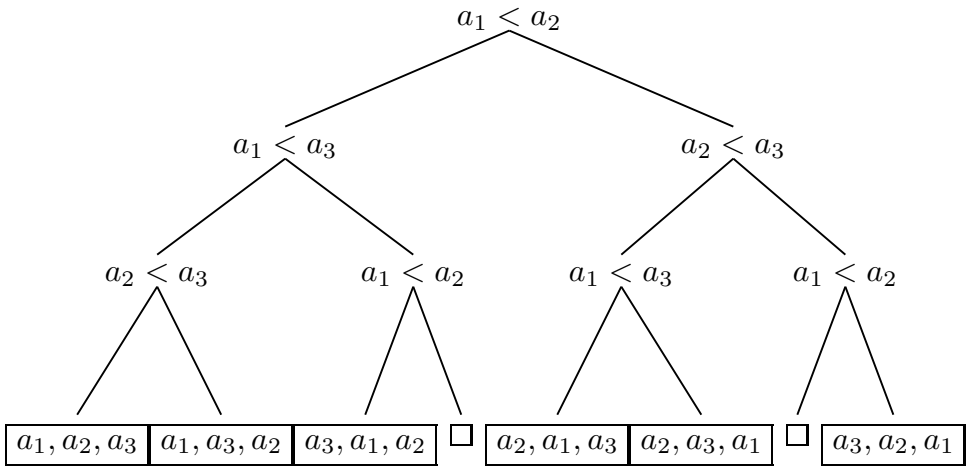
VIII. Rozhodovací stromy

Většina obecných třídicích algoritmů používá jedinou primitivní operaci mezi prvky vstupní posloupnosti, a to jejich vzájemné porovnání. To znamená, že práci takového algoritmu lze popsat binárním stromem, jehož vnitřní vrcholy jsou ohodnoceny porovnáními dvojic prvků vstupní posloupnosti (např. $a_i < a_j$). Bez újmy na obecnosti předpokládejme, že vstupní posloupnost je permutace π množiny $\{1, 2, \ldots, n\}$. Tato permutace prochází stromem takto:

> Začíná v kořeni stromu. Když je ve vnitřním vrcholu v ohodnoceném porovnáním $a_i \leq a_j$, pak když $\pi(i) < \pi(j)$, pokračuje v levém synu vrcholu v, a když $\pi(j) < \pi(i)$, pokračuje v pravém synu vrcholu v. Proces třídění končí, když se dostane do listu.

Aby byl algoritmus korektní, musí platit, že dvě různé permutace skončí v různých listech. Tedy strom popisující korektní algoritmus pro setřídění n-prvkovýh posloupností musí mít alespoň $n!$ listů. Délka cesty z kořene do listu, kde skončila permutace π, reprezentuje počet porovnání, které potřebuje daný algoritmus k setřídění dané posloupnosti π. Protože porovnání vyžaduje alespoň jednotku času, dostáváme tím i dolní odhad na čas potřebný k setřídění této posloupnosti algoritmem odpovídajícím danému stromu. Dolní odhad počtu porovnání i času pro daný algoritmus a všechny n-prvkové posloupnosti je pak délka nejdelší cesty z kořene do listu v odpovídajícím stromu. To nám umožňuje získat obecně platný dolní odhad času potřebného k setřídění $n-$prvkové posloupnosti, kterým je minimum přes všechny binární stromy s alespoň $n!$ listy z jejich maximálních délek cest z kořene do listu. Korektnost těchto úvah plyne z pozorování, že když porovnání je jediná primitivní operace, pak algoritmus není závislý na konkrétních prvcích vstupní posloupnosti, ale jen na jejich vzájemném vztahu. Proto stačí uvažovat pouze permutace n-prvkové množiny, protože zachycují všechny možné vztahy v n-prvkové posloupnosti. Dále je třeba si uvědomit, že vztah mezi stromem pro n-prvkové posloupnosti a stromem pro $(n+1)$-prvkové posloupnosti je dán konkrétním algoritmem a nedá se popsat obecně.

V nevhodném algoritmu se může stát, že v některém listu neskončí žádná permutace. To nastane, když strom pro n-prvkové posloupnosti má více než $n!$ listů, nebo, jinak řečeno, když porovnání dvou stejných prvků se na nějaké cestě vyskytne alespoň dvakrát.



Obr. 1

Následující obrázek ilustruje naše úvahy na **SELECTIONSORTU** pro 3-prvkové posloupnosti. Listy jsou ohodnoceny permutacemi vstupní množiny $\{a_1, a_2, a_3\}$, které v nich skončí, nebo jsou prázdné.

Definice. Mějme třídicí algoritmus **A**, který jako jedinou primitivní operaci s prvky vstupní posloupnosti používá jejich porovnání. Řekneme, že binární strom T, jehož vnitřní vrcholy jsou ohodnoceny porovnáními $a_i \leq a_j$ pro $i, j = 1, 2, \ldots, n$, $i \neq j$, je rozhodovacím stromem algoritmu **A** pro n-prvkové posloupnosti, když pro každou permutaci π n-prvkové množiny platí

> posloupnost porovnání při třídění permutace π algoritmem **A** je stejná jako posloupnost porovnání při průchodu permutace π stromem T.

Pak korektnost algoritmu zajišťuje, že dvě různé permutace množiny $\{1, 2, \ldots, n\}$ skončí v různých listech stromu T a dolním odhadem pro čas algoritmu **A** v nejhorším případě je délka nejdelší cesty z kořene do listu. Při rovnoměrném rozdělení vstupních posloupností je očekávaný čas algoritmu **A** roven průměrné délce cesty z kořene do listu.

Definujme
$S(n)$ jako minimum přes všechny stromy T s alespoň $n!$ listy z délek nejdelších cest z kořene do listu v T,
$A(n)$ jako minimum přes všechny stromy T s alespoň $n!$ listy z průměrných délek cest z kořene do listu v T.
Naším cílem je spočítat dolní odhady těchto veličin.

Když nejdelší cesta z kořene do listu v binárním stromě T má délku k, pak T má nejvýše 2^k listů. Proto $n! \leq 2^{S(n)}$. Odtud plyne, že $S(n) \geq \log_2 n!$. Připomeňme si Stirlingův vzorec pro faktoriál:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + O\left(\frac{1}{n^2}\right)\right).$$

Protože pro $n \geq 1$ je $\frac{1}{12n}, \frac{1}{n^2} \geq 0$, můžeme předpokládat, že $(1 + \frac{1}{12n} + O(\frac{1}{n^2})) \geq 1$ pro

všechna $n \geq 1$. Po zlogaritmování vzorce dostáváme

$$\log_2 n! \geq \frac{1}{2}\log_2 n + n(\log_2 n - \log_2 e) + \log_2 \sqrt{2\pi} \geq (n + \frac{1}{2})\log_2 n - n\log_2 e.$$

Protože

$$e^1 = e = 2^{\log_2 e} = (e^{\ln 2})^{\log_2 e} = e^{\ln 2 \log_2 e},$$

platí, že $\frac{1}{\ln 2} = \log_2 e$, a tedy

$$S(n) \geq \log_2 n! \geq (n + \frac{1}{2})\log_2 n - \frac{n}{\ln 2}.$$

Dále pro binární strom T označme $B(T)$ součet všech délek cest z kořene do listů a položme

$$B(k) = \min\{B(T) \mid T \text{ je binární strom s } k \text{ listy}\}.$$

Když ukážeme, že $B(k) \geq k\log_2 k$, pak bude

$$A(n) \geq \frac{B(n!)}{n!} \geq \frac{n!\log_2 n!}{n!} = \log_2 n! \geq (n + \frac{1}{2})\log_2 n - \frac{n}{\ln 2}.$$

Dokažme tedy, že $B(T) \geq k\log_2 k$ pro každý binární strom T s k listy. Když ve stromě T vynecháme každý vrchol, který má jen jednoho syna, a tohoto syna spojíme s jeho předchůdcem, dostaneme úplný binární strom T' s k listy takový, že $B(T') \leq B(T)$. Proto se stačí omezit na úplné binární stromy. Když T je úplný binární strom s jedním listem, pak $B(T) = 0 = 1\log_2 1$, když T je úplný binární strom se dvěma listy, pak $B(T) = 2 = 2\log_2 2$. Tedy platí $B(1) \geq 1\log_2 1$ a $B(2) \geq 2\log_2 2$. Předpokládejme, že $B(i) \geq i\log_2 i$ pro $i < k$, a nechť T je úplný binární strom s k listy. Nechť T_1 a T_2 jsou podstromy určené syny kořene a nechť T_i má k_i listů, kde $i = 1, 2$. Pak $1 \leq k_1, k_2$ a $k_1 + k_2 = k$, tedy $k_1, k_2 < k$ a podle indukčního předpokladu $B(k_i) \geq k_i\log_2 k_i$. Odtud

$$B(T) = k_1 + B(T_1) + k_2 + B(T_2) \geq k + B(k_1) + B(k_2) \geq k + k_1\log_2 k_1 + k_2\log_2 k_2.$$

Tedy stačí ukázat, že

$$k + k_1\log_2 k_1 + k_2\log_2 k_2 \geq k\log_2 k$$

pro všechna $k_1, k_2 > 0$ taková, že $k = k_1 + k_2$. To je ekvivalentní s tvrzením, že pro $k > 0$ platí

$$f(x) = x\log_2 x + (k - x)\log_2(k - x) + k - k\log_2 k \geq 0,$$

kde $x \in (0, k)$. Abychom to dokázali, všimněme si, že $f(\frac{k}{2}) = 0$ a počítejme derivaci f.

$$f'(x) = \log_2 x + \log_2 e - \log_2(k - x) - \log_2 e = \log_2 \frac{x}{k - x}.$$

Nyní když $x \in (0, \frac{k}{2})$, pak $f'(x) < 0$ a f je na tomto intervalu klesající, když $x \in (\frac{k}{2}, k)$, pak $f'(x) > 0$ a f je na tomto intervalu rostoucí. Odtud plyne, že $f(x) \geq 0$ pro $x \in (0, k)$. Tím jsme dokázali, že $A(n) \geq (n + \frac{1}{2})\log_2 n - \frac{n}{\ln 2}$. Shrneme naše výsledky.

Věta. *Každý třídicí algoritmus, jehož jedinou primitivní operací s prvky vstupní posloupnosti je porovnání, vyžaduje v nejhorším i v očekávaném případě alespoň* $cn \log n$ *času pro nějakou konstantu* $c > 0$. *V nejhorším případě použije alespoň* $\lceil (n + \frac{1}{2}) \log_2 n - \frac{n}{\ln 2} \rceil$ *porovnání a očekávaný počet porovnání při rovnoměrném rozdělení vstupních posloupností je alespoň* $(n + \frac{1}{2}) \log_2 n - \frac{n}{\ln 2}$.

Tato věta platí i pro širší třídu primitivních operací, proto v ní lze oslabit předpklady. Dolní odhad (v nejhorším i průměrném případě) bude platit i za předpokladu, že třídicí algoritmus nepoužívá nepřímé adresování a celočíselné dělení. (Na druhé straně následující klasický algoritmus **BUCKETSORT** ukazuje, že předpoklady ve větě nelze zcela vynechat.) Tato metoda pro nalezení dolního odhadu se používá i pro vyčíslování algebraických funkcí a při algoritmickém řešení geometrických úloh.

IX. Přihrádkové třídění

V následujících algoritmech předpokládáme, že Q_i jsou spojové seznamy, nový prvek se vkládá na konec seznamu a konkatenace seznamů závisí na jejich pořadí. V seznamech máme okamžitý přístup k prvnímu a poslednímu prvku (pomocí ukazatelů na tyto prvky). Algoritmus **BUCKETSORT** třídí posloupnost přirozených čísel $a_1, a_2, \ldots, a_n$ z intervalu $< 0, m >$.

BUCKETSORT$(a_1, a_2, \ldots, a_n, m)$:
for every $i = 0, 1, \ldots, m$ **do** $Q_i = \emptyset$ **enddo**
for every $i = 1, 2, \ldots, n$ **do**
  a_i vlož na konec seznamu Q_{a_i}
enddo
$i := 0$, $P := \emptyset$
while $i \leq m$ **do**
  P :=konkatenace P a Q_i, $i := i + 1$
enddo
Výstup: P je neklesající posloupnost prvků $a_1, a_2, \ldots, a_n$

Algoritmus nevyžaduje, aby prvky ve vstupní posloupnosti byly různé. Ve výstupní posloupnosti se daný prvek opakuje tolikrát, kolikrát se opakoval ve vstupní posloupnosti, se zachováním pořadí (tj. třídění je stabilní). Konkatenace dvou seznamů a vložení prvku do seznamu vyžadují čas $O(1)$. Proto první a třetí cyklus vyžadují čas $O(m)$ a druhý cyklus čas $O(n)$. Celkem algoritmus vyžaduje $O(n + m)$ času a paměti. Zřejmě když $m = O(n)$, tak pro tento algoritmus neplatí tvrzení věty z předchozího odstavce. Důvodem je, že nejsou splněny předpoklady, protože druhý cyklus používá nepřímé adresování.

Nyní uvedeme dvě sofistikovanější verze tohoto algoritmu. V první předpokládáme, že $a_1, a_2, \ldots, a_n$ je posloupnost navzájem různých reálných čísel z intervalu $< 0, 1 >$ a α je pevně zvolené kladné reálné číslo.

HYBRIDSORT$(a_1, a_2, \ldots, a_n)$:
$k := \alpha n$
for every $i = 0, 1, \ldots, k$ **do** $Q_i = \emptyset$ **enddo**

for every $i = 1, 2, \ldots, n$ **do**
  a_i vlož na konec seznamu $Q_{\lceil ka_i \rceil}$
enddo
$i := 0$, $P := \emptyset$
while $i \leq k$ **do**
  HEAPSORT(Q_i) P :=konkatenace P a Q_i, $i := i + 1$
enddo
Výstup: P je rostoucí posloupnost prvků $a_1, a_2, \ldots, a_n$

Věta. *Algoritmus* **HYBRIDSORT** *setřídí posloupnost reálných čísel z intervalu* $< 0, 1 >$ *v nejhorším případě v čase* $O(n \log n)$. *Když prvky* a_i *mají rovnoměrné rozložení a jsou na sobě nezávislé, pak očekávaný čas je* $O(n)$.

Důkaz. První dva cykly v algoritmu vyžadují čas $O(n)$, i-tý běh třetího cyklu vyžaduje nejvýše čas $O(1 + |Q_i| \log |Q_i|)$. Proto čas celého třetího cyklu je

$$O(\sum_{i=0}^{k}(1 + |Q_i| \log |Q_i|) = O(\sum_{i=0}^{k}(1 + |Q_i| \log n) = O(k + (\sum_{i=0}^{k} |Q_i|) \log n) = O(n \log n)$$

a celkový čas **HYBRIDSORTU** v nejhorším případě je nejvýše $O(n \log n)$.

Nyní odhadneme očekávaný čas. Položme $X_i = |Q_i|$. Pak X_i je náhodná proměnná a protože pravděpodobnost, že $x \in Q_i$, je $\frac{1}{k}$, dostáváme, že

$$\text{Prob}(X_i = q) = \binom{n}{q}(\frac{1}{k})^q(1 - \frac{1}{k})^{n-q}.$$

Očekávaný čas vyžadovaný třetím cyklem se pak rovná

$$E(\sum_{i=0}^{k} 1 + X_i \log X_i) \leq k + k \sum_{q=2}^{n} q^2 \binom{n}{q}(\frac{1}{k})^q(1 - \frac{1}{k})^{n-q} = k + k(\frac{n(n-1)}{k^2} + \frac{n}{k}) = O(n),$$

protože $k = \alpha n$ a

$$q^2 \binom{n}{q} = (q(q-1) + q)\binom{n}{q} = n(n-1)\binom{n-2}{q-2} + n\binom{n-1}{q-1}.$$

(Jedná se vlastně o známý výpočet 2. momentu binomického rozdělení). □

Poznámka: V důkazu jsme použili odhad $q \log q \leq q^2$ a důsledkem toho je, že jsme dokázali, že očekávaná složitost **HYBRIDSORTU** zůstane lineární, i kdybychom v něm místo **HEAPSORTU** použili nějaký třídicí algoritmus s kvadratickou složitostí, např. **INSERTIONSORT**.

Nyní použijeme modifikaci **BUCKETSORTU** pro třídění slov. Máme totálně uspořádanou abecedu a chceme lexikograficky setřídit slova $a_1, a_2, \ldots, a_n$ nad touto abecedou. Připomeňme, že když $a = x_1 x_2 \ldots x_n$ a $b = y_1 y_2 \ldots y_m$ jsou dvě slova nad totálně uspořádanou abecedou Σ, pak $a < b$ v lexikografickém uspořádání, právě když existuje $i = 0, 1, \ldots, \min\{n, m\}$

takové, že $x_j = y_j$ pro každé $j = 1, 2, \ldots, i$ a buď $n = i < m$ nebo $i < \min\{n, m\}$ a $x_{i+1} < y_{i+1}$. Předpokládejme, že $a_i = a_i^1 a_i^2 \ldots a_i^{l(i)}$, kde $a_i^j \in \Sigma$ a $l(i)$ je délka i-tého slova a_i.

WORDSORT$(a_1, a_2, \ldots, a_n)$:
for every $i = 1, 2, \ldots, n$ **do** $l(i)$:=délka slova a_i **enddo**
$l = \max\{l(i) \mid i = 1, 2, \ldots, n\}$
for every $i = 1, 2, \ldots, l$ **do** $L_i = \emptyset$ **enddo**
for every $i = 1, 2, \ldots, n$ **do**
  a_i vlož do $L_{l(i)}$
enddo
Komentář: Pro každé i obsahuje L_i všechna slova z množiny $\{a_1, a_2, \ldots, a_n\}$ délky i.
$P := \{(j, a_i^j) \mid 1 \le i \le n,\ 1 \le j \le l(i)\}$
P_1 :=**BUCKETSORT**(P) podle druhé komponenty
P_2 :=**BUCKETSORT**(P_1) podle první komponenty
for every $i = 1, 2, \ldots, l$ **do** $S_i = \emptyset$ **enddo**
(i, x) :=první prvek P_2
while $(i, x) \neq NIL$ **do**
  (i, x) vlož do S_i
  while(i, x) =následník (i, x) v P_2 **do**
    (i, x) :=následník (i, x) v P_2
  enddo
  (i, x) :=následník (i, x) v P_2
enddo
Komentář: V S_i jsou všechny dvojice (i, x) takové, že x je i-tým písmenem některého vstupního slova a když $x < y$, pak (i, x) je před (i, y).
for every $s \in \Sigma$ **do** $T_s := \emptyset$ **enddo**
$T := \emptyset$, $i := l$
while $i > 0$ **do**
  $T :=$ konkatenace L_i a T, a :=první slovo v T
  while $a \neq NIL$ **do**
    $s := i$-té písmeno a, vlož a do T_s
    a :=následník a v T
  enddo
  (i, x) :=první prvek v S_i, $T := \emptyset$
  while $(i, x) \neq NIL$ **do**
    $T :=$ konkatenace T a T_x, $T_x := \emptyset$
    (i, x) :=následník (i, x) v S_i
  enddo
  $i := i - 1$
enddo
Výstup: T je setříděná posloupnost slov $a_1, a_2, \ldots, a_n$

Uvažujme jeden běh posledního cyklu algortimu pro určité i. Po jeho skončení jsou v T všechna slova z množiny $a_1, a_2, \ldots, a_n$, která mají délku alespoň i, a když slovo a_r je před a_q v seznamu T, pak existuje $j = i-1, i, \ldots, l$ takové, že $a_r^k = a_q^k$ pro každé $k = i, i+1, \ldots, j$

a buď $l(r) = j \leq l(q)$ nebo $j < \min\{l(r), l(q)\}$ a $a_r^{j+1} < a_q^{j+1}$. To plyne z vlastností algoritmu **BUCKETSORT** indukcí podle i. Jediný a hlavní rozdíl proti **BUCKETSORTU** je, že neprocházíme všechny přihrádky T_x, ale pouze neprázdné. To nám zajišťuje množina S_i (viz Komentář).

Označme $L = \sum_{i=1}^{n} l(i)$ a připomeňme, že $l = \max\{l(i) \mid i = 1, 2, \ldots, n\}$. Pak první cyklus (výpočet délek slov) vyžaduje čas $O(L)$. Druhý cyklus (inicializace seznamů L_i) vyžaduje čas $O(l) = O(L)$ a třetí cyklus (zařazení slov do L_i podle délek) čas $O(n) = O(L)$. Vytvoření seznamu P vyžaduje čas $O(L)$ a jeho setřídění podle obou komponent čas $O(L+l) = O(L)$, protože P i P_1 mají nejvýše L prvků. Další cyklus (založení seznamů S_i) vyžaduje čas $O(l)$ a následující cyklus vytvářející seznamy S_i čas $O(L)$. Cyklus zakládající seznamy T_x vyžaduje čas $O(|\Sigma|)$. Běhy dalšího cyklu jsou indexovány $i = 1, 2, \ldots, l$. Pro každé i označme m_i počet slov z množiny $\{a_1, a_2, \ldots, a_n\}$, která mají délku alespoň i. Pak $L = \sum_{i=1}^{l} m_i$ a první vnitřní cyklus v i-tém běhu vnějšího cyklu vyžaduje čas $O(m_i)$ a druhý vnitřní cyklus čas $O(|S_i|) = O(m_i)$. Tedy celkový čas algoritmu je $O(L+m)$, kde $m = |\Sigma|$ a L je součet délek všech slov z množiny $a_1, a_2, \ldots, a_n$.

X. Pořádkové statistiky

Na závěr popíšeme dva algoritmy pro hledání k-tého nejmenšího prvku v dané podmnožině totálně uspořádaného univerza. První z nich využívá stejný princip jako **QUICKSORT**. Nejprve zadáme přesné znění naší úlohy (úloha i algoritmy se dají snadno přeformulovat pro případ, kdy hledáme $k-$tý největší prvek).

Pracujeme s totálně uspořádaným univerzem U.
Vstup: množina prvků $M = \{a_1, a_2, \ldots, a_n\} \subseteq U$ a číslo i takové, že $1 \leq i \leq n$.
Výstup: prvek a_k takový, že $|\{j \mid 1 \leq j \leq n,\, a_j \leq a_k\}| = i$.
Když $i = \frac{n}{2}$, pak a_k se nazývá medián.

FIND$(M = (a_1, a_2, \ldots, a_n), i)$:
zvol $a \in M$
$M_1 := \{b \in M \mid b < a\}$, $M_2 := \{b \in M \mid b > a\}$
if $|M_1| > i - 1$ **then**
  FIND(M_1, i)
else
  if $|M_1| < i - 1$ **then**
    FIND$(M_2, i - |M_1| - 1)$
  else
    Výstup: a je hledaný prvek
  endif
endif

Důkaz korektnosti algoritmu je založen na následujícím jednoduchém pozorování: mějme množinu M a prvek x a položme $M_1 = \{m \in M \mid m < x\}$. Když $k \leq |M_1|$, pak k-tý nejmenší prvek v M_1 je stejný jako k-tý nejmenší prvek v M. Když $k > |M_1|$, pak $(k - |M_1|)$-tý nejmenší prvek v $M \setminus M_1$ je k-tý nejmenší prvek v M. Zbývá vyšetřit složitost.

V nejhorším případě voláme **FIND** n-krát a jedno volání vyžaduje čas $O(|M|)$. Tedy časová složitost algoritmu **FIND** v nejhorším případě je $O(n^2)$. Dobré volby prvku a mohou algoritmus značně zrychlit. V tomto případě platí stejná diskuse jako pro **QUICKSORT**. Spočítáme očekávaný čas za předpokladu, že prvek a byl vybrán náhodně. Pak pravděpodobnost, že je k-tým nejmenším prvkem, je $\frac{1}{n}$, kde $n = |M|$. Označme $T(n, i)$ očekávaný čas algoritmu **FIND** pro nalezení i-tého nejmenšího prvku v n-prvkové množině M. Platí

$$T(n,i) = n + \frac{1}{n}(\sum_{k=1}^{i-1} T(n-k, i-k) + \sum_{k=i+1}^{n} T(k,i)),$$

protože procedura **FIND** bez rekurzivního volání sebe sama vyžaduje čas $O(n)$. Předpokládejme, že $T(m, i) \leq 4m$ pro každé $m < n$ a každé i takové, že $1 \leq i \leq m$. Pak

$$T(n,i) = n + \frac{1}{n}(\sum_{k=1}^{i-1} T(n-k, i-k) + \sum_{k=i+1}^{n} T(k,i)) \leq n + \frac{1}{n}(\sum_{k=1}^{i-1} 4(n-k) + \sum_{k=i+1}^{n} 4k) =$$
$$n + \frac{4}{n}(\frac{(2n-i)(i-1)}{2} + \frac{(n+i+1)(n-i)}{2}) = n + \frac{4}{n}(\frac{n^2 + 2ni - n - 2i^2}{2}).$$

Výraz v čitateli zlomku nabývá svého maxima pro $i = \frac{n}{2}$ a jeho maximalní hodnota je $\frac{3}{2}n^2 - n = \frac{3n^2-2n}{2}$. Tedy

$$T(n,i) \leq n + \frac{4}{n}(\frac{3n^2 - 2n}{4}) = n + 3n - 2 = 4n - 2 < 4n.$$

Protože tento odhad platí také pro $n = 1$ a $n = 2$, dokázali jsme indukcí, že $T(n, i) \leq 4n$ pro všechna n a všechna i taková, že $1 \leq i \leq n$. Platí tedy

Věta. *Algoritmus* **FIND** *nalezne i-tý nejmenší prvek v n prvkové totálně uspořádané množině a v nejhorším případě vyžaduje čas $O(n^2)$. Když se pivot volí náhodně nebo když všechny vstupní množiny mají stejnou pravděpodobnost, pak očekávaný čas je $O(n)$.*

Pro velmi malá i nebo pro i velmi blízká n pracuje rychleji přímý přirozený algoritmus (udržuje si posloupnost i nejmenších nebo $n - i$ největších prvků a k ní přidává další tak, že ten prvek, který překročil danou hranici, je zapomenut). Tento algoritmus však není efektivní pro obecná i.

Následující algoritmus nalezne i-tý nejmenší prvek v lineárním čase i v nejhorším případě. Vstupem je opět podmnožina M totálně uspořádaného univerza U a přirozené číslo i takové, že $1 \leq i \leq |M|$.

SELECT(M, i):
$n := |M|$
if $n \leq 100$ **then**
  setřiď množinu M, $m :=$ i-tý nejmenší prvek M
else
  rozděl M do navzájem disjunktních pětiprvkových podmnožin $A_1, A_2, \ldots, A_{\lceil \frac{n}{5} \rceil}$
  (poslední z podmnožin může mít méně než 5 prvků).

for every $j = 1, 2, \ldots, \lceil \frac{n}{5} \rceil$ **do**
najdi medián m_j množiny A_j
enddo
$\bar{m}$:=**SELECT**$(\{m_j \mid j = 1, 2, \ldots, \lceil \frac{n}{5} \rceil\}, \lceil \frac{n}{10} \rceil)$
$M_1 := \{m \in M \mid m < \bar{m}\}$, $M_2 := \{m \in M \mid \bar{m} < m\}$
if $|M_1| > i - 1$ **then**
m :=**SELECT**(M_1, i)
else
if $|M_1| < i - 1$ **then**
m :=**SELECT**$(M_2, i - |M_1| - 1)$
else
$m := \bar{m}$
endif
endif
Výstup: m
endif

Důkaz korektnosti algoritmu je stejný jako u algoritmu **FIND**. Zbývá vyšetřit složitost. Nejprve dokážeme následující lemma.

Lemma. *Když* $n \geq 100$, *pak* $|M_1|, |M_2| \leq \frac{8n}{11}$.

Důkaz. Pro $j \leq \lfloor \frac{n}{5} \rfloor$ platí, že když $m_j < \bar{m}$, pak $|A_j \cap M_1| \geq 3$, když $m_j > \bar{m}$, pak $|A_j \cap M_2| \geq 3$, když $m_j = \bar{m}$, pak $|A_j \cap M_1| = |A_j \cap M_2| = 2$. Protože $|\{j = 0, 1, \ldots, \lfloor \frac{n}{5} \rfloor \mid m_j < \bar{m}\}|, |\{j = 0, 1, \ldots, \lfloor \frac{n}{5} \rfloor \mid m_j > \bar{m}\}| \geq \lfloor \frac{n}{10} \rfloor$, dostáváme, že $|M_1|, |M_2| \geq \lfloor \frac{3n}{10} \rfloor - 1$. Dále platí $M_1 \cap M_2 = \emptyset$, $M_1 \cup M_2 = M \setminus \{\bar{m}\}$ a protože $\frac{8n}{11} + \lfloor \frac{3n}{10} \rfloor - 1 \geq \frac{113n}{110} - 2 \geq n$ když $n > 100$, dostáváme požadovaný odhad. □

Maximální čas vyžadovaný algoritmem **SELECT**(M, i) pro $|M| = n$ označme $T(n)$. Když $n \leq 100$, pak zřejmě existuje konstanta a taková, že $T(n) \leq an$. Když $n > 100$, pak $\lceil \frac{n}{5} \rceil \leq \frac{21n}{100}$, a protože **SELECT**$(M, i)$ pro $|M| > 100$ bez rekurentních volání vyžaduje čas $O(|M|)$, platí, že $T(n) \leq T(\frac{21n}{100}) + T(\frac{8n}{11}) + bn$ pro nějakou konstantu b. Zvolme $c \geq \max\{a, \frac{1100b}{69}\}$. Ukážeme, že $T(n) \leq cn$ pro všechna n. Když $n \leq 100$, tak tvrzení zřejmě platí, protože $a \leq c$. Když $n > 100$, pak $\lceil \frac{21n}{100} \rceil, \lceil \frac{8n}{11} \rceil < n$, a protože z volby c plyne $b \leq \frac{69}{1100}c$, dostáváme

$$T(n) \leq c\frac{21n}{100} + c\frac{8n}{11} + bn = (\frac{1031c}{1100} + b)n \leq cn.$$

Tedy

Věta. *Algoritmus* **SELECT** *nalezne i-tý nejmenší prvek v lineárním čase.*

Algoritmus **FIND** je ve velké většině případů rychlejší než algoritmus **SELECT**, proto je v praxi doporučován, i když existují případy (velmi řídké), kdy potřebuje kvadratický čas. Je známo, že medián n-prvkové množiny lze nalézt s méně než $3n$ porovnáními a že každý algoritmus hledající medián a používající porovnání jako jedinou primitivní operaci mezi prvky množiny vyžaduje více než $2n$ porovnání.

Historický přehled

Algoritmus **HEAPSORT** navrhl v roce 1964 Williams a vylepšil Floyd (rovněž 1964). Návrh na použití d-regulárních hald je folklor stejně tak jako algoritmus **MERGESORT**. Algoritmy **QUICKSORT** a **FIND** zavedl Hoare (1962). Analýza operace **MERGE** a hledání optimálního stromu pochází od Huffmana (1952) a lineární implementaci algoritmu navrhl van Leeuwen (1976). Analýza rozhodovacích stromů je folklor. Algoritmus **HYBRIDSORT** navrhli Meijer a Akl (1980), vylepšená verze **BUCKETSORTU** (nazvaná **WORDSORT**) pochází od Aho, Hopcrofta a Ullmana (1974). Algoritmus **SELECT** byl navržen Blumem, Floydem, Prattem, Rivestem a Tarjanem (1972).