Kukaččí hašování

Nyní popíšeme kukaččí hašování, to je alternativní metoda k perfektnímu hašování. Autoři položili důraz na rychlou realizaci operace **MEMBER** i v nejhorším případě a uvolnili požadavky na operaci **INSERT**. Při použití perfektního hašování je problém s operací **INSERT**. To řešila dynamická verze perfektního hašování, kde operace **INSERT** a **DELETE** vyžadovaly konstantní očekávaný amortizovaný čas. Kukaččí hašování řeší tento problém jinou metodou.
Kukaččí hašování pracuje s dvěma tabulkami T_0 a T_1 o velikosti m a s dvěma různými hašovacími funkcemi h_0 a h_1. Řekneme, že množina S je reprezentována funkcemi h_0 a h_1, když

$$S = \{s \mid s \text{ je uložené na nějakém řádku tabulky } T_0 \text{ nebo } T_1\}$$

a pro každé $s \in S$ platí, že s je uložené buď v T_0 na řádku $h_0(s)$ nebo v T_1 na řádku $h_1(s)$, ale nikoliv na obou řádcích. Pak operace **MEMBER**(x) se provede jednoduše, prohlédneme řádek $h_0(x)$ v tabulce T_0 a řádek $h_1(x)$ v tabulce T_1 a $x \in S$, právě když jsme na některém z těchto řádků prvek x našli.
Operace **DELETE**(x) je také jednoduchá. Prohlédneme, zda řádek $h_0(x)$ v tabulce T_0 nebo řádek $h_1(x)$ v tabulce T_1 neobsahuje x. Pokud ano, tak ho z této tabulky odstraníme a skončíme, pokud ne, tak hned končíme.

Problém je, kdy existuje reprezentace S. Základní výsledek autorů této metody říká, že když S má jen o málo méně než m prvků, tak pro rozumné funkce je velká pravděpodobnost, že taková dvojice funkcí existuje. Co to jsou rozumné funkce? Zde se obrátíme k univerzálním systémům funkcí. Řekneme, že systém funkcí $H = \{h_i \mid i \in I\}$ je c-téměř silně k-univerzální, kde h_i pro $i \in I$ jsou funkce z univerza U do $\{0, 1, \ldots, m-1\}$, když pro každou prostou posloupnost prvků $x_1, x_2, \ldots, x_k$ z univerza U a každou posloupnost prvků $y_1, y_2, \ldots, y_k$ z množiny $\{0, 1, \ldots, m-1\}$ je

$$\text{Prob}\{i \in I \mid h_i(x_j) = y_j \text{ pro každé } j = 1, 2, \ldots, k\} \leq \frac{c}{m^k}.$$

Když $U = \{0, 1, \ldots, N-1\}$, kde N je prvočíslo a $m = N$, pak polynomy stupně $k+1$ tvoří 1-téměř silně k-univerzální systém. Dále existují konstanty $\varepsilon, c > 0$ takové, že když $|S| = n$, $m \geq (1+\varepsilon)n$ a H je $(1, \lceil c \log n \rceil)$-univerzální, pak když zvolíme $h_0, h_1 \in H$ náhodně a nezávisle, tak pravděpodobnost, že S není reprezentovatelná pomocí h_0 a h_1, je menší než $\frac{1}{n}$. Kukaččí hašování je založeno na tomto výsledku.
Popíšeme operaci **INSERT**(x) za předpokladu, že S je reprezentována pomocí funkcí h_0 a h_1. Když máme vložit prvek x a řádek $h_0(x)$ v tabulce T_0 je obsazen prvkem $y_0 \neq x$, pak nahradíme prvek y_0 prvkem x a zkusíme vložit y_0 do tabulky T_1. Navíc si označíme řádek v tabulce T_0. Pak spočítáme $h_1(y_0)$. Pokud řádek $h_1(y_0)$ je volný, pak do něho vložíme prvek y_0 a skončíme. V opačném případě je obsazen prvkem $y_1 \neq y_0$. Když řádek $h_1(y_0)$ není označen, pak y_0 nahradí y_1 na řádku $h_1(y_0)$ v tabulce T_1 a my pokračujeme tak, že označíme řádek $h_1(y_0)$ v tabulce T_1 a pokusíme se vložit y_1 do tabulky T_0. Když je řádek označen, tak končíme s informací, že x nelze vložit do tabulky T_0. Tento proces cyklicky opakujeme.
Důvod, že jsme mohli říct, že x nelze vložit, je ten, že jsme nalezli posloupnost prvků $y_0, y_1, \ldots, y_k$ takovou, že

(1) y_0 je na $h_0(x)$-tém řádku v tabulce T_0;

(2) když $2i+1 < k$, pak y_{2i+1} je na $h_1(y_{2i})$-tém řádku v tabulce T_1;
(3) když $2i+2 < k$, pak y_{2i+2} je na $h_0(y_{2i+1})$-ním řádku v tabulce T_0;
(4) když k je liché, pak pro nějaké $2i+1 < k$ platí $h_0(y_k) = h_0(y_{2i+1})$, když k je sudé, pak pro nějaké $2i < k$ platí $h_1(y_k) = h_1(y_{2i})$.

To znamená, že jsme nalezli posloupnost prvků, která se zacyklila a operace **INSERT** by jí jen posunovala v cyklu. Důsledkem je, že v této reprezentaci nelze x vložit do tabulky T_0. V tom případě vrátíme tabulky do původního stavu. Stejný postup lze provést i s vkládáním prvku x do tabulky T_1.
Na začátku předpokládáme, že není označen žádný řádek. Pokus o vložení prvku x na k-tý řádek tabulky T_i pro $i = 0, 1$ realizuje následují procedura, která má za parametry i a k. Navíc předpokládáme, že k-tý řádek v tabulce T_i je obsazen prvkem různým od x.

Vloz(i, k)
y :=prvek na k-tém řádku tabulky T_i, Q :=prázdná fronta
vlož y do Q, x na k-tý řádek v tabulce T_i a označ ho
$j := i$, $i := 1 - i$, $l := k$, $k := h_i(y)$
while k-tý řádek tabulky T_i je obsazen a není označen **do**
  z :=prvek na k-tém řádku tabulky T_i
  y vlož na k-tý řádek tabulky T_i a označ ho
  $i := 1 - i$, $y := z$, $k := h_i(y)$, vlož y do Q
enddo
if k-tý řádek tabulky T_i není obsazen **then**
  y vlož na k-tý řádek tabulky T_i
  zruš označení všech řádků, vyprázdni frontu Q
else
  while Q není prázdná **do**
    zruš označení l-tého řádku v tabulce T_j, z :=vrchol fronty Q
    odstraň z z Q, z vlož na l-tý řádek tabulky T_j, $j := 1 - j$, $l := h_j(u)$
  enddo
  Výstup: vložení se nepovedlo
endif

Bohužel první cyklus, který hledá volný řádek se může až $|S|$-krát opakovat, kde S je reprezentovaná množina. Aby toto nemohlo nastat (i když tento jev není moc pravděpodobný), tak je stanovená hodnota q závislá na velikosti reprezentované množiny, která omezuje počet opakování tohoto cyklu a tedy podprocedura vyžaduje čas $O(q)$.

Nyní popíšeme vlastní algoritmus **INSERT**(x). Spočítáme $h_0(x)$ a $h_1(x)$. Pokud $h_0(x)$-tý řádek tabulky T_0 nebo $h_1(x)$-tý řádek tabulky T_1 obsahuje x, tak končíme. Pokud $h_0(x)$-tý tabulky T_0 je prázdný, tak tam vložíme x a končíme. Když není prázdný, tak testujeme, zda $h_1(x)$-tý řádek tabulky T_1 je prázdný, v tom případě tam vložíme x a končíme.
Když oba řádky jsou neprázdné ale neobsahují x, pak zkusíme vložit x, tak, že nejprve zavoláme podproceduru **Vlož**$(0, h_0(x))$, a když úspěšně přerovná prvky, tak končíme. Když dostaneme zprávu "vložení se nepovedlo", tak zavoláme podproceduru **Vlož**$(1, h_1(x))$. Když se jí povede přerovnat prvky, tak končíme. Když obě volání podprocedury **Vlož** končí hláškou, "vložení se nepovedlo", tak oznámíme,

že **INSERT**(x) nelze provést, reprezentace vložené množiny nelze rozšířit o prvek x. Tady je formální zápis algoritmu.

INSERT(x)
Spočítej $k_0 := h_0(x)$ a $k_1 := h_1(x)$
if $T_0(k_0) = x$ nebo $T_1(k_1) = x$ **then** stop **endif**
if $T_0(k_0) =$ empty **then** $T_0(k_0) := x$, stop **endif**
if $T_1(k_1) =$ empty **then** $T_1(k_1) := x$, stop **endif**
Vlož$(0, k_0)$
if podprocedura **Vlož** hlásí, že vnoření se nepovedlo **then**
  Vlož$(1, k_1)$
  if podprocedura **Vlož** hlásí, že vnoření se nepovedlo **then**
    Výstup: prvek x nelze do této struktury vložit
  endif
endif

Pokud dostaneme hlášení, že současná reprezentace množiny S neumožňuje její rozšíření o prvek x, pak provedeme přehašování. To znamená, že zvolíme náhodně a nezávisle dvě funkce v systému H a hledáme reprezentaci $S \cup \{x\}$ pomocí nových dvou funkcí tak, že opakujeme algoritmus operace **INSERT**(y) pro $y \in S \cup \{x\}$ (S je reprezentovaná množina).

Všimněme si, že nalezení navštíveného vrcholu znamená, že prvky uložené pomocí h_0 a h_1 se "zacyklily". Pokud se to stane pro tabulku T_0 i T_1, tak to znamená, že reprezentace $S \cup \{x\}$ pomocí h_0 a h_1 neexistuje.
Je třeba ještě hlídat velikosti reprezentovaných množin. Když po **INSERT**(x) je v množině S mnoho prvků nebo po operaci **DELETE**(x) málo prvků, tak se tabulky zdvojnásobí nebo se stanou poloviční.

Autoři ukázali, že když H je 1-téměř silně $\lceil \log n \rceil$-univerzální systém, pak operace **INSERT** a **DELETE** mají očekávaný amortizovaný čas konstantní.

Analýzu kukaččího hašování si ukážeme v letním semestru.

Kukaččí hašování navrhli a analyzovali R. Pagh a F. F. Rodler v publikaci z roku 2004.

Analýza operace **MEMBER** v binárních vyhledávacích stromech

Z minulých přednášek víme, že klasické algoritmy pro vyhledávání v binárních vyhledávacích stromech vyžadují čas úměrný hloubce vrcholu reprezentujícího argument operace. Ukážeme si vztah, který spojuje binární vyhledávací stromy (bez vyvažování) s **QUICKSORT**em. Tento vztah nám umožní analýzovat čas vyžadovaný operacemi **MEMBER** a **INSERT**. Budeme pracovat s verzí algoritmu, kde pivot je volen jako první člen vstupní posloupnosti. Vstupní posloupnosti algoritmu **QUICKSORT** budou prosté náhodně vybrané posloupnosti $\mathcal{P} = \{x_1, x_2, \ldots, x_n\}$ z totálně uspořádaného univerza. Dále vytvoříme binární vyhledávací strom T reprezentující množinu $\{x_1, x_2, \ldots, x_n\}$ tak, že aplikujeme posloupnost operací

$$\textbf{INSERT}(x_1), \textbf{INSERT}(x_2), \ldots, \textbf{INSERT}(x_n)$$

na binární vyhledávací strom reprezentující prázdnou množinu (použijeme algoritmus **INSERT** bez vyvažovacích operací).
Množina reprezentovaná podstromem určeným levým synem kořene T je $\{x_i \mid i \in \{2,3,\ldots,n\},\ x_i < x_1\}$ a množina reprezentovaná podstromem určeným pravým synem kořene T je množina $\{x_i \mid i \in \{2,3,\ldots,n\},\ x_i > x_1\}$. Přitom tyto stromy vznikly aplikacemi posloupností

$$\{\mathbf{INSERT}(x_i) \mid i = 1,2,\ldots,n,\ x_i < x_1\} \text{ a}$$
$$\{\mathbf{INSERT}(x_i) \mid i = 1,2,\ldots,n,\ x_i > x_1\}.$$

Když předpokládáme, že **QUICK** vytváří nové posloupnosti, ale nemění pořadí prvků v těchto posloupnostech, tak **QUICK** je rekurzivně volán právě na posloupnosti (pivot byl prvek x_1)

$$\{x_i \mid i = 2,3,\ldots,n,\ x_i < x_1\} \text{ a}$$
$$\{x_i \mid i = 2,3,\ldots,n,\ x_i > x_1\}.$$

Protože tyto posloupnosti zpracovává **QUICK** v rostoucím pořadí vzhledem k posloupnosti $\mathcal{P}$, dostáváme indukcí podle n:

> podstrom určený levým synem kořene T je izomorfní s podstromem rekurzivních volání **QUICK**u na posloupnost $\{x_i \mid i = 1,2,\ldots,n, \mid x_i < x_1\}$;
> podstrom určený pravým synem kořene T je izomorfní s podstromem rekurzivních volání **QUICK**u na posloupnost $\{x_i \mid i = 1,2,\ldots,n,\ x_i > x_1\}$

(pivot je reprezentován ve vrcholu, který odpovídá rekurzivnímu volání **QUICK**u, když byl zvolen pivotem). Z toho plyne, že strom T je izomorfní se stromem rekurzivních volání procedury **QUICK** v algoritmu **QUICKSORT** na vstupní posloupnost $\mathcal{P}$.
Jak jsme si ukázali při analýze **QUICKSORT**u, počet porovnání provedených algoritmem **QUICKSORT** (tj. porovnání ve všech volání podprocedury **QUICK**) je $\sum\{|T_v| \mid v \in T\}$, kde T_v je podstrom určený vrcholem v (připomínám, že každé volání procedury **QUICK** porovnáná pivot s každým členem vyšetřované posloupnosti, ale jiné porovnání neprovádí). Za předpokladu, že vstupní posloupnosti jsou vybírány náhodně s rovnoměrným rozdělením, tak jsme si ukázali, že očekávaná hodnota součtu $\sum\{|T_v| \mid v \in T\}$ je nejvýše $2n \ln n$.
Nyní vyšetříme součet času pro provedení posloupnosti operací

$$\mathbf{MEMBER}(x_1),\mathbf{MEMBER}(x_2),\ldots,\mathbf{MEMBER}(x_n)$$

v binárním vyhledávacím stromě T. Když označíme p_i délku cesty z kořene stromu T do vrcholu reprezentujícího x_i pro $i = 1,2,\ldots,n$, pak můžeme psát, že tento součet je $\sum_{i=1}^{n}(p_i+1)$ (předpokládáme, že porovnání argumentu operace s prvkem reprezentovaným ve vyšetřovaném vrcholu a posun na další vrchol vyžaduje jednotku času). Všimněme si, že pro vrchol $v \in T$ je $|T_v|$ právě počet i takových, že v leží na cestě z kořene do vrcholu reprezentujícího x_i. Tedy přispěvek x_i do T_v je 1 pro každý vrchol v na cestě z kořene do vrcholu reprezentujícího vrchol x_i a takových vrcholů je právě $p_i + 1$. Dostáváme, že

$$\sum_{v \in T} |T_v| = \sum_{i=1}^{n}(p_i + 1).$$

Když použijeme odhad nalezený při analýze **QUICKSORT**u a fakt, že všechny vrcholy jsou argumentem operace se stejnou pravděpodobností, tak dostáváme následující výsledek

Věta. *Za předpokladu rovnoměrného rozdělení dat, je očekávaná hloubka vrcholu reprezentujícího náhodný prvek x v binárním vyhledávacím stromě reprezentujícího n-prvkovou množinu nejvýše $2 \ln n$. Tedy očekávaný čas operací* **MEMBER** *a* **INSERT** *je $O(\log n)$.* □

Tuto analýzu nemůžeme použít na posloupnosti operací, které obsahují **DELETE**, protože operace **DELETE** porušuje rovnoměrné rozdělení dat.

Huffmanův kód

Další důležitým problémem je komprese velkých dat. Nejde jen o ušetření paměti, ale např. když se data přenáší po pomalé lince, tak je výhodné, aby přenášená data byla co možná nejmenší. Základní idea je použít kódy, kde zakódování symbolu nemá stejnou délku. Ukážeme si ideu zakódování do slov z 0 a 1.

Nechť C je množina symbolů, které chceme zakódovat. Pak kód je reprezentován úplným binárním stromem T (tj. každý vnitřní vrchol má dva syny) a bijekcí mezi listy stromu T a prvky množiny C. Když pro každý symbol $c \in C$ je dáná četnost $d(c)$ symbolu c v daném textu, který má být zakódován, pak velikost kódu je $H(T) = \sum d(c)h(c)$, kde $h(c)$ je hloubka listu reprezentujícího symbol c. Problém je nalézt pro danou množinu C s četnostmi $d(c)$, kód T, který má minimální velikost $H(T)$. Ten se nazývá optimální. Postup pro řešení problému navrhnul v roce 1952 Huffman.
Když T je kód množiny C, pak pro každý vnitřní vrchol $v \in T$ označíme syny vrcholu v číslicemi 0 a 1. Každý list l je jednoznačně určen cestou z kořene do l a ta jednoznačně odpovídá nějakému slovu z $\{0, 1\}$. Toto slovo je pak kódem symbolu reprezentovaného listem l. Je vidět, že text zakódovaný nějakým kódem lze snadno dekódovat a pokud jsou známé četnosti, tak optimální kód dává nejúspornější zápis textu.

Algoritmus pro nalezení optimálního kódu používá ideje **MERGESORT**u. Nejprve několik pojmů. Kořenový les je množina kořenových stromů. Vrchol se nazývá list lesa, když je listem některého stromu z lesa V. Velikost lesa V, značíme ji $|V|$, je počet stromů ve V. Předpokládejme, že ϕ je zobrazení z množiny listů kořenového lesa V do reálných čísel.
Pak

$$\mathrm{Cont}(V, \phi) = \sum_{l \text{ je list } V} d(l)\phi(l),$$

kde $d(l)$ je hloubka listu l, tj. délka cesty ve V z kořene stromu ve V obsahujícího l do vrcholu l. Všimněme si, že můžeme mluvit o $\mathrm{Cont}(V', \phi)$ i pro podmnožinu stromů $V' \subseteq V$.

Nyní sformulujeme náš problém:
Vstup: Množina L a zobrazení ϕ z L do reálných čísel.
Výstup: úplný binární strom T s množinou listů L takový, že $\mathrm{Cont}(T, \phi)$ je minimální.

Řekneme, že T je optimální strom vzhledem k L a ϕ, když L je množina listů T a $\mathrm{Cont}(T, \phi)$ je nejmenší možný.

Uvažujme následující algoritmus (verze hladového algoritmu), kde s každým stromem T vyšetřovaného kořenového lesa jsou spojené dvě proměnné, $\mathrm{Cont}(T)$ a

$c(T) = \sum\{\phi(l) \mid l \in T \cap L\}$. Algoritmus se volá s dvěma parametry – funkcí ϕ a množinou L listů kořenového lesa. Na počátku se bude kořenový les skládat z jednoprvkových stromů $T = \{t\}$ pro $t \in L$. Tedy L je množina listů tohoto kořenového lesa. Když kořenový les není strom, tak algoritmus vezme z tohoto lesa dva stromy T_1 a T_2 takové, že mají minimální součet $\mathrm{Cont}(T_1) + \mathrm{Cont}(T_2) + c(T_1) + c(T_2)$. Pak stromy T_1 a T_2 v kořenovým lesem nahradí stromem T, který zkonstruuje tak, že vezme nový vrchol v, ten bude kořenem T a jeho synové budou kořeny T_1 a T_2. Pak platí $c(T) = c(T_1) + c(T_2)$ a

$$\mathrm{Cont}(T) = \mathrm{Cont}(T_1) + \mathrm{Cont}(T_2) + c(T_1) + c(T_2).$$

Zřejmě se množina listů kořenového stromu nezměnila a pokud kořenový les bude strom, tak algoritmus končí a tento strom je výsledek.

Optim(L, ϕ):
V je kořenový les skládající se z jednoprvkových stromů takových, že množina listů V je L
for every strom $T = \{t\}$ z V **do** $c(T) := \phi(t)$ **enddo**
while $|V| > 1$ **do**
  vezmeme z V dva stromy T_1 a T_2 s nejmenším součtem
  $\mathrm{Cont}(T_1) + \mathrm{Cont}(T_2) + c(T_1) + c(T_2)$
  odstraníme T_1 a T_2 z V
  vezmeme nový vrchol v, synové v jsou kořeny stromů T_1 a T_2
  v je kořen nového stromu T
  $\mathrm{Cont}(T) = \mathrm{Cont}(T_1) + \mathrm{Cont}(T_2) + c(T_1) + c(T_2)$
  $c(T) = c(T_1) + c(T_2)$, T vložíme do V
enddo
Výstup: (V, ϕ).

Věta. *Pro danou množinu L o velikosti n a funkci ϕ z L do reálných čísel algoritmus* **Optim** *nalezne optimální strom T pro L a ϕ. Když L je zadána jako seznam takový, že $\phi(a) \le \phi(\text{následník } a)$ pro každý prvek L (kromě posledního prvku), pak* **Optim** *vyžaduje čas $O(n)$.*

Důkaz. Nejprve si všimněme, že v průběhu algoritmu se nemění množina listů lesa. Algoritmus končí, když $|V| = 1$, tedy když V je strom. Když v daném okamžiku V je množina stromů $T_1, T_2, \ldots, T_k$ a stromy T_1 a T_2 nahradíme stromem T, pak dostaneme les V' složený ze stromů $T, T_3, \ldots, T_k$, tedy $|V'| + 1 = |V|$. Protože na začátku V obsahoval jen n jednoprvkových stromů, tak algoritmus po konečném počtu kroků skončí. Ukázali jsme, že každý běh cyklu **while do** zmenší počet stromů o jeden, ale nezmění množinu listů. Proto výsledný les V je strom s n listy a ϕ je bijekce z množiny listů V do čísel $\{1, 2, \ldots, n\}$.
Zbývá ukázat, že V je optimální strom vzhledem k L a ϕ. Tvrzení dokážeme indukcí podle velikosti. Když $|L| \le 2$, pak tvrzení zřejmě platí. Předpokládejme, že tvrzení platí pro každou množinu M a funkci ψ, když $|M| < |L|$. Nechť $L = \{x_1, x_2, \ldots, x_n\}$ a $\phi(x_i) \le \phi(x_{i+1})$ pro každé $i = 1, 2, \ldots, n-1$. Bez újmy na obecnosti můžeme předpokládat, že v prvním kroku algoritmus **Optim** zvolil jednoprvkové stromy $T_1 = \{x_1\}$ a $T_2 = \{x_2\}$. Uvažujme množinu $M = (L \setminus \{x_1, x_2\}) \cup \{y\}$, kde y je nový prvek $\phi(y) = \phi(x_1) + \phi(x_2)$. Když vytvoříme strom V' ze stromu V tak, že odstraníme listy x_1 a x_2 a vrchol y ztotožníme s jejich otcem, pak V' je výsledkem

algoritmu **Optim** na vstup M a ϕ, a protože $|M| < |L|$, tak podle indukčního předpokladu V' je optimální vzhledem k M a ϕ. Protože L je konečna, tak existuje optimální strom U vzhledem k L a ϕ. Nechť U optimální strom vzhledem k L a ϕ. Všimněme si, že když u je vnitřní vrchol U s největší hloubkou, pak jeho synové jsou listy. Předpokládejme, že $u \in U$ je vnitřní vrchol s největší hloubkou a s nejmenším součtem $\sum\{\phi(u') \mid u' \text{ je syn } u\}$. Nechť u_1 a u_2 jsou synové u takové, že $\phi(u_1) \leq \phi(u_2)$. Z vlastnosti L pak platí $\phi(x_1) \leq \phi(u_1)$ a $\phi(x_2) \leq \phi(u_2)$. Dokážeme $\phi(u_1) = \phi(x_1)$ a $\phi(u_2) = \phi(x_2)$. Vytvořme strom U' tak, že vyměňíme u_i a x_i pro $i = 1, 2$. Pro strom U' pak platí

$$\begin{aligned}
&\mathrm{Cont}(U', \phi) - \mathrm{Cont}(U, \phi) = \\
&\quad (d(x_1) - d(u_1))\phi(u_1) + (d(x_2) - d(u_2))\phi(u_2) + \\
&\qquad (d(u_1) - d(x_1))\phi(x_1) + (d(u_2) - d(x_2))\phi(x_2) = \\
&\quad (d(u_1) - d(x_1))(\phi(x_1) - \phi(u_1)) + \\
&\qquad (d(u_2) - d(x_2))(\phi(x_2) - \phi(u_2)).
\end{aligned}$$

Protože $d(u_1) = d(u_2) \geq d(x_1), d(x_2)$ (z definice u) tak, kdyby některá z nerovnosti $\phi(x_1) \leq \phi(u_1)$ a $\phi(x_2) \leq \phi(u_2)$ byla ostrá, tak bychom dostali $\mathrm{Cont}(U', \phi) - \mathrm{Cont}(U, \phi) < 0$ a to je spor s optimalitou U. Tedy můžeme předpokládat, že synové u jsou x_1 a x_2. Vytvořme strom T' ze stromu U tak, že vynecháme listy x_1 a x_2 a otce vrcholu x_1 ztotožníme s y. Pak množina listů T' je M a tedy platí

$$\begin{aligned}
\mathrm{Cont}(V', \phi) &= \mathrm{Cont}(V, \phi) - \phi(x_1) - \phi(x_2) \leq \mathrm{Cont}(T', \phi) \\
&= \mathrm{Cont}(U, \phi) - \phi(x_1) - \phi(x_2),
\end{aligned}$$

protože $\phi(y) = \phi(x_1) + \phi(x_2)$ a hloubka y je v obou případech o 1 menší než hloubka x_1 a x_2. Odtud plyne, že $\mathrm{Cont}(V, \phi) \leq \mathrm{Cont}(U, \phi)$ a tedy V je optimální strom vzhledem k L a ϕ.
Předpokládejme, že L je dán seznamem $x_1, x_2, \ldots, x_n$ takovým, že $\phi(x_i) \leq \phi(x_{i+1})$ pro každé $i = 1, 2, \ldots, n-1$. Algoritmus **Optim** postupně tvoří víceprvkové stromy $T_1, T_2, \ldots, T_k$. Pak indukcí okamžitě dostáváme, že

$$\mathrm{Cont}(T_1) \leq \mathrm{Cont}(T_2) \leq \cdots \leq \mathrm{Cont}(T_k).$$

Tedy stačí, když použijeme následující strukturu:
daný seznam L a v něm ukazatel na první prvek, který je jednoprvkový strom v kořenovým lese V (pak před ukazatelem jsou listy, které jsou ve víceprvkových stromech ve V a za ukazatelem jsou prvky tvořící ještě jednoprvkový stromy ve V); a frontu víceprvkových stromů (to znamená, že stromy odebíráme zepředu a ukládáme je dozadu). Udržovat tyto struktury vyžaduje čas $O(1)$ stejně jako nalezení dvou stromů s nejmenším ohodnocením. Tedy algoritmus **Optim** zkonstruuje optimální strom v čase $O(n)$. □

Při aplikaci na naši původní úlohu musíme ještě setřídit vstupní množinu symbolů podle četnosti (které budou listy). Tato četnost je vyjádřena přirozenými čísly a na její setřídění pak můžeme použít algoritmus **BUCKETSORT** (minulá přednáška). Pro určení četnosti musíme projít vstupní text.

V mnoha úlohách v praxi se setkáváme s různými verzemi tohoto problému, např při optimalizaci násobení matic, které nejsou čtvercové. Proto je vhodné znát toto řešení.

Obecné vyvážené stromy

Na závěr uvedeme další verzi vyvážených binárních vyhledávacích stromů. Jejich vynik byl motivován zjištěním z konce minulého století, že pro programátory velkých firem jsou červeno-černé nebo AVL-stromy příliš složité (efektivní programy pro tyto struktury lze nyní nalézt na Internetu). Přitom použití vyvážených stromů by značně zrychlilo jejich produkty. Proto byla navržena následující datová struktura.

Množina $S \subseteq U$ je reprezentována binarním vyhledávacím stromem a navíc jsou dány horní hranice pro hloubku stromu a počet úspěšných operací **DELETE** provedených od posledního vyvažování. Hloubka stromu je vždy shora omezena hodnotou $\lceil c \log |S| \rceil$ pro vhodné $c > 1$ (to je podmínka obecné vyváženosti). Autor experimentálně ověřoval chování datové struktury pro $c \approx 1.3$ (to je lepší než jsou odhady na největší hloubku červeno-černých stromů a AVL-stromů).

Operace **MEMBE**R(x) a **DELETE**(x) se provedou stejným způsobem jako v klasických (nevyvážených) binárních vyhledávacích stromech. Po úspěšném provedení operace **DELETE** se počet operací **DELETE** zvětší o 1 a když překročí stanovenou hranici, provede se vyvážení celého stromu reprezentujícího množinu S. Operace **INSERT**(x) se také provede jako v klasických binárních vyhledávacích stromech, ale navíc při ní počítáme délku cesty z kořene do listu, který bude reprezentovat x. Po přidání x ji zvětšíme o 1 a když překročí výšku stromu, zvětšíme výšku stromu. Pokud výška překročí stanovené omezení na výšku stromu (toto omezení závisí na velikosti reprezentované množiny), pak se provede vyvážení stromu.

Vyvažování stromu se provádí tak, že se najde vrchol v, jehož podstrom se má vyvážit, a tento podstrom se nahradí podstromem reprezentujícím stejnou množinu, ale s nejmenší výškou. Při operaci **INSERT**(x) vrcholem v bude vrchol s nejmenší výškou takový, že jeho podstrom není vyvážený. Nahrazením tohoto podstromu podstromem s nejmenší výškou se zmenší výška celého stromu a ten pak bude splňovat omezení dané na výšku stromu (protože před operací toto omezení splňoval). Při vyvažování po operaci **DELETE** provedeme vyvažování pro kořen stromu, tj. nahradíme původní strom novým binárním vyhledávacím stromem, který reprezentuje stejnou množinu, ale má nejmenší výšku.

Abychom mohli realizovat tuto operaci, je potřeba znát v každém vrcholu velikost množiny reprezentované podstromem tohoto vrcholu. Proto je potřeba po úspěšné operaci **INSERT** nebo **DELETE** (tj. když se přidal nebo odebral prvek) projít cestu od upravovaného vrcholu nazpět ke kořeni a aktualizovat velikost množin reprezentovaných podstromy na této cestě.

Autor navrhl nevyvážený podstrom převést pomocí rotací a dvojitých rotací do úplně zdegenerovaného tvaru (tj. je to jediná cesta z kořene, k níž jsou přidány listy) a pak opět pomocí rotací a dvojitých rotací vytvořit binární vyhledávací strom s nejmenší výškou. V literatuře lze nalézt celou řadu efektivnějších způsobů pro konstrukci binárního vyhledávacího stromu s nejmenší výškou, ale nikde jsem nenašel vzájemné porovnání jejich efektivity. Proto lze říct, že obecné vyvážené binární vyhledávací stromy otevírají celou řadu (praktických) problémů, které by bylo dobré vyřešit.

V současné době, kdy je datová struktura uložena na serveru a má k ní přístup více uživatelů, začal hrát důležitou roli problém aktualizace. Když se strom aktualizuje, tak se může znehodnotit práce jiného uživatele, která je vykonána ve stejné době. To vedlo k návrhu paralelní implementace datových struktur (nebo také vyvažování shora dolů) – byla uvedena pro (a, b)-stromy. Podobně lze také modifikovat algoritmy pro červeno-černé stromy (pro AVL-stromy nevím, jestli existují takové algoritmy). Na konci minulého století byla navržena jiná strategie řešící tento problém, která je použitelná pro téměř všechny vyvážené struktury.

Relaxované stromy

Idea této strategie je oddělit vyvažování od práce s daty. S touto ideou jsme se setkali už při líné implementaci binomiálních hald. Vedle zamezení konfliktů, které mohou vzniknout kvůli vyvažování můžeme i ušetřit čas, když dojde k vyrušení konfliktů. Nevýhodou této ideje je fakt, že ztrátou vyváženosti se může výrazně prodloužit čas potřebný k vyhledávání (logaritmický vyhledávací čas může vzrůst až na lineární). Na druhé straně víme, že při rovnoměrném rozdělení dat je tento nárůst nepravděpodobný. V praxi se sice s rovnoměrným rozdělením dat nesetkáváme, ale velký nárůst vyhledávacího času je nepravděpodobný i pro rozdělení, které jsou blízká rovnoměrnému rozdělení, což jsou mnohá rozdělení z praxe.

Metoda od sebe odděluje vyhledávací a vyvažovací operace. Místo toho, aby se provedlo vyvažování, se jen zaznamená požadavek na vyvažování do fronty požadavků. Výhoda této metody se zvláště projevuje v časových špičkách, kdy přichází mnoho požadavků od uživatelů a nelze stihnout vyvažování, nebo při práci v dávkovém režimu (např. když administrátor odeslal dávku požadavků, aby odstranil část datové struktury poškozenou výpadkem proudu). Další výhoda této metody spočívá ve faktu, že ji lze snadno modifikovat pro všechny běžně používané vyvážené stromy.

Použití vyžaduje ošetření dvou problémů. O prvním jsme se již zmínili, je to možný nárůst času pro vyhledávání způsobený degenerací datové struktury. Druhý problém je, zda lze jednoduše vyvážit datovou strukturu, která vznikla několika aktualizačními operacemi bez následného vyvažování bez nového budování celé datové struktury z reprezentovaných dat. S tím je spojena otázka strategie řešení vyvažovacích požadavků.

Stromy vzniklé touto metodou se nazývají relaxované. Přitom konkrétních realizací této základní ideje je pro každý typ vyvážených stromů několik (záleží na tom, který parametr je preferován). Popíšeme jeden relaxovaný model pro červeno-černé stromy.

Uvažovaný model má data uložená v binárním vyhledávacím stromu, jehož vnitřní vrcholy jsou obarveny buď červeně nebo černě (obarvení oběma barvami není přípustné) a listy jsou obarveny černě. Navíc je dán soubor vyvažovacích požadavků (budeme ho nazývat fronta vyvažovacích požadavků) a pokud je tento soubor prázdný, pak strom reprezentující data je vyvážený červeno-černý strom. Nad daty pracuje současně více procesů, které jsou dvou typů:

uživatelský – provádí pouze vyhledávání, přidávání a ubírání prvků a když po aktualizaci vznikne požadavek na vyvažování (význam i formu upřesníme později), dá tento požadavek do fronty vyvažovacích požadavků.

správcovský – bere vhodné požadavky z fronty vyvažovacích požadavků a provádí je. Může se stát, že buď daný požadavek úplně ošetří nebo ho transformuje v jiný požadavek bližší ke kořeni stromu. Tím se postupně odstraňuje nevyváženost stromu.

Důležité je, že když činností některého procesu (uživatelského nebo správcovského) se v daném vrcholu porušila podmínka červeno-černých stromů v tomto vrcholu, tak vznikl požadavek spojený s tímto vrcholem. Tento požadavek zanikl, když to opravil některý správcovský proces (přitom mohl porušit podmínky na červeno-černé stromy v jiném vrcholu, a proto musí vznést požadavek spojený s tímto vrcholem).

Ideálem je v jistých periodách vyprázdnit frontu požadavků a tím vytvořit klasický červeno-černý strom. Počet pracujících správcovských procesů není fixní a závisí na délce fronty vyvažovacích požadavků.

Popíšeme sémantiku a formu požadavků. Máme požadavky dvou typů – b a v. Když je s vrcholem v svázán požadavek b, pak s ním už není svázán žádný další požadavek (tj. požadavek b je neslučitelný s každým dalším požadavkem, zatímco požadavků typu v na vrchol v může být víc). Pokud je s vrcholem svázán nějaký požadavek, pak vrchol a požadavek s ním svázaný ve frontě vyvažovacích požadavků jsou spojeny ukazateli. Požadavek b na vrchol v znamená, že v je červený a v okamžiku vznesení požadavku má červeného otce (toto se později může změnit). Požadavek v na vrchol v znamená, že v je černý a v cestách z kořene stromu do listů procházejících vrcholem v chybí jeden černý vrchol. Z toho plyne, že když je na vrchol v vloženo k požadavků v, pak v cestách z kořene do listů procházejících vrcholem v chybí k černých vrcholů, tj. když každý vrchol s k požadavky v nahradíme $k+1$ černými vrcholy, pak všechny cesty z kořene do listů mají stejný počet černých vrcholů. Tento invariant bude vždy splněn.

Ve frontě vyvažovacích požadavků je každý požadavek specifikován svým typem a ukazatelem na vrchol, kde vznikl.

Neformálně popíšeme práci jednotlivých procesů. Uživatelský proces provádí jednu ze tří operací: **MEMBER**, **INSERT** a **DELETE**.

Operace **MEMBER**(x) klasickým vyhledáváním zjistí, zda x patří do reprezentované množiny, a oznámí to uživateli (pokud patří, oznámí také adresu dat spojených s prvkem x).

Operace **INSERT**(x) klasickým vyhledáváním zjistí, zda x patří do reprezentované množiny. Když patří, operace končí (zde je několik přirozených alternativ – např. oznámí neúspěšné vložení prvku nebo oznámí adresu dat spojených s prvkem x atd.) Když x nepatří do reprezentované množiny, tak vyhledávání skončilo v listu t (který reprezentuje interval obsahující x). Nyní změní list t na vnitřní vrchol, vytvoří dva černé syny vrcholu t, které budou listy, uloží data spojená s x a jejich adresu spolu s prvkem x spojí s vrcholem t. Když vrchol t vznášel požadavek v, pak smaže jeden požadavek v vznesený vrcholem t a nechá vrchol t černý, když vrchol t nevznášel žádný požadavek v, tak změní jeho barvu na červenou a když otec vrcholu t je červený, vytvoří pro vrchol t požadavek b a vloží ho do fronty vyvažovacích požadavků (propojí vrchol t a tento požadavek ukazateli).

Operace **DELETE**(x) klasickým způsobem zjistí, zda x patří do reprezentované množiny. Když nepatří, operace končí (případně oznámí neúspěch). V opačném

případě nalezne vrchol t a jeho syna l, které mají být odstraněny, uvolní data spojená s prvkem x. Když t nereprezentoval x, pak data spojená s t přesune na vrchol reprezentující x. Pak odstraní vrchol t a list l a na místo vrcholu t dá bratra listu l, kterého obarví na černo. Když t i bratr l byly obarveny černě, vytvoří požadavek v spojený s bratrem l a vloží ho do fronty vyvažovacích požadavků (a propojí ukazateli bratra l s tímto požadavkem). Požadavky b spojené s vrcholy t a bratrem l se ruší a odstraní se z fronty vyvažovacích požadavků (když t nebo bratr l měly požadavek b, pak jejich odstraněním žádný nový požadavek nevzniká). Navíc bratr l dědí všechny požadavky v spojené s vrcholem t (tím se mohou hromadit požadavky v spojené s jedním vrcholem).

Nyní neformálně popíšeme práci správcovského procesu. Proces provede jednu z následujících akcí:

Zruší (a odstraní z fronty) jakýkoliv požadavek na kořen stromu.

Zruší (a odstraní z fronty) požadavek b na vrchol v, když otec vrcholu v je černý.

Když je na vrchol v vloženo i požadavků v a na bratra vrcholu v je vloženo j požadavků v, pak zruší $\min(i, j)$ požadavků v na vrcholy v a bratr(v). Když otec vrcholu v je černý, pak vloží nových $\min(i, j)$ požadavků v na otce vrcholu v. Když otec vrcholu v je červený, pak změní jeho barvu na černou a vloží na něho $\min(i, j) - 1$ požadavků v.

Když je na vrchol v vložen požadavek b a jeho otec je červený a je kořen stromu, pak obarvíme otce vrcholu v na černo a požadavek zrušíme.

Když je na vrchol v vložen požadavek b, otec vrcholu v je červený, děd vrcholu v je černý, bratr u otce vrcholu v je červený, pak obarvíme otce vrcholu v a vrchol u na černo a zrušíme požadavek na vrchol v. Když na děda vrcholu v je vloženo i požadavků v pro $i > 0$, pak odstraníme jeden požadavek v na děda vrcholu v. Pokud na děda vrcholu v není vložen žádný požadavek v (požadavek b na něho nemůže být vložen, protože je černý), obarvíme ho na červeno a pokud otec děda vrcholu v je také červený, pak vložíme na děda vrcholu v požadavek b. Když na vrchol u je vložen požadavek b, tak ho zrušíme.

Když na vrchol v je vložen požadavek b, otec z vrcholu v je červený, děd w vrcholu v je černý, bratr vrcholu z je černý a v není lomený, pak provedeme rotaci vrcholů z a w, w obarvíme červeně, z obarvíme černě, vrchol z zdědí všechny požadavky v vrcholu w a zrušíme požadavek b na vrchol v.

Když na vrchol v je vložen požadavek b, otec z vrcholu v je červený, děd w vrcholu v je černý, bratr otce vrcholu v je černý a v je lomený, pak provedeme dvojitou rotaci vrcholů v, z a w, w obarvíme červeně, v obarvíme černě, zrušíme požadavek b na vrchol v a vrchol v zdědí všechny požadavky v na vrchol w.

Srovnejte tyto tři akce s vyvažováním po operaci **INSERT** v klasickém červeno-černém stromu.

Když na vrchol v je vloženo j požadavků v pro $j > 0$, bratr u vrcholu v je černý a není na něho vložen žádný požadavek, synové vrcholu u jsou černí, pak obarvíme vrchol u červeně, zrušíme jeden požadavek v na vrchol v a když byl otec vrcholu v červený, tak ho obarvíme na černo, a pokud byl černý, tak vytvoříme požadavek v na otce vrcholu v a vložíme ho do fronty vyvažovacích požadavků.

Když na vrchol v je vloženo j požadavků v pro $j > 0$, bratr u vrcholu v je černý a není na něho vložen žádný požadavek, syn w vrcholu u, který je lomený, je červený,

pak provedeme dvojitou rotaci na vrcholy w, u a otec(v), obarvíme vrchol otec(v) černě, zrušíme jeden požadavek v na vrchol v a vrchol w zdědí barvu i požadavky spojené s vrcholem otec(v). Zrušíme také případný požadavek b na vrchol w.

Když na vrchol v je vloženo j požadavků v pro $j > 0$, bratr u vrcholu v je černý a není na něho vložen žádný požadavek, syn w vrcholu u, který není lomený, je červený, pak provedeme rotaci na vrcholy u a otec(v), obarvíme vrcholy otec(v) a w černě, zrušíme jeden požadavek v na vrchol v a vrchol u zdědí barvu i požadavky spojené s vrcholem otec(v). Zrušíme také případný požadavek b na vrchol w.

Když na vrchol v je vloženo j požadavků v pro $j > 0$, a bratr u vrcholu v je červený a není na něho vložen žádný požadavek a jeho lomený syn je černý, pak provedeme rotaci na vrcholy u a otec(v)z, obarvíme otec(v) na červeno (a nebude na něm žádný požadavek), vrchol u obarvíme na černo a zdědí všechny požadavky vrcholu otec(v). Nyní bratr vrcholu v je černý a spravcovský proces opakuje akci na vrchol v.

Předchozí akce srovnejte s operací **DELETE** pro klasické červeno-černé stromy.

Nyní stručně popíšeme práci s frontou vyvažovacích požadavků. Správcovský proces se náhodně rozhodne, zda chce odstranit požadavky na kořen, nebo zda chce odstranit požadavek typu b, nebo zda chce odstranit požadavek typu v.

Když chce odstranit požadavek na kořen, pak prohledáním fronty vyvažovacích požadavků nalezne požadavek na kořen. Zablokuje kořen. Během této akce nesmí být kořen argumentem žádné rotace nebo dvojité rotace (pak by přestal být kořenem a požadavek nejde odstranit). Požadavek odstraní a kořen uvolní. Místo prohledávání fronty je rychlejší začít u kořene, zjistit, zda je na něj položen požadavek, a nalézt tento požadavek pomocí ukazatele.

Když chce odstranit požadavek typu b, pak nejprve nalezne vrchol v, který není kořen, je na něho položen požadavek b a na otce vrcholu v není položen požadavek b. Zablokuje vrchol v a testuje, zda otec vrcholu není černý nebo není kořen stromu. Když otec vrcholu v je černý nebo je kořen stromu, pak zablokuje otce vrcholu v a provede akci. Po jejím provedení uvolní oba vrcholy. Když otec vrcholu v je červený a není kořen stromu a děd vrcholu v je černý, tak zablokuje otce vrcholu v, bratra otce vrcholu v a děda vrcholu v a podle vlastností bratra otce vrcholu v provede akci a vrcholy uvolní.

Když chce odstranit požadavek v, tak nalezne vrchol v, na který je vložen požadavek v a není kořen a na bratra vrcholu v není vložen požadavek b. Pak zablokuje otce vrcholu v, bratra vrcholu v i vrchol v. Podle vlastností bratra vrcholu v provede akci. V případě, že bratr vrcholu v je černý a není na něho vložen žádný požadavek, tak zablokuje i syny bratra vrcholu v. Po provedení akce uvolní vrcholy. Pokud bratr vrcholu v byl obarven červeně a nebyl na něho položen žádný požadavek a jeho lomený syn je černý, tak provede akci na vrchol v, a pak teprve uvolní vrcholy.

Při volbě, jaký požadavek bude ošetřovat správcovský server, by největší prioritu mělo mít ošetřování kořene, pak ošetření požadavku b a pak požadavku v. To by se mělo odrazit při volbě akce správcovského procesu. Vhodný poměr priorit není znám. Zdá se, že je také výhodné začít hledat požadavky ve stromě a pak přejít do fronty vyvažovacích požadavků pomocí ukazatele. Zablokovaný vrchol znamená,

že není přístupný jinému správcovskému procesu. Jen při provádění rotace nebo dvojité rotace jsou její argumenty zablokovány i pro uživatelské procesy.

Následující tvrzení se ověří přímo (viz vyvažování pro červeno-černé stromy).

Věta. *Když fronta vyvažovacích požadavků je neprázdná, tak v ní vždy existuje požadavek, který může obsloužit správcovský proces. Když vrchol v binárního vyhledávacího stromu reprezentujícího data je červený a jeho otec je také červený, pak na vrchol v byl vložen požadavek b. V každém okamžiku platí, že když každý vrchol binárního vyhledávacího stromu reprezentujícího data, na který je vloženo i požadavků* v *pro* $i > 0$*, je nahrazen* $i+1$ *černými vrcholy, pak všechny cesty z kořene do listů mají stejný počet černých vrcholů.*

Z této věty plyne, že když je fronta požadavků prázdná, pak binární vyhledávací strom reprezentující data je červeno-černý. Vzniká však otázka, zda každá posloupnost akcí správcovských procesů vede k vyprázdnění fronty požadavků. Řešení tohoto problému dává následující pozorování:

Každá akce správcovského procesu buď zmenší počet požadavků nebo zmenší součet všech 'hloubek požadavků', kde 'hloubka požadavku' je hloubka vrcholu, který je s tímto požadavkem spojen.

Všimněte si, že když se odstraní požadavek pomoci **Rotace** nebo **Dvojité-rotace**, tak se mohl součet zvětšit, protože vrcholy v podstromu změnili svoji hloubku.

Tedy definujme dva potenciály, $\phi_1(t)$ je počet požadavků v čase t a $\phi_2(t) = \sum\{h(p) \mid p$ je požadavek$\}$, kde $h(p)$ je hloubka vrcholu spojena s požadavkem p. Platí:

Lemma. *Každa akce správcovského procesu buď zmenší* ϕ_1 *nebo se* ϕ_1 *nezmění, ale zmenší se* ϕ_2*.*

Z předchozího tvrzení plyne, že každá posloupnost akcí správcovského procesu vede k vyprázdnění fronty vyvažovacích požadavků, a je zřejmé, že posloupnost operací v klasických červeno-černých stromech vyžaduje více vyvažovacích akcí. Jsou zde otevřené otázky:

Lze najít optimální strategii pro správcovské procesy? S jakou pravděpodobností je binární strom v závislosti na délce fronty ještě blízký pravidelnému úplnému binárnímu stromu?

Podobné modely byly studovány i pro AVL-stromy a (a, b)-stromy.