

# Datové struktury

přednáší Václav Koubek      z<sub>TeX</sub>ali Martin Vidner <mvidner@atlas.cz>,  
Vladimír Kotal <vlada@devnull.cz> <sup>1</sup>

16. června 2003

<sup>1</sup>Přispěli: Lišák, Jéňa, Žabička, Jindřich, Martin Mareš, Pavel Machek, . . .

# Obsah

<b>1 Úvod</b>	<b>5</b>
1.1 Předpoklady . . . . .	5
1.2 Jaké typy složitosti nás zajímají . . . . .	5
1.2.1 Paměťová složitost reprezentované struktury . . . . .	5
1.2.2 Časová složitost algoritmů pracujících na datové struktuře . . . . .	5
<b>2 Slovníkový problém</b>	<b>7</b>
2.1 Pole . . . . .	7
2.2 Seznam . . . . .	7
<b>3 Hašování I</b>	<b>8</b>
3.1 Hašování se separovanými řetězci . . . . .	8
3.1.1 Očekávaná délka seznamu . . . . .	9
3.1.2 Očekávaný čas posloupnosti operací . . . . .	9
3.1.3 Očekávaný počet testů . . . . .	9
3.1.4 Očekávaná délka nejdelšího seznamu . . . . .	10
3.2 Hašování s uspořádanými řetězci . . . . .	11
3.2.1 Očekávaný čas . . . . .	11
3.3 Hašování s přesuny . . . . .	11
3.4 Hašování se dvěma ukazateli . . . . .	12
3.5 Hašování s lineárním přidáváním . . . . .	12
3.6 Hašování se dvěma funkcemi (otevřené h., h. s otevřenou adresací) . . . . .	13
3.6.1 Algoritmus INSERT . . . . .	14
3.6.2 Očekávaný počet testů . . . . .	14
3.7 Srůstající hašování (standardní: LISCH a EISCH) . . . . .	15
3.8 Srůstající hašování s pomocnou pamětí (obecně: LICH, EICH, VICH) . . . . .	16
3.8.1 Srovnávací graf . . . . .	16
3.9 Srovnání metod . . . . .	16
3.10 Přehašování . . . . .	17
<b>4 Hašování II</b>	<b>19</b>
4.1 Univerzální hašování . . . . .	19
4.1.1 Očekávaná délka řetězce . . . . .	20
4.1.2 Velikost $c$ -univerzálního systému . . . . .	21
4.1.3 Reprezentace a (MEMBER), INSERT, DELETE . . . . .	22
4.2 Perfektní hašování . . . . .	23
4.2.1 Perfektní hašovací funkce do tabulky velikosti $n^2$ . . . . .	24
4.2.2 Perfektní hašovací funkce do tabulky velikosti $3n$ . . . . .	25
4.2.3 GPERF . . . . .	26

<b>5</b>	<b>Trie</b>	<b>27</b>
5.1	Základní varianta . . . . .	27
5.1.1	Algoritmus MEMBER . . . . .	27
5.1.2	Algoritmus INSERT . . . . .	27
5.1.3	Algoritmus DELETE . . . . .	28
5.1.4	Časová a paměťová složitost . . . . .	28
5.2	Komprimované trie . . . . .	28
5.2.1	MEMBER . . . . .	29
5.2.2	INSERT . . . . .	29
5.2.3	DELETE . . . . .	29
5.2.4	Časová a paměťová složitost . . . . .	29
5.3	Ještě komprimovanější trie . . . . .	32
5.3.1	Popis $A$ a $rd$ . . . . .	33
5.3.2	Algoritmus pro hledání $rd$ a hod . . . . .	33
5.3.3	Vertikální posun sloupců . . . . .	34
5.3.4	Úsporné uložení řídkého vektoru . . . . .	36
<b>6</b>	<b>Uspořádaná pole</b>	<b>38</b>
6.1	Unární, binární a interpolační vyhledávání . . . . .	38
6.2	Zobecněné kvadratické vyhledávání . . . . .	38
<b>7</b>	<b>Binární stromy</b>	<b>41</b>
7.1	Obecně . . . . .	41
7.1.1	Algoritmus MEMBER . . . . .	41
7.1.2	Algoritmus INSERT . . . . .	41
7.1.3	Algoritmus DELETE . . . . .	41
7.2	Optimální binární vyhledávací stromy . . . . .	41
7.2.1	Algoritmus konstrukce . . . . .	41
7.2.2	Snížení složitosti z kubické na kvadratickou . . . . .	41
7.3	Skorooptimální binární vyhledávací stromy . . . . .	41
7.3.1	Co je to optimální binární vyhledávací strom . . . . .	41
7.3.2	Aproximace optimálních stromů . . . . .	42
7.3.3	Podrobnější popis naznačené metody . . . . .	42
7.3.4	Časová složitost . . . . .	44
7.3.5	Hledání $k$ . . . . .	44
7.4	Červenočerné stromy . . . . .	45
7.4.1	Operace INSERT . . . . .	46
7.4.2	Operace DELETE . . . . .	46
7.4.3	Závěry . . . . .	48
<b>8</b>	<b><math>(a, b)</math> stromy</b>	<b>49</b>
8.1	Základní varianta . . . . .	49
8.1.1	Reprezentace množiny $S(a, b)$ stromem . . . . .	49
8.1.2	MEMBER( $x$ ) v $(a, b)$ stromu . . . . .	50
8.1.3	INSERT( $x$ ) do $(a, b)$ stromu . . . . .	50
8.1.4	DELETE( $x$ ) z $(a, b)$ stromu . . . . .	50
8.1.5	Shrnutí . . . . .	51
8.1.6	Jak volit parametry $(a, b)$ . . . . .	51
8.2	Další operace . . . . .	51
8.2.1	Algoritmus JOIN( $T_1, T_2$ ) pro $(a, b)$ stromy . . . . .	51
8.2.2	Algoritmus SPLIT( $x, T$ ) pro $(a, b)$ strom . . . . .	52
8.2.3	Algoritmus STACKJOIN( $Z$ ) pro zásobník $(a, b)$ stromů . . . . .	52
8.2.4	Algoritmus FIND( $T, k$ ) pro $(a, b)$ strom . . . . .	52
8.2.5	A-sort . . . . .	53

8.3	Paralelní přístup do $(a, b)$ stromů . . . . .	54
8.3.1	Paralelní INSERT( $x$ ) do $(a, b)$ stromu . . . . .	55
8.3.2	Paralelní DELETE( $x$ ) z $(a, b)$ stromu . . . . .	55
8.4	Složitost posloupnosti operací na $(a, b)$ stromu . . . . .	56
8.4.1	přidání/ubrání listu . . . . .	57
8.4.2	štěpení . . . . .	58
8.4.3	spojení . . . . .	58
8.4.4	přesun . . . . .	58
8.5	Propojené $(a, b)$ stromy s prstem . . . . .	60
8.5.1	Algoritmus MEMBER . . . . .	60
8.5.2	Algoritmus FINGER . . . . .	60
8.5.3	Amortizovaná složitost . . . . .	61
<b>9</b>	<b>Samoopravující se struktury</b> . . . . .	<b>62</b>
9.1	Amortizovaná složitost . . . . .	62
9.2	Seznamy . . . . .	62
9.2.1	Algoritmus MEMBER . . . . .	62
9.2.2	Algoritmus INSERT . . . . .	62
9.2.3	Algoritmus DELETE . . . . .	62
9.2.4	Algoritmus MFR (Move Front Rule) . . . . .	62
9.2.5	Algoritmus TR (Transposition Rule) . . . . .	64
9.3	Splay stromy . . . . .	65
9.3.1	Operace SPLAY . . . . .	65
9.3.2	Podporované operace . . . . .	65
9.3.3	Algoritmus MEMBER . . . . .	66
9.3.4	Algoritmus JOIN2 . . . . .	66
9.3.5	Algoritmus JOIN3 . . . . .	66
9.3.6	Algoritmus SPLIT . . . . .	66
9.3.7	Algoritmus DELETE . . . . .	67
9.3.8	Algoritmus INSERT . . . . .	67
9.3.9	Algoritmus CHANGEWEIGHT . . . . .	67
9.3.10	Algoritmus SPLAY . . . . .	68
9.3.11	Amortizovaná složitost SPLAY . . . . .	68
9.3.12	Amortizovaná složitost ostatních operací . . . . .	69
<b>10</b>	<b>Haldy</b> . . . . .	<b>70</b>
10.1	$d$ -regulární haldy . . . . .	70
10.1.1	Algoritmus UP . . . . .	70
10.1.2	Algoritmus DOWN . . . . .	71
10.1.3	Operace na haldě . . . . .	71
10.1.4	Algoritmus MAKEHEAP . . . . .	71
10.1.5	Složitost operací . . . . .	72
10.1.6	Dijkstrův algoritmus . . . . .	72
10.2	Leftist haldy . . . . .	73
10.3	Binomiální haldy . . . . .	74
10.3.1	Operace . . . . .	75
10.3.2	Líná implementace binom. hald . . . . .	76
10.3.3	Zobecněné binomiální haldy . . . . .	77
10.4	Fibonacciho haldy . . . . .	77

<b>11 Dynamizace</b>	<b>78</b>
11.1 Zobecněný vyhledávací problém . . . . .	78
11.1.1 Operace INSERT a DELETE . . . . .	78
11.2 Semi-dynamizace . . . . .	79
11.3 Dynamizace . . . . .	82
11.3.1 Repräsentace množiny A . . . . .	83
11.3.2 Paměťové nároky . . . . .	83
11.3.3 čas pro výpočet f . . . . .	83
11.3.4 Amortizovaný čas operace DELETE . . . . .	84
<b>12 Vícedimenzionální vyhledávání</b>	<b>85</b>

# Kapitola 1

## Úvod

Chceme reprezentovat data, provádět s nimi operace. Cíle téhle přednášky jsou popsat ideje, jak datové struktury reprezentovat, popsat algoritmy pracující s nimi a přesvědčit vás, že když s nimi budete pracovat, měli byste si ověřit, jak jsou efektivní.

Problém měření efektivity: většinou nemáme šanci vyzkoušet všechny případy vstupních dat. Musíme buď doufat, že naše vzorky jsou dostatečně reprezentativní, nebo to vypočítat. Tehdy ale zase nemusíme dostat přesné výsledky, pouze odhady.

### 1.1 Předpoklady

1. Datové struktury jsou nezávislé na řešeném problému; abstrahujeme. Například u slovníkových operací *vyhledej*, *přidej*, *vyjmi*, nás nezajímá, jestli slovník reprezentuje body v prostoru, vrcholy grafu nebo záznamy v databázi.
2. V programu, který řeší nějaký problém, se příslušné datové struktury používají *velmi často*.

### 1.2 Jaké typy složitosti nás zajímají

#### 1.2.1 Paměťová složitost reprezentované struktury

Je důležitá, ale obvykle jednoduchá na spočítání a není šance ji vylepšit — jedině použít úplně jinou strukturu. Proto ji často nebudeme ani zmiňovat.

#### 1.2.2 Časová složitost algoritmů pracujících na datové struktuře

##### Časová složitost v nejhorším případě

Její znalost nám zaručí, že nemůžeme být nepříjemně překvapeni (dobou běhu algoritmu). Hodí se pro *interaktivní* režim — uživatel sedící u databáze průměrně dobrou odezvu neocení, ale jediný pomalý případ si zapamatuje a bude si stěžovat. Za vylepšení nejhoršího případu obvykle platíme zhoršením průměrného případu.

##### Očekávaná časová složitost

Je to vlastně vážený průměr — složitost každého případu vstupních dat násobíme pravděpodobností jeho výskytu a sečteme. Je zajímavá pro dávkový režim zpracování. Například Quicksort patří mezi nejrychlejší známé třídící algoritmy, ale v nejhorším případě má složitost kvadratickou.

Pozor na předpoklady o rozdělení vstupních dat. Je známý fakt, že pro každé  $k$  existuje číslo  $n_k$  takové že každý náhodný graf s alespoň  $n_k$  vrcholy s velkou pravděpodobností obsahuje kliku

velikosti  $k$ . To vede k následujícímu algoritmu, který určí zda graf je nejvýše  $k - 1$  barevný s očekávaným konstantním časem.

Algoritmus: vstup graf  $(V, E)$ .

1. Když velikost  $V$  je menší než  $n_k$ , pak prohledáním všech možností určí barevnost grafu  $(V, E)$  a vydej odpověď, jinak pokračuj na následující bod.
2. Zvol náhodně  $n_k$  vrcholů v množině  $V$  a zjisti zda indukovaný podgraf na této množině obsahuje kliku velikosti  $k$ . Pokud ano, odpověď je ne, jinak pokračuj na následující bod.
3. Prohledáním všech možností určí barevnost grafu  $(V, E)$  a vydej odpověď.

Tento algoritmus se ale pro praktické použití moc nehodí, protože normálně se například s náhodnými grafy na 200 vrcholech nesetkáváme.

### Amortizovaná složitost

Pro každé  $n$  nalezneme nejhorší čas vyžadovaný posloupností  $n$  operací a tento čas vydělíme  $n$ . Amortizovaná složitost je limitou těchto hodnot pro  $n$  jdoucí do nekonečna. To nás zajímá proto, že některé datové struktury mají takovou vnitřní organizaci, že na ní závisí složitost, a ta organizovanost se během posloupnosti operací mění. Nejhorší případ vlastně „uklízí“ za následující nebo předchozí rychlé případy.

Například u přičítání jedničky ke  $k$ -cifernému binárnímu číslu je časová složitost počet jedniček ve vstupu. Nejhorší případ s lineární složitostí nastane pro číslo ze samých jedniček (tedy  $2^k - 1$ ), ale těch případů je málo a amortizovaná složitost nakonec vyjde konstantní.

Nebo určité algoritmy v překladačích v praxi běží rychle, přestože jednotlivé operace mají velkou složitost v nejhorším případě. To se také podařilo vysvětlit pomocí amortizované složitosti.

### Asymptotická notace

Skutečná složitost závisí na implementaci algoritmu, na konkrétním počítači, vlastně se nedá přesně spočítat v obecném případě. Abychom mohli spočítat aspoň něco, začaly se používat odhady složitosti až na multiplikativní konstantu. Tyto odhady popisují růst složitosti vzhledem ke zvětšujícím se vstupům, ale neurčují konkrétní funkci (čísla).

Nechť  $f, g$  jsou funkce na přirozených číslech. Značíme:

$$f = O(g), \quad \text{když} \quad \exists c > 0 \forall n : f(n) \leq cg(n) \quad (1.1)$$

$$f = \omega(g) \quad \forall c > 0 \exists \text{nekonečně mnoho } n : f(n) > cg(n) \quad (1.2)$$

$$f = \Theta(g) \quad \exists c, d > 0 \forall n : dg(n) \leq f(n) \leq cg(n) \quad (1.3)$$

Budeme převážně používat  $O$ , protože chceme hlavně horní odhady, kdežto dolní odhady bývá obvykle těžší zjistit. Pro úplnost:

$$f = o(g), \text{ když } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

## Kapitola 2

# Slovníkový problém

Je dáno universum  $U$ , máme reprezentovat jeho podmnožinu  $S \subseteq U$ .

Budeme používat operace

- $\text{MEMBER}(x), x \in U$ , odpověď je *ano*, když  $x \in S$ , *ne*, když  $x \notin S$ .
- $\text{INSERT}(x), x \in U$ , vytvoří reprezentaci množiny  $S \cup \{x\}$
- $\text{DELETE}(x), x \in U$ , vytvoří reprezentaci množiny  $S \setminus \{x\}$
- $\text{ACCESS}(x)$ . Ve skutečných databázích  $\text{MEMBER}$  nestačí, protože se kromě klíče prvku zajímáme i o jeho ostatní atributy. Tady se ale o ně starat nebudeme — obvyklé řešení je mít u klíče pouze ukazatel na ostatní data, což usnadňuje přemísťování jednotlivých prvků datové struktury.

Předpokládá se znalost těchto základních datových struktur: pole, spojový seznam, obousměrný seznam, zásobník, fronta, strom.

### 2.1 Pole

Do pole velikosti  $|U|$  uložíme charakteristickou funkci  $S$ .

- + Velmi jednoduché a rychlé — všechny operace probíhají v konstantním čase  $O(1)$
- Paměťová náročnost  $O(|U|)$ , což je kámen úrazu. Např. databáze všech lidí v Česku, kódovaných rodným číslem, by snadno přerostla možnosti 32-bitového adresního prostoru (10 miliard RČ  $\times$  4B ukazatel) Ale pro grafové algoritmy je tahle reprezentace velmi vhodná.

Najít lepší příklad?

### 2.2 Seznam

Vytvoříme seznam prvků v  $S$ , operace provádíme prohledáním seznamu. Časová i paměťová složitost operací je  $O(|S|)$  (a to i pro  $\text{INSERT}$  — musíme zjistit, jestli tam ten prvek už není).



# Kapitola 3

## Hašování I

Universum  $U$ , reprezentovaná podmnožina  $S, S \subseteq U, |S| = n$ . Velikost tabulky:  $m$ .

Charakteristická funkce je velké plýtvání paměti pokud  $n \ll |U|$ , např. pro  $n = \log \log |U|$ .

S prvky, které nesou kladnou informaci ( $x \in S$ ), moc nenaděláme. Ale záporné můžeme nějak sdrcnout do jednoho nebo i překrýt s těmi kladnými. To je idea hašování.

Máme hašovací funkci  $h : U \rightarrow \{0..m-1\}$ . Množina  $S$  je reprezentována polem  $P[0..m-1]$  tak, že prvek  $s \in S$  je uložen na místě  $h(s)$ .

Problémy:

1. Jak rychle spočítáme  $h(s)$ .
2. Co znamená *uložen na místě*  $h(s)$ . Co když  $h(s) = h(t)$ , ale  $s \neq t$ .

Řešení:

1. Omezíme se na rychle spočítatelné hašovací funkce. Předpokládáme, že  $h(s)$  spočteme v čase  $O(1)$ .
2. Tento případ se nazývá *kolize* a jednotlivé druhy hašování se dělí podle toho, jak řeší kolize.

### 3.1 Hašování se separovanými řetězci

Hašovací tabulka je pole lineárních seznamů, ne nutně uspořádaných.

MEMBER(x):

1. Spočítáme  $h(x)$ .
2. Prohledáme  $h(x)$ -tý seznam.
3. Když tam  $x$  je, vrátíme *true*, jinak *false*.

INSERT(x):

1. Spočítáme  $h(x)$ . (*Jako MEMBER*)
2. Prohledáme  $h(x)$ -tý seznam. (*Jako MEMBER*)
3. Když  $x$  není v  $h(x)$ -tém seznamu, tak ho tam vložíme.

DELETE(x):

1. Spočítáme  $h(x)$ . (*Jako MEMBER*)
2. Prohledáme  $h(x)$ -tý seznam. (*Jako MEMBER*)

3. Když  $x$  je v  $h(x)$ -tém seznamu, tak ho odstraníme.

Očekávaná doba operace je stejná jako očekávaná délka seznamu. Ale pozor na prázdný seznam, u něj nedosáhneme nulového času operace. Ukážeme, že očekávaná doba operace je konstantní.

Předpoklady:

1.  $h$  rozděluje prvky  $U$  do seznamů nezávisle a rovnoměrně (např.  $h(x) = x \bmod m$ ). Tedy pro  $\forall i, j : 0 \leq i, j < m$  se počty prvků  $S$  zobrazených na  $i$  a  $j$  liší nejvýš o 1.
2. Množina  $S$  má rovnoměrné rozdělení — výběr konkrétní množiny  $S$  má stejnou pravděpodobnost. To je dost omezující, protože na rozdíl od hašovací funkce nejsme schopni  $S$  ovlivnit.

### 3.1.1 Očekávaná délka seznamu

Označme  $p(\ell) = \mathcal{P}(\text{seznam je dlouhý } \ell)$ .

Z předpokladů má  $p(\ell)$  binomické rozdělení, neboli

$$p(\ell) = \binom{n}{\ell} \left(\frac{1}{m}\right)^\ell \left(1 - \frac{1}{m}\right)^{n-\ell},$$

tedy očekávaná délka seznamu je

$$\begin{aligned} E &= \sum_{\ell=0}^n \ell \cdot p(\ell) \\ &= \sum_{\ell=0}^n \ell \frac{n!}{\ell!(n-\ell)!} \left(\frac{1}{m}\right)^\ell \left(1 - \frac{1}{m}\right)^{n-\ell} = \sum_{\ell=0}^n n \frac{(n-1)!}{(\ell-1)![(n-1)-(\ell-1)]!} \left(\frac{1}{m}\right)^\ell \left(1 - \frac{1}{m}\right)^{n-\ell} \\ &= \sum_{\ell=0}^n \frac{n}{m} \binom{n-1}{\ell-1} \left(\frac{1}{m}\right)^{\ell-1} \left(1 - \frac{1}{m}\right)^{(n-1)-(\ell-1)} = \frac{n}{m} \sum_{k=-1}^{n-1} \binom{n-1}{k} \left(\frac{1}{m}\right)^k \left(1 - \frac{1}{m}\right)^{(n-1)-k} \\ &\quad \text{README.1st: všechny úpravy směřují k tomuto součtu podle binomické věty} \\ &= \frac{n}{m} \left(\frac{1}{m} + \left(1 - \frac{1}{m}\right)\right)^{n-1} = \frac{n}{m} = \alpha, \quad (3.1) \end{aligned}$$

kde  $\alpha = n/m$  je tzv. faktor naplnění, load factor, obvykle je důležité, je-li větší či menší než 1.

### 3.1.2 Očekávaný čas posloupnosti operací

Když máme posloupnost  $P$  operací MEMBER, INSERT, DELETE splňující předpoklad rovnoměrného rozdělení a aplikovanou na prázdnou hašovací tabulku, pak očekávaný čas je  $O(|P| + \frac{|P|^2}{2m})$

### 3.1.3 Očekávaný počet testů

Složitost prohledání seznamu se může lišit podle toho, jestli tam hledaný prvek je nebo není. Úspěšným případem nazveme takovou Operaci( $x$ ), kde  $x \in S$ , neúspěšný případ je  $x \notin S$ . V úspěšném případě prohledáváme průměrně jenom polovinu seznamu.

Očekávaný čas pro neúspěšný případ EČN =  $O\left(\left(1 - \frac{1}{m}\right)^n + \frac{n}{m}\right)$

Očekávaný čas pro úspěšný případ EČÚ =  $O\left(\frac{n}{2m}\right)$

co je to test?  
porovnání klíčů,  
nahlédnutí do  
tabulky?

#### Neúspěšný případ

Projdeme celý seznam, musíme nahlédnout i do prázdného seznamu.

$$\text{EČN} = 1 \cdot p(0) + \sum_{\ell=1}^n \ell \cdot p(\ell) = p(0) + \sum_{\ell=0}^n \ell \cdot p(\ell) = \left(1 - \frac{1}{m}\right)^n + \frac{n}{m} \approx e^{-\alpha} + \alpha$$

### Úspěšný případ

Počet testů pro vyhledání všech prvků v seznamu délky  $\ell$  je

$$1 + 2 + \dots + \ell = \binom{\ell+1}{2}.$$

Očekávaný počet testů je  $\sum_{\ell} \binom{\ell+1}{2} p(\ell)$ , očekávaný počet testů pro vyhledání všech prvků v tabulce je  $m \cdot \sum_{\ell} \binom{\ell+1}{2} p(\ell)$ .

Ještě budeme potřebovat následující sumu, kterou spočítáme podobně jako v 3.1:

$$\sum_{l=0}^n l^2 \binom{n}{l} \left(\frac{1}{m}\right)^l \left(1 - \frac{1}{m}\right)^{n-l} = \dots = \frac{n}{m} \left(1 - \frac{1}{m}\right) + \left(\frac{n}{m}\right)^2$$

Očekávaný počet testů pro vyhledání jednoho prvku

$$\begin{aligned} \text{EČÚ} &= \frac{m}{n} \sum_{\ell} \binom{\ell+1}{2} p(\ell) = \frac{m}{n} \cdot \frac{1}{2} \left( \sum_{\ell} \ell^2 p(\ell) + \sum_{\ell} \ell \cdot p(\ell) \right) \\ &= \frac{m}{2n} \left( \frac{n}{m} \left(1 - \frac{1}{m}\right) + \frac{n^2}{m^2} + \frac{n}{m} \right) \\ &= \frac{1}{2} - \frac{1}{2m} + \frac{n}{2m} + \frac{1}{2} = 1 + \frac{n-1}{2m} \\ &\sim 1 + \frac{\alpha}{2} \quad (3.2) \end{aligned}$$

### 3.1.4 Očekávaná délka nejdelšího seznamu

Známe očekávané hodnoty, ale ty nám samy o sobě moc neřeknou. Hodila by se nám standardní odchylka, ta se ale složitě počítá. Místo toho vypočteme očekávaný nejhorší případ:

Dokážeme, že za předpokladů 1 a 2 a  $|S| = n \leq m$  je očekávaná délka maximálního seznamu  $\text{EMS} = O\left(\frac{\log n}{\log \log n}\right)$ .

Z definice

$$\text{EMS} = \sum_j j \cdot \mathcal{P}(\text{maximální délka seznamu} = j).$$

Použijeme trik: necht

$$q(j) = \mathcal{P}(\text{existuje seznam, který má délku alespoň } j).$$

Pak

$$\mathcal{P}(\text{maximální délka seznamu} = j) = q(j) - q(j+1)$$

a

$$\text{EMS} = \sum_j q(j)$$

(teleskopická suma)

Spočteme  $q(j)$ :

$$q'(j) = \mathcal{P}(\text{daný seznam má délku alespoň } j) \leq \binom{n}{j} \left(\frac{1}{m}\right)^j$$

$$q(j) \leq m \cdot q'(j)$$

$$\text{EMS} \leq \sum \min\left(1, m \binom{n}{j} \left(\frac{1}{m}\right)^j\right) \leq \sum \min\left(1, m \left(\frac{n}{m}\right)^j \frac{1}{j!}\right) \leq \sum \min\left(1, \frac{n}{j!}\right)$$

Necht

$$j_0 = \max\{k : k! \leq n\} \leq \max\{k : (k/2)^{k/2} < n\} = O\left(\frac{\log n}{\log \log n}\right),$$

pak

$$\begin{aligned}
 \text{EMS} &\leq \sum_{j=0}^{j_0} 1 + \sum_{j=j_0}^{\infty} \frac{n}{j!} = j_0 + \sum_{j=j_0}^{\infty} \frac{n}{j_0!} \frac{j_0!}{j!} \\
 &\leq j_0 + \sum_{j=j_0}^{\infty} \frac{j_0!}{j!} \leq j_0 + \sum_{j=j_0}^{\infty} \left(\frac{1}{j_0}\right)^{(j-j_0)} \leq j_0 + \frac{1}{1-1/j_0} \\
 &= O(j_0) = O\left(\frac{\log n}{\log \log n}\right) \quad \square \quad (3.3)
 \end{aligned}$$

## 3.2 Hašování s uspořádanými řetězci

Uspořádání řetězců vylepší neúspěšný případ.

### 3.2.1 Očekávaný čas

Očekávaný čas v neúspěšném případě se od času v úspěšném případě liší jen o aditivní konstantu.

## 3.3 Hašování s přesuny

Zatím jsme předpokládali, že řetězce kolidujících prvků jsou uloženy někde v dynamicky alokované paměti. To není výhodné, protože vyžaduje použití další paměti i když některé řetězce jsou prázdné. Proto nyní budeme ukádat řetězce přímo v tabulce.

Řetězec na  $i$ -tém místě uložíme do tabulky tak, že první prvek je na  $i$ -tém místě a pro každý prvek řetězce je v položce **vpřed** adresa následujícího prvku řetězce a v položce **vzad** je adresa předchozího prvku. Začátek, resp. konec řetězce má prázdnou položku **vzad**, resp. **vpřed**.

Například pro  $U = \mathbb{N}$ ,  $m = 10$ ,  $h(x) = x \bmod 10$ , hašujeme posloupnost 10, 50, 21, 60:

	klíč	vpřed	vzad
0	10	5	
1	21		
2			
3	50		5
4			
5	60	3	0
6			
7			
8			
9			

MEMBER je jednoduchý.

Při INSERT musíme zjistit, zda  $h(x)$ -tý řetězec začíná na  $h(x)$ -tém místě. Pokud ano, prvek přidáme do  $h(x)$ -tého řetězce, pokud ne, přemístíme prvek na  $h(x)$ -tém místě na jiné volné místo, upravíme **vpřed** a **vzad** a prvek vložíme na  $h(x)$ -té místo.

Předchozí tabulka po INSERT(53):

	klíč	vpřed	vzad
0	10	5	
1	21		
2			
3	53		
4	50		5
5	60	4	0
6			
7			
8			
9			

Při DELETE musíme testovat, zda odstraňovaný prvek není na 1. místě svého řetězce a pokud ano a řetězec má více prvků, musíme přesunout jiný prvek z tohoto řetězce na místo odstraňovaného prvku.

Jak zjistíme, že jiný prvek  $y$  patří tam, kde je uložen? Spočítat  $h(y)$  může být relativně pomalé. Pokud  $T[i].vzad$  někam ukazuje, pak víme, že  $h(y) \neq h(x)$ .

Tady mám zmatek. Zavést lepší značení.

### 3.4 Hašování se dvěma ukazateli

Při hašování s přesuny ztrácíme čas právě těmi přesuny, obzvláště, když jsou záznamy velké. To motivuje následující implementaci hašování s řetězci.

Použijeme dva ukazatele, **vpřed** a **začátek**.  $T[i].vpřed$  je index následujícího prvku v řetězci, který je zde uložen. (Nemusí to být řetězec s  $h(x) = i$ .)  $T[i].začátek$  je index začátku řetězce, který obsahuje prvky, jejichž  $h(x) = i$ .

Ukládáme 50, 90, 31, 60:

	klíč	vpřed	začátek
0	50	3	0
1	31		1
2	60		
3	90	2	
4			
5			
6			
7			
8			
9			

Přidáme 42, 72, 45:

	klíč	vpřed	začátek
0	50	3	0
1	31		1
2	60		5
3	90	2	
4	45		
5	42	6	4
6	72		
7			
8			
9			

### 3.5 Hašování s lineárním přidáváním

Jde to i bez ukazatelů.

Je dáno  $m$  míst, která tvoří tabulku. Pokud je příslušné políčko již zaplněno, hledáme cyklicky první volné následující místo a tam zapíšeme. Vhodné pro málo zaplněnou tabulku ( $< 60\%$ , pro  $80\%$  vyžaduje už hodně času). Téměř nemožné DELETE: buď označit místo jako smazané, nebo celé přehašovat.

	klíč
0	120
1	51
2	72
3	
4	
5	
6	
7	
8	
9	

Přidáme 40, 98, 62, 108:

	klíč
0	120
1	51
2	72
3	40
4	62
5	
6	
7	
8	98
9	108

### 3.6 Hašování se dvěma funkcemi (otevřené h., h. s otevřenou adresací)

Použijeme dvě hašovací funkce,  $h_1$  a  $h_2$ , je zde však účelné předpokládat, že  $h_2(i)$  a  $m$  jsou nesoudělné pro každé  $i \in U$ . Při INSERTu pak hledáme nejmenší  $i$  takové, že  $T[h_1(x) + ih_2(x)]$  je volné.

Mějme  $h_1(x) = x \bmod 10$

	klíč
0	10
1	31
2	
3	
4	
5	
6	
7	
8	
9	

INSERT(100):  $h_1(100) = 0$  a předpokládejme, že  $h_2(100) = 3$ . Volné místo najdeme pro  $i = 1$ .  
 INSERT(70):  $h_1(70) = 0$  a předpokládejme, že  $h_2(70) = 1$ . Volné místo najdeme pro  $i = 2$ .

	klíč
0	10
1	31
2	70
3	100
4	
5	
6	
7	
8	
9	

Neuvedli jsme explicitní vzorec pro  $h_2$ . Její sestavení je totiž složitější. Všimněte si, že nemůžeme vzít  $h_2(100) = 4$ . Všechny hodnoty  $h_2$  totiž musí být nesoudělné s velikostí tabulky.

### 3.6.1 Algoritmus INSERT

1. spočti  $i = h_1(x)$
2. když tam  $x$  je, pak skonči  
když je místo prázdné, vlož tam  $x$  a skonči
3. když je  $i$ -té místo obsazeno prvkem  $\neq x$ , pak:  
spočti  $h_2(x)$   
 $k = (h_1(x) + h_2(x)) \bmod m$   
while  $k$ -té místo je obsazeno prvkem  $\neq x$  a  $i \neq k$  do  
 $k = (k + h_2(x)) \bmod m$   
enddo
4. když je  $k$ -té místo obsazeno prvkem  $x$ , pak nedělej nic,  
když  $i = k$ , pak ohlaš přeplněno, jinak vlož  $x$  na  $k$ -té místo

Ještě rozmyslet  
značení a  
sjednotit zápis  
algoritmů

Test  $k \neq i$  v kroku 3 brání zacyklení algoritmu. Tento problém má alternativní řešení, nedovolíme vložení posledního prvku (místo testu v cyklu si pamatujeme údaje navíc). Analogické problémy nastávají u hašování s lineárním přidáváním.

Tato metoda pracuje dobře až do 90% zaplnění.

### 3.6.2 Očekávaný počet testů

Předpokládáme, že posloupnost  $h_1(x), h_1(x) + h_2(x), h_1(x) + 2h_2(x), \dots$  je náhodná, tedy že pro každé  $x$  mají všechny permutace řádků tabulky stejnou pravděpodobnost, že se stanou touto posloupností.

#### při neúspěšném vyhledávání

Označíme jej  $C(n, m)$ , kde  $n$  je velikost reprezentované množiny a  $m$  je velikost hašovací tabulky.

Bud'  $q_j(n, m)$  pravděpodobnost, že v tabulce velikosti  $m$  s uloženou množinou velikosti  $n$  jsou při INSERT( $x$ ) obsazená místa  $h_1(x) + ih_2(x)$  pro  $i = 0..j-1$  (tedy řetězec má alespoň  $j$  prvků).

$$\begin{aligned}
 C(n, m) &= \sum_{j=0}^n (j+1)(q_j(n, m) - q_{j+1}(n, m)) \\
 &= \left( \sum_{j=0}^n q_j(n, m) \right) - (n+1)q_{n+1}(n, m) = \sum_{j=0}^n q_j(n, m) \quad (3.4)
 \end{aligned}$$

Protože

$$q_j(n, m) = \frac{n}{m} \frac{n-1}{m-1} \cdots \frac{n-j+1}{m-j+1} = \frac{n}{m} q_{j-1}(n-1, m-1) \quad (3.5)$$

dostaneme po dosazení:

$$\begin{aligned} \dots &= 1 + \sum_{j=1}^{\infty} \frac{n}{m} q_{j-1}(n-1, m-1) = 1 + \frac{n}{m} \sum_{j=1}^{\infty} q_{j-1}(n-1, m-1) \\ &= 1 + \frac{n}{m} C(n-1, m-1) \end{aligned} \quad (3.6)$$

Řešením tohoto rekurentního vzorce je

$$C(n, m) = \frac{m+1}{m-n+1}, \quad (3.7)$$

což dokážeme indukcí:

$$\begin{aligned} C(n, m) &= 1 + \frac{n}{m} C(n-1, m-1) \\ &= 1 + \frac{n}{m} \frac{m}{m-n+1} = \frac{m-n+1+n}{m-n+1} = \frac{m+1}{m-n+1} \approx \frac{1}{1-\alpha} \end{aligned} \quad (3.8)$$

při úspěšném vyhledávání

Součet očekávaných testů všech INSERTů přes vytváření reprezentované množiny dělený velikostí množiny.

$$\begin{aligned} &= \frac{1}{n} \sum_{i=0}^{n-1} C(i, m) = \frac{1}{n} \sum_{i=0}^{n-1} \frac{m+1}{m-i+1} = \frac{m+1}{n} \sum_{i=0}^{n-1} \frac{1}{m-i+1} \\ &= \frac{m+1}{n} \left( \left( \sum_{i=1}^{m+1} \frac{1}{i} \right) - \left( \sum_{i=1}^{m-n+1} \frac{1}{i} \right) \right) \approx \frac{m+1}{n} \ln \frac{m+1}{m-n+1} \approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \end{aligned} \quad (3.9)$$

Následující tabulka dává očekávanou dobu vyhledávání pro různé zaplnění hašovací tabulky.

$\alpha$	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	1.7	2.55	3.15	4.65	6.9

### 3.7 Srůstající hašování (standardní: LISCH a EISCH)

Tabulka má dvě položky: klíč a adresu následujícího prvku. Prvek  $s \in S$  je reprezentován v řetězci, který pokračuje v místě  $h(s)$ .

V následující tabulce jsou srostlé řetězce pro 0 a 3:

	klíč	vpřed
0	10	3
1	21	
2		
3	40	4
4	33	7
5		
6		
7	70	
8		
9		

INSERT(x)



1. spočti  $i = h(x)$
2. prohledej řetězec začínající na místě  $i$  a pokud nenajdeš  $x$ , přidej ho do tabulky a připoj ho do toho řetězce.

Kam do toho řetězce máme připojit nový prvek? (To je jiná otázka, než které volné místo zvolit.) Metoda LISCH (late insert standard coalesced hashing) ho připojí na poslední místo řetězce, metoda EISCH (early insert standard coalesced hashing) ho připojí hned za  $h(x)$ -té místo.

LISCH INSERT(103), EISCH INSERT(94):

	klíč	vpřed
0	10	3
1	21	
2		
3	40	4
4	33	6
5		
6	94	7
7	70	9
8		
9	103	

Při úspěšném vyhledání je EISCH asi o 15% rychlejší než LISCH. (Při neúspěšném jsou samozřejmě shodné).

### 3.8 Srůstající hašování s pomocnou pamětí (obecné: LICH, EICH, VICH)

Standardní srůstající hašování má tu nevýhodu, že se při větším zaplnění tabulky mohou vytvořit dlouhé řetězce. Tabulku tedy prodloužíme o pomocnou pamět („sklep“), do které se nedostane hašovací funkce, a kolidující prvky přidáváme odspodu. Řetězce tedy srostou až po zaplnění sklepa.

Opět existují varianty připojení nového prvku do řetězce: LICH a EICH jsou analogické k LISCH a EISCH. VICH (variable insert coalesced hashing) připojuje na konec řetězce, jestliže řetězec končí ve sklepe, jinak na místo, kde řetězec opustil sklep.

INSERT 10, 41, 60, 70, 71, 90, 69, 40, 79:

	LICH		EICH		VICH	
	klíč	vpřed	klíč	vpřed	klíč	vpřed
0	10	12	10	(12)(11)(9)7	10	12
1	41	10	41	10	41	10
2						
3						
4						
5						
6	79		79	8	79	8
7	40	6	40	9	40	9
8	69	7	69	11	69	
9	90	8	90	(11)(8)6	90	(8)6
10	71		71		71	
11	70	9	70	12	70	(9)7
12	60	11	60		60	11

#### 3.8.1 Srovnávací graf

### 3.9 Srovnání metod

Zde uvádíme porovnání podle počtu testů, protože to se dá *vypočítat*. Doba běhu se musí *měřit*.

V neúspěšném případě:

1. h. s uspořádanými řetězci (nejlepší)
2. h. s přesuny
3. VICH=LICH a h. se 2 ukazateli (VICH je lepší až do  $\alpha = 3/4$ )
4. EICH
5. LISCH=EISCH (až sem je vše  $O(1)$ )
6. h. se 2 funkcemi
7. h. s lineárním přidáváním (nejhorší)

Počet testů pro úplně zaplněnou tabulku ( $m = n$  nebo  $m = n - 1$ )

h. s přesuny	1.5
h. se 2 ukazateli	1.6
VICH=LICH	1.8
EICH	1.92
LISCH=EISCH	2.1
h. se 2 funkcemi	$n$
h. s lineárním přidáváním	$n$

typ	úspěšné vyhledání	neúspěšné hledání
s řetězci	$1 + \frac{\alpha}{2}$	$e^{-\alpha} + \alpha$
s uspořádanými řetězci	$1 + \frac{\alpha}{2}$	$e^{-\alpha} + 1 + \frac{\alpha}{2} - \frac{1}{\alpha}(1 - e^{\alpha})$
s přemísťováním	$1 + \frac{\alpha}{2}$	$e^{-\alpha} + \alpha$
se 2 ukazateli	$1 + \frac{\alpha}{2} + \frac{\alpha^2}{2}$	$1 + \frac{\alpha^2}{2} + \alpha + e^{-\alpha}(2 + \alpha) - 2$
s lineárním přidáváním	$\frac{1}{2}(1 + \frac{1}{1-\alpha})$	$\frac{1}{2}(1 + (\frac{1}{1-\alpha})^2)$
dvojitě hašování	$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$
LISCH	$1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha)$	$1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$
EISCH	$\frac{1}{\alpha}(e^{\alpha} - 1)$	$1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$

Následující vzorce už Koubek neprobírá, ale nechám je tady, abyste si vážili toho, že jste jich byli ušetřeni :)

$$\text{LICH úspěšné: } \begin{cases} 1 + \frac{\alpha}{2\beta} & \text{když } \alpha \leq \lambda\beta \\ 1 + \frac{\beta}{8\alpha}(e^{2(\alpha/\beta-\lambda)} - 1 - 2(\alpha/\beta - \lambda)) \times (3 - \frac{2}{\beta} + 2\lambda) + \frac{1}{4}(\frac{\alpha}{\beta} + \lambda) + \frac{\lambda}{4}(1 - \frac{\lambda\beta}{\alpha}) & \text{když } \alpha \geq \lambda\beta \end{cases}$$

$$\text{LICH neúspěšné: } \begin{cases} e^{-\alpha/\beta} + \frac{\alpha}{\beta} & \text{když } \alpha \leq \lambda\beta \\ \frac{1}{\beta} + \frac{1}{4}(e^{2(\alpha/\beta-\lambda)} - 1)(3 - \frac{2}{\beta} + 2\lambda) - \frac{1}{2}(\alpha/\beta - \lambda) & \text{když } \alpha \geq \lambda\beta \end{cases}$$

kde  $\beta = \frac{m}{m'}$  je podíl adresové části tabulky na její celkové velikosti a  $\lambda$  je jediné nezáporné řešení rovnice  $e^{-\lambda} + \lambda = \frac{1}{\beta}$ .

### 3.10 Přehašování

V předchozích metodách jsme narazili na případy, že při velkém zaplnění tabulky je nutné ji přehašovat. Zde si ukážeme metodu, jak se to dělá.

Máme hašovací funkce:  $h_0$  hašuje do tabulky velikosti  $m = 2^0m$ ,  $h_1$  do  $2m = 2^1m$ ,  $h_2$  do  $4m = 2^2m \dots$ ,  $h_i$  do  $2^i m$ . Množinu  $S$  reprezentujeme takto:

Uložena je velikost  $S$  a číslo  $i$  takové, že

$$2^{i-2}m < |S| < 2^i m \quad (3.10)$$

a  $S$  je zahašována funkcí  $h_i$ .

MEMBER funguje normálně, při INSERT a DELETE kontrolujeme porušení podmínky (3.10) a případně přehašujeme pro  $i \pm 1$ :

INSERT: Provedeme operaci INSERT a když máme přidat prvek, testujeme, zda  $|S| + 1 < 2^i m$ . Pokud nerovnost platí, dokončíme INSERT. Pokud neplatí, zvětšíme  $i$  o 1 a spočítáme uložení  $S \cup \{x\}$  vzhledem k nové hašovací funkci  $h_i$ .

DELETE: Provedeme operaci DELETE a když máme odstranit prvek, testujeme, zda  $i > 0$  a  $|S| - 1 = 2^{i-2}m$ . Pokud rovnost neplatí, dokončíme DELETE. Pokud platí, zmenšíme  $i$  o 1 a spočítáme uložení  $S - \{x\}$  vzhledem k nové hašovací funkci  $h_i$ .

Tato metoda má malou amortizovanou složitost. Když se spočítá hašovací tabulka pro novou hašovací funkci  $h_i$ , pak obsahuje  $2^{i-1}m$  prvků a proto je třeba alespoň  $2^{i-2}m$  úspěšných operací DELETE nebo  $2^{i-1}m$  úspěšných operací INSERT, abychom přepočítávali tabulku pro jinou hašovací funkci. Tedy amortizovaná složitost přepočítávání tabulky je  $O(1)$  (tato metoda není vhodná pro práci v interaktivním režimu).

# Kapitola 4

## Hašování II

### 4.1 Univerzální hašování

Idea univerzálního hašování má odstranit požadavek na rovnoměrné rozložení vstupních dat. Tento požadavek chceme nahradit tím, že budeme mít soubor  $H$  hašovacích funkcí do tabulky velikosti  $m$  takový, že pro každou podmnožinu  $S$  univerza  $U$  je pravděpodobnost, že funkce z  $H$  se chová dobře, hodně velká (tj. je jen málo kolizí). V tomto případě, když vybereme  $h$  z  $H$  náhodně s rovnoměrným rozložením, pak pro každou podmnožinu  $S \subseteq U$  takovou, že  $|S| \leq m$ , bude očekávaný čas (počítaný přes všechny funkce z  $H$ ) konstantní. Rozdíl proti tradičnímu hašování je, že předpoklad rovnoměrného výběru hašovací funkce z množiny  $H$  můžeme zajistit (nebo se k splnění tohoto požadavku přiblížit), ale výběr vstupních dat ovlivnit nemůžeme. Nyní tuto ideu zformalizujeme.

zajímá nás  
jednak  $c$ , jednak  
velikost systému

**Definice 4.1.1.** Třída hašovacích funkcí  $H \subseteq \{h|h : \{0 \dots N-1\} \rightarrow \{0 \dots m-1\}\}$  je  $c$ -univerzální, kde  $c \in \mathbb{R}^+$ , jestliže

$$\forall x \neq y \in \{0 \dots N-1\} : |\{h \in H : h(x) = h(y)\}| \leq c \frac{|H|}{m},$$

Nejprve ukážeme, že  $c$ -univerzální systémy existují. Předpokládáme, že  $U = \{0 \dots N-1\}$ , kde  $N$  je prvočíslo. Definujme

$$H = \{h_{ab} : h_{ab}(x) = ((ax + b) \bmod N) \bmod m; a, b \in \{0 \dots N-1\}\}$$

**Věta 4.1.1.**  $H$  je  $c$ -univerzální a  $c = \left(\lceil \frac{N}{m} \rceil / \frac{N}{m}\right)^2$ .

*Důkaz.*  $|H| = N^2$ , což je počet dvojic  $(a, b)$ .

Nechť  $(x, y)$  jsou libovolné, ale pevné. Kolize nastane v případech, když:

$$h_{ab}(x) = h_{ab}(y),$$

neboli

$$\begin{aligned} ax + b &= q + rm \pmod{N} \\ ay + b &= q + sm \pmod{N} \end{aligned}$$

kde  $(a, b)$  jsou neznámé a parametry  $(q, r, s)$  nabývají všech hodnot takových, že

$$q \in \{0 \dots m-1\} \wedge r, s \in \{0 \dots \lceil N/m \rceil - 1\}.$$

$N$  je prvočíslo, tedy  $\mathbb{Z}_N$  je těleso a pro každou trojici parametrů  $(q, r, s)$  má soustava právě jedno řešení  $(a, b)$ . Počet kolidujících funkcí je přesně tolik, jako počet trojic  $(q, r, s)$ , který je  $m \cdot \lceil N/m \rceil^2$ .

$$|\{h_{ab} : h_{ab}(x) = h_{ab}(y)\}| \leq m \lceil \frac{N}{m} \rceil^2 = \frac{\lceil \frac{N}{m} \rceil^2}{\left(\frac{N}{m}\right)^2} \frac{N^2}{m} = c \frac{|H|}{m}$$

□

### 4.1.1 Očekávaná délka řetězce

Mějme libovolnou pevnou  $S \subseteq U$ , libovolné pevné  $x \in U$  a funkci  $h : U \rightarrow \{0 \dots m-1\}$ . Definujme

$$S_{h,x} = \text{řetězec prvků } y \in S, \text{ pro které platí } h(y) = h(x). \quad (4.1)$$

Chceme spočítat průměrnou délku  $S_x$ , kde průměr počítáme přes všechny  $h \in H$ , kde  $H$  je  $c$ -univerzální systém.

Zavedme

$$\delta_h(x, y) = [x \neq y \wedge h(x) = h(y)] \quad (4.2)$$

$$\delta_h(x, S) = \sum_{y \in S} \delta_h(x, y), \quad (4.3)$$

Iversonova  
konvence:  
[true]=1,  
[false]=0

$$\begin{aligned} \sum_{h \in H} \delta_h(x, S) &= \sum_{h \in H} \sum_{\substack{y \in S \\ y \neq x}} \delta_h(x, y) = \sum_{\substack{y \in S \\ y \neq x}} \sum_{h \in H} \delta_h(x, y) \\ &= \sum_{\substack{y \in S \\ y \neq x}} |\{h \in H; h(x) = h(y)\}| \leq \sum_{\substack{y \in S \\ y \neq x}} \frac{c|H|}{m} \\ &= \begin{cases} \frac{cn|H|}{m} & x \notin S \\ \frac{c(n-1)|H|}{m} & x \in S \end{cases} \end{aligned}$$

Tedy průměrná hodnota  $\delta_h(x, S) \leq \frac{cn}{m}$ .

**Věta 4.1.2.** Pro každou množinu  $S \subseteq U$ ,  $|S| = n$  a každé  $x$  je očekávaný čas operací MEMBER, INSERT, DELETE  $O(c \cdot n/m)$ , přičemž je bráný přes všechny funkce  $h \in H$  při jejich rovnoměrném rozdělení.

**Věta 4.1.3.** Markovova nerovnost:  $\mathcal{P}(X \geq tEX) \leq 1/t$

*Důkaz.*  $X$  je rovnoměrně rozdělená náhodná veličina nabývající hodnot  $\{x_i : i \in I\}$ ,  $I \subset \mathbb{N}$ ,  $I' = \{i \in I : x_i \geq t\mu\}$ , pak

$$\begin{aligned} \mu &= \frac{1}{|I|} \sum_{i \in I} x_i && I' \subset I \\ &> \frac{1}{|I|} \sum_{i \in I'} x_i && \text{z definice } I' \\ &\geq \frac{1}{|I|} \sum_{i \in I'} t\mu \\ &= \frac{|I'|}{|I|} t\mu \end{aligned}$$

a tedy

$$\mathcal{P}(X \geq t\mu) = \frac{|I'|}{|I|} < \frac{1}{t}$$

□

Varianta Markovovy nerovnosti:

**Věta 4.1.4.** Za stejných předpokladů jako u věty 4.1.2, když  $\mu$  je průměrná délka řetězce  $S_{h,x}$ , pak

$$\forall t > 1 \mathcal{P}(|S_{h,x}| \geq t\mu) < \frac{1}{t}$$

*Důkaz.* plyne z Markovovy nerovnosti.

□

### 4.1.2 Velikost $c$ -univerzálního systému

#### Dolní mez

Řekli jsme, že při použití  $c$ -univerzálního systému z něj hašovací funkce vybíráme náhodně. V praxi ale budeme většinou používat pseudonáhodný generátor, který se po určité periodě opakuje. Abychom zajistili co největší náhodnost, potřebujeme, aby systém  $H$  měl co nejméně funkcí.

**Věta 4.1.5.** *Když  $H$  je  $c$ -univerzální systém funkcí z univerza  $U$  do  $\{0 \dots m-1\}$ , pak*

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

*Důkaz.* Mějme  $c$ -univerzální systém  $H = \{h_1 \dots h_{|H|}\}$ . Nechť  $U_0 = U$ .

Nechť  $U_1$  je největší podmnožina  $U_0$  taková že  $h_1$  je na  $(U_1)$  konstantní.

Nechť  $U_2$  je největší podmnožina  $U_1$  taková že  $h_2$  je na  $(U_2)$  konstantní. (Také  $h_1$  je na  $(U_2)$  konstantní) A tak dále.

Platí

$$\begin{aligned} |U_0| &= N \\ |U_1| &\geq \left\lfloor \frac{N}{m} \right\rfloor \\ |U_2| &\geq \left\lfloor \frac{\lfloor N/m \rfloor}{m} \right\rfloor \geq \dagger \left\lfloor \frac{N}{m^2} \right\rfloor \\ |U_i| &\geq \left\lfloor \frac{N}{m^i} \right\rfloor \end{aligned}$$

Nechť  $t = \lceil \log_m N \rceil - 1$ . Platí  $\lceil x \rceil - 1 < x$  a  $\log$  je rostoucí, tedy  $m^t < N$  a

$$|U_t| \geq \left\lfloor \frac{N}{m^t} \right\rfloor > 1,$$

neboli  $U_t$  obsahuje alespoň 2 různé prvky,  $a \neq b$

Nechť  $*$  =  $|\{h \in H : h(a) = h(b)\}|$ . Z definice  $c$ -univerzálního systému  $* \leq \frac{c|H|}{m}$ . Protože  $h_1, \dots, h_t$  jsou na  $U_t$  konstantní, dostáváme  $* \geq t$ . Zbytek je jednoduchý.  $\square$

Nás zajímá  $\log_2 |H|$ , tedy kolik bitů potřebujeme od pseudonáhodného generátoru na určení náhodné hašovací funkce. Zjistili jsme, že potřebujeme nejméně  $\log_2 m + \log_2 \lceil (\log_m N) - 1 \rceil - \log_2 c$  bitů.

#### Příklad malého $c$ -univerzálního systému

My známe  $c$ -univerzální systém velikosti  $N^2$ , tedy  $\log_2 |H| = 2 \log_2 N$ , což je hodně velké proti právě spočítanému dolnímu odhadu. Nyní zkonstruujeme  $c$ -univerzální hašovací systém, který tento dolní odhad v jistém smyslu nabývá.

Buď  $p_1, p_2 \dots$  rostoucí posloupnost všech prvočísel. Z teorie čísel bychom si měli pamatovat, že  $p_t = O(t \log t)$ .

Zvolíme nejmenší  $t$  takové, že

$$t \ln p_t \geq m \ln N \tag{4.4}$$

Definujme

$$h_{c,d,l}(x) = (c(x \bmod p_l) + d) \bmod p_{2t} \bmod m \tag{4.5}$$

$$H = \{h_{c,d,l} : c, d \in \{0 \dots p_{2t} - 1\}, t < l \leq 2t\}, \tag{4.6}$$

<sup>†</sup> vysvětlit

pak  $|H| = tp_{2t}^2$ , a tedy  $\log_2 |H| = \log_2 t + 2 \log_2 p_{2t} = O(\log t + 2 \log 2t + 2 \log \log 2t) = O(\log t) = O(\log m + \log \log N)$ , čímž jsme se dostali na dolní hranici odvozenou výše.

Dokážeme, že  $H$  je 5-univerzální systém.

Zvolme  $x \neq y \in U$ , spočteme odhad  $|\{h \in H : h_{c,d,l}(x) = h_{c,d,l}(y)\}|$ , tedy musíme odhadnout ze shora počet trojic  $c, d, l$  takových, že  $h_{c,d,l}(x) = h_{c,d,l}(y)$ . Rozdělíme je do dvou skupin:

1.  $c, d, l$  taková, že  $h_{c,d,l}(x) = h_{c,d,l}(y)$ , ale  $x \bmod p_l \neq y \bmod p_l$
2.  $c, d, l$  taková, že  $h_{c,d,l}(x) = h_{c,d,l}(y)$ , a  $x \bmod p_l = y \bmod p_l$

1) Platí

$$\begin{aligned} c(x \bmod p_l) + d &= k + qm \pmod{p_{2t}} \\ c(y \bmod p_l) + d &= k + rm \pmod{p_{2t}} \end{aligned}$$

pro nějaká  $k \in \{0 \dots m-1\}$ ,  $q, r \in \{0 \dots \lceil \frac{p_{2t}}{m} \rceil - 1\}$ . Protože  $p_l$  je prvočíslo, je počet trojic splňujících (1) roven

$$\begin{aligned} \text{počet trojic} &\leq tm \left\lceil \frac{p_{2t}}{m} \right\rceil^2 && \#l \leq t, \#(c, d) = \#(k, q, r) \\ &\leq tm \left(1 + \frac{p_{2t}}{m}\right)^2 \\ &= tm \frac{p_{2t}^2}{m^2} \left(1 + \frac{m}{p_{2t}}\right)^2 && \text{vytknutím} \\ &= \left(1 + \frac{m}{p_{2t}}\right)^2 \frac{|H|}{m} \\ &\leq 4 \frac{|H|}{m} && \text{jestliže } m \leq p_{2t} \end{aligned}$$

Ještě tedy musíme ukázat, že  $m < p_{2t}$ . Kdyby ale  $p_{2t} \leq m$ , pak dostaneme tento spor:  $t \ln p_t < p_{2t} \ln p_{2t} \leq m \ln m \leq m \ln N \leq t \ln p_t$ .

**Poznámka 4.1.1.**  $\forall(k, q, r) \exists!(c, d)$ , které řeší rovnici, jelikož  $\mathbb{Z}_{p_{2t}}$  je těleso a  $x \bmod p_l \neq y \bmod p_l$ .

2) Nechť  $L = \{l : x \bmod p_l = y \bmod p_l \wedge t < l \leq 2t\}$ . Pak počet trojic splňujících (2) je roven

$$\begin{aligned} \text{počet trojic} &= |L| p_{2t}^2 \\ &\leq \frac{tp_{2t}}{m} && \text{jestliže } |L| \leq t/m \\ &= 1 \frac{|H|}{m} \end{aligned}$$

Pokud tedy ještě dokážeme, že  $|L| \leq t/m$ , pak  $|\{h \in H : h_{c,d,l}(x) = h_{c,d,l}(y)\}| \leq 4 \frac{|H|}{m} + \frac{|H|}{m} = 5 \frac{|H|}{m}$  a  $H$  je 5-univerzální systém.

Nechť  $P = \prod_{l \in L} p_l$ . Z definice  $L$  všechna  $p_l$  dělí  $|x - y|$ , tedy i  $P$  dělí  $|x - y|$ , a proto  $P \leq |x - y| \leq N$ . Protože  $P \geq p_t^{|L|}$ , dostaneme  $|L| \leq \ln N / \ln p_t$ , a z definice  $t$  (4.4) plyne  $|L| \leq t/m$ .

### 4.1.3 Reprezentace a (MEMBER), INSERT, DELETE

Máme  $m$  a pro všechna  $i = 0, 1, \dots$  je dán  $c_i$ -univerzální systém funkcí  $H_i$  hašující do tabulky velikosti  $2^i m$ . Reprezentace  $S \subseteq U$ :

- $|S|$
- $i$  takové, že  $2^{i-2} m < |S| < 2^i m$

- funkce  $h \in H_i$
- reprezentace  $S$  vůči  $h_i$
- $\forall j \in \{0..2^i m - 1\}$  je dána délka řetězce reprezentujícího prvky s  $h(x) = j$
- konstanty  $d_i$  omezující délky řetězce

MEMBER normálně.

INSERT:

1. zjistíme, zda máme přidat do  $S$
2. když délka  $j$ -tého řetězce  $+1 > d_i$ ,  
pak spočítáme novou reprezentaci
3. když  $|S| + 1 = 2^i m$ ,  
pak inkrementujeme  $i$  a spočítáme novou reprezentaci
4. jinak přidáme prvek do řetězce  $h(x)$

DELETE:

1. zjistíme, zda  $x \in S$
2. když  $x \in S$  a  $|S| - 1 = 2^{i-2} m$  a  $i > 0$ ,  
pak dekrementujeme  $i$  a spočítáme novou reprezentaci
3. jinak  $x$  odstraníme z  $h(x)$

Spočítání nové reprezentace:

1. loop
2. zvolíme náhodně  $h \in H_i$
3. spočítáme reprezentaci  $S$  vůči  $h$
4. until všechny řetězce mají délku  $\leq d_i$

Kolikrát proběhne ten cyklus? Závisí to na více parametrech a Koubek to nikde uspokojivě spočítané neviděl.

## 4.2 Perfektní hašování

Perfektním hašováním myslíme úlohu nalézt pro danou pevnou množinu  $S \subseteq U, |S| = n$  perfektní hašovací funkci, tj. funkci, která nemá na množině  $S$  kolize. Tato úloha nepřipouští přirozenou implementaci operace INSERT, protože přidaný prvek může způsobit kolizi. Typický příklad použití je tabulka klíčových slov kompilátoru.

**Definice 4.2.1.** Funkce  $h : U \rightarrow \{0 .. m - 1\}$  je perfektní pro  $S$ , když je na  $S$  prostá ( $\forall x \neq y \in S$  platí  $h(x) \neq h(y)$ ).

Za jakých podmínek lze povolit INSERT? Musí být málo pravděpodobný. Prvky navíc se dávají jinam a po jisté době se vše přepočítá do jedné tabulky pro novou perfektní hašovací funkci.

Požadavky na hledanou hašovací funkci:

1.  $h$  je perfektní na  $S$
2.  $\forall x$  je  $h(x)$  rychle spočítatelná
3.  $m$  řádově srovnatelné s  $n$
4. zakódování  $h$  vyžaduje málo prostoru

Požadavky 2) a 3) jdou proti sobě. A až se nám je podaří skloubit, budeme mít problémy s 4). A navíc hledání  $h$  potrvá dlouho.



### 4.2.1 Perfektní hašovací funkce do tabulky velikosti $n^2$

Využijeme, co už víme o univerzálním hašování. Pro  $k \in \{1 \dots N-1\}$  a pro pevné  $m$  definujeme

$$h_k(x) = (kx \bmod N) \bmod m, \quad \text{kde } N = |U| \text{ je prvočíslo.} \quad (4.7)$$

Budeme hledat vhodná  $k, m$ . Definujeme míru perfektnosti

$$d = \sum_{k=1}^{N-1} \sum_{x \neq y \in S} \delta_{h_k(x), y} \quad (4.8)$$

a pro  $k \in \{1 \dots N-1\}$  položme

$$b_k(i) = |\{x \in S : h_k(x) = i\}| \quad (4.9)$$

Jednak platí

$$d = \sum_{k=1}^{N-1} \left( \sum_{i=0}^{m-1} (b_k(i))^2 - n \right) \quad (4.10)$$

a také

$$d = \sum_{x \neq y \in S} |\{k : h_k(x) = h_k(y)\}| \quad \text{prohozením sum} \quad (4.11)$$

$$(4.12)$$

Co znamená  $h_k(x) = h_k(y)$  pro  $x \neq y$ ? Následující tvrzení jsou ekvivalentní:

$$\begin{aligned} kx \bmod N &= ky \bmod N && (\bmod m) \\ k(x-y) \bmod N &= 0 && (\bmod m) \\ k(x-y) \bmod N &= rm && \text{pro } r \in \{-\lfloor N/m \rfloor \dots \lfloor N/m \rfloor\} - \{0\}, \end{aligned}$$

tedy

$$d \leq \sum_{x \neq y \in S} 2 \frac{N}{m} = \frac{2n(n-1)N}{m}$$

a dosazením do (4.10), podle přihrádkového principu

$$\exists k : \sum_{i=0}^{m-1} (b_k(i))^2 \leq n + \frac{2n(n-1)}{m} \quad (4.13)$$

Pro speciální velikosti tabulky dostáváme dosazením do (4.13):

$$\text{Pro } m = n : \quad \exists k \text{ nalezitelné v čase } O(nN) : \sum_{i=0}^{m-1} (b_k(i))^2 < 3n \quad (4.14)$$

$$\text{Pro } m = 1 + n(n-1) : \quad \exists k \text{ nalezitelné v čase } O(nN) : h_k \text{ je perfektní} \quad (4.15)$$

*Důkaz.* Probíráme všechny možnosti pro  $k$ , těch je  $O(N)$ .

(4.14) Pro dané  $k$  spočítáme  $\sum (b_k(i))^2$  v čase  $O(n) = O(m)$ .

(4.15)  $\sum (b_k(i))^2 \leq n + \frac{2n(n-1)}{1+n(n-1)} < n + 2$ . Kdyby  $h_k$  nebyla perfektní, pak  $\exists j : b_k(j) \geq 2$  a  $\sum (b_k(i))^2 \geq (n-2)1^2 + 1 \cdot 2^2 = n+2$ , spor. Při hledání  $k$  ověříme perfektnost  $h_k$  v čase  $O(n)$ .

□

Nyní máme perfektní hašovací funkci, která ale porušuje požadavek (3).

## 4.2.2 Perfektní hašovací funkce do tabulky velikosti $3n$

Zkombinujeme oba výsledky z předchozí části.

Podle (4.14) nalezneme  $k$  takové, že  $\sum (b_k(i))^2 < 3n$ .

Pro každé  $i \in \{0 \dots n-1\}$  vezmeme množinu kolidujících prvků  $S_i = \{s \in S : h_k(s) = i\}$ .

Označme  $n_i = |S_i|$ .

Podle (4.15) pro každé  $i$  nalezneme  $k_i$  takové, že pro  $m_i = 1 + n_i(n_i - 1)$  je  $h_{k_i}$  perfektní pro  $S_i$ .

Každou zahašovanou množinu  $S_i$  uložíme ve výsledné tabulce od pozice  $d_i$ :

$$d_i = \sum_{j=0}^{i-1} (1 + n_j(n_j - 1)).$$

Konečně definujeme

$$g(x) = d_i + h_{k_i}(x), \quad \text{kde } i = h_k(x),$$

která je perfektní a velikost tabulky je

$$m = d_n = \sum_{j=0}^{n-1} (1 + n_j(n_j - 1)) \leq \sum_{j=0}^{n-1} n_j^2 = \sum_{j=0}^{n-1} (b_k(j))^2 < 3n$$

Ovšem na zakódování této funkce potřebujeme hodně paměti: nevdává nám  $d_i$ , ale  $k$  a každé  $k_i$  je velikosti  $O(N)$ , tedy potřebujeme  $n \log_2 N$  bitů, což odporuje našemu požadavku (4). V dalších krocích budeme zmenšovat čísla definující hašovací funkci.

### Podobná funkce daná číslem velikosti $O(N)$

Zvolme prvočíslo  $p_1$  takové, že  $1 + n(n-1) \leq p_1 \leq 1 + 2n(n-1)$ . Nějaké takové musí existovat (Bertrandův postulát:  $\forall n > 1 \exists$  prvočíslo  $p$ , že  $n < p < 2n$ ). Podle (4.15)  $\exists k : h_k(x) = (kx \bmod N) \bmod p_1$  je perfektní na  $S$ .

Vytvoříme

$$S_1 = \{h_k(s) : s \in S\} \subset \{0 \dots p_1 - 1\}$$

a na  $S_1$  aplikujeme předchozí sekci, kde  $N = p_1$ .

Dostáváme hašovací funkci  $g_1$ , která

- je perfektní pro  $S$
- je spočítatelná v čase  $O(1)$
- hašuje do tabulky  $< 3n$
- je určena 1 číslem velikosti  $O(N)$   
a  $O(n)$  čísly velikosti  $O(n^2)$

### Podobná funkce daná číslem velikosti $O(n^2 \log N)$

Pro extrémní případy typu  $N = 2^{10^6}$  ještě postup vylepšíme, čímž zmenšíme velikost čísel kódujících perfektní hašovací funkci na  $O(\log N)$ .

**Lemma 4.2.1.** Pro každou množinu  $S \subseteq \{0 \dots N-1\}$  velikosti  $n$  existuje prvočíslo  $p$  takové, že  $f_p(x) = x \bmod p$  je perfektní na  $S$  a  $p = O(n^2 \log N)$ .

Využití: pro  $S$  najdeme prvočíslo  $p_0$  velikosti  $O(n^2 \log N)$  takové, že  $f_{p_0}$  je perfektní na  $S$ . Vytvoříme

$$S_0 = \{f_{p_0}(s) : s \in S\} \subset \{0 \dots p_0 - 1\}$$

a na  $S_0$  aplikujeme předchozí postup, kde  $N = p_0$ .

Tedy pro každou množinu  $S$  velikosti  $n$  existuje hašovací funkce  $f$ , která

lepší jména  
proměnných!

- je perfektní pro  $S$
- je spočitatelná v čase  $O(1)$
- hašuje do tabulky  $< 3n$
- je určena 2 čísly velikosti  $O(n^2 \log N)$   
a  $O(n)$  čísly velikosti  $O(n^2)$

**Lemma 4.2.2.** *Nechť  $r$  je číslo a  $p_1, \dots, p_q$  jsou všechny jeho prvočíselné dělitele. Pak  $q = O(\log r / \log \log r)$ .*

*Důkaz.*

$$\begin{aligned}
 r &\geq \prod_{i=1}^q p_i \\
 &> q! \\
 &= \exp\left(\sum_{i=1}^q \ln i\right) \\
 &> \exp\left(\int_1^q \ln x \, dx\right) \\
 &\geq \left(\frac{q}{e}\right)^q \quad \text{kde } \exp(x) = e^x
 \end{aligned}$$

Tedy

$$q \leq c \frac{\ln r}{\ln \ln r} \quad \text{pro vhodnou konstantu } c.$$

skok

□

*Důkaz lemmatu 4.2.1.* Předpokládejme  $S = \{x_1 < \dots < x_n\}$ . Hašovací funkce  $f_t(x) = x \bmod t$  je perfektní právě když  $t$  je nesoudělné s číslem

$$D = \prod_{i>j} (x_i - x_j) < N^{n^2}$$

Podle 4.2.2 je mezi prvními  $(c \ln D / \ln \ln D) + 1$  prvočísla alespoň jedno, které nedělí  $D$ . Víme, že  $p_k = O(k \ln k)$ , tedy  $(c \ln D / \ln \ln D) + 1$ -ní prvočísla má velikost  $O(\ln D) = O(n^2 \ln N)$ . □

nalezení  
prvocísla  $p_0$   
vyžaduje čas  
 $O(n^2 \log N)$ .

### 4.2.3 GPERF

Jiná konstrukce perfektní hašovací funkce je použita v programu `gperf`. Distribuován pod GPL. Jeho návrh je popsán v Douglas C. Schmidt, "GPERF: A Perfect Hash Function Generator," in Proceedings of the 2nd C++ Conference, San Francisco, California, April 1990, USENIX, pp. 87–102. Článek se dá stáhnout z <http://citeseer.nj.nec.com/schmidt90gperf.html>.

pořádnou  
bibliografii

# Kapitola 5

## Trie

### 5.1 Základní varianta

Trie je rovinná implementace slovníku. Máme abecedu  $\Sigma$  velikosti  $k$ . Universum jsou všechna slova nad  $\Sigma$  délky právě  $l$  (nekonečnou množinu si nemůžeme dovolit a kratší slova doplníme zprava mezerami). Chceme reprezentovat množinu slov  $S \subseteq U$ .

**Definice 5.1.1.** *Trie* nad  $\Sigma$  je konečný strom, jehož každý vnitřní vrchol má právě  $k$  synů, které jsou jednoznačně ohodnoceny prvky  $\Sigma$ . Každému vnitřnímu vrcholu trie odpovídá slovo nad  $\Sigma$  délky nejvýše  $l$ : kořenu odpovídá prázdné slovo  $\Lambda$ ; když vrcholu  $v$  odpovídá slovo  $\alpha$ , pak  $v[a]$ , synu  $v$  ohodnocenému písmenem  $a$ , odpovídá slovo  $\alpha a$ .

**Definice 5.1.2.** Řekneme, že trie nad  $\Sigma$  *reprezentuje množinu*  $S$ , když:

- Listům je přiřazena boolovská funkce náležení  $\text{Nal}$ :  $\text{Nal}(t)$  je true právě když slovo, které odpovídá listu  $t$ , je v  $S$ .
- Když  $v$  je vnitřní vrchol trie odpovídající slovu  $\alpha$ , pak existuje  $\beta \in S$  takové, že  $\alpha$  je prefix  $\beta$ .
- Pro každé slovo  $\alpha \in S$  existuje v trie list odpovídající  $\alpha$ .

#### 5.1.1 Algoritmus MEMBER

```
{vyhledání  $x = x_1 \dots x_l$ }  
 $t :=$  kořen  
 $i := 1$   
while  $t$  není list do  
   $t := t[x_i]$   
   $i := i + 1$   
end while  
{test}  
return  $\text{Nal}(t)$ 
```

#### 5.1.2 Algoritmus INSERT

```
{vyhledej  $x$ }  
if not  $\text{Nal}(t)$  then {trie nemusí být tak hluboké, jak potřebujeme}  
  while  $i \leq l$  do  
    vrcholu  $t$  přidej  $k$  listů ohodnocených písmeny z  $\Sigma$ , jejich  $\text{Nal} := \text{false}$   
     $t := t[x_i]$   
     $i := i + 1$ 
```

Obrázek 5.1: Nekomprimované trie

```

end while
Nal( $t$ ) := true
end if

```

### 5.1.3 Algoritmus DELETE

```

{vyhledej  $x$ }
if Nal( $t$ ) then
  Nal( $t$ ) := false
   $t$  := otec  $t$ 
  {opravíme prefixovou podmínku}
  while všichni synové  $t$  jsou listy s Nal = false do
    zruš listy  $t$ 
    Nal( $t$ ) := false
     $t$  := otec  $t$ 
  end while
end if

```

Použili jsme obrat  $t := \text{otec } t$ . To lze provést buď tak, že se vrchol kromě svých synů odkazuje i na svého otce a spotřebuje tak paměť navíc, nebo se cesta z kořene do aktuálního vrcholu během sestupu ve stromu pamatuje na zásobníku. Tento trik se používá u všech stromových struktur.

### 5.1.4 Časová a paměťová složitost

Jedna iterace cyklu zabere konstantní čas. Čas pro MEMBER je  $O(l)$ , čas pro INSERT a DELETE je  $O(lk)$ . Paměťová složitost trie v nejhorsím případě je počet uložených slov násobený délkou cesty a počtem synů, tedy  $O(|S|lk)$ .

**Poznámka 5.1.1.** V případě, kdy  $S$  obsahuje (skoro) všechna slova délky  $l$ , tak může mít složitost jen  $O(|S|)$ .

## 5.2 Komprimované trie

Mějme  $\Sigma = \{0, 1, 2\}$ ,  $l = 7$ .  $S = \{0202011, 0202012, 0202021, 1212102, 1212111, 1212121, 1212122\}$ . Nekomprimované trie pro tuto množinu je na obrázku 5.1. Vidíme, že písmena na druhé až páté pozici jsou vždy stejná a předchází algoritmy se jimi musí „prokousat“. Přesně řečeno, prohlížení vrcholu  $v$ , který má jediného syna, který není list s hodnotou  $\text{Nal} = \text{false}$ , nepřináší žádnou kladnou informaci, protože množiny prvků z  $S$ , které jsou reprezentovány vrcholy v podstromu otce  $v$  a v podstromu vrcholu  $v$  jsou stejné. To vedlo k ideji tyto vrcholy ze stromu vynechat a tím zmenšit (komprimovat) trie.

Ke každému vrcholu  $v$  přidáme funkci  $\text{uroven}(v)$  vyjadřující číslo úrovně, ve které se  $v$  nachází v původním trie. Ke každému listu  $v$  přidáme funkci  $\text{slovo}(v)$  — slovo, které odpovídá  $v$ .

Nyní můžeme vynechávat vrcholy podle následujícího kritéria: je-li  $v$  vnitřní vrchol a všichni jeho synové kromě  $w$  jsou listy s  $\text{Nal} = \text{false}$ , pak  $v$  vynech a zařaď  $w$  na jeho místo. Tento proces opakujeme dokud trie obsahuje nějaký vnitřní vrchol, jehož všichni synové s výjimkou jednoho jsou listy, pro něž  $\text{Nal} = \text{false}$ . Všimněte si, že každý vnitřní vrchol má právě  $k$  synů, které jsou v jednoznačné korespondenci s písmeny abecedy  $\Sigma$ .

**Poznámka 5.2.1.** Komprimované trie je tvořený množinou trie, kde pro  $\beta$  je  $\text{hladina}(\beta) = |\beta|$  a otec  $\beta$  je největší vlastní prefix, který patří do trie + přidané listy. Listy jsou prvky z  $S$  + slova  $\beta a$ , kde  $\beta \in \text{trie}$  a  $\beta a$  není prefixem žádného slova v  $S$ . Pro prvky z  $S$  je  $\text{Nal} = \text{True}$ , jinak false. Platí  $\text{prvek}(\gamma) = \gamma$  pro každý list.

Když  $\beta \in \text{trie}$  a  $a \in \Sigma \rightarrow \begin{cases} \beta a \text{ list,} & \text{je } a\text{-tým synem } \beta \\ \exists \delta \in S, & \text{že } \beta a \text{ je prefixem } \delta \end{cases}$   
 Potom  $a$ -tý syn  $\beta$  je nejkratší prefix v množině trie v  $S$ , který obsahuje  $\beta a$ .

### 5.2.1 MEMBER

Viz algoritmus 5.1

---

**Algoritmus 5.1** MEMBER pro komprimované trie

---

```
{vyhledání  $x = x_1 \dots x_l$ }
 $t :=$  kořen
while  $t$  není list do
   $i :=$  uroven( $t$ ) + 1
   $t := t[x_i]$  //  $x_i$ -tý list
end while
{test}
return  $\text{Nal}(t) \wedge \text{slovo}(t) = x$ 
```

---

zde

něco chybí

### 5.2.2 INSERT

Viz algoritmus 5.2

### 5.2.3 DELETE

Viz algoritmus 5.3

### 5.2.4 Časová a paměťová složitost

Paměťová složitost takto komprimovaných trie je  $O(nk)$ , kde  $n$  je velikost reprezentované množiny. (maximálně  $n - 1$  vnitřních vrcholů, každý s polem délky  $k$ ). Časová složitost operace MEMBER je v nejhorším případě  $O(l)$ , pro INSERT a DELETE je to  $O(l + k)$ . (může být nutné přidat/odebrat jeden vnitřní vrchol).

V průměrném případě (za předpokladu rovnoměrného rozložení vstupních dat) je to očekávaná hloubka trie. Tu teď spočítáme.

Nechť

$$q_d = \mathcal{P}(\text{trie má hloubku alespoň } d)$$

Očekávaná hloubka trie reprezentující  $n$  slov je

$$E_n = \sum_{d=0}^{\infty} d(q_d - q_{d+1}) = \sum_{d=0}^{\infty} q_d$$

Když funkce  $\text{pref}_{d-1}$ , přiřazující slovu  $\alpha$  jeho prefix délky  $d - 1$ , je na množině  $S$  prostá, pak trie reprezentující množinu  $S$  má hloubku nejvýše  $d$ . Spočítáme počet množin o velikosti  $n$ , na nichž je funkce  $\text{pref}_{d-1}$  prostá. Tyto množiny získáme tak, že vybereme  $n$  prefixů délky  $d - 1$  a každý doplníme všemi sufiky délky  $l - d + 1$ . Proto těchto množin je

$$\binom{k^{d-1}}{n} k^{n(l-d+1)}.$$

---

**Algoritmus 5.2** INSERT pro komprimované trie

---

```

{vyhledej  $x$ }
if  $\text{Nal}(t) \wedge \text{slovo}(t) = x$  then
  {Trie už obsahuje  $x$ , nedělej nic.}
else
  if  $\text{slovo}(t) = x$  then
    {Trie obsahuje správný list, pouze nastav příznak. Např. "0202010"}
     $\text{Nal}(t) := \text{true}$ 
  else
    {Bude potřeba vložit nový list.}
    {Najdi, kam ho připojit.}
     $\alpha :=$  nejdelší společný prefix slov  $x$  a  $\text{slovo}(t)$ . Délku  $\alpha$  označme  $|\alpha|$ .
     $v :=$  vrchol na cestě z kořene do  $t$  takový, že  $\text{uroven}(v)$  je největší, která je  $\leq |\alpha|$ 
    if  $\text{uroven}(v) = |\alpha|$  then
      { $v$  je otec nového listu}
    else { $\text{uroven}(v) < |\alpha|$ }
      {Bude potřeba vytvořit otce nového listu}
       $a := \text{uroven}(v) + 1$ -ní písmeno  $\alpha$ 
       $u := v[a]$ 
      {Mezi  $v$  a  $u$  vytvoř nový vnitřní vrchol odpovídající slovu  $\alpha$ }
       $w :=$  nový vrchol,  $\text{uroven}(w) := |\alpha|$ 
       $v[a] := w$ 
       $c := |\alpha| + 1$ -ní písmeno  $\text{slovo}(t)$ 
       $w[c] := u$ 
      for all  $b \in \Sigma, b \neq c$  do
         $z :=$  nový vrchol,  $\text{uroven}(z) := |\alpha| + 1, \text{Nal}(z) := \text{false}, \text{slovo}(z) := \alpha b,$ 
         $w[b] := z$ 
      end for
      end if
       $v := w$ 
    end if
    {Správnému listu přiřaď  $x$ }
     $d := |\alpha| + 1$ -ní písmeno  $x$ 
     $s := v[d]$ 
     $\text{uroven}(s) := l, \text{Nal}(s) := \text{true}, \text{slovo}(s) := x$ 
  end if
end if

```

---

**Algoritmus 5.3** DELETE pro komprimované trie

---

```

{vyhledej x}
if Nal(t)  $\wedge$  slovo(t) = x then
  u := otec t
  i := uroven(u)
  Nal(t) := false
  uroven(t) := i + 1, slovo(t) := prefix slova x délky i + 1
  {vrchol u má alespoň jednoho syna, který není list s Nal = false}
  if všichni synové u kromě syna w jsou listy s Nal = false then
    v := otec u
    smaž u a všechny syny u kromě w
    j := uroven(v) + 1
    v[xj] := w // xj-tý syn v je w
  end if
end if

```

---

Protože všech podmnožin velikosti  $n$  je  $\binom{k^l}{n}$  dostáváme, že

$$\begin{aligned}
q_d &\leq 1 - \frac{\binom{k^{d-1}}{n} k^{n(l-d+1)}}{\binom{k^l}{n}} \Bigg\} \text{pravděpodobnost} \\
&\leq 1 - \frac{k^{d-1}(k^{d-1}-1)\dots(k^{d-1}-(n-1))k^{n(l-d+1)}}{k^{ln}} \\
&= 1 - \prod_{i=0}^{n-1} \left(1 - \frac{i}{k^{d-1}}\right) \\
&\leq 1 - \exp\left(\frac{-n^2}{k^{d-1}}\right) \\
&\leq \frac{n^2}{k^{d-1}},
\end{aligned}$$

poněvadž

$$\begin{aligned}
\prod_{i=0}^{n-1} \left(1 - \frac{i}{k^{d-1}}\right) &= \exp\left(\sum_{i=0}^{n-1} \ln\left(1 - \frac{i}{k^{d-1}}\right)\right) \\
&\geq \exp\left(\int_0^n \ln\left(1 - \frac{i}{k^{d-1}}\right)\right) \\
&= \exp\left(\frac{-n^2}{k^{d-1}}\right),
\end{aligned}$$

(užijte integrální kritérium a substituci  $x = k^{d-1}(1-t)$ ) a  $e^x - 1 \geq x$  (odtud  $1 - e^x \leq -x$ ). Tedy



pro  $c = 2\lceil \log_k n \rceil$  dostáváme

$$\begin{aligned}
 E_n &= \sum_{d=1}^c q_d + \sum_{d=c+1}^{\infty} q_d \\
 &\leq c + \sum_{d=c}^{\infty} \frac{n^2}{k^d} \\
 &\leq 2\lceil \log_k n \rceil + \left(\frac{n^2}{k^c}\right) \sum_{d=0}^{\infty} k^{-d} \\
 &\leq 2\lceil \log_k n \rceil + \frac{1}{1-1/k} \\
 &= 2\lceil \log_k n \rceil + \frac{k}{k-1}.
 \end{aligned}$$

Tedy očekávaný čas operace MEMBER je  $O(\log_k(n))$  ( $O(\frac{\log n}{\log k})$ ) a  $O(\log_k(n) + k)$  pro INSERT a DELETE pro komprimované trie (za předpokladů rovnoměrného rozložení vstupních dat) Zde parametr  $k$  vyjadřuje vztah mezi prostorovými a časovými nároky.

L.Prošek:  
Možná v té očekávané složitosti by šlo +k zanedbat, ale ne na základě toho tvrzení, které dokazuje jen očekávanou hloubku

---

#### Algoritmus 5.4 INSERT pro komprimované trie, analogie 5.2 (verze Koubek 2002)

---

```

INSERT( $x = x_1, \dots, x_l$ )
 $t \leftarrow$  kořen
while  $t$  není list do
     $i \leftarrow$  hladina( $t$ ),  $t \leftarrow$  ( $a_{i+1}$ )-ní syn  $t$ 
end while
if prvek( $t$ ) není prefix  $x$  then
     $\beta =$  největší společný prefix  $x$  a prvek( $t$ )
     $\beta a =$  prefix  $\alpha$ 
     $\beta b =$  prefix prvek( $t$ )
    while hladina( $t$ ) >  $|\beta a|$  do  $t \leftarrow$  otec( $t$ ) done
    if hladina( $t$ ) <  $|\beta|$  then
        vytvoříme nový vrchol  $w$ , jehož synové, kromě  $b$ -tého syna budou listy s
        funkcemi Nal = false
        prvek( $t$ ) =  $\beta$  + označení syna
        hladina( $w$ ) =  $|\beta|$ ,  $\beta = (a_1, \dots, a_i)$ 
        necht  $v = a_{\text{hladina}(t)+1}$  - tý syn  $t$ ,  $b$ -tý syn  $w$  je  $v$ 
         $w = a_{\text{hladina}(t)+1}$ -tý syn  $t$ 
    end if
     $z \leftarrow$   $a$ -tý syn  $t$ , Nal( $z$ ) = true, prvek( $z$ ) =  $x$ 
else
    Nal( $t$ ) = true, prvek( $t$ ) =  $x$ 
end if

```

---

XXX další  
neznámý  
algoritmus z  
přednásky 2002

### 5.3 Ještě komprimovanější trie

Př:

mějme následující komprimovaný trie:

a jeho matici:

tady chybí  
obrázek

**Algoritmus 5.5 DELETE** pro komprimované trie (?)DELETE( $x = x_1, \dots, x_l$ )

t ← kořen

**while** t není list **do**i ← hladina(t), t ← ( $a_{i+1}$ -ní syn t**end while****if** Nal(t) = true a prvek(t) = j **then**

Nal(t) = false

v ← otec(t)

prvek(t) ← prefix prvek(t) o délce hladina v+1

**if** vsichni synové vrcholu v až na jednoho jsou listy s Nal = false **then**

w ← syn(v), který je buď list s Nal(w) = true nebo není list

necht v je a-tý ( $a_i$ -ty ???) syn svého otce, v smažeme a smažemevšechny syny  $v \neq w$ w ← a-tý ( $a_i$ -tý ???) syn otce v**end if****end if**

	0	1	2
root	NIL	a	b
a	102	NIL	c
b	210	211	212
c	120	121	NIL

Chceme se zbavit položek NIL v matici reprezentující trie. Další komprese dosáhneme pomocí vektorů hod (vektor hodnot) a rd. Tyto vektory budou reprezentovat původní matici.

co znamená rd ?

**5.3.1 Popis A a rd**

Zpět k našemu příkladu:

I.

hod	210	211	212	120	121	NIL
rd	root	a	b	c		
			0	3		

II.

hod	210	211	212	120	121	a	b	102	NIL	c
rd	root	a	b	c						
	4	7	0	3						

Řádek i začíná na místě rd(i) a musí být splněna podmínka:

Když  $M_{i,j} \neq NIL \neq M_{i',j'}$ , pak  $rd(i) + j \neq rd(i') + j'$ Když na místě hod chceme zapsat prvek  $\neq NIL$  a NIL, pak zapíšeme prvek  $\neq NIL$ .**5.3.2 Algoritmus pro hledání rd a hod**Nechť M je matice typu r x s, má m významných míst  $\neq NIL$ .

- pro každý řádek nalezneme počet míst  $\neq NIL$
- setřídíme řádky Bucketsortem, tak že řádky s větším počtem míst  $\neq NIL$  předcházejí řádky s menším počtem míst  $\neq NIL$
- procházíme řádky v daném setřídění a pro každý řádek i nalezneme nejmenší číslo rd(i), že nedochází ke kolizi s předchozími řádky (tj. když  $M_{i',j'} \neq NIL \neq M_{i,j}$ ) a řádek i' byl zařazen, pak  $rd(i) + j \neq rd(i') + j'$ . Pak  $M_{i,j} \neq NIL$  je uloženo ve vektoru hod na místě rd(i)+j.

$m(l)$  - počet míst  $\neq NIL$  v řádcích s počtem míst  $\geq l + 1 \neq NIL$ .

**Věta 5.3.1.** *Když  $m(l)(l + 1) \leq m$  pro každé  $l$ , pak  $rd(i) < m$  pro každý řádek  $i$  a algoritmus vyžaduje čas  $O(rsm)$ .*

*Důkaz.* Předpokládejme, že hledáme  $rd$  pro řádek  $i$ , který má  $l$  míst  $\neq NIL$ .

ve vektoru  $hod$  je obsazeno méně než  $m(l-1)$  míst.

zkoušíme  $rd(i)=1,2,\dots$

$rd(i) = 1,2,\dots$  je zakázané, když vznikne kolize.

tj.  $\exists$  řádek  $i'$  předcházející a  $\exists j, j'$  takové, že  $M_{i',j'} \neq NIL \neq M_{i,j}$  a platilo by  $rd(i') + j' = rd(i) + j$ .

$\rightarrow$  těchto možností je  $< lm(l - 1) \leq m$ .

$O(rs)$  - zjistíme pro každý řádek počet míst  $\neq NIL$ .

$O(m+r)$  - třídění Bucketsortem

$O(mrs)$  - krok 2

□

Př: jedna možnost

$M =$

$\rightarrow$  budeme mít moc řádků - nevhodné

Př:

M	0	1	2
root	NIL	a	b
a	102	NIL	c
b	210	211	212
c	120	121	NIL

rd	root	a	b	c						
	4	7	0	3						
hod	210	211	212	120	121	a	b	102	NIL	c

M'	0	1	2
b	210	211	212
c	120	121	NIL
root	NIL	a	b
a	102	NIL	c

(přehodili jsme pouze řádky)

210	NIL	NIL
120	211	212
NIL	121	NIL
102	a	b
NIL	NIL	c

### 5.3.3 Vertikální posun sloupců

$cd$  - vektor sloupcového posunutí, slouží k zápisu transformace

	0	1	2
cd	0	1	2

	0	1	2	3	4
rd	6	0	6	3	6

hod	120	211	212	102	a	b	210	121	c
-----	-----	-----	-----	-----	---	---	-----	-----	---

Jak najdeme nazpátek místa ? Platí, když  $M_{i,j} \neq NIL$ , pak  $hod(rd(i + cd(j) + j)) = M_{i,j}$

je ten vzorec správně ?

- $f(-,-)$  je fce dvou proměnných
- $B_j$  matice posunutých prvních sloupců
- $m_j$  počet míst  $\neq NIL$  v  $B_j$

- $m_j(l)$  počet míst  $\neq NIL$  v řádcích matice  $B_j$ , které mají aspoň  $l+1$  míst  $\neq NIL$

Budeme chtít, aby  $\forall j \forall l$  platilo  $m_j(l) \leq \frac{m}{f(l, m_j)}$ .

Okrajové podmínky na  $f$ :  $f$  musí splňovat:

- $\forall l$  platí  $f(l, m) \geq l + 1$
- $\forall j$  platí  $f(0, m_j) \leq \frac{m}{m_j}$

### Algoritmus na posunutí sloupců

1) pro každý sloupec v pořadí  $0, 1, \dots$  nalezneme nejmenší  $cd(j)$  takové, aby matice  $B_j$  splňovala  $\forall l$   $m_j(l) \leq \frac{m}{f(l, m_j)}$  (každý sloupec posunujeme dokud nesplňuje podmínku)

Na získanou matici  $B = B_s$  pak použijeme předchozí algoritmus.

Platí  $m(l) = m_s(l) \leq \frac{m}{f(l, m)} \leq \frac{m}{l+1}$ .

Hledáme hodnotu  $cd(j)$  a předp., že pro nějakou hodnotu  $cd(j)$  není splněna podmínka pro  $l$ , tj.

platí  $m_j(l) > \frac{m}{f(l, m)}$  ... platila pro  $B_{j-1}$ , tj.  $m_{j-1}(l) \leq \frac{m}{f(l, m_{j-1})}$

Z toho plyne  $m_j(l) - m_{j-1}(l) > \frac{m}{f(l, m_j)} - \frac{m}{f(l, m_{j-1})}$ .

Jak roste číslo  $m_j(l)$  ?

1. v matici  $B_{j-1}$  existuje řádek s aspoň  $l+1$  místy  $\neq NIL$  a s tímto řádkem se střetne místo  $\neq NIL$  (v  $j$ -tém sloupci  $\leftarrow m_{j-1}(l)$  vzroste o 1)
2. v matici  $B_{j-1}$  existuje řádek s  $l$  místy  $\neq NIL$  a s tímto řádkem se střetne místo  $\neq NIL$  v  $j$ -tém sloupci. Pak  $m_{j-1}(l)$  vzroste o  $l+1$ .

střet - řádek v  $B_{j-1}$  s aspoň  $l$  místy  $\neq NIL$  a místo  $\neq NIL$  v  $j$ -tém sloupci. Aby nebyla splněna podmínka pro  $l$ , musí být počet střetů pro danou hodnotu  $cd(j)$  být aspoň

$$\frac{\frac{m}{f(l, m_j)} - \frac{m}{f(l, m_{j-1})}}{l + 1}$$

V matici  $B_{j-1}$  je nejvýše  $\frac{m_{j-1}(l-1)}{l} \leq \frac{m}{lf(l-1, m_{j-1})}$  řádků s aspoň  $l$  místy  $\neq NIL$ , v  $j$ -tém sloupci je  $m_j - m_{j-1}$  míst  $\neq NIL$ .

Podmínka pro  $l$  může zakázat nejvýše

$$\frac{\frac{m(m_j - m_{j-1})}{lf(l-1, m_{j-1})}}{\frac{\frac{m}{f(l, m_j)} - \frac{m}{f(l, m_{j-1})}}{l+1}} \text{ hodnot } cd = \frac{l+1}{l} \frac{((m_j - m_{j-1}) \frac{f(l, m_{j-1})}{f(l, m_j)} - 1)}{\frac{f(l, m_{j-1})}{f(l, m_j)}} \quad (5.1)$$

Stačí nám sčítat přes hodnoty  $l$  takové, že

$mm_{j-1}(l) \leq l + 1$  tj. přes  $l \leq l_0 = \min\{l; \frac{m}{f(l, m_{j-1})} < l\}$ ,

$m_{j-1}(l) \leq \frac{m}{f(l, m_{j-1})} \leq l + 1$ .

Celkový počet zakázaných hodnot  $cd$  je menší než

$$\sum_{l=0}^{l_0} \frac{l+1}{l} \frac{(m_j - m_{j-1})}{\frac{f(l, m_{j-1})}{f(l, m_j)} - 1} \frac{f(l, m_{j-1})}{f(l-1, m_{j-1})} \quad (5.2)$$

Zvolíme  $f(l, m_j) = 2^{l(2 - \frac{m_j}{m})}$ .

**Poznámka 5.3.1.** Jelikož se  $f$  vyskytuje v sumě jen v podílech, výraz se zjednoduší, zvolíme-li  $f(l, m_j) = 2^{g(l, m_j)}$ , kde  $g$  je nějaká vhodná funkce. Dosadíme-li, můžeme si všimnout, že dostaneme v exponentech rozdíly  $g(l, m_{j-1}) - g(l, m_j)$  a  $g(l, m_{j-1}) - g(l-1, m_{j-1})$ , které vznikly vhodnou předchozí úpravou výrazu.

(... suma z Mehlhorna na straně 10, třetí suma od spoda ...)

Teď se lze zbavit  $-1$  ve jmenovateli použitím nerovnosti  $2^x - 1 = e^{x \ln 2} - 1 \geq x \ln 2$ .

(... suma z Mehlhorna na straně 10, druhá suma od spoda ...)

Dalším pozorováním zjistíme, že v takto získaných rozdílech se mění jenom jedna proměnná.

Výraz se dále zjednoduší, bude-li  $g(l, m_j) = h(l)k(m_j)$ , kde  $h(l), k(m_j)$  budou vhodné lineární funkce.

U funkce  $k$  linearitou využijeme rozdíl  $m_{j-1} - m_j$  v čitateli, který teď můžeme zkrátit.

(... suma z Mehlhorna na straně 10, první suma od spoda ...)

Dalšími heuristikami a s využitím okrajových podmínek pro  $f$  nakonec zjistíme, že dobrou volbou jsou funkce  $h(l) = l, k(m_j) = 2 - \frac{m_j}{m}$ .

Takto definovaná  $f$  splňuje okrajové podmínky:

$$f(l, m) = 2^l \geq l + 1 \quad \forall l = 0, 1, \dots$$

$$f(0, m_j) = 1 \leq \frac{m_j}{m} \quad \forall j = 0, 1, \dots, s$$

dosadíme do odhadu 5.2 a dostaneme

$$\sum_{l=1}^{l_0} \frac{l+1}{l} \frac{(m_j - m_{j-1})}{2^{l(\frac{m_j}{m} - \frac{m_{j-1}}{m})}} 2^{(2 - \frac{m_{j-1}}{m})} \leq$$

využijeme, že  $2^x - 1 \geq x \ln(2)$

$$\sum_{l=1}^{l_0} \frac{l+1}{l} \frac{(m_j - m_{j-1})}{l(\frac{m_j}{m} - \frac{m_{j-1}}{m})} 4 =$$

$$\frac{4m}{\ln(2)} \sum_{l=1}^{l_0} \frac{l+1}{l^2} = \frac{4m}{\ln(2)} \left( \sum_{l=1}^{l_0} \frac{1}{l} + \sum_{l=1}^{l_0} \frac{1}{l^2} \right) \leq$$

integrální kritérium

$$\frac{4m}{\ln(2)} (1 + \ln(l_0)) + \frac{\pi^2}{6} \leq 4m \log_2(l_0) + 15.3m$$

$$\text{odhadneme } l_0: l_0 = \min\left\{l; \frac{m}{f(l, m_{j-1})} < l\right\} \rightarrow l_0 < \log(m)$$

$$\text{pak } \leq 4m \log \log m + 15.3m \quad (5.3)$$

Celý algoritmus spočítá uložení matice  $M$  typu  $r \times s$  do vektorů

$cd$  - dimenze  $s$ ,

$rd$  - dimenze  $4m \log \log m + 15.3m + r$ ,

$hod$  dimenze  $m + s$ ,

přitom hodnoty  $cd(j) < 4m \log \log m + 15.3m$  a  $rd(i) < m$ .

Čas potřebný k výpočtu je  $O(sr(m \log \log(m))^2)$ , kde  $m$  je počet míst  $\neq NIL$  v matici  $M$ .

### 5.3.4 Úsporné uložení řídkého vektoru

Máme vektor v dimenze  $n \cdot d$  (rozdělený na  $n$  bloků velikosti  $d$ ) a  $i_0 < i_1 < \dots < i_{t-1}$  jsou všechny indexy  $i$  takové, že  $v(i) \neq 0$ .

Vytvoříme vektor  $cv$  dimenze  $t$ ,  $cv(j) = v(i_j)$ .

Náš úkol - pro dané  $l$  zjistit, zda  $l = i_j$  a případně nalézt toto  $j$ .

Sestavíme vektor  $base$  dimenze  $n$ :

$$\text{base}(j) = \begin{cases} -1 & i_k \text{ div } d \neq j \quad \forall k = 0, 1, \dots, t-1 \\ \min\{l; i_l \text{ div } d = j\} & \exists l, \text{ že } i_l \text{ div } d = j \end{cases}$$

a matici  $offset$  typu  $n \times d$

$$\text{offset}(j, k) = \begin{cases} -1 & i_l \neq jd + k \quad \forall l = 0, 1, \dots, t-1 \\ l - \text{base}(j) & i_l = jd + k \end{cases}$$

Nyní uložíme matici *offset* do vektoru *off* dimenze *n* tak, že z každého řádku vytvoříme číslo v soustavě o základu *d + 1*:

$$\text{off}(j) = \sum_{k=0}^{d-1} (\text{offset}(j, k) + 1)(d + 1)^k$$
  
 potřebujeme  $\text{base}(\text{dim } n)$ ,  $\text{off}(\text{dim } n)$   
 smysluplné když  $d \ll n$  a  $t < n$  (např.  $d = \log \log n$ )

Platí následující vztahy:

1.  $v(h) = 0 \leftrightarrow \text{offset}(h \div d, h \bmod d) = -1$
2.  $v(h) = 1 \rightarrow h = \text{base}(h \div d) + \text{offset}(h \div d, h \bmod d)$
3.  $\text{offset}(i, j) = \text{off}(i) \div (d + 1)^j \bmod (d + 1) - 1$

pro dané *i* - nalezení hodnoty  $v(i)$  když  $\text{base}(i \text{ div } d) = -1$ , pak  $v(i) = 0$   
 $\text{base}(i \text{ div } d) \neq -1$ , pak  $k = i \bmod d$   
 $j = i \text{ div } d$   
 $l = \text{off}(j) \text{ div } (d + 1)^k$   
 $l = l \bmod (d + 1)$   
 $l = l - 1 + \text{base}(j)$   
 $v(i) = cv(l)$

Lze použít pro malé *t* a  $(d + 1)^d$  v rozsahu velikosti registru - vhodné např. pro  $d \approx \log \log n$ .

Př.:  $v =$ 

0	1	0	1	0	1	0	0	0	1
0			1			-1			3

$i_0 = 1, i_1 = 3, i_2 = 5, i_3 = 11, d = 3$

$cv =$ 

$v(1)$	$v(3)$	$v(5)$	$v(11)$
--------	--------	--------	---------

$base =$ 

0	1	-1	3
---	---	----	---

offset	0	1	2	3
0	-1	0	-1	-1
1	0	-1	-1	-1
2	-1	1	-1	0

3. sloupec tabulky offset repr. nuly

$off =$ 

4	33	0	16
---	----	---	----

$\text{off}(7) = (\text{offset}(1, 0) + 1)4^0 + (\text{offset}(1, 1) + 1)4^1 + (\text{offset}(1, 2) + 1)4^2$   $\text{off}(1) = 1 + 0 + 2 \cdot 4^2 =$

# Kapitola 6

## Uspořádaná pole

### 6.1 Unární, binární a interpolační vyhledávání

Napsal Pavel  
Machek

Uspořádané pole je datová struktura, která vznikne z pole jeho setříděním. Jediná operace, která se na ní dá (rozumně rychle) provádět, je MEMBER.

Mějme slovník  $S$  uložený jako pole prvků tak, že  $s[i] < s[i + 1]$ .

**Algoritmus 6.1** MEMBER pro uspořádané pole

```

{vyhledání hodnoty  $x$  mezi  $s[i] \dots s[j]$ }
{odpověď ANO, když  $\exists h : i \leq h \leq j \wedge s[h] = x$ }
 $d := i$  {aktuální dolní a horní odhad}
 $h := j$ 
 $next := f(d, h)$  { Předpokládáme, že  $d \leq f(d, h) \leq h$  }
while  $s[next] \neq x \wedge d < h$  do
  if  $s[next] < x$  then
     $d := next + 1$ 
  else
     $h := next - 1$ 
  end if
   $next := f(d, h)$ 
end while
{řekni ANO pokud  $s[next] = x$ , jinak řekni ne}
    
```

Tento algoritmus může provádět unární, binární, nebo interpolační vyhledávání; stačí jen dodat správnou funkci  $f$ ; zobecněné kvadratické vyhledávání bude definováno dále:

metoda	odpovídající funkce	nejhorší př.	průměrný případ
unární vyhledávání	$f(d, h) = d$	$O(n)$	$O(n)$
binární vyhledávání	$f(d, h) = \lceil \frac{d+h}{2} \rceil$	$O(\log(n))$	$O(\log(n))$
interpolační vyhledávání	$f(d, h) = d + \lceil \frac{x-s[d]}{s[h]-s[d]} * (h - d + 1) \rceil$	$O(n)$	$O(\log(\log(n)))$
zobecněné kvadratické v. kvadratické vyhledávání	$f(d, h) = f_{kvadrat}$	$O(\log(n))$ $O(\log(n))$	$O(\log(\log(n)))$ $O(\log(\log(n)))$

Z těch zápisů,  
co mám, to  
opravdu  
vypadá, jako že  
zobecněné  
kvadratické a  
kvadratické jsou  
2 různé věci

### 6.2 Zobecněné kvadratické vyhledávání

Na interpolačním vyhledávání se nám líbí jeho čas  $O(\log \log |S|)$  v průměrném případě (při rovnoměrném rozdělení dat). Avšak jeho čas v nejhorším případě je až  $O(|S|)$ . Zato binární vyhledávání má čas nejvýše  $O(\log |S|)$ . Zobecněné kvadratické vyhledávání je tak trochu kombinace předchozích dvou vyhledávání.

algoritmus  
vychází z  
Pavlova, výklad  
napsal Jakub  
Černý

Jak zobecněné kvadratické vyhledávání funguje? Využívá funkci MEMBER s funkcí *fkvadrat* tak, jak byla popsána v předchozím odstavci. Tomu, že se zvolí hodnota *next* a podle ní se opraví hodnota *d* nebo *h*, budeme říkat, že se položí dotaz. Celé vyhledávání funguje tak, že se nejprve položí interpolační dotaz. To je vždy, když je *nastav* true. Položení dalších dotazů si můžeme představovat jako skoky z místa posledního dotazu ve směru, kde leží *x*. (Skočíme na nový index v poli).<sup>1</sup> Po interpolačním dotazu se neustále střídají skoky o  $\sqrt{\text{delka}}$  s binárními dotazy, až dokud nepřeskočíme *x*. (Toto střídání zajišťuje proměnná *parita*). Pak se znova položí interpolační dotaz a vše se opakuje.

---

**Algoritmus 6.2** Krok zobecněného kvadratického vyhledávání — *fkvadrat(d, h)*


---

```
{Proměnné nastav, parita a nahoru jsou statické, tj. jejich hodnoty se mezi voláními tohoto
algoritmu zachovávají.}
{Nechť nastav je na začátku true.}
{Dokud je nastav false (pracuje se v rámci bloku), je parita střídavě true (skok o  $\sqrt{\text{delka}}$ ) a false
(binární vyhledání)}
if nastav then
  parita := true
  delka :=  $h - d + 1$ 
  next :=  $d + \left\lceil \frac{x - s[d]}{s[h] - s[d]} \cdot \text{delka} \right\rceil$  {= finterp(d, h)}
  nahoru :=  $s[\text{next}] < x$ 
  nastav := false
  return next
end if
if not parita then
  next :=  $\lceil (d + h)/2 \rceil$  {= fbin(d, h)}
  parita := true
  return next
end if
next := nahoru ?  $d + \sqrt{\text{delka}}$  :  $h - \sqrt{\text{delka}}$ 
if  $s[\text{next}] < x$  xor nahoru then
  nastav := true
else
  parita := false
end if
return next
```

---

Jaký čas má vyhledávání v nejhorším případě? Rozdíl mezi *d* a *h* se během nejvýše 3 dotazů zmenší na polovinu. Proto je nejhorší čas  $O(\log n)$ .

Jaký čas má vyhledávání v průměrném případě? Tím myslíme při rovnoměrném rozložení dat. To už je malinko složitější otázka. Vyhledávání si rozdělíme do několika fází. Fáze začíná interpolačním dotazem a pokračuje až do dalšího interpolačního dotazu. Ukážeme, že v jedné fázi se položí v průměru jen konstantně dotazů. Pojdme tedy zanalyzovat jednu fázi. Souvislý úsek pole mezi pozicemi *d* a *h* na začátku fáze označme jako blok. Proměnná *delka* udává délku bloku a má hodnotu  $h - d + 1$ . Označme *X* náhodnou proměnnou, *X* = počet *i* na začátku bloku takových, že  $i \geq d$  a  $s[i] < x$ . Jinak řečeno *X* udává vzdálenost *x* od začátku bloku.

Položme  $p = \mathcal{P}(\text{náhodně zvolený prvek mezi } s[d] \text{ a } s[h] \text{ je menší než } x) = (x - s[d]) / (s[h] - s[d])$

$$\mathcal{P}(X = j) = \binom{h - d + 1}{j} p^j (1 - p)^{h - d + 1 - j}$$

*X* má tedy binomické rozdělení a tudíž je jeho očekávaná hodnota  $p(h - d + 1)$  a jeho rozptyl je  $p(1 - p)(h - d + 1)$ . Označme *prv* pozici v vrácenou prvním (interpolačním) dotazem v této fázi

<sup>1</sup> zde by byl vhodný obrázek - usečka, která má na krajích *d* a *h* a je na ni videt první interpolační dotaz a skoky po  $\sqrt{n}$  a bin. a  $\sqrt{n}$  ...



vzhledem k počátku bloku. Srovnaj  $prv$  s očekávanou hodnotou  $X$ .

$$|X - prv| \geq \frac{\text{počet dotazů v rámci bloku} - 2}{2} \sqrt{delka}$$

protože když vynecháme první dva dotazy, tak se dále střídá binární dotaz se skokem o  $\sqrt{delka}$ . Vynecháme-li i binární dotazy—vezmu každý druhý—zůstanou jen skoky o  $\sqrt{delka}$  a ty dohromady naskáčou méně než je vzdálenost  $x$  od prvního dotazu.

Označme  $p_i = \mathcal{P}$ (v rámci bloku bylo položeno alespoň  $i$  dotazů). Pak jistě platí

$$\mathcal{P}(|X - prv| \geq \frac{i-2}{2} \sqrt{delka}) \geq p_i$$

Nyní použijeme Čebyševovu nerovnost, která říká, že

$$\mathcal{P}(|X - EX| > t) \leq \frac{\text{rozptyl } X}{t^2}$$

$$p_i \leq \mathcal{P}(|X - prv| \geq \frac{i-2}{2} \sqrt{delka}) \leq \frac{p(1-p) delka}{(\frac{i-2}{2})^2 delka} \leq \frac{1}{(i-2)^2}$$

protože  $prv$  je očekávaná hodnota  $X$  a  $p(1-p) \leq 1/4$  pro  $0 \leq p \leq 1$ . Celkem jsme dostali  $p_i \leq 1/(i-2)^2$ .

Očekávaný čas pro práci v jednom bloku (pro jednu fázi) je  $O$ (očekávaný počet dotazů v bloku) =  $O(\sum_{i=0}^{\infty} p_i) = O(3 + \sum_{i=3}^{\infty} 1/(i-2)^2) = O(3 + \pi^2/6) = O(4.6)$ . To jsme pouze odhadli první tři členy jedničkou a sečetli řadu, kterou asi znáte z analýzy.

Teď už snadno dopočítáme očekávaný čas zobecněného kvadratického vyhledávání. Ten je  $O$ (počet bloků) (očekávaný čas pro 1 blok) =  $O(\log \log(|S|) O(1)) = O(\log \log(|S|))$ . Kde jsme vzali počet bloků? Ten je určitě menší než počet dotazů v interpolačním vyhledávání (jen interpolační dotazy).

# Kapitola 7

## Binární stromy

### 7.1 Obecně

#### 7.1.1 Algoritmus MEMBER

#### 7.1.2 Algoritmus INSERT

#### 7.1.3 Algoritmus DELETE

### 7.2 Optimální binární vyhledávací stromy

XXX dopsat

#### 7.2.1 Algoritmus konstrukce

XXX dopsat

#### 7.2.2 Snížení složitosti z kubické na kvadratickou

XXX dopsat

### 7.3 Skorooptimální binární vyhledávací stromy

#### 7.3.1 Co je to optimální binární vyhledávací strom

referát by  
Ladislav Strojil

Prvky jsou uloženy ve vnitřních vrcholech stromu. Listy jsou intervaly  $(-\infty, x_1)$ ,  $(x_1, x_2)$ ,  $\dots$ ,  $(x_n, \text{inf})$ . Listy nemusíme implicitně ve stromě zaznamenávat. U optimálních stromů dále předpokládáme, že známe pravděpodobnosti operací  $\text{Access}(x)$ .

**Definice 7.3.1.** Nechť  $S = \{x_1 < x_2 < \dots < x_n\}$  a nechť  $\beta_i$  (resp.  $\alpha_j$ ) je pravděpodobnost operace  $\text{Access}(a, S)$ , kde  $a = x_i$  (resp.  $x_j < a < x_{j+1}$ ) pro  $1 \leq i \leq n$  (resp.  $0 \leq j \leq n$ ). Potom  $\beta_i \geq 0$ ,  $\alpha_i \geq 0$  a  $\sum \beta_i + \sum \alpha_i = 1$ .  $(2n+1)$ -tice  $(\alpha_0, \beta_1, \alpha_1, \dots, \beta_n, \alpha_n)$  se nazývá rozdělení (pravděpodobnosti) přístupu.

Strom potom konstruujeme rekurzí tak, aby *průměrná vážená cesta* ve stromě byla co nejkratší.

Takový strom lze konstruovat pomocí rekurzivního výpočtu zkoušením všech kandidátů na kořen. To lze v čase  $O(n^2)$ , protože volba kořene jednoznačně určuje prvky pravého i levého podstromu (neboť se jedná o vyhledávací strom).

### 7.3.2 Aproximace optimálních stromů

Při konstrukci se budeme snažit volbou kořene podstromu rozdělit prvky na dvě stejně pravděpodobné množiny.

Uvažujme následující situaci.  $S = \{x_1, x_2, x_3, x_4\}$  s pravděpodobnostmi přístupu  $(\alpha_0, \beta_1, \alpha_1, \beta_2, \alpha_2, \beta_3, \alpha_3, \beta_4, \alpha_4) = (\frac{1}{6}, \frac{1}{24}, 0, \frac{1}{8}, 0, \frac{1}{8}, \frac{1}{8}, 0, \frac{5}{12})$ .

Doporučuji si představit uvedené body na reálné ose tak, že  $\alpha_0$  je v bodě 0,  $\alpha_4$  v bodě 1.

Bod  $\frac{1}{2}$  padne buď do  $\beta_i$  nebo do  $\alpha_j$  pro nějaké  $i$ , resp.  $j$ . V prvním případě zvolíme  $x_i$  jako kořen stromu, jinak volíme mezi  $x_j$  a  $x_{j+1}$ , podle toho, zda  $\frac{1}{2}$  leží v levé nebo pravé polovině  $\alpha_j$ .

V našem případě volíme  $x_3$  jako kořen stromu.

Pro rozhodnutí o kořenu levého podstromu vrchol, který popsáním způsobem odpovídá bodu  $\frac{1}{4}$ . Tento postup není totožný s postupem, kdy se bere bod  $\frac{1}{2}$  v nově vzniklé podúloze, protože při konstrukci podstromu bychom zanedbali část intervalu  $\alpha_3$ .

### 7.3.3 Podrobnější popis naznačené metody

Nechť

$$s_0 = \frac{\alpha_0}{2} s_i = s_{i-1} + \frac{\alpha_{i-1}}{2} + \beta_i + \frac{\alpha_i}{2}. \quad (7.1)$$

Uvědomte si, že  $s_i$  jsou středy intervalů příslušejících "neúspěšnému vyhledávání", tj.  $(x_i, x_{i+1})$ .

Volání funkce `construct_tree(0, n, 0, 1)` vytvoří skoro optimální vyhledávací strom dle popsané metody.

**procedure** `construct_tree(i,j,cut,l)`

*Poznámka: Předpokládáme, že parametry volané funkce splňují následující podmínky.*

- 1)  $i$  a  $j$  jsou celá čísla taková, že  $0 \leq i < j \leq n$
- 2)  $l$  je celé číslo,  $l \geq 0$
- 3)  $cut = \sum_{p=1}^{l-1} x_p 2^{-p}$ , kde  $x_p \in \{0, 1\}$  pro všechna  $p$
- 4)  $cut \leq s_i \leq s_j \leq cut + 2^{-l+1}$

Volání `construct_tree(i, j, , )` vytvoří binární vyhledávací strom pro vrcholy  $i+1, \dots, j$  a listy  $i, \dots, j$ .

**begin**

**if**  $i + 1 = j$  (případ A) **return** kořen= $j$ , levý list= $i$ , pravý list= $j$ ;

**else**

najdi  $k$  takové, že

5)  $i < k \leq j$

6)  $k = i + 1$  nebo  $s_{k-1} \leq cut + 2^{-l}$

7)  $k = j$  nebo  $s_k \geq cut + 2^{-l}$

*Takové  $k$  vždy existuje, protože parametry funkce splňují podmínku 4.*

**if**  $k = i + 1$  (případ B) **return**

kořen= $i+1$

levý list= $i$

pravý list=`construct_tree(i+1,j,cut+2-l,l+1)`;

**if**  $k = j$  (případ C) **return**

kořen= $j$

levý list=`construct_tree(i,j-1,cut,l+1)`

pravý list= $j$ ;

**if**  $i + 1 < k < j$  (případ D) **return**

kořen= $k$

levý list=`construct_tree(i,k-1,cut,l+1)`

pravý list=`construct_tree(k,j,cut+2-l,l+1)`;

**end**

**Věta 7.3.1.** *Nechť  $b_i$  je hloubka vrcholu  $x_i$  a  $a_j$  je hloubka listu  $(x_j, x_{j+1})$  ve stromě  $T_{BB}$  vytvořeném funkcí **construct\_tree**(0, n, 0, 1). Potom*

$$b_i \leq \lfloor \log 1/\beta_i \rfloor, \quad a_j \leq \lfloor \log 1/\alpha_j \rfloor + 2$$

*Důkaz.* Věta říká, že hloubka vrcholu roste s klesající pravděpodobností přístupu k tomuto vrcholu. Plyne z následujících faktů.  $\square$

**Fakt 1.** Jestliže hodnoty parametrů funkce **construct\_tree** splňují podmínky 1-4 a  $i + 1 \neq j$ , potom k splňující požadované podmínky existuje a hodnoty parametrů rekurzivních volání **construct\_tree** splňují 1-4.

*Důkaz.*

Předpokládejme, že parametry splňují 1 - 4 a  $i + 1 \neq j$ . Potom zřejmě platí  $cut \leq s_j \leq cut + 2^{-l+1}$ . Pro spor předpokládejme, že neexistuje žádné  $k$ ,  $i < k \leq j$ , pro které by platilo  $s_{k-1} \leq cut + 2^{-l}$  a  $s_k \geq cut + 2^{-l}$ .

Potom ovšem buď pro všechna  $k$  taková, že  $i < k \leq j$ , platí  $s_k < cut + 2^{-l}$  nebo pro všechna  $k$  taková, že  $i < k \leq j$ , platí  $s_{k-1} > cut + 2^{-l}$ .

V prvním případě  $k = j$  odpovídá požadovaným podmínkám, v druhém jim odpovídá  $k = i + 1$ . Tedy  $k$  vždy existuje.

Zbývá ukázat, že nové parametry volání funkce splňují požadované podmínky. To ale plyne z toho, že  $k$  splňuje 5-7 a  $i + 1 \neq j$ .  $\square$

**Fakt 2.** Hodnoty parametrů všech volání **construct\_tree** splňují podmínky 1-4.

*Důkaz.* Indukcí. **construct\_tree**(0, n, 0, 1) splňuje 1-4 a pomocí předchozího faktu.  $\square$

Řekneme, že vrchol  $h$  (resp. list  $h$ ) je vytvořen voláním **construct\_tree**(i, j, cut, l), jestliže  $h = j$  (resp.  $h = j$  nebo  $h = i$ ) a byl proveden případ A, nebo  $h = i + 1$  (resp.  $h = i$ ) a byl proveden případ B, nebo  $h = j$  a byl proveden případ C, nebo  $h = k$  a byl proveden případ D.

Dále nechť  $b_i$  je hloubka vrcholu  $i$  a  $a_j$  hloubka listu  $j$  ve stromě vráceném **construct\_tree**(0, n, 0, 1).

**Fakt 3.** Je-li vrchol  $h$  (resp. list  $h$ ) vytvořen voláním **construct\_tree**(i, j, cut, l), potom  $b_h + 1 = l$  (resp.  $a_h = l$ ).

*Důkaz.* Indukcí podle  $l$ .  $\square$

**Fakt 4.** Je-li vrchol  $h$  (resp. list  $h$ ) vytvořen voláním **construct\_tree**(i, j, cut, l), potom  $\beta_h \leq 2^{-l+1}$  (resp.  $\alpha_h \leq 2^{-l+2}$ ).

*Důkaz.* Parametry splňují 4 a tedy  $2^{l+1} \geq s_j - s_i = (\alpha_i + \alpha_j)/2 + \beta_{i+1} + \alpha_{i+1} + \dots + \beta_j \geq \beta_h$  (resp.  $\alpha_h/2$ )  $\square$

*věty.* Z faktů 3 a 4 obdržíme  $\beta_h \leq 2^{-b_h}$  a  $\alpha_h \leq 2^{-a_h+2}$ . Zlogaritmováním a převedením na celočíselné hodnoty dostáváme tvrzení věty.  $\square$

Věty 1 a 2 ukazují, že hloubka vrcholu je přibližně rovna logaritmu převrácené hodnoty pravděpodobnosti přístupu k tomuto vrcholu.

**Definice 7.3.2.** Nechť  $(\gamma_1, \gamma_2, \dots, \gamma_n)$  je diskrétní rozdělení pravděpodobnosti. Potom se funkce  $H(\gamma_1, \gamma_2, \dots, \gamma_n) = -\sum_{i=1}^n \gamma_i \log \gamma_i$  nazývá entropie rozdělení.

Povšimněte si, že entropie nezáleží na vytvořeném stromě, jenom na pravděpodobnostech přístupu.

**Věta 7.3.2.** *Nechť  $P_{BB}$  je vážená délka cesty zkonstruovaného stromu. Potom*

$$\begin{aligned} P_{BB} &\leq \sum \beta_i \lfloor \log 1/\beta_i \rfloor + \sum \alpha_j \lfloor \log 1/\alpha_j \rfloor + 1 + \sum \alpha_j \leq \\ &\leq H(\alpha_0, \beta_1, \alpha_1, \dots, \beta_n, \alpha_n) + 1 + \sum \alpha_j \end{aligned} \quad (7.2)$$

Navíc

**Věta 7.3.3.** *Nechť  $P_{BB}$  je vážená délka cesty zkonstruovaného stromu a necht'  $P_{opt}$  je vážená délka cesty v optimálním stromu. ( $B = \sum \beta_i$ ) Potom*

$$1) \max\left\{\frac{H-dB}{\log(2+2^{-d})}; d \in \mathbf{R}\right\} \leq P_{opt} \leq P_{BB} \leq H + 1 + \sum \alpha_j$$

$$2) P_{BB} \leq P_{opt} + B(\log e + \log(P_{opt}/B)) + 2 \sum \alpha_j$$

*Důkaz.* Plyne z věty 5 (původní číslování, viz [1], strana 175) a věty 2.  $\square$

Je to jenom složité počítání, jde o to, že dovedeme odhadnout, jak velká dovede být ta vážená cesta v námi vytvořeném stromě.

$$P_{BB} - P_{opt} \leq \log P_{opt}, \text{ což je přibližně } \log H.$$

### 7.3.4 Časová složitost

Pro sestrojení stromu s jedním vrcholem potřebuje metoda konstantní čas, tj.  $T(1) = c_1$ .

Pro  $n > 1$  je potřeba najít  $k$  a dojde až ke dvěma rekurzivním voláním **construct\_tree**. Necht'  $T(m, n)$  je čas potřebný pro nalezení  $k$ , kde  $m = k - i$  (tj. vzdálenost  $k$  od počátku zkoumaného úseku). **construct\_tree** je voláno maximálně dvakrát, v případě D první volání sestrojí strom s  $k - 1 - i = m - 1$  vrcholy a druhé volání strom s  $j - k = n - m$ .

$$\text{Tedy } T(n) \leq \max(T(m-1) + T(n-m) + T_S(n, m) + c_2).$$

Zde  $c_2$  je konstanta měřící složitost předávání parametrů.

Dodefinujeme-li  $T(0) = 0$ , potom uvedená nerovnost platí i pro případy B a C. Zavedeme-li dále konvenci  $T_S(1, m) = 0$  a  $c = \max(c_1, c_2)$ , dotáváme zjednodušený výraz:

$$T(0) = 0T(n) \leq \max_{1 \leq m < n} (T(m-1) + T(n-m) + T_S(n, m) + c) \quad (7.3)$$

### 7.3.5 Hledání $k$

Číslo  $k$  můžeme hledat binárním vyhledáváním (půlení intervalu), ale výsledná časová složitost by byla  $O(n \log n)$

Principiální problém s vyhledáváním pomocí půlení intervalu je, že nám nalezení  $k$  trvá dlouho i v případě, že je blízko  $i$  nebo  $j$  a tedy neredukuje velikost podúlohy podstatným způsobem.

Řešením je kombinace exponenciálního a binárního vyhledávání. Tím dosáhneme toho, že  $k$ , která jsou blízko krajním bodům intervalu, nalezneme rychleji.

Budeme vyhledávat od konců intervalu, ale ne v konstantních krocích.

1) Porovnáme  $s_r$  s  $cut + 2^{-l}$ , kde  $r = \lfloor (i+1+j)/2 \rfloor$ . Je-li  $s_r \geq cut + 2^{-l}$ , potom  $k \in \{i+1, \dots, r\}$ . Je-li  $s_r \leq cut + 2^{-l}$ , potom  $k \in \{r, \dots, j\}$ . V dalším budeme předpokládat, že  $k \in \{i+1, \dots, r\}$ .

Tento krok trvá konstantní čas.

2) Nalezneme nejmenší  $t$ ,  $t = 0, 1, 2, \dots$ , takové, že  $s_{i+2^t} \geq cut + 2^{-l}$ . Nazvěme jej  $t_0$ .  $t_0$  lze nalézt v čase  $d_2(t_0 + 1)$  pro nějakou konstantu  $d_2$ .

Potom  $i + 2^{t_0-1} < k \leq i + 2^{t_0}$ , tj.  $2^{t_0} \geq k - i = m > 2^{t_0-1}$  a odtud  $\log m > t_0 - 1$ . Tedy trvání kroku 2 je omezené  $d_2(2 + \log m)$ .

3) Binárním vyhledáváním na intervalu  $i + 2^{t_0-1} + 1, \dots, i + 2^{t_0}$  zjistíme přesnou hodnotu  $k$ .

Tohle zabere  $d_3(\log(2^{t_0} - 2^{t_0-1}) + 1) = d_3 t_0 < d_3(1 + \log m)$  (pro nějakou konstantu  $d_3$ ).

Tedy pro  $i < k \leq \lfloor (i+1+j)/2 \rfloor$  nalezneme  $k$  v čase menším než  $d_3(1 + \log m)$ . Zde se  $m = k - i$ . Symetricky lze  $k$  nalézt v čase  $\leq d(1 + \log(n - m + 1))$ , v případě, že  $\lfloor (i+1+j)/2 \rfloor < k$ .

Tedy  $T_S(n, m) = d(1 + \log \min(m, n - m + 1))$

Dostáváme pro **construct\_tree** následující rekurzivní vztah.

$$T(0) = 0$$

$$T(n) = \max_{1 \leq m < n} (T(m-1) + T(n-m) + d(1 + \log \min(m, n - m + 1)) + c)$$

**Věta 7.3.4.** Je-li vyhledávání  $k$  implementováno pomocí uvedené kombinace exponenciálního a binárního vyhledávání, potom  $T(n) = O(n)$ .

*Důkaz.* Indukcí podle  $n$  ukážeme  $T(n) \leq (2d + c)n - d \log(n + 1)$ .

Pro  $n = 0$  vztah zřejmě platí.

Pro  $n > 0$  máme

$$T(n) = \max_{1 \leq m < n} (T(m - 1) + T(n - m) + d(1 + \log \min(m, n - m + 1)) + d + c) \quad (7.4)$$

Tedy podle symetrie výrazu v  $m - 1$  a  $n - m$  dostáváme:

$$T(n) \leq \max_{1 \leq m < (n+1)/2} (T(m - 1) + T(n - m) + d \log m + d + c) \quad (7.5)$$

Podle indukčního předpokladu dostáváme

$$\begin{aligned} T(n) &\leq \max_{1 \leq m < (n+1)/2} ((2d + c)(m - 1 + n - m) - \\ &\quad - d(\log m + \log(n - m + 1)) + d \log m + (d + c)) \end{aligned} \quad (7.6)$$

Což se rovná

$$(2d + c)n + \max_{1 \leq m < (n+1)/2} (-d(1 + \log m - m + 1)) \quad (7.7)$$

Výraz v závorce je vždy menší než nula a je největší pro  $m = (n + 1)/2$ . Tedy dostáváme následující nerovnost:

$$T(n) \leq (2d + c)n - d(1 + \log(n + 1)/2) = (2d + c)n - d \log(n + 1) \quad (7.8)$$

□

## 7.4 Červenočerné stromy

**Definice 7.4.1.** Binární vyhledávací strom  $T$  se nazývá červenočerný, jestliže každý vrchol je obarven červeně nebo černě a platí následující podmínky:

- 1) Listy jsou černé.
- 2) Pokud má červený vrchol otce, je otec černý.
- 3) Všechny cesty z kořene do listu mají stejný počet černých vrcholů.

**Věta 7.4.1.** Pro binární vyhledávací červenočerné stromy reprezentující množinu  $S$  platí, že jejich hloubka je  $O(\log n)$ .

*Důkaz.* Je-li  $k$  počet černých vrcholů na cestě z kořene do listu, pak

$$2^k - 1 \leq |S| \leq 2^{2k} - 1$$

neboli

$$k \leq \log_2 |S| + 1 \leq 2k$$

přičemž prvky  $S$  jsou reprezentovány pouze ve vnitřních vrcholech, ne v listech.

□ je to novinka?

nejdelší cesta je  
max.  $2 \times$  delší  
než nejkratší

### 7.4.1 Operace INSERT

Uvedeme pouze odlišnost od operace INSERT v obecném binárním vyhledávacím stromě.

Situace: list  $t$  se změnil na vnitřní vrchol reprezentující prvek  $x$  a přidali jsme mu 2 listy.

Vrchol  $t$  obarvíme červeně a jeho syny černě. Podmínky 1 a 3 stále platí, ale podmínka 2 platit nemusí.

**Definice 7.4.2.** Strom a jeho vrchol  $(T, t)$  nazveme *2-téměř červenočerný strom (2tččs)*, jestliže platí

- 1 Listy jsou černé. (*nezměněno*)
- 2' Pokud má červený vrchol *různý od  $t$*  otec, je otec černý.
- 3 Všechny cesty z kořene do listu mají stejný počet černých vrcholů. (*nezměněno*)

Srovnej: Každý červený vrchol různý od  $t$  má černého otce.

**Definice 7.4.3.** Je-li vrchol  $t$  červený a jeho otec je také červený, pak řekneme, že  $t$  je *porucha*.

Tedy nyní máme 2tččs  $(T, t)$  Je-li  $t$  porucha, pak ji musíme nějak opravit. Situace je na obrázku 7.1. Nejprve záleží na tom, jakou barvu má  $s$ , strýc  $t$ :

Obrázek 7.1: Obecná situace při INSERTu

- 1)  $s$  je červený. Pak pouze přebarvíme  $o$ ,  $d$  a  $s$  podle obrázku 7.2. Podmínky 1 a 3 jsou splněny. Nyní  $d$  může být porucha, ovšem posunutá o 2 hladiny výše. Vznikl 2tččs  $(T, d)$ .

Obrázek 7.2: Oprava INSERTu přebarvením

- 2)  $s$  je černý. Záleží na tom, zda hodnota  $t$  leží mezi hodnotami  $o$  a  $d$  nebo ne. Jinými slovy, zda cesta  $t$ - $o$ - $d$  obsahuje zatáčku.
  - (a) Bez zatáčky: Provedeme rotaci a přebarvíme podle obrázku 7.3. Splněny budou podmínky 1, 2 i 3, tedy máme červenočerný strom.

Obrázek 7.3: Oprava INSERTu rotací a přebarvením

- (b) Se zatáčkou: Provedeme dvojitou rotaci a přebarvíme podle obrázku 7.4. Splněny budou podmínky 1, 2 i 3, opět máme rovnou červenočerný strom.

Obrázek 7.4: Oprava INSERTu dvojitou rotací a přebarvením

### 7.4.2 Operace DELETE

Zatímco INSERT se příliš nelišil od své obdoby u AVL stromů, operace DELETE u červenočerných stromů je oproti AVL stromům složitější mentálně, ovšem jednodušší časově.

Situace: odstraňujeme vrchol  $t$  (který nemusí reprezentovat odstraňovaný prvek — viz DELETE v obecných binárních vyhledávacích stromech) a jeho syna, který je list.

Druhého syna  $t$ ,  $u$ , dáme na místo smazaného  $t$  a začerníme ho. Tím máme splněné podmínky 1 a 2. Pokud byl ale  $t$  černý, chybí nám na cestách procházejících nyní  $u$  jeden černý vrchol.

**Definice 7.4.4.** Strom a jeho vrchol  $(T, u)$  nazveme *3-téměř červenočerný strom (3tččs)*, jestliže platí

- 1 Listy jsou černé. (*nezměněno*)
- 2 Pokud má červený vrchol otce, je otec černý. (*nezměněno*)
- 3' Všechny cesty z kořene do listu neprocházející  $u$  mají stejný počet černých vrcholů, nechť je to  $k$ . Všechny cesty z kořene do listu procházející  $u$  mají stejný počet černých vrcholů, nechť je to  $\ell$ . A platí  $k - 1 \leq \ell \leq k$ .

Když  $u$  není kořen a  $\ell < k$ , pak řekneme, že  $u$  je *porucha*.

Nechť vrchol  $u$  je porucha. Pak můžeme předpokládat, že je obarven černě, jinak bychom ho přebarvili na černo a tím by se porucha odstranila a vznikl červenočerný strom.

Situace: máme 3tččs  $(T, u)$ ,  $u$  je porucha s otcem  $o$ , bratrem  $b$  a synovci  $s_1, s_2$ , viz obrázek 7.5. Oprava záleží na barvě  $b$ :

Obrázek 7.5: Obecná situace při DELETE

- 1) Bratr je černý. Rozlišujeme dále 4 případy, z nichž jeden propaguje poruchu o hladinu výš a ostatní skončí s červenočerným stromem.
  - (a) Otec i synovci jsou černí. Přebarvíme  $b$  na červenou, viz obrázek 7.6. Dostáváme 3tččs  $(T, o)$ , tedy porucha je o hladinu výše.

Obrázek 7.6: Částečná oprava DELETE přebarvením

- (b) Otec je červený, synovci černí. Přebarvíme otce a bratra podle obrázku 7.7 a dostáváme červenočerný strom.

Obrázek 7.7: Oprava DELETE přebarvením

- (c) Synovec  $s_1$ , jehož hodnota leží mezi hodnotami otce a bratra, je černý, druhý synovec je červený. Přebarvíme a zrotujeme podle obrázku 7.8, barva otce se nemění (tj., vrchol  $b$  bude mít barvu, kterou původně měl vrchol  $o$ ). Dostáváme červenočerný strom.

Obrázek 7.8: Oprava DELETE přebarvením a rotací

- (d) Synovec  $s_1$ , jehož hodnota leží mezi hodnotami otce a bratra, je červený, druhý synovec má libovolnou barvu. Přebarvíme a dvojitě zrotujeme podle obrázku 7.9 (tj., vrchol  $s_1$  bude mít barvu, kterou původně měl vrchol  $o$  a barva vrcholu  $s_2$  se nezmění). Dostáváme červenočerný strom.

Obrázek 7.9: Oprava DELETE přebarvením a dvojitou rotací

- 2) Bratr je červený. Přebarvíme a zrotujeme podle obrázku 7.10. Dostáváme 3tččs  $(T, u)$ , přičemž porucha je o hladinu níže. I když to tak na první pohled nevypadá, máme vyhráno, protože bratr poruchy je černý a otec červený, tedy příští oprava bude případ 1b, 1c, nebo 1d a skončíme s červenočerným stromem.

Obrázek 7.10: Částečná oprava DELETE přebarvením a rotací



### 7.4.3 Závěry

Pro binární vyhledávací červenočerné stromy lze implementovat MEMBER, INSERT a DELETE tak, že vyžadují čas  $O(\log n)$  a INSERT používá nejvýše jednu (dvojitou) rotaci a DELETE používá nejvýše dvě rotace nebo rotaci a dvojitou rotaci.

Jsou lepší než AVL stromy, které při DELETE spotřebují až  $\log n$  rotací. Oproti váhově vyváženým stromům i proti AVL stromům jsou červenočerné stromy jen konstantně lepší, ale i to je dobré. Při použití binárních vyhledávacích stromů ve výpočetní geometrii nese informaci i rozložení prvků ve stromě, a tato informace se musí po provedení rotace nebo dvojitě rotace aktualizovat. To znamená prohledání celého stromu a tedy čas  $O(n)$  za každou rotaci a dvojitou rotaci navíc. Pro tyto problémy jsou červenočerné stromy obzvláště vhodné, protože minimalizují počet použitých rotací a dvojitých rotací.

Červenočerné stromy se používají při implementaci  $(2, 4)$ -stromů, se kterými se seznámíme v další kapitole. Vrchol se dvěma syny je nahrazen jedním černým vrcholem, vrchol se třemi syny je nahrazen černým vrcholem s jedním červeným synem a vrchol se čtyřmi syny je nahrazen černým vrcholem se dvěma syny. Pozor! Aktualizační operace pro  $(2, 4)$ -stromy neodpovídají aktualizačním operacím na červenočerných stromech (i reprezentace prvků je odlišná).

Červenočerné stromy se používají například ve standardní šablonové knihovně jazyka C++ od SGI, která je zahrnuta do GCC. Máte-li Linux, zkuste se podívat do `/usr/include/g++-2/stl\_tree.h`.<sup>1</sup>

---

<sup>1</sup>A pokud víte o podobně dostupných implementacích jiných datových struktur z téhle přednášky, sem s nimi!

# Kapitola 8

## $(a, b)$ stromy

### 8.1 Základní varianta

Nechť  $a, b \in \mathbb{N}, a \leq b$ . Strom je  $(a, b)$  strom, když platí

- 1) Každý vnitřní vrchol kromě kořene má alespoň  $a$  a nejvýše  $b$  synů.
- 2) Kořen má nejvýše  $b$  synů. Pokud  $a \geq 2$ , pak má alespoň 2 syny, nebo je listem.
- 3) Všechny cesty z kořene do listu jsou stejně dlouhé.

**Definice 8.1.1.** Jsou-li synové každého vrcholu očíslováni, můžeme definovat *lexikografické uspořádání vrcholů na stejné hladině*.

$u \leq_l v$ , jestliže otec  $u <_l$  otec  $v$  nebo otec  $u =$  otec  $v$ ,  $u$  je  $i$ -tý syn,  $v$  je  $j$ -tý syn a  $i \leq j$ .

Pozorování: Buď  $T$   $(a, b)$  strom s hloubkou  $h$ . Platí

$$2a^{h-1} \leq \text{počet listů } T \leq b^h,$$

tedy pro libovolné  $n$  má každý  $(a, b)$  strom  $T$  s  $n$  listy hloubku  $\Theta(\log n)$ .

#### 8.1.1 Reprezentace množiny $S$ $(a, b)$ stromem

Mějme  $S \subseteq U$ , přičemž universum je lineárně uspořádané.  $(a, b)$  strom  $T$  reprezentuje množinu  $S$ , jestliže existuje jednoznačné přiřazení prvků  $S$  listům  $T$ , které zachovává uspořádání.

Potřebujeme navíc podmínku

4.  $a \geq 2$  a  $b \geq 2a - 1$

Struktura vnitřního vrcholu  $v$ :

- $\rho_v$  je počet synů
- $S_v[1 \dots \rho_v]$  je pole ukazatelů na syny
- $H_v[1 \dots \rho_v - 1]$ :  $H_v[i]$  je maximální prvek v podstromu  $S_v[i]$

Cvičení: Co by se stalo, kdybychom definici zjednodušili a místo podmínek 1 a 2 požadovali, aby každý vrchol měl  $a$  až  $b$  synů?

Obrázek 8.1: Příklad  $(a, b)$  stromu

### 8.1.2 MEMBER( $x$ ) v ( $a, b$ ) stromu

```

{vyhledání  $x$ }
 $t :=$  kořen
while  $t$  není list do
   $i := 1$ 
  while  $H_t[i] < x \wedge i < \rho_t$  do
     $i := i + 1$ 
  end while
   $t := S_t[i]$ 
end while
{testování  $x$ }
if  $t$  reprezentuje  $x$  then
   $x \in S$ 
else
   $x \notin S$ 
end if

```

### 8.1.3 INSERT( $x$ ) do ( $a, b$ ) stromu

```

vyhledání  $x$ 
if  $t$  nereprezentuje  $x$  then
   $o :=$  otec  $t$ 
  vrcholu  $o$  přidej nového syna  $t'$  reprezentujícího  $x$ 
  zařaď  $t'$  na správné místo mezi jeho bratry a uprav  $\rho_o, S_o$  a  $H_o$ 
   $t := o$ 
  while  $\rho_t > b$  do
    {Štěpení — můžeme provést díky podmínce 4}
    rozděl  $t$  na  $t_1$  a  $t_2$ 
    k  $t_1$  dej prvních  $\lfloor (b+1)/2 \rfloor$  synů  $t$ 
    k  $t_2$  dej zbylých  $\lceil (b+1)/2 \rceil$  synů  $t$ 
     $o :=$  otec  $t$ 
    uprav  $\rho_o, S_o$  a  $H_o$ 
    {při štěpení kořene ještě musíme vytvořit nový kořen}
     $t := o$ 
  end while
end if

```

### 8.1.4 DELETE( $x$ ) z ( $a, b$ ) stromu

```

vyhledání  $x$ , navíc si zapamatuj vrchol  $u$ , v jehož poli  $H_u$  je  $x$ 
if  $t$  reprezentuje  $x$  then
   $o :=$  otec  $t$ 
  odstraň  $t$ 
  uprav  $H_o, H_u \{ \dots \}$ 
  uprav  $S_o$  a  $\rho_o$ 
   $t := o$ 
  while  $\rho_t < a \wedge t$  není kořen do
     $v :=$  bezprostřední bratr  $t$ 
    if  $\rho_v = a$  then {smíme spojit}
      {Spojení}
       $o :=$  otec  $t$ 
      sluč  $v$  a  $t$  do  $t$ 
      uprav  $\rho_o, S_o$  a  $H_o$ 
       $t := o$ 
    end if
  end while

```

```

    else { $\rho_v > a$ , spojení by mohlo mít více než  $b$  synů}
      {Přesun}
      přesuň krajního syna  $v$  do  $t$ 
      uprav  $H_{\text{otec } t}$ 
    end if
  end while
  if  $t$  je kořen a má jen jednoho syna then
    smaž  $t$ 
  end if
end if

```

### 8.1.5 Shrnutí

Operace štěpení, přesun i spojení vyžadují konstantní čas.

**Věta 8.1.1.** Operace *MEMBER*, *INSERT* a *DELETE* pro  $(a, b)$  stromy vyžadují čas  $O(\log n)$ , kde  $n$  je velikost reprezentované množiny.

S  $H$  a  $S$  jsme pracovali jako se seznamy, nepotřebujeme, aby to byla pole. Tím se zjednoduší implementace.

Výhodnost pro  
vnější paměti?

### 8.1.6 Jak volit parametry $(a, b)$

Pro vnitřní paměť je vhodné  $a = 2$  nebo  $a = 3$ ,  $b = 2a$ . Pro vnější paměť je vhodné  $a \approx 100$ ,  $b = 2a$ .

Pro minimalizaci paměťových nároků je výhodné  $b = 2a - 1$ , pro minimalizaci časových nároků je výhodné  $b = 2a$ .

proč? prý se k  
tomu ještě  
dostaneme

## 8.2 Další operace

Pro operaci JOIN je vhodné spolu se stromem uchovávat také největší prvek reprezentované množiny.

### 8.2.1 Algoritmus JOIN( $T_1, T_2$ ) pro $(a, b)$ stromy

**Require:**  $T_1$  reprezentuje  $S_1$ ,  $T_2$  reprezentuje  $S_2$  a  $\max S_1 < \min S_2$

$n :=$  hloubka  $T_1$  – hloubka  $T_2$

**if**  $n \geq 0$  **then**

$t :=$  kořen  $T_1$

**while**  $n > 0$  **do**

$t :=$  poslední syn  $t$

$n := n - 1$

**end while**

Spoj  $t$  s kořenem  $T_2$  a vytvoř nový vrchol  $t'$ . {zde se využije znalost největšího prvku množiny  $S_1$ }

**while**  $\rho_t > b$  **do**

Štěpení  $t$

$t :=$  otec  $t$

**end while**

**else**

{analogicky: kořen  $T_2$ , první syn ... }

**end if**

JOIN vyžaduje čas  $O(\text{rozdíl hloubek stromů}) \leq O(\log(|S_1| + |S_2|))$ .

### 8.2.2 Algoritmus SPLIT( $x, T$ ) pro ( $a, b$ ) strom

**Ensure:** Vytvoří  $T_1$  reprezentující  $\{s \in S : s < x\}$  a  $T_2$  reprezentující  $\{s \in S : s > x\}$

Nechť  $Z_1$  a  $Z_2$  jsou prázdné zásobníky

$t :=$  kořen  $T$

**while**  $t$  není list **do**

$i := 1$

**while**  $H_t[i] < x \wedge i < \rho_t$  **do**

$i := i + 1$

**end while**

  Vytvoř strom  $T_1$ , jehož kořen má syny  $S_t[1] \dots S_t[i-1]$

  Vytvoř strom  $T_2$ , jehož kořen má syny  $S_t[i+1] \dots S_t[\rho_t]$

**if**  $T_1$  není jednoprvkový strom **then**

    Push( $Z_1, T_1$ )

**end if**

**if**  $T_2$  není jednoprvkový strom **then**

    Push( $Z_2, T_2$ )

**end if**

$t := S_t[i]$

**end while**

**if**  $t$  reprezentuje prvek různý od  $x$  **then**

  Udělej z  $t$  ( $a, b$ ) strom a vlož ho do příslušného zásobníku.

**end if**

$T_1 :=$  STACKJOIN( $Z_1$ ) {viz dále}

$T_2 :=$  STACKJOIN( $Z_2$ )

Čas rozřezávání stromu je úměrný jeho hloubce. Celkový čas operace SPLIT ovšem závisí ještě na složitosti operace STACKJOIN.

### 8.2.3 Algoritmus STACKJOIN( $Z$ ) pro zásobník ( $a, b$ ) stromů

$T :=$  Pop( $Z$ )

**while**  $Z \neq \emptyset$  **do**

$T' :=$  Pop( $Z$ )

$T :=$  JOIN( $T, T'$ )

**end while**

Nechť  $Z$  obsahuje ( $a, b$ ) stromy  $T_1 \dots T_k$ , přičemž  $T_1$  je vrchol zásobníku. Platí

$$\forall i : \text{hloubka } T_i \leq \text{hloubka } T_{i+1}$$

$$\begin{aligned} \text{čas STACKJOIN} &= \text{hloubka } T_2 - \text{hloubka } T_1 + 1 \\ &+ \text{hloubka } T_3 - \text{hloubka } T_2 + 1 \\ &+ \dots \\ &+ \text{hloubka } T_k - \text{hloubka } T_{k-1} + 1 \\ &= \text{hloubka } T_k - \text{hloubka } T_1 + \text{počet JOINů} \\ &= O(\text{hloubka } T) = O(\log |S|) \end{aligned}$$

Tedy i operace SPLIT vyžaduje čas  $O(\log |S|)$ .

### 8.2.4 Algoritmus FIND( $T, k$ ) pro ( $a, b$ ) strom

Nalezení  $k$ -tého nejmenšího prvku.

Rozšíříme reprezentaci stromu a každému vnitřnímu vrcholu  $v$  přidáme:

- $K_v[1 \dots \rho_v]$ :  $K_v[i]$  je počet listů v podstromu  $S_v[i]$

```

t := kořen T
while t není list do
  i := 1
  while  $K_t[i] < k \wedge i < \rho_t$  do
    k := k -  $K_t[i]$ 
    i := i + 1
  end while
  t :=  $S_t[i]$ 
end while
if k > 1 then
  return nil {k > |S|}
else
  return t
end if

```

Časová složitost je opět logaritmická, přičemž dříve uvedené operace nejsou zpomaleny tím, že aktualizují pole (seznam)  $K$ .

### 8.2.5 A-sort

Na první pohled se zdá, že použití  $(a, b)$  stromů ke třídění není výhodné. Paměťové nároky budou oproti běžnému třídění v poli asi pětikrát větší. Aby se tedy třídění  $(a, b)$  stromem vyplatilo, muselo by přinést zvýšení rychlosti. V této části předvedeme, že to skutečně je možné, jestliže vstupní data jsou již částečně setříděná.

Pro účely A-sortu rozšíříme reprezentaci takto:

- Listy stromu jsou propojeny do seznamu
- Je známa cesta z nejmenšího (nejlevějšího) listu do kořene (uložená např. v zásobníku)

Použijeme  $(2, 3)$ -strom.

Nechť vstupní posloupnost je  $a_1, \dots, a_n$ . Postupně odzadu vkládáme její prvky do stromu modifikovaným INSERTem:

```

k := n
while k > 1 do
  A-INSERT( $a_k$ )
  k := k - 1
end while

```

proč?

Na konci přečteme setříděnou posloupnost pomocí spojového seznamu listů.

A-INSERT, stejně jako původní INSERT, najde správný list a potom případně přidá nový prvek. K nalezení správného listu ovšem využívá cestu z nejmenšího listu. Zde uvedená verze A-INSERTu odstraňuje duplicitní prvky, operaci lze pochopitelně upravit tak, že nechává duplicitní prvky, které zůstávají ve stejném pořadí.

```

{Nalezení}
t := nejmenší list
repeat
  t := otec t
until t je kořen  $\vee x \leq H_t[1]$ 
{nyní jako v původním INSERTu, pouze jsme jinak inicializovali t}
while t není list do
  i := 1
  while  $H_t[i] < x \wedge i < \rho_t$  do
    i := i + 1
  end while
  t :=  $S_t[i]$ 

```

```

end while
{Přidání}
if t nereprezentuje x then
  o := otec t
  vrcholu o přidej nového syna t' reprezentujícího x
  zařaď t' na správné místo mezi jeho bratry a uprav  $\rho_o$ ,  $S_o$  a  $H_o$ 
  t := o
  while  $\rho_t > b$  do
    {Štěpení — můžeme provést díky podmínce 4}
    rozděl t na  $t_1$  a  $t_2$ 
    k  $t_1$  dej prvních  $\lfloor (b+1)/2 \rfloor$  synů t
    k  $t_2$  dej zbylých  $\lceil (b+1)/2 \rceil$  synů t
    o := otec t
    uprav  $\rho_o$ ,  $S_o$  a  $H_o$ 
    {při štěpení kořene ještě musíme vytvořit nový kořen}
  end while
end if

```

Čas A-sortu =  $\sum$  času vyhledání +  $\sum$  času přidání + čas vytvoření výstupní posloupnosti.  
 Čas vytvoření výstupní posloupnosti =  $O(n)$ .

$\sum$  času přidání = počet přidávaných vrcholů  $\cdot$  čas přidání vrcholu + počet štěpení  $\cdot$  čas štěpení =  $O(n) \cdot O(1) + \text{počet štěpení} \cdot O(1)$ . Protože se zde neprovádí operace DELETE, lze každému štěpení přiřadit vnitřní vrchol, který byl při tomto štěpení vytvořen (štěpení rozdělí vrchol  $t$  na dva vrcholy  $t_1$  a  $t_2$ , budeme předpokládat, že vrchol  $t_1$  je pokračováním vrcholu  $t$  a vrchol  $t_2$  je vrchol vzniklý při štěpení). Tedy počet štěpení je menší než počet vnitřních vrcholů (při štěpení kořene vzniká navíc ještě nový kořen), tedy  $\sum$  času přidání =  $O(n)$ .

Čas A-sortu tedy závisí hlavně na celkovém čase vyhledání prvků. Označme

$$f_i = |\{j > i : a_j < a_i\}|,$$

tedy počet prvků posloupnosti, které v nesetříděné posloupnosti následují  $a_i$ , ale v setříděné patří před  $a_i$ . Při vyhledání  $a_i$  ve stromu vyjadřuje  $f_i$  počet listů nalevo od  $a_i$ . Čas vyhledání  $a_i$  je tedy  $O(\log f_i)$  a celkový čas vyhledání je  $O(\sum \log f_i)$ .

Hodnota  $F = \sum f_i$ , zvaná *počet inverzí*, vyjadřuje uspořádanost vstupní posloupnosti. Pro správně uspořádanou posloupnost je  $F = 0$ , pro obráceně uspořádanou posloupnost je  $F = n(n-1)/2$ . To jsou také mezní hodnoty, jichž může  $F$  nabývat.

Z vlastností logaritmu a srovnáním geometrického a aritmetického průměru dostáváme

$$\sum \log f_i = \log \prod f_i = n \log \sqrt[n]{\prod f_i} \leq n \log(F/n).$$

A-sort tedy vyžaduje čas  $O(n \max(1, \log((F+1)/n)))$ . V nejhorším případě to je  $O(n \log n)$  a Mehlhorn a Tsakalidis ukázali, že A-sort je lepší než Quicksort v případě, že  $F \leq 0.02n^{1.57}$ . Naproti tomu Insertsort, jednoduchý algoritmus, který postupně lineárním prohledáním zatřídí prvky pole do jeho již setříděného počátečního úseku, vyžaduje čas  $O(n + F)$ , což je v nejhorším případě  $O(n^2)$ .

Zbývá ještě zdůvodnit, proč použít (2,3)-stromy. Víme, že (2,3)-stromy mají nejmenší prostoro-  
 rové nároky mezi (a,b)-stromy. Na druhé straně však (2,3)-stromy v obecném případě vyžadují  
 zbytečně mnoho vyvažovacích operací, a proto jsou výrazně pomalejší než např. (2,4)-stromy. Pro-  
 tože však A-sort nepoužívá operaci DELETE, ukázali jsme (viz počet operací Štěpení), že pro  
 A-sort to není pravda. Zde (2,3)-stromy patří mezi nejrychleji pracující (a,b)-stromy.

### 8.3 Paralelní přístup do (a, b) stromů

Při operacích INSERT a DELETE jsme nejprve sestupovali stromem dolů až k listům, potom jsme se vraceli nahoru a štěpili nebo spojovali vrcholy. To znemožňuje dovolit paralelní přístup do

ošetřit  $\log 0$   
 nebo  
 transpozic?  
 standardní  
 termín?

stromu. Procesu, který je ve fázi vyhledání, by se mohlo stát, že mu jiný proces změní strom “pod rukama”. Stávající operace INSERT a DELETE tedy požadují výlučný přístup ke stromu.

Nyní předvedeme paralelní verzi těchto operací, kde se štěpení nebo spojování provádí již při sestupu. Potom již není nutné se vracet a je tedy možné rovnou odemykat části stromu, ke kterým již daný proces nebude přistupovat. Cenou za tento přístup jsou zbytečná štěpení/spojení.

Potřebujeme omezit  $b$ : podmínku  $b \geq 2a - 1$  zpřísníme na

$$4'. \quad a \geq 2 \text{ a } b \geq 2a$$

udělat obrázek  
ilustrující  
zbytečná š/s

### 8.3.1 Paralelní INSERT( $x$ ) do ( $a, b$ ) stromu

$o := \text{lock}(\text{nadkořen})$  {Nadkořen je implementační pomůcka. Slouží k zamknutí přístupu k celému stromu a uchovává  $\max(S)$ }

$t := \text{kořen}$

{Invariant mezi průchody cyklem:  $o$  je otec  $t$ ,  $o$  je jediný vrchol zamknutý tímto procesem.}

**while**  $t$  není list **do**

$i := 1$

**while**  $i < \rho_t \wedge H_t[i] < x$  **do**

$i := i + 1$

**end while**

$s := S_t[i]$

    {preventivní rozštěpení:}

**if**  $\rho(t) = b$  **then**

        rozděl  $t$  na  $t_1$  a  $t_2$ : {viz 4'}

        k  $t_1$  dej prvních  $\lfloor (b+1)/2 \rfloor$  synů  $t$

        k  $t_2$  dej zbylých  $\lceil (b+1)/2 \rceil$  synů  $t$

$t_1$  předchází  $t_2$

        uprav  $\rho_o, S_o$  a  $H_o$

        {implic.: uprav  $\rho_{t_1}, \dots, H_{t_2}$ }

        {při štěpení kořene ještě musíme vytvořit nový kořen}

$n := t_j$ , kde  $s$  je syn  $t_j$

**else**

$n := t$

**end if**

$\text{lock}(n)$

$\text{unlock}(o)$

$o := n$

$t := s$

**end while**

**if**  $t$  nereprezentuje  $x$  **then**

    vrcholu  $o$  přidej nového syna  $t'$  reprezentujícího  $x$

    zařaď  $t'$  na správné místo mezi jeho bratry a uprav  $\rho_o, S_o$  a  $H_o$

**end if**

$\text{unlock}(o)$

### 8.3.2 Paralelní DELETE( $x$ ) z ( $a, b$ ) stromu

$o := \text{lock}(\text{nadkořen})$  {Nadkořen je implementační pomůcka. Slouží k zamknutí přístupu k celému stromu a uchovává  $\max(S)$ }

$t := \text{kořen}$

$h := \text{nil}$  {Jakmile  $h \neq \text{nil}$ ,  $x \in H_h$  a  $h$  bude zamčený do konce procesu.}

{Invariant mezi průchody cyklem:  $o$  je otec  $t$ ,  $o$  je kromě  $h$  jediný vrchol zamknutý tímto procesem.}

**while**  $t$  není list **do**

$i := 1$



```

while  $i < \rho_t \wedge H_t[i] < x$  do
   $i := i + 1$ 
end while
if  $H_t[i] = x$  then
   $h := t$ 
end if
 $s := S_t[i]$ 
{preventivní spojení/přesun:}
if  $\rho(t) = a$  then
   $v :=$  bezprostřední bratr  $t$ 
  if  $\rho_v = a$  then {smíme spojit}
    {Spojení}
    sluč  $v$  a  $t$  do  $t$  {viz 4'}
    uprav  $\rho_o$ ,  $S_o$  a  $H_o$ 
     $t := o$ 
  else  $\{\rho_v > a$ , spojení by mělo více než  $b$  synů}
    {Přesun}
    přesuň krajního syna  $v$  do  $t$ 
    uprav  $H_o$ ,  $H_v$  a  $H_t$ 
  end if
end if
lock( $t$ )
if  $o \neq h$  then
  unlock( $o$ )
end if
 $o := t$ 
 $t := s$ 
end while
if  $t$  reprezentuje  $x$  then
  odstraň  $t$ 
  uprav  $H_o$ ,  $H_h$ 
  uprav  $S_o$  a  $\rho_o$ 
  unlock( $h$ )
end if
unlock( $o$ )

```

## 8.4 Složitost posloupnosti operací na $(a, b)$ stromu

A-sort funguje jednak proto, že v předtříděné posloupnosti rychle najde místo, kam se má vkládat, jednak proto, že se při samých INSERTech (*a díky správným  $a$ ,  $b$ ?*) provádí málo vyvažovacích kroků. V této sekci se podíváme na počet vyvažovacích kroků pro posloupnost operací INSERT a DELETE.

Nechť  $b \geq 2a$ .

**Věta 8.4.1.** *Mějme posloupnost  $n$  operací INSERT a DELETE aplikovanou na prázdný  $(a, b)$  strom. Označme  $P$  počet přesunů při provádění posloupnosti,  $SP$  počet spojení a  $ST$  počet štěpení. Dále označme  $P_h$ ,  $SP_h$  a  $ST_h$  počet přesunů, spojení a štěpení, které nastanou ve výšce  $h$  (listy mají výšku 0).*

*Nechť*

$$c = \min \left( \min \left( 2a - 1, \left\lceil \frac{b+1}{2} \right\rceil \right) - a, \right. \\ \left. b - \max \left( 2a - 1, \left\lfloor \frac{b+1}{2} \right\rfloor \right) \right) \quad (8.1)$$

Pak platí

$$P \leq n \quad (8.2)$$

$$(2c-1)ST + cSP \leq n + c + \frac{c}{a+c-1}(n-2) \quad (8.3)$$

$$P_h + SP_h + ST_h \leq \frac{2n^{c+2}}{(c+1)^h} \quad (8.4)$$

Platí  $c \geq 1$  (při  $b = 2a$  dokonce  $c = 1$ ). Z toho

$$ST + SP \leq \frac{n}{c} + 1 + \frac{n-2}{a}, \quad (8.5)$$

tedy lineárně vzhledem k  $n$ .

Pro paralelní verze INSERT a DELETE platí obdobná věta, když  $b \geq 2a + 2$ .

Pro důkaz použijeme *bankovní paradigma*: datovou strukturu ohodnotíme podle toho, jak je “uklizená”. Operace, které datovou strukturu “uklidí”, zvětší její “zůstatek na účtě”. Ty, které ji “naruší”, zůstatek zmenší. Potom najdeme vztah mezi zůstatkem a spotřebovaným časem. *Tohle pokulhává. Myslím jsem si, že zůstatek je něco jako čas v konzervě, který si pomalé operace berou od rychlých . . . , ale v tomhle případě to asi funguje jinak.*

( $a, b$ ) stromy jsou uklizené, když mají vrcholy počet synů někde uprostřed mezi  $a$  a  $b$ . Tehdy nenastane v brzké době vyvažovací operace. V tomto smyslu definujeme:

$$z(v) = \min(\rho_v - a, b - \rho_v, c) \quad v \text{ je vnitřní vrchol různý od kořene} \quad (8.6)$$

$$z(\text{kořen}) = \min(\rho_v - 2, b - \rho_v, c) \quad (8.7)$$

Pro strom  $T$  definujeme

$$z(T) = \sum_{v \in T} z(v)$$

$$z_h(T) = \sum_{\substack{v \in T \\ v \text{ má výšku } h}} z(v)$$

Platí

$$z(T) = \sum_h z_h(T)$$

Podobně jako u červenočerných stromů definujeme parciální ( $a, b$ )-strom:

**Definice 8.4.1.** ( $T, v$ ) je *parciální ( $a, b$ )-strom*, když  $v$  je vnitřní vrchol  $T$  různý od kořene a kromě  $v$  jsou splněny podmínky pro ( $a, b$ )-strom a  $a - 1 \leq \rho_v \leq b + 1$ .

Z definice zůstatku vyplývají tyto vlastnosti:

$$\rho_v = a - 1 \text{ nebo } b + 1 \implies z(v) = -1 \quad (8.8)$$

$$\rho_v = a \text{ nebo } b \implies z(v) = 0 \quad (8.9)$$

$$\rho_v = 2a - 1 \implies z(v) = c \quad (8.10)$$

$$\rho_u = \left\lfloor \frac{b+1}{2} \right\rfloor \wedge \rho_v = \left\lceil \frac{b+1}{2} \right\rceil \implies z(u) + z(v) \geq 2c - 1 \quad (8.11)$$

$$|\rho_u - \rho_v| \leq 1 \implies z(u) \geq z(v) - 1 \quad (8.12)$$

### 8.4.1 přidání/ubrání listu

Mějme ( $a, b$ )-strom  $T$  a přidáme nebo ubereme list, jehož otec je  $v$ . Pak vznikne parciální ( $a, b$ )-strom ( $T', v$ ) a platí:

$$z_1(T') \geq z_1(T) - 1 \quad (8.13)$$

$$z_h(T') = z_h(T) \quad h > 1 \quad (8.14)$$

$$z(T') \geq z(T) - 1 \quad (8.15)$$

vyjasnit

použiju  $z$  jako zůstatek místo  $b$  jako balance, protože souvislost s vyvažováním stromu je zde spíš matoucí

### 8.4.2 štěpení

Mějme parciální  $(a, b)$ -strom  $(T, v)$ , kde  $v$  je ve výšce  $h$ . Nechť  $T'$  vznikl štěpením  $v$ . Pak  $(T'$ , otec  $v$  je parciální  $(a, b)$ -strom a platí:

$$z_h(T') \geq 2c + z_h(T) \quad \text{z 8.8 a 8.11} \quad (8.16)$$

$$z_{h+1}(T') \geq z_{h+1}(T) - 1 \quad (8.17)$$

$$z_i(T') = z_i(T) \quad i \neq h, h+1 \quad (8.18)$$

$$z(T') \geq z(T) + 2c - 1 \quad (8.19)$$

### 8.4.3 spojení

Mějme parciální  $(a, b)$ -strom  $(T, v)$ , kde  $\rho_v = a - 1$  a  $v$  je ve výšce  $h$ ,  $y$  je bezprostřední bratr  $v$ . Nechť  $\rho_y = a$  a  $T'$  vznikl spojením  $v$  a  $y$ . Pak  $(T'$ , otec  $v$  je parciální  $(a, b)$ -strom a platí:

$$z_h(T') \geq c + 1 + z_h(T) \quad \text{z 8.8, 8.9 a 8.10} \quad (8.20)$$

$$z_{h+1}(T') \geq z_{h+1}(T) - 1 \quad (8.21)$$

$$z_i(T') = z_i(T) \quad i \neq h, h+1 \quad (8.22)$$

$$z(T') \geq z(T) + c \quad (8.23)$$

### 8.4.4 přesun

Mějme parciální  $(a, b)$ -strom  $(T, v)$ , kde  $\rho_v = a - 1$  a  $v$  je ve výšce  $h$ ,  $y$  je bezprostřední bratr  $v$ . Nechť  $\rho_y > a$  a  $T'$  vznikl přesunem syna od  $y$  k  $v$ . Pak  $T'$  je  $(a, b)$ -strom a platí:

$$z_h(T') \geq z_h(T) \quad \text{z 8.8, 8.9 a 8.12} \quad (8.24)$$

$$z_i(T') = z_i(T) \quad i \neq h \quad (8.25)$$

$$z(T') \geq z(T) \quad (8.26)$$

Nechť po skončení posloupnosti operací máme  $(a, b)$ -strom  $T_k$ . Sečteme předchozí výsledky:

$$z(T_k) \geq (2c - 1)ST + cSP - n \quad (8.27)$$

$$z_1(T_k) \geq 2cST_1 + (c + 1)SP_1 - n \quad (8.28)$$

$$z_h(T_k) \geq 2cST_h + (c + 1)SP_h - ST_{h-1} - SP_{h-1} \quad h > 1 \quad (8.29)$$

Vadí nám, že jsou ve výrazu i spojení a štěpení z jiné hladiny.

$$c \geq 1 \implies 2c \geq c + 1.$$

$$z_h(T_k) \geq (c + 1)(ST_h + SP_h) - ST_{h-1} - SP_{h-1}$$

$$ST_h + SP_h \leq \frac{z_h(T_k)}{c + 1} + \frac{ST_{h-1} + SP_{h-1}}{c + 1} \leq \frac{z_h(T_k)}{c + 1} + \frac{z_{h-1}(T_k)}{(c + 1)^2} + \frac{ST_{h-2} + SP_{h-2}}{(c + 1)^2} \quad (8.30)$$

$$\leq \left( \sum_{i=0}^{h-1} \frac{z_{h-i}(T_k)}{(c + 1)^{i+1}} \right) + \frac{ST_0 + SP_0}{(c + 1)^h} \quad j = h - i, \text{ rozšíříme } (c + 1)^{h-i} \quad (8.31)$$

$$= \left( \sum_{j=1}^h \frac{z_j(T_k)(c + 1)^j}{(c + 1)^{h+1}} \right) + \frac{n}{(c + 1)^h} \quad (8.32)$$

Nechť  $T$  je  $(a, b)$ -strom s  $m$  listy. Chceme shora odhadnout  $z(T)$ .

$$m_j = \text{počet vnitřních vrcholů různých od kořene} \begin{cases} \text{s právě } a + j \text{ syny} & \text{když } j \in \{0 \dots c-1\} \\ \text{s alespoň } a + j \text{ syny} & \text{když } j = c \end{cases} \quad (8.33)$$

Když  $v$  je vnitřní vrchol různý od kořene s právě  $a + j$  syny,  $j \in \{0 \dots c-1\}$ , pak  $z(v) \leq j$ .

Když  $v$  je vnitřní vrchol různý od kořene s alespoň  $a + c$  syny, pak  $z(v) \leq c$ .

Tedy

$$z(T) \leq c + \sum_{j=0}^c j m_j = * \quad (8.34)$$

Spočítáme hrany v  $T$ : nalevo jsou hrany vycházející z kořene a vnitřních vrcholů, napravo jsou hrany přicházející do vnitřních vrcholů a listů.

$$2 + \sum_{j=0}^c (a + j) m_j \leq \text{počet hran} = \binom{c}{\sum_{j=0}^c m_j} + m \quad (8.35)$$

Tedy  $m - 2 \geq \sum_{j=0}^c (a + j - 1) m_j$ .

$$* = c + \sum_{j=0}^c \frac{j}{a + j - 1} (a + j - 1) m_j \leq c + \sum_{j=0}^c \frac{c}{a + c - 1} (a + j - 1) m_j \leq c + \frac{c}{a + c - 1} (m - 2) \quad (8.36)$$

Spojením tohoto výsledku s 8.27 dostaneme 8.3.

K důkazu 8.4 využijeme 8.32.

$$m_{h,j} = \text{počet vnitřních vrcholů ve výšce } h \begin{cases} \text{s právě } a + j \text{ syny} & \text{když } j \in \{0 \dots c-1\} \\ \text{s alespoň } a + j \text{ syny} & \text{když } j = c \end{cases} \quad (8.37)$$

$$z_h(T) \leq \sum_{j=0}^c j m_{h,j} \quad (8.38)$$

$$\sum_{j=0}^c m_{h,j} = \text{počet vrcholů ve výšce } h \geq \sum_{j=0}^c (a + j) m_{h+1,j} \quad (8.39)$$

$$\sum_{j=0}^c j m_{h,j} \leq \sum_{j=0}^c m_{i-1,j} - a \sum_{j=0}^c m_{i,j} \quad (8.40)$$

$$\sum_{i=1}^h z_i(T_k) (c + 1)^i \leq \sum_{i=1}^h (c + 1)^i \left( \sum_{j=0}^c j m_{i,j} \right) \quad (8.41)$$

označme  $s_i = \sum_{j=0}^c m_{i,j}$

$$\stackrel{8.40}{\leq} \sum_{i=1}^h (c + 1)^i (s_{i-1} - a s_i) \quad (8.42)$$

$$= (c + 1) s_0 - (c + 1)^h a s_h + \sum_{i=2}^h (c + 1)^i \left( s_{i-1} - \frac{a}{c + 1} s_{i-1} \right) \quad (8.43)$$

$$\leq (c + 1) m \quad \text{protože } \frac{a}{c + 1} \geq 1 \text{ a } s_0 = m \quad (8.44)$$

$$ST_h + SP + h \leq \frac{m}{(c+1)^h} + \frac{n}{(c+1)^h} \leq \frac{2n}{(c+1)^h}$$

$$P_h \leq SP_{h-1} - SP_h \leq SP_{h-1} + ST_{h-1} \leq \frac{2n}{(c+1)^{h-1}}$$

Tím dostáváme 8.4:

$$ST_h + SP_h + P_h \leq \frac{2n(c+2)}{(c+1)^h}$$

## 8.5 Propojené $(a, b)$ stromy s prstem

Struktura vnitřního vrcholu  $v$ :

$\rho(v)$  = počet synů  $v$

$Syn[1..\rho(v)]$  je pole ukazatelů na syny vrcholu  $v$

$Hod[1..\rho(v)]$  je pole hodnot, platí

$Hod(i-1) <$  prvky reprezentované v podstromu  $i$ -tého syna  $\leq Hod(i)$

$otec(v)$  = ukazatel na otce  $v$

$\left. \begin{array}{l} \text{Předchůdce}(v) \\ \text{Následník}(v) \end{array} \right\}$  ukazatele na bezprostř. předchůdce (následníka)  $v$  na hladině vrcholu  $v$   
 (v lexikogr. uspořádání)

$h$  - hodnota, která leží mezi největším prvkem podstromu  $v$  a nejmenším prvkem podstromu následníka

zde chybí větší  
obrázek

### 8.5.1 Algoritmus MEMBER

Viz algoritmus 5.1

---

#### Algoritmus 8.1 MEMBER $(a, b)$ stromy s prstem

---

MEMBER( $x$ )

1) Necht'  $y$  je hodnota, na kterou ukazuje Prst.

**if**  $x < y$  **then**

    pokračuju 2)

**else**

    3)

**end if**

2)  $v \leftarrow otec(y)$

dokud  $x < h(\text{Předchůdce}(\text{Předchůdce}(v)))$

    jdu na otce( $\text{Předchůdce}(v)$ )

    v opačném případě

    když  $x \leq h(\text{Předchůdce}(v))$  pak

$v \leftarrow \text{Předchůdce}(v)$

    a pokračuji normálním vyhledáváním

3) symetrické ke 2)

---

### 8.5.2 Algoritmus FINGER

FINGER( $x$ )

nastaví hodnotu na list, který reprezentuje prvek nejbližší k  $x$ .

Použití:

když lze operace přirozeným způsobem rozdělit do segmentů a operace v ? segmentu mají operace blízko sebe

- vyhledání  $x$  vyžaduje čas  $O(1 + \log(l))$

- nastavím prst na nějakou vhodnou hodnotu

**Věta 8.5.1.** *Nechť  $T$  je propojovaný  $(a,b)$  strom s prstem a necht  $P$  je posloupnost příkazů MEMBER, INSERT, DELETE, FINGER, kterou provedeme na  $T$ . Pak  $P$  vyžaduje čas  $O(\log(n) + \text{časnavyhledn})$  kde  $n$  je velikost množiny reprezentované stromem  $T$ . ( $b \geq 2a$ )*

### 8.5.3 Amortizovaná složitost

Vezmeme posloupnost  $n$  operací, spočítáme maximální čas, který vyžadují a ten vydělíme  $n$ . Limita takto získaných čísel pro  $n \rightarrow \infty$  je amortizovaná složitost.

#### Bankovní paradigma

$D \xrightarrow{o} D'$

$D$  - vstupní situace

$o$  - operace

$D'$  - výstupní operace

Amortizovaná složitost operace  $o$  je  $\text{Čas}(O) + \text{bal}(D') - \text{bal}(D)$ , kde  $\text{bal}()$  je ohodnocení konfigurace.

$D_0 \xrightarrow{O_1} D_1 \xrightarrow{O_2} D_2 \rightarrow \dots \rightarrow D_n$

$$\sum_{i=1}^n \text{čas}(O_i) + \text{bal}(D_n) - \text{bal}(D_0) = \sum a(O_i) \leq \sum i(O_i)$$

Obvykle platí, že  $\text{bal} \geq 0$  nebo  $\text{bal} \leq 0$ .

Když  $\text{bal} \geq 0$ , pak:

$$\sum \text{čas}(O_i) \leq \sum a(O_i) + \text{bal}(D_0) \leq \sum i(O_i) + \text{bal}(D_0)$$

Když  $\text{bal} \leq 0$ , pak

$$\sum \text{čas}(O_i) \leq \sum a(O_i) - \text{bal}(D_n) \leq \sum i(O_i) - \text{bal}(D_0)$$

začínáme na prázdném  $(a,b)$  stromě  $\rightarrow \text{bal} = 0$ .

# Kapitola 9

## Samoopravující se struktury

Upravující algoritmy pracují na seznamech, mohou přemístit prvek, který je argumentem operace. (pokud zůstává v seznamu) Čas na vyhledání - to je pozice hledaného prvku. Pokud není v seznamu, je to délka seznamu + 1.

Pokud byl prvek na  $i$ -tém místě a přesune se na  $j$ -té, tak je-li

$j < i$ , provedou  $i-j$  volných výměn

$j > i$ , provedou  $j-i$  placených výměn

Volné výměny se nezapočítávají do složitosti. Pokud  $x$  není v seznamu při operaci  $\text{INSERT}(x)$ , tak předpokládejme, že je na 1. pozici po ukončení seznamu.

### 9.1 Amortizovaná složitost

XXX

### 9.2 Seznamy

XXX

#### 9.2.1 Algoritmus MEMBER

XXX

#### 9.2.2 Algoritmus INSERT

XXX

#### 9.2.3 Algoritmus DELETE

XXX

#### 9.2.4 Algoritmus MFR (Move Front Rule)

**Pravidlo MFR:** Při operaci  $\text{MEMBER}(x)$  je  $x$  v seznamu nebo při operaci  $\text{INSERT}(x)$  bude  $x$  po skončení operace na 1. místě seznamu.

přednáška z  
18.3.2003

**Věta 9.2.1.** *Mějme posloupnost  $P$  operací MEMBER, INSERT a DELETE a mějme dva prosté seznamy  $S_1, S_2$  množiny  $S$ .*

*Pak pro každý upravující algoritmus  $A$  platí:*

*Když MFR provede  $P$  na seznam  $S_1$  a  $A$  provede  $P$  na seznam  $S_2$ , tak platí:*

- a) čas MFR ≤ čas na vyhledání A + počet placených výměn A - počet volných výměn A - |P| když S1 = S2
- b) čas MFR ≤ čas na vyhledání A + počet placených výměn A - počet volných výměn A - |P| +  $\binom{|S1|}{2}$  když S1 ≠ S2

**Definice 9.2.1.** Necht S1, S2 jsou dva prosté seznamy množiny S, pak bal(S1,S2) je počet neuspořádaných dvojic x,y, x ≠ y, x,y ∈ S takových že x je před y v S1 a y je před x v S2.

**Poznámka 9.2.1.** Platí

bal(S1,S2) = 0 ⇔ S1 = S2 (prvky jsou ve stejném pořadí ⇔ seznamy jsou stejné)

bal(S1,S2) ≤  $\binom{|S1|}{2}$  (všechny dvojice jsou přeházené)

*Důkaz věty 9.2.1.* Přes amortizovanou složitost A.

Předpokládejme, že A i MFR mají provést operaci O.

A ... provádí na seznam S<sub>A</sub>, výsledek bude S'<sub>A</sub>

MFR .. provádí O na seznam S<sub>MFR</sub>, výsledek bude S'<sub>MFR</sub>

amortizovaná složitost operace O bude čas MFER pro operaci O + bal(S'<sub>A</sub>, S'<sub>MFR</sub>) - bal(S<sub>A</sub>, S<sub>MFR</sub>)

balance je def. vzhledem k algoritmu A.

Ukážeme, že amortizovaná složitost O pro MFR ≤ 2\*čas na vyhledání A + počet placených výměn A - počet volných výměn A - 1

$$S_A \xrightarrow{\text{vyhledání}} S''_A \xrightarrow{\text{výměny}} S'_A$$

$$S_{MFR} \rightarrow S'_{MFR} \rightarrow S''_{MFR}$$

kde po operaci

DELETE(x)	S'' <sub>A</sub> = S' <sub>A</sub>
MEMBER(x)	S'' <sub>A</sub> = S <sub>A</sub>
INSERT(x)	x je v seznamu, S'' <sub>A</sub> = S <sub>A</sub> x není v seznamu, S'' <sub>A</sub> vznikne z S' <sub>A</sub> přidáním x za poslední prvek seznamu

Podstatné je, že seznamy jsou nad stejnou množinou

Amort. složitost první části ≤ 2\*čas na vyhledání pro A - 1

Amort. složitost druhé části = počet placených výměn A - počet volných výměn A

- (i) Předpokládejme, že x není v seznamu a délka seznamů je n. Čas MFR je n+1, čas na vyhledání pro algoritmus je n+1 operace MEMBER(x) a DELETE(x) S''<sub>A</sub> = S'<sub>MFR</sub> a tedy amort. slož. MFR = čas operace = n+1 ≤ 2(n+1) - 1

n+1 je čas na vyhledání pro A - 1

S''<sub>A</sub> vznikne z S<sub>A</sub> přidáním x za posl. prvek S<sub>A</sub>

S'<sub>MFR</sub> vznikne z S<sub>MFR</sub> přidáním x na zač. seznamu tedy

$$bal(S''_A, S'_{MFR}) - bal(S_A, S_{MFR}) = n$$

Amort. slož. operace MFR = n+1 + n=2n + 1=2(n+1) - 1 = 2\*čas na vyhledání A - 1

- (ii) x je v seznamu. Předpokládejme, že x je na i-tém místě v seznamu S<sub>A</sub> na j-tém místě v seznamu S<sub>MFR</sub> Čas operace pro MFR je j, čas na vyhledání pro A je i. Označme k počet y v seznamu takových, že y je v S<sub>A</sub> za x, v S<sub>MFR</sub> před x.

Pak i+k ≥ j (i+k ≥ i-k+j) amort. slož. pro MFR = j + bal(S''<sub>A</sub>, S'<sub>MFR</sub>) - bal(S<sub>A</sub>, S<sub>MFR</sub>)

DELETE(x)

bal(S''<sub>A</sub>, S'<sub>MFR</sub>) - bal(S<sub>A</sub>, S<sub>MFR</sub>) ≤ -k amort. slož. ≤ j - k ≤ 2i - 1 = 2\*čas na vyhledání A - 1

MEMBER(x), INSERT(x)

bal(S''<sub>A</sub>, S'<sub>MFR</sub>) - bal(S<sub>A</sub>, S<sub>MFR</sub>) ≤ -k + i-1 (nějaké dvojice mohly přibýt) amort. slož. operace MFR ≤ j-k+i-1 ≤ i+i-1 = 2i - 1 = 2\*čas na vyhledání A-1



**Amort. složitost**

1. fáze operace  $\leq 2^*$ čas na vyhledání A-1 2. fáze operace = počet placených výměn A - počet volných výměn A Při placené výměně si v seznamu  $S''_A$  vymění x místo z za x, tedy dvojice x,z přibude při počítání  $bal(S'_A, S'_{MFR}) - bal(S''_A, S'_{MFR})$  (v  $S_{MFR}$  je x první) Při volné výměně se v seznamu  $S''_A$  vymění x místo s prvku u před x, tedy dvojice x,u se vynechá při počítání bal. Amort. slož.  $MFR \leq 2^*$ čas na vyhledání A + počet placených výměn A - počet volných výměn A - 1

Tedy platí:

čas posloupnosti P pro  $MFR \leq$  odhad amort. složitosti +  $bal(S_1, S_2) = 2^*$ čas na vyhledání v P algoritmem A + počet placených výměn A při P - počet volných výměn A při P -  $|P| + bal(S_1, S_2)$

když  $S_1 = S_2$  pak  $bal(S_1, S_2)=0$  a platí a)  $S_1 \neq S_2$  pak  $bal(S_1, S_2) \leq \binom{|S|}{2}$  a platí b) □

$|P| \dots$  za každou operaci je -1

**Poznámka 9.2.2.** S tímto jsme se setkali při EISCH je to důvod, proč je EISCH lepší než LISCH VICH lepší než LICH

**9.2.5 Algoritmus TR (Transposition Rule)**

Když je x při operaci MEMBER(x) a INSERT(x) na i-tém místě, tak ho dá na (i-1)-ní místo, při INSERT(x), kdy x není v seznamu, dá x na předposl. místo.

**Poznámka 9.2.3.** Lze najít posloupnost příkazů P lib. délky, že MFR vyžaduje čas ( $|P|$ ) a TR vyžaduje čas ( $|P|^2$ ). Na druhou stranu očekávaný čas TR  $\leq$  očekávaný čas MFR.

Chceme spočítat očekávaný čas MFR pro posloupnosti P aplikované na seznam S, kde P obsahuje jen operace MEMBER(x) pro  $x \in S$ .

Předpokládejme, že  $S=1,2, \dots, n$  a  $\beta_1 =$  pravděpodobnost operace MEMBER(x) pro  $x \in S$ .  $S = \{1,2,3\} \dots$  stavy Markovova řetězce jsou všechny permutace S pravděpodobnost přechodu je pst. operace převádějící jeden stav do druhého

chybí obr.

Tyto Markovovy řetězce jsou nerozložitelné a aperiodické a to znamená, že existují asymptot. pravděpodobnosti, tj. pro seznam  $\Pi$  je dána pravděpodobnost  $\kappa_\Pi$ , že po provedení náhodné posloupnosti P s daným rozložením operací skončíme u seznamu  $\Pi$ .

Pak očekávaný čas je  $\sum_\Pi \kappa_\Pi \sum_i \beta_i \Pi(i)$ ,  $\Pi(i)$  je pozice i v seznamu  $\Pi$ .  $p_1 = \sum_\Pi \kappa_\Pi \Pi(i) \dots$  očekávaná pozice prvku i  $\delta(j, i) =$  asymptot. pst., že prvek j je před i, pak platí

$$\delta(j, i) = \sum \{ \kappa_\Pi, \Pi \text{ seznam}, \Pi(j) < \Pi(i) \}$$

pak

$$p_i = \sum_\Pi \kappa_\Pi \Pi(i) = \sum_\Pi \kappa_\Pi (1 + |j, \Pi(j) < \Pi(i)|) = 1 + \sum_j j, \Pi \{ \kappa_\Pi, \Pi(j) < \Pi(i) \} = 1 + \sum_j \delta(j, i)(1) \quad (9.1)$$

Zkusíme  $\delta(j, i)$  spočítat jiným způsobem:

Idea: jak se může stát, že ve výsledném seznamu je j před i ? V posloupnosti P existovala operace MEMBER(x) a po ní se už nevyskytovala operace MEMBER(i) ani MEMBER(j).

Jaká je pravděpodobnost tohoto jevu ?

$$\beta_j \sum_{k=0}^{\infty} [1 - (\beta_i - \beta_j)]^k = \beta_j \frac{1}{1 - (1 - (\beta_i + \beta_j))} = \frac{\beta_j}{\beta_j + \beta_i} \stackrel{(1)}{=} 1 + \sum_{\substack{j,i \\ j \neq i}} \frac{\beta_j}{\beta_j + \beta_i} \quad (9.2)$$

očekávaný čas operace je

$$\sum_i \beta_i p_i = \sum_{\substack{j,i \\ j \neq i}} \frac{\beta_i \beta_j}{\beta_i + \beta_j}$$

Předpokládejme, že  $\beta_1 \geq \beta_2 \geq \dots \geq \beta_n$  pak nejrychlejší algoritmus na seznam  $x_1 - x_2 - \dots - x_n$  je klasický algoritmus bez přemísťování prvků. Klasický algoritmus je takový, algoritmus, který předem ví, jaké jsou pravděpodobnosti přístupu a má předem seznam srovnaný sestupně podle těchto pravděpodobností. Očekávaný čas tohoto algoritmu je

$$\begin{aligned} \sum_{i=1}^n i \beta_i &= \\ 1 + \sum_{i,j=1}^n 2 \frac{\beta_j \beta_i}{\beta_i + \beta_j} &\leq 1 + \sum_{\substack{i,j \\ j < i}} 2 \beta_i = 1 + \sum_{i=1}^n 2(i-1) \beta_i = 1 + 2 \cdot \sum_i i \beta_i - 2 \sum_i \beta_i = \\ &2 \sum_{i=1}^n i \beta_i - 1 \quad (9.3) \end{aligned}$$

Platí

$$\frac{\beta_j}{\beta_j + \beta_i} \leq 1$$

## 9.3 Splay stromy

Datová struktura - binární vyhledávací stromy s ohodnocenými prvky

### 9.3.1 Operace SPLAY

Základní operací je pro práci s těmito stromy je  $SPLAY(x)$ , která zjistí, zda  $x$  je reprezentován v dané množině. Pokud  $x$  leží v množině, algoritmus ho přemístí do kořene.

Když  $x$  neleží v množině, pak algoritmus přemístí do kořene buď nejmenší prvek větší než  $x$  nebo největší prvek menší než  $x$  (který leží v reprezent. množině)

### 9.3.2 Podporované operace

MEMBER, INSERT, DELETE, JOIN2( $T_1, T_2$ ), JOIN3( $x, T_1, T_2$ ) (nebo asi taky JOIN3( $T_1, x, T_2$ )), SPLIT( $x$ ), CHANGEWEIGHT( $x, \Delta$ ).

- JOIN2( $T_1, T_2$ )  
předpokládá, že  $\forall$  prvky reprezentované  $T_1 < \forall$  prvky reprezentované  $T_2$   
výsledný strom reprezentuje  $T_1 \cup T_2$ .
- JOIN3( $T_1, x, T_2$ )  
předpokládá, že  $\forall$  prvky reprezentované  $T_1 < x < \forall$  prvky reprezentované  $T_2$   
výsledný strom reprezentuje  $T_1 \cup T_2 \cup x$ .
- SPLIT( $x$ )  
výsledek: strom  $T_1 : \forall$  prvky  $\in T_1 < x$   
strom  $T_2 : \forall$  prvky  $\in T_2 > x$   
+ informace, zda  $x$  ležel v reprezentované množině
- CHANGEWEIGHT( $x, \Delta$ )  
zjistí, zda  $x$  leží ve stromě a pokud ano, pak k jeho váze přičte  $\Delta$ .

### 9.3.3 Algoritmus MEMBER

Viz algoritmus 9.1

---

#### Algoritmus 9.1 MEMBER pro Splay stromy

---

```
SPLAY(x)
  if x je reprezentován v kořeni then
    "x je v S"
  else
    "x není v S"
  end if
```

---

### 9.3.4 Algoritmus JOIN2

Viz algoritmus 9.2

---

#### Algoritmus 9.2 JOIN2( $T_1, T_2$ )

---

```
SPLAY( $T_2, -\infty$ ) // nejmenší prvek
kořen  $T_1$  bude levý syn kořene  $T_2$ 
```

---

Tím se z  $T_2$  stane strom, jehož kořen bude (list, vrchol) a místo toho listu navěšíme  $T_1$ . Pak budou v levém podstromu všechny prvky menší než hodnota v rootu a v pravém všechny větší, což chceme.

chybí obrázek

### 9.3.5 Algoritmus JOIN3

Viz algoritmus 9.3

---

#### Algoritmus 9.3 JOIN3( $T_1, x, T_2$ )

---

```
vytvoříme vrchol t reprezentující x
kořen  $T_1$  je levý syn t
kořen  $T_2$  je pravý syn t
```

---

Vytvoříme nový vrchol reprezentující  $x$  a jeho synové budou  $T_1$  – levý,  $T_2$  – pravý.

### 9.3.6 Algoritmus SPLIT

Viz algoritmus 9.4

---

#### Algoritmus 9.4 SPLIT(x)

---

```
SPLAY(x)
y = prvek reprezentovaný kořenem
 $T_1$  = podstrom levého syna kořene
 $T_2$  = podstrom pravého syna kořene
if y = x then
  výstup  $T_1, T_2$ 
else if y < x then
  výstup  $T \setminus T_2, T_2$ 
else
  výstup  $T_1, T \setminus T_1$ 
end if
```

---

zde chybi  
obrazek, ale  
celkem není pro  
pochopení  
potřeba :)

### 9.3.7 Algoritmus DELETE

Viz algoritmus 9.5

---

#### Algoritmus 9.5 DELETE(x)

---

```
SPLAY(x)
if kořeni reprezentuje x then
    T1 je podstrom levého syna kořene T
    T2 je podstrom pravého syna kořene T
    T ← JOIN2(T1, T2)
end if
```

---

jiný zápis:  $T_1, T_2 \leftarrow SPLIT(x, T)$   $T \leftarrow JOIN2(T_1, x, T_2)$

### 9.3.8 Algoritmus INSERT

Viz algoritmus 9.6

---

#### Algoritmus 9.6 INSERT(x)

---

```
SPLAY(x)
if kořeni nereprezentuje x then
    if kořen stromu reprez. prvek < x then
        T2 je podstrom pravého syna kořene
        T1 = T - T2
    else
        T1 je podstrom levého syna kořene
        T2 = T - T1
    end if
    JOIN3(T1, x, T2)
end if
```

---

jiný zápis:  $T_1, T_2 \leftarrow SPLIT(x, T)$   $T \leftarrow JOIN3(T_1, x, T_2)$

### 9.3.9 Algoritmus CHANGEWEIGHT

Viz algoritmus 9.7

---

#### Algoritmus 9.7 CHANGEWEIGHT(x, $\Delta$ )

---

```
SPLAY(x)
if x je v kořeni then
    k váze x přičti  $\Delta$ 
end if
```

---

Předpokládejme, že  $w(x)$  je váha prvku a je to kladné celé číslo.  $tw(x)$  - totální váha  $x$ , je to součet vah všech prvků v podstromě určeném  $x$

Př.

$$tw(a) = w(a) + w(b) + w(c)$$

$r(x)$  je rank(x)  $r(x) = \lfloor \log tw(x) \rfloor$

$$bal(konfigurace) = \sum \{r(x) : x \in konfigurace\}$$

Pro strom  $T$  je  $tw(x) = tw(\text{kořen } T)$   $r(T) = r(\text{kořen } T)$

**Lemma 9.3.1.** *Necht  $T$  je binární vyhledávací strom,  $t$  je vnitřní vrchol a  $u, v$  jsou synové  $t$ . Pak  $r(t) > \min\{r(u), r(v)\}$  ( $r(list) = -\infty$ ).*

chybí obrázek

Důkaz. Předpokládejme, že  $tw(u) \leq tw(v)$

$$r(t) = \lfloor \log tw(t) \rfloor \geq \lfloor \log 2tw(u) \rfloor = 1 + \lfloor \log tw(u) \rfloor = 1 + r(u)$$

□

### 9.3.10 Algoritmus SPLAY

Viz algoritmus 9.8

---

#### Algoritmus 9.8 CHANGEWEIGHT( $x, \Delta$ )

---

```

SPLAY(x)
t ← kořen
while t není list a t reprezentuje x do
  if x < t then
    t ← levý syn t
  else
    t ← pravý syn t
  end if
end while
if t je list then
  t ← otec(t)
end if
while t není kořen do
  if otec(t) je kořen then
    rotace(t, otec(t))
  else
    if otec(t) i t jsou leví synové (praví) then
      rotace(otec(t), děd(t))
      rotace(t, otec(t))
    else
      dvojitá rotace(t, otec(t), děd(t))
    end if
  end if
end if
end while

```

---

t se po skončení operace SPLAY(x) dostane do kořene

chybí obrázky

### 9.3.11 Amortizovaná složitost SPLAY

čas operace SPLAY = počet opakování cyklu, když vrchol t transportujeme do kořene

**Lemma 9.3.2.** *Amortizovaný čas operace  $SPLAY(x, T) \leq 3(r(T) - r(t)) + 1$ , kde t je vrchol, který transportujeme do kořene. (když x je pvekem repoz. množiny, pak t reprezentuje x, jinak je to buď největší nebo nejmenší pveke menší (větší) než x)*

Důkaz. rozdělíme podle akce, která se provádí ve while cyklu a) while cyklus provádí rotace  
 Amortizovaná složitost tohoto kroku = čas operace + bal(nová konf.) - bal(původní konf.)  
 $= 1 + r'(u) - r(v) \leq 1 + r(u) + r(v)$   
 $\leq 1 + 3(r(u) - r(v))$   
 protože x má v původním i novém stromě stejné prvky  $r(u) = r'(t)$   
 $r'(u) \leq r'(t) = r(u)$   
 b) while cyklus provádí dvojitou rotaci

chybí obrázek  
 tady je děsný  
 zmatek

nečitelné

tady chybi obr.

Amortizovaná složitost této operace

$$\begin{aligned}
 &= \text{čas operace} + \text{bal}(\text{nová konf.}) - \text{bal}(\text{původní konf.}) = \\
 &= 1 + r'(u) - r(v) - r(u) - r(t) \quad (9.4)
 \end{aligned}$$

pro  $x \neq t, u, v$  platí  $r(x) = r'(x)$   
 $r(v) = r'(t)$   
 b1)

$$r(v) > r(t), \text{ pak } r'(u), r'(v) \leq r'(t) = r(v)$$

$$\begin{aligned}
 r(u) \geq r(t), 1 \leq r(v) - r(t) &\stackrel{9.4}{\leq} \\
 r(v) - r(t) + 2r(u) - 2r(t) &= 3(r(v) - r(t)) \quad (9.5)
 \end{aligned}$$

b2)  $r(v) = r(t)$ , pak podle lemmatu  $r'(t) > \min\{r'(u), r'(v)\}$  platí

$$\begin{aligned}
 2r'(t) \geq r'(u) + r'(v) + 1 &\stackrel{9.4}{\leq} \\
 2r(u) - 2r(t) = 2(r(v) - r(t)) &= (r(t) = 0)3(r(u) - 3r(t)) \quad (9.6)
 \end{aligned}$$

□

### 9.3.12 Amortizovaná složitost ostatních operací

XXX

# Kapitola 10

## Haldy

**Definice 10.0.1.** Haldy jsou stromové struktury, které splňují

- lokální podmínku na uspořádání - prvek reprezentující otce je menší než prvek reprezentovaný synem apod.
- strukturální podmínku na stromy, ze kterých jsou vytvořené

**Poznámka 10.0.1.** podle těchto podmínek se haldy rozdělují na Fibonacciho, Leftist, d-regulární apod. (mohou se lišit jak lokální, tak strukturální podmínkou)

### 10.1 *d*-regulární haldy

**Definice 10.1.1.** *d*-regulární halda, *d* celé číslo  $d \geq 2$

Je to strom *T* takový, že existuje jednoznačná korespondence mezi vrcholy stromů a prvky reprezentované množiny a platí:

- 1) strom *T* splňuje strukturální podmínky
  - (a) každý vrchol s výjimkou nejvýše jednoho je buď list nebo má *d* synů
  - (b) každý vrchol má nejvýše *d* synů
  - (c) existuje očíslování synů každého vrcholu tak, že po očíslování průchodem šířky platí: když vrchol není list, pak každý vrchol s menším číslem má *d* synů.
- 2) podmínku na lokální uspořádání: když *x* je prvek přiřazený vrcholu *t*, pak  $\text{otci}(t)$  je přiřazen prvek  $\leq x$  pak po očíslování průchodem do šířky platí: když vrchol má číslo *i*, jeho synové mají čísla  $d(i-1)+2, d(i-1)+3, \dots, di+1$  a otec má číslo  $\lceil \frac{i-1}{d} \rceil$ .

Př. 3-regulární halda

když takto očíslované prvky dáme do pole, pak platí: když je vrchol na *i*-tém místě, čísla synů jsou  $3(i-1)+2, 3i, 3i+1$  a otec je na  $\lceil \frac{i-1}{3} \rceil$  místě v poli. to využijeme pro implementaci polem - ušetříme místo.

**Poznámka 10.1.1.** nejpobulárnější jsou 2-reg. haldy, protože synové *i*-tého vrcholu jsou na místech  $2(i-1)+2 = 2i, 2(i-1)+3 = 2i+1$ , otec je na  $\lceil \frac{i-1}{2} \rceil + 1 = \lceil \frac{i}{2} \rceil$ .  $\Rightarrow$  snadné počítání (bitový posun)

#### 10.1.1 Algoritmus UP

UP(*x*) ... srovná haldu směrem nahoru

A: if prvek reprezentovaný *x* je < prvek reprezentovaný otcem(*x*) then *x* a otcem(*x*) vyměníme a pokračujeme v A

### 10.1.2 Algoritmus DOWN

A: if prvek reprezentovaný  $x >$  prvek reprezentovaný některým synem  $x$  then vyměníme  $x$  a syna  $x$ , který reprezentuje nejmenší prvek, pokračujeme v A

**Poznámka 10.1.2.** když má hlada hloubku  $h$ , pak UP( $x$ ) vyžaduje čas  $O(h)$ , DOWN( $x$ ) čas  $O(dh)$ .

### 10.1.3 Operace na haldě

#### INSERT

přidáme poslední list  $t$  reprezentující  $x$  UP( $t$ )

#### MIN

... prvek reprezentovaný v kořeni

#### DELETEMIN

prvek reprezentovaný posledním listem dáme do kořene, odstraníme poslední list DOWN(kořen)

#### DECREASEKEY( $x, \Delta$ )

musíme znát polohu vrcholu  $t$  reprezentujícího  $x$ , toto halda neumožňuje nalézt.  
změníme uspořádání v bodě  $x$   
UP( $x$ ) mohl by být menší než jeho otec, proto provedeme UP

#### INCREASEKEY( $x, \Delta$ )

musíme znát polohu vrcholu  $t$  reprezentujícího  $x$ , toto halda neumožňuje nalézt  
změníme uspořádání v bodě  $x$   
DOWN( $x$ )

#### DELETE

Musíme znát polohu vrcholu  $t$  reprezentujícího  $x$ , toto halda neumožňuje nalézt  
Vezmeme prvek  $y$  reprezentovaný posledním listem, odstraníme poslední list, prvek  $t$ , který reprezentoval  $x$  bude reprezentovat  $y$

```

if  $y < x$  then
    UP( $t$ ) else DOWN( $t$ )
end if

```

### 10.1.4 Algoritmus MAKEHEAP

Dána prostá posloupnost  $x_1, x_2, \dots, x_n$ . Chceme vytvořit  $d$ -reg. haldu reprezentující množinu  $x_1, x_2, \dots, x_n$ .  
Vezmeme "d-reg. strom"  $T$  s vrcholy přiřadíme prvky  $x_1, x_2, \dots, x_n$ . Pro všechny vrcholy, které nejsou listy podle očíslování v pořadí od největšího k nejmenšímu provedeme DOWN( $t$ ).

chybí obrázek

Invariant: v okamžiku, kdy provádím DOWN( $t$ ), tak vrcholy, které reprezentující větší prvky splňují směrem dolů podmínku



### 10.1.5 Složitost operací

V d-reg. haldě reprezentující n-prvkovou množinu implementace operací vyžaduje časy dané tabulkou:

Operace	složitost
MIN	$O(1)$
INSERT, DECREASEKEY	$O(\log_d(n))$
DELETEMIN, INCREASEKEY, DELETE	$O(d \cdot \log_d(n))$

Máme vrchol v i-té hladině a "d-reg. strom" má hloubku h. Kolik času potřebuje DOWN(t) ? Je to  $O(d(h-1))$  počet vrcholů v i-té hladině je  $d^i$  čas MAKEHEAP je  $O(\sum_{i=0}^h d^i) = O(dS)$ , kde

$$S = \sum_{i=0}^h d^i = \frac{d^{h+1} - 1}{d - 1}$$

Budeme počítat

$$\begin{aligned} dS - S &= \sum_{i=0}^h d^{i+1} - \sum_{i=0}^h d^i = d^{h+1} - 1 \\ &= d^h - h + \sum_{i=0}^h d^i = d^h - h + \frac{d^{h+1} - 1}{d - 1} \\ &\Rightarrow S = \frac{d^h - h}{d - 1} + \frac{d^{h+1} - 1}{(d - 1)^2}, h = \log_d(n) \Rightarrow S \approx O\left(\frac{n}{d}\right) \quad (10.1) \end{aligned}$$

### 10.1.6 Dijkstrův algoritmus

K čemu jsou d-reg. haldy dobré ? např. pro implementaci Dijkstrova algoritmu.

Vstup: orientovaný graf (V,E), fce  $c : E \rightarrow R^+$ , vrchol z

Výstup:  $d(v)$ ,  $v \in V$

$d(v)$  je délka nejkratší cesty ze z do v

---

#### Algoritmus 10.1 Dijkstra pro d-regulární haldy

---

$d(z) = 0, d(v) = \infty \forall v \in V, v \neq z, U = z$

**while**  $U \neq \emptyset$  **do**

    vezmeme z U prvek  $u \in U$  s nejmenší hodnotou  $d(u)$ ,

    odstraníme ho z U.

**for**  $\forall (u, v) \in E$  **do**

**if**  $d(v) > d(u) + c(u, v)$  **then**

$d(v) = d(u) + c(u, v)$ , v přidáme do U

**end if**

**end for**

**end while**

---

U reprezentujeme pomocí d-reg. haldy. Pak čas Dijkstrova algoritmu je  $O(|V| \cdot \text{čas na INSERT} + |V| \cdot \text{čas na DELETEMIN} + |E| \cdot \text{čas na DECREASEKEY})$

když  $d = 2$ , pak to je  $O(|E| \log_2(|V|))$   $d = \max\left\{\frac{|E|}{|V|}, 2\right\}$ , vyjde čas  $O(|E| \log_d(|V|))$

když  $\exists \epsilon$ , že  $|E| \geq c|V|^{1+\epsilon}$  pro nějaké c, pak čas je  $O(|E|)$ . (graf je dostatečně hustý)

$|E| \geq c|V| \log^\epsilon |V|$  pro nějaké c,  $\epsilon$ , pak čas  $O(|E| \log \log |V|)$ .

Další aplikace: třídění

HEAPSORT - viz. alg. 10.2

Vstup: prostá posloupnost prvků  $x_1, x_2, \dots, x_n$ .

Výstup: uspořádaná posl. prvků  $x_1, x_2, \dots, x_n$ .

**Algoritmus 10.2** Heapsort pro d-regularní haldy

---

```

MAKEHEAP( $x_1, x_2, \dots, x_n$ )
i = 1
while HEAP  $\neq \emptyset$  do
   $x_1 = \text{MIN}(\text{HEAP})$ 
  DELETETEMIN(HEAP)
  i = i + 1
end while

```

---

**Poznámka 10.1.3.** Optimum pro d-reg. haldy je někde mezi  $d=6$  a  $d=7$ .

## 10.2 Leftist haldy

**Definice 10.2.1.** LEFTIST strom

Mějme binární strom a pro každého syna máme určeno, zda je levý nebo pravý. Pro vrchol  $v$  definujeme  $\text{npl}(v)$  jako délku nejkratší cesty z  $v$  do vrcholu  $v$  podstromu v s nejvýše jedním synem.

Binární strom je LEFTIST, když

a když  $v$  má jednoho syna, pak je to levý syn  $b$  když  $v$  má dva syny, pak  $\text{npl}(\text{levého syna}) \geq \text{npl}(\text{pravého syna})$

**Definice 10.2.2.** Cesta  $x_1, x_2, \dots, x_n$  se nazývá pravá, když  $x_i$  je pravý syn  $x_{i-1}$  pro  $i=2,3,\dots,n$  a  $x_n$  nemá pravého syna.

Vlastnosti:

- 1) každý podstrom leftist stromu je leftist
- 2) délka pravé cesty z  $\forall$  vrcholu  $v$  je  $\leq \log(\text{počet vrcholů v podstromu vrcholu } v)$

**Definice 10.2.3.** Letist halda reprezentující množinu  $S$  je leftist strom  $T$  s  $n$  vrcholy takový, že existuje jednoznačná korespondence mezi prvky  $S$  a vrcholy  $T$  taková, že  $\forall v$  prvek přiřazený vrcholu  $v$   $\geq$  prvek přiřazený otci  $v$ .

**Definice 10.2.4.** MERGE( $T_1, T_2$ )

podmínka:  $T_1, T_2$  reprezentují disjunktní množiny  $S_1, S_2$ . Výsledek: halda reprezentující  $S_1 \cup S_2$ .

**Poznámka 10.2.1.** podobné operaci JOIN v AVL stromech (ale...)

**Poznámka 10.2.2.** Časová složitost operace MERGE v leftist haldách je  $O(\log(n_1 + n_2))$ , kde  $n_1, n_2$  jsou velikosti reprezentovaných množin.

**Věta 10.2.1.** Operace MIN v leftist haldách vyžaduje čas  $O(1)$ , operace MERGE, INSERT, a DELETETEMIN vyžadují čas  $O(\log n)$ , kde  $n$  je počet prvků ve výsledné haldě.

**Poznámka 10.2.3.** koukneme se jak vypadá výsledný strom a podíváme se na vrcholy, se kterými jsme něco museli provádět - tyto vrcholy leží na pravé cestě, tj. je jich omezený počet.

**Poznámka 10.2.4.** npl, které jsem musel přepisovat je vždycky pravý syn

**Věta 10.2.2.** Operace DECREASEKEY, INCREASEKEY a DELETE vyžadují čas  $O(\log n)$  v leftist haldách (a počet prvků výsledné reprezent. množiny)

---

**Algoritmus 10.3** MERGE pro leftist haldy

---

```

MERGE( $T_1, T_2$ )
if  $T_1 = 0$  then
    MERGE( $T_1, T_2$ )  $\rightarrow T_2$  konec
end if
if  $T_2 = 0$  then
    MERGE( $T_1, T_2$ )  $\rightarrow T_1$  konec
end if
if kořen  $T_2$  reprezentuje prvek  $<$  prvek repr. kořenem  $T_1$  then
    vyměníme  $T_1$  a  $T_2$ 
end if
pravý syn kořene  $T_1 \rightarrow$  MERGE( $T_2$ , podstrom pravého syna kořene  $T_1$ )
if  $npl(\text{levého syna kořene } T_1) < npl(\text{pravého syna kořene } T_1)$  then
    prohodím syny kořene  $T_1$ 
end if
 $npl(\text{kořene } T_1) = npl(\text{pravého syna kořene } T_1) + 1$ 
MERGE( $T_1, T_2$ )  $\rightarrow T_1$ 

```

---



---

**Algoritmus 10.4** INSERT pro leftist haldy

---

```

INSERT(x)
vytvoříme novou haldu  $T_1$  reprezentující pouze prvek x
 $T \rightarrow$  MERGE( $T_1, T_2$ )
DELETEMIN
 $T_1 \rightarrow$  podstrom levého syna kořene T
 $T_2 \rightarrow$  podstrom pravého syna kořene T
 $T \rightarrow$  MERGE( $T_1, T_2$ )

```

---



---

**Algoritmus 10.5** DECREASEKEY pro leftist haldy

---

```

DECREASEKEY(x)
odtrhneme podstrom  $T_1$  vrcholu x,  $y \rightarrow$  otec(x)
 $T_2 = T - T_1$ 
zmenšíme ohodnocení kořene stromu  $T_1$ 
if y má jen pravého syna then
    změníme tohoto syna na levého,  $npl(y) = 0$ 
end if
 $y \rightarrow$  otec(y)
while  $npl(y) > \min\{npl(\text{levý syn } y), npl(\text{pravý syn } y) + 1\}$  do
    if  $npl(\text{levého syna } y) \leq npl(\text{pravého syna } y)$  then
        prohodíme syny y
    end if
     $npl(y) = npl(\text{pravého syna } y) + 1$ ,  $y \rightarrow$  otec(y)
end while
 $T \rightarrow$  MERGE( $T_1, T_2$ )

```

---

## 10.3 Binomiální haldy

jsou založeny na binomiálním rozvoji čísel, jsou definovány na speciálních stromech.

Vlastnosti stromů  $H_i$

- $H_i$  má  $2^i$  vrcholů
- hloubka  $H_i$  je  $i$

- kořen  $H_i$  má i synů
- $\forall j < i$  existuje syn kořene  $H_i$  takový, že jeho podstrom je izomorfní s  $H_j$ .

*Důkaz.* Důkaz: indukcí přes  $i$  (elementární) □

**Definice 10.3.1.** Binomická halda je soubor stromů takových, že

- každý strom je izomorfní s nějakým  $H_i$
- žádné dva stromy nejsou izomorfní
- existuje jednoznačná korespondence mezi vrcholy reprezentované množiny a vrcholy stromů taková, že prvek odpovídající otcí je menší než prvek odpovídající vrcholu.

### 10.3.1 Operace

---

**Algoritmus 10.6** MERGE pro binomialni haldy

---

```

MERGE( $T_1, T_2$ )
( $T_1, T_2$  binom. haldy velikosti  $n_1, n_2$ )
P = 0, i = 0, T = 0
while i  $\leq$   $\log(n_1 + n_2)$  do
   $S_1$  je strom v  $T_1$  izomorfní s  $H_i$  (pokud neexistuje, tak  $S_1 = 0$ )
   $S_2$  je strom v  $T_2$  izomorfní s  $H_i$  (pokud neexistuje, tak  $S_2 = 0$ )
  if  $S_1, S_2, P = 0$  then
    neprovedeme nic
  end if
  if jeden strom z  $S_1, S_2, P$  je neprázdný then
    vložím tento strom do T, P=0
  end if
  if dva stromy z  $S_1, S_2, P$  jsou neprázdné then
    spojím tyto stromy a výsledek vložím do P
  end if
  if všechny stromy z  $S_1, S_2, P$  jsou neprázdné then
    vložím do T, spojení  $S_1, S_2$  vložím do P
  end if
  i = i + 1
end while

```

---

**Poznámka 10.3.1.** P odpovídá přenosu v binárním sčítání, T je výsledná halda

---

**Algoritmus 10.7** Spojení dvou binomi. hald

---

```

Spojení( $S_1, S_2$ )
 $S_1, S_2$  jsou stromy izomorfní s  $H_i$  pro nějaké i
if prvek reprezentovaný kořenem  $S_1 \leq$  prvek reprezentovaný kořenem  $S_2$  then
  kořen  $S_2$  se stane dalším synem kořene  $S_1$ 
else
  kořen  $S_1$  se stane dalším synem kořene  $S_2$ 
end if

```

---

**Poznámka 10.3.2.** INSERT - stejný jako v leftist haldách

MIN - prohledáme prvky reprezentované kořeny stromů a najdeme nejmenší

---

**Algoritmus 10.8** DELETETEMIN pro binom. haldy

---

DELETETEMIN

prohledáním prvků reprezentovaných kořeny stromů nalezneme strom  $S$ , jehož kořen reprezentuje nejmenší prvek $T_1 = T \setminus S, T_2$  je tvořen podstormy všech synů kořene  $S$ 

(tj. utrhneme kořen a zbytek dáme do haldy) - je to halda díky vlastnosti 4

 $T \rightarrow \text{MERGE}(T_1, T_2)$ 

---

**Poznámka 10.3.3.** operace DELETE se nedá rozumně provést, museli bychom přebudovat celý strom

**Věta 10.3.1.** Operace MERGE, INSERT, MIN, DELETETEMIN a DECREASEKEY vyžadují čas  $O(\log n)$ . Operace INCREASEKEY vyžaduje čas  $O(\log^2 n)$ .

**Poznámka 10.3.4.** sčítání v binárních číslech  $O(1)$  1 0 0 ... 0 1 1 1 ... 1 amort. slož. neplatí něco podobného pro binom. stromy? Ano, pro operaci INSERT.

**Věta 10.3.2.** Amortizovaná složitost operace INSERT je  $O(1)$ .

**Poznámka 10.3.5.** MERGE zabírá dost času - musíme ho dělat?

### 10.3.2 Líná implementace binom. hald

změníme definici - vynecháme podmínku 2, tj. teď v naší binom. haldě mohou být izomorfní stromy.

Operace  $\text{MERGE}(T_1, T_2)$  - provedeme konkatenaci seznamů  $T_1$  a  $T_2$ . Jenom to by nefungovalo, musíme ještě změnit operace MIN, DELETETEMIN.

---

**Algoritmus 10.9** DELETETEMIN pro line binom. haldy

---

MIN

při prohledávání prvků reprezentovaných kořeny stromů seřadíme stromy do množin  $Q_i, i=0, \dots, n$ , kde  $Q_i$  je množina všech stromů v  $T$  izomorfních s  $H_i$ . $i = 0, T = 0$ **while**  $\exists Q_i \neq 0$  **do**  **while**  $|Q_i| > 1$  **do**    vezmeme dva stromy z  $Q_i$ , spojíme je, výsledek dáme do  $Q_{i+1}$   **end while**  **if**  $Q_i \neq 0$  **then**    strom z  $Q_i$  dáme do  $T$   **end if**   $i = i + 1$ **end while**

---

DELETETEMIN dá stromy po odtržení nejmenšího prvku do odpovídajících množin  $Q_i$

**Věta 10.3.3.** Operace MERGE a INSERT při líné implementaci vyžadují čas  $O(1)$ , operace DELETETEMIN a MIN vyžadují čas  $O(\text{počet stromů})$ .

**Poznámka 10.3.6.** bal(konfigurace) = počet všech stromů ve všech haldách v konfiguraci

**Amort. slož.**

amort. slož. = čas pro operaci + bal(výsledné konfigurace) - bal(původní konfigurace)

MERGE  $O(1)$ INSERT  $O(1)$ MIN  $O(\log n)$ DELETETEMIN  $O(\log n)$

### 10.3.3 Zobecněné binomiální haldy

XXX

přednášelo se to  
tento rok vůbec  
?

## 10.4 Fibonacciho haldy

XXX dopsat !

# Kapitola 11

## Dynamizace

V uspořádaném poli umíme rychle vyhledávat, ale přidat prvky znamená celé ho přebudovat. Ve strůstaajícím hašování zase nešly prvky mazat, ve velmi komprimovaných trie ani přidávat, ani mazat. V této kapitole ukážeme obecnou metodu, jak tyto problémy řešit, podobnou přístupu u binomiálních hald.

### 11.1 Zobecněný vyhledávací problém

**Definice 11.1.1.** *Vyhledávací problém* je funkce  $f : U_1 \times 2^{U_2} \rightarrow U_3$ , kde  $U_1, U_2$  a  $U_3$  jsou univerza.

**Definice 11.1.2.** *Řešení vyhledávacího problému* pro  $x \in U_1, A \subseteq U_2$  je nalezení hodnoty  $f(x, A)$ .

**Poznámka 11.1.1.** Chceme najít strukturu, která reprezentuje  $A$  a algoritmus, který pro vstup  $x \in U_1$  spočítá  $f(x, A)$ . Takové struktury se říká statická struktura pro vyhledávací problém.

Př.:

**Klasický vyhledávací problém:**  $U_1 = U_2 = U$ , univerzum prvků;  $U_3 = \{0, 1\}, A \subseteq U_2$

$$f(x, A) = \begin{cases} 0 & \text{když } x \notin A \\ 1 & \text{když } x \in A \end{cases} \text{ (rozložitelný)}$$

**Euklidovská vzdálenost bodů v rovině:**  $U_1 = U_2 =$  euklidovská rovina;  $U_3 = \mathbb{R}^+$ ;  $f(x, A) = \text{dist}(x, A)$  vzdálenost bodu  $x \in U_1$  od množiny  $A$ . (rozložitelný,  $\oplus \dots$  operace min)

Nalezení předchůdce  $U_1 = U_2 = U_3$  pro  $x \in U_1$  a  $A \subseteq U_1$  a je  $f(x, U_1)$  je největší prvek  $A \leq x$  (rozložitelný, je potřeba disjunkce)

**Příslušnost ke konvexnímu obalu**  $U_1 = U_2 =$  rovina;  $U_3 = \{0, 1\}$ ;

$$f(x, A) = \begin{cases} 0 & \text{když } x \text{ nepatří do konvexního obalu } A \\ 1 & \text{když } x \text{ patří do konvexního obalu } A \end{cases} \text{ (není rozložitelný problém)}$$

chybí obrázek

#### 11.1.1 Operace INSERT a DELETE

Pro množinu  $A \subseteq U_2$  a pro statickou strukturu  $S$  řešící vyhledávací problém pro  $x \in U_1$ . INSERT( $x, A$ ) - vybudování struktury řešící vyhledávací problém pro množinu  $A \cup \{x\}$ . DELETE( $x, A$ ) - vytvoření struktury řešící vyhl. problém pro množinu  $A - \{x\}$ .

Pozn: ze statické struktury chce vytvořit dynamickou (dynamizace). INSERT je obvykle jednodušší než DELETE, na tu budeme potřebovat dodatečné předpoklady.

Nároky na dynamizaci - chceme aby se  $f(x, A)$  v nové struktury spočítalo přibližně stejně rychle jako v původní struktury - když vytvoření původní struktury pro  $n$  prvňkovou množinu trvalo  $t$ , pak operace INSERT by přibližně měla vyžadovat čas  $t/n$ .

**Definice 11.1.3.** Vyhledávací problém je *rozložitelný*, když existuje operace  $\oplus$  spočitatelná v konstantním čase a platí: když  $x \in U_1$  a  $A$  a  $B$  jsou disjunktí podmnožiny  $U_2$ , pak

$$f(x, A \cup B) = f(x, A) \oplus f(x, B).$$

**Poznámka 11.1.2.** Z výše uvedených příkladů není rozložitelným problémem příslušnost ke konvexnímu obalu, ostatní vyhledávací problémy jsou rozložitelné.

**Definice 11.1.4.** Nechť  $f$  je rozložitelný vyhledávací problém a  $\mathcal{S}$  je “statická” datová struktura, která ho řeší. Neboli  $\mathcal{S}$  je tvořena pro pevnou množinu  $A \subseteq U_2$  a obsahuje operaci, která pro vstup  $x$  počítá  $f(x, A)$ .

Popíšeme důležité parametry  $\mathcal{S}$ : nechť  $n = |A|$ , označme

$$\begin{aligned} Q_{\mathcal{S}}(n) &= \text{čas potřebný pro výpočet } f(x, A) \\ S_{\mathcal{S}}(n) &= \text{paměť potřebná pro vybudování } \mathcal{S} \\ P_{\mathcal{S}}(n) &= \text{čas potřebný pro vybudování } \mathcal{S} \end{aligned}$$

## 11.2 Semi-dynamizace

**Věta 11.2.1.** Máme rozložitelný vyhled. problém  $f$  a máme pro něj statickou strukturu, která ho řeší v čase  $Q(n)$ , vyžaduje  $S(n)$  paměti a vytvoří se v čase  $P(n)$ , kde  $Q(n)$ ,  $\frac{P(n)}{n}$ ,  $\frac{S(n)}{n}$  jsou neklesající funkce. Pak existuje semidynamická dat. struktura  $D$ , řešící  $f$  v čase  $O(Q(n) \log n)$  vyžadující  $O(S(n))$  paměti a umožňující INSERT s amort. složitostí  $O(\frac{P(n)}{n} \cdot \log n)$ .

*Důkaz.* Budeme předpokládat, že  $Q_{\mathcal{S}}(n)$ ,  $S_{\mathcal{S}}(n)/n$  a  $P_{\mathcal{S}}(n)/n$  jsou neklesající funkce.

Máme množinu  $A$  a vytvoříme pro ni novou strukturu  $D$ . Nechť  $A_i \subseteq A$  taková, že buď  $|A_i| = 2^i$  nebo  $A_i = \emptyset$

$$A_i \cap A_j = \emptyset \text{ pro } i \neq j. \bigcup A_i = A$$

Platí  $A_i \neq \emptyset$  právě když  $(i+1)$ -ní bit v dvojkovém rozvoji čísla  $|A|$  je 1.

Chceme navrhnout strukturu, která by uměla

- 1) Pro  $x \in U_1$  a pevné  $A \subseteq U_2$  rychle spočítat  $f(x, A)$ .
- 2) Pro  $A$  a  $y \in U_2$  rychle vytvořit strukturu pro  $A \cup \{y\}$ .

Mějme  $A_0, A_1, \dots$  takové, že

- 1)  $A_i \cap A_j = \emptyset$  pro  $i \neq j$
- 2) buď  $A_i = \emptyset$  nebo  $|A_i| = 2^i$
- 3)  $\bigcup_i A_i = A$

Nová struktura  $D$  reprezentující  $A$  je potom

- nějaká dynamická struktura reprezentující  $A$  (např. (a,b)-strom, červeno-černý strom, AVL-strom)
- Pro každé  $A_i \neq \emptyset$  máme  $S$  strukturu reprezentující  $A_i$ .
- Pro každé  $A_i \neq \emptyset$  seznam prvků v  $A_i$ ; prvky těchto seznamů jsou propojeny s odpovídajícími prvky ve stromě.

Jak v nové struktuře počítáme  $f(x, A)$ ? pro každou  $A_i \neq \emptyset$  spočítáme  $f(x, A_i)$  a pomocí operace  $\oplus$  pak spočítáme  $f(x, A)$ .

**Poznámka 11.2.1.** platí, když  $A_i \neq \emptyset$ , pak  $i \leq \lceil \log_2 |A| \rceil$  čas, který je potřeba v nové struktuře na výpočet  $f(x, A)$



$$\log_2 |A| + \sum_{i=0}^{\log |A|} Q(2^i) \leq \log |A| + \sum_{i=0}^{\log |A|} Q(|A|) = \log |A| (Q(|A|) + 1) \quad (11.1)$$

$\log_2(|A|)$  - vyhodnocení  $f(x, A)$  z  $f(x, A_i)$ ,  $i = 0, 1, \dots$

tedy algoritmus potřebuje  $O(\log |A| \cdot Q(|A|))$  času když  $Q(n) = \Theta(n^\epsilon)$  pro  $\epsilon > 0$ , pak platí že nová struktura pro výpočet  $f$  potřebuje

$$\begin{aligned} \log |A| + \sum_{i=0}^{\log n} Q(2^i) &= \\ |A| + \sum_{i=0}^{\log |A|} \frac{S(2^i)}{2^i} 2^i &\leq |A| + \sum_{i=0}^{\log |A|} \frac{S(|A|)}{|A|} 2^i \\ &= |A| - \frac{S(|A|)}{|A|} 2^i = |A| - \frac{S(|A|)}{|A|} \left( \sum_{i=0}^{\log |A|} 2^i \right) \\ &= O(S(|A|)) \quad (11.2) \end{aligned}$$

INSERT(x)

**if**  $x \notin A$  **then**

nalezneme nejmenší  $j$ , že  $A_j = \emptyset$

**end if**

$A_j = \{x\} \cup \bigcup_{i < j} A_i$ ,  $A_i = \emptyset$  pro  $i < j$

vytvoříme strukturu  $S$  spojový seznam pro  $A_j$

$x$  přidáme do reprezentace  $A$ .

Kdy se buduje znovu  $S$  struktura pro  $A_j$  ?

- 1) musí se naplnit všechny  $A_i$  pro  $i < j$  to je  $2^j - 1$  úspěšných INSERTů (ty, které přidaly prvek)
- 2) provede se úspěšný INSERT, který vyprázdní  $A_i$  pro  $i \leq j$
- 3) znovu se musí naplnit  $A_i$  tj.  $2^j - 1$  úspěšných INSERTů
- 4) další úspěšný INSERT vytvoří teprve  $S$  strukturu pro  $A_j$

tj.  $2 \cdot 2^j$  úspěšných INSERTů

Amortizovaný čas operace INSERT je

$$\log |A| + \sum_{i=0}^{\log |A|} \frac{P(2^i)}{2^{i+1}} \leq \log |A| + \sum_{i=0}^{\log |A|} \frac{P|A|}{|A|} = O(\log |A| \cdot \frac{P|A|}{|A|})$$

□

**Věta 11.2.2.** Máme rozložitelný vyhled. problém  $f$  a máme pro něj statickou strukturu, která ho řeší v čase  $Q(n)$ , vyžaduje  $S(n)$  paměti a vytvoří se v čase  $P(n)$ , kde  $Q(n)$ ,  $\frac{P(n)}{n}$ ,  $\frac{S(n)}{n}$  jsou neklesající funkce. Pak existuje semidynamická dat. struktura  $D$ , řešící  $f$  v čase  $O(Q(n) \log n)$  vyžadující  $O(S(n))$  paměti a umožňující INSERT s složitostí  $O(\frac{P(n)}{n} \cdot \log n)$ .

Následuje konstrukce takové semidyn. struktury, která bude podporovat INSERT se složitostí v nejhorsím případě.

**Poznámka 11.2.2.** pokud  $\frac{P(n)}{n} = \Theta(n^\epsilon)$  pro  $\epsilon > 0$ , pak amortizovaný čas pro operaci INSERT bude  $O(\frac{P|A|}{|A|})$ .

Máme množinu A

budeme mít rozklad A na disjunktní množiny  $A_{i,j}$ ,  $i = 0, 1, \dots, j \in 0, 1, \dots, k_j$ , kde  $k_j \in 0, 1, 2$ .  
 $|A_{i,j}| = 2^i$  a platí:

když  $A_{i,0}$  existuje pro  $i > 0$ , pak existují  $A_{i-1,0}, A_{i-1,1}$ .

Struktura:

- 1) reprezentace A (pomocí (a,b)-stromů, červeno-černých stromů, ...)
- 2)  $\forall$  existující  $A_{i,j}$  je S struktura reprezentující  $A_{i,j}$
- 3)  $\forall$  existující  $A_{i,j}$  je spojový seznam reprezentující  $A_{i,j}$
- 4) když  $A_{i,0}$  a  $A_{i,1}$  existují pro nějaké i, pak je "rozpracovaná" S struktura pro množinu  $A_{i-1,k_{i+1}} = A_{i,0} \cup A_{i,1}$ . tj. bylo provedeno několik kroků pro její vytvoření, ale není dokončena.

$$A \subseteq U_2, i_0 \in N$$

$\forall i = 0, 1, \dots, i_0$  je dáno  $j_i \in 0, 1, 2$  takové, že  $j_i > 0$  když  $i < i_0$ .

$\forall i = 0, 1, \dots, i_0$  a  $\forall j = 0, 1, \dots, j_i$  je  $A_{i,j} \in A$  taková, že  $|A_{i,j}| < 2^i$ .

**Definice 11.2.1.**  $A_{i,j}$ ,  $i = 0, 1, \dots, i_0$ ,  $j = 1, 2, \dots, j_i$  je rozklad A.

Pro každé  $A_{i,j}$  je dána S struktura reprezentující  $A_{i,j}$  a spojový seznam prvků z  $A_{i,j}$ , navíc dána dat. struktura reprezentující A. Když  $A_{i,1}$  existuje, pak je rozpracovaná S struktura pro  $A_{i+1,j_{i+1}+1} = A_{i,0} \cup A_{i,1}$ .

pozn. struktura je rozpracovaná, jestliže bylo provedeno několik kroků pro postavění S struktury, ale ještě není dokončena.

- toto je definice nové semidynamické struktury

Paměťové nároky

$$|A| + \sum_{i=0}^{\log|A|} 4S(2^i) = |A| + \sum_{i=0}^{\log|A|} \frac{S(2^i)}{2^i} 2^i \leq |A| + 4 \sum_{i=0}^{\log|A|} \frac{S(|A|)}{|A|} 2^i =$$

$$|A| + \frac{4S(|A|)}{|A|} (\sum 2^i) = |A| + 4S(|A|) = O(S(|A|)) \quad (11.3)$$

**Poznámka 11.2.3.**  $|A|$  - pamet pro pom. struktury

$\sum_{i=0}^{\log|A|} 4S(2^i)$  - pamet potrebna na S struktury

Algoritmus pro výpočet

spočítáme  $f(x, A_{i,j})$  pro každou  $A_{i,j}$  a pomocí operace  $\oplus$  spočítáme  $f(x, A)$ .

čas potřebný pro výpočet A

$$\sum_{i=0}^{\log|A|} 3Q(2^i) + 3 \log |A| \leq 3 \sum_{i=0}^{\log|A|} Q(|A|) + 3 \log |A| = 3Q(|A|) \log |A| = O(Q(|A|) \log |A|) \quad (11.4)$$

Platí:  $Q(n) \geq n^\epsilon$  pro nějaké  $\epsilon$ , pak čas potřebný pro výpočet f je  $O(Q(N))$ .

INSERT(x) viz. alg. 11.1

Čas pro INSERT(x) je

$$\log|A| + \sum_{i=0}^{\log|A|} \left( \frac{P(2^i)}{2^i} + 1 \right) = 2 \log|A| + \sum_{i=0}^{\log|A|} \frac{2P(|A|)}{|A|} =$$

$$2 \log|A| + \frac{2P(|A|)}{|A|} \sum_{i=0}^{\log|A|} 1 = 2 \log|A| + \frac{2P(|A|)}{|A|} \log|A| = O\left(\frac{2P(|A|)}{|A|} \log|A|\right) \quad (11.5)$$

---

**Algoritmus 11.1** Operace INSERT

---

když  $x \notin A$  jinak provedeme:  
 vytvoříme S strukturu pro  $A_{0,j_i+1} = \{x\}$  (zvětšíme  $j_0$  o 1)  
**if**  $j_0$  je liché **then**  
     pak provedeme 1.krok pro vytvoření S struktury  
     pro množinu  $A_{1,j_1+\lceil \frac{j_0}{2} \rceil} = A_{0,j_0-1} \cup A_{0,j_0}$   
**end if**  
 pro  $i = 1, 2, \dots, i_0 + 1$  v roustoucím pořadí:  
**if** S je rozpracovaná struktura pro  $A_{i,j_i+1}$  **then**  
     provedeme dalších  $\frac{P(2^i)}{2^i}$  kroků pro její vybudování  
**end if**  
 když jsme dobudovali S strukturu pro  $A_{i,j_i+1}$ , zvětšíme  $j_i$  o 1  
 zrušíme  $A_{i-1,0} \cup A_{i-1,1}$ , zmenšíme  $j_i$  o 2 a  $A_{i-1,j}$  přepíšeme na  $A_{i-1,j_i+2}$ .  
**if**  $j_i$  liché **then**  
     provedeme 1.krok pro vytvoření Q struktury pro  
     množinu  $A_{1,j_i+1+\lceil \frac{j_i}{2} \rceil} = A_{i,j_i} \cup A_{i,j_i-1}$ .  
**end if**

---

$\log|A|$  - čas pro zjištění zda  $x \in A$

když  $\frac{P(n)}{n} \geq n^\epsilon$  pro  $\epsilon > 0$ , pak INSERT vyžaduje čas  $O(\frac{P(n)}{n})$ .

Př.:

INSERT( $x_1$ ) $A_{0,0} = \{x_1\}$	INSERT( $x_2$ ) $A_{0,0} = \{x_1\}$ $A_{0,1} = \{x_2\}$ 1. krok pro $A_{1,0} = \{x_1, x_2\}$	INSERT( $x_3$ ) $A_{0,0} = \{x_1\}$ $A_{0,1} = \{x_2\}$ $A_{0,2} = \{x_3\}$ $\frac{P(2)}{2}$ kroků pro $A_{1,0} = \{x_1, x_2\}$
INSERT( $x_4$ ) $A_{0,0} = \{x_1\}$ $A_{0,1} = \{x_2\}$ $A_{0,2} = \{x_3\} \rightarrow A_{0,0} = \{x_3\}$ $A_{0,3} = \{x_4\} \rightarrow A_{0,1} = \{x_4\}$ dokončíme $A_{1,0} = \{x_1, x_2\}$ 1. krok pro $A_{1,1} = \{x_3, x_4\}$	INSERT( $x_5$ ) $A_{0,0} = \{x_3\}$ $A_{0,1} = \{x_4\}$ $A_{0,2} = \{x_5\}$ $A_{1,0} = \{x_1, x_2\}$ $\frac{P(2)}{2}$ kroků pro $A_{1,1} = \{x_3, x_4\}$	INSERT( $x_6$ ) $A_{0,0} = \{x_3\}$ $A_{0,1} = \{x_4\}$ $A_{0,2} = \{x_5\} \rightarrow A_{0,0} = \{x_5\}$ $A_{0,3} = \{x_6\} \rightarrow A_{0,1} = \{x_6\}$ $A_{1,0} = \{x_1, x_2\}$ dokončeno $A_{1,1} = \{x_3, x_4\}$ 1. krok pro $A_{1,2} = \{x_5, x_6\}$ 1. krok pro $A_{2,0} = \{x_1, x_2, x_3, x_4\}$ $\frac{P(4)}{4}$ kroků

**Věta 11.2.3.** *Nechť S je statická struktura pro rozložitelný vyhledávací problém f a necht K je "hladká" funkce. Pak existuje semidynamická struktura D založená na rozkladu určeném funkcí K, tak že platí: když  $K=O(\log n)$ , pak čas pro vyhledání je  $O(K Q(n))$  pro INSERT je  $O(K(n)n^{\frac{1}{K(n)}} \frac{P(n)}{n})$  když  $K = \Omega(\log(n))$  pak platí: čas pro vyhledání je  $O(K(n)Q(n))$  pro INSERT je  $O(\frac{\log(n)}{\log \frac{K(n)}{\log(n)}} \frac{P(n)}{n})$ .*

Důkaz. Důkaz viz [2] □

### 11.3 Dynamizace

Potřebujeme, aby struktura S připouštěla falešný DELETE (prvek pouze škrtneme, ale zůstane tam. falešný - čas. ani paměťové nároky se nezlepší ani nezhorší)

**Definice 11.3.1.** falešný DELETE je operace, která vyškrtně prvek z množiny - tj. umožní počítat  $f(x, A - \{a\})$  (kde a je vyškrtnutý prvek) tak, že časové nároky budou stejné jako když nebyl žádný prvek vyškrtnut.

Budeme předpokládat, že čas pro falešný DELETE je  $O(n)$ , kde  $n$  je velikost původní reprezentované množiny.

### 11.3.1 Reprezentace množiny A

rozložíme A na disjunktní množiny  $A_j, j = 0, 1, \dots, \log|A| + 3$  takové, že buď  $A_j = \emptyset$  nebo  $2^{j-3} < |A_j| \leq 2^j$ .

každá množina  $A_j$  bude reprezentována strukturou, která původně (když nebyly vyškrtnuté žádné prvky) měla velikost  $\leq 2^j$ .

Dále  $\forall A_j \neq \emptyset$  bude dán spojový seznam prvků v  $A_j$ .

Bude dána datová reprezentace množiny A. Pro každý prvek a v spojovém seznamu množiny  $A_j$  bude dán ukazatel na prvek a v dat. struktuře reprezentující A a naopak. Pro každý prvek v dat. struktuře repr. A je dán ukazatel na prvek a v odpovídajícím spojovém seznamu.

### 11.3.2 Paměťové nároky

$$|A| + \sum_{i=0}^{\log|A|+3} S(2^i) = |A| + \sum_{i=0}^{\log|A|+3} \frac{S(2^i)}{2^i} 2^i \leq |A| + \sum_{i=0}^{\log|A|+3} \frac{S(8|A|)}{8|A|} 2^i =$$

$$|A| + \frac{S(8|A|)}{8|A|} 2^i = |A| + \frac{S(8|A|)}{8|A|} \sum 2^i = |A| + S(8|A|) = O(S(8|A|)) \quad (11.6)$$

$|A|$  - pomocne struktury

suma - pamet pro S struktury

závěr: když S je omezená polynomem, pak paměťové nároky jsou  $O(S(n))$ . pokud S je superpolynomiální, pak paměť. nároky jsou  $O(S(8n))$  (a platí  $S(n) = o(S(8n))$ )

výpočet f:

spočítáme  $f(x, A_j)$  a pomocí operace  $\oplus$  spočítáme  $f(x, A)$ .

### 11.3.3 čas pro výpočet f

$$\log(n) + \sum_{i=0}^{\log|A|+3} Q(2^i) = \log(n) + \sum Q(8|A|) = O(Q(8|A|)\log|A|).$$

závěr: čas na výpočet f je  $\begin{cases} \text{když Q je subpolynomiální} & O(Q(n)\log(n)) \\ \text{polynomiální} & O(Q(n)) \\ \text{superplynomiální} & O(Q(8n)) \end{cases}$

INSERT(x) viz. alg. 11.2

---

#### Algoritmus 11.2 Operace INSERT (f)

---

**if**  $x \notin A$  **then**

nalezneme nejmenší j takové, že  $|\bigcup_{i \leq j} A_i| < 2^j$

položíme  $A_j = \bigcup_{i \leq j} A_i \cup \{x\}$

$A_i = \emptyset$  pro  $i < j$

vytvoříme S-strukturu a spojový seznam pro  $A_j$  (x přidáme do struktury reprezentující A a přidáme požadované ukazatele)

**end if**

---

Pozorování:

Když vytváříme při INSERTu S-strukturu pro  $A_j$ , pak  $2^{j-1} < |A_j| \leq 2^j$ .

(když toto neplatí, pak pro j-1 je splněna nerovnost  $|\bigcup_{i < j-1} A_i| < 2^{j-1}$  a to je spor s minimalitou j.

DELETE(x) viz. alg. 11.3

Pozorování:

Když operace DELETE buduje S-strukturu pro množinu  $A_j$ , pak platí:  $2^{j-1} \leq |A_j| \leq 2^{j-1}$ .

**Algoritmus 11.3** Operace INSERT (f)

---

```

if  $x \notin A$  then
  odstraníme x ze struktury pro A
  nalezneme j takové, že  $x \in A_j$  (budeme znát přímo místo x v seznamu pro  $A_j$ )
  if  $|A_j| = 1$  then
    smažeme  $A_j$  (odpovídající S-strukturu a spojový seznam)  $\rightarrow A_j = \emptyset$ 
  end if
  if  $|A_j| > 1$  a zároveň  $|A_j| > 2^{j-3} + 1$  then
    na S strukturu pro  $A_j$  provedeme falešný DELETE, x smažeme ze spojového seznamu
    pro  $A_j \rightarrow A_j = A_j - \{x\}$ 
  end if
  if  $|A_j| > 1$  a zároveň  $|A_j| = 2^{j-3} + 1$  then
    if  $A_{j-1} = \emptyset$  then
       $A_{j-1} = A_{j-1} - \{x\}, A_j = \emptyset$ 
      vybudujeme novou S-strukturu pro  $A_{j-1}$  (x odstraníme ze spojového seznamu pro  $A_{j-1} - 1$ )
    end if
    if  $A_{j-1} = \emptyset$  a zároveň  $|A_{j-1}| > 2^{j-2}$  then
      vyměním  $A_j$  a  $A_{j-1}$ 
      z  $A_{j-1}$  odstraníme x a vytvoříme novou S-strukturu pro  $A_{j-1}$  (původní struktura mohla mít až  $2^j$  prvků)
    end if
    if  $A_{j-1} = \emptyset$  a zároveň  $|A_{j-1}| \leq 2^{j-2}$  then
       $B = (A_j \cup A_{j-1}) - \{x\}$ 
      zrušíme S-struktury pro  $A_j, A_{j-1}$  a vybudujeme S-strukturu a spojový seznam pro B
      if  $|B| \geq 2^{j-2}$  then
         $A_j = B, A_{j-1} = \emptyset$ 
      else
         $A_{j-1} = B, A_j = \emptyset$ 
      end if
    end if
  end if
end if
end if

```

---

**11.3.4 Amortizovaný čas operace DELETE**

$$(\log|A| + D(2^j) + \frac{P(2^j)}{2^{j-3}}) = O(\log|A| + D(|A|) + 8 \frac{P(|A|)}{|A|})$$

$\log|A|$  - zjištění zda  $x \in A$

$D(2^j)$  - falešný DELETE

$\frac{P(2^j)}{2^{j-3}}$  - budování S-struktury - aspoň  $2^{j-3}$  ... před tím

zbytek poslední přednášky chybí

## Kapitola 12

# Vícedimenzionální vyhledávání

XXX vykládalo se to vůbec ?

# Literatura

- [1] Mehlhorn, Kurt. *Data Structures And Algorithms*, Springer Verlag, '83
- [2] K. Mehlhorn, M. H. Overmars *Optimal Dynamization of Decomposable Searching Problems*