# Datové struktury

přednáší Václav Koubek zTEXal Martin Vidner [1] [2]

26. března 2003

[1] martin@artax.karlin.mff.cuni.cz, mvidner@atlas.cz

[2] Přispěli: Lišák, Jéňa, Žabička, Jindřich, MJ, Pavel ... *musím napsat pořádné titulky*

# Obsah


**1 Úvod** **4**
- 1.1 Předpoklady . . . 4
- 1.2 Jaké typy složitosti nás zajímají . . . 4
  - 1.2.1 Paměťová složitost reprezentované struktury . . . 4
  - 1.2.2 Časová složitost algoritmů pracujících na datové struktuře . . . 4

**2 Slovníkový problém** **6**
- 2.1 Pole . . . 6
- 2.2 Seznam . . . 6

**3 Hašování I** **7**
- 3.1 Hašování se separovanými řetězci . . . 7
  - 3.1.1 Očekávaná délka seznamu . . . 8
  - 3.1.2 Očekávaný čas posloupnosti operací . . . 8
  - 3.1.3 Očekávaný počet testů . . . 8
  - 3.1.4 Očekávaná délka nejdelšího seznamu . . . 9
- 3.2 Hašování s uspořádanými řetězci . . . 10
  - 3.2.1 Očekávaný čas . . . 10
- 3.3 Hašování s přesuny . . . 10
- 3.4 Hašování se dvěma ukazateli . . . 11
- 3.5 Hašování s lineárním přidáváním . . . 11
- 3.6 Hašování se dvěma funkcemi (otevřené h., h. s otevřenou adresací) . . . 12
  - 3.6.1 Algoritmus INSERT . . . 13
  - 3.6.2 Očekávaný počet testů . . . 13
- 3.7 Srůstající hašování (standardní: LISCH a EISCH) . . . 14
- 3.8 Srůstající hašování s pomocnou pamětí (obecné: LICH, EICH, VICH) . . . 15
  - 3.8.1 Srovnávací graf . . . 15
- 3.9 Srovnání metod . . . 15
- 3.10 Přehašovávání . . . 16

**4 Hašování II** **18**
- 4.1 Univerzální hašování . . . 18
  - 4.1.1 Očekávaná délka řetězce . . . 19
  - 4.1.2 Velikost $c$-univerzálního systému . . . 20
  - 4.1.3 Reprezentace a (MEMBER), INSERT, DELETE . . . 21
- 4.2 Perfektní hašování . . . 22
  - 4.2.1 Perfektní hašovací funkce do tabulky velikosti $n^2$ . . . 23
  - 4.2.2 Perfektní hašovací funkce do tabulky velikosti $3n$ . . . 24
  - 4.2.3 GPERF . . . 25

**5 Trie** **26**
5.1 Základní varianta . . . 26
5.1.1 Algoritmus MEMBER . . . 26
5.1.2 Algoritmus INSERT . . . 26
5.1.3 Algoritmus DELETE . . . 27
5.1.4 Časová a paměťová složitost . . . 27
5.2 Komprimované trie . . . 27
5.2.1 MEMBER . . . 27
5.2.2 INSERT . . . 27
5.2.3 DELETE . . . 28
5.2.4 Časová a paměťová složitost . . . 28
5.3 Ještě komprimovanější trie . . . 32
5.3.1 Popis $A$ a $rd$ . . . 32
5.3.2 Jak nalézt $rd$ z $A$ . . . 32
5.3.3 Vertikální posun sloupců . . . 32
5.3.4 Úsporné uložení řídkého vektoru . . . 32

**6 Uspořádaná pole** **33**
6.1 Unární, binární a interpolační vyhledávání . . . 33
6.2 Zobecněné kvadratické vyhledávání . . . 33

**7 Binární stromy** **36**
7.1 Obecně . . . 36
7.1.1 Algoritmus MEMBER . . . 36
7.1.2 Algoritmus INSERT . . . 36
7.1.3 Algoritmus DELETE . . . 36
7.2 Optimální binární vyhledávací stromy . . . 36
7.2.1 Algoritmus konstrukce . . . 36
7.2.2 Snížení složitosti z kubické na kvadratickou . . . 36
7.3 Skorooptimální binární vyhledávací stromy . . . 36
7.4 Červenočerné stromy . . . 36
7.4.1 Operace INSERT . . . 37
7.4.2 Operace DELETE . . . 38
7.4.3 Závěry . . . 40

**8 $(a, b)$ stromy** **41**
8.1 Základní varianta . . . 41
8.1.1 Reprezentace množiny $S$ $(a, b)$ stromem . . . 41
8.1.2 MEMBER($x$) v $(a, b)$ stromu . . . 42
8.1.3 INSERT($x$) do $(a, b)$ stromu . . . 42
8.1.4 DELETE($x$) z $(a, b)$ stromu . . . 42
8.1.5 Shrnutí . . . 43
8.1.6 Jak volit parametry $(a, b)$ . . . 43
8.2 Další operace . . . 43
8.2.1 Algoritmus JOIN($T_1, T_2$) pro $(a, b)$ stromy . . . 43
8.2.2 Algoritmus SPLIT($x, T$) pro $(a, b)$ strom . . . 44
8.2.3 Algoritmus STACKJOIN($Z$) pro zásobník $(a, b)$ stromů . . . 44
8.2.4 Algoritmus FIND($T, k$) pro $(a, b)$ strom . . . 45
8.2.5 A-sort . . . 45
8.3 Paralelní přístup do $(a, b)$ stromů . . . 47
8.3.1 Paralelní INSERT($x$) do $(a, b)$ stromu . . . 47
8.3.2 Paralelní DELETE($x$) z $(a, b)$ stromu . . . 48
8.4 Složitost posloupnosti operací na $(a, b)$ stromu . . . 49
8.4.1 přidání/ubrání listu . . . 50

8.4.2 štěpení . . . 50
8.4.3 spojení . . . 50
8.4.4 přesun . . . 50
8.5 Propojené $(a, b)$ stromy s prstem . . . 52
8.5.1 Algoritmus MEMBER . . . 52

**9 Samoopravující se struktury** **53**
9.1 Amortizovaná složitost . . . 53
9.2 Seznamy . . . 53
9.2.1 Algoritmus MEMBER . . . 53
9.2.2 Algoritmus INSERT . . . 53
9.2.3 Algoritmus DELETE . . . 53
9.2.4 Algoritmus MFR (Move Front Rule) . . . 53
9.2.5 Algoritmus TR (Transposition Rule) . . . 55
9.3 Splay stromy . . . 56
9.3.1 Operace SPLAY . . . 56
9.3.2 Podporované operace . . . 56
9.3.3 Algoritmus MEMBER . . . 56
9.3.4 Algoritmus JOIN2 . . . 56
9.3.5 Algoritmus JOIN3 . . . 56
9.3.6 Algoritmus SPLIT . . . 56
9.3.7 Algoritmus DELETE . . . 57
9.3.8 Algoritmus INSERT . . . 57
9.3.9 Algoritmus CHANGEWEIGHT . . . 58
9.3.10 Algoritmus SPLAY . . . 58
9.3.11 Amortizovaná složitost SPLAY . . . 58
9.3.12 Amortizovaná složitost ostatních operací . . . 59

**10 Haldy** **60**
10.1 $d$-regulární haldy . . . 60
10.1.1 Algoritmus UP . . . 60
10.1.2 Algoritmus DOWN . . . 60
10.1.3 Operace . . . 60
10.1.4 Algoritmus MAKEHEAP . . . 60
10.1.5 Dijkstrův algoritmus . . . 60
10.2 Leftist haldy . . . 60
10.3 Binomiální haldy . . . 60
10.3.1 Zobecněné binomiální haldy . . . 60
10.4 Fibonacciho haldy . . . 60

**11 Dynamizace** **61**
11.1 Zobecněný vyhledávací problém . . . 61

**12 Vícedimenzionální vyhledávání** **63**

# 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 alespon $n_k$ 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 urči 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 urči 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:

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

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

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

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

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

# Kapitola 2

# Slovníkový problém

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

Budeme používat operace

- MEMBER$(x), x \in U$, odpověď je *ano,* když $x \in S$, *ne,* když $x \notin S$.
- INSERT$(x), x \in U$, vytvoří reprezentaci množiny $S \cup \{x\}$
- DELETE$(x), x \in U$, vytvoří reprezentaci množiny $S \setminus \{x\}$
- ACCESS$(x)$. Ve skutečných databázích 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 přemísťovaní jednotlivých prvků datové struktury.

Předpokládá se znalost těchto základních datových struktur: pole, spojový seznam, obousměný 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 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 pamětí 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 $h : U \to \{0..m-1\}$. 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 spočitatelné 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

$$\begin{aligned}
E &= \sum_{\ell=0}^{n} \ell \cdot p(\ell) \\
&= \sum_{\ell=0}^{n} \ell \frac{n!}{\ell!(n-\ell)!} (\frac{1}{m})^{\ell} (1 - \frac{1}{m})^{n-\ell} = \sum_{\ell=0}^{n} n \frac{(n-1)!}{(\ell-1)![(n-1)-(\ell-1)]!} (\frac{1}{m})^{\ell} (1 - \frac{1}{m})^{n-\ell} \\
&= \sum_{\ell=0}^{n} \frac{n}{m} \binom{n-1}{\ell-1} (\frac{1}{m})^{\ell-1} (1 - \frac{1}{m})^{(n-1)-(\ell-1)} = \frac{n}{m} \sum_{k=-1}^{n-1} \binom{n-1}{k} (\frac{1}{m})^{k} (1 - \frac{1}{m})^{(n-1)-k}
\end{aligned}$$

README.1st: všechny úpravy směřují k tomuto součtu podle binomické věty

$$= \frac{n}{m} (\frac{1}{m} + (1 - \frac{1}{m}))^{n-1} = \frac{n}{m} = \alpha, \quad (3.1)$$

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

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

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 $= O((1 - \frac{1}{m})^n + \frac{n}{m})$

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

#### Neúspěšný případ

Projdeme celý seznam, musíme nahlédnout i do prázdného seznamu.

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

**Úspěšný případ**

Zde Koubková 1998. Koubek 2000 to má trochu jinak

Počet testů pro vyhledání všech prvků v seznamu délky $\ell$ je
$1+2+\cdots+\ell=\binom{\ell+1}{2}$.

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:

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

Očekávaný počet testů pro vyhledání jednoho prvku

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

### 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 $|S|=n\le m$ je očekávaná délka maximálního seznamu $\text{EMS}=O(\frac{\log n}{\log\log n})$.

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

Použijeme trik: nechť

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

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

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

(teleskopická suma) vysvětlit

Spočteme $q(j)$:

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

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

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

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

pak

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

## 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 $U = \mathbb{N}, m = 10, h(x) = x$ mod 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, ze $h_2(i)$ 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

Ještě rozmyslet značení a sjednotit zápis algoritmů

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 obsazené prvkem $\neq x$ a $i \neq k$ do
   $k = (k + h_2(x)) \bmod m$
   enddo
4. když je $k$-té místo obsazené prvkem $x$, pak nedělej nic,
   když $i = k$, pak ohlaš přeplněno, jinak vlož $x$ na $k$-té místo

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 $h_1(x), h_1(x) + h_2(x), h_1(x) + 2h_2(x), \ldots$ 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.

Buď $q_j(n, m)$ 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).

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

$$= (\sum_{j=0}^{n} q_j(n, m)) - (n + 1)q_{n+1}(n, m) = \sum_{j=0}^{n} q_j(n, m) \quad (3.4)$$

Protože

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

dostaneme po dosazení:

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

Řešením tohoto rekurentního vzorce je

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

což dokážeme indukcí:

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

**při úspěšném vyhledávání**

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

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

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 paměť („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 sklepě, 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 | 121197 | 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 | 1186 | 90 | 86 |
| 10 | 71 | | 71 | | 71 | |
| 11 | 70 | 9 | 70 | 12 | 70 | 97 |
| 12 | 60 | 11 | 60 | | 60 | 11 |

### 3.8.1 Srovnávací graf

Nascanovat obrázek z Vittera a Chena

## 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 | $n$ |

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

| 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 přemisťováním | $1 + \frac{\alpha}{2}$ | $e^{-\alpha} + \alpha$ |
| 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 | $\frac{1}{2}(1 + \frac{1}{1-\alpha})$ | $\frac{1}{2}(1 + (\frac{1}{1-\alpha})^2)$ |
| dvojité hašování | $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ | $\frac{1}{1-\alpha}$ |
| LISCH | $1 + \dfrac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \dfrac{1}{4}\alpha$ | $1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$ |
| EISCH | $\frac{1}{\alpha}(e^{\alpha} - 1)$ | $1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$ |

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

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

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 Přehašovávání

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 $m = 2^0 m$, $h_1$ do $2m = 2^1 m$, $h_2$ do $4m = 2^2 m \ldots$, $h_i$ do $2^i m$. Množinu $S$ reprezentujeme takto:

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

$$2^{i-2}m < |S| < 2^i m \tag{3.10}$$

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 hašovaní 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 hašovaní je, že předpoklad rovnoměrného výběru hašovací funkce z množiny $H$ mužeme 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 $H \subseteq \{h|h : \{0 \,..\, N-1\} \to \{0 \,..\, m-1\}\}$ je *$c$-univerzální*, kde $c \in \mathbb{R}^+$, jestliže

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

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

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

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

*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

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

$N$ 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 $m \cdot \lceil N/m \rceil^2$.

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

$\square$

### 4.1.1 Očekávaná délka řetězce

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

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

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

Zaveďme

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

$$\delta_h(x,y) = [x \neq y \wedge h(x) = h(y)] \tag{4.2}$$

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

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

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

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 braný přes všechny funkce $h \in H$ při jejich rovnoměrném rozdělení.*

Varianta Markovovy nerovnosti:

Markovova: $\mathcal{P}(X \geq t\mathrm{E}X) \leq 1/t$

**Věta 4.1.3.** *Za stejných předpokladů jako u předchozí věty, 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.* H je $c$-univerzální systém. Nechť $H' = \{h \in H; |S_x| \geq t\mu\}$.

$$\begin{aligned}\mu &= \frac{1}{|H|}\sum_{h\in H} |S_{h,x}| && H' \subset H\\ &> \frac{1}{|H|}\sum_{h\in H'} |S_{h,x}| && \text{z definice } H'\\ &\geq \frac{1}{|H|}\sum_{h\in H'} t\mu\\ &= \frac{|H'|}{|H|} t\mu\end{aligned}$$

a tedy

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

□

### 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 4.1.4.** *Když $H$ je $c$-univerzální systém funkcí z univerza $U$ do $\{0 \mathrel{..} m-1\}$, pak*

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

*Důkaz.* Mějme $c$-univerzální systém $H = \{h_1 \dots h_{|H|-1}\}$. 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

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

† vysvětlit

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

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

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. □

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 $\log_2 |H| = 2 \log_2 N$, tj. 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

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

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

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

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

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

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 $i = 0, 1, \ldots$ 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 $S \subseteq U$ 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 $h : U \to \{0 \,..\, m-1\}$ je perfektní pro $S$, když $\forall x \neq y \in S$ 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 spočitatelná
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 $k \in \{1 .. N-1\}$ a pro pevné $m$ definujme

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

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

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

a pro $k \in \{1 .. N-1\}$ položme

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

Jednak platí

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

a také

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

$$\tag{4.12}$$

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

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

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

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

Pro speciální velikosti tabulky dostáváme dosazením do (4.13):

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

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

*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 perfeknost $h_k$ 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 $i \in \{0 \,..\, n-1\}$ 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 definujme

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

která je perfektní a velikost tabulky je

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

Ovšem na zakódování této funkce potřebujeme hodně paměti: nevadí 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 $1 + n(n-1) \le p_1 \le 1 + 2n(n-1)$. Nějaké takové musí existovat (nedokazujeme). Podle (4.15) $\exists k : h_k(x) = (kx \bmod N) \bmod p_1$ je perfektní na $S$. lepší jména proměnných!

Vytvořme

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

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

Dostáváme hašovací funkci $g_1$, která

- je perfektní pro $S$
- je spočitatelná 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 $S \subseteq \{0 \,..\, N-1\}$ velikosti $n$ existuje prvočíslo $p_0$ takové, že $f_{p_0}(x) = x \bmod p_0$ je perfektní na $S$ a $p_0 = O(n^2 \log N)$.*

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

$$S_0 = \{f_p(s) : s \in S\} \subset \{0 \,..\, p - 1\}$$

a na $S_0$ aplikujme předchozí postup, kde $N = p$.

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

- 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 $p_1, \ldots, p_q$ jsou všechny jeho prvočíselné dělitele. Pak $q = O(\log r / \log\log r)$.*

*Důkaz.*

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

Tedy skok

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

□

*Důkaz lemmatu 4.2.1.* Předpokládejme $S = \{x_1 < \ldots < x_n\}$. 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 $(c \ln D / \ln\ln D) + 1$ prvočísly alespoň jedno, které nedělí $D$. Víme, že $p_k = O(k \ln k)$, tedy $(c \ln D / \ln\ln D) + 1$-ní prvočíslo má velikost $O(\ln D) = O(n^2 \ln N)$. □

nalezeni prvocisla $p_0$ vyzaduje cas $O(n^2 \log N)$.

### 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 Nal: 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** Nal($t$)

### 5.1.2 Algoritmus INSERT

{vyhledej $x$}
**if not** 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 Nal := *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 $t$ := 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 je počet uložených slov násobená délkou cesty a počtem synů, tedy $O(|S|lk)$.

## 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 předchozí algoritmy se jimi musí „prokousat“. Přesně řečeno, prohlížení vrcholu $v$, který má jediného syna, který není list s funkci Nal = *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 idei tyto vrcholy ze stromu vynechat a tím zmenšit (kompresovat) trie.

Ke každému vrcholu $v$ přidáme funkci uroven($v$) vyjadřující číslo úrovně, ve které se $v$ nachází v původním trie. Ke každému listu $v$ přidáme funkci 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 Nal = *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 Nal = *false*. Všimněte si, že každý vnitřní vrchol má právě $k$ synů, které jsou v jednoznačné korespondenci s písmeny abecedy $\Sigma$.

### 5.2.1 MEMBER

Viz algoritmus 5.1

zde

něco chybí

**Algoritmus 5.1** MEMBER pro komprimované trie

{vyhledání $x$}
$t :=$ kořen
**while** $t$ není list **do**
  $i :=$ uroven$(t) + 1$
  $t := t[x_i]$
**end while**
{test}
**return** Nal$(t) \wedge$ slovo$(t) = x$

### 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 $O(nl + kl)$, kde $n$ je velikost reprezentované množiny. Časová složitost operací MEMBER, INSERT a DELETE je v nejhorším případě $O(l)$ a 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 sufixy délky $l-d+1$. Proto těchto množin je

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

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

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

**Algoritmus 5.2** INSERT pro komprimované trie

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

**Algoritmus 5.3** DELETE pro komprimované trie

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

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 kriterium 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 operací MEMBER, INSERT a DELETE pro komprimované trie (za předpokladů rovnoměrného rozložení vstupních dat) je $O(\frac{\log n}{\log k})$. Zde parametr $k$ vyjadřuje vztah mezi prostorovými a časovými nároky.

## 5.3 Ještě komprimovanější trie

### 5.3.1 Popis $A$ a $rd$

### 5.3.2 Jak nalézt $rd$ z $A$

### 5.3.3 Vertikální posun sloupců

### 5.3.4 Úsporné uložení řídkého vektoru

# 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] ... s[j]}
{odpověď ANO, když ∃h : i ≤ h ≤ j ∧ s[h] = x}
d := i {aktuální dolní a horní odhad}
h := j
next := f(d, h) { Předpokládáme, že d ≤ f(d, h) ≤ h }
while s[next] ≠ x ∧ 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 dosadit 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. | $f(d,h) = fkvadrat$ | $O(\log(n))$ | $O(\log(\log(n)))$ |
| kvadratické vyhledávání | | $O(\log(n))$ | $O(\log(\log(n)))$ |

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

## 6.2 Zobecněné kvadratické vyhledávání

algoritmus vychází z Pavlova, výklad napsal Jakub Černý

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.

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). [1] Po interpolačním dotazu se neustále střídají skoky o $\sqrt{delka}$ 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 $\sqrt{delka}$) a *false* (binární vyhledání)}
**if** *nastav* **then**
  $parita := true$
  $delka := h - d + 1$
  $next := d + \left\lceil \frac{x-s[d]}{s[h]-s[d]} \cdot delka \right\rceil$ $\{= finterp(d,h)\}$
  $nahoru := s[next] < x$
  $nastav := false$
  return *next*
**end if**
**if** not *parita* **then**
  $next := \lceil (d+h)/2 \rceil$ $\{= fbin(d,h)\}$
  $parita := true$
  return *next*
**end if**
$next := nahoru\ ?\ d + \sqrt{delka} : h - \sqrt{delka}$
**if** $s[next] < x$ 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. Pojďme 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 rámci bloku prvního (interpolačního) dotazu. Srovnej

[1] zde by byl vhodný obrázek - usečka, která má na krajich d a h a je na ni videt prvni interpolačni dotaz a skoky po sqrt(n) a bin. a sqrt(n) ...

$prv$ 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 $p_i = \mathcal{P}(\text{v rámci bloku bylo položeno alespoň } i \text{ dotazů})$. 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 $O(\text{očekávaný počet dotazů v bloku}) = O(\sum_{i=0}^{\infty} p_i) = O(\,3 + \sum_{i=3}^{\infty} 1/(i-2)^2) = O(\,3 + \pi^2/6) = O(4.6)$. 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 $O(\,(\text{počet bloků})\,(\text{očekávaný čas pro 1 blok})) = O(\,\log\log(|S|)\,O(1)) = O(\,\log\log(|S|))$. 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

### 7.2.1 Algoritmus konstrukce

### 7.2.2 Snížení složitosti z kubické na kvadratickou

## 7.3 Skorooptimální binární vyhledávací stromy

Jenom že existují, lineární konstrukce. Aha — viz cvičení.

## 7.4 Červenočerné stromy

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.

Pro binární vyhledávací červenočerné stromy reprezentující množinu $S$ platí: 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?

### 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 7.4.1.** 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* otce, 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 7.4.2.** 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 7.4.3.** 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 $s1$, $s2$, 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 červeno, 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 $s1$, 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 $s1$, 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 $s1$ bude mít barvu, kterou původně měl vrchol $o$ a barva vrcholu $s2$ 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 dvojité 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`. [1]

[1] 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.

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

**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
- $S_v[1 \mathinner{..} \rho_v]$ je pole ukazatelů na syny
- $H_v[1 \mathinner{..} \rho_v - 1]$: $H_v[i]$ je maximální prvek v podstromu $S_v[i]$

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 $H_o$, $H_u$ $\{\ldots\}$
  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 $H_o$
      $t := o$

**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 $S_t[1] \ldots S_t[i-1]$
    Vytvoř strom $T_2$, jehož kořen má syny $S_t[i+1] \ldots S_t[\rho_t]$
    **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 $T_1 \ldots T_k$, přičemž $T_1$ je vrchol zásobníku. Platí

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

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

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:

- $K_v[1 .. \rho_v]$: $K_v[i]$ 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 pětkrát 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. proč?

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

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

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 přidaných vrcholů $\cdot$ čas přidání vrcholu $+$ počet štěpení $\cdot$ čas štěpení $= O(n) \cdot O(1) +$ 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 $O(\sum \log f_i)$. ošetřit $\log 0$

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. nebo transpozic? standardní termín?

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 zatřiďuje 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 prostorové nároky mezi $(a,b)$-stromy. Na druhé straně však $(2,3)$-stromy v obecném přépadě vyžadují zbytečně mnoho vyvažovacích operací, a proto jsou výrazně pomalejší než např. $(2,4)$-stromy. Protože 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 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.

udělat obrázek ilustrující zbytečná š/s

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

4'. $a \geq 2$ a $b \geq 2a$

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

$o :=$ lock(nadkořen) {Nadkořen je implementační pomůcka. Slouží k zamknutí přístupu k celému stromu a uchovává max($S$)}<br>
$t :=$ kořen<br>
{Invariant mezi průchody cyklem: $o$ je otec $t$, $o$ je jediný vrchol zamknutý tímto procesem.}<br>
**while** $t$ není list **do**<br>
  $i := 1$<br>
  **while** $i < \rho_t \wedge H_t[i] < x$ **do**<br>
    $i := i + 1$<br>
  **end while**<br>
  $s := S_t[i]$<br>
  {preventivní rozštěpení:}<br>
  **if** $\rho(t) = b$ **then**<br>
    rozděl $t$ na $t_1$ a $t_2$: {viz 4'}<br>
      k $t_1$ dej prvních $\lfloor (b + 1)/2 \rfloor$ synů $t$<br>
      k $t_2$ dej zbylých $\lceil (b + 1)/2 \rceil$ synů $t$<br>
      $t_1$ předchází $t_2$<br>
    uprav $\rho_o$, $S_o$ a $H_o$<br>
    {implic.: uprav $\rho_{t_1}, \ldots, H_{t_2}$}<br>
    {při štěpení kořene ještě musíme vytvořit nový kořen}<br>
    $n := t_j$, kde $s$ je syn $t_j$<br>
  **else**<br>
    $n := t$<br>
  **end if**<br>
  lock($n$)<br>
  unlock($o$)<br>
  $o := n$<br>
  $t := s$<br>
**end while**<br>
**if** $t$ nereprezentuje $x$ **then**<br>
  vrcholu $o$ přidej nového syna $t'$ reprezentujícího $x$<br>
  zařaď $t'$ na správné místo mezi jeho bratry a uprav $\rho_o$, $S_o$ a $H_o$<br>
**end if**<br>
unlock($o$)

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

$o :=$ lock(nadkořen) {Nadkořen je implementační pomůcka. Slouží k zamknutí přístupu k celému stromu a uchovává max($S$)}<br>
$t :=$ kořen<br>
$h :=$ **nil** {Jakmile $h \neq$ **nil**, $x \in H_h$ a $h$ bude zamčený do konce procesu.}<br>
{Invariant mezi průchody cyklem: $o$ je otec $t$, $o$ je kromě $h$ jediný vrchol zamknutý tímto procesem.}<br>
**while** $t$ není list **do**<br>
  $i := 1$

```
    while i < ρ_t ∧ 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 ρ(t) = a then
      v := bezprostřední bratr t
      if ρ_v = a then {smíme spojit}
        {Spojení}
        sluč v a t do t {viz 4'}
        uprav ρ_o, S_o a H_o
        t := o
      else {ρ_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 ≠ 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 ρ_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*

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

*Pak platí*

$$P \leq n \tag{8.2}$$
$$(2c-1)ST + cSP \leq n + c + \frac{c}{a+c-1}(n-2) \tag{8.3}$$
$$P_h + SP_h + ST_h \leq \frac{2n^{c+2}}{(c+1)^h} \tag{8.4}$$

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

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

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. Myslel 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.* vyjasnit

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

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

$$z(v) = \min(\rho_v - a, b - \rho_v, c) \qquad v \text{ je vnitřní vrchol různý od kořene} \tag{8.6}$$
$$z(\text{kořen}) = \min(\rho_v - 2, b - \rho_v, c) \tag{8.7}$$

Pro strom $T$ definujme

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

Platí

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

Podobně jako u červenočerných stromů definujme 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:

$$\rho_v = a - 1 \text{ nebo } b + 1 \implies z(v) = -1 \tag{8.8}$$
$$\rho_v = a \text{ nebo } b \implies z(v) = 0 \tag{8.9}$$
$$\rho_v = 2a - 1 \implies z(v) = c \tag{8.10}$$
$$\rho_u = \left\lfloor \frac{b+1}{2} \right\rfloor \wedge \rho_v = \left\lceil \frac{b+1}{2} \right\rceil \implies z(u) + z(v) \geq 2c - 1 \tag{8.11}$$
$$|\rho_u - \rho_v| \leq 1 \implies z(u) \geq z(v) - 1 \tag{8.12}$$

### 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:

$$z_1(T') \geq z_1(T) - 1 \tag{8.13}$$
$$z_h(T') = z_h(T) \quad h > 1 \tag{8.14}$$
$$z(T') \geq z(T) - 1 \tag{8.15}$$

### 8.4.2 štěpení

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

$$z_h(T') \geq 2c + z_h(T) \qquad \text{z 8.8 a 8.11} \tag{8.16}$$
$$z_{h+1}(T') \geq z_{h+1}(T) - 1 \tag{8.17}$$
$$z_i(T') = z_i(T) \quad i \neq h, h+1 \tag{8.18}$$
$$z(T') \geq z(T) + 2c - 1 \tag{8.19}$$

### 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 $(T', \text{otec } v$ je parciální $(a,b)$-strom a platí:

$$z_h(T') \geq c + 1 + z_h(T) \qquad \text{z 8.8, 8.9 a 8.10} \tag{8.20}$$
$$z_{h+1}(T') \geq z_{h+1}(T) - 1 \tag{8.21}$$
$$z_i(T') = z_i(T) \quad i \neq h, h+1 \tag{8.22}$$
$$z(T') \geq z(T) + c \tag{8.23}$$

### 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:

$$z_h(T') \geq z_h(T) \qquad \text{z 8.8, 8.9 a 8.12} \tag{8.24}$$
$$z_i(T') = z_i(T) \quad i \neq h \tag{8.25}$$
$$z(T') \geq z(T) \tag{8.26}$$

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

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

$$z_1(T_k) \geq 2cST_1 + (c+1)SP_1 - n \tag{8.28}$$
$$z_h(T_k) \geq 2cST_h + (c+1)SP_h - ST_{h-1} - SP_{h-1} \quad h > 1 \tag{8.29}$$

Vadí nám, že jsou ve výrazu i spojení a štěpení z jiné hladiny.
$c \geq 1 \implies 2c \geq c+1$.

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

$$ST_h + SP_h \leq \frac{z_h(T_k)}{c+1} + \frac{ST_{h-1} + SP_{h-1}}{c+1} \leq \frac{z_h(T_k)}{c+1} + \frac{z_{h-1}(T_k)}{(c+1)^2} + \frac{ST_{h-2} + SP_{h-2}}{(c+1)^2} \tag{8.30}$$
$$\leq \left(\sum_{i=0}^{h-1} \frac{z_{h-i}(T_k)}{(c+1)^{i+1}}\right) + \frac{ST_0 + SP_0}{(c+1)^h} \qquad j = h-i, \text{ rozšíříme } (c+1)^{h-i} \tag{8.31}$$
$$= \left(\sum_{j=1}^{h} \frac{z_j(T_k)(c+1)^j}{(c+1)^{h+1}}\right) + \frac{n}{(c+1)^h} \tag{8.32}$$

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

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

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

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

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

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

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

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

Spojením tohoto výsledku s 8.27 dostaneme 8.3.
K důkazu 8.4 využijeme 8.32.

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

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

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

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

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

označme $s_i = \sum_{j=0}^{c} m_{i,j}$

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

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

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

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

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

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

### 8.5.1 Algoritmus MEMBER

# 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 INSERT(x), tak předpokládejme, že je na 1. pozici po ukončení seznamu.

## 9.1 Amortizovaná složitost

## 9.2 Seznamy

### 9.2.1 Algoritmus MEMBER

### 9.2.2 Algoritmus INSERT

### 9.2.3 Algoritmus DELETE

### 9.2.4 Algoritmus MFR (Move Front Rule)

Při operaci MEMBER(x) je x v seznamu nebo při operaci INSERT(x) bude x po skončení operace na 1. místě seznamu.

**Věta 9.2.1.** *Mějme posloupnost P operací MEMBER, INSERT a DELETE a mějme dva prosté seznamy S1, S2 množiny S.*
*Pak pro každý upravující algoritmus A platí:*
*Když MFR provede P na seznam S1 a A provede P a seznam S2, tak platí:*
*a) čas MFR $\leq$ čas na vyhledání A + počet placených výměn A - počet volných výměn A - $|P|$ když S1 = S2*
*b) čas MFR $\leq$ čas na vyhledání A + počet placených výměn A - počet volných výměn A - $|P|$ + $\binom{|S|}{2}$ když S1 $\neq$ S2*

**Definice 9.2.1.** Nechť S1, S2 jsou dva prosté seznamy množiny S, pak bal(S1,S2) je počet neuspořádaných dvojic x,y, x $\neq$ y, x,y $\in$ 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 $\Leftrightarrow$ S1 = S2 (prvky jsou ve stejném pořadí $\Leftrightarrow$ seznamy jsou stejné)
bal(S1,S2) $\leq \binom{|S|}{2}$ (všechny dvojice jsou přeházené)

*Důkaz věty XXX.* Přes amortizovanou složitost A.
Předpokládejme, že A i MFR mají provést operaci O.
A … provádí na seznam $S_A$, výsledek bude $S'_A$
MFR .. provádí O na seznam $S_{MFR}$, výsledek bude $S'_{MFR}$
amortizovaná složitost operace O bude čas MFER pro operaci O + bal($S'_A$, $S'_{MFR}$) - bal($S_A$, $S_{MFR}$)
balance je def. vzhledem k algoritmu A.

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

$$\mathrm{S_A} \overset{vyhledn}{\rightarrow} S''_A \overset{vmny}{\rightarrow} S'_A$$

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

kde po operaci

| | |
|---|---|
| DELETE(x) | $S''_A = S'_A$ |
| MEMBER(x) | $S''_A = S_A$ |
| INSERT(x) | x je v seznamu , $S''_A = S_A$ |
| | x není v seznamu, $S''_A$ vznikne z $S'_A$ přidáním x za poslední prvek seznamu |

Podstatné je, že seznamy jsou nad stejnou množinou

Amort. složitost první části $\leq$ 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) $S''_A = S'_{MFR}$ a tady amort. slož. MFR = čas operace = n++ $\leq$ 2(n+1) - 1
n+1 je čas na vyhledání pro A - 1
$S''_A$ vznikne z $S_A$ přidáním x za posl. prvek $S_A$
$S'_{MFR}$ vznikne z $S_{MFR}$ 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 $S_A$ na j-tém místě v seznamu $S_{MFR}$ Čas operace pro MFR je j, čas na vyhledání pro A je i. Označme k počet y v seznamu takových, že y je v $S_A$ za x, v $S_{MFR}$ před x.

Pak i+k $\geq$ j (i+k $\geq$ i-k+j) amort. slož. pro MFR = j + bal($S''_A$, $S'_{MFR}$) - bal($S_A$, $S_{MFR}$)

DELETE(x) bal($S''_A$, $S'_{MFR}$) - bal($S_A$, $S_{MFR}$) $\leq$ -k amort. slož. $\leq$ j - k $\leq$ 2i - 1 = 2*čas na vyhledání A - 1

MEMBER(x), INSERT(x) bal($S''_A$, $S'_{MFR}$) - bal($S_A$, $S_{MFR}$) $\leq$ -k + i-1 (nějaké dvojice mohly přibýt) amort. slož. operace MFR $\leq$ j-k+i-1 $\leq$ i+i-1 = 2i - 1 = 2*čas na vyhledání A-1

Amort. slož.
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 prvkem 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) □

**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 pro posloupnosti P aplikované na seznam S, kde P obsahuje jen operace MEMBER(x) pro x $\in$ S.

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

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 $\Pi$.
$p_1 = \sum_\Pi \kappa_\Pi \Pi(i) \ldots$ očekávaná pozice prvku i $\delta(j,i)$ = asmyptot. pst., že prvek j je před i, pak platí

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

pak

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

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 $\beta_1 \geq \beta_2 \geq ... \geq \beta_n$
pak nejrychlejší algoritmus na seznam $x_1 - x_2 - ... - x_n$ je klasický algoritmus bez přemísťování prvků. Očekávaný čas tohoto algoritmu je

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

## 9.3 Splay stromy

patová struktura - binární vyhledávací stromy s ohodnocenými prvky

### 9.3.1 Operace SPLAY

Základní operací je pr 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 reprez. 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, $\triangle$).

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, $\triangle$) zjistí, zda x leží ve stromě a pokud ano, pak k jeho váze přičte $\triangle$.

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

jaké mají být argumenty Splay() ?

### 9.3.5 Algoritmus JOIN3

Viz algoritmus 9.3

zde chybi obrazek

### 9.3.6 Algoritmus SPLIT

Viz algoritmus 9.4

zde chybi obrazek

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

vytvoříme vrchol t reprezentující x
Splay()
kořen $T_1$ je levý syn t
kořen $T_2$ je pravý syn t

**Algoritmus 9.4** SPLIT(x)

```
SPLAY(x)
if kořen T reprezentuje x then
  T1 podstrom levého syna kořene
  T2 podstrom pravého syna kořene
  výstup T1, T2, x, x ∈ S
else
  if kořen T reprezentuje prvek < x then
    T2 podstrom pravého syna kořene T
    T1 = T − T2
    T1 podstrom pravého syna kořene T
    T2 = T − T1
  end if
  výstup T1, T2, x ∈ S
end if
```

### 9.3.7 Algoritmus DELETE

Viz algoritmus 9.5

**Algoritmus 9.5** DELETE(x)

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

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

### 9.3.8 Algoritmus INSERT

Viz algoritmus 9.6

**Algoritmus 9.6** INSERT(x)

```
SPLAY(x)
if x není v kořeni 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, △)

SPLAY(x)
**if** x je v kořeni **then**
  k váze x přičti △
**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) $r(x) = \lfloor logtW(x) \rfloor$

bal(konfigurace) = $\sum r(W), W jeprvekkonfigurace$

Pro strom T je tW(x) = tW(kořene T) r(T) = r(kořen T)

**Lemma 9.3.1.** *Nechť T je binární vyhledávací strom, t t je vnitřní vrchol a u,v jsou synové t. Pak* $r(t) > minr(u), r(v)(r(list) = -\infty)$.

*Důkaz.* Předpokládejme, že $tW(u) \leq tW(v)$

$$r(t) = \lfloor logtW(t) \rfloor \geq \lfloor log2tW(u) \rfloor = 1 + \lfloor logtW(u) \rfloor = 1 + r(u)$$

□

### 9.3.10 Algoritmus SPLAY

Viz algoritmus 9.8

t se po skončení operace SPLAY(x) dostane do kořene

### 9.3.11 Amortizovaná složitost SPLAY

čas operace SPLAY = počet opakování cyklu, když vrchol t transportujeme do kořene

**Lemma 9.3.2.** *Amortizovný čas operace SPLAY(x,T)* $\leq 3(r(T) - r(t)) + 1$, *kde t je vrchol, který transportujeme do kořene. (když x je pvekem reprez. množiny, pak t reprezentuje x, jinak je to buď největší nebo nejmenší prevek menší (větší) než x)*

*Důkaz.* rozdělíme podle akce, která se provádí ve while cyklu a) while cyklus provádí rotace

Amortizovaná složitost tohoto kroku = tady je děsný zmatek

$$= asoperace + bal(novkonf.) - bal(pvodnkonf.) = 1 + r'(u) - r(v) \leq 1 + r(u) + r(v) \leq 1 + 3(r(u) - r(v)) \quad (9.4)$$

protože x má v původním i novém stromě stejné prvky nečitelné

$$r(u) \quad = \quad r'(t)r'(u) \quad \leq \quad r'() \quad = \quad r(u) \quad (9.5)$$

b) while cyklus provádí dvojitou rotaci

Amortizovaná složitost této operace = = čas operace + bal(nová konf.) - bal(původní konf.) = = 1 + r'(u) - r(v) - r(u) - r(t) (*)

**Algoritmus 9.8** CHANGEWEIGHT(x, △)

```
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 while
```

pro $x \neq t, u, v$ platí $r(x) = r'(x)$ r(v) = r'(t)
b1)

$$r(v) > r(t), pakr'(u), r'(v) \leq r'(t) = r(v)r(u) \geq r(t), 1 \leq r(v) - r(t)(*) \leq r(v) - r(t) + 2r(u) - 2r(t) = 3(r(v) - r(t))$$

b2)

$$r(v) = r(t), pakpodlelemmatur'(t) > minr'(u), r'(v)pak2r'(t) \geq r'(u) + r'(v) + 1(*) \leq 2r(u) - 2(r(t) = 2(r(v) - r(t)) = (r(t) = 0)3($$

□

### 9.3.12 Amortizovaná složitost ostatních operací

# Kapitola 10

# Haldy

## 10.1 $d$-regulární haldy

### 10.1.1 Algoritmus UP

### 10.1.2 Algoritmus DOWN

### 10.1.3 Operace

INSERT, MIN, DELETEMIN, DECREASEKEY, INCREASEKEY, DELETE

### 10.1.4 Algoritmus MAKEHEAP

A jeho čas

### 10.1.5 Dijkstrův algoritmus

## 10.2 Leftist haldy

## 10.3 Binomiální haldy

### 10.3.1 Zobecněné binomiální haldy

## 10.4 Fibonacciho haldy

# Kapitola 11

# Dynamizace

V uspořádaném poli umíme rychle vyhledávat, ale přidat prvky znamená celé ho přebudovat. Ve strůstajícím hašování zase nešly prvky mazat, ve velmi komrimovaných 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 $f : Q_1 \times 2^{Q_2} \to Q_3$, kde $Q_1$, $Q_2$ a $Q_3$ jsou univerza.

**Definice 11.1.2.** *Řešení vyhledávacího problému* pro $x \in Q_1, A \subseteq Q_2$ je nalezení hodnoty $f(x, A)$.

Například

**Klasický vyhledávací problém:** $Q_1 = Q_2 = U$, univerzum prvků; $Q_3 = \{0, 1\}$;

$$f(x, A) = \begin{cases} 0 & \text{když } x \notin A \\ 1 & \text{když } x \in A \end{cases}$$

**Vzdálenost bodů v rovině:** $Q_1 = Q_2 =$ rovina, třeba euklidovská; $Q_3 = \mathbb{R}^+$; $f(x, A) =$ vzdálenost bodu $x$ od množiny $A$.

**Příslušnost ke konvexnímu obalu** $Q_1 = Q_2 =$ rovina; $Q_3 = \{0, 1\}$;

$$f(x, A) = \begin{cases} 0 & \text{když } x \text{ nepatří do konvexního obalu } A \\ 1 & \text{když } x \text{ patří do konvexního obalu } A \end{cases}$$

**Definice 11.1.3.** Vyhledávací problém je *rozložitelný*, když existuje operace $\oplus$ spočitatelná v konstantním čase a platí: když $x \in Q_1$ a $A$ a $B$ jsou disjunktní podmnožiny $Q_2$, pak

$$f(x, A \cup B) = f(x, A) \oplus f(x, B).$$

Z výše uvedených příkladů není rozložitelným problémem příslušnost ke konvexnímu obalu, ostatní dva vyhledávací problémy jsou rozložitelné.

Nechť $f$ je rozložitelný vyhledávací problém a $\mathcal{S}$ je "statická" datová struktura, která ho řeší. Neboli $\mathcal{S}$ je tvořena pro pevnou množinu $A \subseteq Q_2$ a obsahuje operaci, která pro vstup $x$ počítá $f(x, A)$.

Popíšeme důležité parametry $\mathcal{S}$: nechť $n = |A|$, označme

$$\begin{aligned} Q_{\mathcal{S}}(n) &= \text{čas potřebný pro výpočet } f(x, A) \\ S_{\mathcal{S}}(n) &= \text{paměť potřebná pro vybudování } \mathcal{S} \\ P_{\mathcal{S}}(n) &= \text{čas potřebný pro vybudování } \mathcal{S} \end{aligned}$$

Požadujeme, aby $Q_{\mathcal{S}}(n)$, $S_{\mathcal{S}}(n)/n$ a $P_{\mathcal{S}}(n)/n$ byly neklesající funkce.
Chceme navrhnout strukturu, která by uměla

1. Pro $x \in Q_1$ a pevné $A \subseteq Q_2$ rychle spočítat $f(x, A)$.
2. Pro $A$ a $y \in Q_2$ rychle vytvořit strukturu pro $A \cup \{y\}$.

Mějme $A_0, A_1, \ldots$ 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 $\mathcal{D}$ reprezentující $A$ je potom

- Seznam statických struktur pro $A_i \neq \emptyset$, $\mathcal{S}_i$
- Vyhledávací strom reprezentující $A$
- Pro každé $A_i \neq \emptyset$ seznam prvků v $A_i$; prvky těchto seznamů jsou projpojeny s odpovídajícími prvky ve stromě.

## Kapitola 12

# Vícedimenzionální vyhledávání