Datové struktury

přednáší RNDr. Václav Koubek, DrSc.

zTEXali a doplnili
Martin Vidner <mvidner@atlas.cz>
Vladimír Kotal <vlada@devnull.cz>

Universita Karlova
Matematicko-fyzikální fakulta
2004

Obsah

Předmluva

Když jsem se někdy na začátku r. 2002 rozhodl doplnit původní neúplná skripta Martina Vidnera, netušil jsem, kolik času to zabere. Původně jsem na nich začal pracovat proto, že jsem neměl žádné rozumné poznámky z přednášek a věděl jsem, že při učení na státnice nebudu chtít trávit čas hledáním a tříděním poznámek z různých zdrojů. Nakonec jsem skončil dopsáním chybějících kapitol, dokreslením obrázků a opravením všech chyb, které jsem našel,

Stále je ovšem pravděpodobné, že skripta obsahují i vážnější chyby. Pokud nějaké při učení najdete, zkuste mi napsat.

Skripta může číst prakticky každý, kdo je obeznámen se základy teoretické informatiky, rozhodně ale doporučuji absolvovat předměty '*Úvod do teorie pravděpodobnosti*' a '*Úvod do složitosti a NP-úplnosti*' nebo jejich ekvivalenty. První se vám bude hodit při důkazech, druhý při obecném chápání algoritmů.

Obsahově skripta pokrývají přednášky Datové struktury I a II RNDr. Václava Koubka, DrSc. Navíc jsou přidány některé věci, přednášené okrajově nebo probírané pouze na cvičení. (např. AVL stromy nebo skorooptimální binární vyhledávací stromy) Některé kapitoly byly psány nezávisle na přednáškách. (např. splay stromy)

Řekl bych "užijte si to", ale ono to zase tak lehké čtivo není :)

V. Kotal

Poděkování patří následujícím lidem:

- Martin Vidner
  původní verze skript (kapitoly Hašování I,II, XXX)
- Lišák, Jéňa, Žabička, Jindřich[1], Martin Mareš, Pavel Machek, Jakub Černý
  opravy a dodatky v původní verzi skript
- Martin Mačok
  faktické opravy
- Tomáš Matoušek
  množství faktických oprav
- Ladislav Prošek
  překlepy, faktické opravy
- Jana Skotáková, Martin Malý
  zapůjčení poznámek
- Vojtěch Fried
  algoritmus pro INSERT v semidyn. systémech
- Michal Kováč
  překlepy, faktické opravy
- Vojta Havránek
  faktické opravy

Některé části skript byly volně převzaty ze zdrojů jiných než z originální přednášky Datové struktury. Konkrétně části sekcí o binomiálních a fibonacciho haldách byly inspirovány přednáškou O. Čepka, sekce o splay stromech byla částečně převzata z textů FIT VUTBR. Sekce o semi-optimálních vyhledávacích stromech je tvořena referátem L. Strojila. Sekce o AVL stromech vznikla kompilací a doplněním materiálů z webu, základ tvoří referát Vojtěcha Muchy.

[1] Lidé, kterým děkoval Martin Vidner. Identita těchto jedinců zřejmě zůstane navždy utajena.

Kapitola 1

Úvod

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

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

1.1 Předpoklady

1. Datové struktury jsou nezávislé na řešeném problému; abstrahujeme. Například u slovníkových operací *vyhledej, přidej, vyjmi,* nás nezajímá, jestli slovník reprezentuje body v prostoru, vrcholy grafu nebo záznamy v databázi.

2. V programu, který řeší nějaký problém, se příslušné datové struktury používají *velmi často.*

1.2 Jaké typy složitosti nás zajímají

1.2.1 Paměťová složitost reprezentované struktury

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

1.2.2 Časová složitost algoritmů pracujících na datové struktuře

Časová složitost v nejhorším případě

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

Očekávaná časová složitost

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

Pozor na předpoklady o rozdělení vstupních dat. Je známý fakt, že pro každé k existuje číslo n_k takové že každý náhodný graf s alespoň n_k vrcholy s velkou pravděpodobností obsahuje kliku velikosti k. To vede k následujícímu algoritmu, který určí zda graf je nejvýše $k-1$ barevný s očekávaným konstantním časem.

Algoritmus: vstup graf (V, E).

1. Když velikost V je menší než n_k, pak prohledáním všech možností 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:

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

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

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

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 × 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

Je dáno universum U, reprezentovaná podmnožina $S, S \subseteq U, |S| = n$. Velikost tabulky, ve které budeme chtít S reprezentovat je m.

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

Charakteristickou vlastností obecné hašovací 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 sloučit do jednoho nebo i překrýt s těmi kladnými. To je základní idea hašování.

Problémy:

1. Jak rychle spočítáme $h(s)$.

2. Co znamená *uložen na místě* $h(s)$? Co když platí $h(s) = h(t)$ a zároveň $s \neq t$?

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

Tento způsob hašování řeší kolize tak, že kolidující prvky ukládá na stejnou pozici do tabulky. Na této pozici jsou uloženy v podobě lineárních seznamů. Takovému seznamu kolidujících prvků se říká *řetězec*. Hašovací tabulka je tedy pole lineárních seznamů, ne nutně uspořádaných. Odtud plyne název této metody, protože řetězce prvků nejsou mezi sebou promíchány - řetězec, který je v tabulce uložen na pozici y v sobě obsahuje pouze ty prvky, pro které platí $h(x) = y$.

Základní operace na této tabulce jsou MEMBER (viz algoritmus 3.1), INSERT (viz alg. 3.2) a DELETE (viz alg. 3.3).

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

Algoritmus 3.1 MEMBER pro hašování se separovanými řetězci

MEMBER(x):

1. Spočítáme $h(x)$.

2. Prohledáme $h(x)$-tý seznam.

3. Když tam x je, vrátíme *true*, jinak *false*.

Algoritmus 3.2 INSERT pro hašování se separovanými řetězci

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.

Algoritmus 3.3 DELETE pro hašování se separovanými řetězci

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.

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*[1], 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 ℓ 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:

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

[1] anglicky *load factor*

Očekávaný počet testů pro vyhledání jednoho prvku

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

3.1.4 Očekávaná délka nejdelšího seznamu

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

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

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 $q(j)$:

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

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

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

Nechť

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

pak

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

3.2 Hašování s uspořádanými řetězci

Uspořádání řetězců vylepší neúspěšný případ, protože hledání v řetězci můžeme zastavit v okamziku, kdy narazíme na prvek větší než hledaný prvek (za předpokladu, že prvky v řetězci jsou uspořádány vzestupně).

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`.

Příklad 3.3.1. Například pro $U = \mathbb{N}, m = 10, h(x) = x \bmod 10$, hašujeme posloupnost 10, 50, 21, 60:

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

MEMBER(x) je jednoduchý - od $h(x)$-tého místa procházíme řetězec prvků až na konec a v případě nalezení prvku operaci ukončíme.

Při INSERT musíme zjistit, zda $h(x)$-tý řetězec začíná na $h(x)$-tém místě[2]. 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.

[2]To je jednoduché - pokud prvek na $h(x)$-tém místě má nulový ukazatel `vzad`, pak je začátkem $h(x)$-tého řetězce.

Příklad 3.3.2. Tabulka z předchozího příkladu po INSERT(53) bude vypadat takto:

	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 $h(x) = i$.

Příklad 3.4.1. Nechť $h(x) = x \bmod 10$. Ukládáme prvky 50, 90, 31, 60:

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

Přidáme 42, 72, 45:

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

3.5 Hašování s lineárním přidáváním

Jde to i bez ukazatelů.

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

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

Přidáme 40, 98, 62, 108:

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

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

Použijeme dvě hašovací funkce, h_1 a h_2, je zde však účelné předpokládat, že $h_2(i)$ a m jsou nesoudělné pro každé $i \in U$. Při INSERTu pak hledáme nejmenší j takové, že $T[h_1(x) + jh_2(x)]$ je volné.

Příklad 3.6.1. 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

viz algoritmus 3.4.

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.

Algoritmus 3.4 INSERT pro hašování se dvěma funkcemi

INSERT(x)

1. spočti $i = h_1(x)$

2. když tam x je, pak skonči
když je místo prázdné, vlož tam x a skonči

3. **if** i-té místo obsazeno prvkem $\neq x$ **then**
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**
$k = (k + h_2(x)) \bmod m$
end while
end if

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

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

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

dostaneme po dosazení:

$$\ldots = 1 + \sum_{j=1}^{\infty}\frac{n}{m}q_{j-1}(n-1,m-1) = 1 + \frac{n}{m}\sum_{j=1}^{\infty}q_{j-1}(n-1,m-1)$$

$$= 1 + \frac{n}{m}C(n-1,m-1) \quad (3.6)$$

Řešením tohoto rekurentního vzorce je

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

což dokážeme indukcí:

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

při úspěšném vyhledávání

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

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

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

α	0.5	0.7	0.9	0.95	0.99	0.999
$\frac{1}{1-\alpha}$	2	3.3	10	20	100	1000
$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$	1.38	1.7	2.55	3.15	4.65	6.9

3.7 Srůstající hašování (standardní: LISCH a EISCH)

XXX doplnit odhady pro úspěšný/neúspěšný případ pro EISCH, LISCH, .. z 2003/2004.

Další metodou jak řešit kolize je neseparovat řetězce. To znamená, že v jednom řetězci se nyní mohou nacházet prvky, které mají různé hodnoty hašovací funkce. Proto se této metodě říká *srůstající hašování*. Řetězce se nepřesouvají, v tabulce tedy nemusí být obsažena informace kde začíná daný řetězec.

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

Příklad 3.7.1. Opět použijeme hašovací funkci $h(x) = x \mod 10$. V následující tabulce jsou srostlé řetězce pro hodnoty 0 a 3 funkce h:

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

3.7.1 Algoritmus INSERT

viz algoritmus 3.5

Algoritmus 3.5 INSERT pro srůstající hašování

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.

Příklad 3.7.2. Po provedení operací LISCH INSERT(103), EISCH INSERT(94) bude tabulka z příkladu 3.7.1 vypadat takto:

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

Poznámka 3.7.1. Při úspěšném vyhledání je EISCH asi o 15% rychlejší než LISCH. (Při neúspěšném jsou samozřejmě shodné, protože v obou případech je nutné projít celý řetězec.)

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

Standardní srůstající hašování má tu nevýhodu, že se při větším zaplnění tabulky mohou vytvořit dlouhé řetězce. Tabulku tedy prodloužíme o pomocnou pamět (tzv. *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.

Příklad 3.8.1. Provedeme operaci INSERT pro následující posloupnost prvků: 10, 41, 60, 70, 71, 90, 69, 40, 79. Tabulka pak bude vypadat takto:

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

Sklep je od "hlavní části" tabulky odlišen čarou.

3.8.1 Operace DELETE

viz algoritmus 3.6

Algoritmus 3.6 DELETE pro srůstající hašování s pomocnou pamětí

1. spočítáme $h(x)$ a prohledáme řetězec začínající na adrese $h(x)$.
když neobsahuje $x \rightarrow$ konec.

2. když x je na adrese $h(x)$ a v řetězci následuje prvek v pomocné části tabulky (sklepě) - odstraníme x. Prvek, který následuje za x přesuneme na místo $h(x)$ a uvolníme jeho místo - konec.

3. x je ve sklepě - zjístíme, zda se nachází v řetězci mimo sklep prvek s hašovací hodnotou $h(x)$ - pokud neexistuje, přesuneme poslední prvek v řetězci, který je ve sklepě na místo x a místo posledního prvku uvolníme.

Když takový prvek existuje, vezmeme první prvek s touto vlastností. Označme ho y, pak y přeneseme na místo x a budeme chtít uvolnit místo prvku y. (obrazně řečeno x a y v řetězci prohodíme)

4. x je v části, kam může hašovací funkce a řetezec v této části pokračuje. pak x odstraníme a prvky za x zahašujeme do tabulky v pořadí jak se do ní ukládaly a pokud vzniká kolize, umístíme je zpátky na místo, kde byly.

Příklad 3.8.2. Použijeme hašovací funkci $h(x) = x \mod 10$
Provedeme operaci DELETE na prvky 41, 42.

? 42 v puv. tabulce nebylo XXX sehnat puvodni priklad

- metoda LICH:
  odstraníme prvky 62,78,82,52 z řetězce a provedeme postupně INSERT: 62,78,82,52
- metoda VICH:
  odstraníme 62,78,82,52 a provedeme INSERT: 82,62,78,52. pokud bychom provedli INSERT 82,62,52,78, pak bychom nepoznali, v jakém to bylo v tabulce pořadí.

	klíč	`vpřed`
0		
1	11	10
2	32	9
3	43	
4		
5	42	9
6	82	
7	78	6
8	62	7
9	52	8
10	21	11
11	41	12
12	61	

3.8.2 Srovnávací graf

XXX

Nascanovat obrázek z Vittera a Chena [6]

3.9 Srovnání metod

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

3.9.1 Pořadí podle neúspěšného případu

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)

3.9.2 Počet testů pro zaplněnou tabulku

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

co znamenají čísla v pravém sloupci ?

typ	XXX
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

3.9.3 Počet testů vyjádřený vzorcem

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	$1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{1}{4}\alpha$	$1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$
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 λ 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.

3.10.1 MEMBER

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

3.10.2 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.

3.10.3 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.

3.10.4 Složitost

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 můžeme zajistit (nebo se k splnění tohoto požadavku přiblížit), ale výběr vstupních dat ovlivnit nemůžeme. Nyní tuto ideu zformalizujeme.

zajímá nás jednak c, jednak velikost systému (tj. počet funkcí v systému)

Definice 4.1.1. Třída hašovacích funkcí $H \subseteq \{h | h : \{0 \,..\, N-1\} \to \{0 \,..\, m-1\}\}$ je *c-univerzální* systém, 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:

Věta 4.1.1. *Předpokládejme, že $U = \{0 \,..\, N-1\}$, kde N je prvočíslo. Definujme*

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

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

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

Nechť (x, y) jsou libovolné, ale pevné. Zároveň platí $x \neq y$. 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 $m \cdot \lceil N/m \rceil^2$.

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

m je počet q, $\lceil \frac{N}{m} \rceil^2$ je počet q, s

$\Rightarrow$ tento univerzální systém lze zkonstruovat. Velikost H je N^2. □

4.1.1 Očekávaná délka řetězce

Definice 4.1.2. Mějme libovolnou pevnou $S \subseteq U$, libovolné pevné $x \in U$ a funkci $h : U \to \{0 \,..\, m-1\}$. Definujme

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

Zaveďme

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

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

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

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

Věta 4.1.2. *Když H je c-universální systém, pak $\forall S \subseteq U$ a $\forall x \in U$ je očekávaná hodnota*

$$\delta_h(x, S) = \begin{cases} \frac{c(|S|-1)}{m} & x \in S \\ \frac{c|S|}{m} & x \notin S \end{cases}$$

kde průměr se bere přes $h \in H$ a předpokládáme rovnoměrný výběr fcí z H.

Důkaz.

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

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

Věta 4.1.3. *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.*

Věta 4.1.4. *Markovova nerovnost: Nechť očekávaná hodnota X je nenulová. Pak platí $\mathcal{P}(X \geq t\mathrm{E}X) \leq 1/t$*

pokud by nebyl uveden předpoklad $X \neq 0$, věta bz neplatila

Poznámka 4.1.1. Jiná formulace Markovovy nerovnosti (věty) může být tato:
Když X je náhodná veličina s očekávanou hodnoutou μ, pak $P(X \geq t\mu) \leq \frac{1}{t}$.

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

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

a tedy

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

□

Varianta Markovovy nerovnosti:

Věta 4.1.5. *Za stejných předpokladů jako u věty 4.1.3, když μ je průměrná délka řetězce $S_{h,x}$, pak*

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

Důkaz. plyne z Markovovy nerovnosti.

□

4.1.2 Velikost *c*-univerzálního systému

Dolní mez

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

Věta 4.1.6. *Když H je c-univerzální systém funkcí z univerza U do $\{0 \,..\, m-1\}$, pak*

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

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

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

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

Platí

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

† 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 $h_1, \ldots, h_t$ 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 $\log_2 |H| = 2 \log_2 N$, což je hodně velké proti právě spočítanému dolnímu odhadu. Nyní zkonstruujeme c-univerzální hašovací systém, který tento dolní odhad v jistém smyslu nabývá.

Buď $p_1, p_2 \ldots$ 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

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

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

pak $|H| = t p_{2t}^2$, 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

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

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

Poznámka 4.1.2. $\forall (k,q,r) \exists! (c,d)$, které řeší rovnici, jelikož $\mathbb{Z}_{p_{2t}}$ je těleso a $x \bmod p_l \neq y \bmod p_l$.

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

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

Pokud tedy ještě dokážeme, že $|L| \le t/m$, pak $|\{h \in H : h_{c,d,l}(x) = h_{c,d,l}(y)\}| \le 4\frac{|H|}{m} + \frac{|H|}{m} = 5\frac{|H|}{m}$ 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 $P \le |x-y| \le N$. Protože $P \ge p_t^{|L|}$, dostaneme $|L| \le \ln N / \ln p_t$, a z definice t (4.4) plyne $|L| \le t/m$.

4.1.3 Reprezentace S a operace 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

MEMBER pracuje normálně.

INSERT

viz algoritmus 4.1

Algoritmus 4.1 INSERT pro universální hašování

---

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

viz algoritmus 4.2

Algoritmus 4.2 DELETE pro universální hašování

---

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

repeat
zvolíme náhodně $h \in H_i$
spočítáme reprezentaci S vůči h
until všechny řetězce mají délku $\leq d_i$

Kolikrát proběhne ten cyklus? Závisí to na více parametrech a není jisté, zda to bylo někdy přesně spočítáno.

4.2 Externí hašování

Máme dáno universum U a vnější paměťové medium, které je rozděleno do stránek. Každá stránka může obsahovat nejvýše b záznamů. Chceme navrhnout ukládání prvků do stránek tak, aby operace MEMBER, INSERT, DELETE vyžadovaly jen konstatní počet přístupů do externí paměti, aby stránky byly dostatečně zaplněné. Tedy jinými slovy: chceme minimalizovat počet přístupů do vnější paměti.

Vyhledáme vždy stránku ve vnější paměti - natáhneme ji do vnitřní paměti a tam provedeme vyhledávání - toto vyhledávání nás ale nezajímá.

Nechť U je universum, $S \subseteq U$. Externí medium má stránky o velikosti b. Je dána hašovací funkce $h : U \to \{0,1\}^k$ prostá (kódovací funkce), k libovolné. Funkce h tedy generuje binární kódy délky k.

Vytvoříme strom množiny S. Vrcholy stromu jsou všechny prefixy slov $h(s), s \in S$. Pro vrchol α jsou jeho synové $\alpha 0$ a $\alpha 1$ (pokud existují). Listy jsou prvky $h(s), s \in S$.

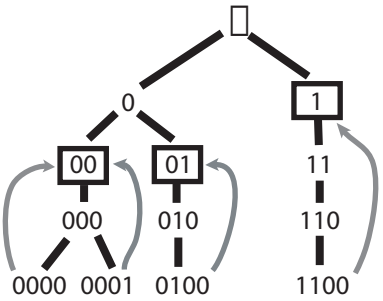
Definice 4.2.1. Pro $s \in S$ definujeme *d(s)* = délka nejkratšího prefixu α slova $h(s)$, že pod α je nejvýše b listů.

Poznámka 4.2.1. Alternativní definice $d(s)$: $d(s)$ = délka nejkratšího prefixu $h(s)$ taková, že počet prvků $t \in S$, jejichž prefix délky $d(s)$ je stejný jako prefix $h(s)$, je nejvýše b.

Definice 4.2.2. $d(S) = max\{d(s), s \in S\}$.

Příklad 4.2.1. Nechť $U = \{0,1\}^4, b = 2$ a $S = \{0100, 0001, 0000, 1001\}$

Tuto strukturu můžeme zobrazit do stromu, kde bude lépe viditelná hodnota $d(S)$.



Obrázek 4.1: Reprezentace množiny S. Vrchol reprezentující prefix 0, by ve svém podstromě měl 3 prvky, což je více, než požadujeme pomocí hodnoty b.

Z obr. 4.1 plyne, že hodnota $d(S) = 2$.

Tvrzení 4.2.1. *Platí: Když $d(s) = k$ a prvek t má stejný prefix délky k jako s, pak $d(s) = d(t)$.*

Reprezentace:

- adresář: ke každému slovu α délky $d(S)$ je přiřazena adresa stránky, která obsahuje prvky $s \in S$, že $h(s)$ má prefix α. Tato slova délky $d(s)$ jsou lexikograficky uspořádána.
- datová část: stránky s přiřazenými prvky

Příklad 4.2.2. Příklad adresáře pro množinu $\{0000, 0001, 0100, 1001\}$:

00	→	0000, 0001
01	→	0100
10	↘	
11	→	1001

Poznámka 4.2.2. Upřesnění reprezentace stránky (stránek):
Prvek $s \in S$ je uložen na stránce, která obsahuje všechny prvky $t \in S$ takové, že prefix $h(t)$ délky $d(s)$ bude je stejný jako $h(s)$, tato stánka bude přiřazena všem slovům α délky $d(S)$ takovým, že prefix $h(s)$ délky $d(s)$ je prefix α.
Pokud α neobsahuje žádný takový prefix, tak je mu přiřazena stránka NIL.

4.2.1 Operace ACCESS

viz algoritmus 4.3

Algoritmus 4.3 ACCESS pro externí hašování

ACCESS(x)

1. Spočítáme $h(x)$, natáhneme adresář, nalezneme $d(S)$ a najdeme stránky odpovídající prefixu $h(x)$ délky $d(S)$

2. Natáhneme odpovídající stránku do paměti, zjistíme, zda obsahuje x a když ano, provedeme požadované úpravy.

3. Stránku uložíme zpět na stejné místo.

Operace ACCESS vyžaduje 3 přístupy na externí medium. (za předpokladu, že adresář je také uložen na externím mediu a zabírá jednu stránku)

Pro rychlou implementaci aktualizačních operací je vhodné mít u každé stránky uloženo informaci kolik je prvků na stránce.

4.2.2 Operace INSERT

viz algoritmus 4.4

Algoritmus 4.4 INSERT pro externí hašování

INSERT(x)

1. Spočítáme $h(x)$, natáhneme adresář, nalezneme $d(S)$ a nalezneme adresu stránky odpovídající prefixu $h(x)$ délky $d(S)$. (XXX odkud vezmu $d(S)$?)

2. Natáhneme do paměti odpovídající stránku. když v ní existuje $x \rightarrow$ konec

3. Když neobsahuje x a má méně prvků než b, vložíme x do této stránky a uložíme ji zpátky na stejné místo a aktualizujeme adresář (počty prvků na stránce)

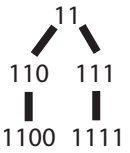
4. Když stránka prvek x neobsahuje a má b prvků, stránku rozdělíme (nalezneme nové $d(s)$ pro s z této stránky i s přidaným x), stránky uložíme a aktualizujeme adresář.

Operace INSERT vyžaduje 6 přístupů na externí medium.

pokud je nově nalezené $d(s) \leq d(S)$, adresář se zvětšovat nebude

Poznámka 4.2.3. Rozštěpení stránky nemusí nutně znamenat zvětšení adresáře.

Příklad 4.2.3. Do množiny z příkladu 4.2.1 vložíme pomocí operace INSERT prvek 1111. Hodnota $d(1100) = 1$ se po vložení prvku 1111 nezmění. Podstrom reprezentující prvky množiny S s prefixem 11 bude mít po vložení prvku 1111 dva syny. (viz obr. 4.2)



Obrázek 4.2: Podstrom reprezentující prvky množiny S s prefixem 11 po vložení prvku 1111

Pokud vezmeme situaci danou po vložení prvku 1111, bude adresář vypadat takto:

00	→	0000, 0001
01	→	0100
10	↘	
		1001
11	↗	

Nyní vložíme prvek 0010. V tomto případě dojde ke zvětšení adresáře:

000	→	0000, 0001
001	→	0010
010	→	0100
011	↗	
100	↘	
101	→	1001, 1111
110	↗	
111	↗	

Hodnota $d(S)$ je nyní 3.

4.2.3 Operace DELETE

viz algoritmus 4.5

Algoritmus 4.5 DELETE pro externí hašování

DELETE(x)

1. spočítáme $h(x)$, natáhneme adresář, nalezneme $d(S)$, nalezneme adresu stránky odpovídající prefixu $h(x)$ délky $d(S)$, zjistíme zda po odebrání prvku x může nastat spojování stránek a pokud ano, nalezneme adresu stránky, která se spojí s naší stránkou

2. natáhneme odpovídající stránku do paměti, zjistíme zda obsahuje x, pokud ne, tak konec

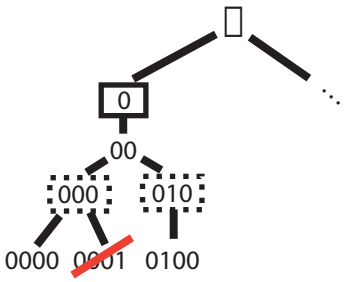
3. když obsahuje x a nemůže dojít ke sloučení stránek, tak odstraníme x, stránku uložíme na stejné místo a aktualizujeme adresář (počty prvků na stránce)

4. když obsahuje a dojde ke slučování, pak odstraníme x, natáhneme druhou stránku a stránky sloučíme, uložíme novou stránku a aktualizujeme adresář.

pro aktualizaci adresáře to mohou být 2 operace - načtení a uložení. zřejmě proto, aby mohlo být současně puštěno víc operací.

Příklad 4.2.4. Pro situaci z příkladu 4.2.3 provedeme DELETE(0100). V adresáři budou pak položky 010 a 011 ukazovat na prázdnou stránku (NIL). Adresář zůstal po této operaci stejný.

Nyní smažeme prvek 0001.



Obrázek 4.3: Část stromu reprezentujícího množinu S před smazáním prvku 0001

Na obr. 4.3 je vidět, že po smazání prvku 0001 může být adresář opět zmenšen.
Po provedení DELETE(0001) bude adresář vypadat takto:

000	→	0000, 0010
001	↗	
010	→	NIL
011	↗	
100	↘	
101	→	1001, 1111
110	↗	
111	↗	

Nyní můžeme provést zmenšení adresáře na:

00	→	0000, 0010
01	↗	
10	→	1100, 1111
11	↗	

Tento adresář můžeme ještě zmenšit:

0	→	0000, 0010
1	→	1100, 1111

Poznámka 4.2.4. Pesimistický odhad pro operaci DELETE je nejvýše 6 přístupů na externí medium.

Aktualizace adresáře

V posledním kroku DELETE se provádí aktualizace adresáře. Nejprve provedeme opravení odkazů na stránky: pokud u sudého záznamu v adresáři dojde k vyprázdnění stránky, bude tam odkaz na NIL. pokud dojde k vyprázdnění u liché stránky, přehodí se odkaz na předchozí (sudou) stránku.

Potom se testuje, zda jde adresář zmenšit. Zmenšování provádíme tak dlouho, dokud to jde.

4.2.4 Reprezentace adresáře

- reprezentovat jako posloupnost dvojic (adresa stránky, počet prvků na stránce), kde i-tá dvojice je přiřazení i-tému slovu v lexikografickém uspořádání.
- zvětšování adresáře znamená zdvojení prvků
- test na zmenšování adresáře - testujeme, zda adresa na lichém místě je stejná jako adresa na následujícím sudém místě - pokud ano, tak zmenším adresář vymazáním sudých členů posloupnosti.

Očekávané zaplnění stránky při rovnoměrném rozložení dat je $b \ln 2 \approx 0.69b$
Očekávaná velikost adresáře je $\frac{e}{b \ln 2} n^{1+\frac{1}{b}}$ (při rovnoměrném rozložení dat)

Poznámka 4.2.5. Člen $n^{1+\frac{1}{b}}$ není lineární, zde je problém, proto se to nehodí pro malá b.

b/n	10^5	10^6	10^8	10^{10}
2	$6.2 * 10^7$	$1.96 * 10^8$	$1.96 * 10^{11}$	$1.96 * 10^{14}$
10	$1.2 * 10^5$	$1.5 * 10^6$	$2.4 * 10^8$	$3.9 * 10^{10}$
50	$9.8 * 10^3$	$1.0 * 10^6$	$1.1 * 10^8$	$1.2 * 10^{10}$
100	$4.4 * 10^3$	$4.5 * 10^4$	$4.7 * 10^6$	$4.9 * 10^8$

Jak je vidět, pro $b = 2$ se to nehodí, adresář je větší než velikost dat.

Poznámka 4.2.6. Když pracujeme s velikostí S kolem 10^{10}, pak je vhodné, aby $b \geq 100$. Pro větší množiny S musí být b větší.

4.3 Perfektní hašování

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

Definice 4.3.1. Funkce $h : U \to \{0 \,..\, m-1\}$ je perfektní pro S, když je na S prostá ($\forall x \neq y \in S$ platí $h(x) \neq h(y)$).

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

Požadavky na hledanou hašovací funkci:

1. h je perfektní na S

2. $\forall x$ je $h(x)$ rychle 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.

4.3.1 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:

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

tedy

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

a dosazením do (4.10), podle přihrádkového principu

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

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 perfektnost h_k v čase $O(n)$.

□

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

4.3.2 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

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

Ovšem na zakódování této funkce potřebujeme hodně paměti: 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 $O(N)$

lepší jména proměnných!

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

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

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

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

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

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 4.3.2. *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 4.3.1. 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 4.3.2 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 $O(n^2 \log N)$.

4.3.3 GPERF

Jiná konstrukce perfektní hašovací funkce je použita v programu `gperf`. Distribuován pod GPL. Jeho návrh je popsán v [3].

Kapitola 5

Trie

Trie[1] je rovinná implementace slovníku.

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

5.1 Základní varianta

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

Definice 5.1.2. Řekneme, že trie nad Σ *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.
- (*prefixová podmínka*) Když v je vnitřní vrchol trie odpovídající slovu α, pak existuje $\beta \in S$ takové, že α je prefix β.
- Pro každé slovo $\alpha \in S$ existuje v trie list odpovídající α.

Poznámka 5.1.1. Na rozdíl od binárního vyhledávacího stromu (viz kapitola 7, sekce 7.1), žádný vrchol ve stromě neobsahuje uložený klíč, který reprezentuje. Namísto toho jeho pozice ve stromě udává klíč, který reprezentuje[2]. Pouze některé vrcholy ve stromě obsahují data - např. pro implementaci slovníku s hesly by data uložená v listech obsahovala popis tohoho hesla[3].

Poznámka 5.1.2. Proč jsou trie výhodné ?

- Vyhledávání klíčů je rychlejší než v BVS. Vyhledání klíče délky m vyžaduje pouze $O(m)$ času. Pro BVS je to $O(m^2)$ v nejhorším případě, protože se musí opakovaně porovnávat počáteční znaky hledaného slova. Další výhoda je použití indexace pomocí znaků v operaci MEMBER.

[1] Název *trie* pochází z anglického "re**trie**val", tedy vyzvednutí. Názory na to, jak vyslovovat "trie" se různí. V češtině se zpravidla vyslovuje tak jak se píše.

[2] Tato a následující poznámka jsou volně převzaty z encyklopedie Wikipedia, heslo Trie.

[3] Např. slovník spisovatelů, kde listy ve trie by odpovídaly jménům jednotlivých spisovatelů a data uložená v nich by obsahovala seznam jejich děl.

- Trie zabírají méně místa. Protože nejsou klíče v trie uloženy explicitně, pro uložení jednoho klíče je potřeba pouze amortizovaný konstantní prostor.
- Pomocí trie lze jednoduše provádět operaci hledání nejdelšího prefixu[4], kde potřebujeme najít klíč, který má nejdelší shodný prefix s hledaným klíčem[5]. Dále trie dovolují asociovat jednu hodnotu s množinou klíčů, které mají shodný prefix[6].

5.1.1 Algoritmus MEMBER

viz algoritmus 5.1.

Algoritmus 5.1 MEMBER pro základní verzi trie

```
{vyhledání x = x_1 ... x_l}
t := kořen
i := 1
while t není list do
  t := t[x_i] // sestup podle znaku x_i
  i := i + 1
end while
{test}
return Nal(t)
```

Na tomto algoritmu je zajímavé to, že používá jednotlivé znaky hledaného slova x k indexaci v jednotlivých vrcholech trie (viz řádek s komentářem ve výpisu algoritmu 5.1.). To dovoluje najít vrchol do kterého se má při hledání sestoupit v čase $O(1)$. Tedy složitost operace MEMBER je $O(l)$, protože musíme projít nejvýše l vrcholů než dosáhneme listu (délka slov je nejvýše l).

5.1.2 Algoritmus INSERT

viz algoritmus 5.2.

Algoritmus 5.2 INSERT pro základní verzi trie

```
{vyhledej x pomocí operace MEMBER(x)}
if not Nal(t) then {trie nemusí být tak hluboké, jak potřebujeme}
  while i ≤ l do
    vrcholu t přidej k listů ohodnocených písmeny z Σ, jejich Nal := false
    t := t[x_i]
    i := i + 1
  end while
  Nal(t) := true
end if
```

[4] anglicky "longest-prefix matching"

[5] To se hodí například pro implementace síťových operačních systémů, kde je potřeba provádět tuto operaci pro hledání v routovacích tabulkách nebo tabulkách pro překlad adres. V případě směrovacích tabulek se posílá paket na další "hop" podle cílové adresy. Routovací tabulka obsahuje záznamy, které udávají adresu sítě a adresu zařízení, na které posílat pakety pro tuto síť - tzv. "hop". Tento "hop" se vybírá tak, aby cílová adresa paketu měla co možná nejdelší shodný prefix s nějakou adresou sítě v routovací tabulce.

[6] Tím, že uložíme data do vnitřních uzlů trie.

Při operaci INSERT se sestupuje až na úroveň délky slova, přičemž se přidávají nové úrovně v případě že nejsou v trie přítomny[7].

5.1.3 Algoritmus DELETE

viz algoritmus 5.3.

Algoritmus 5.3 DELETE pro základní verzi trie

```
{vyhledej x pomocí operace MEMBER(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
```

Analogicky k operaci INSERT dochází k rušení hladin v rámci jedné větve v případě že všechny listy v hladině mají hodnotu Nal = *false*.

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 v nejhorším případě je počet uložených slov násobený délkou cesty a počtem synů, tedy $O(|S|lk)$.

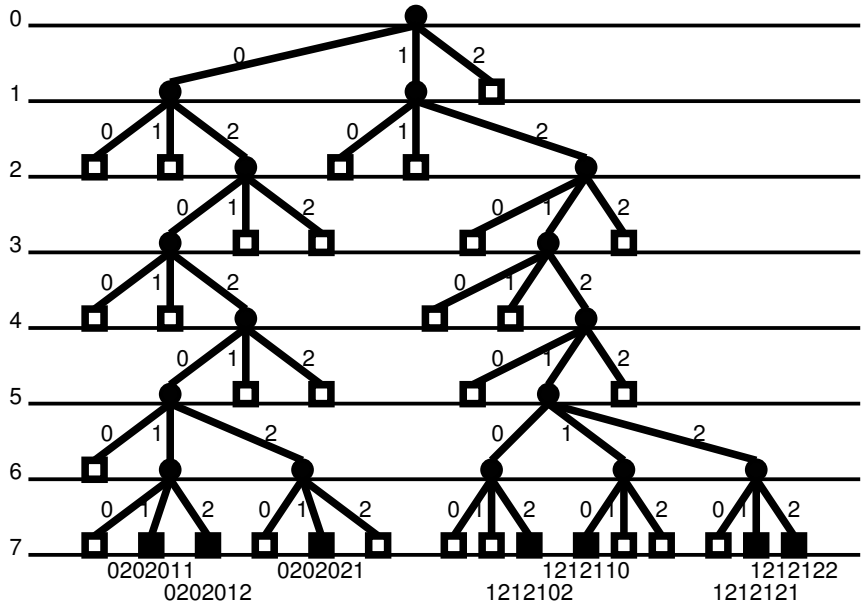
Poznámka 5.1.3. V případě, kdy S obsahuje (skoro) všechna slova délky l, tak může mít složitost jen $O(|S|)$.

5.2 Komprimované trie

Mějme $\Sigma = \{0, 1, 2\}$, $l = 7$. $S = \{0202011, 0202012, 0202021, 1212102, 1212111, 1212121, 1212122\}$. Nekomprimované trie pro tuto množinu je na obrázku 5.1. Vidíme, že písmena na druhé až páté pozici jsou vždy stejná a předchozí algoritmy se jimi musí prokousat. Přesněji řečeno, prohlížení vrcholu v, který má jediného syna, který není list s hodnotou 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 (komprimovat) 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.

[7] Celkem hezky si lze proces přidávání nových hladin v rámci jedné větve představit tak, že v každé hladině, která je nově přidaná, "vyroste smeták" s k vrcholy.



Obrázek 5.1: Nekomprimované trie. Černě vyplněné čtverce znázoňují listy, které odpovíadjí nějakému slovu z (reprezentované) množiny S. Tyto listy mají hodnotu funkce Nal *true*. Bíle vyplněné čtverce žádnému slovu ze S neodpovídají, mají tedy hodnotu Nal *false*.

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 Σ.

Příklad 5.2.1. Nechť $\Sigma = \{0, 1, 2\}$, $l = 3$. $S = \{001, 102, 010, 211, 212\}$.

Nekomprimovaný trie pro množinu S a jeho komprimovaná varianta je na obr. 5.2.

Koubek 2002/2003

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

Když $\beta \in$ trie a $a \in \Sigma \rightarrow \begin{cases} \beta a \text{ list}, & \text{je } a\text{-tým synem } \beta \\ \exists \delta \in S, & \text{že } \beta a \text{ je prefixem } \delta \end{cases}$

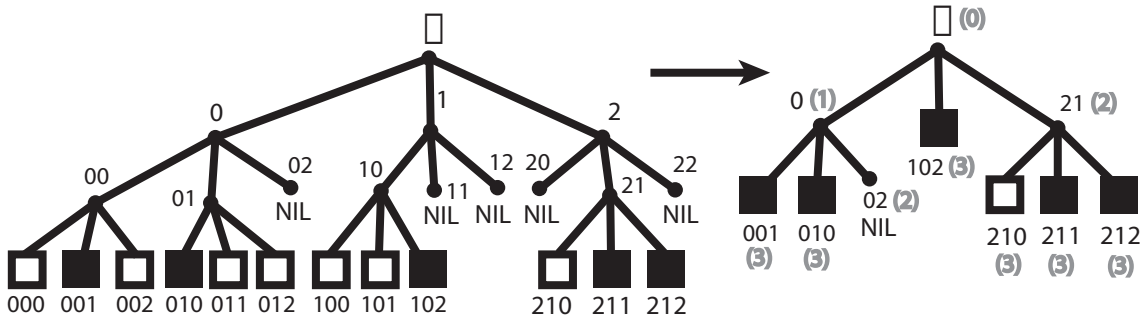
Potom a-tý syn β je nejkratší prefix v množině trie v S, který obsahuje βa.

5.2.1 MEMBER

Viz algoritmus 5.4

5.2.2 INSERT

Viz algoritmus 5.5



Obrázek 5.2: Komprimace trie. Šedá čísla v závorce značí hodnotu *uroven*()

Algoritmus 5.4 MEMBER pro komprimované trie

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

5.2.3 DELETE

Viz algoritmus 5.6

5.2.4 Časová a paměťová složitost

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

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

Nechť

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

Očekávaná hloubka trie reprezentující n slov je

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

Když funkce pref_{d-1}, přiřazující slovu α 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 pref_{d-1} prostá. Tyto množiny získáme tak, že vybereme n prefixů délky $d-1$ a každý

Algoritmus 5.5 INSERT pro komprimované trie

```
{vyhledej x}
if Nal(t) ∧ 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.}
    α := nejdelší společný prefix slov x a slovo(t). Délku α označme |α|.
    v := vrchol na cestě z kořene do t takový, že uroven(v) je největší, která je ≤ |α|
    if uroven(v) = |α| then
      {v je otec nového listu}
    else {uroven(v) < |α|}
      {Bude potřeba vytvořit otce nového listu}
      a := uroven(v) + 1-ní písmeno α
      u := v[a]
      {Mezi v a u vytvoř nový vnitřní vrchol odpovídající slovu α}
      w := nový vrchol, uroven(w) := |α|
      v[a] := w
      c := |α| + 1-ní písmeno slovo(t)
      w[c] := u
      for all b ∈ Σ, b ≠ c do
        z := nový vrchol, uroven(z) := |α| + 1, Nal(z) := false, slovo(z) := αb,
        w[b] := z
      end for
      v := w
    end if
    {Správnému listu přiřaď x}
    d := |α| + 1-ní písmeno x
    s := v[d]
    uroven(s) := l, Nal(s) := true, slovo(s) := x
  end if
end if
```

Algoritmus 5.6 DELETE pro komprimované trie

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

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

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

poněvadž

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

(užijte integrální kriterium a substituci $x = k^{d-1}(1-t)$) a $e^x - 1 \ge x$ (odtud $1 - e^x \le -x$). Tedy pro $c = 2\lceil \log_k n \rceil$ dostáváme

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

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

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

Algoritmus 5.7 INSERT pro komprimované trie, analogie 5.5 (verze Koubek 2002)

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

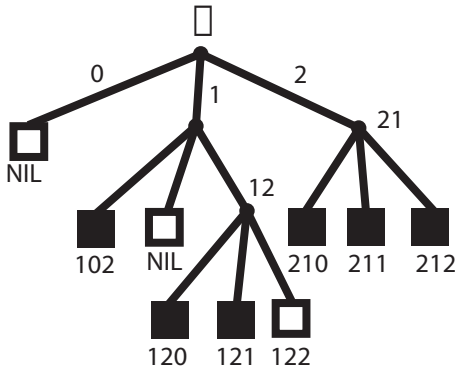
XXX dalsi neznamy algoritmus z prednasky 2002

5.3 Ještě komprimovanější trie

Příklad 5.3.1. Mějme komprimovaný trie z obr. 5.3 a jeho matici:

Algoritmus 5.8 DELETE pro komprimované trie (?)

DELETE($x = x_1, ..., x_l$)
  t ← kořen
  while t neni list **do**
    i ← hladina(t), t ← (a_{i+1}-ni syn t
  end while
  if Nal(t) = true a prvek(t) = j **then**
    Nal(t) = false
    v ← otec(t)
    prvek(t) ← prefix prvek(t) o délce hladina v+1
    if vsichni synove vrchovlu v az na jednoho jsou listy s Nal = false **then**
      w ← syn(v), ktery je bud list s Nal(w) = true nebo neni list
      necht v je a-tý (a_i-ty ???) syn svého otce, v smažeme a smažeme
      všechny syny $v \neq w$
      w ← a-tý (a_i-tý ???) syn otce v
    end if
  end if



Obrázek 5.3: Nekomprimovaný trie pro příklad 5.3.1

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

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

co znamena *rd* ?

5.3.1 Popis *A* a *rd*

Zpět k našemu příkladu:

1.

hod	210	211	212	120	121	NIL

	root	a	b	c
rd			0	3

2.

hod	210	211	212	120	121	a	b	102	NIL	c

	root	a	b	c
rd	4	7	0	3

Řádek i začíná na místě $rd(i)$ a musí být splněna podmínka:
Když $M_{i,j} \neq NIL \neq M_{i',j'}$, pak $rd(i) + j \neq rd(i') + j'$
Když na místě hod chceme zapsat prvek $\neq NIL$ a NIL, pak zapíšeme prvek $\neq NIL$.

5.3.2 Algoritmus pro hledání rd a hod

Nechť M je matice typu rxs, má m významných míst $\neq$ NIL.

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

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

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

Důkaz. Předpokládejme, že hledáme rd pro řádek i, který má l míst $\neq NIL$.
ve vektoru hod je obsazeno méně než $m(l-1)$ míst.
zkoušíme $rd(i) = 1, 2, ...$
$rd(i) = 1, 2, ...$ je zakázané, když vznikne kolize.
tj. $\exists$ řádek i' předcházející a $\exists j, j'$ takové, že $M_{i',j'} \neq NIL \neq M_{i,j}$ a platilo by $rd(i')+j' = rd(i)+j$.
$\rightarrow$ těchto možností je $< lm(l-1) \leq m$.
$O(rs)$ - zjistíme pro každý řádek počet míst $\neq NIL$.
$O(m + r)$ - třídění Bucketsortem
$O(mrs)$ - krok 2 □

Příklad 5.3.2. XXX nejaky komentar

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

	root	a	b	c
rd	4	7	0	3

hod	210	211	212	120	121	a	b	102	NIL	c

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

(přehodili jsme pouze řádky)
sl - vektor posunutých sloupců

- $sl(0) = 0$
- $sl(1) = 1$

M'	0	1	2
1	210	NIL	NIL
2	120	211	212
3	NIL	121	NIL
4	102	a	b
5	NIL	NIL	c

$zac = (6, 0, 6, 3, 6)$
$hod = (120, 211, 212, 102, a, b, 210, 121, c)$
Když $M(i, j)$ je významné místo, pak $M(i, j) = hod(zac(i + sl(j)) + j)$.

5.3.3 Vertikální posun sloupců

cd - vektor sloupcového posunutí, slouží k zápisu transformace

	0	1	2
cd	0	1	2

	0	1	2	3	4
rd	6	0	6	3	6

hod	120	211	212	102	a	b	210	121	c

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

- f(-,-) je fce dvou proměnných
- B_j matice posunutých prvních sloupců
- m_j počet míst $\neq NIL$ v B_j
- $m_j(l)$ počet míst $\neq NIL$ v řádcích matice B_j, které mají aspoň l+1 míst $\neq NIL$

Budeme chtít, aby $\forall j \forall l$ platilo $m_j(l) \leq \frac{m}{f(l,m_j)}$.
Okrajové podmínky na f: f musí splňovat:

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

Algoritmus na posunutí sloupců

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

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

Platí $m(l) = m_s(l) \leq \frac{m}{f(l,m)} \leq \frac{m}{l+1}$.
Hledáme hodnotu $cd(j)$ a předpokládáme, že pro nějakou hodnotu $cd(j)$ není splněna podmínka pro l, tj. platí $m_j(l) > \frac{m}{f(l,m)}$ ... platila pro $B_{j-1}, tj. m_{j-1}(l) \leq \frac{m}{f(l,m_{j-1})}$
Z toho plyne $m_j(l) - m_{j-1}(l) > \frac{m}{f(l,m_j)} - \frac{m}{f(l,m_{j-1}}$.
Jak roste číslo $m_j(l)$?

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

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

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

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

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

Podmínka pro l může zakázat nejvýše

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

Stačí nám sčítat přes hodnoty l takové, že
$mm_{j-1}(l) \leq l+1$ tj. přes $l \leq l_0 = min\{l; \frac{m}{f(l,m_{j-1}} < l\}$,
$m_{j-1}(l) \leq \frac{m}{f(l,m_{j-1})} \leq l+1$.
Celkový počet zakázaných hodnot cd je menší než

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

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

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

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

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

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

Dalším pozorovaním zjistíme, že v takto získaných rozdílech se mění jenom jedna proměnná. Výraz se dále zjednodušší, bude-li $g(l, m_j) = h(l)k(m_j), kdeh(l), k(m_j)$ budou vhodné lineární funkce.
U funkce k linearitou využijeme rozdílu $m_{j-1} - m_j$ v čitateli, který teď můžeme zkrátit.

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

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

Takto definovaná f splňuje okrajové podmínky:
$f(l, m) = 2^l \geq l + 1 \; \forall l = 0, 1, ...$
$f(0, m_j) = 1 \leq \frac{m}{m_j} \; \forall j = 0, 1, ..., s$
dosadíme do odhadu 5.2 a dostaneme

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

využijeme, že $2^x - 1 \geq xln(2)$

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

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

integrální kriterium

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

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

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

Celý algoritmus spočítá uložení matice M typu $r \times s$ do vektorů
cd - dimenze s,
rd - dimenze $4m \log\log m) + 15.3m + r$,
hod dimenze $m + s$,
přitom hodnoty $cd(j) < 4m \log\log m) + 15.3m$ a $rd(i) < m$.

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

5.3.4 Úsporné uložení řídkého vektoru

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

Vytvoříme vektor cv dimenze t, $cv(j) = v(i_j)$.
Náš úkol - pro dané l zjistit, zda $l = i_j$ a případně nalézt toto j.
Sestavíme vektor $base$ dimenze n:

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

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

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

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

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

1. $v(h) = 0 \leftrightarrow offset(h\texttt{div}d, h \bmod d) = -1$

2. $v(h) = 1 \rightarrow h = base(h\texttt{div}d) + offset(h\texttt{div}d, h \bmod d)$

3. $offset(i, j) = off(i)\texttt{div}(d + 1)^j \bmod (d + 1) - 1$

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

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

Příklad 5.3.3. XXX uvod k prikladu

$v =$

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

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

$cv =$

v(1)	v(3)	v(5)	v(11)

$base =$

0	1	-1	3

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

3. sloupec tabulky offset repr. nuly
off =

4	33	0	16

Potom $off(7) = (offset(1,0)+1)4^0 + (offset(1,1)+1)4^1 + (offset(1,2)+1)4^2$ $off(1) = 1 + 0 + 2 \cdot 4^2 = 33$

Kapitola 6

Uspořádaná pole

6.1 Unární, binární a interpolační vyhledávání

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

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

[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) ...

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

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

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

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

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

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

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

Nyní použijeme Čebyševovu nerovnost, která říká, že

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

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

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

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

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

Kapitola 7

Binární vyhledávací stromy

7.1 Obecně

Definice 7.1.1. *Binární vyhledávací strom* reprezentující množinu S je takový binární strom, že

1. každý vnitřní vrchol má dva syny, levého a pravého

2. existuje jednoznačná korespondence mezi vrcholy S a vnitřními vrcholy stromu

3. pro každý vnitřní vrchol v platí, že vnitřní vrcholy v podstromu jeho levého syna reprezentují prvky menší než reprezentuje v a vnitřní vrcholy v podstromu jeho pravého syna reprezentují prvky větší než reprezentuje vrchol v.

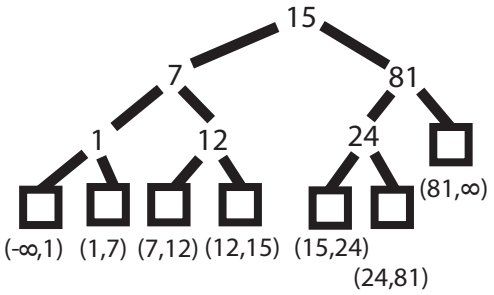
Poznámka 7.1.1. Nechť

$$S = \{s_1 < s_2 < ... < s_n\}$$
$$s_0 = -\infty, s_{n+1} = \infty$$

Pak i-tý list (ve smyslu zleva doprava) reprezentuje interval $< s_{i-1}, s_i >$.

Příklad 7.1.1. Nechť $S = \{1, 7, 12, 15, 24, 81\}$. Binární vyhledávací strom reprezentující množinu S je na obr. 7.1.

Strom na obr. 7.1 není možná nejlepší příklad, protože BVS mohou vypadat více nepravidelně - na rozdíl od haldy *nemusí* mít všechny uzly umístěny co možná nejvíce "vlevo".



Obrázek 7.1: Příklad binárního vyhledávacího stromu

Poznámka 7.1.2. V listech máme jen intervaly, každý vrchol implicitně určuje tyto intervaly. (viz obr. 7.1) Tato vazba je jednotlivými operacemi udržována.

7.1.1 Algoritmus MEMBER

Algoritmus 7.1 MEMBER pro binární vyhledávací stromy

```
t ← kořen
while t není list a zároveň t nereprezentuje x do
  if x > prvek reprezentovaný t then
    t ← pravý syn
  else
    t ← levý syn t
  end if
end while
```

7.1.2 Algoritmus INSERT

tohle je pouze pokus o alg. XXX dopsat podle nějakého důvěryhodného zdroje !

1. najdi list kam patří x ($x_{i-1} < x < x_i$)

2. udělej z něj vnitřní vrchol s hodnotou x

3. přidej mu dva syny s intervaly $< x_{i-1}, x >, < x, x_i >$

4. uprav strukturální podmínku na strom

7.1.3 Algoritmus DELETE

DELETE(x) provedeme následovně:

1. nalezneme x (vrchol reprezentující x)

2. pokud jeden jeho syn je list, pak odstraníme vrchol + syna, který je list, druhý syn nahradí odstraněný vrchol

co když oba synové jsou listy ?

3. pokud žádný jeho syn není list, pak: nalezneme vrchol u, reprezentující nejmenší prvek v S větší než x. levý syn tohoto vrcholu je list. přemístíme prvek reprezentovaný tímto vrcholem do vrcholu reprezentujícího x odstraníme u a jeho levého syna, pravého syna u dáme na místo u.

Jak nalezneme vrchol reprezentující nejmenší prvek v S větší než x ? Jsme ve vrcholu t reprezentujícím x a hledáme vrchol u:

```
u ← pravý syn t
while levý syn u ≠ list do
  u ← levý syn u
end while
```

Čas pro operace MEMBER, INSERT a DELETE v binárním vyhledávacím stromě je O(délka stromu) = O(výška stromu).

7.2 Optimální binární vyhledávací stromy

Budeme chtít reprezentaci binárního vyhledávacího stromu takovou, že bude optimalizovaná vzhledem k operaci MEMBER za předpokladu, že známe pravděpodobnost provedení této operace na jednotlivé vrcholy stromu.

7.2.1 Co je to optimální binární vyhledávací strom

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

7.2.2 Algoritmus konstrukce

Dána množina $S = \{x_1 < x_2 < ... < x_n\}$, provádí se pouze operace MEMBER(x) a jsou dány pravděpodobnosti $\alpha_1, ..., \alpha_n, \beta_0, ..., \beta_n$, kde

P značí pravděpodobnost

- $\alpha_i = P$(prováděla se operace $MEMBER(x_i)$)
- $\beta_i = P$(prováděla se operace $MEMBER(x)$ pro $x \in < x_i, x_{i+1} >$)

kde $x_0 = -\infty, x_{n+1} = \infty$.

Tedy jsou dány pravděpodobnosti přístupu k vnitřním vrcholům a k listům.

Chceme nalézt binární vyhledávací strom reprezentující S takový, že operace MEMBER má nejmenší očekávaný čas.

Očekávaný čas operace MEMBER je $\sum_{i=1}^{n} \alpha_i(a_i + 1) + \sum_{i=0}^{n} \beta_i b_i$, kde a_i je hloubka prvku reprezentujícího prvek x_i, b_i je hloubka listu reprezentujícího interval $< x_i, x_{i+1} >$.

očekávaná hodnota = střední hodnota náhodné veličiny; n.v. s diskrétním rozdělením má stř. hodnotu $\sum x_i p_i$

Zobecníme úlohu:

Dána množina $S = \{x_1 < x_2 < ... < x_n\}$ a ohodnocení α_i prvku x_i a β_i intervalu (x_i, x_{i+1}). Chceme zkonstruovat binární vyhledávací strom T reprezentující S takový, že $hod(T) = \sum_{i=1}^{n} \alpha_i(a_i + 1) + \sum_{i=0}^{n} \beta_i b_i$ je minimální.
T je pak optimální binární vyhledávací strom. Pro $1 \le i \le j \le n$, $U_{i,j}$ je úloha nalézt optimální binární vyhledávací strom pro $S_{i,j} = x_i < x_{i+1} < ... < x_j$ a hodnoty $\alpha_i, \alpha_{i+1}, ..., \alpha_j, \beta_{i-1}, \beta_i, ..., \beta_j$.

XXX sjednotit indexy a závorky - buď $c(i,j)$ nebo $c_{i,j}$

Pozorování 7.2.1. Nechť T je strom reprezentující množinu $S_{i,j}$ a kořen T hodnotí prvek x_k pro $i \le k \le j$. Nechť T_l je podstrom levého syna kořene, T_p je podstrom pravého syna kořene. Pak:
$hod(T) = hot(T_l) + hod(T_p) + \sum_{l=i}^{j} \alpha_l + \sum_{l=i-1}^{j} \beta_l$
T_l reprezentuje množinu $S_{i,k-1}$ a T_p reprezentuje množinu $S_{k+1,j}$.

Pozorování 7.2.2. Nechť platí předpoklady pozorování 7.2.1 a nechť T je optimální binární vyhledávací strom pro $S_{i,j}$. Pak T_l je optimální binární vyhledávací strom pro $S_{i,k-1}$ a T_p je optimální binární vyhledávací strom pro $S_{k+1,j}$.

Pozorování 7.2.3. Když známe $hod(T_{k,k'})$ pro opt. bin. vyhl. strom reprezentující množinu $S_{k,k'}$ kde $i \le k \le k' \le j$ a $k' - k < j - i$, pak $hod(T_{i,j})$ pro opt. bin. vyhl. strom je

$$\sum_{l=i}^{j} \alpha_l + \sum_{l=i-1}^{j} \beta_l + min\{hod(T_{i,k-1}) + hod(T_{k+1,j}), i \le k \le j\}$$

Když

$$hod(T_{i,k-1}) + hod(T_{k+1,j}) \le min\{hod(T_{i,k'-1}) + hod(T_{k'+1,j}), i \le k' \le j\}$$

pak $\exists$ opt. bin. vyhl. strom pro $S_{i,j}$, jehož kořen reprezentuje x_k.

Systematicky spočítáme $w_{i,j} = \sum_{l=i}^{j} \alpha_l + \sum_{l=i-1}^{j} \beta_l$ pro všechny $1 \le i \le j \le n$. Inicializujeme matice H,K typu $nxn, H = K = 0$.

XXX typu n krat n

for i=1,2,...,n **do**

$H_{i,i} = w_{i,i}, K_{i,i} = i$
end for

$H_{i,j}$ pro $i \leq j$ bude $H_{i,j} = hod(T_{i,j})$ pro opt. bin. vyhl. strom reprezentující $S_{i,j}$ a $K_{i,j}$ bude index prvku reprezentovaného v kořeni $T_{i,j}$.

Algoritmus 7.2 Výpočet matic H, K

for every $i = 1, 2, .., n$ **do**
$H_{i,i} = w_{i,i}, K_{i,i} = i$
end for
$j = 1$
while $j < n$ **do**
$i = 1$
while $i + j \leq n$ **do**
$m = i, hod = H_{i+1,j}$ (1)
$k = i + 1$ (2)
while $k \leq i + j$ (3) **do**
if $hod > H_{i,k-1} + H_{k+1,i+j}$ **then**
$hod = H_{i,k-1} + H_{k+1,i+j}$
$m = k$
end if
$k = k + 1$
end while
$H_{i,i+j} = hod + w_{i,i+j}, K_{i,i+j} = m, i = i + 1$
end while
$j = j + 1$
end while

Věta 7.2.1. *Uvedený algoritmus 7.2 spočítá korektně matice H a K v čase $O(n^3)$ a vyžaduje $O(n^2)$ paměti.*

$K_{i,j} \leq K_{i,j+1} \leq K_{i+1,j+1}$

XXX obr. matice

Konstrukce opt. bin. vyhl, stromu ze znalosti matice K

- kořen stromu bude $x_{K(1,n)}$.
- podstrom levého syna bude optim. strom pro $S_{1,K(1,n)-1}$
- podstrom pravého syna bude optim. strom pro $S_{K(1,n)+1,n}$

To nám dává rekurzivní algoritmus pro výstavbu opt. bin. vyhl. stromu z matice K, strom takto zkonstruujeme v čase $O(n)$.

7.2.3 Snížení složitosti z kubické na kvadratickou

XXX sloučit s částí předch subsekce

Chceme snížit ryhchlost konstrukce opt. bin. vyhl. stromu z kubické na kvadratickou. Pro tento úkol použijeme *kvadratické programování*. Pomocí této techniky lze modifikovat algoritmus pro konstrukci optimální BVS tak, že bude mít místo složitosti $O(n^3)$ složitost $O(n^2)$.

Algoritmus 7.3 Modifikovaný algoritmus pro výpočet matic H, K

for every $i = 1, 2, .., n$ **do**
  $H_{i,i} = w_{i,i}, K_{i,i} = i$
end for
$j = 1$
while $j < n$ **do**
  $i = 1$
  while $i + j \leq n$ **do**
    $m = K_{i,i} + j, hod = H_{i,m-1} + H_{m+i,i+j}$ (1')
    $k = K_{i,i+j} + 1$ (2')
    while $k \leq K_{i+1,j+1}$ (3') **do**
      if $hod > H_{i,k-1} + H_{k+1,i+j}$ **then**
        $hod = H_{i,k-1} + H_{k+1,i+j}$
        $m = k$
      end if
      $k = k + 1$
    end while
    $H_{i,i+j} = hod + w_{i,i+j}, K_{i,i+j} = m, i = i + 1$
  end while
  $j = j + 1$
end while

Algoritmus 7.2 pro výpočet H a K jsme modifikovali tak, že řádky (1),(2),(3) jsme nahradili řádky (1'),(2'),(3'). Tím vznikl algoritmus 7.3.

Modifikovaný algoritmus: Třetí vnořený cyklus vyžaduje čas $O(K_{i+1,j+1} - K_{i,j})$. Tedy druhý vnořený cyklus vyžaduje čas $O(K_{2,2+j} - K_{1,1+j} + K_{3,3+j} - K_{2,2+j} + K_{4,4+j} - K_{3,3+j} + ... + K_{n-j,n} - K_{n-j-1,n-1}) = O(K_{n-j,n}) = O(n)$.

Věta 7.2.2. *Za předpokladu $K_{i,j} \leq K_{i,j+1} \leq K_{i+1,j+1}$ pro každé $1 \leq i \leq j \leq n-1$ modifikovaný algoritmus korektně spočítá matice H a K v čase $O(n^2)$ a vyžaduje paměť $O(n^2)$.*

Vstup: dána čísla $w(i,j), 1 \leq i \leq j \leq n$

Výstup: definujeme $c(i,j) = \begin{cases} 0 & \text{pro } i = j, \text{kde } i = 1, 2, ..., n \\ w(i,j) + min\{c(i,k-1) + c(k,j), i < k \leq j\} & \text{pro } i \neq j \end{cases}$

Úkolem je tedy nalézt čísla $c(i,j)$. Čísla $c(i,j)$ budou tvořit výstup algoritmu. Když použijeme modifikaci algoritmu pro hledání opt. bin. vyhl. stromu, pak spočítáme $c(i,j)$ v čase $O(n^3)$.

Definice 7.2.1. $K(i,j) = min\{l, l = i+1, ..., j$ a platí $c(i,l-1) + c(l,j) \leq c(i,k-1) + c(k,j) \forall k = i+1, ..., j\}$

Když $K(i,j) \leq K(i,j+1) \leq K(i+1,j+1)$ pro $i \leq j$, pak lze použít algoritmus vyžadující čas $O(n^2)$. (tj. můžeme použít rychlejší modifikaci algoritmu pro hledání opt. bin. vyhl. stromu a spočítáme $c(i,j)$ v čase $O(n^2)$)

Poznámka 7.2.1. Vztah kvadratického programování a hledání opt. bin. vyhl. stromu: Položme $w(i,j) = \sum_{l=i}^{j} \alpha_l + \sum_{l=i-1}^{j} \beta_l$, pak $c(i,j) = H_{i+1,j}$.

Chceme ukázat, že když platí

- (A) $w(i,j) \leq w(i',j')$ pro $i' \leq i \leq j$ $leq j'$

- (B) $w(i,j) + w(i',j') \leq w(i,j') + w(i',j)$ pro $i \leq i' \leq j$ *leq*j'

pak platí $K(i,j) \leq K(i,j+1) \leq K(i+1,j+1)$ pro $1 \leq i \leq j$.

Lemma 7.2.1. *Za uvedených předpokladů platí*
$c(i,j) + c(i',j') \leq c(i,j') + c(i',j)$ *pro* $i \leq i' \leq j \ \leq j'$

Důkaz. indukcí dle $j' - i$:
platí: když $i = i'$ nebo $j = j'$, pak triviálně platí
$c(i,j) + c(i',j') \leq c(i,j') + c(i',j)$
iniciální krok: když $j' - i \leq 0$, pak platí buď $i = i'$ nebo $j = j'$.
indukční krok: předpokládejme, že nerovnost platí, když $j' - i < n$ a nechť $j' - i = n$, kde $n \geq 2$.

1. $j = i'$ pak máme dokázat, že
$c(i,j) + c(j,j') \leq c(i,j')$. označme $K(i,j') = l$.

1. (1a) $l \leq j$ pak $c(i,j) + c(j,j') \overset{\text{z def } c(i,j)}{\leq} w(i,j) + c(i,l-1) + c(l,j) + c(j,j') \overset{\text{z (A) a ind. předp.}}{\leq}$
$w(i,j') + c(i,l-1) = c(i,j)$
víme, že $i < l \leq j$, tedy $j' - l < j' - i$.
$\Rightarrow$ (1a) platí.

2. (1b) $l \geq j$, důkaz stejný jako pro 1a.
$\Rightarrow$ 1 platí.

3. $j > j'$
označme $k = K(i,j'), l = K(i',j)$

1. (2a) $l \leq k$, pak $i' < l <\leq j, i < k < j \leq j'$

$$\begin{aligned} & c(i,j) + c(i',j') \\ \leq & w(i,j) + c(i,l-1) + c(l,j) + w(i',j') + c(i',k-1) + c(k,j') \\ = & w(i,j) + w(i',j') + c(i,l-1) + c(i',k-1) + c(l,j) + c(k,j') \\ \overset{\text{podle (B)}}{\leq} & w(i,j') + w(i',j) + c(i,k-1) + c(i',l-1) + c(l,j) + c(k,j') = \end{aligned}$$

$c(i,k-1) + c(i',l-1)$ jsme dostali z ind. předp.

víme, že $i \leq i' \leq l - 1 \leq k - 1$ a $k - 1 - i < j' - i$.
Potom

$$\begin{aligned} = & w(i,j') + c(i,k-1) + c(k,j') + w(i',j) + c(i,l-1)c(l,j) \\ = & c(i,j') + c(i',j) \end{aligned}$$

2. (2b) $k \leq l$ důkaz je analogický jako pro (2a).

$\Rightarrow$ 2 platí.

$\Rightarrow$ lemma je dokázané. □

Lemma 7.2.2. *Když* $c(i,j)+c(i',j') \leq c(i,j')+c(i',j)$ *pro* $i \leq i' \leq j \leq j'$*, pak platí* $K(i,j) \leq K(i,j+1) \leq K(i+1,j+1)$

Důkaz. Ukážeme, že platí $K(i,j) \leq K(i,j+1)$, důkaz druhé nerovnosti je analogický.
Abychom dokázali $(i,j) \leq K(i,j+1)$, tak stačí ukázat, že platí
$c(i,k-1)+c(k,j) < c(i,k'-1)+c(k',j)$ pak $c(i,k-1)+c(k,j+1) < c(i,k'-1)+c(k',j+1)$
pro $i < k' < k \leq j$.

požadovaná nerovnost plyne z této nerovnosti:

$$\begin{aligned} & c(i,k'-1)+c(k',j)-c(i,k-1)-c(k,j) \\ \overset{c(i,k'-1)>0}{\leq} \; & c(i,k'-1)+c(k',j+1)-c(i,k-1)-c(k,j+1) \end{aligned}$$

skončili jsme s triky, upravíme nerovnost a vyjde to:

$$\begin{aligned} & c(i,k'-1)+c(k',j)+c(i,k-1)+c(k,j+1) \\ \leq & c(i,k'-1)+c(k',j+1)+c(i,k-1)+c(k',j+1) \end{aligned}$$

$c(k',j)+c(k,j+1) \leq c(k',j+1)+c(k,j)$ platí $k' < k \leq j \leq j+1$ ... nerovnost platí dle předpokladu (položíme i = k', i'=k, j=j, j'=j+1) □

XXX pokud se divíte $j = j$, nedivte se, znamená to, že j z 1. části se rovná j z druhé části :) chce to lepší značení

7.3 Skorooptimální binární vyhledávací stromy

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

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

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

7.3.1 Aproximace optimálních stromů

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

Uvažujme následující situaci. $S = \{x_1, x_2, x_3, x_4\}$ s pravděpodobnostmi přístupu (α_0, β_1, α_1, β_2, α_2, β_3, α_3, β_4, α_4) = $(\frac{1}{6}, \frac{1}{24}, 0, \frac{1}{8}, 0, \frac{1}{8}, \frac{1}{8}, 0, \frac{5}{12})$.

Doporučuji si představit uvedené body na reálné ose tak, že α_0 je v bodě 0, α_4 v bodě 1.

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

V našem případě volíme x_3 jako kořen stromu.

Pro rozhodnutí o kořenu levého podstromu vrchol, který popsaným způsobem odpovídá bodu $\frac{1}{4}$. Tento postup není totožný s postupem, kdy se bere bod $\frac{1}{2}$ v nově vzniklé podúloze, protože při konstrukci podstromu bychom zanedbali část intervalu α_3.

7.3.2 Podrobnější popis naznačené metody

Nechť

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

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

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

procedure contruct_tree(i,j,cut,l)

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

1. i a j jsou celá čísla taková, že $0 \leq i < j \leq n$

2. l je celé číslo, $l \geq 0$

3. $cut = \sum_{p=1}^{l-1} x_p 2^{-p}$, kde $x_p \in \{0, 1\}$ pro všechna p

4. $cut \leq s_i \leq s_j \leq cut + 2^{-l+1}$

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

begin
if $i + 1 = j$ (případ A) **return** kořen=j, levý list=i, pravý list=j;
else
najdi k takové, že
5) $i < k \leq j$
6) $k = i + 1$ nebo $s_{k-1} \leq cut + 2^{-l}$
7) $k = j$ nebo $s_k \geq cut + 2^{-l}$
Takové k vždy existuje, protože parametry funkce splňují podmínku 4.
if $k = i + 1$ (případ B) **return**
kořen=i+1
levý list=i
pravý list=**construct_tree**(i+1,j,cut+2^{-l},l+1);
if $k = j$ (případ C) **return**
kořen=j
levý list=**construct_tree**(i,j-1,cut,l+1)
pravý list=j;
if $i + 1 < k < j$ (případ D) **return**
kořen=k
levý list=**construct_tree**(i,k-1,cut,l+1)
pravý list=**construct_tree**(k,j,cut+2^{-l},l+1);
end

Věta 7.3.1. *Nechť b_i je hloubka vrcholu x_i a a_j je hloubka listu (x_j, x_{j+1}) ve stromě T_{BB} vytvořeném funkcí **construct_tree**(0,n,0,1). Potom*
$b_i \leq \lfloor \log 1/\beta_i \rfloor$, $a_j \leq \lfloor \log 1/\alpha_j \rfloor + 2$

Důkaz. Věta říká, že hloubka vrcholu roste s klesající pravděpodobností přístupu k tomuto vrcholu.
Plyne z následujících faktů. □

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

Důkaz.

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

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

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

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

Fakt 2. Hodnoty parametrů všech volání **construct_tree** splňují podmínky 1-4.

Důkaz. Indukcí. **construct_tree**(0, n, 0, 1) splňuje 1-4 a pomocí předchozího faktu. □

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

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

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

Důkaz. Indukcí podle l. □

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

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

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

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

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

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

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

$$P_{BB} \leq \sum \beta_i \lfloor \log 1/\beta_i \rfloor + \sum \alpha_j \lfloor \log 1/\alpha_j \rfloor + 1 + \sum \alpha_j \leq$$
$$\leq H(\alpha_0, \beta_1, \alpha_1, \ldots, \beta_n, \alpha_n) + 1 + \sum \alpha_j \tag{7.2}$$

Navíc

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

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

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

Důkaz. Plyne z věty 5 (původní číslování, viz [1], strana 175) a věty 2. □

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

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

7.3.3 Časová složitost

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

Pro $n > 1$ je potřeba najít k a dojde až ke dvěma rekurzivním voláním **construct_tree**. Nechť $T(m, n)$ je čas potřebný pro nalezení k, kde $m = k - i$ (tj. vzdálenost k od počátku zkoumaného úseku). **construct_tree** je voláno maximálně dvakrát, v případě D první volání sestrojí strom s $k - 1 - i = m - 1$ vrcholy a druhé volání strom s $j - k = n - m$.

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

Zde c_2 je konstanta měřící složitost předávání parametrů.

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

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

7.3.4 Hledání k

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

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

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

Budeme vyhledávat od konců intervalu, ale ne v konstatních krocících.

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

Tento krok trvá konstantní čas.

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

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

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

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

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

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

Dostáváme pro **construct_tree** následující rekurzivní vztah.

$$T(0) = 0$$
$$T(n) = \max_{1 \leq m < n} (T(m-1) + T(n-m) + d(1 + \log \min(m, n - m + 1)) + c)$$

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

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

Pro $n = 0$ vztah zřejmě platí.

Pro $n > 0$ máme

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

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

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

Podle indukčního předpokladu dostáváme

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

Což se rovná

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

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

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

□

7.4 AVL stromy

Binární vyhledávací stromy jsou poměrně příjemné jednoduchou implementací operací nad nimi, ale není přitom nijak omezena jejich hloubka. Může se tedy stát, že strom může vypadat spíše jako seznam, a časová složitost operací bude tedy lineární. Pokud bychom však chtěli, aby měl strom co nejmenší výšku (vzdálenost listů od kořene by se mohla lišit maximálně o jedničku), byly by operace INSERT a DELETE náročné. Rozumným kopmromisem mohou být právě AVL-stromy.

AVL stromy jsou nazvané podle jmen jejich tvůrců (**A**del'son-**V**elskii a **L**andis). Původní článek o AVL stromech lze nalézt v [4].

Definice 7.4.1. Nechť v je vnitřní vrchol stromu T. Potom

- $l(v)$ je délka nejdelší cesty z v do listu v podstromu levého syna v.
- $p(v)$ je délka nejdelší cesty z v do listu v podstromu pravého syna v.

Pokud takový podstrom neexistuje, položme $l(v)$ resp. $p(v)$ rovno -1. Dále označme $b(v) = l(v) - p(v)$. Vrchol v nazveme *vyvážený*, jestliže $b(v)$ nabývá hodnot -1, 0 nebo 1.

Definice 7.4.2. AVL strom je binární vyhledávací strom takový, že pro každý vnitřní vrchol v platí $l(v) - p(v) \in \{-1, 0, 1\}$.

Poznámka 7.4.1. AVL stromy jsou jen jednou z možností jak vyvažovat stromy. *Dokonale vyvážené stromy* stanovují podmínku, že pro každý uzel ve stromu platí, že *počty* uzlů v levém a pravém podstromu tohoto uzlu se liší nejvýše o jedničku. AVL stromy tento požadavek zeslabují tak, že vyžadují, aby se *výšky* levého a pravého podstromu libovolného vrcholu lišily nejvýše o jedničku[1]. Úvod do vyvážených binárních stromů lze nalézt např. v [5].

Věta 7.4.1. *AVL-strom o n vrcholech má výšku nejvýše* $2 \log n$.

Důkaz. Označme $N(h)$ minimální počet vrcholů AVL stromu výšky h. Můžeme tedy $N(h)$ definovat takto:

$$N(0) = 1$$
$$N(1) = 2$$
$$N(h) = 1 + N(h-1) + N(h-2)$$

Je zřejmé, že platí $N(h) \geq 2N(h-2)$ nebo $N(h-2) \leq N(h-1)$.

Odtud dostaneme, že $N(h) \geq 2^{h/2}$ a tedy $h \leq 2 \log N(h)$. Nerovnost $N(h) \geq 2^{h/2}$ dokážeme indukcí:

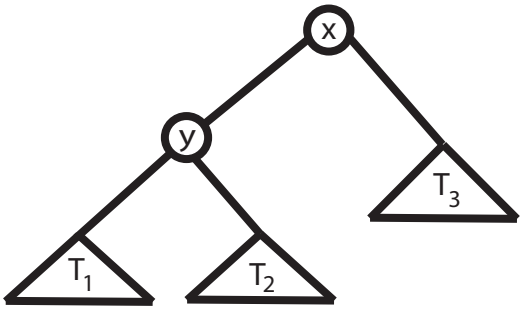
1. $N(0) = 1 \geq 2^{0/2} = 1$
$N(1) = 2 \geq 2^{1/2}$

2. $N(h) \geq 2N(h-2) \geq 2 \cdot 2^{h/(2-1)} = 2^{h/2}$

Všechny vrcholy AVL stromu jsou tedy vyvážené. Nevyvážené vrcholy mohou vzniknout při operacích INSERT a DELETE. Při tom se může hodnota $b(v)$ změnit maximálně o jedničku. Hodnota $b(v)$ nevyváženého vrcholu bude tedy -2 nebo 2. □

7.4.1 Algoritmus INSERT

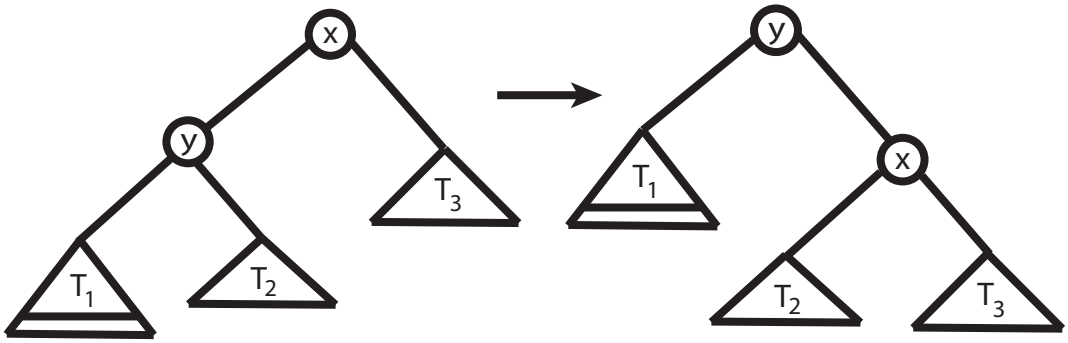
Vložení nového vrcholu do AVL-stromu se provádí stejně jako v nevyvažovaných binárních vyhledávacích stromech. Při tom se však u některých vrcholů může porušit podmínka na vyváženost. Mějme strom na obrázku 7.2.



Obrázek 7.2: AVL strom

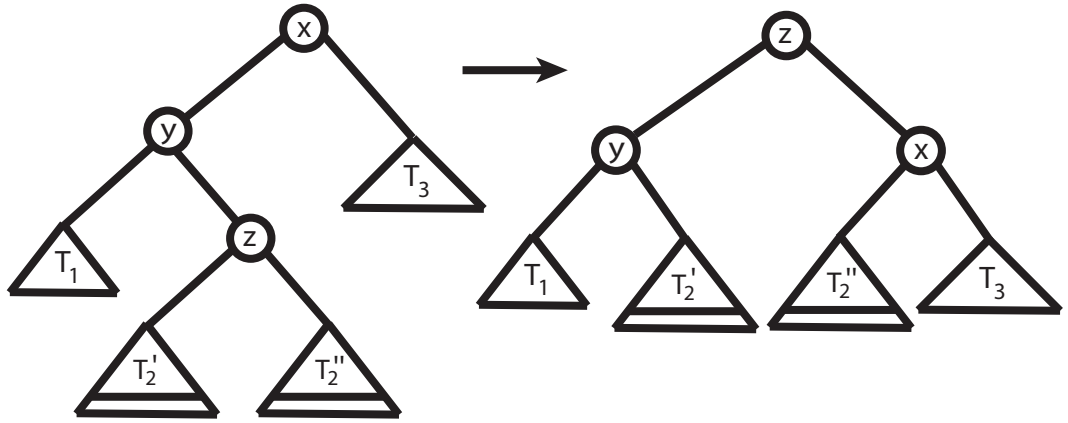
V tomto stromě jsou výšky podstromů T_1, T_2 a T_3 stejné. Pokud při vložení nového vrcholu do podstromu T1 vzroste výška tohoto podstromu, bude vrchol x nevyvážený. Jeho vyvážení se však provede jednoduše pomocí tzv. LL-rotace. (viz obr. 7.3)

[1] Platí, že každý dokonale vyvážený strom je zároveň AVL stromem. Opačné tvrzení neplatí.



Obrázek 7.3: LL-rotace pro AVL stromy

Pokud bychom však chtěli nový vrchol vložit do T_2 a výška tohoto podstromu by vzrostla, byl by opět vrchol x nevyvážený. To se může napravit pomocí LR-rotace. (viz obr. 7.4)



Obrázek 7.4: LR-rotace pro AVL stromy

Poznamenejme, že na vyváženost nemá vliv, zda jsme vrchol vložili do T_2', nebo do T_2". Při vkládání nového vrcholu do pravého podstromu vrcholu x se pro vyvážení používají rotace RR-rotace a RL-rotace, které jsou symetrické k již uvedeným rotacím.

LL-rotaci a RR-rotaci se také někdy říká jednoduchá rotace, kdežto LR-rotaci a RL-rotaci se říká dvojitá rotace. Všimněte si, že po rotaci je výška podstromu, se kterým se rotace prováděla, stejná jako jeho výška před vložením nového vrcholu. Tedy po rotaci není narušena vyváženost nějakého předka. Stačí tedy vyvážit ten nevyvážený vrchol, který je ve stromu nejníže. Pokusme se nyní charakterizovat ten vrchol, který je třeba vyvážit. Samozřejmě, že vrchol x leží na cestě od kořene k přidanému vrcholu a platí:

- buď $b(x) = 1$ a nový vrchol je přidáván vlevo od x
- nebo $b(x) = -1$ a nový vrchol je přidáván vpravo od x

Navíc pro každý vrchol y na cestě od x do přidaného listu je $b(y) = 0$, neboť jinak by byl sám nevyvážený, nebo by nezměnil svou výšku a tedy by nebylo třeba vyvažovat ani x.

Operace INSERT je formálně popsána algoritmem 7.4.

Algoritmus 7.4 INSERT pro AVL stromy

```
INSERT(c)
r ← kořen
while r != NIL do
  if prvek reprezentovaný r = c then
    END
  end if
  if balance(r) != 0 then
    x := r
  end if
  if prvek reprezentovaný r > c then
    r := left(r)
  else
    r := right(r)
  end if
end while
{vložení nového vrcholu}
r := nový vrchol
key(r) := c
b(r) := 0
left(r) := nil
right(r) := nil
r := x
VYVAZUJ(r)
```

7.4.2 Algoritmus DELETE

Ubírání vrcholů z AVL-stromu se provádí stejně jako u nevyvážených binárních vyhledávacích stromů. To znamená, že se ubíraný vrchol nahradí nejpravějším vrcholem levého podstromu nebo nejlevějším vrcholem pravého podstromu[2]. Při tom se samozřejmě mohou také některé vrcholy stát nevyváženými. To se opět řeší pomocí rotací.

Operace DELETE je formálně popsána algoritmem 7.6. V algoritmu se používají dvě procedury pro vyvažování popsané v algoritmech 7.7 a 7.8.

Problém je, že ne při všech rotacích se zachovává výška podstromu, jak tomu bylo u operace INSERT. Proto se zde vyvažování neomezí pouze na jeden vrchol. Při operaci DELETE se může provést až $\log n$ rotací. Každopádně složitost operace DELETE je stejně jako složitost operace INSERT $O(\log n)$.

XXX dokázat max. počet rotací při DELETE

[2] To je klasický postup operace DELETE pro BVS popsaný v [5], str. 70.

Algoritmus 7.5 VYVAZUJ pro AVL stromy

```
{úprava balance (stačí od x do vloženého listu)}
while r != NIL do
  if prvek reprezentovaný r > c then
    balance(r) := balance(r) + 1
    r := left(r)
  end if
  if prvek reprezentovaný r < c then
    balance(r) := balance(r) − 1
    r := right(r)
  else
    r := NIL
  end if
end while
if balance(x) = 2 then
  if prvek reprezentovaný left(x) > c then
    LL-rotace
  else
    LR-rotace
  end if
end if
if balance(x) = −2 then
  if prvek reprezentovaný right(x) < c then
    RR-rotace
  else
    RL-rotace
  end if
end if
```

Algoritmus 7.6 DELETE pro AVL stromy

```
DELETE(x)
r ← otec vrcholu reprezentovaného x
{vrcholu left := r jsme odebrali levého syna}
while r! = NIL do
  {Prochází vrcholy od otce ubraného vrcholu ke kořeni}
  if left then
    if b(r) = 0 then
      {Vrchol r je stále vyvážený a výška jeho podstromu se nezměnila}
      b(r) := −1
    end if
    if b(r) = 1 then
      {Vrchol r je stále vyvážený, ale výška jeho podstromu se snížila}
      b(r) := 0
      {Je třeba vyvažovat}
    else
      VYVAZUJ_RIGHT(right(r))
    end if
  else
    if b(r) = 0 then
      {Vrchol r je stále vyvážený a výška jeho podstromu se nezměnila}
      b(r) = 1
      END
    end if
    if b(r) = −1 then
      {Vrchol r je stále vyvážený, ale výška jeho podstromu se snížila}
      b(r) := 0
      {Je třeba vyvažovat}
    else
      VYVAZUJ_LEFT(left(r))
    end if
  end if
  x := r
  r := otec(r)
  left := left(r) = x
end while
```

Algoritmus 7.7 VYVAZUJ_RIGHT pro AVL stromy

```
VYVAZUJ_RIGHT(x)
if b(x) = 0 then
  RR-rotace
  END
end if
{Výška podstromu se nezměnila}
if b(x) = −1 then
  RR-rotace
else
  RL-rotace
end if
```

Algoritmus 7.8 VYVAZUJ_LEFT pro AVL stromy

```
VYVAZUJ_LEFT(x)
if b(x) = 0 then
  LL-rotace
  {Výška podstromu se nezměnila}
  END
end if
if b(x) = 1 then
  LL-rotace
else
  LR-rotace
end if
```

7.5 Červenočerné stromy

Definice 7.5.1. Binární vyhledávací strom T se nazývá *červenočerný*, jestliže každý vrchol je obarven červeně nebo černě a platí následující podmínky:

1. Listy jsou černé.

2. Pokud má červený vrchol otce, je otec černý.

3. Všechny cesty z kořene do listu mají stejný počet černých vrcholů.

nejdelší cesta je max. 2× delší než nejkratší

Věta 7.5.1. *Pro binární vyhledávací červenočerné stromy reprezentující množinu S, $|S| = n$ platí, že jejich hloubka je $O(\log n)$.*

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

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

To plyne z toho, že cesta z kořene do listu se může skládat v extrémních případech buď z k černých vrcholů, pak je počet vnitřních vrcholů stromu $1 + 2 + ... + 2^{k-1} = 2^k - 1$ nebo z cesty, kde se střídají černé a červené vrcholy, pak je počet vnitřních vrcholů $1 + 2 + ... + 2^{2k-1} = 2^2k - 1$. Tedy platí

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

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

to platí pro všechny bin. vyhl. stromy

7.5.1 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 7.5.2. Strom a jeho vrchol (T, t) nazveme *2-téměř červenočerný strom (2tččs),* jestliže platí

- 1 Listy jsou černé. *(nezměněno)*
- 2' Pokud má červený vrchol *různý od t* 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 7.5.3. Je-li vrchol t červený a jeho otec je také červený, pak řekneme, že t je *porucha*.

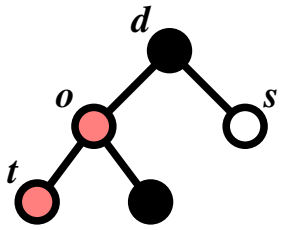
Poznámka 7.5.1. Poruše v 2tččs se také někdy říká *2-porucha*.

Tedy nyní máme 2tččs (T, t) Je-li t porucha, pak ji musíme nějak opravit. Situace je na obrázku 7.5.

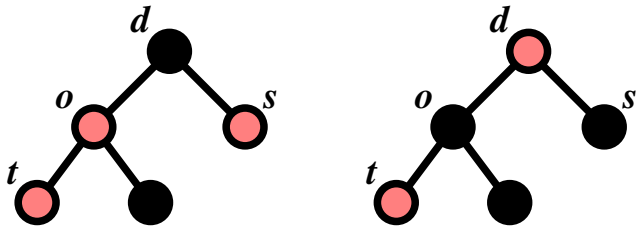
Nejprve záleží na tom, jakou barvu má s, strýc t:

1. s je červený. Pak pouze přebarvíme o, d a s podle obrázku 7.6. Podmínky 1 a 3 jsou splněny. Nyní d může být porucha, ovšem posunutá o 2 hladiny výše. Vznikl 2tččs (T, d).

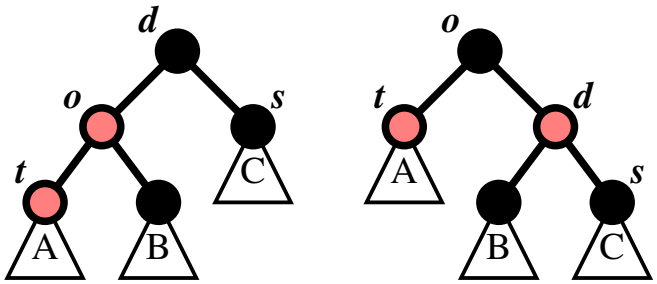
2. s je černý. Záleží na tom, zda hodnota t leží mezi hodnotami o a d nebo ne. Jinými slovy, zda cesta t-o-d obsahuje *zatáčku*.



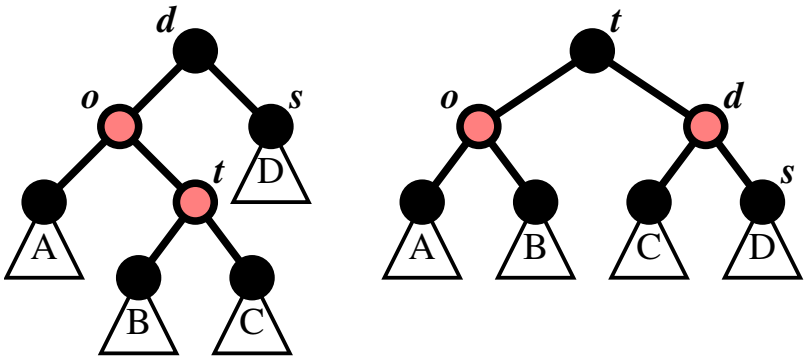
Obrázek 7.5: Obecná situace při INSERTu



Obrázek 7.6: Oprava INSERTu přebarvením



Obrázek 7.7: Oprava INSERTu rotací a přebarvením



Obrázek 7.8: Oprava INSERTu dvojitou rotací a přebarvením

1. Bez zatáčky: Provedeme rotaci a přebarvíme podle obrázku 7.7. Splněny budou podmínky 1, 2 i 3, tedy máme červenočerný strom.

2. Se zatáčkou: Provedeme dvojitou rotaci a přebarvíme podle obrázku 7.8. Splněny budou podmínky 1, 2 i 3, opět máme rovnou červenočerný strom.

7.5.2 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 vrcholem u jeden černý vrchol.

Definice 7.5.4. Strom a jeho vrchol (T, u) nazveme *3-téměř červenočerný strom (3tččs),* jestliže platí

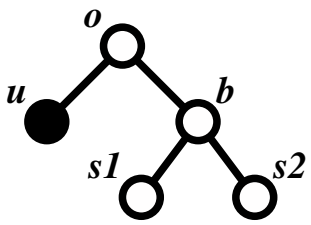
- 1 Listy jsou černé. *(nezměněno)*
- 2 Pokud má červený vrchol otce, je otec černý. *(nezměněno)*
- 3' Všechny cesty z kořene do listu neprocházející u mají stejný počet černých vrcholů, nechť je to k. Všechny cesty z kořene do listu procházející u mají stejný počet černých vrcholů, nechť je to ℓ. A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

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

Poznámka 7.5.2. Takovému vrcholu v 3tččs se někdy říká *3-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 7.9.



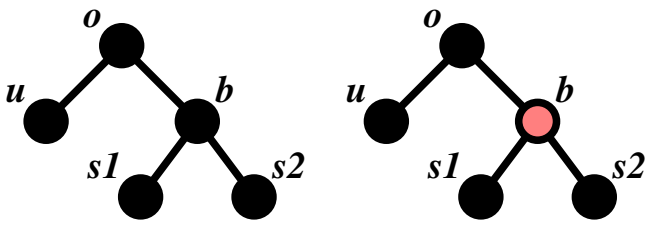
Obrázek 7.9: Obecná situace při DELETE

Oprava záleží na barvě vrcholu b:

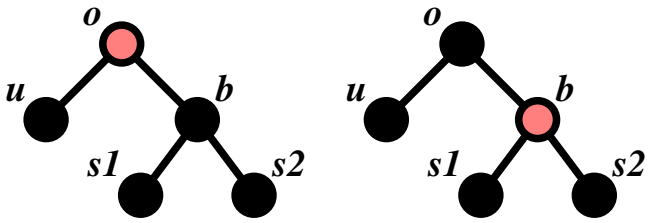
1. Bratr je černý. Rozlišujeme dále 4 případy, z nichž jeden propaguje poruchu o hladinu výš a ostatní skončí s červenočerným stromem.

1. Otec i synovci jsou černí. Přebarvíme b na červeno, viz obrázek 7.10. Dostáváme 3tččs (T, o), tedy porucha je o hladinu výše.

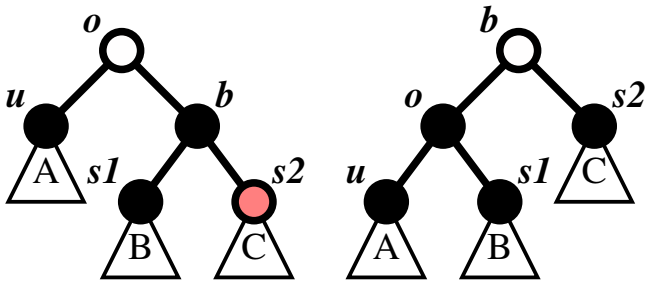
2. Otec je červený, synovci černí. Přebarvíme otce a bratra podle obrázku 7.11 a dostáváme červenočerný strom.



Obrázek 7.10: Částečná oprava DELETE přebarvením



Obrázek 7.11: Oprava DELETE přebarvením



Obrázek 7.12: Oprava DELETE přebarvením a rotací

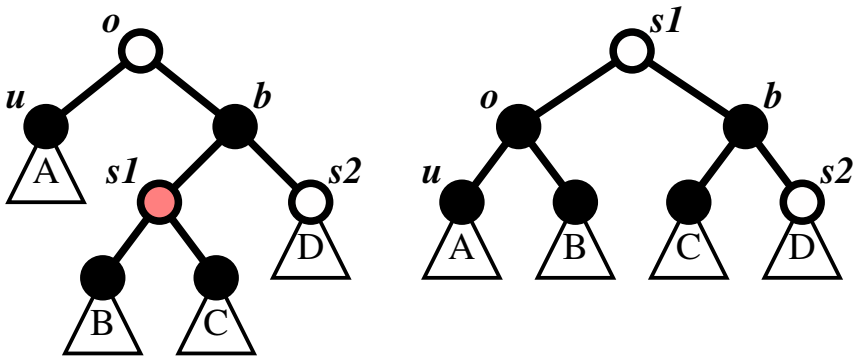
3. Synovec $s1$, jehož hodnota leží mezi hodnotami otce a bratra, je černý, druhý synovec je červený. Přebarvíme a zrotujeme podle obrázku 7.12, barva otce se nemění (tj., vrchol b bude mít barvu, kterou původně měl vrchol o). Dostáváme červenočerný strom.

4. Synovec $s1$, jehož hodnota leží mezi hodnotami otce a bratra, je červený, druhý synovec má libovolnou barvu. Přebarvíme a dvojitě zrotujeme podle obrázku 7.13 (tj., vrchol $s1$ bude mít barvu, kterou původně měl vrchol o a barva vrcholu $s2$ se nezmění). Dostáváme červenočerný strom.

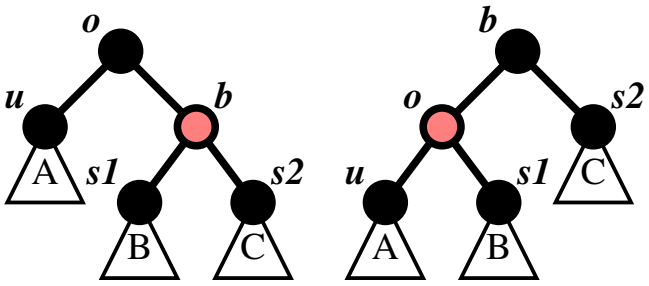
5. Bratr je červený. Provedeme rotaci. Dostaneme strom ve tvaru, který je na 7.14. a aplikujeme předchozí případ č.1.
Přestože to tak na první pohled nevypadá, máme vyhráno, protože bratr poruchy je černý a otec červený, tedy příští oprava bude případ 2, 3, nebo 4 a skončíme s červenočerným stromem.

7.5.3 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



Obrázek 7.13: Oprava DELETE přebarvením a dvojitou rotací



Obrázek 7.14: Částečná oprava DELETE přebarvením a rotací

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[3].

Poznámka 7.5.3. 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).

[3]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`; Co se týče "real-world" aplikací červeno-černých stromů, je možné zmínit packet filter (PF) v OpenBSD, kde se tyto stromy používají k reprezentaci pravidel pro firewall. Pro firewally je čas vyhodnocení jednotlivých paketů proti pravidlům kritický. Zajímavé je, že původní implementaci PF používala AVL stromy. Červeno-černé stromy se ukázaly jako výhodnější. Implementaci červeno-černých stromů lze v OpenBSD najít v `/usr/include/sys/tree.h` v podobě maker jazyka C. Tento soubor obsahuje rovněž makra pro implementaci Splay stromů. 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.*

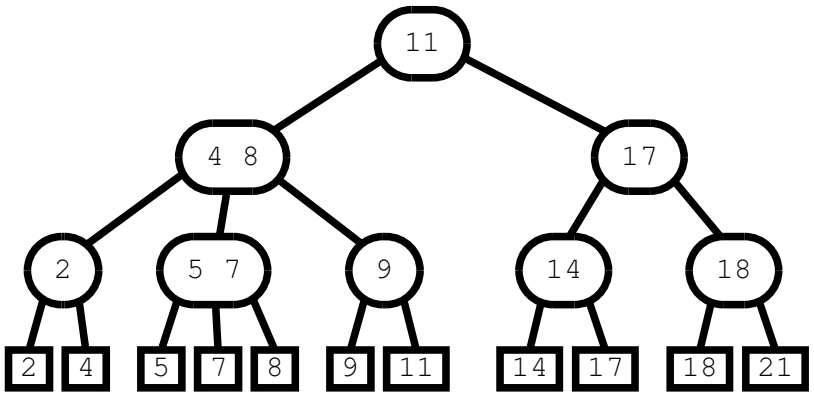
$u \leq_l v$, jestliže otec $u <_l$ otec v nebo otec $u =$ otec v, u je i-tý syn, v je j-tý syn a $i \leq j$.

Poznámka 8.1.1. Prvky z množiny S korespondují s listy T tak, že $s < s', s, s' \in S$, právě když list odpovídající $s <_l$ list odpovídající s'.

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 $\Theta(\log n)$.



Obrázek 8.1: Příklad (a, b) stromu

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:

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

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

viz algoritmus 8.1

Algoritmus 8.1 MEMBER pro (a, b) stromy

```
{vyhledání x}
t := kořen
while t není list do
  i := 1
  while H_t[i] < x ∧ i < ρ_t do
    i := i + 1
  end while
  t := S_t[i]
end while
{testování x}
if t reprezentuje x then
  x ∈ S
else
  x ∉ S
end if
```

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

viz algoritmus 8.2

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

viz algoritmus 8.3

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.*

Algoritmus 8.2 INSERT pro (a, b) stromy

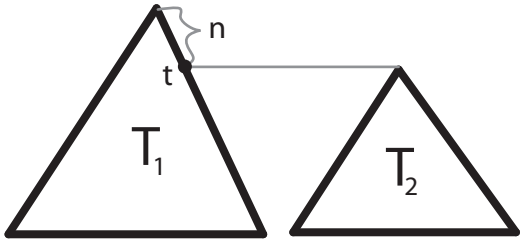
```
vyhledání x
if t nereprezentuje x then
  o := otec t
  vrcholu o přidej nového syna t' reprezentujícího x
  zařaď t' na správné místo mezi jeho bratry a uprav ρ_o, S_o a H_o
  t := o
  while ρ_t > b do
    {Štěpení — můžeme provést díky podmínce 4}
    rozděl t na t_1 a t_2
      k t_1 dej prvních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

Algoritmus 8.3 DELETE pro (a, b) stromy

```
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 {...}
  uprav S_o a ρ_o
  t := o
  while ρ_t < a ∧ t není kořen do
    v := bezprostřední bratr t
    if ρ_v = a then {smíme spojit}
      {Spojení}
      o := otec t
      sluč v a t do t
      uprav ρ_o, S_o a H_o
      t := o
    else {ρ_v > a, spojení by mohlo mít více než b synů}
      {Přesun}
      přesuň krajního syna v do t
      uprav H_{otec t}
    end if
  end while
  if t je kořen a má jen jednoho syna then
    smaž t
  end if
end if
```

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

Výhodnost pro vnější paměti?



Obrázek 8.2: Idea operace JOIN

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

MIN, MAX (XXX)

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

Operace JOIN provede spojení dvou (a,b)-stromů T_1 a T_2 do jednoho (a,b)-stromu za předpokladu, že všechny prvky, které reprezentuje strom T_1 jsou menší než prvky reprezentované stromem T_2.

Algoritmus najde vrchol pro stromu T_2, spojí stromy do jednoho (viz obr. 8.2) a provede štěpění.

Přepis viz algoritmus 8.4

Časová složitost operace JOIN

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

Operace SPLIT(x, T) provede rozdělení (a,b)-stromu T na dva (a,b)-stromy T_1 a T_2 tak, že:

- T_1 je (a,b)-strom reprezentující prvky z $S < x$
- T_2 je (a,b)-strom reprezentující prvky z $S > x$

kde S je množina, reprezentovaná (a,b)-stromem T. Na výstupu této operace dále dostaneme informaci, zda $x \in S$.

Základní myšlenkou pro implementace této operace je použití dvou zásobníků (a,b)-stromů. Procházíme strom T od kořene k listům a na každé úrovni vložíme do prvního zásobníku ty podstromy bratrů aktuálního vrcholu, které obsahují prvky menší než prvek reprezentovaný aktuálním vrcholem. Do druhého zásobníku vložíme podstromy s většími prvky. (viz obr. 8.3) Po projití

Algoritmus 8.4 JOIN pro (a, b) stromy

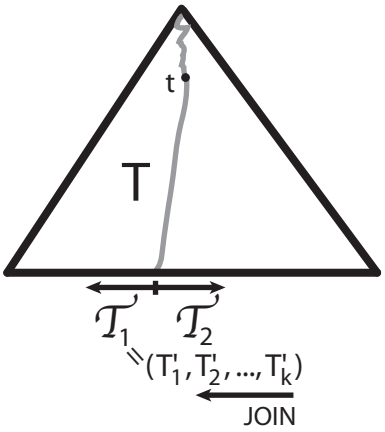
Require: T_1 reprezentuje S_1, T_2 reprezentuje S_2 a $\max S_1 < \min S_2$
  $n :=$ hloubka T_1 − hloubka T_2
  if $n \geq 0$ **then**
    $t :=$ kořen T_1
    while $n > 0$ **do**
      $t :=$ poslední syn t
      $n := n - 1$
    end while
    Spoj t s kořenem T_2 a vytvoř nový vrchol t'. {zde se 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

stromu provedeme slití těchto dvou zásobníků do stromů T_1 a T_2 pomocí operace STACKJOIN. (viz sekce 8.2.3)

Přepis operace SPLIT viz algoritmus 8.5

Algoritmus 8.5 SPLIT pro (a, b) stromy

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)



Obrázek 8.3: Idea operace SPLIT

Časová složitost operace SPLIT

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

Operace STACKJOIN provede JOIN všech (a,b)-stromů uložených na zásobníku. Výsledkem je jediný (a,b)-strom.

Přepis viz algoritmus 8.6

Algoritmus 8.6 STACKJOIN pro (a, b) stromy

```
T := Pop(Z)
while Z ≠ ∅ do
  T' := Pop(Z)
  T := JOIN(T, T')
end while
```

Časová složitost operace STACKJOIN

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

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

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

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

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

Nalezení k-tého nejmenšího prvku.

Rozšíříme reprezentaci stromu a každému vnitřnímu vrcholu v přidáme:

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

viz algoritmus 8.7

Algoritmus 8.7 FIND pro (a, b) stromy

```
t := kořen T
while t není list do
  i := 1
  while K_t[i] < k ∧ i < ρ_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.

8.3 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:

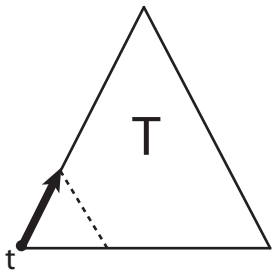
- Listy stromu jsou propojeny do seznamu
- Je známa cesta z nejmenšího (nejlevějšího) listu do kořene (uložená např. v zásobníku)

Použijeme $(2, 3)$-strom. Proč, to si zdůvodníme až po odvození složitosti A-sortu.

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.



Obrázek 8.4: Idea algoritmu A-INSERT

8.3.1 A-INSERT

A-INSERT (viz algoritmus 8.8) pracuje téměř stejně jako původní INSERT - najde správný list a potom případně přidá nový prvek. K nalezení správného listu ovšem využívá cestu z nejmenšího listu. (viz obr. 8.4)

Zde uvedená verze A-INSERTu odstraňuje duplicitní prvky, operaci lze pochopitelně upravit tak, že nechává duplicitní prvky, které zůstávají ve stejném pořadí.

8.3.2 Složitost A-sortu

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 · čas přidání vrcholu + počet štěpení · 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 případě vyžadují

Algoritmus 8.8 A-INSERT(x)

```
{Nalezení}
t := nejmenší list stromu T
repeat
  t := otec t
until t je kořen ∨ x ≤ 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 ∧ i < ρ_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 ρ_o, S_o a H_o
  t := o
  while ρ_t > b do
    {Štěpení — můžeme provést díky podmínce 4}
    rozděl t na t_1 a t_2
      k t_1 dej prvních ⌊(b + 1)/2⌋ synů t
      k t_2 dej zbylých ⌈(b + 1)/2⌉ synů t
    o := otec t
    uprav ρ_o, S_o a H_o
    {při štěpení kořene ještě musíme vytvořit nový kořen}
  end while
end if
```

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.

8.4 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 $b \geq 2a$

8.4.1 Paralelní INSERT(x) do (a, b) stromu

viz algoritmus 8.9

Algoritmus 8.9 paralelní INSERT pro (a, b) stromy

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

8.4.2 Paralelní DELETE(x) z (a, b) stromu

viz algoritmus 8.10

8.5 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

Algoritmus 8.10 paralelní DELETE pro (a, b) stromy

```
o := lock(nadkořen) {Nadkořen je implementační pomůcka. Slouží k zamknutí přístupu k celému
stromu a uchovává max(S)}
t := kořen
h := nil {Jakmile h ≠ nil, x ∈ H_h a h bude zamčený do konce procesu.}
{Invariant mezi průchody cyklem: o je otec t, o je kromě h jediný vrchol zamknutý tímto proce-
sem.}
while t není list do
  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)
```

kroků. V této sekci se podíváme na počet vyvažovacích kroků pro posloupnost operací INSERT a DELETE.

Nechť $b \geq 2a$.

Věta 8.5.1. *Mějme posloupnost n operací INSERT a DELETE aplikovanou na prázdný (a,b) strom. Označme P počet přesunů při provádění posloupnosti, SP počet spojení a ST počet štěpení. Dále označme P_h, SP_h a ST_h počet přesunů. spojení a štěpení, které nastanou ve výšce h (listy mají výšku 0).*

Nechť

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

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 8.5.1. (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:

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

8.5.1 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:

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

8.5.2 Štěpení

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

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

8.5.3 Spojení

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

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

8.5.4 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 $z(T)$.

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

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

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

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

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

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

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

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

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

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

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

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

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

Tím dostáváme 8.4:

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

8.6 Propojené (a, b) stromy s prstem

Variantou (a, b) stromů jsou (a, b) stromy, které mají propojené jednotlivé hladiny a dále obsahují ukazatel na jeden z listů. Těmto stromům se také někdy říká jenom stromy s prstem (předpokládá se, že jsou propojené) nebo hladinově propojené. V anglické literatuře se vyskytují pod pojmem *finger trees*.

Struktura vnitřního vrcholu v obsahuje následující položky:

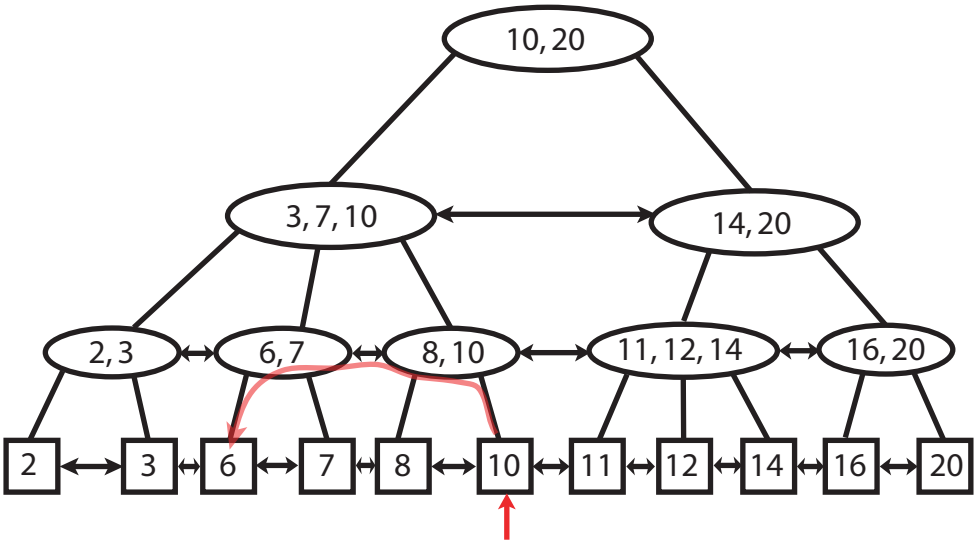
- $\rho(v)$ = počet synů v
- $Syn[1..\rho(v)]$ je pole ukazatelů na syny vrcholu v

- $Hod[1..\rho(v)]$ je pole hodnot, platí
- $Hod(i-1) <$ prvky reprezentované v podstromu i-tého syna $\leq Hod(i)$
- $otec(v) =$ ukazatel na otce vrcholu v

Předchůdce(v), Následník(v) } ukazatele na bezprostř. předchůdce (následníka) v na hladině vrcholu v (v lexikogr. uspořádání)

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

Příklad (a,b)-stromu s prstem je vidět na obr. 8.5



Obrázek 8.5: (a,b)-strom s prstem při provedení operace MEMBER(6)

8.6.1 Algoritmus MEMBER

Viz algoritmus 8.11

XXX alg. MEMBER je velmi podivny, prepracovat

8.6.2 Algoritmus FINGER

FINGER(x)
nastaví hodnotu na list, který reprezentuje prvek nejbližší k x.

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

- vyhledání x vyžaduje čas $O(1 + \log(l))$
- nastavím prst na nějakou vhodnou hodnotu

Algoritmus 8.11 MEMBER (a,b) stromy s prstem

MEMBER(x)
1) Nechť y je hodnota, na kterou ukazuje Prst.
if $x < y$ **then**
  pokračuju 2)
else
  3)
end if
2) $v \leftarrow otec(y)$
dokud $x < h$(Předchůdce(Předchůdce(v))
jdu na otce(Předchůdce(v))
v opačném případě
když $x \leq$ h(Předchůdce(v)) pak
$v \leftarrow$ Předchůdce(v)
a pokračuji normálním vyhledáváním
3) symetrické ke 2)

Věta 8.6.1. *Nechť T je propojovaný (a,b) strom s prstem a nechť P je posloupnost příkazů MEMBER, INSERT, DELETE, FINGER, kterou provedeme na T. Pak P vyžaduje čas $O(log(n) +$ čas na vyhledání$)$ kde n je velikost množiny reprezentované stromem T. ($b \geq 2a$)*

8.6.3 Amortizovaná složitost

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

Bankovní paradigma

Použijeme následující značení pro přechodu mezi stavy (situacemi): $D \xrightarrow{o} D'$

- D - vstupní situace
- o - operace
- D' - výstupní operace

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

$D_0 \xrightarrow{O_1} D_1 \xrightarrow{O_2} D_2 \to \ldots \to D_n$

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

Obvykle platí, že $bal \geq 0$ nebo $bal \leq 0$.
Když $bal \geq 0$, pak:

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

Když $bal \leq 0$, pak

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

Začínáme na prázdném (a,b) stromě $\rightarrow$ $bal = 0$.
XXX nechybi tady neco ?

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 Seznamy

x

XX operace nad obycejne seznamy jsou velmi jednoduche. Všechny musí lineárně projít celý seznam než provedou danou operaci.

MEMBER INSERT DELETE

9.1.1 Algoritmus MFR (Move Front Rule)

přednáška z 18.3.2003

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

Věta 9.1.1. *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 na seznam $S2$, tak platí:

Označíme:

- s = *čas na vyhledání* A
- p = *počet placených výměn* A
- f = *počet volných výměn* A

$$\text{Pak čas MFR} \leq \begin{cases} s + p - f - |P| & \text{když } S1 = S2 \\ s + p - f - |P| + \binom{|S|}{2} & \text{když } S1 \neq S2 \end{cases}$$

Definice 9.1.1. 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 9.1.1. 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)

tento dukaz je nejaky podivny

Důkaz věty 9.1.1. 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:

$$= \text{čas MFR pro operaci } O + bal(S'_A, S'_{MFR}) - bal(S_A, S_{MFR})$$

Balance *bal* je definována vzhledem k algoritmu A.

Ukážeme, že amortizovaná složitost O pro MFR

$$\leq 2 * \text{čas na vyhledání A} + \text{počet placených výměn A} - \text{počet volných výměn A} - 1$$

$$\mathrm{S_A} \overset{\text{vyhledání}}{\rightarrow} S''_A \overset{\text{výměny}}{\rightarrow} S'_A$$
$$\mathrm{S_{MFR}} \rightarrow S'_{MFR} \rightarrow S'_{MFR}$$

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 $\leq 2 * \text{čas na vyhledání pro } A - 1$
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 tedy amort. slož. MFR = čas operace = $n + 1 \leq 2(n + 1) - 1$
$n + 1$ 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 = $n + 1 + n = 2n + 1 = 2(n + 1) - 1 = 2 * \text{čas na vyhledání A} - 1$

(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. složitost

1. fáze operace $\leq 2 *$ čas na vyhledání A $- 1$

2. fáze operace = počet placených výměn A $-$ počet volných výměn A

Při placené výměně si v seznamu S''_A vymění x místo z za x, tedy dvojice $\{x, z\}$ přibude při počítání $bal(S'_A, S'_{MFR}) - bal(S''_A, S'_{MFR})$
(v S_{MFR} je x první)

Při volné výměně se v seznamu S''_A vymění x místo s 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 $- |P| + bal(S_1, S_2)$

$|P|$ ... za každou operaci je -1

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

□

temporary solution by T.Matoušek

Očekávaná složitost MFR

Důkaz. $E_{MFR} = \sum l_i p_i$ kde l_i je očekávaná vzdálenost x_i od začátku seznamu a p_i je pravděpodobnost, že se budeme ptát na x_i.

$$l_i = 1 + E(\sum_{j=1}^{n} Y_{ij})$$

Jednička je tam za prvek x_i, kde Y_{ij} je náhodná proměnná s alternativním rozdělením s pravděpodobností p_{ij} a říká, jestli je prvek x_j před x_i.

Průměr součtu je součet průměrů, takže platí:

$$l_i = 1 + \sum_{j=1}^{n} EY_{ij}$$

Dále víme, že $EY_{ij} = p_{ij}$, nebo-li

$$l_i = 1 + \sum_{j=1}^{n} p_{ij}$$

Zbývá tedy spočítat p_{ij} :

$$\begin{aligned}
p_{ij} &= p(\text{"}x_j \text{ bude před } x_i\text{"}) \\
&= p(\text{"poslední MEMBER, který byl na } x_i \text{ nebo } x_j\text{,byl na } x_j\text{"}) \\
&= p(\text{"poslední zavolání MEMBER bylo na } x_j\text{"}|\text{"poslední zavolání MEMBER bylo na } x_i \text{ nebo } x_j\text{"}) \\
&\stackrel{*}{=} p(\text{"zavolání MEMBER na } x_j\text{"}|\text{"zavolání MEMBER na } x_i \text{ nebo } x_j\text{"}) \\
&= \frac{p_j}{p_i + p_j}
\end{aligned} \tag{9.1}$$

* - každé volání MEMBER na daný prvek je stejne pravděpodobné

Když to dáme dohromady:

$$E_{MFR} = \sum l_i p_i = \sum [1 + \sum_{j=1}^{n} \frac{p_j}{p_i + p_j}] p_i = 1 + \sum_{i,j=1}^{n} p_i p_j p_i + p_j$$

□

Poznámka 9.1.2. S tímto jsme se setkali při EISCH.
Je to důvod, proč je EISCH lepší než LISCH, VICH lepší než LICH.

9.1.2 Algoritmus TR (Transposition Rule)

Když je x při operaci MEMBER(x) a INSERT(x) na i-tém místě, tak ho dá přísl. operace na $(i-1)$-ní místo.

Pokud při INSERT(x) není x v seznamu, INSERT umístí x na předposlední místo.

Poznámka 9.1.3. Lze najít posloupnost příkazů P lib. délky, že MFR vyžaduje čas ($|P|$) a TR vyžaduje čas ($|P|^2$). Na druhou stranu očekávaný čas TR $\leq$ očekávaný čas MFR.

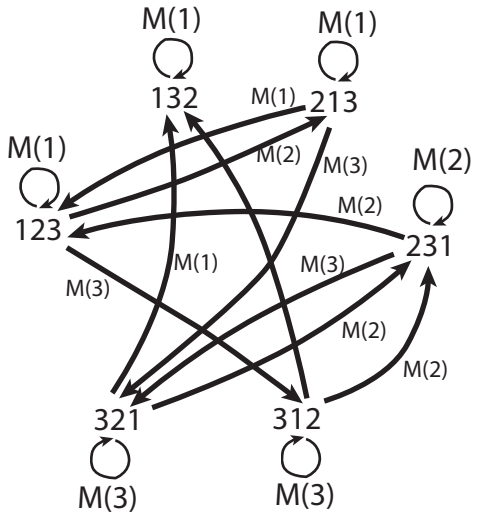
Chceme spočítat očekávaný čas MFR pro posloupnosti P aplikované na seznam S, kde P obsahuje jen operace MEMBER(x) pro $x \in S$.

Předpokládejme, že $S = 1, 2, ..., n$ a β_1 = pravděpodobnost operace MEMBER(x) pro $x \in S$.
$S = \{1, 2, 3\}$ ... 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 Π je dána pravděpodobnost κ_Π, že po provedení náhodné posloupnosti P s daným rozložením operací skončíme u seznamu Π.

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

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



Obrázek 9.1: Přechody mezi stavy

pak

$$\begin{aligned} p_i &= \sum_{\Pi} \kappa_\Pi \Pi(i) \\ &= \sum_{\Pi} \kappa_\Pi (1 + |j, \Pi(j) < \Pi(i)| \\ &= 1 + \sum j, \Pi\{\kappa_\Pi, \Pi(j) < \Pi(i)\} \\ &= 1 + \sum_j \delta(j,i)(1) \end{aligned} \tag{9.2}$$

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 ?

$$\beta_j \sum_{k=0}^{\infty} [1 - (\beta_i - \beta_j)]^k = \beta_j \frac{1}{1-(1-(\beta_i+\beta_j)} = \frac{\beta_j}{\beta_j+\beta_i} \overset{(1)}{=} 1 + \sum_{\substack{j,i \\ j\neq i}} \frac{\beta_j}{\beta_j+\beta_i} \tag{9.3}$$

Očekávaný čas operace je

jake operace ?
XXX

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

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. Klasický algoritmus je takový algoritmus, který předem ví, jaké jsou pravděpodobnosti přístupu k jednotlivým prvkům a má předem seznam srovnaný sestupně podle těchto pravděpodobností.

Očekávaný čas tohoto algoritmu je

$$\begin{aligned}\sum_{i=1}^{n} i\beta_i &= 1 + \sum_{i,j=1} 2\frac{\beta_j\beta_i}{\beta_i+\beta_j} \\ &\leq 1 + \sum_{\substack{i,j \\ j<i}} 2\beta_i = 1 + \sum_{i=1}^{n} 2(i-1)\beta_i \\ &= 1 + 2\cdot\sum_i i\beta_i - 2\sum_i \beta_i \\ &= 2\sum_{i=1}^{n} i\beta_i - 1\end{aligned} \tag{9.4}$$

Platí

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

9.2 Splay stromy

Splay strom (splay tree, rozvinutý strom) patří do kategorie adaptabilních datových struktur určených k vyhledávání. Má základní vlastnosti binárních vyhledávacích stromů - obsahuje ohodnocené prvky. Každému reprezentovanému prvku $s \in S$, kde $S \subseteq U$, (U je universum) je přiřazena váha.

V průběhu operací nad touto strukturou se však mění její uspořádání ve prospěch celkového snížení časové složitosti.

9.2.1 Operace SPLAY

Základní operací je pro práci s těmito stromy je SPLAY(x) - rozšíření, 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)

Tento mechanismus se začíná od stanoveného uzlu, a postupnými rotacemi způsobuje, že stanovený uzel se stane kořenem stromu, při zachování vyhledávacích relací. Celkovým výsledkem je skutečnost, že často používané položky se hromadí v blízkosti kořene. Na rozdíl od BVS, jehož nejhorší případ pro degenerovaný (lineární) strom má složitost $O(N)$ a je složitost splay stromu pro "k" různých po sobě jdoucích operací $O(k * \log(N))$. Tato složitost není stanovena tradičním přístupem "nejhorší případ", který hledá nejnevýhodnější situaci izolované operace, ale metodou "amortizované analýzy" (amortized analysis), která hodnotí celou sekvenci různých operací. Některé z nich jsou delší, některé kratší než $\log(N)$, ale v průměru vychází složitost $O(ln(N))$.

Splay stromy představují jeden z příkladů adaptabilních datových struktur, jejichž vnitřní uspořádání se mění vlivem jako vedlejší jev operací nad těmoto strukturami. Mají dobrou tendenci vyvažovat stromovou strukturu a svou vlastností přibližovat často vyhledávané klíče kořeni se podobají adaptibilní lineární struktuře pro sekvenční vyhledávání, v níž se každý vyhledaný uzel vymění se svým levým předchůdcem. I ve stromové podobě si algoritmus zachovává jednoduchost a průhlednost.

9.2.2 Podporované operace

MEMBER(x,T), INSERT(x,T), DELETE(x,T), 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, tj. $maxT_1 < minT_2$.

  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, tj. $maxT_1 < x < minT_2$.

  Výsledný strom reprezentuje $T_1 \cup T_2 \cup x$.

- SPLIT(x,T)
  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 $\triangle$.

9.2.3 Algoritmus MEMBER

Mechanismus vyhledání (splay search), pracuje stejně jako u BVS, ale po vyhledání se aplikuje na vyhledaný uzel mechanismus Splay, jehož výsledkem je přesunutí uzlu na místo kořene.

Viz algoritmus 9.1

Algoritmus 9.1 MEMBER pro Splay stromy

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

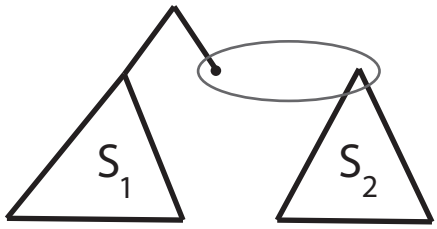
9.2.4 Algoritmus JOIN2

Viz algoritmus 9.2

Algoritmus 9.2 JOIN2(T_1,T_2)

```
SPLAY(∞, T1) // XXX (největší ?) nejmenší prvek
kořen T2 bude pravý syn kořene T1
```

Operací SPLAY se z T_1 stane strom, kde pravý syn kořene bude list. Místo toho listu navěsíme strom T_2.
Pak budou v levém podstromu kořene tohoto nového stromu všechny prvky menší než hodnota v kořenu a v pravém všechny větší, což chceme.



Obrázek 9.2: JOIN2 pro splay stromy

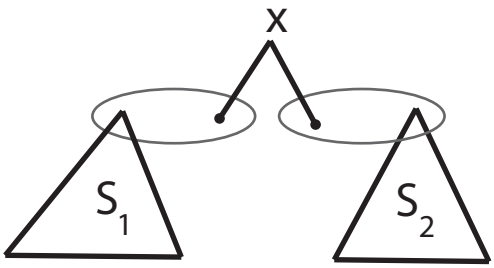
9.2.5 Algoritmus JOIN3

Viz algoritmus 9.3

Algoritmus 9.3 JOIN3(T_1, x, T_2)

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

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



Obrázek 9.3: JOIN3 pro splay stromy

9.2.6 Algoritmus SPLIT

Viz algoritmus 9.4

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

9.2.7 Algoritmus DELETE

Mechanismus rušení uzlu (splay delete) je poněkud složitější. Uzel, který se má zrušit, se mechanismem splay přesune na pozici kořene. Zrušením kořene získáme 2 podstromy. Mechanismus splay se dále aplikuje na bezprostředního předchůdce a není-li tak následníka zrušeného uzlu (ve smyslu relace uspořádání - v průchodu inorder). Tím se tento uzel dostane do pozice kořene levého podstromu. Podle pravidel vyhledávacího stromu musí být všechny uzly levého podstromu menší než

Algoritmus 9.4 SPLIT(x,T)

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

jeho kořen a všechny uzly pravého podstromu větší. Proto musí mít levý podstrom kořen vpravo volný a na toto místo se připojí pravý podstrom. Tento postup má symetrickou - levou versi. Operace "Splay Delete", rušící uzel D XXX je uvedena na obr.2.2. XXX

Viz algoritmus 9.5

Algoritmus 9.5 DELETE(x)

SPLAY(x)
if kořen reprezentuje x **then**
  T_1 je podstrom levého syna kořene T
  T_2 je podstrom pravého syna kořene T
  T $\leftarrow$ JOIN2(T_1, T_2)
end if

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

9.2.8 Algoritmus INSERT

Mechanismus vkládání (splay insert) vloží uzel jako list stejným způsobem jako BVS, ale potom se aplikuje na vložený uzel mechanismus "splay", který opět posune vložený uzel na pozici kořene. Operace "Splay insert" uzlu s klíčem C XXX je uvedena na obr. 9.4.

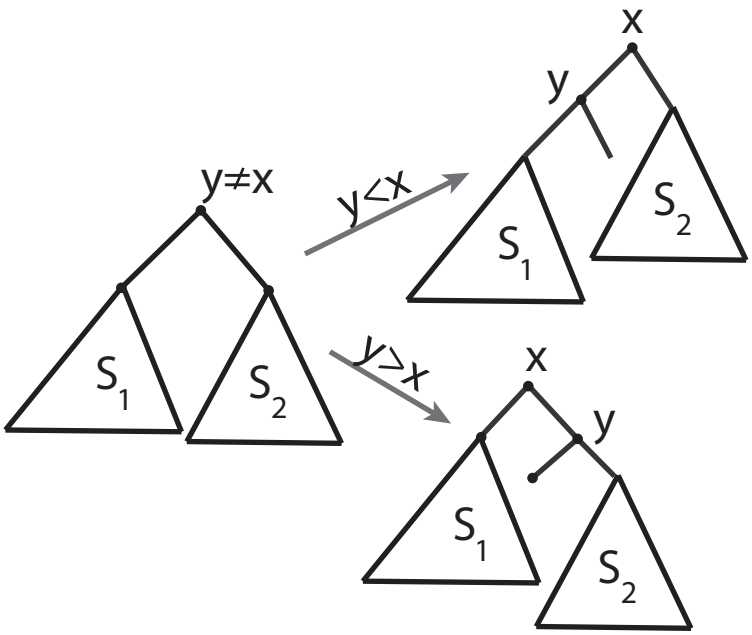
Viz algoritmus 9.6

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

9.2.9 Algoritmus CHANGEWEIGHT

Viz algoritmus 9.7

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.



Obrázek 9.4: INSERT(x) pro splay stromy

Algoritmus 9.6 INSERT(x)

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

Algoritmus 9.7 CHANGEWEIGHT(x, $\triangle$)

```
SPLAY(x)
if x je reprezentován v kořeni then
  k váze x přičti △
end if
```

chybí obrázek, tady je to nejake zmatene

Příklad 9.2.1. $tw(a) = w(a) + w(b) + w(c)$

$r(x)$ je $rank(x)$
$r(x) = \lfloor \log tw(x) \rfloor$

$bal(konfigurace) = \sum\{r(x) : x \in konfigurace\}$
Pro strom T je $tw(x) = tw(\text{kořen } T)$
$r(T) = r(\text{kořen } T)$

Lemma 9.2.1. *Nechť T je binární vyhledávací strom, t je vnitřní vrchol a u,v jsou synové t. Pak $r(t) > min\{r(u), r(v)\}(r(list) = -\infty)$.*

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

$$r(t) = \lfloor \log tw(t) \rfloor \geq \lfloor \log 2tw(u) \rfloor = 1 + \lfloor \log tw(u) \rfloor = 1 + r(u)$$

□

9.2.10 Algoritmus SPLAY

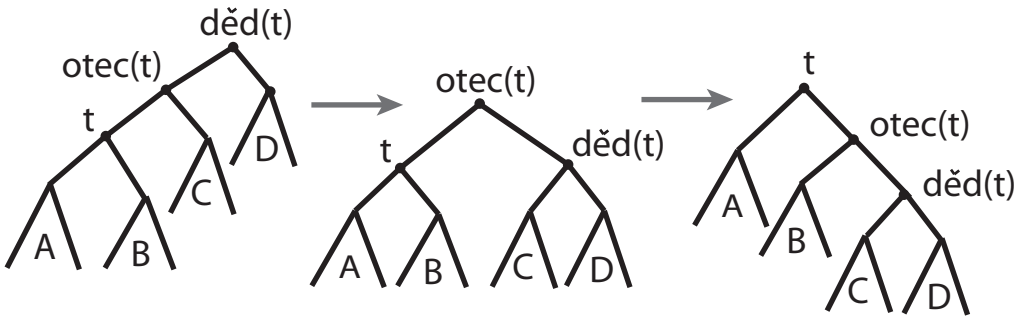
Volání algoritmu SPLAY se většinou zapisuje jako SPLAY(x), kde explitictně neuvádíme strom, na kterém je operace prováděna - to většinou vyplyne z kontextu. Tam, kde je nutné uvést, na kterém stromě se operace SPLAY provádí (např. v implementaci operace JOIN2, píšeme volání jako SPLAY(x,T).

Viz algoritmus 9.8

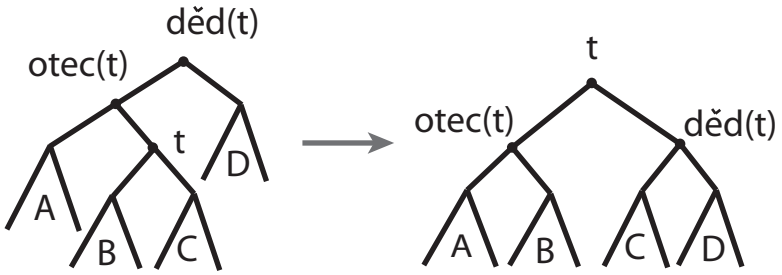
Algoritmus 9.8 SPLAY(x)

```
SPLAY(x)
t ← kořen
while t není list a t nereprezentuje 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 oba leví (praví) synové then
      rotace(otec(t), děd(t))
      rotace(t, otec(t))
    else
      dvojitá rotace(t, otec(t), děd(t))
    end if
  end if
end while
```

V algoritmu SPLAY (algoritmus 9.8) se používá jednoduché (obr. 9.5) a dvojité (obr. 9.6) rotace. Vrchol t se po skončení operace SPLAY(x) dostane do kořene. Toho dosáhneme tak, že v prvním cyklu najdeme vrchol t reprezentující prvek x, v druhém cyklu přesouváme vrchol t do kořene.



Obrázek 9.5: Dvakrát jednoduchá rotace pro SPLAY(x)



Obrázek 9.6: Dvojrotace pro SPLAY(x)

9.2.11 Amortizovaná složitost SPLAY

Budeme předpokládat, že $v(x) \geq 1$ pro každé x.
Totální váha $x = w(x)$, což je součet vah všech prvků v podstromu vrcholu reprezentujícího x.

Značíme $r(x) = \lfloor log_2 w(x) \rfloor$ = rank vrcholu x

Pro strom T:

$$w(T) = w(\text{kořen}(T)) r(T) = r(\text{kořen}(T))$$

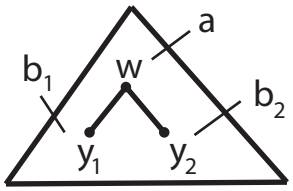
bal(konfigurace) = součet ranků všech vrcholů v množinách tvořících konfigurace

Náš cíl : Budeme chtít ukázat, že amortizovaná složitost operací je $O(\log \frac{w(T)}{v(x)})$, když T reprezentuje S.

Čas operace SPLAY = počet běhů druhého cyklu, který vrchol t transportuje do kořene.

Lemma 9.2.2. *Pomocné lemma: Mějme vrchol w ve stromě T se syny y_1 a y_2 a předpokládejme, že y_1, y_2 nejsou listy. Když w reprezentuje a, y_r reprezentuje b_i pro $i = 1, 2$, pak $rank(a) > min\{r(b_1), r(b_2)\}$*

Důkaz. Situaci lze vidět na obrázku 9.7.



Obrázek 9.7: Pro důkaz pomocného lemmatu pro splay stromy

Zřejmě $r(a) \geq r(b_1)$ a $r(a) \geq r(b_2)$, tedy $r(b_1) \neq r(b_2) \Rightarrow r(a) > min\{r(b_1), r(b_2)\}$

$$\begin{aligned} r(a) &= \lfloor \log_2 w(a) \rfloor \geq \lfloor (w(b_1) + w(b_2)) \rfloor \\ &\geq \lfloor \log_2 (2 - min\{w(b_1), w(b_2)\}) \\ &= 1 + \log_2 min\{w(b_1), w(b_2)\} \\ &= 1 + min\{w(b_1), w(b_2)\} \end{aligned} \tag{9.5}$$

□

Věta 9.2.1. *Amortizovaná složitost operace SPLAY(x,T)* $\leq 3(r(T) - r(t)) + 1$*, kde t je vrchol, který transportujeme do kořene. T je strom reprezentující S. (když x je prvek reprez. množiny, pak t reprezentuje x, jinak je to buď největší nebo nejmenší prvek menší (větší) než x)*

Důkaz. Označme T_0 původní strom, T_i strom po i-tém běhu druhého cyklu v SPLAY a předpokládejme, že druhý cyklus běží k-krát. (tj. T_k je výsledný strom)

Amortizovaná složitost (SPLAY(x,T)) "čas operace" = k

$$\begin{aligned} &= \text{čas operace} + bal(\text{výsledná konfigurace}) - bal(\text{původní konfigurace}) \\ &= k + bal(T_k) - bal(T_0) \\ &= \sum_{i=1} k(1 + bal(T_i) + bal(T_{i-1}) \end{aligned} \tag{9.6}$$

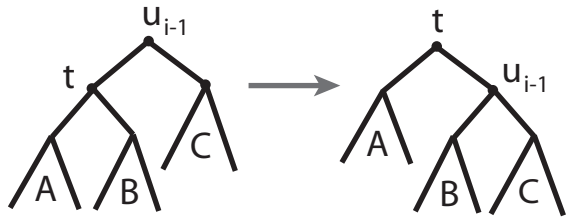
$balance =$ $\sum$ rank v T_k

Označme r_i rank ve stromě T_i, nechť u_i je otec t ve stromě T_i a když u_i není kořen T_i, pak v_i je otec u_i v T_i.

- a) u_i je kořen:
  chci odhadnout $1 + bal(T_i) - bal(T_{i-1})$:

$$\begin{aligned} &1 + bal(T_i) - bal(T_{i-1}) \\ &= 1 + \sum_{z \text{ reprezentován v } T_i} r_i(z) + \sum_z r_{i-1}(z) \\ &= 1 + r_i(u_{i-1}) + r_i(t) - r_{i-1} - r_{i-1}(t) \\ &= 1 + r_i(u_{i-1}) - r_{i-1}(t) \leq 1 + 3(r_{i-1}(u_{i-1}) - r_{i-1}(t)) \end{aligned} \tag{9.7}$$

  Platí $r_i(t) = r_{i-1}(u_{i-1})$, protože stromy A,B,C na obr. 9.8 jsou stejné.



Obrázek 9.8: Pro důkaz amort. složitosti operace SPLAY

Platí

$$r_i(u_{i-1}) \leq r_{i-1}(u_{i-1})$$
$$r_{i-1}(u_{i-1}) \geq r_{i-1}(t)$$

- b) u_{i-1} není kořen:

  1. u_{i-1} je jiný syn v T_{i-1} než t

  2. u_{i-1} je stejný syn v T_{i-1} jako t

- ad b1) :

$$\begin{aligned} & 1 + bal(T_i) - bal(T_{i-1} = \sum_z r_i(z) - \sum_z r_{i-1}(z) \\ & = 1 + r_i(t) - r_i(u_{i-1}) + r_i(v_{i-1}) - r_{i-1}(t) - r_i(u_{i-1}) - r_{i-1}(v_{i-1}) \\ & = 1 + r_i(u_{i-1}) + r_i(v_{i-1}) - r_{i-1}(t) - r_{i-1}(u_{i-1}) \\ & \leq 2(r_{i-1}(v_{i-1}) - r_{i-1}(t)) \\ & \leq 3(r_{i-1}(v_{i-1}) - r_{i-1}(t)) \end{aligned} \tag{9.8}$$

  První nerovnost v odvození platí, protože $1 + r_i(u_{i-1}) + r_i(v_{i-1}) = 2r_{i-1}(t) = 2r_{i-1}(v_{i-1})$.
  Amortizovaná složitost během cyklu: $\leq 3(r_i(v_{i-1}) - r_{i-1}(t))$

- ad b2) :

  $1 + bal(T_i) - bal(T_{i-1}) = ... = 1 + r_i(u_{i-1}) + r_i(v_{i-1}) -_{i-1}(t) - r_{i-1}(u_{i-1}) \leq$

  - α
    Předpoklad: $r_{i-1}(t) < r_i(v_{i-1})$
    Pak platí:
    $\leq 1 + 2(r_{i-1}(v_{i-1}) - r_{i-1}(t)) \leq 3(r_i(v_{i-1}) - r_{i-1}(t))$
  - β
    Předpoklad: $r_{i-1}(t) = r_i(v_{i-1})$, $r_i(t) > r_i(v_{i-1})$
    $... = 1 + r_i(u_{i-1}) + r_i(v_{i-1}) - r_{i-1}(t) - r_{i-1}(u_{i-1}) \leq 2(2(r_{i-1}(v_{i-1}) - r_{i-1}(t)) \leq 3(r_i(v_{i-1}) - r_{i-1}(t))$

– γ
Předpoklad: $r_i(t) = r_i(v_{i-1}) - r_{i-1}(v_{i-1}) = r_{i-1}(t)$
Vím: $r_{i-1}(t) = r'(t) = r_{i-1}(v_{i-1}) = r'(u_{i-1}) = r_i(t) = r_i(v_{i-1}) = r'(v_{i-1})$ spor s lemmatem 9.2.2. $\Rightarrow$ případ γ nemůže nastat

Závěr pro b) : Amortizovaná složitost během cyklu $\leq 3(r_i(v_{i-1}) - r_{i-1}(t))$

Vždy platí $r_i(v_{i-1}) = r_i(t)$

$$\begin{aligned}&\sum_{i=1} k(1 + bal(T_i) - bal(T_{i-1}))\\ &\leq \sum_{i=1} k3(r_i(v_{i-1}) - r_{i-1}(t))\\ &= 1 + 3(r_{k-1}(v_{k-1}) - r_0(t)) = 1 + 3(r_o(T) - r_0(t))\end{aligned} \tag{9.9}$$

□

9.2.12 Amortizovaná složitost ostatních operací

Definice 9.2.1. Amortizovaná složitost operace $SPLAY(x, T) \leq 1+3(r(T)-r(t))$, kde t je prvek, který se přemístí do kořene.

Označme t_- prvek ve stromě T, který reprezentuje největší prvek $\leq x$.
Označme t_+ prvek ve stromě T, který reprezentuje nejmenší prvek $\geq x$.
Když x je reprezentováno T, pak $t_- - t_+$ je prvek reprezentující x.
Jednotlivé operace mají následující amortizované složitosti:

- $SPLAY(x, T) = O(\log \frac{w(T)}{min\{w(t_-), w(t_+)\}})$
- $MEMBER(x, T) = O(\log \frac{w(T)}{min\{w(t_-), w(t_+)\}})$
- $SPLIT(x, T) = O(\log \frac{w(T)}{min\{w(t_-), w(t_+)\}})$
- $CHANGEWEIGHT(x, \triangle) = O(\log \frac{w(T)-max\{\triangle,0\}}{min\{w(t_-), w(t_+)\}})$
- $JOIN3(T_1, x, T_2) = O(\log \frac{w(T_1)+w(T_2)+v(x)}{v(x)})$

Označme t_∞ prvek v T_1, který reprezentuje největší prvek z T_1. Pak amortizované složitosti pro zbývající operace jsou následujicí:

- $JOIN2(T_1, T_2) = O(\log \frac{w(T_1)+w(T_2)}{w(t_\infty)})$
- $DELETE(x, T) = O(\log \frac{w(T)}{min\{w(t_-), w(t_+), w(t_1\}})$

Prvek t_1 je prvek T_1, který reprezentuje v T největší prvek $\leq x$.

- $INSERT(x,T) = O(\log \frac{w(T)+v(x)}{min\{w(t_-),w(t_+)\}})$

Příklad 9.2.2. Mějme množinu $X = x_1, ..., x_n$ a pravděpodobnosti pro výskyt operace MEMBER(x). Nechť U je optimální binární vyhledávací strom. Nechť T je binární vyhledávací strom reprezentující X. P je posloupnost operací MEMBER(x) vyhovující daným pravděpodobnostem.

Chceme aplikovat P na strom T, kde pro implementaci použijeme strategii SPLAY stromů.

Srovnáme čas, který tato strategie vyžaduje s časem obvyklé implementace MEMBER při aplikaci P na U.

Definujeme $v(x) = 3^{d-d(x)}$, kde d je hloubka stromu U a $d(x)$ je hloubka prvku v U reprezentujícího prvek x.

Spočítáme totální váhu prvku x:

$$w(x) = \sum_{i=0} d - d(x)2^i 3^{d-d(x)-i} \leq 3^{d-d(x)} \sum_{i=0} d - d(x)(\frac{2}{3})^i \leq 3^{d-d(x)+1}$$

Pak platí:
$r(x) \leq d - d(x) + 1$
$r(T) \leq d + 1$, (prvek v kořeni má hloubku 0)

Amortizovaná složitost operace MEMBER(x)

$$\leq O(d(x)) = O(r(T) - r(x)) = O(d + 1 - d + d(x) - 1) = O(d(x))$$

Čas posloupnosti P použité na strom T a implementované strategií SPLAY

$$= (\sum_{\text{operace v P}} \text{amortizovaná složitost operací v P}) + bal(T) = O(\text{čas P pro strom U} + bal(T))$$

$bal(T)$ je balance stromu T.

$$bal(T) = \sum_{x \in X} r(x) = \sum_{x \in X} d + 1 = O(x^2)$$

$$\Rightarrow O(\text{čas P pro strom U}) + bal(T) = O(\text{čas P pro U} + x^2)$$

tady bylo ve vzorci něco jako $|x^2|$ (?)

Závěr: pro dlouhé posloupnosti snad téměř stejné jako opt. BVS.

Kapitola 10

Haldy

Definice 10.0.2. Haldy jsou stromové struktury, které splňují

- lokální podmínku na uspořádání - prvek reprezentující otce je menší než prvek reprezentovaný synem apod.
- strukturální podmínku na stromy, ze kterých jsou vytvořené

Poznámka 10.0.1. Podle těchto podmínek se haldy rozdělují na Fibonacciho, Leftist, d-regulární apod. (mohou se lišit jak lokální, tak strukturální podmínkou)

10.1 d-regulární haldy

Definice 10.1.1. d-regulární halda, d celé číslo $d \geq 2$
Je to strom T takový, že existuje jednoznačná korespondence mezi vrcholy stromů a prvky reprezentované množiny a platí:

1. strom T splňuje strukturální podmínky:

- každý vrchol s vyjímkou nejvýše jednoho je buď list nebo má d synů
- každý vrchol má nejvýše d synů
- existuje očíslování synů každého vrcholu tak, že po očíslování průchodem šířky platí: když vrchol není list, pak každý vrchol s menším číslem má d synů.

2. podmínku na lokální uspořádání:
když x je prvek přiřazený vrcholu t, pak otci(t) je přiřazen prvek $\leq x$ pak po očíslování průchodem do šířky platí: když vrchol má číslo i, jeho synové mají čísla $d(i-1)+2, d(i-1)+3, ..., di+1$ a otec má číslo $\lceil \frac{i-1}{d} \rceil$.

Příklad 10.1.1. Příklad 3-regulární haldy je na obrázku **??**. XXX chybi obr.

Když takto očíslované prvky dáme do pole, pak platí: když je vrchol na i-tém místě, čísla synů jsou $3(i-1)+2, 3i, 3i+1$ a otec je na $\lceil \frac{i-1}{3} \rceil$ místě v poli. to využijeme pro implementaci polem - ušetříme místo.

Poznámka 10.1.1. Nejpopulárnější jsou 2-reg. haldy, protože synové i-tého vrcholu jsou na místech $2(i-1)+2 = 2i, 2(i-1)+3 = 2i+1$, otec je na $\lceil \frac{i-1}{2} \rceil + 1 = \lceil \frac{i}{2} \rceil$. $\Rightarrow$ snadné počítání (bitový posun)

10.1.1 Algoritmus UP

Operace UP(x) srovná haldu směrem nahoru.

Algoritmus 10.1 UP pro d-regulární haldy

A:
if prvek reprezentovaný x je $<$ prvek reprezentovaný otcem(x) **then**
x a otce(x) vyměníme
pokračujeme v A
end if

10.1.2 Algoritmus DOWN

Algoritmus 10.2 DOWN pro d-regulární haldy

A:
if prvek reprezentovaný $x >$ prvek reprezentovaný některým synem x **then**
vyměníme x a syna x, který reprezentuje nejmenší prvek,
pokračujeme v A
end if

Poznámka 10.1.2. Když má hlada hloubku h, pak UP(x) vyžaduje čas $O(h)$, DOWN(x) čas $O(dh)$.

10.1.3 Operace na haldě

INSERT

Algoritmus 10.3 INSERT pro d-regulární haldy

přidáme poslední list t reprezentující x
UP(t)

MIN

Algoritmus 10.4 MIN pro d-regulární haldy

vrátí prvek reprezentovaný v kořeni

DELETEMIN

viz algoritmus 10.5.

DECREASEKEY(x, Δ)

Provedení této operace předpokládá, že musíme znát polohu vrcholu t reprezentujícího x, toto halda neumožňuje nalézt.

viz algoritmus 10.6.

Algoritmus 10.5 DELETEMIN pro d-regulární haldy

prvek reprezentovaný posledním listem dáme do kořene
odstraníme poslední list
DOWN(kořen)

Algoritmus 10.6 DECREASEKEY pro d-regulární haldy

změníme uspořádání v bodě x
UP(x) mohl by být menší než jeho otec, proto provedeme UP

INCREASEKEY(x, Δ)

Musíme znát polohu vrcholu t reprezentujícího x, toto halda neumožňuje nalézt. viz algoritmus 10.7.

Algoritmus 10.7 INCREASEKEY pro d-regulární haldy

změníme uspořádání v bodě x
DOWN(x)

DELETE

Musíme znát polohu vrcholu t reprezentujícího x, toto halda neumožňuje nalézt.

Vezmeme prvek y reprezentovaný posledním listem, odstraníme poslední list, prvek t, který reprezentoval x bude reprezentovat y.

Algoritmus 10.8 DELETE pro d-regulární haldy

if $y < x$ **then**
UP(t) else DOWN(t)
end if

10.1.4 Algoritmus MAKEHEAP

Dána prostá posloupnost $x_1, x_2, ..., x_n$. Chceme vytvořit d-reg. haldu reprezentující množinu $x_1, x_2, ..., x_n$. Vezmeme "d-reg. strom" T s vrcholy přiřadíme prvky $x_1, x_2, ..., x_n$. Pro všechny vrcholy, které nejsou listy podle očíslování v pořadí od největšího k nejmenšímu provedeme DOWN(t). chybí obrázek

Invariant: v okamžiku, kdy provádím DOWN(t), tak vrcholy, které reprezentující větší prvky splňují směrem dolů podmínku

10.1.5 Složitost operací

V d-reg. haldě reprezentující n-prvkovou množinu implementace operací vyžaduje časy dané tabulkou:

Operace	Složitost
MIN	O(1)
INSERT, DECREASEKEY	$O(log_d(n))$
DELETEMIN, INCREASEKEY, DELETE	$O(d \cdot log_d(n))$

Máme vrchol v i-té hladině a "d-reg. strom" má hloubku h. Kolik času potřebuje DOWN(t) ? Je to $O(d(h-1))$.

Počet vrcholů v i-té hladině je di.
Čas MAKEHEAP je $O(\sum i = 0h-1 d^i d(h-i)) = O(dS)$, kde

$$S = \sum i = 0h-1 d^i (h-i)$$

Budeme počítat

$$dS - S = \sum i = 0h-1 d^{i+1}(h-i) - \sum i = 0h-1 d^i (h-i) =$$

$$d^h - h + \sum i = 0h-1 d^i (h-i-(h-i-1)) = d^h - h\frac{d^h-1}{d-1}$$

$$\Rightarrow S = \frac{d^h-h}{d-1} + d\frac{d^{h-1}-1}{(d-1)^2}, h = log_d(n) \Rightarrow S \approx O(\frac{n}{d}) \quad (10.1)$$

10.1.6 Dijkstrův algoritmus

K čemu jsou d-reg. haldy dobré ? např. pro implementaci Dijkstrova algoritmu.

Vstup: orientovaný graf (V, E), fce $c : E \to R^+$, vrchol z

Výstup: $d(v)$, $v \in V$
$d(v)$ je délka nejkratší cesty ze z do v

Algoritmus 10.9 Dijkstrův algoritmus pro d-regulární haldy

```
d(z) = 0, d(v) = ∞ ∀v ∈ V, v ≠ z, U = z
while U ≠ ∅ do
  vezmeme z U prvek u ∈ U s nejmenší hodnotou d(u),
  odstraníme ho z U.
  for ∀(u, v) ∈ E do
    if d(v) > d(u) + c(u, v) then
      d(v) = d(u) + c(u, v) , v přidáme do U
    end if
  end for
end while
```

U reprezentujeme pomocí d-reg. haldy. Pak čas Dijkstrova algoritmu je

$$O(|V| \cdot \text{čas na INSERT } + |V| \cdot \text{čas na DELETEMIN } + |E| \cdot \text{čas na DESCREASEKEY})$$

Když $d = 2$, pak to je $O(|E| log_2(|V|))$
$d = max\frac{|E|}{|V|}, 2$, vyjde čas $O(|E| log_d(|V|))$
Když $\exists \epsilon$, že $|E| \geq c|V|^{1+\epsilon}$ pro nějaké c, pak čas je $O(|E|)$. (graf je dostatečně hustý)
$|E| \geq c|V| \log^{\epsilon} |V|$ pro nějaké c, ϵ, pak čas je $O(|E| \log log |V|)$.

10.1.7 Heapsort

Třídící algoritmus Heapsort je další aplikací d-regulárních hald.
HEAPSORT - viz alg. 10.10

Vstup: prostá posloupnost prvků $x_1, x_2, ..., x_n$

Výstup: uspořádaná psl. prvků $x_1, x_2, ..., x_n$

Algoritmus 10.10 Heapsort pro d-regulární haldy

```
MAKEHEAP(x_1, x_2, ..., x_n)
i = 1
while HEAP ≠ ∅ do
  x_1 = MIN(HEAP)
  DELETEMIN(HEAP)
  i = i + 1
end while
```

Poznámka 10.1.3. Optimum pro d-reg. haldy je někde mezi $d = 6$ a $d = 7$.

10.2 Leftist haldy

Definice 10.2.1. Mějme binární strom a pro každého syna máme určeno, zda je levý nebo pravý. Pro vrchol v definujeme npl(v) jako délku nejkratší cesty z v do vrcholu v podstromu v s nejvýše jedním synem.

Binární strom je LEFTIST, když

a) když vrchol v má jednoho syna, pak je to levý syn

b) když vrchol v má dva syny, pak npl(levého syna) $\geq npl$(pravého syna)

Definice 10.2.2. Cesta $x_1, x_2, ..., x_n$ se nazývá *pravá*, když x_i je pravý syn x_{i-1} pro $i = 2, 3, ..., n$ a x_n nemá pravého syna.

Vlastnosti:

1. každý podstrom leftist stromu je leftist

2. délka pravé cesty z $\forall$ vrcholu v je $\leq log$(počet vrcholů v podstromu vrcholu v)

Definice 10.2.3. Letist halda reprezentující množinu S je leftist strom T s n vrcholy takový, že existuje jednoznačná korespondence mez prvky S a vrcholy T taková, že $\forall$ prvek přiřazený vrcholu $v \geq$ prvek přiřazený otci v.

10.2.1 MERGE

Operace MERGE s argumenty T_1, T_2 předpokládá, že T_1, T_2 reprezentují disjunktní množiny S_1, S_2. Výsledkem této operace je halda reprezentující $S_1 \cup S_2$.

Formální zápis viz algoritmus 10.11

Poznámka 10.2.1. Časová složitost operace MERGE v leftist haldách je $O(log(n_1 + n_2))$, kde n_1, n_2 jsou velikosti reprezentovaných množin.

Algoritmus 10.11 MERGE pro leftist haldy

MERGE(T_1, T_2)
if $T_1 = 0$ **then**
  MERGE(T_1, T_2) $\rightarrow T_2$ konec
end if
if $T_2 = 0$ **then**
  MERGE(T_1, T_2) $\rightarrow T_1$ konec
end if
if kořen T_2 reprezentuje prvek < prvek repr. kořenem T_1 **then**
  vyměníme T_1 a T_2
end if
pravý syn kořene $T_1 \rightarrow$ MERGE(T_2, podstrom pravého syna kořene T_1)
if npl(levého syna kořene T_1) < npl(pravého syna kořene T_1) **then**
  prohodím syny kořene T_1
end if
npl(kořene T_1) = npl(pravého syna kořene T_1) + 1
MERGE(T_1, T_2) $\rightarrow T_1$

10.2.2 INSERT

viz algoritmus 10.12

Algoritmus 10.12 INSERT pro leftist haldy

INSERT(x)
vytvoříme novou haldu T_1 reprezentující pouze prvek x
T $\leftarrow$ MERGE(T_1, T_2)
DELETEMIN
$T_1 \leftarrow$ podstrom levého syna kořene T
$T_2 \leftarrow$ podstrom pravého syna kořene T
$T \leftarrow$ MERGE(T_1, T_2)

Věta 10.2.1. *Operace MIN v leftist haldách vyžaduje čas $O(1)$, operace MERGE, INSERT, a DELETEMIN vyžadují čas $O(logn)$, kde n je počet prvků ve výsledné haldě.*

XXX obr.

Poznámka 10.2.2. Podíváme se jak vypadá výsledný strom a podíváme se na vrcholy, se kterými jsme něco museli provádět - tyto vrcholy leží na *p*ravé cestě, tj. je jich omezený počet.

10.2.3 DECREASEKEY

viz algoritmus 10.13

Poznámka 10.2.3. *npl*, které jsem musel přepisovat, je vždycky pravý syn.

Věta 10.2.2. *Operace DECREASEKEY, INCREASEKEY a DELETE vyžadují v leftist haldách čas $O(logn)$. (n je počet prvků výsledné reprez. množiny)*

Algoritmus 10.13 DECREASEKEY pro leftist haldy

DECREASEKEY(x)
odtrhneme podstrom T_1 vrcholu x, $y \rightarrow$ otec(x)
$T_2 = T - T_1$
zmenšíme ohodnocení kořene stromu T_1
if y má jen pravého syna **then**
  změníme tohoto syna na levého, $npl(y) = 0$
end if
$y \rightarrow otec(y)$
while $npl(y) > min\{npl(\text{levý syn } y), npl(\text{pravý syn } y)\} + 1$ **do**
  if $npl(\text{levého syna } y) < npl(\text{pravého syna } y)$ **then**
    prohodíme syny y
  end if
  $npl(y) = npl(\text{pravého syna } y) + 1$, $y \rightarrow otec(y)$
end while
$T \leftarrow$ MERGE(T_1, T_2)

10.3 Binomiální haldy

Definice 10.3.1. *Binomiální strom* B_i je rekurzivně definován jako strom sestávající se z kořene a jeho dětí $B_0, B_1, ..., B_{i-1}$. Každý strom má *vlastnost haldy*, tj. pro každou stromovou hranu platí klíč otce $\leq$ klíč syna.

Definice 10.3.2. *Binomiální halda* je soubor stromů takových, že

- každý strom je izomorfní s nějakým B_i
- žádné dva stromy nejsou izomorfní
- existuje jednoznačná korespondence mezi vrcholy reprezentované množiny a vrcholy stromů taková, že prvek odpovídající otci je menší než prvek odpovídající vrcholu.

Poznámka 10.3.1. Nejčastěji je binom. halda implementována jako pole ukazatelů, kde i-tý ukazatel ukazuje na kořen stromu B_i nebo je NIL. To, jak dlouhé pole budeme potřebovat, je kardinální pro amortizovanu složitost. Binární zápis čísla n má délku $\lfloor log_2 n \rfloor \Rightarrow$ stromy řádu vyššího než $\lfloor log_2 n \rfloor$ se nebudou vyskytovat. (jinak by měl graf více než n vrcholů)

Binomiální stromy rostou exponenciálně spolu s řádem. (proto funguje amort. analýza)

Poznámka 10.3.2. Na binomiální strom se můžeme dívat i jinak: strom B_i sestává ze 2 kopií B_{i-1} (viz obr. 10.1) a získá se z nich operací zvanou *spojení*. Binomiální haldy souvisí s binomiálním rozvojem čísel.

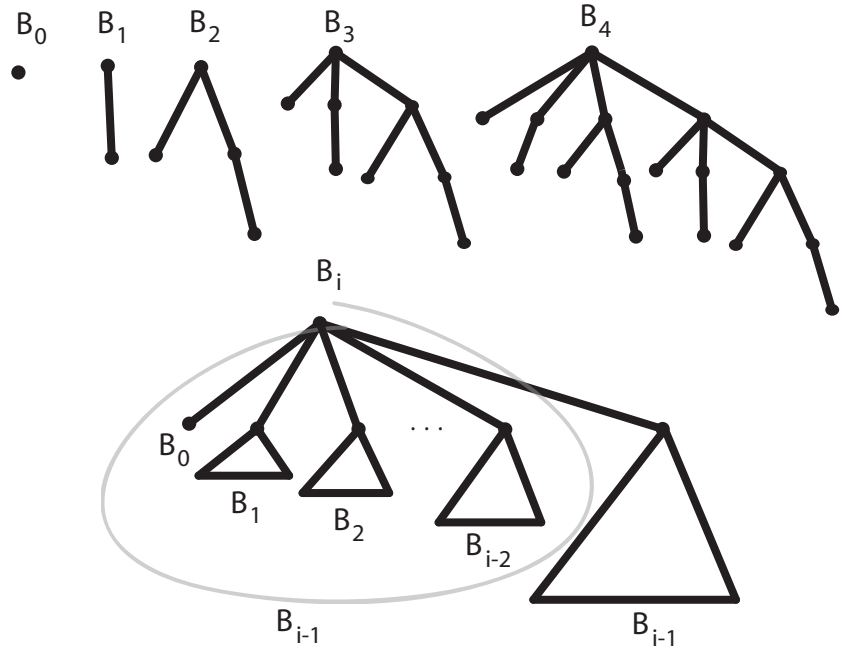
Tvrzení 10.3.1. *Binomiální halda je tvořena binomiálními stromy B_i, které mají následující vlastnosti:*

- *B_i má 2^i vrcholů*
- *hloubka B_i je i*
- *kořen B_i má i synů*
- *$\forall j < i$ existuje syn kořene B_i takový, že jeho podstrom je izomorfní s B_j.*

Důkaz. indukcí přes i (elementární) □

Algoritmus 10.14 Spojení dvou binomiálních stromů

Spojeni(S_1, S_2)
S_1, S_2 jsou stromy izomorfní s B_i pro nějaké i
if prvek reprezentovaný kořenem $S_1 \leq$ prvek reprezentovaný kořenem S_2 **then**
  kořen S_2 se stane dalším synem kořene S_1
else
  kořen S_1 se stane dalším synem kořene S_2
end if



Obrázek 10.1: Binomiální stromy

10.3.1 MERGE

Algoritmus MERGE (viz algoritmus 10.15) pracuje jako "binární sčítání" - 2 stromy B_i (= 2 jedničky v řádu i) slije do $Bi-1$ (= přenos do $i+1$)

Pracuje v $O(\log_2 n)$ - nejvyšší možný řád je $\lfloor \log_2 n \rfloor$. Toto je složitost v nejhorším případě.

Ukazatel MIN nové haldy je nastaven na menší z MIN(h_1), MIN(h_2) - to zabere $O(1)$.

Poznámka 10.3.3. V algoritmu MERGE (viz algoritmus 10.15) odpovídá P přenosu v binárním sčítání, T je výsledná halda.

10.3.2 MIN

MIN(h) - prohledáme prvky reprezentované kořeny stromů a najdeme nejmenší. V praxi je pro každou haldu držen ukazatel, ukazující na kořen reprezentující nejmenší prvek haldy. Tento ukazatel je obnovován při operaci DELETE_MIN.

Algoritmus 10.15 MERGE pro binomiální haldy

MERGE(T_1, T_2)
(T_1, T_2 binom. haldy velikosti n_1, n_2)
$P = 0, i = 0, T = 0$
while $i \leq log(n_1 + n_2)$ **do**
  S_1 je strom v T_1 izomorfní s H_i (pokud neexistuje, tak $S_1 = 0$)
  S_1 je strom v T_1 izomorfní s H_i (pokud neexistuje, tak $S_1 = 0$)
  if S_1, S_2, P $= 0$ **then**
    neprovedeme nic
  end if
  if jeden strom z S_1, S_2, P je neprázdný **then**
    vložím tento strom do T, $P = 0$
  end if
  if dva stromy z S_1, S_2, P jsou neprázdné **then**
    spojím tyto stromy a výsledek vložím do P
  end if
  if všechny stromy z S_1, S_2, P jsou neprázdné **then**
    vložím do T, spojení S_1, S_2 vložím do P
  end if
  $i = i + 1$
end while

10.3.3 INSERT

Operace INSERT(h,i) se provede příkazem MERGE(h, MAKEHEAP(i)). Tato operace je analogická s inkrementací binárního čítače.

Dijkstrův algoritmus provádí na začátku n operací INSERT, nám tedy nejde o jednotlivé operace, ale o posloupnost INSERTů.

Poznámka 10.3.4. INSERT je stejný jako v leftist haldách.

Věta 10.3.1. *Amortizovaná složitost operace INSERT je* $O(1)$.

Důkaz. Využijeme účetní metody:
Algoritmus INSERT udržuje následující invariant:
Každý binom. strom v haldě má na svém účtu 1 jednotku. (Ten, který přestává být kořenem, zaplatí, ten kdo vyhrál, si 1 jednotku ponechal.) Při vytváření stromu ji zaplatí operace, která strom vytvořila:

- MAKEHEAP vytvoří 1 strom $\Rightarrow$ zaplatí 1
- DELETE_MIN vytvoří $\leq logn$ stromů $\Rightarrow$ zaplatí $\leq logn$

Pokud INSERT spustí kaskádu slévání, pak je každé slití zaplaceno z účtu stromu, který daným slitím zanikne. (jeho kořen se stane synem) □

10.3.4 DELETEMIN

Operace DELETEMIN (viz algoritmus 10.16) je provedena tak, že ze stromu B_k, na který ukazuje ukazatel MIN, utrhneme kořen. Tím vzniknou nové stromy $B_0, B_1, ..., B_{k-1}$, ze kterých vytvoříme novou haldu, nastavíme pro ni ukazatel MIN a zavoláme MERGE.

DELETEMIN pracuje v $O(log_2 n)$, protože $k \leq log_2 n$. Toto je složitost v nejhorším případě.

Algoritmus 10.16 DELETEMIN pro binom. haldu

DELETEMIN
prohledáním prvků reprezentovaných kořeny stromů naleznu strom S, jehož kořen reprezentuje nejmenší prvek
$T_1 = T\ S, T_2$ je tvořen podstormy všechn synů kořene S
(tj. utrhnu kořen a zbytek dám do haldy) - je to halda díky vlastnosti 4
$T \rightarrow \mathrm{MERGE}(T_1, T_2)$

Poznámka 10.3.5. Operace DELETE se nedá rozumně provést, museli bychom přebudovat celý strom.

Věta 10.3.2. *Operace MERGE, INSERT, MIN, DELETEMIN a DECREASEKEY vyžadují čas* $O(logn)$. *Operace INCREASEKEY vyžaduje čas* $O(log^2 n)$.

Poznámka 10.3.6. MERGE zabírá dost času - musíme ho dělat ?

10.3.5 Líná implementace binom. hald

Líná implementace vychází z toho, že chceme operaci MERGE provádět v čase $O(1)$.

Změníme definici - vynecháme podmínku 2 z definice 10.3.2, tj. teď v naší binom. haldě mohou být izomorfní stromy. (i když jen dočasně) Další změna spočívá ve změně reprezentace binomiální haldy - haldu reprezentujeme dvojitým kruhovým spojovým seznamem přes kořeny stromů. (kruhový spojový seznam umožňuje přidávání a odebírná prvků v čase $O(1)$.)

Operaci $\mathrm{MERGE}(T_1, T_2)$ pak můžeme provést konkatenací seznamů T_1 a T_2. Jenom to by nefungovalo, musíme ještě změnit operace MIN, DELETEMIN.

Algoritmus 10.17 DELETEMIN pro líné binom. haldy

MIN
při prohledávání prvků reprezentovaných kořeny stromů seřadíme stromy do množin Q_i, $i = 0, ..., n$, kde Q_i je množina všech stromů v T izomorfních s B_i.
$i = 0, T = 0$
while $\exists Q_i \neq 0$ **do**
  while $|Q_i| > 1$ **do**
    vezmeme dva stromy z Q_i, spojíme je, výsledek dáme do Q_{i+1}
  end while
  if $Q_i \neq 0$ **then**
    strom z Q_i dám do T
  end if
  $i = i + 1$
end while

DELETEMIN umístí stromy po odtržení nejmenšího prvku do odpovídajících množin Q_i. (v množině Q_i jsou stromy izomorfní s B_i) Poté provede *konsolidaci* - upraví haldu do podoby, kdy je každý řád zastoupen nejvýše jedním stromem.

Konsolidace běží v $O(logn)$ plus vyčerpá účty stromů, které zaniknou při slévání.

Konsolidace probíhá takto:

1. vytvořím pole délky $logn$, které je prázdné $\Rightarrow O(logn)$

2. procházím spojový seznam vrcholů stromů v haldě a jeden strom za druhým vyjmu a dávám do pole vytvořeného v kroku 1, přičemž se vždy provede příslušné slití.

- pokud strom zanikne, tak práci zaplatíme z jeho účtu
- pokud strom nezanikne, tak práci platíme z účtu konsolidace $\rightarrow O(logn)$

3. z pole vytvoříme spojový seznam $\rightarrow O(logn)$

DELETEMIN tedy potřebuje

- $O(logn)$ do účtů nově vytvořených stromů
- $O(logn)$ na jejich zavedení do spojového seznamu
- $O(logn)$ na konsolidaci

Při konsolidaci vždy zároveň vyhledáme nové minimum.

Věta 10.3.3. *Operace MERGE a INSERT při líné implementaci vyžadují čas* $O(1)$*, operace DELETEMIN a MIN vyžadují čas* $O($*počet stromů v haldě*$)$*.*

Operace	Amort. složitost
MERGE	$O(1)$
INSERT	$O(1)$
MIN	$O(logn)$
DELETEMIN	$O(logn)$

Tabulka 10.1: Amortizovaná složitost pro líné binomiální haldy

10.4 Fibonacciho haldy

Fibonacciho haldy vycházejí z binomiálních hald, formálně se liší v podstatě pouze tím, že v haldě povolíme i jiné stromy než binomiální. Toto nám umožní implementaci operace DECREASE_KEY, která nebyla v binomiálních haldách při zachování složitosti ostatních operací možná.

Řád uzlu a stromu je definován jako u binomiálních hald. Slévání se provádí pouze mezi stromy stejného řádu.

10.4.1 MERGE, INSERT, EXTRACT_MIN

Implementace operací MERGE(h_1,h_2), INSERT(h,i), EXTRACT_MIN(h) je stejná jako u binomiálních hald v "líné" verzi.

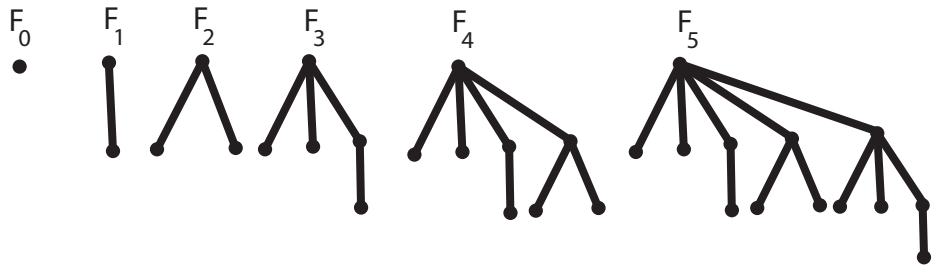
Algoritmus 10.18 DECREASE_KEY pro Fibonacciho haldy

DECREASE_KEY(h,i,δ)

1. snížím klíč prvku i o δ

2. podstrom i s kořenem i odřízneme a jako samostatný strom ho zavedeme haldy (tj. zařadím do spojového seznamu kořenů stromů v haldě) $\Rightarrow O(1)$

3. Abychom udrželi stromy dostatečně "košaté"[1] tak od každého vrcholu x mohou být odříznuti nejvýše 2 synové $\Rightarrow$ po odříznutí 2. syna je odříznut i sám vrchol x.



Obrázek 10.2: Počty vrcholů stromů $F_0, F_1, ...$ tvoří Fibonacciho posloupnost.

10.4.2 DECREASE_KEY

DECREASE_KEY provádí snížení hodnoty klíče pro daný prvek. To se děje za cenu přítomnosti jiných než binomiálních stromů v haldě.

Poznámka 10.4.1. Přestože jedna operace DECREASE_KEY může vyvolat kaskádu řezů, je její amortizovaná složitost $O(1)$.

Poznámka 10.4.2. Pomocí *účetní metody*[2] dokážeme, že to platí:
Při odřezávání syna vrcholu x zaplatí operace DECREASE_KEY

- 2 jednotky na účet x
- 1 jednotku na účet vzniklého stromu
- 1 jednotku za práci (odříznutí a zařazení)

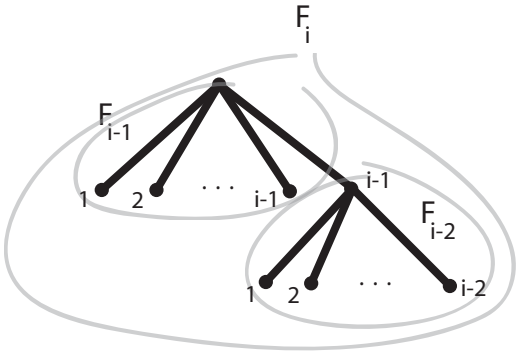
Při odříznutí druhého syna jsou na účtu vrcholu x 4 jednotky $\Rightarrow$ mohu zopakovat body 1) - 3).

Věta 10.4.1. *Nejvyšší řád stromu ve Fibonacciho haldě je* $\lfloor log_\varphi n \rfloor = \Theta(log_2 n)$ *pro nějaké* $\varphi > 1$.

Lemma 10.4.1. *Nechť x je vrchol a $y_1, ..., y_m$ jeho synové v pořadí, v jakém byli k x sliti. Potom $\forall \in 1, ..., m$ je řád y_i aspoň $i - 2$.*

Důkaz. V okamžiku, kdy byl y_i slit pod x, měl x řád $\geq i - 1$. ($y_1, ..., y_{i-1}$ již v té chvíli byli synové x) V tomto okamžiku byl také řád $y_{i-1} \geq i - 1$. (sléváme pouze stromy stejného řádu) Od té doby mohl y_i ztratit nejvýše jednoho syna, jinak by byl sám odříznut a přestal by být synem x. $\Rightarrow$ y_i má řád $\geq i - 2$. □

[2]Pro definici účetní metody viz přednášky ze "*Složitosti a NP úplnosti*".



Obrázek 10.3: K důkazu věty 10.4.1

Důkaz. Dokazujeme větu 10.4.1, která jinými slovy říká: Strom řádu i ve Fibonacciho haldě má velikost alespoň φ^i pro nějaké $\varphi > 1$.

Nechť F_j je nejmenší možný (tj. má ořezané podstromy na max. možnou úroveň - byl z nich odříznut 1 syn) strom řádu j splňující tvrzení lemma 10.4.1 a nechť $|F_j| = f_j$. Pak

1. F_i vznikne "slitím" F_{i-1} a $F_{i-2} \Rightarrow f_i = f_{i-1} + f_{i-2}$

2. $f_i \geq \varphi^i$, kde $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ ... zlatý řez

ad 1) viz obr. 10.3

Slití je nepřesný termín - sléváme pouze stromy stejného řádu. F_{i-2} je výsledek DECREASE_KEY. (tím se strom "oholil") Uříznu posledního syna, pod kterým je největší podstrom (abych dostal nejmenší možný podstrom)

ad 2) φ je kladý kořen rovnice $x^2 - x - 1 = 0$
neboli platí $\varphi^2 = \varphi + 1$, $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$
dokážeme indukcí:

- i = 0 : $f_0 = 1 \geq \varphi^0 = 1$
- i = 1 : $f_1 = 2 \geq \varphi^1 = 1.618$
- indukční krok: IP: $f_i \geq \varphi^i$, $f_{i+1} \geq \varphi^{i+1}$
  $f_{i+2} = f_{i+1} + f_i \geq \varphi^{i+1} + \varphi^i = \varphi^i(\varphi + 1) = \varphi^{i+2}$

□

Kapitola 11

Dynamizace

V uspořádaném poli umíme rychle vyhledávat, ale přidat prvky znamená celé ho přebudovat. Ve srůstajícím hašování zase nešly prvky mazat, ve velmi komprimovaných trie ani přidávat, ani mazat. V této kapitole ukážeme obecnou metodu, jak tyto problémy řešit, podobnou přístupu u binomiálních hald.

Takové struktuře, která neumožňuje vkládání (operace INSERT) ani mazání (operace DELETE) prvků budeme říkat *statická struktura*. Chceme vytvořit takovou reprezentaci, která bude využívat výhod statické struktury, ale zároveň umožní operace INSERT a DELETE.

K tomu se dopracujeme postupně. Nejdříve provedeme *semidynamizaci*, která umožní (v nové reprezentaci původní množiny) operaci INSERT, pak *dynamizaci*, která přidá operaci DELETE.

11.1 Zobecněný vyhledávací problém

Definice 11.1.1. *Vyhledávací problém* je funkce $f : U_1 \times 2^{U_2} \to U_3$, kde U_1, U_2 a U_3 jsou univerza.

Definice 11.1.2. *Řešení vyhledávacího problému* pro $x \in U_1, A \subseteq U_2$ je nalezení hodnoty $f(x, A)$.

Poznámka 11.1.1. Chceme najít strukturu, která reprezentuje A a algoritmus, který pro vstup $x \in U_1$ spočítá $f(x, A)$. Takové struktuře se říká *statická struktura* pro vyhledávací problém.

Příklad 11.1.1. Klasický vyhledávací problém: $U_1 = U_2 = U$, univerzum prvků;
$U_3 = \{0, 1\}, A \subseteq U_2$

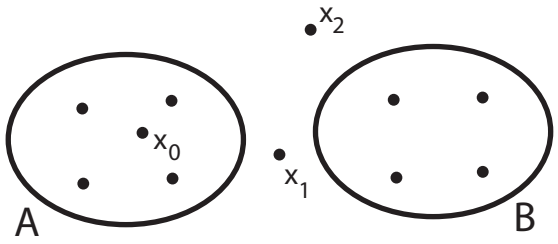
$$f(x, A) = \begin{cases} 0 & \text{když } x \notin A \\ 1 & \text{když } x \in A \end{cases} \text{ (rozložitelný)}$$

Euklidovská vzdálenost bodů v rovině: $U_1 = U_2 =$ euklidovská rovina; $U_3 = \mathbb{R}^+$; $f(x, A) = dist(x, A$ vzdálenost bodu $x \in U_1$ od množiny A. (rozložitelný, $\oplus$ ... operace min)

Nalezení předchůdce $U_1 = U_2 = U_3$ pro $x \in U_1$ a $A \subseteq U_!$ a je $f(x, U_!)$ je největší prvek $A \leq x$ (rozložitelný, je potřeba disjunkce)

Příslušnost ke konvexnímu obalu $U_1 = U_2 =$ rovina; $U_3 = \{0, 1\}$;

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



Obrázek 11.1: Konvexní obal

11.1.1 Operace INSERT a DELETE

Pro množinu $A \subseteq U_2$ a pro statickou strukturu S řešící vyhledávací problémf pro $x \in U_2$.

- INSERT(x,A) - vybudování struktury řešící vyhledávací problém pro množinu $A \cup \{x\}$
- DELETE(x,A) - vytvoření struktury řešící vyhl. problém pro množinu $A - \{x\}$

Poznámka 11.1.2. Ze statické struktury chce vytovořit dynamickou (dynamizace). INSERT je obvykle jednodušší než DELETE, na ten budeme potřebovat dodatečné předpoklady.

Nároky na dynamizaci

- chceme aby se $f(x, A)$ v nové struktuře spočítalo přibližně stejně rychle jako v původní struktuře
- když vytvoření původní struktury pro n prvnkovou množinu trvalo t, pak operace INSERT by přibližně měla vyžadovat čas t/n.

Definice 11.1.3. Vyhledávací problém je *rozložitelný*, když existuje operace $\oplus$ spočitatelná v konstantním čase a platí: když $x \in U_1$ a A a B jsou disjunktní podmnožiny U_2, pak

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

Poznámka 11.1.3. Z výše uvedených příkladů není rozložitelným problémem příslušnost ke konvexnímu obalu, ostatní vyhledávací problémy jsou rozložitelné.

Definice 11.1.4. Nechť f je rozložitelný vyhledávací problém a $\mathcal{S}$ je "statická" datová struktura, která ho řeší. Neboli $\mathcal{S}$ je tvořena pro pevnou množinu $A \subseteq U_2$ a obsahuje operaci, která pro vstup x počítá $f(x, A)$.

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

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

11.2 Semi-dynamizace

Semi-dynamizace umožní operaci INSERT nad novou reprezentací původní množiny. Tato reprezentace bude využívat statické struktury. Nejprve provedeme "základní" semidynamizaci, poté ji vylepšíme pro INSERT se složitostí v nejhorším případě. Vylepšení bude vyžadovat jiný rozklad původní množiny a algoritmus INSERT (viz algoritmus 11.2) bude složitější.

Věta 11.2.1. *Máme rozložitelný vyhled. problém f a máme pro něj statickou strukturu, která ho řeší v čase $Q(n)$, vyžaduje $S(n)$ paměti a vytvoří se v čase $P(n)$, kde $Q(n), \frac{P(n)}{n}, \frac{S(n)}{n}$ jsou neklesající funkce. Pak existuje semidynamická dat. struktura D, řešící f v čase $O(Q(n)\log n)$ vyžadující $O(S(n))$ paměti a umožňující INSERT s amort. složitostí $O(\frac{P(n)}{n} \cdot \log n)$.*

Důkaz. Budeme předpokládat, že $Q_{\mathcal{S}}(n)$, $S_{\mathcal{S}}(n)/n$ a $P_{\mathcal{S}}(n)/n$ jsou neklesající funkce.

Máme množinu A a vytvoříme pro ni novou strukturu D. Nechť $A_i \subseteq A$ taková, že buď $|A_i| = 2^i$ nebo $A_i = \emptyset$
$A_i \cap A_j = \emptyset proi \neq j$. $\bigcup A_i = A$

Platí $A_i \neq \emptyset$ právě když $(i+1)$-ní bit v dvojkovém rozvoji čísla $|A|$ je 1.

Chceme navrhnout strukturu, která by uměla

1. Pro $x \in U_1$ a pevné $A \subseteq U_2$ rychle spočítat $f(x, A)$.
2. Pro A a $y \in U_2$ rychle vytvořit strukturu pro $A \cup \{y\}$.

Mějme $A_0, A_1, \cdots$ 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

- nějaká dynamická struktura reprezentující A (např. (a,b)-strom, červeno-černý strom, AVL-strom)
- Pro každé $A_i \neq \emptyset$ máme S strukturu reprezentující A_i.
- Pro každé $A_i \neq \emptyset$ seznam prvků v A_i; prvky těchto seznamů jsou projpojeny s odpovídajícími prvky ve stromě.

Jak v nové struktuře spočítáme $f(x, A)$?
Pro každou $A_i \neq \emptyset$ spočítáme $f(x, A_i)$ a pomocí operace $\oplus$ pak spočítáme $f(x, A)$.

Poznámka 11.2.1. Platí, že když $A_i \neq \emptyset$, pak $i \leq \lceil log_2|A| \rceil$ čas, který je potřeba v nové struktuře na výpočet $f(x, A)$

XXX jak ma vypadat tento vzorec ?

$$log_2|A| \quad + \quad \sum_{i=0}^{log|A|} Q(2^i) \quad \leq \quad log|A| \quad + \quad \sum_{i=0}^{log|A|} Q(|A|) \quad = \quad log|A|(Q(|A|) \quad + \quad 1) \quad (11.1)$$

Poznámka 11.2.2. První nerovnost plyne z toho, že $Q(n)$ je nerostoucí funkce. V dalších důkazech pro S a P se využívá opět této vlastnosti pro $\frac{S(n)}{n}$ a $\frac{P(n)}{n}$.

$log_2(|A|)$ - vyhodnocení $f(x, A)$ z $f(x, A_i), i = 0, 1, ...$

Tedy algoritmus potřebuje $O(log|A|Q(|A|))$ času když $Q(n) = \Theta(n^\epsilon) pro \epsilon > 0$, pak platí, že nová struktura pro výpočet f potřebuje

$$\begin{aligned} & log|A| + \sum_{i=0}^{logn} Q(2^i) \\ & = |A| + \sum_{i=0}^{log|A|} \frac{S(2^i)}{2^i} 2^i \leq |A| + \sum_{i=0}^{log|A|} \frac{S(|A|)}{|A|} 2^i \\ & = |A| - \frac{S(|A|)}{|A|} 2^i = |A| - \frac{S(|A|)}{|A|} (\sum_{i=0}^{log|A|} 2^i) \\ & = O(S(|A|)) \end{aligned} \tag{11.2}$$

11.2.1 INSERT

Algoritmus 11.1 INSERT pro semidynamizaci (rozklad A na množiny A_i)

INSERT(x)
if $x \notin A$ **then**
  nalezneme nejmenší j, že $A_j = \emptyset$
end if
$A_j = \{x\} \cup \bigcup i < j A_i, A_i = \emptyset$ pro $i < j$
vytvoříme strukturu S spojový seznam pro A_j
x přidáme do reprezentace A.

Kdy se buduje znovu (tedy podruhé) S struktura pro A_j (měřeno počtem INSERTů) ?

1. musí se naplnit všechny A_i pro i¡j to je $2^j - 1$ úspěšných INSERTů (ty, které přidaly prvek)

2. provede se úspěšný INSERT, který vyprázdní A_i pro $i \leq j$

3. znovu se musí naplnit A_i tj. $2^j - 1$ úspěšných INSERTů

4. daší úspešný INSERT vytvoří teprve S strukturu pro A_j

tj. $2^j - 1 + 1 + 2^j - 1 + 1 = 2 \cdot 2^j = 2^{j+1}$ úspěšných INSERTů.
Amortizovaný čas operace INSERT je

$$log|A| + \sum_{i=0}^{log|A|} \frac{P(2^j)}{2^{j+1}} \leq log|A| + \sum_{i=0}^{log|A|} \frac{P|A|}{|A|} = O(log|A| \cdot \frac{P|A|}{|A|})$$

□

Věta 11.2.2. *Máme rozložitelný vyhledávací problém f a máme pro něj statickou strukturu, která ho řeší v čase $Q(n)$, vyžaduje $S(n)$ paměti a vytvoří se v čase $P(n)$, kde$Q(n), \frac{P(n)}{n}, \frac{S(n)}{n}$ jsou neklesající funkce. Pak existuje semidynamická dat. struktura D, řešící f v čase $O(Q(n) \log n)$ vyžadující $O(S(n))$ paměti a umožňující INSERT se složitostí $O(\frac{P(n)}{n} \cdot \log n)$.*

11.2.2 INSERT se složitostí v nejhorším případě

Následuje konstrukce takové semidynamické struktury, která bude podporovat INSERT se složitostí v nejhorším případě.

Poznámka 11.2.3. Pokud $\frac{P(n)}{n} = \Theta(n^\varepsilon)$ pro $\varepsilon > 0$, pak amortizovaný čas pro operaci INSERT bude $O(\frac{P|A|}{|A|})$.

Máme množinu A
budeme mít roklad A na disjunktní množiny $A_{i,j}, i = 0, 1, ..., j \in 0, 1, ..., k_j$, kde $k_j \in 0, 1, 2$.
$|A_{i,j}| = 2^i$ a platí:
když $A_{i,0}$ existuje pro $i > 0$, pak existují $A_{i-1,0}, A_{i-1,1}$.
Struktura:

1. reprezentace A (pomocí (a,b)-stromů, červeno-černých stromů, ...)

2. $\forall$ existující $A_{i,j}$ je S struktura reprezentující $A_{i,j}$

3. $\forall$ existující $A_{i,j}$ je spojový seznam reprezentující $A_{i,j}$

4. když $A_{i,0}$ a $A_{i,1}$ existují pro nějaké i, pak je "rozpracovaná" S struktura pro množinu $A_{i-1,k_i+1} = Ai, 0 \cup A_{i,1}$. tj. bylo provedeno několik kroků pro její vytvoření, ale není dokončena.

$A \subseteq U_2, i_0 \in N$
$\forall i = 0, 1, ..., i_0$ je dáno $j_i \in 0, 1, 2$ takové, že $j_i > 0$ když $i < i_0$.
$\forall i = 0, 1, ..., i_0$ a $\forall j = 0, 1, ..., j_i$ je $A_{i,j} \in A$ taková, že $|A_{i,j}| < 2^i$.

Definice 11.2.1. $A_{i,j}, i = 0, 1, ..., i_0, j = 1, 2, ..., j_i$ je rozklad A.

Pro každé $A_{i,j}$ je dána S struktura reprezentující $A_{i,j}$ a spojový seznam prvků z $A_{i,j}$, navíc dána dat. struktura reprezentující A. Když $A_{i,1}$ existuje, pak je rozpracovaná S struktura pro $A_{i+1,j_{i+1}+1} = A_{1,0} \cup A_{i,1}$.

Poznámka 11.2.4. Struktura je rozpracovaná, jestliže bylo provedeno několik kroků pro postaverní S struktury, ale ještě není dokončena.

- toto je definice nové semidynamické struktury.
Paměťové nároky

$$\begin{aligned} & |A| + \sum_{i=0}^{log|A|} 4S(2^i) \\ & = |A| + \sum_{i=0}^{log|A|} \frac{S(2^i)}{2^i} 2^i \leq |A| + 4 \sum_{i=0}^{log|A|} \frac{S(|A|)}{|A|} 2^i \\ & = |A| + \frac{4S(|A|)}{|A|} (\sum 2^i) = |A| + 4S(|A|) \\ & = O(S(|A|)) \end{aligned} \tag{11.3}$$

Poznámka 11.2.5. $|A|$ - paměť pro pom. struktury
$\sum_{i=0}^{log|A|} 4S(2^i)$ - paměť potřebná na S struktury

Algoritmus pro výpočet :
spočítáme $f(x, A_{i,j})$ pro každou $A_{i,j}$ a pomocí operace $\oplus$ spočítáme $f(x, A)$.
Čas potřebný pro výpočet A

$$\sum_{i=0}^{log|A|} 3Q(2^i) + 3\log|A| \leq 3\sum_{i=0}^{log|A|} Q(|A|) + 3\log|A| = 3Q(|A|)log|A| = O(Q(|A|)\log|A|) \quad (11.4)$$

Platí: $Q(n) \geq n^\epsilon$ pro nějaké ϵ, pak čas potřebný pro výpočet f je $O(Q(N))$.
INSERT(x) viz alg. 11.1

Algoritmus 11.2 INSERT pro semidynamizaci (rozklad A na $A_{i,j}$)

```
INSERT(x)
if x ∉ A then
  postavíme S-strukturu pro množinu A_{0,j_0} = x
  j_0++
  i = 1
  while j[i] > 0 do
    if S-struktura pro A_{i,j_i-1} není dostavěna then
      provedeme dalších P(2^i)/2^i kroků pro vystavění S-stry pro A_{i,j_i-1}
      if S-stra pro A_{i,j_i-1} je dostavěna then
        A_{i-1,0} = A_{i-1,2}
        A_{i-1,1} = A_{i-1,3}
        if i - 1 > 0 then
          // na všech úrovních kromě 0-té, dojde k tom, že j_i = 5
          // tj. S-struktura pro A_{i,4} je rozestavěna
          // poprvé k tomu dojde při 10. INSERTu, takže trpělivost
          A_{i-1,2} = A_{i-1,4}
        end if
        j_{i-1} = j_{i-1} - 2
        A_{i,j_i} = A_{i-1,0} + A_{i-1,1}
        provedeme první krok pro vystavění S-stry pro A[i,j[i]]
        j_i++
      end if
    end if
    i++
  end while
  if j[i-1] > 1 a S-struktury pro A_{i-1,0} a A_{i-1,1} jsou dostavěny then
    A_{i,0} = A_{i-1,0} + A_{i-1,1}
    provedeme první krok pro vystavění S-struktury pro A_{i,0}
    j_i++
  end if
end if
```

Alg. INSERT pro semidyn. byl ověřen doktorem Koubkem

Poznámka 11.2.6. Může stát, že se vytvoří nová množina $A[i, j(i)]$, pak $j(i)$ má hodnotu 5, tj. musí se po dokončení struktury $A[i+1, j(i+1)-1]$ zmenšit o dvě hodnoty $A(i, 2)$, $A(i, 3)$ a i $A(i, 4)$ (nestačí jen pro první dvě hodnoty).

Čas pro INSERT(x) je

$$log|A| + \sum_{i=0}^{log|A|} (\frac{P(2^i)}{2^i} + 1) = 2log|A| + \sum_{i=0}^{log|A|} \frac{2P(|A|)}{|A|} =$$

$$2log|A| + \frac{2P(|A|)}{|A|} \sum_{i=0}^{log|A|} 1 = 2log|A| \frac{2P(|A|)}{|A|} log|A| = O(\frac{2P(|A|)}{|A|} log|A|) \quad (11.5)$$

$log|A|$ - čas pro zjištění zda $x \in A$

Když $\frac{P(n)}{n} \geq n^{\varepsilon}$ pro $\varepsilon > 0$, pak INSERT vyžaduje čas $O(\frac{P(n)}{n})$.

Příklad 11.2.1. XXX

INSERT(x_1)	INSERT(x_2)	INSERT(x_3)
$A_{0,0} = \{x_1\}$	$A_{0,0} = \{x_1\}$ $A_{0,1} = \{x_2\}$ 1. krok pro $A_{1,0} = \{x_1, x_2\}$	$A_{0,0} = \{x_1\}$ $A_{0,1} = \{x_2\}$ $A_{0,2} = \{x_3\}$ $\frac{P(2)}{2}$ kroků pro $A_{1,0} = \{x_1, x_2\}$
INSERT(x_4)	INSERT(x_5)	INSERT(x_6)
$A_{0,0} = \{x_1\}$ $A_{0,1} = \{x_2\}$ $A_{0,2} = \{x_3\} \rightarrow A_{0,0} = \{x_3\}$ $A_{0,3} = \{x_4\} \rightarrow A_{0,1} = \{x_4\}$ dokončíme $A_{1,0} = \{x_1, x_2\}$ 1. krok pro $A_{1,1} = \{x_3, x_4\}$	$A_{0,0} = \{x_3\}$ $A_{0,1} = \{x_4\}$ $A_{0,2} = \{x_5\}$ $A_{1,0} = \{x_1, x_2\}$ $\frac{P(2)}{2}$ kroků pro $A_{1,1} = \{x_3, x_4\}$	$A_{0,0} = \{x_3\}$ $A_{0,1} = \{x_4\}$ $A_{0,2} = \{x_5\} \rightarrow A_{0,0} = \{x_5\}$ $A_{0,3} = \{x_6\} \rightarrow A_{0,1} = \{x_6\}$ $A_{1,0} = \{x_1, x_2\}$ dokončeno $A_{1,1} = \{x_3, x_4\}$ 1. krok pro $A_{1,2} = \{x_5, x_6\}$ 1. krok pro $A_{2,0} = \{x_1, x_2, x_3, x_4\}$ $\frac{P(4)}{4}$ kroků

Věta 11.2.3. *Nechť S je statická struktura pro rozložitelný vyhledávací problém f a nechť K je "hladká" funkce. Pak existuje semidynamická struktura D založená na rozkladu určeném funkcí K, tak že platí:*
když $K = O(logn)$, pak čas pro vyhledání je $O(KQ(n))$
pro INSERT je $O(K(n)n^{\frac{1}{K(n)}} \frac{P(n)}{n})$
Když $K = \Omega(log(n))$, pak platí:
čas pro vyhledání je $O(K(n))Q(n))$
Pro INSERT je $O(\frac{log(n)}{log \frac{K(n)}{log(n)}} \frac{P(n)}{n})$.

Důkaz. viz [2]. □

11.3 Dynamizace

Potřebujeme, aby struktura S připouštěla falešný DELETE (prvek pouze škrtneme, ale zůstane tam. falešný - čas. ani paměťové nároky se nezlepší ani nezhorší)

Definice 11.3.1. *Falešný DELETE* je operace, která vyškrtne prvek z množiny - tj. umožní počítat $f(x, A - \{a\})$ (kde a je vyškrtnutý prvek) tak, že časové nároky budou stejné jako když nebyl žádný prvek vyškrtnut.

Budeme předpokládat, že čas pro falešný DELETE je $O(n)$, kde n je velikost původní reprezentované množiny.

11.3.1 Reprezentace množiny A

Rozložíme A na disjunktní množiny $A_j, j = 0, 1, ..., log|A| + 3$ takové, že buď $A_j = \emptyset$ nebo $2^{j-3} < |A_j| \leq 2^j$.

každá množina A_j bude reprezentována strukturou, která původně (když nebyly vyškrtnuté žádné prvky) měla velikost $\leq 2^j$.

Dále $\forall A_j \neq \emptyset$ bude dán spojový seznam prvků v A_j.

Bude dána datová reprezentace množiny A. Pro každý prvek a v spojovém seznamu množiny A_j bude dán ukazatel na prvek a v dat. struktuře reprezentující A a naopak. Pro každý prvek v dat. struktuře repr. A je dán ukazatel na prvek a v odpovídajícím spojovém seznamu.

11.3.2 Paměťové nároky

$$|A| + \sum_{i=0}^{log|A|+3} S(2^i) = |A| + \sum \frac{S(2^i)}{2^i} 2^i \leq |A| + \sum_{i=0}^{log|A|+3} \frac{S(8|A|)}{8|A|} 2^i =$$

$$|A| + \frac{S(8|A|)}{8|A|} 2^i = |A| + \frac{S(8|A|)}{8|A|} \sum 2^i = |A| + S(8|A|) = O(S(8|A|)) \quad (11.6)$$

$|A|$ - pomocné struktury
suma - paměť pro S struktury

Závěr: Když S je omezená polynomem, pak paměťové nároky jsou $O(S(n))$. Pokud S je superpolynomiální, pak paměť. nároky jsou $O(S(8n))$ (a platí $S(n) = o(S(8n))$)

Výpočet f:
spočítáme $f(x, A_j)$ a pomocí operace $\oplus$ spočítáme $f(x, A)$.

11.3.3 Čas pro výpočet f

$log(n) + \sum_{i=0}^{log|A|+3} Q(2^i) = log(n) + \sum Q(8|A|) = O(Q(8|A|)log|A|)$.

Závěr: čas na výpočet f je
- když Q je subpolynomiální: $O(Q(n)log(n))$
- polynomiální: $O(Q(n))$
- superplynomiální: $O(Q(8n))$

INSERT(x) viz alg. 11.3

Algoritmus 11.3 Operace INSERT (f)

if $x \notin A$ **then**
nalezneme nejmenší j takové, že $|\bigcup i \leq j A_i| < 2^j$
položíme $A_j = \bigcup i \leq j A_i \cup \{x\}$
$A_i = \emptyset proi < j$
vytvoříme S-strukturu a spojový seznam pro A_j (x přidáme do struktury reprezentující A a přidáme požadované ukazatele)
end if

Pozorování:
Když vytváříme při INSERTu S-strukturu pro A_j, pak $2^{j-1} < |A_j| \leq 2^j$.

(když toto neplatí, pak pro $j-1$ je splněna nerovnost $|\bigcup i < j-1 A_i| < 2^{j-1}$ a to je spor s minimalitou j.

DELETE(x) viz alg. 11.4

Algoritmus 11.4 Operace INSERT (f)

if $x \notin A$ **then**
  odstraníme x ze struktury pro A
  nalezneme j takové, že $x \in A_j$ (budeme znát přímo místo x v seznamu pro A_j)
  if $|A_j| = 1$ **then**
    smažeme A_j (odpovídající S-strukturu a spojový seznam) $\rightarrow A_j = \emptyset$
  end if
  if $|A_j| > 1$ a zároveň $|A_j| > 2^{j-3} + 1$ **then**
    na S strukturu pro A_j provedeme falešný DELETE(x), x smažeme ze spojového seznamu pro $A_j \rightarrow A_j = A_j - \{x\}$
  end if
  if $|A_j| > 1$ a zároveň $|A_j| = 2^{j-3} + 1$ **then**
    if $A_{j-1} = \emptyset$ **then**
      $A_{j-1} = A_{j-1} - \{x\}, A_j = \emptyset$
      vybudujeme novou S-strukturu pro A_{j-1} (x odstraníme ze spojového seznamu pro $A_{j-1} - 1$)
    end if
    if $A_{j-1} = \emptyset$ a zároveň $|A_{j-1}| > 2^{j-2}$ **then**
      vyměním $A_j a A_{j-1}$
      z A_{j-1} odstraníme x a vytvoříme novou S-strukturu pro A_{j-1} (původní struktura mohla mít až 2^j prvků)
    end if
    if $A_{j-1} = \emptyset$ a zároveň $|A_{j-1}| \leq 2^{j-2}$ **then**
      $B = (A_j \cup A_{j-1}) - \{x\}$
      zrušíme S-struktury pro A_j, A_{j-1} a vybudujeme S-strukturu a spojový seznam pro B
      if $|B| \geq 2^{j-2}$ **then**
        $A_j = B, A_{j-1} = \emptyset$
      else
        $A_{j-1} = B, A_j = \emptyset$
      end if
    end if
  end if
end if

Pozorování:
Když operace DELETE buduje S-strukturu pro množinu A_j, pak platí: $2^{j-1} \leq |A_j| \leq 2^{j-1}$.

11.3.4 Amortizovaný čas operace DELETE

$$(log|A| + D(2^j) + P(2^j) = (log|A| + D(2^j) + \frac{P(2^j)}{2^{j-3}}) = O(log|A| + D(|A|) + 4\frac{P(|A|)}{|A|})$$

- $log|A|$ - zjištění zda $x \in A$
- $D(2^j)$ - falešný DELETE

- $\frac{P(2^j)}{2^{j-3}}$ - budování S-struktury pro A_i

Aby DELETE znovu vytvářel S-strukturu pro množinu v A_i, musím provést aspoň 2^{j-3} operací DELETE.

11.3.5 Amortizovaný čas operace INSERT

Když INSERT vytvářel S-strukturu pro A_i, pak $A_j = \emptyset$ pro $j < i$ a aby se znovu vytvářela struktura pro A_i, musí platit:

$$1 + \sum_{j \leq i} |A_j| > 2^{j-1}$$

DELETE zaplní A_j jen do poloviny. To znamená, že se musí provést alespoň 2^{j-2} INSERTŮ, tedy amortizovaná složitost je

$$O(log|A| + \sum \frac{P(2^i)}{2^{i-2}}) = O(\frac{P(|A|)}{|A|} logn)$$

Práce s pomocnými stukturami zabere právě $log|A|$ času.
Když $P(n) = n^\epsilon$ pro $\epsilon > 0$, pak amortizovaná složitost je $O(\frac{P(|A|)}{|A|})$.

Literatura

[1] Mehlhorn, Kurt. (1983): *Data Structures And Algorithms*, Springer Verlag

[2] Mehlhorn K., Overmars M. H. (XXX): *Optimal Dynamization of Decomposable Searching Problems*

[3] Douglas C. Schmidt (1990): *GPERF: A Perfect Hash Function Generator*, in Proceedings of the 2nd C++ Conference, San Francisco, California, USENIX, pp. 87–102. Článek se dá stáhnout z citeseer.

[4] Adel'son-Velskii G. M., Landis E. M. (1962): *An Algorithm for the Organization of Information*, Soviet Math. Dockl.

[5] Topfer P. (1995): Algoritmy a programovací techniky, nakl. Prometheus, ISBN 80-85849-83-6

[6] Chen Wen-Chin, Vitter Jeffrey Scott (1984): *Analysis of new variants of coalesced hashing*, ACM Transactions on Database Systems (TODS) archive Volume 9 , Issue 4 (December 1984) table of contents Pages: 616 - 645, Year of Publication: 1984, ISSN:0362-5915, Publisher ACM Press New York, NY, USA.