Datové struktury I
NTIN066

Jirka Fink
https://ktiml.mff.cuni.cz/~fink/

Katedra teoretické informatiky a matematické logiky
Matematicko-fyzikální fakulta
Univerzita Karlova v Praze

Zimní semestr 2018/19
Poslední změna 8. ledna 2019

Základní informace

Kontakt

E-mail	fink@ktiml.mff.cuni.cz
Homepage	https://ktiml.mff.cuni.cz/˜fink/
Konzultace	Individuální domluva

Cíle předmětu

- Naučit se navrhovat a analyzovat netriviální datové struktury
- Porozumět jejich chování – jak asymptoticky, tak na reálném počítači
- Zajímá nás nejen chování v nejhorším případě, ale i průměrně/amortizovaně
- Nebudujeme obecnou teorii všech DS ani neprobíráme všechny varianty DS, ale ukazujeme na příkladech různé postupy a principy

Obsah

Zápočet: Domácí úkoly

Podmínky

- Bude zadáno pět domácích úkolů po 110 bodech
- K získání zápočtu je nutné získat alespoň 320 bodů
- Úkol = implementace DS + měření + grafy + zdůvodnění výsledků
- Úkol musí být odevzdán včas a DS musí být funkční
- Důrazně doporučujeme používat C/C++, i když povolujeme i Javu a C#
- Nepoužívejte cizí kód ani knihovny třetích stran
- K implementaci DS nepoužívejte ani standardní knihovny (ani std::list, std::vector, etc.)
- Úkoly jsou zadávány centrálně, ale opravuje je vyučující, u kterého jste registrováni na SISu
- Při odevzdání v předtermínu můžete navíc získat 10 bodů nebo poslat opravené řešení
- Přesná pravidla a vzorový příklad jsou na webu přednášky

Nové předměty

Motivace

- Na cvičeních se především rozebírají domácí úkoly
- Nezbývá mnoho času na procvičení
- Řada studentů neumí implementovat datové struktury v rozumném čase bez zdlouhavého hledání chyb

Analýza datových struktur (NTIN105)

- Návrh a analýza datových struktur, které zazněly na přednášce, ale na jejich zkoumání není na klasickém cvičení čas
- Vyučující: Martin Mareš
- Rozvrh bude umluven po emailu (mares+ds@kam.mff.cuni.cz)

Nové předměty

Implementace datových struktur (NTIN106)

- Naučit studenty efektivně implementovat datové struktury v rozumném čase bez únavného hledání chyb
- Vyučující: Jirka Fink
- Rozvrh: středa, 17:20, S4

Obsah

- Návrh implementace datových struktur
- Application programming interface
- Unit a integrity testy
- Implementace bez rekurze (a zásobníku)
- Spojový seznam, stromy, . . .
- Správa paměti
- Paralelní programování bez zámků
- Implementace nástrojů vyšších programovacích jazyků v C/RAM
- Diskuze různých implementací domácích úkolů, testování a zkušeností
- Zápočet: recenze řešení domácích úkolů ostatních studentů (code review)

Literatura

- A. Koubková, V. Koubek: Datové struktury I. MATFYZPRESS, Praha 2011.
- T. H. Cormen, C.E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms. MIT Press, 2009
- K. Mehlhorn: Data Structures and Algorithms I: Sorting and Searching. Springer-Verlag, Berlin, 1984
- D. P. Mehta, S. Sahni eds.: Handbook of Data Structures and Applications. Chapman & Hall/CRC, Computer and Information Series, 2005
- E. Demaine: Cache-Oblivious Algorithms and Data Structures. 2002.
- R. Pagh: Cuckoo Hashing for Undergraduates. Lecture note, 2006.
- M. Thorup: High Speed Hashing for Integers and Strings. Lecture notes, 2014.
- M. Thorup: String hashing for linear probing (Sections 5.1-5.4). In Proc. 20th SODA, 655-664, 2009.

Obsah

Amortizovaná analýza

Motivace

- Uvažujme datovou struktury, která zvládá nějakou operaci většinou velmi rychle.
- Ale občas potřebuje reorganizovat svoji vnitřní struktury, což operaci v těchto výjimečných případech značně zpomaluje.
- Tudíž je časová složitost v nejhorším případě velmi špatná.
- Představme si, že naše datová struktura je použita v nějakém algoritmu, který operaci zavolá mnohokrát.
- V této situaci složitost algoritmu ovlivňuje celkový čas mnoha operací, nikoliv složitost operace v nejhorším případě.
- Cíl: Chceme zjistit "průměrnou" hodnotu časových složitostí posloupnosti operací, případně celkovou složitost posloupnosti operací.

Metody výpočtu amortizované složitosti

- Agregovaná analýza
- Účetní metoda
- Potenciální metoda

Obsah

Inkrementace binárního čítače: Agregovaná analýza

Binární čítač

- Máme n-bitový čítač inicializovaný libovolnou hodnotou
- Při operaci INCREMENT se poslední nulový bit změní na 1 a všechny následující jedničkové bity se změní na 0
- Počet změněných bitů v nejhorším případě je n
- Kolik bitů se změní při k operacích INCREMENT?

Agregovaná analýza

- Poslední bit se změní při každé operaci — tedy k-krát
- Předposlední bit se změní při každé druhé operaci — nejvýše $\lceil k/2 \rceil$-krát
- i-tý bit od konce se změní každých 2^i operací — nejvýše $\lceil k/2^i \rceil$-krát
- Celkový počet změn bitů je nejvýše $\sum_{i=0}^{n-1} \lceil k/2^i \rceil \leq \sum_{i=0}^{n-1} (1 + k/2^i) \leq n + k \sum_{i=0}^{n-1} 1/2^i \leq n + 2k$
- Časová složitost k operací INCREMENT nad n-bitovým čítačem je $\mathcal{O}(n+k)$
- Jestliže $k = \Omega(n)$, pak amortizovaná složitost na jednu operace je $\mathcal{O}(1)$

Inkrementace binárního čítače: Účetní metoda

Účetní metoda

- Změna jednoho bitu stojí jeden žeton a na každou operaci dostaneme dva žetony
- Invariant: U každého jedničkového bitu si uschováme jeden žeton
- Při inkrementu máme vynulování jedničkových bitů předplaceno
- Oba žetony poskytnuté k vykonání operace využijeme na jedinou změnu nulového bitu na jedničku a předplacení jeho vynulování
- Na začátku potřebujeme dostat nejvýše n žetonů
- Celkově dostaneme na k operací $n + 2k$ žetonů
- Amortizovaný počet změněných bitů při jedné operaci je $\mathcal{O}(1)$ za předpokladu $k = \Omega(n)$

Potenciální metoda

- Potenciál nulového bitu je 0 a potenciál jedničkového bitu je 1
- Potenciál čítače je součet potenciálů všech bitů ①
- Potenciál po provedení j-té operace označme Φ_j skutečný počet změněných bitů při j-té operaci označme T_j ②
- Chceme spočítat amortizovaný počet změněných bitů, který označíme A
- Pro každou operaci j musí platit $T_j \leq A + (\Phi_{j-1} - \Phi_j)$ pro libovolnou operaci j ③
- Podobně jako v účetní metodě dostáváme $A \geq T_j + (\Phi_j - \Phi_{j-1}) \geq 2$
- Celkový počet změněných bitů při k operacích je

$$\sum_{j=1}^{k} T_j \leq \sum_{j=1}^{k} (2 + \Phi_{j-1} - \Phi_j) \leq 2k + \Phi_0 - \Phi_k \leq 2k + n,$$

protože $0 \leq \Phi_j \leq n$ ④

1. V tomto triviálním příkladu je potenciál přesně počet žetonů v účetní metodě.
2. Φ_0 je potenciál před provedení první operace a Φ_k je potenciál po poslední operaci.
3. Toto je zásadní fakt amortizované analýzy. Potenciál je jako banka, do které můžeme uložit peníze (čas), jestliže operace byla levná (rychle provedená). Při drahých (dlouho trvajících) operacích musíme naopak z banky vybrat (snížit potenciál), abychom operaci zaplatili (stihli provést v amortizovaném čase). V amortizované analýze je cílem najít takovou potenciální funkci, že při rychle provedené operaci potenciál dostatečně vzroste a naopak při dlouho trvajících operací potenciál neklesne příliš moc.
4. Součtu $\sum_{j=1}^{k}(\Phi_{j-1} - \Phi_j) = \Phi_0 - \Phi_k$ se říká teleskopická suma a tento nástroj budeme často používat.

Potenciální metoda

Definice

Potenciál Φ je funkce, která každý stav datové struktury ohodnotí nezáporným reálným číslem. Operace nad datovou strukturou má amortizovanou složitost A, jestliže libovolné vykonání operace splňuje

$$T \leq A + \left(\Phi(S) - \Phi(S')\right),$$

kde T je skutečný čas nutný k vykonání operace, S je stav před jejím vykonáním a S' je stav po vykonání operace.

Příklad: Inkrementace binárního čítače

- Potenciál Φ je definován jako počet jedničkových bitů v čítači
- Skutečný čas T je počet změněných bitů při jedné operaci INCREMENT
- Amortizovaný čas je 2
- Platí $T \leq A + (\Phi(S) - \Phi(S'))$

Obsah

Dynamické pole

Dynamické pole

- Máme pole, do kterého přidáváme i mažeme prvky
- Počet prvků označíme n a velikost pole p
- Jestliže $p = n$ a máme přidat další prvek, tak velikost pole zdvojnásobíme
- Jestliže $p = 4n$ a máme smazat prvek, tak velikost pole zmenšíme na polovinu ①

Intuitivní přístup ②

- Zkopírování celého pole trvá $\mathcal{O}(n)$
- Jestliže po realokaci pole máme n prvků, pak další realokace nastane nejdříve po $n/2$ operacích INSERT nebo DELETE ③
- Amortizovaná složitost je $\mathcal{O}(1)$

Agregovaná analýza: Celkový čas

- Nechť k_i je počet operací mezi $(i-1)$ a i-tou realokací $\Rightarrow \sum_i k_i = k$
- Při první realokaci se kopíruje nejvýše $n_0 + k_1$ prvků, kde n_0 je počáteční počet
- Při i-té realokaci se kopíruje nejvýše $2k_i$ prvků, kde $i \geq 2$ ④
- Celkový počet zkopírovaných prvků je nejvýše $n_0 + k_1 + \sum_{i\geq 2} 2k_i \leq n_0 + 2k$

1. Přesněji: Uvažujeme přidávání a mazání prvků ze zásobníku.
2. V analýze počítáme pouze čas na realokaci pole. Všechny ostatní činnost při operacích INSERT i DELETE trvají $\mathcal{O}(1)$ v nejhorším čase. Zajímá nás počet zkopírovaných prvků při realokaci, protože předpokládáme, že kopírování jednoho prvku trvá $\mathcal{O}(1)$.
3. Po realokaci a zkopírování je nové pole z poloviny plné. Musíme tedy přidat n prvků nebo smazat $n/2$ prvků, aby došlo k další realokaci.
4. Nejhorším případem je posloupnost INSERT, kdy zdvojnásobíme počet prvků, které poté musíme realokovat.

Dynamické pole

Potenciální metoda

- Uvažujme potenciál

$$\Phi = \begin{cases} 0 & \text{pokud } p = 2n \\ n & \text{pokud } p = n \\ n & \text{pokud } p = 4n \end{cases}$$

  a tyto tři body rozšíříme po částech lineární funkcí
- Explicitně

$$\Phi = \begin{cases} 2n - p & \text{pokud } p \leq 2n \\ p/2 - n & \text{pokud } p \geq 2n \end{cases}$$

- Změna potenciálu při jedné operaci bez realokace je $\Phi' - \Phi \leq 2$ ①
- Skutečný počet zkopírovaných prvků T vždy splňuje $T + (\Phi' - \Phi) \leq 2$
- Celkový počet zkopírovaných prvků při k operacích je nejvýše $2k + \Phi_0 - \Phi_k \leq 2k + n_0$
- Celkový čas k operací je $\mathcal{O}(n_0 + k)$
- Amortizovaný čas jedné operace je $\mathcal{O}(1)$

1

$$\Phi' - \Phi = \begin{cases} 2 & \text{pokud přidáváme a } p \leq 2n \\ -2 & \text{pokud mažeme a } p \leq 2n \\ -1 & \text{pokud přidáváme a } p \geq 2n \\ 1 & \text{pokud mažeme a } p \geq 2n \end{cases}$$

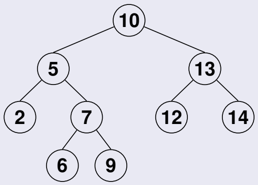
Obsah

Binární vyhledávací strom

Vlastnosti

- Binární strom (každý vrchol obsahuje nejvýše dva syny)
- Klíč v každém vnitřním vrcholu je větší než všechny klíče v levém podstromu a menší než všechny klíče v pravém podstromu
- Prvky mohou být uloženy pouze v listech nebo též ve vnitřních vrcholech (u každého klíče je uložena i hodnota)

Příklad



Složitost

- Paměť: $\mathcal{O}(n)$
- Časová složitost operace Find je lineární ve výšce stromu
- Výška stromu může být až $n - 1$

Obsah

Váhově vyvážené stromy: BB[α]-strom

BB[α]-strom (Nievergelt, Reingold [18])

- Binární vyhledávací strom ①
- Počet vrcholů v podstromu vrcholu u označme s_u ②
- Pro každý vrchol u platí, že podstromy obou synů u musí mít nejvýše αs_u vrcholů ③
- Zřejmě musí platit $\frac{1}{2} < \alpha < 1$ ④

Výška BB[α]-stromu

- Podstromy všech vnuků kořene mají nejvýše $\alpha^2 n$ vrcholů
- Podstromy všech vrcholů v i-té vrstvě mají nejvýše $\alpha^i n$ vrcholů
- $\alpha^i n \geq 1$ jen pro $i \leq \log_{\frac{1}{\alpha}}(n)$
- Výška BB[α]-stromu je $\Theta(\log n)$

Operace Build: Vytvoření BB[α]-stromu ze setříděného pole

- Prostřední prvek dáme do kořene
- Rekurzivně vytvoříme oba podstromy
- Časová složitost je $\mathcal{O}(n)$

1. V přednášce budeme předpokládat, že prvky jsou uloženy ve všech vrcholech, i když existuje varianta BB[α]-stromů mající prvky jen v listech.
2. Do s_u započítáváme i vrchol u.
3. V literatuře můžeme najít různé varianty této podmínky. Podstatné je, aby oba podstromy každého vrcholu měli „zhruba“ stejný počet vrcholů.
4. Pro $\alpha = \frac{1}{2}$ lze BB[α]-strom sestrojit, ale operace INSERT a DELETE by byly časově náročné. Pro $\alpha = 1$ by výška BB[α]-strom mohla být lineární.

BB[α]-strom: Operace INSERT a DELETE

Operace INSERT (DELETE je analogický)

- Najít list pro nový prvek a uložit do něho nový prvek (složitost: $\mathcal{O}(\log n)$)
- Jestliže některý vrchol porušuje vyvažovací podmínku, tak celý jeho podstrom znovu vytvoříme operací BUILD (složitost: amortizovaná analýza) ① ②

Amortizovaná časová složitost operací INSERT a DELETE: Agregovaná metoda

- Jestliže podstrom vrcholu u po provedení operace BUILD má s_u vrcholů, pak další porušení vyvažovací podmínky pro vrchol u nastane nejdříve po $\Omega(s_u)$ přidání/smazání prvků v podstromu vrcholu u (cvičení)
- Rebuild podstromu vrcholu u trvá $\mathcal{O}(s_u)$
- Amortizovaný čas vyvažovaní jednoho vrcholu je $\mathcal{O}(1)$ ③
- Při jedné operaci INSERT/DELETE se prvek přidá/smaže v $\Theta(\log n)$ podstromech
- Amortizovaný čas vyvažovaní při jedné operaci INSERT nebo DELETE je $\mathcal{O}(\log n)$
- Jaký je celkový čas k operací? ④

1. Při hledání listu pro nový vrchol stačí na cestě od kořene k listu kontrolovat, zda se přidáním vrcholu do podstromu syna neporuší vyvažovací podmínka. Pokud se v nějakém vrcholu podmínka poruší, tak se hledání ukončí a celý podstrom včetně nového prvku znovu vybuduje.
2. Existují pravidla pro rotování BB[α]-stromů, ale ta se nám dnes nehodí.
3. Operace BUILD podstromu vrcholu u trvá $\mathcal{O}(s_u)$ a mezi dvěma operacemi BUILD podstromu u je $\Omega(s_u)$ operací INSERT nebo DELETE do podstromu u. Všimněte si analogie a dynamickým polem.
4. Intuitivně bychom mohli říct, že v nejhorším případě BB[α]-strom nejprve vyvážíme v čase $\mathcal{O}(n)$ a poté provádíme jednotlivé operace, a proto celkový čas je $\mathcal{O}(n + k \log n)$, ale není to pravda. Proč?

BB[α]-strom: Operace INSERT a DELETE

Amortizovaná časová složitost operací INSERT a DELETE: Potenciální metoda

- V této analýze uvažujeme jen čas na postavení podstromu, zbytek trvá $\mathcal{O}(\log n)$
- Potenciál vrcholu u definován

$$\Phi(u) = \begin{cases} 0 & \text{pokud } |s_{l(u)} - s_{r(u)}| \leq 1 \\ |s_{l(u)} - s_{r(u)}| & \text{jinak,} \end{cases}$$

  kde $l(u)$ a $r(u)$ jsou levý a pravý synové u.
- Potenciál BB[α]-stromu Φ je součet potenciálů vrcholů
- Při vložení/smazání prvku se potenciál $\Phi(u)$ jednoho vrcholu zvýší nejvýše o 2 ①
- Pokud nenastane Rebuild, pak se potenciál stromu zvýší nejvýše o $\mathcal{O}(\log(n))$ ②
- Pokud nastane Rebuild vrcholu u, pak $\Phi(u) \geq \alpha s_u - (1-\alpha)s_u \geq (2\alpha - 1)s_u$
- Po rekonstrukci mají všechny vrcholy v podstromu u nulový potenciál ③
- Při rekonstrukci poklesne potenciál Φ alespoň o $\Omega(s_u)$, což zaplatí čas na rekonstrukci
- Dále platí $0 \leq \Phi \leq hn = \mathcal{O}(n \log n)$, kde h je výška stromu ④
- Celkový čas na k operací INSERT nebo DELETE je $\mathcal{O}((k+n)\log n)$

1. Potenciál se změní právě o 2, jestli rozdíl velikostí podstromů se změní z 1 na 2 nebo opačně. Jinak se potenciál změní právě o 1.
2. Potenciál se může změnit pouze vrcholům na cestě z kořene do nového/smazaného vrcholu a těch je $\mathcal{O}(\log n)$.
3. Právě zde potřebujeme, aby potenciál vrcholu byl nulový, i když se velikosti podstromů jeho synů liší o jedna.
4. Součet potenciálů všech vrcholů v jedné libovolné vrstvě je nejvýše n, protože každý vrchol patří do nejvýše jednoho podstromu vrcholu z dané vrstvy. Tudíž potenciál stromu Φ je vždy nejvýše nh. Též lze nahlédnout, že každý vrchol je započítán v nejvýše h potenciálech vrcholů.

Obsah

Staticky optimální strom

Cíl

Pro danou posloupnost operací FIND najít binární vyhledávací strom minimalizující celkovou dobu vyhledávání.

Formálně

Máme prvky $x_1, \dots, x_n$ s váhami $w_1, \dots, w_n$. Cena stromu je $\sum_{i=1}^{n} w_i h_i$, kde h_i je hloubka prvku x_i. Staticky optimální strom je binární vyhledávací strom s minimální cenou.

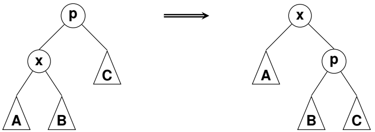
Konstrukce (cvičení)

- $\mathcal{O}(n^3)$ – triviálně dynamickým programováním
- $\mathcal{O}(n^2)$ – vylepšené dynamické programování (Knuth [14])

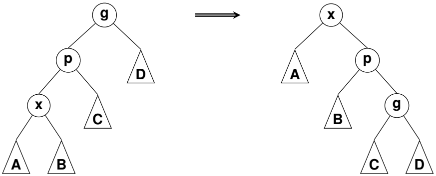
Jak postupovat, když neznáme váhy předem?

- Pomocí rotací bude udržovat často vyhledávané prvky blízko kořene
- Operací SPLAY „rotujeme“ zadaný prvek až do kořene
- Operace FIND vždy volá SPLAY na hledaný prvek

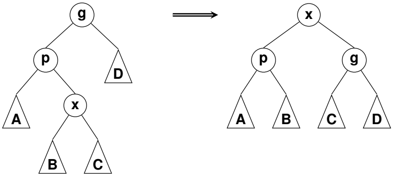
- Zig rotace: Otec *p* prvku *x* je kořen



- Zig-zig rotace: *x* a *p* jsou oba pravými nebo oba levými syny

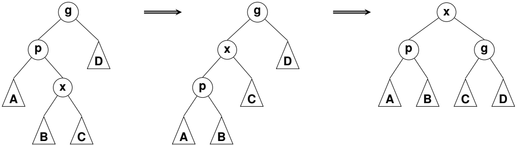


- Zig-zag rotace: *x* je pravý syn a *p* je levý syn nebo opačně

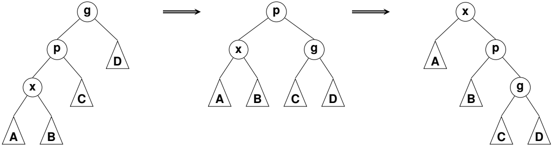


Uvažujeme *buttom-up* verzi, tj. prvek x nejprve najdeme a poté jej postupně rotujeme nahoru, což znamená, že x vždy značí stejný vrchol postupně se přesouvající ke kořeni a ostatní vrcholy stromu jsou sousedé odpovídající dané rotaci.
Existuje též *top-down* verze [16], která vždy rotuje vnuka kořene, jehož podstrom obsahuje prvek x. Tato verze je sice v praxi rychlejší, ale postup a analýza jsou složitější.

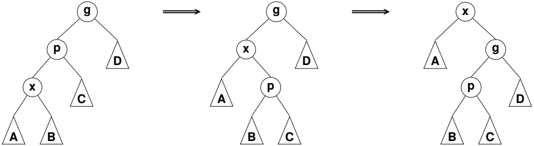
- Zig-zag rotace jsou pouze dvě jednoduché rotace prvku x s aktuálním otcem



- Zig-zig rotace jsou taky dvě rotace,



- ale dvě rotace prvku x s aktuálním otcem by vedli ke špatnému výsledku



Splay strom: Amortizovaná analýza

Lemma

Pro $a, b, c \in \mathbb{R}^+$ splňující $a + b \leq c$ platí $\log_2(a) + \log_2(b) \leq 2\log_2(c) - 2$.

Důkaz

- Platí $4ab = (a+b)^2 - (a-b)^2$
- Z nerovností $(a-b)^2 \geq 0$ a $a + b \leq c$ plyne $4ab \leq c^2$
- Zlogaritmováním dostáváme $\log_2(4) + \log_2(a) + \log_2(b) \leq \log_2(c^2)$

Značení

- Nechť velikost $s(x)$ je počet vrcholů v podstromu x (včetně x)
- Potenciál vrcholu x je $\Phi(x) = \log_2(s(x))$
- Potenciál Φ stromu je součet potenciálů všech vrcholů
- s' a Φ' jsou velikosti a potenciály po jedné rotaci
- Předkládáme, že jednoduchou rotaci zvládneme v jednotkovém čase

Lemma můžeme též dokázat pomocí Jensenovy nerovnosti, která tvrdí:
Jestliže f je konvexní funkce, $x_1, \ldots, x_n$ jsou čísla z definičního oboru f a $w_1, \ldots, w_n$ jsou kladné váhy, pak platí nerovnost

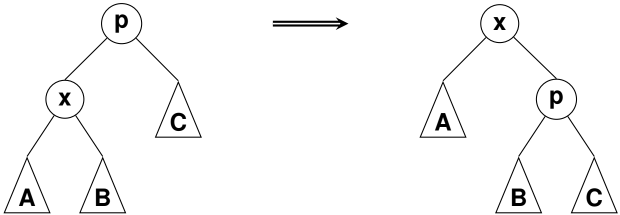
$$f\left(\frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i}\right) \leq \frac{\sum_{i=1}^n w_i f(x_i)}{\sum_{i=1}^n w_i}.$$

Jelikož funkce $\log$ je rostoucí a konkávní, dostáváme

$$\frac{\log(a) + \log(b)}{2} \leq \log\left(\frac{a+b}{2}\right) \leq \log(c) - 1,$$

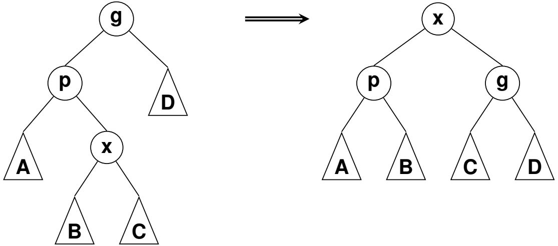
z čehož plyne znění lemmatu.

Splay strom: Zig rotace



Analýza

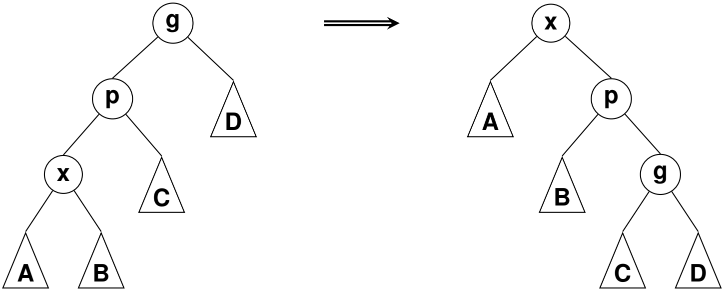
- $\Phi'(x) = \Phi(p)$
- $\Phi'(p) < \Phi'(x)$
- $\Phi'(u) = \Phi(u)$ pro všechny ostatní vrcholy u
- $$\begin{aligned} \Phi' - \Phi &= \sum_u (\Phi'(u) - \Phi(u)) \\ &= \Phi'(p) - \Phi(p) + \Phi'(x) - \Phi(x) \\ &\leq \Phi'(x) - \Phi(x) \end{aligned}$$



Analýza

1. $\Phi'(x) = \Phi(g)$
2. $\Phi(x) < \Phi(p)$
3. $\Phi'(p) + \Phi'(g) \leq 2\Phi'(x) - 2$
   - $s'(p) + s'(g) \leq s'(x)$
   - Z lemmatu plyne $\log_2(s'(p)) + \log_2(s'(g)) \leq 2\log_2(s'(x)) - 2$
4. $\Phi' - \Phi = \Phi'(g) - \Phi(g) + \Phi'(p) - \Phi(p) + \Phi'(x) - \Phi(x) \leq 2(\Phi'(x) - \Phi(x)) - 2$

Splay strom: Zig-zig rotace



Analýza

- $\Phi'(x) = \Phi(g)$
- $\Phi(x) < \Phi(p)$
- $\Phi'(p) < \Phi'(x)$
- $s(x) + s'(g) \leq s'(x)$
- $\Phi(x) + \Phi'(g) \leq 2\Phi'(x) - 2$
- $\Phi' - \Phi = \Phi'(g) - \Phi(g) + \Phi'(p) - \Phi(p) + \Phi'(x) - \Phi(x) \leq 3(\Phi'(x) - \Phi(x)) - 2$

Splay strom: Analýza

Amortizovaný čas ①

- Amortizovaný čas jedné zigzig nebo zigzag rotace:
  $T + \Phi' - \Phi \leq 2 + 3(\Phi'(x) - \Phi(x)) - 2 = 3(\Phi'(x) - \Phi(x))$ ②
- Amortizovaný čas jedné zig rotace:
  $T + \Phi' - \Phi \leq 1 + \Phi'(x) - \Phi(x) \leq 1 + 3(\Phi'(x) - \Phi(x))$
- Nechť Φ_i je potenciál po i-té rotaci a T_i je skutečný čas i-té rotace
- Amortizovaný čas (počet jednoduchých rotací) jedné operace SPLAY:

$$\sum_{i\text{-tá rotace}} (T_i + \Phi_i - \Phi_{i-1}) \leq 1 + \sum_{i\text{-tá rotace}} 3(\Phi_i(x) - \Phi_{i-1}(x))$$

③

$$\leq 1 + 3(\Phi_{\text{konec}}(x) - \Phi_0(x))$$

$$\leq 1 + 3\log_2 n = \mathcal{O}(\log n)$$

- Amortizovaný čas jedné operace SPLAY je $\mathcal{O}(\log n)$

Skutečný čas k operací SPLAY

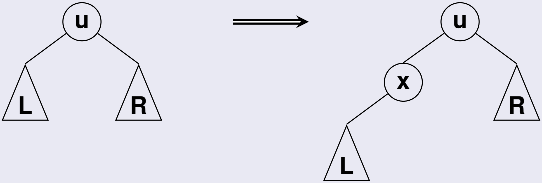
- Potenciál vždy splňuje $0 \leq \Phi \leq n \log_2 n$
- Rozdíl mezi konečným a počátečním potenciálem je nejvýše $n \log_2 n$
- Celkový čas k operací SPLAY je $\mathcal{O}((n + k) \log n)$

1. Časy na nalezení prvku a jeho SPLAY jsou stejné, a proto analyzujeme počet rotací při operaci SPLAY. Neúspěšná operace FIND dojde až k vrcholu mající NULL v ukazateli, na kterém se vyhledávání zastaví. Na tento poslední vrchol je nutné zavolat SPLAY, aby opakovaná neúspěšná vyhledávání měla amortizovanou logaritmickou složitost.
2. T značí skutečný čas rotace, což je počet jednoduchých rotací k provedení rotace zig, zigzig nebo zigzag.
3. Zig rotaci použijeme nejvýše jednou a proto započítáme „+1". Rozdíly $\Phi'(x) - \Phi(x)$ se teleskopicky odečtou a zůstane nám rozdíl potenciálů vrcholu x na konci a na začátku operace SPLAY. Na počátku je potenciál vrcholu x nezáporný a na konci je x kořenem, a proto jeho potenciál je $\log_2(n)$.

Splay strom: Operace INSERT

Vložení prvku x

1. Začneme vyhledáváním klíče x, které skončí ve vrcholu u ①
2. SPLAY(u)
3. Vložit nový vrchol s prvkem x ②



Amortizovaná složitost

- Operace FIND a SPLAY: $\mathcal{O}(\log n)$
- Vložením nového vrcholu potenciál Φ vzroste nejvýše o $\Phi'(x) + \Phi'(u) \leq 2\log_2 n$
- Amortizovaná složitost operace INSERT je $\mathcal{O}(\log n)$

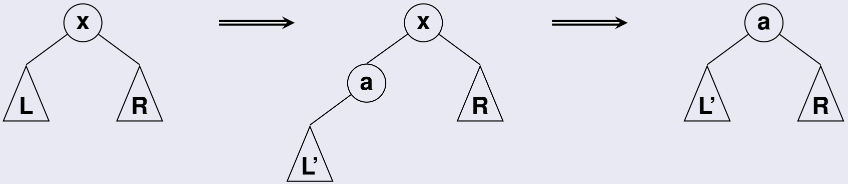
1. u obsahuje klíč, který je největší ze všech uložených klíčů menších než x nebo nejmenší ze všech uložených klíčů než x. Pokud je x menší nebo větší než všechny klíče, pak je u určen jednoznačně, jinak máme volbu ze dvou možných vrcholů. Rozmysleme si, že poslední navštívený vrchol při operaci FINDZ klasických binárních vyhledávacích stromů splňuje popsané požadavky na vrchol u.
2. Pokud nechceme mít ve stromu duplicitní klíče a klíč vrcholu u je roven x, pak vložení neprovedeme. Nicméně je stále nutné zavolat SPLAY na vrchol u, jinak by opakované pokusy o vložení prvku x mohli být hodně drahé.

Splay strom: Operace Delete

Algoritmus

Splay(x)
$L \leftarrow$ levý podstrom x
if L *je prázdný* **then**
  Smazat vrchol x
else
  Najít největší prvek a v L
  Splay(a)
  $L' \leftarrow$ levý podstrom a
  `#` a `nemá pravého syna`
  Sloučit vrcholy x a a

Pokud L je neprázdný, tak



Další vlastnosti splay stromů

Věta (vyhledávání prvků v rostoucím pořadí)

Jestliže posloupnost vyhledávání S obsahuje prvky v rostoucím pořadí, tak celkový čas na vyhledávání S ve splay stromu je $\mathcal{O}(n)$. ①

Věta (statická optimalita)

Nechť T je statický strom, $c_T(x)$ je počet navštívených vrcholů při hledání x a $x_1, \dots, x_m$ je posloupnost obsahující všechny prvky. Pak libovolný splay strom provede $\mathcal{O}\left(\sum_{i=1}^m c_T(x_i)\right)$ operací při hledání $x_1, \dots, x_m$. ②

Hypotéza (dynamická optimalita)

Nechť T je binární vyhledávací strom, který prvek x hledá od kořene vrcholu obsahující x a přitom provádí libovolné rotace. Cena jednoho vyhledání prvky je počet navštívených vrcholů plus počet rotací a $c_T(S)$ je součet cen vyhledání prvků v posloupnosti S. Pak cena vyhledání posloupnosti S v splay stromu je $\mathcal{O}(n + c_T(S))$. ③

1. n je opět počet prvků ve stromu a počáteční splay strom může mít prvky rozmístěné libovolně.
2. Každý prvek uložený ve stromě musíme aspoň jednou najít. Počáteční splay strom může mít prvky rozmístěné libovolně.
3. V dynamické optimalitě může T při vyhledávání provádět rotace, takže může být rychlejší než staticky optimální strom, například když S často po sobě vyhledává stejný prvek.

Splay stromy: Výhody a aplikace

Výhody a nevýhody Splay stromů

+ Nepotřebuje paměť na speciální příznaky (1)
+ Efektivně využívají procesorové cache (Temporal locality)
- Rotace zpomalují vyhledávání
- Vyhledávání nelze jednoduše paralelizovat
- Výška stromu může být i lineární (2)

Aplikace

- Cache, virtuální paměť, sítě, file system, komprese dat, . . .
- Windows, gcc compiler and GNU C++ library, sed string editor, Fore Systems network routers, Unix malloc, Linux loadable kernel modules, . . .

1. Červeno-černé stromy potřebují v každém vrcholu jeden bit na barvu, AVL stromy jeden bit na rozdíl výšek podstromů synů.
2. Když vyhledáme všechny prvky v rostoucím pořadí, pak strom zdegeneruje na cestu. Proto splay strom není vhodný v real-time systémech.

1. domácí úkol: Splay stromy

Stručné zadání

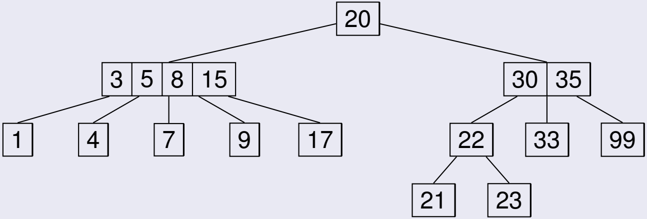
- Implementujte Splay strom s operacemi SPLAY, FIND, INSERT
- Implementujte „naivní Splay strom“, který v operaci SPLAY naivně používá jen jednoduché rotace místo dvojitých
- Měřte průměrnou hloubku hledaného prvku při operacích FIND
- Analyzujte závislost průměrné hloubky hledaných prvků na počtu prvků v Splay stromu a velikosti hledané podmnožiny
- Analyzujte průměrnou hloubku hledaných prvků v několika testech
- Napište program, který spočítá průměrnou hloubek prvků ve staticky optimálním stromu pro danou posloupnost vyhledávání
- Srovnejte průměrné hloubky hledaných prvků ve Splay stromu a ve staticky optimálním stromu
- Termín odevzdání: 28. 10. 2018, předtermín 21.10.2018
- Generátor dat a další podrobnosti: `https://ktiml.mff.cuni.cz/~fink/`

Obsah

Vlastnosti

- Vnitřní vrcholy mají libovolný počet synů (typicky alespoň dva)
- Vnitřní vrchol s k syny má $k-1$ setříděných klíčů
- V každém vnitřním vrcholu je i-tý klíč větší než všechny klíče v i-tém podstromu a menší než všechny klíče v $(i+1)$ podstromu pro všechny klíče i
- Prvky mohou být uloženy pouze v listech nebo též ve vnitřních vrcholech (u každého klíče je uložena i hodnota)

Příklad

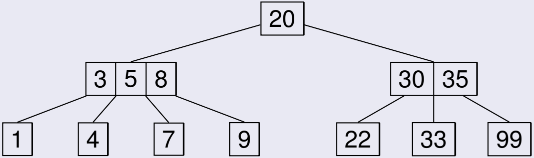


(a,b)-strom (Bayer, McCreight [1])

Vlastnosti

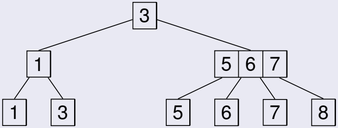
- a, b jsou celá čísla splňující $a \geq 2$ a $b \geq 2a - 1$
- (a,b)-strom je vyhledávací strom
- Všechny vnitřní vrcholy kromě kořene mají alespoň a synů a nejvýše b synů
- Kořen má nejvýše b synů
- Všechny listy jsou ve stejné výšce
- Pro zjednodušení uvažujeme, že prvky jsou jen v listech

Příklad: (2,4)-strom

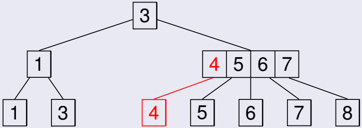


(a,b)-strom: Operace Insert

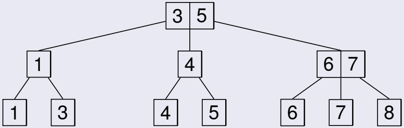
Vložte prvek s klíčem 4 do následujícího (2,4)-stromu



Nejprve najdeme správného otce, jemuž přidáme nový list



Opakovaně rozdělujeme vrchol na dva



(a,b)-strom: Operace Insert

Algoritmus

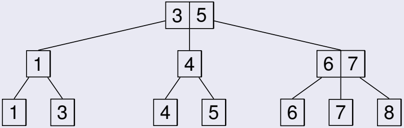
Najít otce v, kterému nový prvek patří
Přidat nový list do v
while $\deg(v) > b$ **do**
    `# Najdeme otce` u `vrcholu` v
    if v *je kořen* **then**
        Vytvořit nový kořen u s jediným synem v
    else
        $u \leftarrow$ otec v
    `# Rozdělíme vrchol` v `na` v `a` v'
    Vytvořit nového syna v' otci u a umístit jej vpravo vedle v
    Přesunout nejpravějších $\lfloor (b+1)/2 \rfloor$ synů vrcholu v do v'
    Přesunout nejpravějších $\lfloor (b+1)/2 \rfloor - 1$ klíčů vrcholu v do v'
    Přesunout poslední klíč vrcholu v do u
    $v \leftarrow u$

Časová složitost

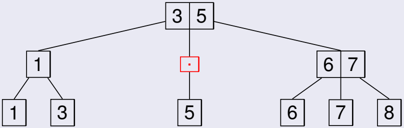
Lineární ve výšce stromu (předpokládáme, že a, b jsou pevné parametry)

Musíme ještě dokázat, že po provedení všech operací doopravdy dostaneme (a,b)-strom. Ověříme, že rozdělené vrcholy mají alespoň a synů (ostatní požadavky jsou triviální). Rozdělovaný vrchol má na počátku právě $b+1$ synů a počet synů po rozdělení je $\left\lfloor\frac{b+1}{2}\right\rfloor$ a $\left\lceil\frac{b+1}{2}\right\rceil$. Protože $b \geq 2a-1$, počet synů po rozdělení je alespoň $\left\lfloor\frac{b+1}{2}\right\rfloor \geq \left\lfloor\frac{2a-1+1}{2}\right\rfloor = \lfloor a \rfloor = a$.

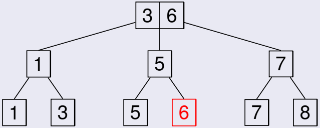
Smažte prvek s klíčem 4 z následujícího (2,4)-stromu

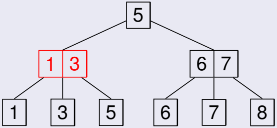


Nalezneme a smažeme list



Přesuneme jedno syna od bratra nebo spojíme vrchol s bratrem





Algoritmus

```
Najít list l obsahující prvek s daným klíčem
v ← otec l
Smazat l
while deg(v) < a & v není kořen do
    u ← sousední bratr v
    if deg(u) > a then
        Přesunout správného syna u pod v 1
    else
        Přesunout všechny syny u pod v 2
        Smazat u
        if v nemá žádného bratra then
            Smazat kořen (otec v) a nastavit v jako kořen
        else
            v ← otec v
```

1. Při přesunu je nutné upravit klíče ve vrcholech u, v a jejich otci.
2. Vrchol u měl a, vrchol v měl $a - 1$ synů. Po jejich sjednocení máme vrchol s $2a - 1 \leq b$ syny.

(a,b)-strom: Analýza

Výška

- (a,b)-strom výšky d má alespoň a^{d-1} a nejvýše b^d listů.
- Výška (a,b)-stromu splňuje $\log_b n \leq d \leq 1 + \log_a n$.

Složitost

Časová složitost operací Find, Insert and Delete je $\mathcal{O}(\log n)$.

Počet modifikovaných vrcholů při vytvoření stromu operací Insert

- Vytváříme (a,b)-strom pomocí operace Insert
- Zajímá nás celkový počet vyvažovacích operací ①
- Při každém štěpení vrcholu vytvoříme nový vnitřní vrchol
- Po vytvoření má strom nejvýše n vnitřních vrcholů
- Celkový počet štěpení je nejvýše n a počet modifikací vrcholů je $\mathcal{O}(n)$
- Amortizovaný počet modifikovaných vrcholů na jednu operaci Insert je $\mathcal{O}(1)$

1. Při jedné vyvažovací operaci (štěpení vrcholu) je počet modifikovaných vrcholů omezený konstantou (štěpený vrchol, otec a synové). Asymptoticky jsou počty modifikovaných vrcholů a vyvažovacích operací stejné.

(a,b)-strom: Paralelní přístup

Cíl

Umožnit efektní paralelizaci operací Find, Insert a Delete (předpoklad: $b \geq 2a$).

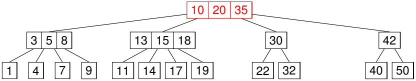
Operace Insert

Preventivně rozdělit každý vrchol na cestě od kořene k hledanému listu s b syny na dva vrcholu.

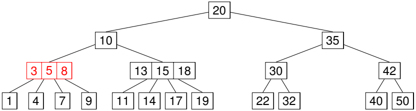
Operace Delete

Preventivně sloučit každý vrchol na cestě od kořene k hledanému listu s a syny s bratrem nebo přesunout synovce.

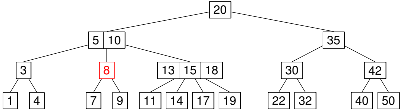
- Vložte prvek s klíčem 6 do následujícího (2,4)-stromu



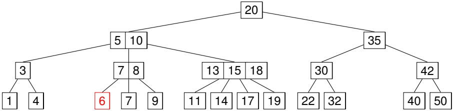
- Nejprve rozdělíme kořen



- Pak pokračujeme do levého syna, který taky rozdělíme



- Vrchol s klíčem 8 není třeba rozdělovat a nový klíč můžeme vložit



A-sort (Guibas, McCreight, Plass, Roberts [10])

Cíl

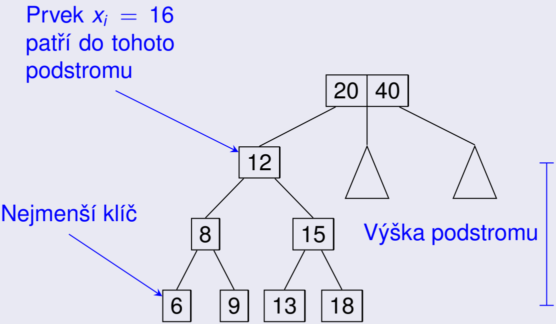
Setřídit „skoro“ setříděné pole

Modifikace (a,b)-stromu

Máme uložený ukazatel na vrchol s nejmenším klíčem

Příklad: Vložte klíč s hodnotou $x_i = 16$

- Začneme od vrcholu s nejmenším klíčem a postupujeme ke kořeni, dokud x_i nepatří podstromu aktuálního vrcholu
- V rámci tohoto podstromu spustíme operaci Insert
- Výška podstromu je $\Theta(\log f_i)$, kde f_i je počet klíčů menších než x_i



A-sort: Algoritmus

Input: Posloupnost $x_1, x_2, \ldots, x_n$

$T \leftarrow$ prázdný (a,b)-strom
for $i \leftarrow n$ **to** 1 `# Prvky procházíme od konce`
 do
  `# Najdeme podstrom, do kterého vložíme` x_i
  $v \leftarrow$ list s nejmenším klíčem
  while v *není kořen a* x_i *je větší než nejmenší klíč v otci vrcholu* v **do**
   $v \leftarrow$ otec v
  Vložíme x_i do podstromu vrcholu v

Output: Projdeme celý strom a vypíšeme všechny klíče (in-order traversal)

A-sort: Složitost (Brown, Tarjan [3], Mehlhorn [17])

Nerovnost mezi aritmetickým a geometrickým průměrem

Jestliže $a_1, \dots, a_n$ nezáporná reálná čísla, pak platí

$$\frac{\sum_{i=1}^n a_i}{n} \geq \sqrt[n]{\prod_{i=1}^n a_i}.$$

Časová složitost

1. Nechť $f_i = |\{j > i; x_j < x_i\}|$ je počet klíčů menších než x_i, které již jsou ve stromu při vkládání x_i
2. Nechť $F = |\{(i,j);\ i > j, x_i < x_j\}| = \sum_{i=1}^n f_i$ je počet inverzí
3. Složitost nalezení podstromu, do kterého x_i patří: $\mathcal{O}(\log f_i)$
4. Nalezení těchto podstromů pro všechny podstromy $\sum_i \log f_i = \log \prod_i f_i = n \log \sqrt[n]{\prod_i f_i} \leq n \log \frac{\sum_i f_i}{n} = n \log \frac{F}{n}$. ①
5. Rozdělování vrcholů v průběhu všech operací Insert: $\mathcal{O}(n)$
6. Celková složitost: $\mathcal{O}(n + n\log(F/n))$
7. Složitost v nejhorším případě: $\mathcal{O}(n \log n)$ protože $F \leq \binom{n}{2}$
8. Jestliže $F \leq n \log n$, pak složitost je $\mathcal{O}(n \log\log n)$ ②

1. Místo AG nerovnosti můžeme použít Jensenovu nerovnost, ze které přímo plyne $\frac{\sum_i \log f_i}{n} \leq \log \frac{\sum_i f_i}{n}$.
2. Tento algoritmus je bohužel efektivní jen pro "hodně skoro"setříděné posloupnosti. Jestliže počet inverzí je $n^{1+\epsilon}$, pak dostáváme složitost třídění $\mathcal{O}(n \log n)$, kde ϵ je libovolně malé kladné číslo.

(a,b)-strom: Závěr

Počet modifikovaných vrcholů při operací Insert a Delete [12]

- Předpoklad: $b \geq 2a$
- Počet modifikovaných vrcholů při l operacích Insert a k Delete je $\mathcal{O}(k + l + \log n)$
- Amortizovaný počet modifikovaných vrcholů při operacích Insert a Delete je $\mathcal{O}(1)$

Podobné datové struktury

- B-tree, B+ tree, B* tree
- 2-4-tree, 2-3-4-tree, etc.

Aplikace

- File systems např. Ext4, NTFS, HFS+
- Databáze

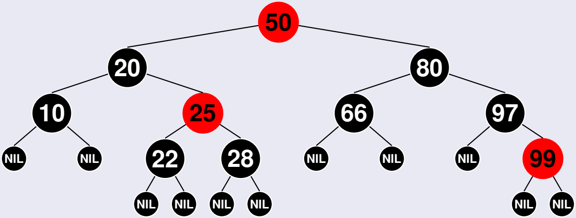
Obsah

Červeno-černé stromy (Guibas, Sedgewick [11])

Definice

1. Binární vyhledávací strom s prvky uloženými ve všech vrcholech
2. Každý vrchol je černý nebo červený
3. Všechny cesty od kořene do listů obsahují stejný počet černých vrcholů
4. Otec červeného vrcholu musí být černý
5. Listy jsou černé ①

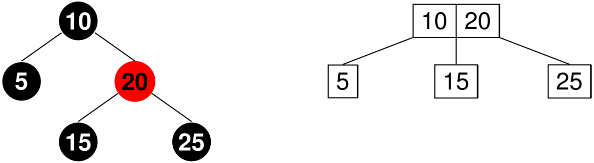
Příklad



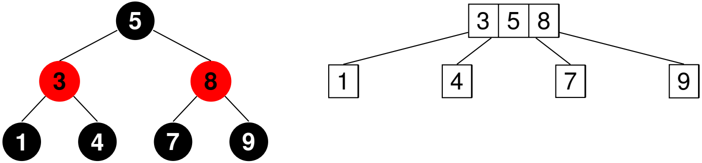
1. Nepovinná podmínka, která jen zjednodušuje operace. V příkladu uvažujeme, že listy jsou reprezentovány NIL/NULL ukazateli, a tedy imaginární vrcholy bez prvků. Někdy se též vyžaduje, aby kořen byl černý, ale tato podmínka není nutná, protože kořen můžeme vždy přebarvit na černo bez porušení ostatních podmínek.

- Vrchol bez červených synů

- Vrchol s jedním čevneným synem 1



- Vrchol s dvěma červenými syny

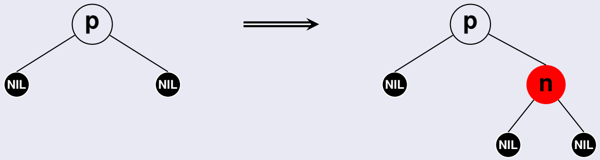


1. Převod mezi červeno-černými stromy a (2,4)-stromy není jednoznačný, protože vrchol (2,4)-stromu se třemi syny a prvky $x < y$ lze převést na černý vrchol červeno-černého stromu s prvkem x a pravým červeným synem y nebo s prvkem y a levým červeným synem x.

Červeno-černé stromy: Operace Insert

Vytvoření nového vrcholu

- Najít list pro nový prvek *n*
- Přidat nový vrchol



- Pokud otec *p* je červený, pak je nutné strom vybalancovat

Balancování

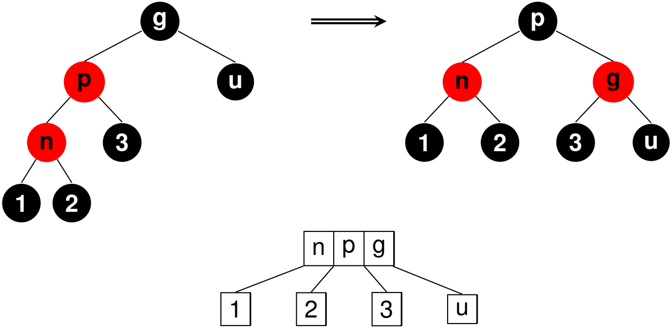
- Vrchol *n* a jeho otec *p* jsou červené vrcholy a toto je jediná porušená podmínka
- Děda *g* vrcholu *n* je černý

Musíme uvažovat tyto případy:

- Strýc *u* je černý nebo červený
- Vrchol *n* je pravým nebo levým synem *p* (podobně pro vrchol *p*) ①

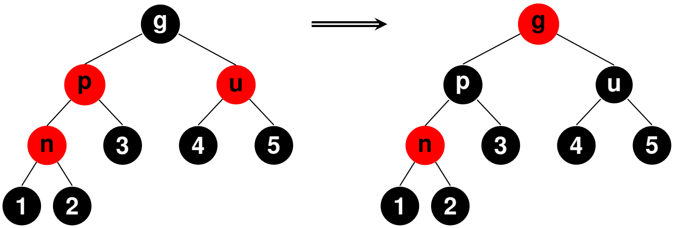
1. S využitím symetrií lze počet případů snížit.

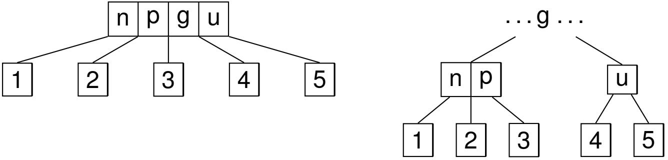
Červeno-černé stromy: Operace Insert, strýc je černý



Pořadí prvků v (2,4)-stromu a výsledný červeno-černý strom závisí na tom, zda vrchol *n* je pravým nebo levým synem *p* a zda vrchol *p* je pravým nebo levým synem *g*.

Červeno-černé stromy: Operace Insert, strýc je červený





Po rozdělení vrchol (2,4)-stromu se prvek *g* přesouvá do otce, a proto je vrchol *g* červený.

Červeno-černé stromy: Vlastnosti

Důsledky ekvivalence s (2,4)-stromy

- Výška červeno-černého stromu je $\Theta(\log n)$ ①
- Časová složitost operací Find, Insert a Delete je $\mathcal{O}(\log n)$
- Amortizovaný počet modifikovaných vrcholů při operací Insert a Delete je $\mathcal{O}(1)$
- Paralelní přístup (top-down balancování)

Aplikace

- Asociativní pole např. std::map and std::set v C++, TreeMap v Java
- The Completely Fair Scheduler in the Linux kernel
- Computational Geometry Data structures

1. Počet černých vrcholů na cestě ke kořeni je stejný jako výška odpovídajícího (2,4)-stromu, a tedy výška červeno-černého stromu je nejvýše dvojnásobek výšky (2,4)-stromu.

Obsah

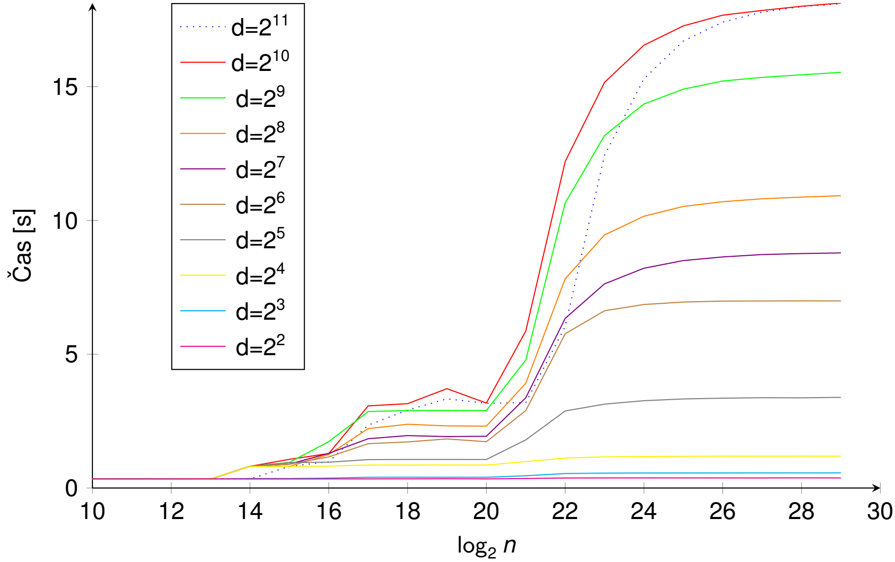
Příklad velikostí a rychlostí různých typů pamětí

	velikost	rychlost
L1 cache	32 KB	223 GB/s
L2 cache	256 KB	96 GB/s
L3 cache	8 MB	62 GB/s
RAM	32 GB	23 GB/s
SDD	112 GB	448 MB/s
HDD	2 TB	112 MB/s

Triviální program

```
# Inicializace pole 32-bitových čísel velikosti n
for (i=0; i+d<n; i+=d) do
    A[i] = i+d # Vezmeme každou d-tou pozici a vytvoříme cyklus
A[i]=0, i=0
# Měříme dobu průběhu cyklu v závislosti na parametrech n a d
# Počet operací je nezávislý na n a d
for (j=0; j< 2^28; j++) do
    i = A[i] # Dokola procházíme cyklus d-tých pozic
```

Paměťová hierarchie: Triviální program



Cache-oblivious (Frigo, Leiserson, Prokop, Ramachandran, 1999 [9])

Zjednodušený model paměti

- Uvažujme pouze na dvě úrovně paměti: pomalý disk a rychlá cache
- Paměť je rozdělená na bloky (stránky) velikosti B ①
- Velikost cache je M, takže cache má $P = \frac{M}{B}$ bloků
- Procesor může přistupovat pouze k datům uložených v cache
- Paměť je plně asociativní ②
- Data se mezi diskem a cache přesouvají po celých blocích a našim cílem je určit počet bloků načtených do cache

Cache-aware algoritmus

Algoritmus zná hodnoty M a B a podle nich nastavuje parametry (např. velikost vrcholu B-stromu při ukládání dat na disk).

Cache-oblivious algoritmus

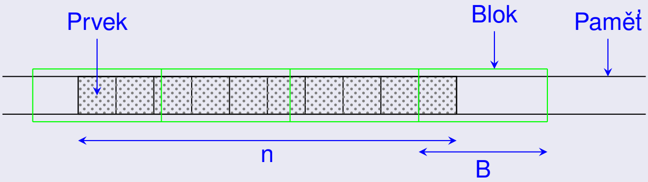
Algoritmus musí efektivně fungovat bez znalostí hodnot M a B. Důsledky:

- Není třeba nastavovat parametry programu, který je tak přenositelnější
- Algoritmus dobře funguje mezi libovolnými úrovněmi paměti (L1 – L2 – L3 – RAM)

1. Pro zjednodušení předpokládáme, že jeden prvek zabírá jednotkový prostor, takže do jednoho bloku se vejde B prvků.
2. Předpokládáme, že každý blok z disku může být uložený na libovolné pozici v cache. Tento předpoklad výrazně zjednodušuje analýzu, i když na reálných počítačích moc neplatí, viz `https://en.wikipedia.org/wiki/CPU_cache#Associativity`.

Cache-oblivious analýza: Scanning

Přečtení souvislého pole (výpočet maxima, součtu a podobně)

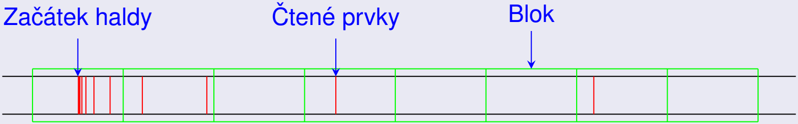


- Minimální možný počet přenesených bloků je $\lceil n/B \rceil$.
- Skutečný počet přenesených bloků je nejvýše $\lceil n/B \rceil + 1$.
- Předpokládáme, že máme k dispozici $\mathcal{O}(1)$ registrů k uložení iterátoru a maxima.

Obrácení pole

Počet přenesených bloků je stejný za předpokladu, že $P \geq 2$.

Binární halda v poli: Průchod od listu ke kořeni



1. Cesta má $\Theta(\log n)$ vrcholů
2. Posledních $\Theta(\log B)$ vrcholů leží v nejvýše dvou blocích
3. Ostatní vrcholy jsou uloženy v po dvou různých blocích
4. $\Theta(\log n - \log B) = \Theta(\log \frac{n}{B})$ přenesených bloků ①

Binární vyhledávání

- Porovnáváme $\Theta(\log n)$ prvků s hledaným prvkem ②
- Posledních $\Theta(\log B)$ prvků je uloženo v nejvýše dvou blocích
- Ostatní prvky jsou uloženy v po dvou různých blocích
- $\Theta(\log n - \log B)$ přenesených bloků

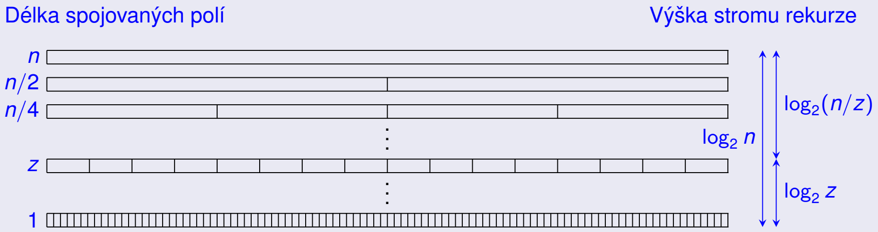
1. Přesněji $\Theta(\max\{1, \log n - \log B\})$. Dále předpokládáme, že $n \geq B$.
2. Pro jednoduchost uvažujeme neúspěšné vyhledávání.

Cache-oblivious analýza: Mergesort

Případ $n \leq M/2$

Celé pole se vejde do cache, takže přenášíme $2n/B + \mathcal{O}(1)$ bloků. ①

Schéma



Případ $n > M/2$

1. Nechť z je maximální velikost pole, která může být setříděná v cache ②
2. Platí $z \leq \frac{M}{2} < 2z$
3. Slití jedné úrovně vyžaduje $2\frac{n}{B} + 2\frac{n}{z} + \mathcal{O}(1) = \mathcal{O}\left(\frac{n}{B}\right)$ přenosů. ③
4. Počet přenesených bloků je $\mathcal{O}\left(\frac{n}{B}\right)\left(1 + \log_2 \frac{n}{z}\right) = \mathcal{O}\left(\frac{n}{B} \log \frac{n}{M}\right)$. ④

1. Polovina cache je použita na vstupní pole a druhá polovina na slité pole.
2. Pro jednoduchost předpokládáme, že velikosti polí v jedné úrovni rekurze jsou stejné. z odpovídá velikosti pole v úrovni rekurze takové, že dvě pole velikost $z/2$ mohou být slity v jedno pole velikost z.
3. Slití všech polí v jedné úrovni do polovičního počtu polí dvojnásobné délky vyžaduje přečtení všech prvků. Navíc je třeba uvažovat nezarovnání polí a bloků, takže hraniční bloky mohou patřit do dvou polí.
4. Funnelsort přenese $\mathcal{O}\left(\frac{n}{B}\log_P \frac{n}{B}\right)$ bloků.

Cache-oblivious analýza: Transpozice matic: Triviální přístup

Strategie pro výměnu stránek v cache

- OPT: Optimální off-line algoritmus předpokládající znalost všech přístupů do paměti
- FIFO: Z cache smažeme stránku, která je ze všech stránek v cachi nejdelší dobu
- LRU: Z cache smažeme stránku, která je ze všech stránek v cachi nejdéle nepoužitá

Triviální algoritmus pro transpozici matice A velikost $k \times k$

for $i \leftarrow 1$ **to** k **do**
    for $j \leftarrow i + 1$ **to** k **do**
        Swap(A_{ij}, A_{ji})

Předpoklady

Uvažujeme pouze případ

- $B < k$: Do jednoho bloku cache se nevejde celá řádka matice
- $P < k$: Do cache se nevejde celý sloupec matice

Cache-oblivious analýza: Transpozice matic: Triviální přístup

Příklad: Representace matice 5×5 v paměti

11	12	13	14	15	21	22	23	24	25	31	32	33	34	35	41	42	43	44	45	51	52	53	54	55

LRU a FIFO strategie

Při čtení matice po sloupcích si cache pamatuje posledních P řádků, takže při čtení prvku $A_{3,2}$ již prvek $A_{3,1}$ není v cache. Počet přenesených bloků je $\Omega(k^2)$.

OPT strategie

1. Transpozice prvního řádku/sloupce vyžaduje alespoň $k-1$ přenosů.
2. Nejvýše P prvků z druhého sloupce zůstane v cache.
3. Proto transpozice druhého řádku/sloupce vyžaduje alespoň $k-P-2$ přenosů.
4. Transpozice i-tého řádku/sloupce vyžaduje alespoň $\max\{0, k-P-i\}$ přenosů.
5. Celkový počet přenosu je alespoň $\sum_{i=1}^{k-P} k-P-i = \Omega\left((k-P)^2\right)$.

Cache-aware algoritmus pro transpozici matice A velikost $k \times k$

```
# Nejprve si rozdělíme danou matici na submatice velikosti z × z
for (i = 0; i < k; i+ = z) do
    for (j = i; j < k; j+ = z) do
        # Transponujeme submatici začínající na pozici (i, j)
        for (ii = i; ii < min(k, i + z); ii + +) do
            for (jj = max(j, ii + 1); jj < min(k, j + z); jj + +) do
                Swap(A_{ii,jj}, A_{jj,ii})
```

Hodnocení

- Optimální hodnota z závisí na konkrétním počítači
- Využíváme jen jednu úroveň cache
- Při správně zvolené hodnotě z bývá tento postup nejrychlejší

Idea

Rekurzivně rozdělíme na submatice

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad A^T = \begin{pmatrix} A_{11}^T & A_{21}^T \\ A_{12}^T & A_{22}^T \end{pmatrix}$$

Matice A_{11} a A_{22} se transponují podle stejného schématu, ale A_{12} a A_{21} se prohazují.

Cache-oblivious analýza: Transpozice matic: Rekurzivní transpozice

Procedure `transpose_on_diagonal` (A)
    if *Matice A je malá* **then**
        Transponujeme matici A triviálním postupem
    else
        $A_{11}, A_{12}, A_{21}, A_{22} \leftarrow$ souřadnice submatic
        `transpose_on_diagonal` (A_{11})
        `transpose_on_diagonal` (A_{22})
        `transpose_and_swap` (A_{12}, A_{21})

Procedure `transpose_and_swap` (A, B)
    if *Matice A a B jsou malé* **then**
        Prohodíme a transponujeme matice A a B triviálním postupem
    else
        $A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22} \leftarrow$ souřadnice submatic
        `transpose_and_swap` (A_{11}, B_{11})
        `transpose_and_swap` (A_{12}, B_{21})
        `transpose_and_swap` (A_{21}, B_{12})
        `transpose_and_swap` (A_{22}, B_{22})

- Všimněme si, že matice A a B musí mít posice symetrické podle hlavní diagonály původní matice, a proto ve skutečnosti funkci `transpose_and_swap()` stačí předávat pozice matici A.
- Ve funkci `transpose_on_diagonal` musí být matice A čtvercová a ležet na hlavní diagonále, a proto stačí předávat x-ovou souřadnici a řád matice.

Analýza počtu přenesených bloků

1. Předpoklad „Tall cache“: $M \geq 4B^2$, tj. počet bloků je alespoň $4B$ ①
2. Nechť z je maximální velikost submatice, ve které se jeden řádek vejde do jednoho bloku ②
3. Platí: $z \leq B \leq 2z$
4. Jedna submatice $z \times z$ je uložena v nejvýše $2z \leq 2B$ blocích
5. Dvě submatice $z \times z$ se vejdou do cache ③
6. Transpozice matice typu $z \times z$ vyžaduje nejvýše $4z$ přenosů
7. Máme $(k/z)^2$ submatic velikosti z
8. Celkový počet přenesených bloků je nejvýše $\frac{k^2}{z^2} \cdot 4z \leq \frac{8k^2}{B} = \mathcal{O}\left(\frac{k^2}{B}\right)$
9. Tento postup je optimální až na multiplikativní faktor ④

1. Stačilo by předpokládat, že počet bloků je alespoň $\Omega(B)$. Máme-li alespoň $4B$ bloků, pak je postup algebraicky jednodušší.
2. Pokud začátek řádky není na začátku bloku, tak je jeden řádek submatice uložen ve dvou blocích.
3. Funkce `transpose_and_swap` pracujeme se dvěma submaticemi.
4. Celá matice je uložena v alespoň $\frac{k^2}{B}$ blocích paměti.

Cache-oblivious analýza: Reprezentace binárních stromů

Cíl

Sestrojit reprezentaci binárního stromu efektivně využívající cache.
Počítáme počet načtených bloků při průchodu cesty z listu do kořene.

Binární halda

Velmi neefektivní: Počet přenesených bloků je $\Theta(\log n - \log B) = \Theta(\log \frac{n}{B})$

B-regulární halda, B-strom

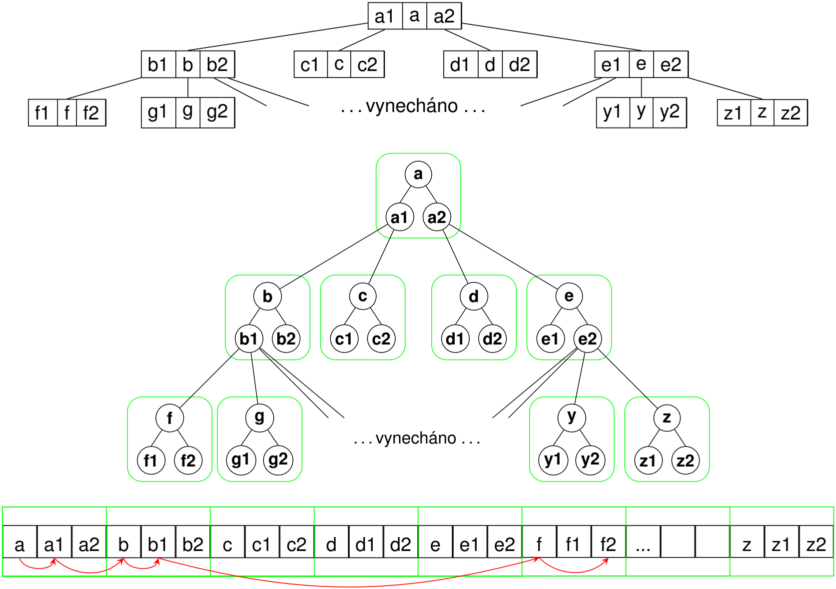
- Výška stromu je $\log_B(n) + \Theta(1)$ ①
- Jeden vrchol je uložen v nejvýše dvou blocích
- Počet načtených bloků je $\Theta(\log_B(n))$ ②
- Nevýhody: cache-aware a chtěli jsme binární strom

Převedení na binární strom

Každý vrchol B-regulární haldy nahradíme binárním stromem.

1. Platí pro B-regulární haldu. B-strom má výšku $\Theta(\log_B(n))$.
2. Asymptoticky optimální řešení — důkaz je založen na Information theory.

Cache-oblivious analýza: Reprezentace binárních stromů

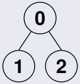


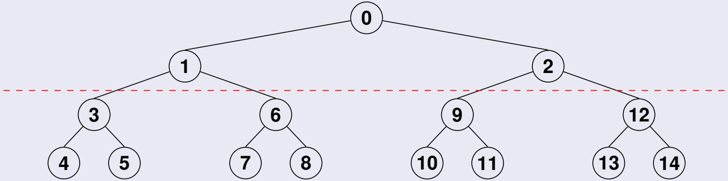
Cesta z kořene do listu *f2*

Rekurzivní „bottom-up“ konstrukce van Emde Boas rozložení

- van Emde Boas rozložení vEB_0 řádu 0 je jeden vrchol
- vEB_k obsahuje jednu „horní“ kopii vEB_{k-1} a každému listu „horní“ kopie má dvě „dolní“ kopie vEB_{k-1}
- V poli jsou nejprve uložena „horní“ kopie a pak následují všechny „dolní“ kopie

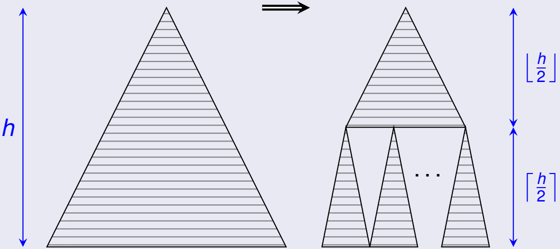
Pořadí vrcholů v poli podle van Emde Boas rozložení





Cache-oblivious analýza: Reprezentace binárních stromů

Rekurzivní „top-down“ konstrukce van Emde Boas rozložení



Výpočet počtu načtených bloků při cestě z kořene do listu

- Nechť $h = \log_2 n$ je výška stromu
- Nechť z je maximální výška podstromu, který se vejde do jednoho bloku
- Platí: $z \leq \log_2 B \leq 2z$
- Počet podstromů výšky z na cestě z kořene do listu je $\frac{h}{z} \leq \frac{2 \log_2 n}{\log_2 B} = 2 \log_B n$
- Počet načtených bloků je $\Theta(\log_B n)$

Cache-oblivious analýza: Srovnání OPT a LRU strategií

Věta (Sleator, Tarjan [25])

- Nechť $s_1, \ldots, s_k$ je posloupnost přístupů do paměti ①
- Nechť P_{OPT} a P_{LRU} je počet bloků v cache pro strategie OPT a LRU ②
- Nechť F_{OPT} a F_{LRU} je počet přenesených bloků ③
- $P_{\text{LRU}} > P_{\text{OPT}}$

Pak $F_{\text{LRU}} \leq \frac{P_{\text{LRU}}}{P_{\text{LRU}} - P_{\text{OPT}}} F_{\text{OPT}} + P_{\text{OPT}}$

Důsledek

Pokud LRU může uložit dvojnásobný počet bloků v cache oproti OPT, pak LRU má nejvýše dvojnásobný počet přenesených bloků oproti OPT (plus P_{OPT}). ④

Zdvojnásobení velikosti cache nemá většinou vliv na asymptotický počet přenesených bloků

- Scanning: $\mathcal{O}(n/B)$
- Mergesort: $\mathcal{O}\left(\frac{n}{B} \log \frac{n}{M}\right)$
- Funnelsort: $\mathcal{O}\left(\frac{n}{B} \log_P \frac{n}{B}\right)$
- The van Emde Boas layout: $\mathcal{O}(\log_B n)$

1. s_i značí blok paměti, se kterým program pracuje, a proto musí být načten do cache. Posloupnost $s_1, \dots, s_k$ je pořadí bloků paměti, ve kterém algoritmus pracuje s daty. Při opakovaném přístupu do stejného bloku se blok posloupnosti opakuje.
2. Představme si, že OPT strategie pustíme na počítači s P_{OPT} bloky v cache a LRU strategie spustíme na počítači s P_{OPT} bloky v cache.
3. Srovnáváme počet přenesených bloků OPT strategie na počítači s P_{OPT} bloky a LRU strategie na počítači s P_{OPT} bloky.
4. Formálně: Jestliže $P_{\mathrm{LRU}} = 2P_{\mathrm{OPT}}$, pak $F_{\mathrm{LRU}} \leq 2F_{\mathrm{OPT}} + P_{\mathrm{OPT}}$.

Důkaz ($F_{\text{LRU}} \leq \frac{P_{\text{LRU}}}{P_{\text{LRU}}-P_{\text{OPT}}} F_{\text{OPT}} + P_{\text{OPT}}$)

1. Pokud LRU má $f \leq P_{\text{LRU}}$ přenesených bloků v podposloupnosti s, pak OPT přenese alespoň $f - P_{\text{OPT}}$ bloků v podposlopnosti s
   - Pokud LRU načte v podposloupnost f různých bloků, tak podposloupnost obsahuje alespoň f různých bloků
   - Pokud LRU načte v podposloupnost jeden blok dvakrát, tak podposloupnost obsahuje alespoň $P_{\text{LRU}} \geq f$ různých bloků
   - OPT má před zpracováním podposloupnosti nejvýše P_{OPT} bloků z podposloupnosti v cache a zbylých alespoň $f - P_{\text{OPT}}$ musí načíst
2. Rozdělíme posloupnost $s_1, \ldots, s_k$ na podposlopnosti tak, že LRU přenese P_{LRU} bloků v každé podposloupnosti (kromě poslední)
3. Jestliže F'_{OPT} and F'_{LRU} jsou počty přenesených bloků při zpracování libovolné podposloupnosti, pak $F'_{\text{LRU}} \leq \frac{P_{\text{LRU}}}{P_{\text{LRU}}-P_{\text{OPT}}} F'_{\text{OPT}}$ (kromě poslední)
   - OPT přenese $F'_{\text{OPT}} \geq P_{\text{LRU}} - P_{\text{OPT}}$ bloků v každé podposloupnosti
   - Tedy $\frac{F'_{\text{LRU}}}{F'_{\text{OPT}}} \leq \frac{P_{\text{LRU}}}{P_{\text{LRU}}-P_{\text{OPT}}}$
4. V poslední posloupnosti platí $F''_{\text{LRU}} \leq \frac{P_{\text{LRU}}}{P_{\text{LRU}}-P_{\text{OPT}}} F''_{\text{OPT}} + P_{\text{OPT}}$
   - Platí $F''_{\text{OPT}} \geq F''_{\text{LRU}} - P_{\text{OPT}}$ a $1 \leq \frac{P_{\text{LRU}}}{P_{\text{LRU}}-P_{\text{OPT}}}$
   - Tedy $F''_{\text{LRU}} \leq F''_{\text{OPT}} + P_{\text{OPT}} \leq \frac{P_{\text{LRU}}}{P_{\text{LRU}}-P_{\text{OPT}}} F''_{\text{OPT}} + P_{\text{OPT}}$

Srovnání rychlosti čtení a zápisu z paměti

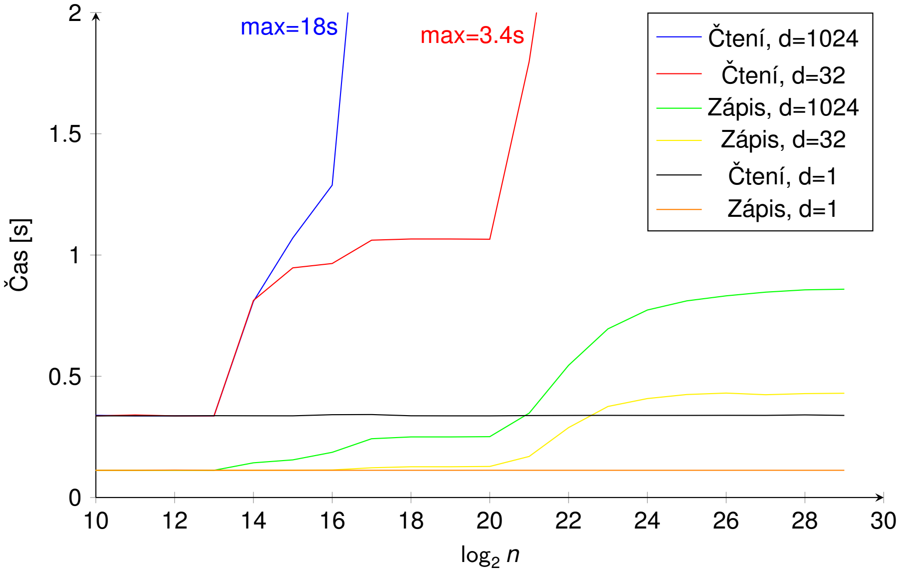
Čtení z paměti

```
# Inicializace pole 32-bitových čísel velikosti n
for (i=0; i+d<n; i+=d) do
    A[i] = i+d # Vezmeme každou d-tou pozici a vytvoříme cyklus
A[i=0]=0
# Měříme dobu průběhu cyklu v závislosti na parametrech n a d
for (j=0; j< 2^28; j++) do
    i = A[i] # Dokola procházíme cyklus d-tých pozic
```

Zápis do paměti

```
# Měříme dobu průběhu cyklu v závislosti na parametrech n a d
for (j=0; j< 2^28; j++) do
    A[(j*d) % n] = j # Dokola zapisujeme na d-té pozice
```

Srovnání rychlosti čtení a zápisu z paměti



Která varianta je rychlejší a o kolik?

```
# Použijeme modulo:
for (j=0; j< 2^28; j++) do
    A[(j*d) % n] = j
# Použijeme bitovou konjunkci:
mask = n − 1 # Předpokládáme, že n je mocnina dvojky
for (j=0; j< 2^28; j++) do
    A[(j*d) & mask] = j
```

Jak dlouho poběží výpočet vynecháme-li poslední řádek?

```
for (i=0; i+d<n; i+=d) do
    A[i] = i+d
A[i=0]=0
# Měříme dobu průběhu cyklu v závislosti na parametrech n a d
for (j=0; j< 2^28; j++) do
    i = A[i]
printf("%d\n", i);
```

Obsah

Halda

Základní pojmy

Každý prvek obsahuje

- jednoznačný a neměnný klíč identifikující prvek a
- prioritu, která nemusí být jednoznačná a může se měnit.

Základní operace

- INSERT
- FINDMIN: Nalezení prvku s nejmenší prioritou
- DELETEMIN: Smazání prvku s nejmenší prioritou
- DECREASE: Snížit hodnotu priority v daném vrcholu

Haldový invariant

Priorita v každém vrcholu větší nebo rovna prioritě otce.

Halda: Aplikace

Aplikace

- Prioritní fronta
- Heap-sort
- Dijkstrův algoritmus (nejkratší cesta)
- Jarníkův (Primův) algoritmus (minimální kostra)

Klíč

- Umístění prvků ve stromu nemusí splňovat podmínku vyhledávání ①
- Halda nemusí umět efektivně vyhledávat prvky podle klíče!
- Algoritmus využívající haldu si musí pamatovat, kde je který prvek uložený. ②
- Halda hodnoty klíčů vůbec nevyužívá ③

1. Podmínka vyhledávacích stromů (klíč v každém vnitřním vrcholu je větší než všechny klíče v levém podstromu a menší než všechny klíče v pravém podstromu) není v haldě splněna.
2. Přesněji: pozici prvku je nutné si pamatovat, pokud potřebujeme operaci DECREASE. Operace INSERT, FINDMIN a DELETEMIN pozice prvků nepotřebují. Algoritmus si například může pamatovat ukazatel na vrchol stromu obsahující daný prvek.
3. Operace FINDMIN vrací prvek i s klíčem, který může být využit v dalším algoritmu.

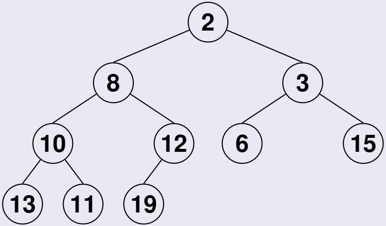
Obsah

d-regulární halda (Johnson [13])

Definice

- Každý vrchol má nejvýše d synů
- Všechny vrstvy kromě poslední jsou úplně zaplněné
- Poslední hladina je zaplněná zleva
- Haldový invariant (priorita v každém vrcholu větší nebo rovna prioritě v otci)

Příklad 2-regulární (binární) haldy

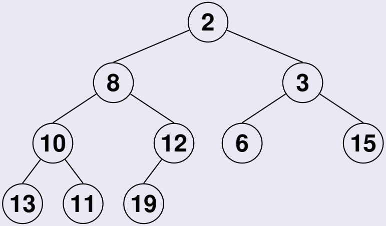


Cvičení

Jaká je přesná výška d-regulární haldy s n prvky? ①

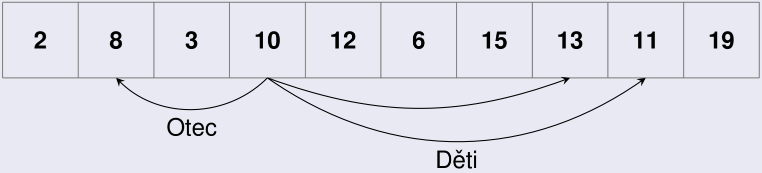
1. Nechť h je nejnižší plná hladina. Jelikož h-tá hladina obsahuje d^h vrcholů, tak platí $n \geq d^h$, z čehož plyne $h \leq \log_d n$. Tudíž výška d-regulární haldy s n prvky je nejvýše $1 + \log_d n$. Najděte formuli udávající přesnou výšku d-regulární haldy.

Binární halda uložená ve stromu



Binární halda uložená v poli

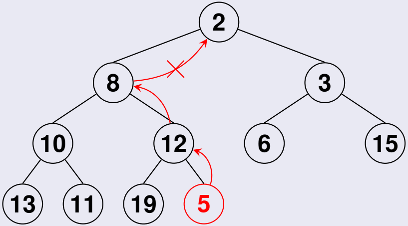
A vrchol na pozici i má otce na pozici $\lfloor (i-1)/2 \rfloor$ a syny na pozici $2i+1$ a $2i+2$:



Cvičení: Určete pozice otce a synů pro obecnou *d*-regulární haldu

d-regulární halda: Operace INSERT

Příklad: Vložme prvek s prioritou 5



2	8	3	10	12	6	15	13	11	19	5

d-ary heap: Operace INSERT a DECREASE

INSERT: Algoritmus

Input: Nový prvek s prioritou x

$v \leftarrow$ první volný blok v poli
Nový prvek uložíme na pozici v
while *v není kořen a otec p vrcholu v má priority větší než x* **do**
    Prohodíme prvky na pozicích v a p
    $v \leftarrow p$

Operace DECREASE

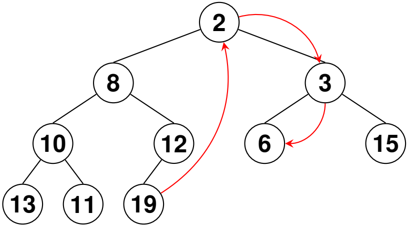
Snížíme prioritu a pokračujeme podobně jako při operaci INSERT

Časová složitost ①

$\mathcal{O}(\log_d n)$

1. Pro přesnější analýzu nás zajímá závislost složitosti na hodnotě d. Později se nám bude hodit nastavovat d podle hodnot na vstupu.

Operace DeleteMin



Algoritmus

Přesuneme poslední prvek do kořene *v*
while *Některý ze synů vrcholu v má prioritu menší než v* **do**
    $u \leftarrow$ syn vrcholu v s menším prioritou
    Prohodíme prvky ve vrcholech u a v
    $v \leftarrow u$

Složitost

$\mathcal{O}(d \log_d n)$

d-regulární halda: Operace Build

Cíl

Vytvořit haldu z daného pole prvků

Algoritmus

for $r \leftarrow$ *poslední pozice* **to** *první pozice v poli* **do**
  `# Zpracujeme vrchol` r `podobně jako při operaci DeleteMin`
  $v \leftarrow r$
  while *Některý ze synů vrcholu v má prioritu menší než v* **do**
    $u \leftarrow$ syn vrcholu v s menším prioritu
    Prohodíme prvky ve vrcholech u a v
    $v \leftarrow u$

Korektnost

Podstromy všech zpracovaných vrcholů tvoří haldu

d-regulární halda: Operace Build

Lemma (Cvičení)

$$\sum_{h=0}^{\infty} \frac{h}{d^h} = \frac{d}{(d-1)^2}$$

Složitost

- Zpracování vrcholu s podstromem výšky h: $\mathcal{O}(dh)$
- Úplný podstrom výšky h má d^h listů ①
- Každý list patří do nejvýše jednoho úplného podstromu výšky h.
- Počet vrcholů s podstromy výšky h je nejvýše $\frac{n}{d^h} + 1 \leq \frac{2n}{d^h}$ ②
- Celková časová složitost

$$\sum_{h=0}^{\lceil \log_d n \rceil} \frac{2n}{d^h} dh \leq 2nd \sum_{h=0}^{\infty} \frac{h}{d^h} = 2n \left(\frac{d}{d-1} \right)^2 \leq 2n2^2 = \mathcal{O}(n)$$

- Složitost je $\mathcal{O}(n)$ pro libovolné d

1. Podstromem vrcholu u rozumíme vrchol u a všechny vrcholy pod u.
2. Člen „$+1$“ započítáváme, protože jeden podstrom může být neúplný.

Obsah

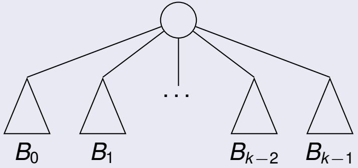
Definice

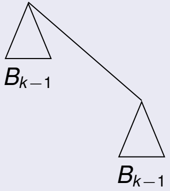
- Binomiální strom B_0 řádu 0 je jeden vrchol
- Binomiální strom B_k řádu $k \geq 1$ má kořen, jehož synové jsou kořeny binomiálních stromů řádu $0, 1, \ldots, k-1$.

Alternativně

Binomální strom řádu k je vytvořen z dvou binomiálních stromů řádu $k-1$ tak, že se jeden strom připojí jako nejpravější syn kořene druhého stromu.

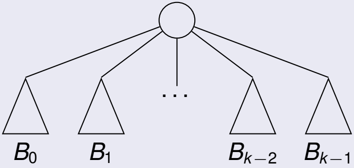
Rekurzivní definice binomiálního stromu

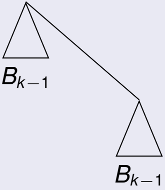




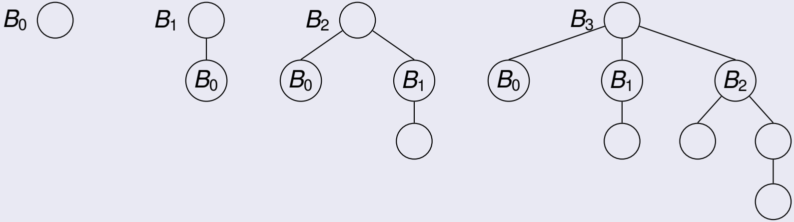
Binomiální strom: Příklad

Rekurzivní definice binomiálního stromu

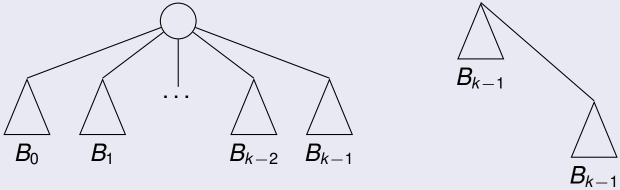




Binomiální stromy řádu 0, 1, 2 a 3



Rekurzivní definice binomiálního stromu



Vlastnosti

Binomiální strom B_k má

- 2^k vrcholů,
- výšku k,
- k synů v kořeni,
- maximální stupeň k,
- $\binom{k}{d}$ vrcholů v hloubce d.

Podstrom vrcholu s k syny je izomorfní B_k.

Množina binomiálních stromů

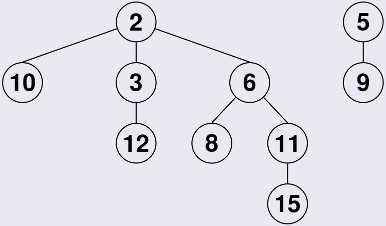
Pozorování

Pro každé n existuje (právě jedna) množina binomiálních stromů různých řádů taková, že celkový počet vrcholů je n.

Vztah mezi binárními čísly a binomiálními stromy

$$\begin{array}{rcccccccccc} \text{Binární číslo } n & = & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ & & \downarrow & & & \downarrow & \downarrow & & \downarrow & \\ \text{Binomiální halda obsahuje:} & & B_7 & & & B_4 & B_3 & & B_1 & \end{array}$$

Příklad pro 1010_2 prvků



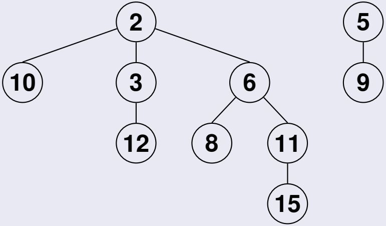
Binomiální halda (Vuillemin [27])

Definice

Binomiální halda je množina binomiálních stromů taková, že:

- Každý prvek je uložen právě v jednom vrcholu jednoho binomiálního stromu
- Každý binomiální strom je halda (otec má menší prioritu než syn)
- Žádné dva binomiální stromy nemají stejný řád

Příklad



Binomiální halda: Výška

Pozorování

Binomiální halda obsahuje nejvýše $\log_2(n+1)$ stromů a každý má výšku nejvýše $\log_2 n$.

Vztah mezi binárními čísly a binomiálními stromy

$$
\begin{array}{rcccccccccc}
\text{Binární číslo } n & = & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
 & & \downarrow & & & \downarrow & \downarrow & & \downarrow & \\
\text{Binomiální halda obsahuje:} & & B_7 & & & B_4 & B_3 & & B_1 &
\end{array}
$$

Binomiální halda: Representace

Struktura pro vrchol binomiálního stromu obsahuje

- prvek (klíč a priorita),
- ukazatel na otce,
- ukazatel na nejlevějšího a nejpravějšího syna,
- ukazatel na levého a pravého bratra a ①
- řád postromu.

Binomiální halda

- Binomální stromy jsou uloženy ve spojovém seznamu pomocí ukazatelů na bratry. ②
- Odstraněním kořene binomiálního stromu vznikne binomiální halda v čase $\mathcal{O}(1)$.
- Binomiální halda si udržuje ukazatel na strom s prvkem s minimální prioritou.

Operace FINDMIN

Triviálně v čase $\mathcal{O}(1)$

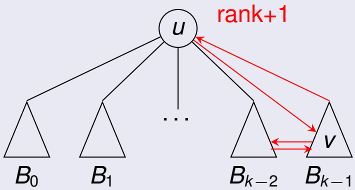
Operace DECREASE

Stejně jako v regulární haldě.

1. Ukazatele tvoří obousměrný spojový seznam synů a tento seznam udržujeme setříděný podle řádu.
2. Binomiální stromy jsou ve spojovém seznamu taky setříděné podle řádu.

Binomiální halda: Spojení dvou binomiálních hald

Spojení dvou binomiálních stromů stejného řádu v čase $\mathcal{O}(1)$



Spojení binomiálních hald

Spojení dvou binomiálních hald je jako sčítání binární čísel: sjednocujeme binomiální stromy od nejmenších. Složitost je $\mathcal{O}(\log n)$, kde n je celkový počet prvků.

Příklad

Binomiální strom	B_6	B_5	B_4	B_3	B_2	B_1	B_0
První halda	0	1	1	0	1	1	0
Druhá halda	0	1	1	0	1	0	0
Spojení	1	1	0	1	0	1	0

Binomiální halda: Operace INSERT a DELETEMIN

Operace INSERT

- Vytvoříme binomiální strom řádu 0 s novým prvkem
- Procházíme seznam stromů od nejmenších: ①
  - Pokud strom B_0 je v haldě, tak jej sjednotíme s novým stromem B_0, čímž vytvoříme B_1
  - Pokud strom B_1 je v haldě, tak jej sjednotíme s novým stromem B_1, čímž vytvoříme B_2
  - Takto pokračujeme až k ke stromu s nejmenším řádem, který není uložený v haldě, a nový strom vložíme do haldy ②
- Složitost v nejhorším případě je $\mathcal{O}(\log n)$
- Amortizovaná složitost je $\mathcal{O}(1)$ podobně jako inkrementace binárního čítače
  - Zde je důležité, že neprocházíme všechny stromu v haldě

Operace DELETEMIN

Odstraníme kořen s minimálním prvkem, čímž vznikne nová binomiální halda, kterou sjednotíme se zbytkem původní haldy v čase $\mathcal{O}(\log n)$.

1. Stromy v haldě udržujeme setříděné podle řádu.
2. Nový strom je nejmenší, takže jej vložíme na začátek seznamu.

Líná binomiální halda

Změna v poctivé binomiální haldě

Líná binomiální halda může obsahovat libovolný počet binomiálních stromů stejného řádu.

Operace INSERT a spojení dvou líných binomiálních hald

- Pouze spojíme seznamy stromů
- Složitost $\mathcal{O}(1)$ v nejhorším případě

Operace DELETEMIN

- Smažeme kořen s minimálním prvkem
- Spojíme seznam synů smazaného kořene s ostatními stromy v haldě
- Zrekonstruujeme poctivou binomiální haldu
- Najdeme nový minimální prvek

Líná binomiální halda: Rekonstrukce poctivé binomiální haldy

Idea

- Dokud máme v haldě binomiální haldy stejného řádu, tak je spojujeme
- Použijeme pole indexované řádem stromu k vyhledávání stromů stejného řádu

Algoritmus

Inicializujeme pole velikosti $\lceil \log_2(n+1) \rceil$ ukazatelem NIL
for *pro každý strom h v líné binomiální haldě* **do**
    $o \leftarrow$ řád stromu h
    while *pole[o]* $\neq$ *NIL* **do**
        $h \leftarrow$ spojení stromů h a pole[o]
        pole[o] $\leftarrow$ NIL
        $o \leftarrow o + 1$
    pole[o] $\leftarrow h$
Pole stromů převedeme na spojový seznam, čímž vytvoříme poctivou binomiální haldu

Cvičení

Amortizovaná složitost operace DELETEMIN je $\mathcal{O}(\log n)$.

Haldy: Přehled

Složitosti různých hald

	Binární	Binomiální		Líná binomiální	
	nejhorší	nejhorší	amortizovaně	nejhorší	amortizovaně
INSERT	$\log n$	$\log n$	1	1	1
DECREASE	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
DELETEMIN	$\log n$	$\log n$	$\log n$	n	$\log n$

Cvičení

Je možné vytvořit haldu, která má amortizovanou složitost operací INSERT a DELETEMIN lepší než $\mathcal{O}(\log n)$?

Další cíl

Zrychlit operaci DECREASE

Postup

V líné binomiální haldě musí každý strom být izomorfní binomiálnímu stromu. Ve Fibonacciho haldě tento požadavek neplatí.

Obsah

Fibonacciho halda (Fredman, Tarjan [8])

Základní vlastnosti a povolené operace ①

- Fibonacciho halda je seznam haldových stromů ②
- Řád stromu je počet synů kořene ③
- Smíme spojit dva stromy stejného řádu ④
- Každému vrcholu kromě kořene smíme odpojit nejvýše jednoho syna
  - Do reprezentace vrcholu přidáme bitovou informaci, zda vrchol již o syna přišel
- Kořen může přijít o libovolný počet synů
  - Stane-li se vrchol kořenem, tak jej odznačíme
  - Je-li kořen připojen do jiného stromu, tak smí ztratit nejvýše jednoho syna, dokud se nestane znovu kořenem
- Smíme vytvořit nový strom s jediným prvkem ⑤
- Smíme smazat kořen stromu ⑥

Operace stejné jako v líné binomiální haldě

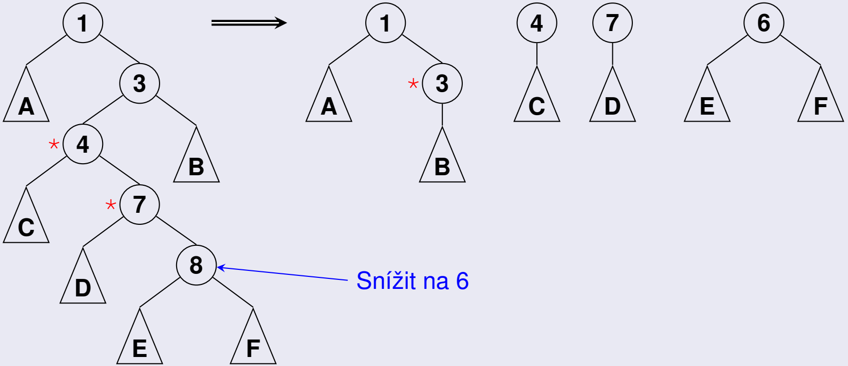
INSERT, FINDMIN, DELETEMIN

1. Doposud probírané datové struktury mají jasně definovanou strukturu a operace jsou navrženy tak, aby tuto strukturu zachovávaly. Fibonacciho halda je definovaná povolenými operacemi a vlastnosti se odvozují z operací.
2. Podobně jako binomiální halda, ale stromy nemusí být izomorfní s binomiálními stromy.
3. Podobně jako binomiální halda, ale vztahy pro počet vrcholů nebo výšku neplatí. Tvrzení, že podstromy vrcholu řádu k mají řád $0, 1, \dots, k-1$ budeme muset upravit.
4. Podobně jako v binomiální haldě kořen jednoho stromu přípojíme jako syna kořene druhého stromu.
5. Nové prvky vkládáme podobně jako v líné binomiální haldě.
6. Nejmenší prvek mažeme podobně jako v líné binomiální haldě, a to včetně následné rekonstrukce, kde spojujeme stromy stejného řádu.

Idea

- Danému vrcholu snížíme hodnotu priority a odpojíme jej od otce
- Pokud otec je označený, tak jej taky odpojíme
- Pokud je děda taky označený, tak jej taky odpojíme
- Takto pokračuje, dokud nenarazíme na neoznačený vrchol nebo kořen

Příklad



Fibonacciho halda: Operace DECREASE

Algoritmus

Input: Vrchol u a nová priorita k

Snížit prioritu vrcholu u
if *u je kořen nebo otec p vrcholu u má prioritu nejvýše k* **then**
    return `# Haldový invariant je zachovaný`
$p \leftarrow$ otec vrcholu u
Odznačit vrchol u
Odpojit vrchol u od otce p a připojit u k seznamu stromů
while *p není kořen a p je označený* **do**
    $u \leftarrow p$
    $p \leftarrow$ otec vrcholu u
    Odznačit vrchol u
    Odpojit vrchol u od otce p a připojit u k seznamu stromů
if *p není kořen* **then**
    Označit vrchol p

Fibonacciho halda: Invariant

Invariant

Pro každý vrchol p a jeho i-tého syna s platí, že s má alespoň

- $i-2$ synů, pokud s je označený, a
- $i-1$ synů, pokud s není označený. ①

Důkaz

Všechny povolené operace zachovávají platnost invariantu

- Init: Prázdná halda invariant splňuje ②
- INSERT: Vytvoření nového stromu s jedním vrcholem ③
- DELETEMIN: Pro nesmazané vrcholy se počty synů ani jejich pořadí nezmění
- Připojení stromu u řádu $k-1$ jako k-tého syna vrcholu p ④
- Odstranění i-tého syna x z vrcholu u řádu k, který je kořenem
  - Pořadí $(i+1)$-tého až k-tého syna vrcholu u se sníží o jedna ⑤
- Odstranění i-tého syna x z neoznačeného vrcholu u řádu k, který j-tým synem p
  - Pořadí $(i+1)$-tého až k-tého syna vrcholu u se sníží o jedna
  - Před odstraněním x platilo $k \geq j-1$ a po odstranění x je vrchol u označený a počet synů u splňuje $k-1 \geq j-2$ ⑥

1. Předpokládáme, že synové jsou očíslovaní podle „věku“, tj. později vložený syn má větší index
2. Halda nemá žádný vrchol, a proto neexistuje vrchol porušující invariant.
3. Nový vrchol nemá žádného syna, a tak nemá syna porušující invariant.
4. Spojujeme stromy u a p řádu $k-1$. Po spojení je k-tý syn u vrcholu p neoznačený a má $k-1$ synů. Pořadí ostatních synů vrcholu p je zachováno.
5. Invariant je zachován, protože se minimální počet požadovaných synů těchto vrcholů sníží o jedna a skutečný počet je zachován.
6. Neoznačený j-tý vrchol u musel mít alespoň $j-1$ synů. Po odstranění vrcholu x se počet synů vrcholu u snížil o jedna, a proto má alespoň $j-2$ synů, což je minimální požadovaný počet synů j-tého označeného syna vrcholu p.

Fibonacciho halda: Struktura

Invariant

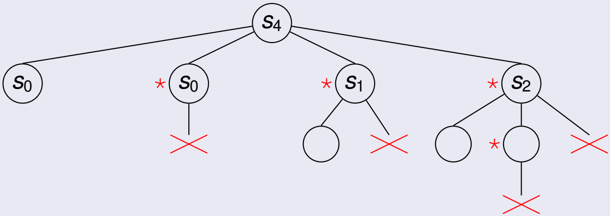
Pro každý vrchol p a jeho i-tého syna s platí, že s má alespoň

- $i - 2$ synů, pokud s je označený, a
- $i - 1$ synů, pokud s není označený.

Velikost podstromu

Nechť s_k je minimální počet vrcholů v podstromu vrcholu s k syny.
Pak platí $s_k \geq s_{k-2} + s_{k-3} + s_{k-4} + \cdots + s_2 + s_1 + s_0 + s_0 + 1$.

Příklad



Fibonacciho halda: Struktura

Velikost podstromu

Nechť s_k je minimální počet vrcholů v podstromu vrcholu s k syny.
Pak platí $s_k \geq s_{k-2} + s_{k-3} + s_{k-4} + \cdots + s_2 + s_1 + s_0 + s_0 + 1$.

Fibonacciho čísla (Cvičení)

- $F_0 = 0$ a $F_1 = 1$ a $F_k = F_{k-1} + F_{k-2}$
- $\sum_{i=1}^{k} F_i = F_{k+2} - 1$
- $F_k = \frac{(1+\sqrt{5})^k - (1-\sqrt{5})^k}{2^k\sqrt{5}}$
- $F_{k+1} \geq \left(\frac{1+\sqrt{5}}{2}\right)^k$
- $s_k \geq F_{k+2}$
    - $s_k \geq 1 + s_0 + \sum_{i=0}^{k-2} s_i \geq 1 + F_1 + \sum_{i=0}^{k-2} F_{i+2} \geq 1 + \sum_{i=1}^{k} F_i = 1 + F_{k+2} - 1$

Důsledek

Strom řádu k má alespoň $s_k \geq F_{k+2} \geq \left(\frac{1+\sqrt{5}}{2}\right)^{k+1}$ vrcholů. Proto,

- kořen stromu s m vrcholy má $\mathcal{O}(\log m)$ synů a
- Fibonacciho halda má $\mathcal{O}(\log n)$ stromů po operaci DELETEMIN. 1

1. Obecně může mít Fibonacciho halda až n stromů, ale po konsolidaci (součást operace DELETEMIN) mají každé dva stromy různý řád a maximální řád stromu je $\mathcal{O}(\log n)$.

Fibonacciho halda: Složitost

Složitost v nejhorším případě

- Operace INSERT: $\mathcal{O}(1)$
- Operace DECREASE: $\mathcal{O}(n)$ (Cvičení)
- Operace DELETEMIN: $\mathcal{O}(n)$

Amortizovaná složitost: Potenciál

Uvažujme potenciál $\Phi = t + 2m$, kde

- t je počet stromů v haldě
- m je celkových počet označených vrcholů

Amortizovaná složitost: Operace INSERT

- Skutečný čas: $\mathcal{O}(1)$
- Změna potenciálu $\Phi' - \Phi = 1$
- Amortizovaná složitost $\mathcal{O}(1)$

Fibonacciho halda: Amortizovaná složitost operace DECREASE

Potenciál

$\Phi = t + 2m$, kde t je počet stromů v haldě a m je celkových počet označených vrcholů

Jedna iterace while-cyklu (odznačení vrcholu a odpojení od otce)

- Skutečný čas: $\mathcal{O}(1)$
- Změna potenciálu $\Phi' - \Phi = 1 - 2 = -1$
- Amortizovaná složitost: 0 ①

Ostatní instrukce

- Skutečný čas: $\mathcal{O}(1)$
- Změna potenciálu $\Phi' - \Phi = 3$
  V nejhorším případě vytvoříme nový strom a jeden vrchol označíme
- Amortizovaná složitost: $\mathcal{O}(1)$

Amortizovaná složitost operace DECREASE

$\mathcal{O}(1)$

1. Formálně nemůžeme napsat $\mathcal{O}(1) - 1 = 0$. Zde je nutné říct, že jednička v hodnotě potenciálu značí čas potřebný na provedení jedné iterace while-cyklu v operaci DECREASE.

Fibonacciho halda: Amortizovaná složitost operace DELETEMIN

Smazání kořene a připojení synů k seznamu stromů

- Skutečný čas: $\mathcal{O}(\log n)$
- Změna potenciálu $\Phi' - \Phi = \mathcal{O}(\log n)$
- Amortizovaná složitost: $\mathcal{O}(\log n)$

Jedna iterace while-cyklu při rekonstrukci (spojení dvou stromů)

- Skutečný čas: $\mathcal{O}(1)$
- Změna potenciálu $\Phi' - \Phi = -1$
- Amortizovaná složitost: 0 ①

Ostatní instrukce

- Skutečný čas: $\mathcal{O}(\log n)$
- Změna potenciálu $\Phi' - \Phi = 0$
- Amortizovaná složitost: $\mathcal{O}(\log n)$

Amortizovaná složitost operace DELETEMIN

$\mathcal{O}(\log n)$

1. Zde vidíme, že musíme ještě trochu upravit význam hodnoty potenciálu: Jednička v hodnotě potenciálu značí maximum z časů potřebných na provedení jedné iterace while-cyklu v operaci DECREASEa jedné iterace while-cyklu v rekonstrukci. Podstatné je, že jednička v hodnotě potenciálu značí nějaký pevný konstatní čas.

Haldy: Shrnutí

Přehled časových složitostí

	Binární	Binomiální		Líná binomiální		Fibonacciho	
	worst	worst	amort	worst	amort	worst	amort
INSERT	$\log n$	$\log n$	1	1	1	1	1
DECREASE	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	n	1
DELETEMIN	$\log n$	$\log n$	$\log n$	n	$\log n$	n	$\log n$

Obsah

Intervalový dotaz (Range query)

Popis problému

- Máme dánu množinu S obsahující n bodů z $\mathbb{R}^d$
- Intervalem rozumíme d-dimenzionální obdélník, např. $\langle a_1, b_1\rangle \times \cdots \times \langle a_d, b_d\rangle$
- Operace QUERY: Najít všechny body v daném intervalu
- Operace COUNT: Určit počet bodů v daném intervalu

Aplikace

- Počítačová grafika, výpočetní geometrie
- Databázové dotazy, např. určit zaměstnance ve věku 20-35 a platem 20-30 tisíc

Intervalový dotaz v $\mathbb{R}^1$

Staticky

Body uložíme do pole

BUILD: $\mathcal{O}(n \log n)$

COUNT: $\mathcal{O}(\log n)$

QUERY: $\mathcal{O}(k + \log n)$

k je počet vyjmenovaných bodů

Dynamicky

Body uložíme do vyhledávacího stromu

BUILD: $\mathcal{O}(n \log n)$

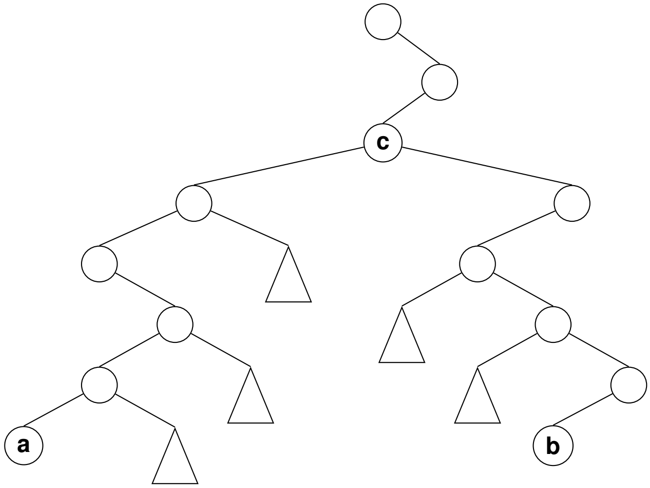
INSERT: $\mathcal{O}(\log n)$

DELETE: $\mathcal{O}(\log n)$

COUNT: $\mathcal{O}(\log n)$

QUERY: $\mathcal{O}(k + \log n)$

Intervalový dotaz v $\mathbb{R}^1$: Příklad



Vrchol *a* je nejmenší prvek v intervalu, *b* je největší prvek v intervalu a *c* je poslední společný vrchol na cestách z kořene do vrcholů *a* a *b*.

Obsah

k-d stromy

Popis

- Body uložíme do binárního stromu
- Do kořene uložíme medián podle první souřadnice
- Do levého (pravého) podstromu uložíme body mající první souřadnice menší (větší) než medián
- Vrcholy v první vrstvě pod kořenem se body rozdělují podle druhé souřadnice
- V dalších vrstvách dělíme (cyklicky) podle dalších souřadnice
- Výška stromu je $\log_2 n + \Theta(1)$
- Operace BUILD v čase $\mathcal{O}(n \log n)$
- Body je též možné ukládat jen do listů a vrcholy pak obsahují jen rozdělující nadroviny

Algoritmus

Procedure QUERY(*vrchol stromu v*, *interval R*)
    if *v je list* **then**
        Vypiš *v*, pokud leží v *R*
    else if *rozdělující nadrovina vrcholu v protíná R* **then**
        QUERY(*levý syn v*, *R*)
        QUERY(*pravý syn v*, *R*)
    else if *R je „vlevo“ od rozdělující nadroviny vrcholu v* **then**
        QUERY(*levý syn v*, *R*)
    else
        QUERY(*pravý syn v*, *R*)

k-d stromy: Operace QUERY COUNT: Analýza

Příklad nejhoršího případu pro $\mathbb{R}^2$

- Máme množinu bodů $S = \{(x, y); x, y \in [m]\}$, kde $n = m^2$
- Chceme najít množinu všech bodů v intervalu $\langle 1,2; 1,8 \rangle \times \mathbb{R}$
- V každé vrstvě rozdělující podle y-ové souřadnice musíme prozkoumat oba podstromy
- Výška stromu je $\log_2 n + \Theta(1)$ a v polovině vrstev prozkoumáváme oba podstromy
- Celkem navštívíme $2^{\frac{1}{2}\log_2 n + \Theta(1)} = \Theta(\sqrt{n})$ listů

Příklad nejhoršího případu pro $\mathbb{R}^d$

- Mějme množinu bodů $S = [m]^d$, kde $n = m^d$
- Chceme najít množinu všech bodů v intervalu $\langle 1,2; 1,8 \rangle \times \mathbb{R}^{d-1}$
- V každé vrstvě nerozdělující podle první souřadnice musíme prozkoumat oba podstromy
- V $\frac{d-1}{d} \log_2 n + \Theta(1)$ vrstvách prozkoumáváme oba podstromy
- Celkem navštívíme $2^{\frac{d-1}{d}\log_2 n + \Theta(1)} = \Theta(n^{1-\frac{1}{d}})$ listů

$[m]^d$ značí tzv. mřížové body, tj. body $\mathbb{R}^d$, jejichž každá souřadnice je celé číslo od 0 od $m-1$.

kd-stromy jsou mají nejlepší možnou časovou složitost, pokud datová struktura smí používat pouze $\mathcal{O}(n)$ paměti. Intervalové stromy umí vyhodnotit intervalový dotaz v čase $\mathcal{O}\left(\log^d n\right)$, ale potřebují $\mathcal{O}\left(n \log^{d-1} n\right)$ paměti.

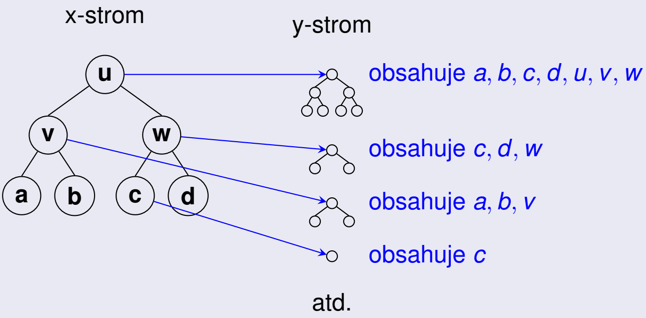
Obsah

Intervalové stromy v $\mathbb{R}^2$ (Range tree)

Konstrukce

- Vybudujeme binární vyhledávací strom podle x-ové souřadnice bodů (x-strom)
- Nechť S_u je množina bodů v podstromu vrcholu u
- Každý vrchol u vybudujeme jeden binární vyhledávací strom podle y-ové souřadnice obsahující S_u
- Bodu můžou být uloženy ve všech vrcholech nebo jen v listech ①

Příklad



1. Podobně jako ve vyhledávacích stromech můžeme uvažovat dvě varianty: prvky mohou být uloženy ve všech vrcholech nebo jen v listech. I když ukládání bodů do všech vrcholů může být paměťově jednodušší, k vysvětlení a analýze bude jednodušší uvažovat, že prvky jsou jen v listech.

Intervalové stromy v $\mathbb{R}^2$: Paměť

Vertikální pohled

Každý bod p je uložen v právě jednom vrcholu v x-stromu a dále je bod p uložen v každém y-stromu přiřazenému vrcholu na cestě z x-kořene do v.

Horizontální pohled

Každá vrstva x-stromu rozkládá body podle x-ové souřadnice. Proto každý bod je uložen v nejvýše jednom y-stromu z každé vrstvy x-stromu.

Předpoklad

Předpokládejme, že binární vyhledávací strom použitý v intervalových stromech je vyvážený, a tedy jeho výška je $\Theta(\log n)$.

Paměťová složitost

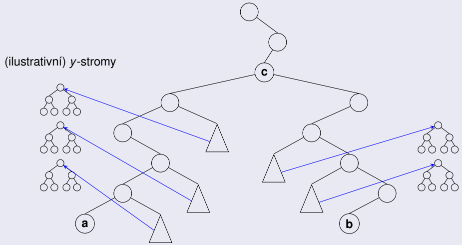
Každý bod je uložen v $\mathcal{O}(\log n)$ y-stromech a celková paměťová složitost je $\mathcal{O}(n \log n)$.
1

1. Bod b je uložen v právě tolika y-stromech, jaká je hloubka vrcholu obsahující b.

Idea algoritmu vyhledávání

1. Najít klíče a_x a b_x v x-stromu ①
2. Určit vrcholy u x-stromu takové, že S_u obsahuje pouze body s x-ovou souřadnicí v intervalu $\langle a_x, b_x \rangle$ ②
3. V těchto vrcholech položme y-ový dotaz $\langle a_y, b_y \rangle$

Příklad



Složitost dotazu Count

$\mathcal{O}(\log^2 n)$ protože y-ový dotaz je volán v $\mathcal{O}(\log n)$ y-stromech ③ ④

1. Přesněji: najít ze všech prvků v x-stromu dva body mající nejmenší a největší x-ovou souřadnic ležící intervalu $\langle a_x, b_x \rangle$.
2. Je zřejmé, že když vrchol tuto podmínku splňuje, tak ji splňují i synové vrcholu. Proto nás zajímají nejvýše umístěné vrcholy s touto vlastností, tj. vrcholy splňující tuto podmínky, ale jejichž otec tuto podmínku nesplňuje.
3. Všimněme si, že prohledávané y-stromy obsahují po dvou disjunktní množiny prvků.
4. V dotazu QUERY je nutné vyjmenovat všechny body, a proto složitost je $\mathcal{O}(k + \log^2 n)$, kde k je počet bodů v obdélníku.

Intervalové stromy v $\mathbb{R}^d$ (předpokládáme $d \geq 2$)

Popis

- i-strom je binární vyhledávací strom podle i-té souřadnice pro $i = 1, \ldots, d$
- Pro $i < d$ má každý vrchol u i-stromu ukazatel na $(i+1)$-strom obsahující S_u
- Intervalovým stromem rozumíme všechny výše popsané stromy

Reprezentace

Struktura vrcholu intervalového stromu obsahuje

key	nadrovina rozdělující prostor mezi syny ①
left, right	ukazatel na levého a pravého syna
tree	ukazatel na kořen $(i+1)$-stromu
size	počet bodů v podstromu (pokud potřebujeme dotaz COUNT)

Poznámka

Nechť u je vrchol i-stromu a T jsou všechny vrcholy dosažitelné opakovaným přístupem k ukazatelům left, right a tree z vrcholu u. Pak T tvoří intervalový strom na vrcholech S_u a souřadnicích $i, \ldots, d$.

1. Operace QUERY musí umět body ležící v obdélníku i vypsat, a proto potřebuje mít ve všech vrcholech uloženy všechny souřadnice bodu. Operaci QUERY stačí klíč, což v i-stromu je i-tá souřadnice bodu.

Intervalové stromy v $\mathbb{R}^d$: Struktura a paměťová složitost

V kolika listech je uložený bod b?

- Existuje $\mathcal{O}(\log n)$ vrcholů u 1-stromu takových, že S_u obsahuje b
- Tedy počet 2-stromů obsahující b je $\mathcal{O}(\log n)$
- Uvažujme i-strom T obsahující bod b
- Pak v T je $\mathcal{O}(\log n)$ vrcholů w takových, že $b \in S_w$ ①
- Počet $(i+1)$-stromů přiřazených nějakému vrcholu T obsahujících b je $\mathcal{O}(\log n)$
- Každou dimenzí se počet stromů obsahujících b zvyšuje o multiplikativní faktor $\mathcal{O}(\log n)$
- Celkový počet stromů/listů obsahujících b je $\mathcal{O}\left(\log^{d-1} n\right)$
- Celková paměťová složitost je $\mathcal{O}\left(n \log^{d-1} n\right)$

Kolik má intervalový strom i-stromů?

- Počet vrcholů ve všech $(i-1)$-stromech je $\mathcal{O}\left(n \log^{i-2} n\right)$
- Každému vrcholu $(i-1)$-stromu je přiřazen jeden i-strom
- Počet i-stromů je $\mathcal{O}\left(n \log^{i-2} n\right)$ ②

1. Strom T nemusí obsahovat všechny prvky, a proto jeho výška nemusí být $\Omega(\log n)$. Dokonce většina stromů obsahuje celkem malý počet bodů.
2. Platí pro $i \geq 2$. Pro $i = 1$ máme jeden 1-strom.

Intervalové stromy v $\mathbb{R}^d$: BUILD

Algoritmus (Body M jsou v poli setříděné podle poslední souřadnice)

Procedure BUILD (*množina bodů M, dimenze stromu d, aktuální souřadnice i*)
  if $|M| = 1$ **then**
    return nový list obsahující jediný vrchol M [1]
  if $i = d$ **then**
    return kořen stromu vytvořený ze setříděného pole
  $v \leftarrow$ nový vrchol
  $v.tree \leftarrow$ BUILD $(M, d, i+1)$
  $v.key \leftarrow$ medián i-tých souřadnic bodů M
  $M_l, M_r \leftarrow$ rozděl M na body mající i-tou souřadnici menší a větší než $v.key$
  $v.left \leftarrow$ BUILD (M_l, d, i)
  $v.right \leftarrow$ BUILD (M_r, d, i)
  return v

Složitost jednoho volání funkce BUILD (bez rekurze)

- Pro $i = d$ je složitost $\mathcal{O}(1)$ [2]
- Pro $i < d$ je složitost $\mathcal{O}(|S|)$ [3]

1. Zde je otázka, zda list musí mít přiřazený (triviální) strom další dimenze. Je to implementační detail, intervalový strom může fungovat v obou verzích a asymptotické složitosti se nemění.
2. Předpokládáme, že množina stromů M předávaná v rekurzi se udržuje setříděná. Čas jednoho volání funkce $\textsc{Build}$ je $\mathcal{O}(1)$ pro $i = d$, protože medián leží uprostřed pole M a rozdělení M na M_l a M_r je jen otázka předání správných ukazatelů.
3. Pro $i < d$ není pole M setříděné podle aktuální souřadnice i, a proto nalezení mediánu a rozdělení pole trvá $\mathcal{O}(n_T)$. Časovou složitost vytvoření i-stromu T lze popsat rekurentní formulí $f(n) = 2f(n/2) + \mathcal{O}(n)$, jejíž řešení je $\mathcal{O}(n_T \log n_T)$.

Intervalové stromy v $\mathbb{R}^d$: Analýza operace Build

Vytvoření d-stromů

- d-stromy vytváříme v lineárním čase (tj. konstantní čas na vrchol)
- Počet vrcholů ve všech d-stromech je $\mathcal{O}(n \log^{d-1} n)$
- Časová složitost vytvoření všech d-stromů je $\mathcal{O}(n \log^{d-1} n)$

Vytvoření i-stromu pro $i = 1, \ldots, d-1$ (nepočítaje $(i+1)$-stromy, ...)

- Počet vrcholů ve všech i-stromech je $\mathcal{O}(n \log^{i-1} n)$
- Nechť n_T je počet vrcholů v i-stromu T
- Vybudování samotného stromu T trvá $\mathcal{O}(n_T \log n_T)$
- Vybudování všech i-stromů trvá

$$\sum_{i\text{-strom } T} n_T \log n_T \leq \log n \sum_{i\text{-strom } T} n_T = \log n \cdot n \log^{i-1} n = n \log^i n$$

Časová složitost operace Build

$\mathcal{O}(n \log^{d-1} n)$

Intervalové stromy v $\mathbb{R}^d$: QUERY $\langle a_1, b_1\rangle \times \cdots \times \langle a_d, b_d\rangle$

Procedure `Query` (*vrchol v, aktuální souřadnice i*)
    if $v = NIL$ **then**
        return
    if $v.key \leq a_i$ **then**
        `Query` ($v.right, i$)
    else if $v.key \geq b_i$ **then**
        `Query` ($v.left, i$)
    else
        if *v.point leží v obdélníku* **then**
            Vypiš *v.point*
        `Query_left` ($v.left, i$)
        `Query_right` ($v.right, i$)

```
Procedure Query_left(vrchol v, aktuální souřadnice i)
    if v = NIL then
        return
    if v.key < a_i then
        Query_left(v.right, i)
    else
        if v.point leží v obdélníku then
            Vypiš v.point
        Query_left(v.left, i)
        if i < d then
            Query(v.right.tree, i + 1)
        else
            Vypiš všechny body v podstromu vrcholu v.right
```

Intervalové stromy v $\mathbb{R}^d$: QUERY

Složitost operace COUNT

- V každém stromě přistoupíme k nejvýše dvěma vrcholům z každé vrstvy
- Z každého navštíveného i-stromu pokračujeme do $\mathcal{O}(\log n)$ $(i+1)$-stromů
- Počet navštívených i-stromů je $\mathcal{O}\left(\log^{i-1} n\right)$
- Celková složitost je $\mathcal{O}\left(\log^d n\right)$

Složitost operace QUERY

- Vypsání všech bodů v podstromu trvá $\mathcal{O}(k)$, kde k je počet nalezených bodů
- Celková složitost je $\mathcal{O}\left(k + \log^d n\right)$

Intervalové stromy v $\mathbb{R}^d$: Dynamizace pomocí BB[α]-stromů

BB[α]-strom

- Binární vyhledávací strom
- Počet listů v podstromu vrcholu u označme s_u
- Podstromy obou synů každého vrcholu u mají nejvýše αs_u listů

Operace Insert (Delete je analogický)

- Najít list pro nový prvek a uložit do něho nový prvek (složitost: $\mathcal{O}(\log n)$)
- Jestliže některý vrchol u porušuje vyvažovací podmínku, tak celý jeho podstrom znovu vytvoříme operací BUILD (složitost $\mathcal{O}(s_u)$)

Amortizovaná časová složitost operací Insert a Delete: Agregovaná metoda

- Jestliže podstrom vrcholu u po provedení operace BUILD má s_u listů, pak další porušení vyvažovací podmínky pro vrchol u nastane nejdříve po $\Omega(s_u)$ přidání/smazání prvků v podstromu vrcholu u
- Amortizovaný čas vyvažovaní jednoho vrcholu je $\mathcal{O}(1)$
- Při jedné operaci Insert/Delete se prvek přidá/smaže v $\mathcal{O}(\log n)$ podstromech
- Amortizovaný čas vyvažovaní při jedné operaci Insert nebo Delete je $\mathcal{O}(\log n)$

Intervalové stromy v $\mathbb{R}^d$: Dynamizace pomocí BB[α]-stromů

Použití BB[α]-stromů v intervalových stromech

- Binární vyhledávací stromy implementujeme pomocí BB[α]-stromů
- Vyžaduje-li BB[α]-strom vyvážení, pak přebudujeme všechny přiřazené stromy

Složitost operace Insert a Delete

- Navštívených i-stromů je $\mathcal{O}(\log^{i-1} n)$ a v každém navštívíme $\mathcal{O}(\log n)$ vrcholů
- Složitost bez přebudování je $\mathcal{O}(\log^d n)$; analyzujme přebudování
- Uvažujme libovolný vrchol u, který leží v i-stromu
- Přebudování vrcholu u trvá $\mathcal{O}(s_u \log^{d-i} s_u)$
- Přebudování vrcholu u může nastat po $\Omega(s_u)$ po přidání/smazání prvků v podstromu u
- Amortizovaná cena přidání/smazání do vrcholu u je $\mathcal{O}(\log^{d-i} s_u) \leq \mathcal{O}(\log^{d-i} n)$
- Amortizovaný čas operace Insert a Delete je $\sum_{i=1}^{d} \mathcal{O}(\log^{i-1} n)\mathcal{O}(\log n)\mathcal{O}(\log^{d-i} n) = \mathcal{O}(\log^d n)$

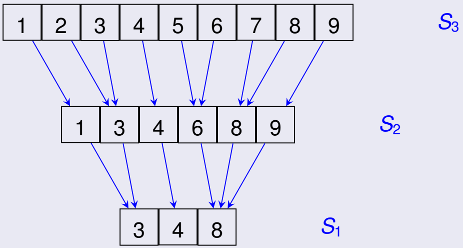
Kaskádování (Fractional cascading)

Motivační problém

Dány množiny $S_1 \subseteq \cdots \subseteq S_k$, kde $|S_k| = n$, vymyslete datovou strukturu pro rychlé vyhledání prvku $x \in S_1$ ve všech množinách $S_1, \ldots, S_k$. 1

Kaskádování

Všechny množiny jsou setříděné a navíc každý prvek v poli S_i má ukazatel na stejný prvek v poli S_{i-1}. 2



Složitost hledání ve všech m množinách

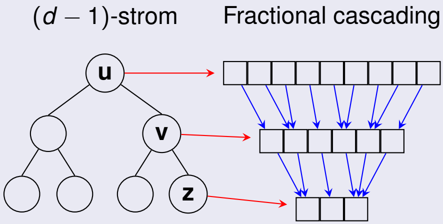
$\mathcal{O}(k + \log n)$

1. Triviálním řešením získáme složitost $\mathcal{O}(k \log n)$, kterou bychom chtěli zlepšit.
2. Prvky $S_i \setminus S_{i-1}$ ukazují na své předchůdce nebo následovníky.

Intervalové stromy v $\mathbb{R}^d$ s kaskádováním v poslední dimenzi

Použití

- Každému $(d-1)$-stromu je přiřazena je kaskáda místo d-stromů
- Každý prvek v kaskádě musí mít dva ukazatele do pole nižší úrovně (pro levého a pravého syna vrcholu v $(d-1)$-stromu)



Složitost operace QUERY

- Dotaz v jednom $(d-1)$-stromu trvá $\mathcal{O}(\log n)$ včetně vyhodnocení kaskády
- Dotazů v $(d-1)$-stromech je $\mathcal{O}\left(\log^{d-2} n\right)$
- Složitost operace QUERY je $\mathcal{O}\left(k + \log^{d-1} n\right)$

Intervalové stromy v $\mathbb{R}^d$ s kaskádováním v poslední dimenzi

Paměťová složitost

- Místo d-stromu T s s_T vrcholy máme pole velikosti s_T
- Paměťová složitost je $\mathcal{O}\left(n \log^{d-1} n\right)$

Složitost operace Build

- Vybudování $(d-1)$-stromu s s_u vrcholy včetně kaskádování trvá $\mathcal{O}(s_u \log s_u)$
- Složitost vybudování i-stromů pro $i < d$ se nemění
- Složitost operace Build je $\mathcal{O}\left(n \log^{d-1} n\right)$

Operace Insert a Delete (Cvičení)

- Je možné efektivně přidávat a mazat body?
- Je možné reprezentovat kaskády tak, aby bylo možné efektivně hledat, přidávat i mazat body?

Další známá vylepšení intervalových stromů

Popsaný postup

QUERY: $\mathcal{O}(k + \log^{d-1} n)$

Paměť: $\mathcal{O}(n \log^{d-1} n)$

Chazelle [4, 5]

QUERY: $\mathcal{O}(k + \log^{d-1} n)$

Paměť: $\mathcal{O}\left(n \left(\frac{\log n}{\log \log n}\right)^{d-1}\right)$

Chazelle, Guibas [6] pro $d \geq 3$

QUERY: $\mathcal{O}(k + \log^{d-2} n)$

Paměť: $\mathcal{O}(n \log^{d} n)$

Fractional cascading: obecnější verze

Původní problém

Dány množiny $S_1 \subseteq \dots \subseteq S_k$, kde $|S_k| = n$, vymyslete datovou strukturu pro rychlé vyhledání prvku $x \in S_1$ ve všech množinách $S_1, \dots, S_k$.

Změny vedoucí ke zobecnění

- Množiny $S_1, \dots, S_k$ nemusí být v inkluzi.
- $n = \sum_{i=1}^k |S_i|$ je celkový počet prvků

Fractional cascading: obecnější verze

Dány množiny $S_1, \dots, S_k$, vymyslete datovou strukturu pro rychlé vyhledání prvku x ve všech množinách $S_1, \dots, S_k$. 1

1. Triviálním řešením je mít matici $A_{i,x} = 1$, jestliže $x \in S_i$, a jinak $A_{i,x} = 0$. Vyhodnocení dotazu trvá $\mathcal{O}(k + \log n)$, ale paměťová složitost je $\mathcal{O}(kn)$.

Fractional cascading: obecnější verze

Pomocná pole $P_1, \ldots, P_k$

- V polích $P_1, \ldots, P_k$ jsou prvky setříděné
- P_k obsahuje prvky množiny S_k
- P_i obsahuje S_i a každý druhý prvek z P_{i+1} ①
- U každého prvku x v poli P_i máme navíc ukazatel prvek nejbližší k x v poli P_{i+1} ②

Paměťová složitost je $\mathcal{O}(n)$

- Celková spotřebovaná paměť je $\sum_{i=1}^{k} |P_i|$
- $|P_i| \leq |S_i| + \frac{1}{2}|P_{i+1}|$
- $|P_i| \leq |S_i| + \frac{1}{2}|S_{i+1}| + \frac{1}{4}|S_{i+2}| + \frac{1}{8}|S_{i+3}| + \cdots$
  - P_i obsahuje S_i
  - P_{i-1} obsahuje nejvýše polovinu S_i
  - P_{i-2} obsahuje nejvýše čtvrtinu S_i
- Příspěvek S_i do $\sum_{i=1}^{k} |P_i|$ je nejvýše $2|S_i|$
- $\sum_{i=1}^{k} |P_i| \leq 2\sum_{i=1}^{k} |S_i| \leq 2n$

1. Duplicitní prvky můžeme vynechávat. Jelikož je P_{i+1} setříděné, tak do P_i dáváme každý druhý prvek podle setříděného pořadí.
2. Podobně jako v předchozí verzi Fractional cascading máme ukazatel na nejmenší větší nebo největší menší prvek v následujícím poli.

Fractional cascading: obecnější verze

Pomocná pole $P_1, \dots, P_k$

- V polích $P_1, \dots, P_k$ jsou prvky setříděné
- P_k obsahuje prvky množiny S_k
- P_i obsahuje S_i a každý druhý prvek z P_{i+1}
- U každého prvku x v poli P_i máme navíc ukazatel prvek nejbližší k x v poli P_{i+1}

Složitost hledání prvku x je $\mathcal{O}(k + \log n)$

- Nalezení x nebo nejbližšího prvku v P_1 trvá $\mathcal{O}(\log n)$
- Hledání v P_{i+1} pomocí pozice v P_i trvá $\mathcal{O}(1)$

Obsah

Hešovaní

Základní pojmy

- Máme univerzum $U = \{0, 1, \dots, u-1\}$ všech prvků
- Chceme uložit podmnožinu $S \subseteq U$ velikosti n
- Uložíme S do pole velikosti m pomocí hešovací funkce $h : U \to M$, kde $M = \{0, 1, \dots, m-1\}$
- Dva prvky $x, y \in S$ kolidují, jestliže $h(x) = h(y)$
- Hešovací funkce h je perfektní na S, jestliže h nemá žádnou kolizi S

Nepřátelská podmnožina

Pokud $u \geq mn$, pak pro každou hešovací funkci h existuje $S \subseteq U$ velikosti n taková, že h hešuje všechny prvky z S do jedné přihrádky.

Poznámky

- Není možné sestrojit jednu funkci dobře hešující libovolnou podmnožinu $S \subseteq U$
- Pro danou podmnožinu $S \subseteq U$ lze sestrojit perfektní hešovací funkci
- Sestrojíme množinu hešovacích funkcí $\mathcal{H}$ takovou, že náhodně zvolená funkce $h \in \mathcal{H}$ hešuje libovolnou podmnožinu S v průměrném případě uspokojivým způsobem

Obsah

Universální hešování

Cíl

Sestrojit systém $\mathcal{H}$ hešovacích funkcí $f : U \to M$ takový, že náhodně zvolená funkce $f \in \mathcal{H}$ hešuje libovolnou množinu S „většinou dobře“.

Úplně náhodná hešovací funkce

- Systém $\mathcal{H}$ obsahuje všechny funkce $f : U \to M$
- Platí $P[h(x) = z] = \frac{1}{m}$ pro všechna $x \in U$ a $z \in M$
- Náhodné přihrádky $h(x)$ a $h(y)$ jsou nezávislé pro různé $x, y \in U$
- Nepraktické: k zakódování funkce z $\mathcal{H}$ potřebujeme $\Theta(u \log m)$ bitů
- Někdy se používá k analýze hešování

Hešování náhodných dat

- Předpokládejme, že S je náhodná podmnožina U velikosti n
- Každá rozumná funkce hešuje S dobře (včetně $h(x) = x \mod m$)
- Užitečný model k analýze hešování používající úplně náhodnou hešovací funkci
- V praxi nikdy nedostaneme úplně náhodná data

c-universální systém (ekvivalentní definice)

Systém hešovacích funkcí $\mathcal{H}$ je c-universální, jestliže ①

- počet hešovacích funkcí $h \in \mathcal{H}$ splňujících $h(x) = h(y)$ je nejvýše $\frac{c|\mathcal{H}|}{m}$ pro všechna různá $x, y \in U$
- náhodně zvolená $h \in \mathcal{H}$ splňuje $P[h(x) = h(y)] \leq \frac{c}{m}$ pro každé $x, y \in U$ a $x \neq y$. ②

Příklad c-universálního hešovacího systému (cvičení)

- Parametry: p a m, kde $p > u$ je prvočíslo
- Hešovací funkce

$$h_a(x) = (ax \mod p) \mod m$$

  je závislá na hodnotě a
- Hešovací systém $\mathcal{H} = \{h_a;\ 0 < a < p\}$ je c-universální
- Hešovací funkce ze systému $\mathcal{H}$ je určena hodnotou a
- Tedy náhodný výběr hešovací funkce z $\mathcal{H}$ je náhodné vygenerování $a \in \{1, \ldots, p-1\}$

1. Navíc obvykle vyžadujeme, aby hešovací funkci šlo spočítat v čase $\mathcal{O}(1)$ a aby funkci bylo možné popsat $\mathcal{O}(1)$ parametry.
2. Úplně náhodný hešovací systém je 1-universální, protože $h(x)$ padne do nějaké přihrádky a $h(y)$ má uniformní distribuci nezávislou na $h(x)$, a proto $P[h(x) = h(y)] = \frac{1}{m}$.

(k,c)-nezávislý hešovací systém

(2,c)-nezávislý systém hešovacích funkcí (ekvivalentní definice)

Systém hešovacích funkcí $\mathcal{H}$ je $(2, c)$-nezávislý, pokud

- počet $h \in \mathcal{H}$ splňujících $h(x_1) = z_1$ a $h(x_2) = z_2$ je nejvýše $\frac{c|\mathcal{H}|}{m^2}$ pro každé $x_1, x_2 \in U$ a $x_1 \neq x_2$ a $z_1, z_2 \in M$.
- náhodně zvolená $h \in \mathcal{H}$ splňuje $P[h(x_1) = z_1 \text{ a } h(x_2) = z_2] \leq \frac{c}{m^2}$ pro každé $x_1, x_2 \in U$ a $x_1 \neq x_2$ a $z_1, z_2 \in M$.

(k, c)-nezávislý systém hešovacích funkcí

Systém hešovacích funkcí $\mathcal{H}$ je (k, c)-nezávislý, pokud náhodně zvolená $h \in \mathcal{H}$ splňuje $P[h(x_i) = z_i \text{ pro všechna } i = 1, \ldots, k] \leq \frac{c}{m^k}$ pro všechna po dvou různá $x_1, \ldots, x_k \in U$ a všechna $z_1, \ldots, z_k \in M$.

k-nezávislý systém hešovacích funkcí

- Systém $\mathcal{H}$ je k-nezávislý, pokud je (k, c)-nezávislý pro nějaké $c > 1$.
- Systém $\mathcal{H}$ je silně k-nezávislý, pokud je $(k, 1)$-nezávislý.

Universální a nezávislý hešovací systém

Vlastnosti

- (k, c)-nezávislý systém hešovacích funkcí je $(k-1, c)$-nezávislý ①
- $(2, c)$-nezávislý systém hešovacích funkcí je c-universální ②
- Existuje 1-universální systém, který není 2-nezávislý ③
- Pro všechna $x_1, \dots, x_n \in U$ existují $z_1, \dots, z_k \in M$ taková, že $P[h(x_i) = z_i \text{ pro všechna } i = 1, \dots, k] \geq \frac{1}{m^k}$ ④
- Jestliže $\mathcal{H}$ je silně k-nezávislý, pak $P[h(x_i) = z_i \text{ pro všechna } i = 1, \dots, k] = \frac{1}{m^k}$ pro všechna $z_1, \dots, z_k \in M$

1-nezávislý systém není užitečný

Systém $\mathcal{H} = \{h_a(x) = a;\ a \in M\}$ je 1-nezávislý, ale nepoužitelný.

1. $P[h(x_i) = z_i$ pro všechna $i = 1, \dots, k-1] = P[\exists z_k \in M : \ h(x_i) = z_i$ pro všechna $i = 1, \dots, k] \leq \sum_{z_m \in M} P[h(x_i) = z_i$ pro všechna $i = 1, \dots, k] \leq m\frac{1}{m^k} = \frac{1}{m^{k-1}}$
2. $P[h(x) = h(y)] = P[\exists z \in M : \ h(x) = z$ a $h(y) = z] \leq \sum_{z \in M} P[h(x) = z$ a $h(y) = z] \leq m\frac{c}{m^2} = \frac{c}{m}$
3. Uvažujme systém $\mathcal{H}$ všech funkcí $h : U \to M$ takových, že $h(0) = 0$ a $h(1) = 1$, t.j. dva prvky mají pevné přihrádky a ostatní prvky náhodné přihrádky. Pak $P[h(x) = h(y)] = frac1m$, jestliže $\{x, y\} \neq \{0, 1\}$, ale $P[h(0) = 0$ a $h(1) = 1] = 1$.
4. Kdyby $P[h(x_i) = z_i$ pro všechna $i = 1, \dots, k] < \frac{1}{m^k}$ pro všechna $z_1, \dots, z_k \in M$, pak $1 = P[\exists z_1, \dots, z_k \in M :, h(x_i) = z_i$ pro všechna $i = 1, \dots, k] \leq \sum_{z_1, \dots, z_k \in M} P[h(x_i) = z_i$ pro všechna $i = 1, \dots, k] < m^k \frac{1}{m^k} = 1$.

Pozorování

Jestliže systém hešovacích funkcí $\mathcal{H}$ z U do M je (k,c)-nezávislý, pak $|\mathcal{H}| \geq \frac{|M|^k}{c}$.

Důkaz

- Pro spor předpokládejme, že $|\mathcal{H}| < \frac{|M|^k}{c}$
- Tedy $\frac{c|\mathcal{H}|}{|M|^k} < 1$
- Definice říká, že počet funkcí $h \in \mathcal{H}$ takových, že $h(x_i) = z_i$ pro všechna $i = 1, \ldots, k$ je nejvýše $\frac{c|\mathcal{H}|}{|M|^k}$
- Ale pro libovolná $x_1, \ldots, x_k \in U$ existuje $h \in \mathcal{H}$ a $z_1, \ldots, z_k \in M$ taková, že $h(x_i) = z_i$ pro všechna $i = 1, \ldots, k$

Universální hešování: Multiply-mod-prime

Systém Multiply-mod-prime

- Nechť p je prvočíslo větší než u a $[p]$ značí $\{0, \dots, p-1\}$
- $h_{a,b}(x) = (ax + b \bmod p) \bmod m$
- $\mathcal{H} = \{h_{a,b};\ a, b \in [p], a \neq 0\}$
- Systém $\mathcal{H}$ je 1-universální a 2-nezávislý, ale není 3-nezávislý

Lemma

Pro libovolná různá $x_1, x_2 \in [p]$ rovnice

$$\begin{aligned} y_1 &= ax_1 + b \mod p \\ y_2 &= ax_2 + b \mod p \end{aligned}$$

definují bijekci mezi $(a, b) \in [p]^2$ a $(y_1, y_2) \in [p]^2$
a dále bijekci mezi $\left\{(a, b) \in [p]^2;\ a \neq 0\right\}$ a $\left\{(y_1, y_2) \in [p]^2;\ y_1 \neq y_2\right\}$.

Důkaz

- Pro danou dvojici (y_1, y_2) existuje jediná dvojice (a, b) splňující rovnice
    - Odečtením dostáváme $a(x_1 - x_2) \equiv y_1 - y_2 \mod p$
    - V tělese $GF(p) = \mathbb{Z}_p$ dostáváme $a = (y_1 - y_2)(x_1 - x_2)^{-1}$, $b = y_1 - ax_1$
- Zřejmě platí $a = 0$ právě tehdy, když $y_1 = y_2$

Universální hešování: Multiply-mod-prime

Systém Multiply-mod-prime

- $h_{a,b}(x) = (ax + b \bmod p) \bmod m$, kde p je prvočíslo větší než u
- $\mathcal{H} = \{h_{a,b};\ a, b \in [p], a \neq 0\}$

Lemma

Pro libovolná různá $x_1, x_2 \in [p]$ rovnice

$$y_1 = ax_1 + b \mod p$$
$$y_2 = ax_2 + b \mod p$$

definují bijekci mezi $\{(a, b) \in [p]^2;\ a \neq 0\}$ a $\{(y_1, y_2) \in [p]^2;\ y_1 \neq y_2\}$.

Systém $\mathcal{H}$ je 1-universální

- Pro $x_1 \neq x_2$ platí $h_{a,b}(x_1) = h_{a,b}(x_2)$ právě tehdy, když $y_1 \equiv y_2 \pmod m$ a $y_1 \neq y_2$
- Pro y_1 existuje nejvýše $\left\lceil \frac{p}{m} \right\rceil - 1$ hodnot y_2 takových, že $y_1 \equiv y_2 \pmod m$ a $y_1 \neq y_2$
- Počet takových dvojic (y_1, y_2) je nejvýše $p(\left\lceil \frac{p}{m} \right\rceil - 1) \leq p(\frac{p+m-1}{m} - 1) \leq \frac{p(p-1)}{m}$
- Počet funkcí $h_{a,b} \in \mathcal{H}$ způsobujících kolizi $h_{a,b}(x_1) = h_{a,b}(x_2)$ je nejvýše $\frac{p(p-1)}{m}$
- Tudíž $P[h_{a,b}(x_1) = h_{a,b}(x_2)] \leq \frac{p(p-1)}{m|\mathcal{H}|} \leq \frac{1}{m}$.

Universální hešování: Multiply-mod-prime

Systém Multiply-mod-prime

- $h_{a,b}(x) = (ax + b \bmod p) \bmod m$, kde p je prvočíslo větší než u
- $\mathcal{H} = \{h_{a,b};\ a, b \in [p], a \neq 0\}$

Lemma

Pro libovolná různá $x_1, x_2 \in [p]$ rovnice

$$y_1 = ax_1 + b \mod p$$
$$y_2 = ax_2 + b \mod p$$

definují bijekci mezi $(a, b) \in [p]^2$ a $(y_1, y_2) \in [p]^2$.

Systém $\mathcal{H}$ je 2-nezávislý

- Počet y_1 takových, že $z_1 = y_1 \mod m$ je nejvýše $\lceil \frac{p}{m} \rceil$
- Počet (y_1, y_2) takových, že $z_1 = y_1 \mod m$ a $z_2 = y_2 \mod m$ je nejvýše $\lceil \frac{p}{m} \rceil^2$
- Počet funkcí $h_{a,b}$ takových, že $h_{a,b}(x_1) = z_1$ a $h_{a,b}(x_2) = z_2$ je nejvýše $\lceil \frac{p}{m} \rceil^2$
- $P[h_{a,b}(x_1) = z_1 \text{ a } h_{a,b}(x_2) = z_2] \leq \lceil \frac{p}{m} \rceil^2 \frac{1}{p(p-1)} \leq \left(\frac{p+m}{m}\right)^2 \frac{2}{p^2} \leq \left(\frac{2p}{m}\right)^2 \frac{2}{p^2} = \mathcal{O}\left(\frac{1}{m^2}\right)$

Universální hešování: Multiply-mod-prime

Systém Multiply-mod-prime

- $h_{a,b}(x) = (ax + b \bmod p) \bmod m$, kde p je prvočíslo větší než u
- $\mathcal{H} = \{h_{a,b};\ a, b \in [p], a \neq 0\}$

Systém $\mathcal{H}$ není 3-nezávislý

- (k, c)-nezávislost systému $\mathcal{H}$ znamená, že $\mathcal{H}$ je (k, c)-nezávislý pro libovolná $p \geq u \geq m$, kde p je prvočíslo
- A to včetně případů $p = u = m$
- Kdyby existovalo c takové, že by systém $\mathcal{H}$ byl $(3, c)$-nezávislý, pak $|\mathcal{H}| \geq \frac{m^3}{c}$
- Ale pro $p = u = m$ má $\mathcal{H}$ pouze $m(m-1)$ funkcí

Universální hešování: Poly-mod-prime

Systém Poly-mod-prime

- Nechť p je prvočíslo větší než u a $k \geq 1$ celé číslo
- $h_{a_0,\dots,a_{k-1}}(x) = (\sum_{i=0}^{k-1} a_i x^i \bmod p) \bmod m$
- $\mathcal{H} = \left\{h_{a_0,\dots,a_{k-1}};\ a_0,\dots,a_{k-1} \in [p]\right\}$

Cvičení: k-nezávislost

Systém Poly-mod-prime je k-nezávislý.

Universální hešování: Multiply-shift

Multiply-shift

- Předpokládáme, že $u = 2^w$ a $m = 2^l$
- $h_a(x) = (ax \mod 2^w) >> (w - l)$
- $\mathcal{H} = \{h_a;\ a \text{ je liché } w\text{-bitové číslo } \}$

Implementace v C

```
uint64_t hash(uint64_t x, uint64_t l, uint64_t a)
{ return (a*x) >> (64-l); }
```

Vlastnosti systému multiply-shift

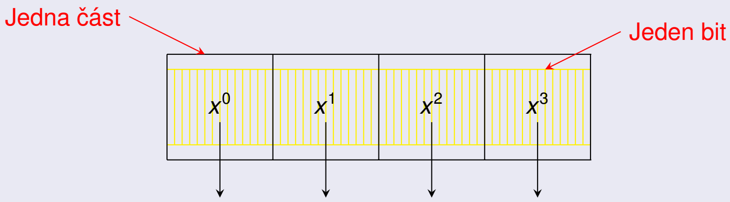
- Silně 2-nezávislý
- Velmi rychlý na reálných počítačích
- V praxi často používaný
- Celý výpočet musí být proveden v neznaménkových celočíselných typech, protože ze součinu *ax* potřebujeme získat posledních *w* bitů

Universální hešování: Tabulkové hešování

Tabulkové hešování

- Předpokládáme, že $u = 2^w$ a $m = 2^l$ a w je násobek d
- Bitový zápis čísla $x \in U$ rozdělíme na d částí $x^0, \ldots, x^{d-1}$ po $\frac{w}{d}$ bitech
- Pro každé $i \in [d]$ vybereme náhodnou hešovací funkci $T_i : [2^{w/d}] \to M$
- Hešovací funkce je $h(x) = T_0(x^0) \oplus \cdots \oplus T_{d-1}(x^{d-1})$

Ilustrativní příklad



$$h(x) = T_0(x^0) \oplus T_1(x^1) \oplus T_2(x^2) \oplus T_3(x^3)$$

Univerzalita

Tabulkové hešování je silně 3-nezávislé, ale není 4-nezávislé.

Universální hešování: Tabulkové hešování

Tabulkové hešování

- Předpokládáme, že $u = 2^w$ a $m = 2^l$ a w je násobek d
- Bitový zápis čísla $x \in U$ rozdělíme na d částí $x^0, \dots, x^{d-1}$ po $\frac{w}{d}$ bitech
- Pro každé $i \in [d]$ vybereme náhodnou hešovací funkci $T_i : [2^{w/d}] \to M$
- Hešovací funkce je $h(x) = T_0(x^0) \oplus \cdots \oplus T_{d-1}(x^{d-1})$

Univerzalita

Tabulkové hešování je 3-nezávislé, ale není 4-nezávislé.

Důkaz 2-nezávislosti (3-nezávislost je ponechána na cvičení)

- Mějme dva prvky x_1 a x_2 lišící se v i-tých částech
- Nechť $h_i(x) = T_0(x^0) \oplus \cdots \oplus T_{i-1}(x^{i-1}) \oplus T_{i+1}(x^{i+1}) \oplus \cdots \oplus T_{d-1}(x^{d-1})$
- $P[h(x_1) = z_1] = P[h_i(x_1) \oplus T_i(x_1^i) = z_1] = P[T_i(x_1^i) = z_1 \oplus h_i(x_1)] = \frac{1}{m}$ 1
- Náhodné jevy $h(x_1) = z_1$ a $h(x_2) = z_2$ jsou nezávislé
  - Náhodné proměnné $T_i(x_1^i)$ a $T_i(x_2^i)$ jsou nezávislé
  - Náhodné jevy $T_i(x_1^i) = z_1 \oplus h_i(x_1)$ a $T_i(x_2^i) = z_2 \oplus h_i(x_2)$ jsou nezávislé
- $P[h(x_1) = z_1 \text{ a } h(x_2) = z_2] = P[h(x_1) = z_1]P[h(x_2) = z_2] = \frac{1}{m^2}$

1. $T_i(x_1^i)$ nabývá všech hodnot z M se stejnou pravděpodobností $\frac{1}{m}$ a náhodné proměnné $T_i(x_1^i)$ a $z_1 \oplus h_i(x_1)$ jsou nezávislé.

Tabulkové hešování není 4-nezávislé

1. Zvolíme prvky x_1, x_2, x_3 a x_4 takové, že
   - části x_1 splňují $x_1^0 = 0$, $x_1^1 = 0$, $x_1^i = 0$ pro $i \geq 2$
   - části x_2 splňují $x_2^0 = 1$, $x_2^1 = 0$, $x_2^i = 0$ pro $i \geq 2$
   - části x_3 splňují $x_3^0 = 0$, $x_3^1 = 1$, $x_3^i = 0$ pro $i \geq 2$
   - části x_4 splňují $x_4^0 = 1$, $x_4^1 = 1$, $x_4^i = 0$ pro $i \geq 2$
2. Platí $h(x_1) \oplus h(x_2) \oplus h(x_3) = h(x_4)$:
3. Zvolme libovolná z_1, z_2, z_3 a nechť $z_4 = z_1 \oplus z_2 \oplus z_3$
4. Jestliže $h(x_1) = z_1$, $h(x_2) = z_2$ a $h(x_3) = z_3$, pak $h(x_4) = z$
5. Podmíněná pravděpodobnost
   $P[h(x_4) = z_4 | h(x_1) = z_1 \text{ a } h(x_2) = z_2 \text{ a } h(x_3) = z_3] = 1$
6. $P[h(x_1) = z_1 \text{ a } h(x_2) = z_2 \text{ a } h(x_3) = z_3 \text{ a } h(x_4) = z_4]$
   $= P[h(x_4) = z_4 | h(x_1) = z_1 \text{ a } h(x_2) = z_2 \text{ a } h(x_3) = z_3]$
   $\cdot P[h(x_1) = z_1 \text{ a } h(x_2) = z_2 \text{ a } h(x_3) = z_3]$
   $\geq \frac{1}{m^3}$ pro nějaká $z_1, z_2, z_3 \in M$
7. Tedy pro libovolné $c \geq 1$ existují $m \in \mathbb{N}, x_1, x_2, x_3, x_4, z_1, z_2, z_3, z_4 \in [u]$ taková, že
   $P[h(x_1) = z_1 \text{ a } h(x_2) = z_2 \text{ a } h(x_3) = z_3 \text{ a } h(x_4) = z_4] \geq \frac{1}{m^3} > \frac{c}{m^4}$

Universální hešování: Vektory

Multiply-shift pro vectory pevné délky k

- Chceme hešovat vektor $x_1, \dots, x_d \in U = [2^w]$ do $S = [2^l]$, zvolme $v \geq w + l - 1$
- $h_{a_1,\dots,a_d,b}(x_1, \dots, x_d) = ((b + \sum_{i=1}^d a_i x_i) \bmod 2^w) >> (w - l)$
- $\mathcal{H} = \{h_{a_1,\dots,a_d,b};\ a_1, \dots, a_d, b \in [2^v]\}$
- Systém $\mathcal{H}$ je 2-nezávislý (bez důkazu)

Poly-mod-prime pro různě dlouhé řetězce I

- Chceme hešovat řetězec $x_0, \dots, x_d \in U$ do $[p]$, kde $p \geq u$ je prvočíslo
- $h_a(x_0, \dots, x_d) = \sum_{i=0}^d x_i a^i \bmod p$ 1
- $\mathcal{H} = \{h_a;\ a \in [p]\}$
- $P[h_a(x_0, \dots, x_d) = h_a(x'_0, \dots, x'_{d'})] \leq \frac{d+1}{p}$ pro různé řetězce délek $d' \leq d$. 2

Poly-mod-prime pro různě dlouhé řetězce II

- Chceme hešovat řetězec $x_0, \dots, x_d \in U$ do M, kde $p \geq m$ je prvočíslo
- $h_{a,b,c}(x_0, \dots, x_d) = \left(b + c \sum_{i=0}^d x_i a^i \bmod p\right) \bmod m$
- $\mathcal{H} = \{h_{a,b,c};\ a, b, c \in [p]\}$
- $P[h_{a,b,c}(x_0, \dots, x_d) = h_{a,b,c}(x'_0, \dots, x'_{d'})] \leq \frac{2}{p}$ pro různé řetězce délek $d' \leq d \leq \frac{p}{m}$.

1. $x_0, \dots, x_d$ jsou koeficienty polynomu stupně d a polynom je v proměnné a.
2. Dva různé polynomy stupně nejvýše d mají nejvýše $d + 1$ společných bodů, takže existuje nejvýše $d + 1$ kolidujících hodnot α.

Obsah

Hešování se separovanými řetězci

Popis

V přihrádce j jsou uloženy všechny prvky $i \in S$ splňující $h(i) = j$ ve spojovém seznamu, dynamickém poli nebo vyhledávacím stromě.

Implementace

- std::unordered_map v C++
- Dictionary v C#
- HashMap v Java
- Dictionary v Python

Hešování se separovanými řetězci: Počet prvků v přihrádce

Značení

- $\alpha = \frac{n}{m}$ je faktor zaplnění; předpokládáme $\alpha = \Theta(1)$
- I_{ij} je náhodná proměnná indikující, zda i-tý prvek patří do j-tého koše
- $A_j = \sum_{i \in S} I_{ij}$ je počet prvků v j-té přihrádce

Pozorování

Pokud je hešovací systém silně 1-nezávislý, pak očekávaný počet prvků v přihrádce je $E[A_j] = \alpha$. ①

- $E[A_j] = E[\sum_{i \in S} I_{ij}] = \sum_{i \in S} E[I_{ij}] = \sum_{i \in S} P[h(i) = j] = \sum_{i \in S} \frac{1}{m} = \frac{n}{m}$ ②

Lemma

Pokud je hešovací systém silně 2-nezávislý, pak

- $E[A_j^2] = \alpha(1 + \alpha - 1/m)$
  - $E[A_j^2] = E[(\sum_{i \in S} I_{ij})(\sum_{k \in S} I_{kj})] = \sum_{i \in S} E[I_{ij}^2] + \sum_{i,k \in S, i \neq k} E[I_{ij} I_{kj}] =$
    $= \sum_{i \in S} P[h(i) = j] + \sum_{i,k \in S, i \neq k} E[h(i) = j \text{ a } h(k) = j] = \alpha + \frac{n(n-1)}{m^2}$ ③
- $Var(A_j) = \alpha(1 - 1/m)$
  - $Var(A_j) = E[A_j^2] - E^2[A_j] = \alpha(1 + \alpha - 1/m) - \alpha^2$

1. Systém hešovacích funkcí $\mathcal{H}$ je silně k-nezávislý, pokud náhodně zvolená $h \in \mathcal{H}$ splňuje $P[h(x_i) = z_i \text{ pro všechna } i = 1, \ldots, k] = \frac{1}{m^k}$ pro všechna po dvou různá $x_1, \ldots, x_k \in U$ a všechna $z_1, \ldots, z_k \in M$.
2. Druhá rovnost plyne z linearity střední hodnoty, druhá z definice střední hodnoty a třetí z 1-nezávislosti.
3. Druhá rovnost plyne z distribučního zákona a linearity střední hodnoty a poslední rovnost plyne ze silné 2-nezávislosti.

Hešování se separovanými řetězci: Složitost operace Find

Očekávaný počet porovnání při úspěšné operaci Find

- Celkový počet porovnání při vyhledání všech prvků dělíme počtem prvků
- Předpokládáme silnou 2-nezávislost hešovacího systému
- Celkový počet porovnání při vyhledání všech prvků je $\sum_j \sum_{k=1}^{A_j} k = \sum_j \frac{A_j(A_j+1)}{2}$
- Očekávaný počet porovnání je $1 + \frac{\alpha}{2} - \frac{1}{2m}$
  - $E\left[\frac{1}{n}\sum_j \frac{A_j(A_j+1)}{2}\right] = \frac{1}{2n}(E[\sum_j A_j] + \sum_j E[A_j^2]) = \frac{1}{2n}(n + m\alpha(1 + \alpha - \frac{1}{m}))$

Očekávaný počet porovnání při neúspěšné operaci Find

- Počet porovnání při neúspěšném hledání prvku x je počet prvků $i \in S$ splňující $h(i) = h(x)$
- Tedy chceme spočítat $E[|\{i \in S;\ h(i) = h(x)\}|]$
- Předpokládáme c-universální hešovací systém
- Použitím linearity střední hodnoty dostáváme
  - $E[|\{i \in S;\ h(i) = h(x)\}|] = \sum_{i \in S} P[h(i) = h(x)] \leq \sum_{i \in S} \frac{c}{m} = c\alpha$

Hešování se separovanými řetězci: Nejdelší řetězec

Definice

Posloupnost náhodných jevů $E_n, n \in \mathbb{N}$ se vyskytuje s velkou pravděpodobností, pokud existují $c > 1$ a $n_0 \in N$ takové, že pro každé $n \geq n_0$ platí $P[E_n] \geq 1 - \frac{1}{n^c}$.

Triviální příklad

Jestliže náhodně hodíme n míčů do n košů, pak s velkou pravděpodobností jsou alespoň dva koše neprázdné. ①

Délka nejdelšího řetězce

Pokud $\alpha = \Theta(1)$ a systém hešovacích funkcí je úplně náhodný, pak délka nejdelšího řetězce $\max_{j \in M} A_j = \Theta(\frac{\log n}{\log \log n})$ s velkou pravděpodobností. Platí i pro

- $\frac{\log n}{\log \log n}$-nezávislý systém (Schmidt, Siegel, Srinivasan [24])
- tabulkové hešování (Pătraşcu, Thorup [22])

Očekávaná délka nejdelšího řetězce (Důsledek)

Pokud $\alpha = \Theta(1)$ a systém hešovacích funkcí je úplně náhodný, pak očekávaná délka nejdelšího řetězce je $E[\max_{j \in M} A_j] = \Theta(\frac{\log n}{\log \log n})$.

1. $P[E_n] = 1 - \frac{1}{n^{n-1}} \geq 1 - \frac{1}{n^2}$ pro $n \geq 3$.

Hešování se separovanými řetězci: Nejdelší řetězec

Délky nejdelšího řetězce

Pokud $\alpha = \Theta(1)$ a systém hešovacích funkcí je úplně náhodný, pak délka nejdelšího řetězce $\max_{j \in M} A_j = \Theta(\frac{\log n}{\log \log n})$ s velkou pravděpodobností.

Chernoffův odhad

Nechť $X_1, \dots, X_n$ jsou nezávislé náhodné proměnné mající hodnoty $\{0, 1\}$. Označme $X = \sum_{i=1}^n X_i$ a $\mu = E[X]$. Pak pro každé $c > 0$ platí

$$P[X > c\mu] < \frac{e^{(c-1)\mu}}{c^{c\mu}}.$$

Důkaz: $\max_{j \in M} A_j = \mathcal{O}\left(\frac{\log n}{\log \log n}\right)$ s velkou pravděpodobností

- Nechť $\epsilon > 0$ a $c = (1+\epsilon)\frac{\log n}{\mu \log \log n}$. Tedy $c\mu = (1+\epsilon)\frac{\log n}{\log \log n}$
- Platí $P[\max_j A_j > c\mu] = P[\exists j : A_j > c\mu] \leq \sum_j P[A_j > c\mu] = mP[A_1 > c\mu]$
- Aplikujeme Chernoffův odhad na proměnné I_{i1} pro $i \in S$: $\mu = E[A_1] = \alpha$
- Platí $P[\max_j A_j > c\mu] \leq mP[A_1 > c\mu] < me^{-\mu} e^{c\mu - c\mu \log c}$

Hešování se separovanými řetězci: Nejdelší řetězec

Důkaz: $\max_{j \in M} A_j = \mathcal{O}\left(\frac{\log n}{\log\log n}\right)$ s velkou pravděpodobností

- Nechť $\epsilon > 0$ a $c = (1+\epsilon)\frac{\log n}{\mu \log\log n}$. Tedy $c\mu = (1+\epsilon)\frac{\log n}{\log\log n}$

$$
\begin{aligned}
P[\max_j A_j > c\mu] &< me^{-\mu} e^{c\mu - c\mu \log c} \\
&= me^{-\mu} e^{(1+\epsilon)\frac{\log n}{\log\log n} - (1+\epsilon)\frac{\log n}{\log\log n}\log\left(\frac{(1+\epsilon)\log n}{\mu\log\log n}\right)} \\
&= me^{-\mu} n^{\frac{1+\epsilon}{\log\log n} - \frac{1+\epsilon}{\log\log n}\log\left(\frac{(1+\epsilon)\log n}{\mu\log\log n}\right)} \\
&= me^{-\mu} n^{\frac{1+\epsilon}{\log\log n} - (1+\epsilon) + \frac{1+\epsilon}{\log\log n}\log\left(\frac{\mu}{1+\epsilon}\log\log n\right)} \\
&= \frac{m}{n^{1+\frac{\epsilon}{2}}} e^{-\mu} n^{-\frac{\epsilon}{2} + \frac{1+\epsilon}{\log\log n} + (1+\epsilon)\frac{\log\left(\frac{\mu}{1+\epsilon}\log\log n\right)}{\log\log n}} \\
&< \frac{1}{\alpha n^{\frac{\epsilon}{2}}} e^{-\mu} n^0 < \frac{1}{n^{\frac{\epsilon}{3}}} \qquad \text{. . . pro dostatečně velká } n
\end{aligned}
$$

Protože $-\frac{\epsilon}{2} + \frac{1+\epsilon}{\log\log n} + (1+\epsilon)\frac{\log\left(\frac{\mu}{1+\epsilon}\log\log n\right)}{\log\log n} < 0$ pro dostatečně velká n.

- Tedy $P[\max_j A_j \leq (1+\epsilon)\frac{\log n}{\log\log n}] > 1 - \frac{1}{n^{\frac{\epsilon}{3}}}$.

Více-přihrádkové hešování se separovanými řetězci

2-přihrádkové hešování

Prvek x může být uložen v přihrádce $h_1(x)$ nebo $h_2(x)$ a nový prvek vkládáme do přihrádky s menším počtem prvků, kde h_1 a h_2 jsou dvě hešovací funkce.

2-přihrádkové hešování: Délka nejdelšího řetězce (bez důkazu)

Očekávaná délka nejdelšího řetězce je $\mathcal{O}(\log \log n)$.

k-přihrádkové hešování

Prvek x může být uložen v přihrádkách $h_1(x), \ldots h_k(x)$ a nový prvek vkládáme do přihrádky s menším počtem prvků, kde $h_1, \ldots h_k$ jsou hešovací funkce.

k-přihrádkové hešování: Délka nejdelšího řetězce (bez důkazu)

Očekávaná délka nejdelšího řetězce je $\mathcal{O}\left(\frac{\log \log n}{\log k}\right)$.

Obsah

Lineární přidávání

Cíl

Chtěli bychom ušetřit paměť, a tak prvky budeme ukládat přímo do tabulky.

Operace Insert

Nový prvek x vložíme do prázdné přihrádky $h(x) + i \mod m$ s nejmenším možným $i \geq 0$.

Operace Find

Iterujeme dokud nenajdeme prvek nebo prázdnou přihrádku.

Operace Delete

- Líná varianta: Přihrádku smazaného prvku označkujeme, aby následné operace Find pokračovali v hledání
- Varianta bez značkování: Zkontroluje a přesouvá prvky v celém řetězci

Lineární přidávání: Složitost operací Find, Insert a Delete

Předpoklady

- $m \geq (1+\epsilon)n$
- Pokud přihrádky po smazaných prvcích jen značkujeme, pak n je součet počtu prvků a označkovaných přihrádek

Očekávaný počet porovnání při operaci Insert je

- $\mathcal{O}\left(\frac{1}{\epsilon^2}\right)$ pro úplně náhodný systém (Knuth, 1963 [15])
- konstantní pro $\log(n)$-nezávislý systém (Schmidt, Siegel, 1990 [23])
- $\mathcal{O}\left(\frac{1}{\epsilon^{\frac{13}{6}}}\right)$ pro 5-nezávislý systém (Pagh, Pagh, Ruzic, 2007 [19])
- $\mathcal{O}(\log n)$ pro 4-nezávislý systém (Pǎtraşcu, Thorup, 2010 [21]) 1
- $\mathcal{O}\left(\frac{1}{\epsilon^2}\right)$ pro tabulkové hešování (Pǎtraşcu, Thorup, 2012 [22])

1. Existuje 4-nezávislý hešovací systém a posloupnost operací Insert nezávislá na vybrané hešovací funkci taková, že očekávaná složitost je $\Omega(\log n)$.

Lineární přidávání: Analýza

Počet prvků od dané přihrádky do nejbližší volné přihrádky

Jestliže $\alpha < 1$ a systém hešovacích funkcí je úplně náhodný, pak očekávaný počet porovnání klíčů je $\mathcal{O}(1)$. ①

Důkaz

1. Nechť $1 < c < \frac{1}{\alpha}$ a $q = \left(\frac{e^{c-1}}{c^c}\right)^\alpha$
   - Platí $0 < q < 1$ ②
2. Nechť $p_t = P[|\{x \in S;\ h(x) \in T\}| = t]$ je pravděpodobnost, že do dané množiny přihrádek T velikosti t je zahešováno t prvků. Pak $p_t < q^t$. ③
   - Nechť X_i je náhodná proměnná indikující, zda prvek i je zahěšován do T
   - Nechť $X = \sum_{i \in S} X_i$ a $\mu = E[X] = t\alpha$
   - Platí $c\mu = c\alpha t < t$
   - Chernoff: $p_t = P[X = t] \leq P[X > c\mu] < \left(\frac{e^{c-1}}{c^c}\right)^\mu = q^{\frac{\mu}{\alpha}} = q^t$
3. Nechť b je nějaká přihrádka. Nechť p'_k je pravděpodobnost, že přihrádky b až $b + k - 1$ jsou obsazeny a $b + k$ je první volná přihrádka. Pak $p'_k < \frac{q^k}{1-q}$. ④
   - $p'_k < \sum_{s=0}^\infty p_{s+k} < q^k \sum_{s=0}^\infty q^s = \frac{q^k}{1-q}$
4. Očekávaný počet porovnání klíčů je $\sum_{k=0}^m kp'_k < \frac{1}{1-q} \sum_{k=0}^\infty kq^k = \frac{2-q}{(1-q)^3}$

1. Neúspěšná operace Find musí dojít až k volné přihrádce a též operace Insert, pokud cestou není přihrádka označená operací Delete. Úspěšná operace Find může porovnat méně prvků. Složitost operace Delete se v různých verzí liší, ale v rozumných implementacích dojde nejhůře k nejbližší volné přihrádce. Knuth [15] spočítal očekávanou složitost přesně, ale výpočet je náročný.
2. Zjevně $q > 0$. Chceme dokázat, že $\frac{e^{c-1}}{c^c} = e^{c-1-c\log c} < 1$. Musíme tedy dokázat, že $c - 1 - c\log c < 0$ pro $c > 1$. Pro $c = 1$ máme $c - 1 - c\log c = 0$, abychom dokázali ostrou nerovnost pro $c > 1$, ukážeme, že funkce $c - 1 - c\log c$ je pro $c > 1$ klesající. Derivace $1 - \log c - 1$ je záporná pro $c > 1$.
3. Zde uvažujeme prvky, které hešovací funkce zobrazí do daných přihrádek, a nikoliv prvky, které se do daných přihrádek dostanou vlivem lineárního přidávání.
4. Tedy přihrádky $b - s$ až $b + k - 1$ jsou obsazeny pro nějaké s. Indexy přihrádek počítáme modulo m.

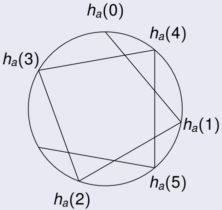
Lineární přidávání: Proč 2-nezávislý systém nestačí?

Modulení reálných čísel

- Pro $x \in \mathbb{R}$ a $m > 0$ existuje právě jedno $k \in \mathbb{Z}$ takové, že $0 \leq x - km < m$
- Pro $x \in \mathbb{R}$ a $m > 0$ definujme $x \bmod m = x - km$

Kombinace systému Multiply-shift a Lineárního přidávání

- Multiply-shift: $h(x) = (ax \bmod 2^w) >> (w - l) = \left\lfloor \frac{ax \bmod 2^w}{2^{w-l}} \right\rfloor = \lfloor h'(x) \rfloor$,
- kde $h'(x) = \frac{ax \bmod 2^w}{2^{w-l}} = \frac{ax}{2^{w-l}} \bmod 2^l = (\frac{a}{2^{w-l}} \bmod m)x \bmod m = h'(1)x \bmod m$
- Platí $h'(x) - h'(y) \bmod m = h'(1)x - h'(1)y \bmod m = h'(x - y) \bmod m$
- Speciálně $h'(x + 1) - h'(x) \bmod m = h'(1)$

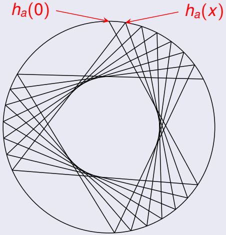


- Jaká je očekávaná složitost vložení prvků $S = \{1, \ldots, n\}$?

Lineární přidávání: Proč 2-nezávislý systém nestačí?

Kombinace systému Multiply-shift a Lineárního přidávání

- Platí $h'(ix) \mod m = h'(1)ix \mod m = ih'(x) \mod m$
- Jestliže $h'(x) \leq 1$ pro nějaké $x \in S$, pak $h'(ix) \leq i$
- Tedy prvky $x, 2x, \ldots, kx$ padnou do nejvýše k sousedních přihrádek
- Navíc prvky $x + j, 2x + j, \ldots, kx + j$ též padnou do nejvýše k sousedních přihrádek pro $j = 0, \ldots, x - 1$



Lineární přidávání a hešovací systémy (Pǎtraşcu, Thorup, 2010 [21])

- Multiply-shift má očekávanou složitost $\Theta(\log n)$ operace Find
- Existuje 2-nezávislý systém s očekávanou složitostí $\Theta(\sqrt{n})$ operace Find

Obsah

Kukačkové hešování (Pagh, Rodler [20])

Popis

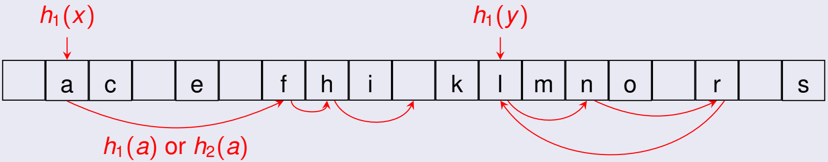
Pro dvě hešovací funkce h_1 a h_2 prvek x musí být uložen v přihrádce $h_1(x)$ nebo $h_2(x)$. V jedné přihrádce může být uložen nejvýše jeden prvek.

Operace Find a Delete

Triviální, složitost $\mathcal{O}(1)$ v nejhorším případě.

Příklad operace Insert

- Úspěšné vložení prvku x do přihrádky $h_1(x)$ po třech realokacích
- Prvek y není možné vložit do $h_1(y)$



Kukačkové hešování: Algoritmus pro operaci Insert

Vložení prvku x do tabulky T

```
pos ← h1(x)
for n krát [1] do
    if T[pos] je prázdná then
        T[pos] ← x
        return
    swap(x, T[pos])
    if pos == h1(x) [2] then
        pos ← h2(x)
    else
        pos ← h1(x)
rehash()
insert(x)
```

Rehash

- Náhodně vygenerujeme nové hešovací funkce h_1 a h_2 z $\mathcal{H}$
- Můžeme zvětšit velikost tabulky
- Vložíme všechny prvky do nové tabulky [3]

1. Po n pokusech jsme už určitě v cyklu. Lze ukázat, že v cyklu jsme s velkou pravděpodobností už po $\Omega(\log n)$ krocích.
2. Potřebuje najít druhou pozici, ve které prvek x může být uložen.
3. Při vkládání prvků do nové tabulky může dojít k Rehash, takže si při implementaci musíme dát pozor, abychom některé prvky neztratili.

Kukačkové hešování: Analýza

Neorientovaný kukačkový graf G

- Vrcholy jsou přihrádky tabulky
- Hrany jsou dvojice $\{h_1(x), h_2(x)\}$ pro všechny prvky $x \in S$.

Vlastnosti kukaččího grafu

- Operace Insert postupuje po cestě z vrcholu $h_1(x)$ do prázdné pozice ①
- Nový prvek nemůže být vložen, jestliže komponenta obsahující $h_1(x)$ obsahuje kružnici

Lemma

Nechť $c > 1$ a $m \geq 2cn$. Pro dané přihrádky i a j je pravděpodobnost, že mezi i a j existuje cesta délky k, nejvýše $\frac{1}{mc^k}$.

Složitost operace Insert bez přehešování

Nechť $c > 1$ a $m \geq 2cn$. Očekávaná délka cesty je $\mathcal{O}(1)$.

Počet přehešování

Nechť $c > 2$ a $m \geq 2cn$. Očekávaný počet přehešování při vkládání n prvků do tabulky velikosti m je $\mathcal{O}(1)$. ②

1. Z přihrádek komponenty tvořící cestu je právě jedna volná, ale nemusí to být přihrádka koncového vrcholu cesty.
2. Předpokládáme, že na začátku je tabulka prázdná a chceme vytvořit tabulku velikosti m s n prvky.

$k = 1$ For one element, the probability that it forms an edge ij is $\frac{2}{m^2}$. So, the probability that there is an edge ij is at most $\frac{2n}{m^2} \leq \frac{1}{mc}$.

$k > 1$ There exists a path between i and j of length k if there exists a path from i to u of length $k-1$ and an edge uj. For one position u, the i-u path exists with probability $\frac{1}{mc^{k-1}}$. The conditional probability that there exists the edge uj if there exists i-u path is at most $\frac{1}{mc}$ because some elements are used for the i-u path. By summing over all positions u, the probability that there exists i-j path is at most $m\frac{1}{mc^{k-1}}\frac{1}{mc} = \frac{1}{mc^k}$.

Insert without rehashing:

- Using the previous lemma for all length k and all end vertices j, the expected length of the path during operation Insert is $m\sum_{k=1}^{n} k\frac{1}{mc^k} \leq \sum_{k=1}^{\infty} \frac{k}{c^k} = \frac{c}{(c-1)^2}$.

Number of rehashes:

- Using the previous lemma for all length k and all vertices $i = j$, the probability that the graph contains a cycle is at most $m\sum_{k=1}^{\infty} \frac{1}{mc^k} = \frac{1}{c-1}$.
- The probability that inserting rehashes z times is at most $\frac{1}{(c-1)^z}$.
- The expected number of rehashes is at most $\sum_{z=0}^{\infty} z\frac{1}{(c-1)^z} = \frac{c-1}{(c-2)^2}$.

Kukačkové hešování: Analýza

Složitost operace Insert bez přehešování

Nechť $c > 1$ a $m \geq 2cn$. Očekávaná délka cesty je $\mathcal{O}(1)$.

Počet přehešování

Nechť $c > 2$ a $m \geq 2cn$. Očekávaný počet přehešování při vkládání n prvků do tabulky velikosti m je $\mathcal{O}(1)$. ①

Složitost operace Insert včetně přehešování

Nechť $c > 2$ a $m \geq 2cn$ a hešovací systém je úplně nezávislý. Pak očekávaná amortizovaná složitost operace Insert je $\mathcal{O}(1)$.

Pagh, Rodler [20]

Jestliže $c > 1$ a $m \geq 2cn$ a hešovací systém je $\log n$-nezávislý, pak očekávaná amortizovaná složitost operace Insert je $\mathcal{O}(1)$.

Pătraşcu, Thorup [22]

Jestliže $c > 1$ a $m \geq 2cn$ a použijeme tabulkové hešování, pak časová složitost vytvoření statické Kukačkové tabulky je $\mathcal{O}(n)$ s velkou pravděpodobností.

1. Předpokládáme, že na začátku je tabulka prázdná a chceme vytvořit tabulku velikosti m s n prvky.

Hešování: Další možnosti

Kvadratické prohledávání

Vložit prvek x do prázdné přihrádky $h(x) + ai + bi^2 \mod m$ s nejmenším možným $i \geq 0$, kde a, b jsou pevné konstanty.

Dvojité hešování

Vložit prvek x do prázdné přihrádky $h_1(x) + ih_2(x) \mod m$ s nejmenším možným $i \geq 0$, kde h_1, h_2 jsou dvě hešovací funkce.

Brentova varianta operace Insert

Jestliže přihrádka

- $b = h_1(x) + ih_2(x) \mod m$ je obsazená prvkem y
- $b + h_2(x) \mod m$ je taky obsazená
- $c = b + h_2(y) \mod m$ je prázdná,

pak přesuneme prvek y to přihrádky c a prvek x vložíme do b. Tímto se zkrátí očekávaná doba hledání.

Obsah

Bloom filtry (Bloom, 1970 [2]

cíl

- chtěli bychom datovou strukturu, která umí přidávat prvky a zjišťovat, zda byl daný prvek vložen
- máme dlouhé klíče a všechny se nám nevejdou do paměti ①
- nevadí nám, když datová struktura občas vrátí špatnou odpověď

Postup

- Máme množinu S velikosti n z univerza U
- Použijeme bitové pole M velikosti m ②
- Zvolíme k hešovacích funkcí $h_1, \ldots, h_k : U \to M$ ③
- Na počátku jsou všechny bity nulové
- Při vložení prvku $x \in S$ nastavíme bity $h_1(x), \ldots, h_k(x)$ na jedničku
- Jestliže pro $y \in U$ je některý z $h_1(y), \ldots, h_k(y)$ nulový, pak určitě y nebyl vložen
- Jestliže jsou všechny bity $h_1(y), \ldots, h_k(y)$ jedničkové, tak si nemůžeme být jisti, že prvek nebyl vložen

1. klíče můžou být uživatelem navštívené url adresy nebo všechna analyzovaná řešení genetického algoritmu. nedostatečná kapacita paměti může být způsobena obrovským množství dat nebo omezeným množstvím paměti, například v sušenkách prohlížečů.
2. Pamatujeme si jen m bitů, nikoliv n klíčů.
3. Pro potřeby analýzy budeme předpokládat, že hešovací systém je úplně nezávislý.

Bloom filtry: Analýza

Cíl

- Jaká je pravděpodobnost, že dostaneme kladnou odpověď, i když prvek nebyl vložen?
- Jak zvolik počet funkcí k, abychom minimalizovali pravděpodobnost špatné odpovědi?

Analýza

- Pravděpodobnost, že pozice $h_1(y)$ je nulová je $\left(1 - \frac{1}{m}\right)^k n$ ①
- $\left(1 - \frac{1}{m}\right)^{kn} = \left(\left(1 + \frac{-1}{m}\right)^m\right)^{\frac{kn}{m}} \approx e^{-\frac{kn}{m}} =: p$ ②
- Pravděpodobnost, že všechny bity $h_1(y), \ldots, h_k(y)$ jsou jedničkové, je $(1-p)^k$
- Chceme najít k minimalizující $(1-p)^k = e^{k \log(1-p)}$ ③
- Z $k = -\frac{m}{n} \log p$ plyne $k \log(1-p) = -\frac{m}{n} \log(p) \log(1-p)$
- Ze symetrií funkcí $\log(p)\log(1-p)$ odhadneme, že maximum nastane uprostřed pro $p = \frac{1}{2}$ ④
- Tedy $k = \frac{m}{n} \log 2 \approx 0.69 \frac{m}{n}$
- Pravděpodobnost „false positive“ je přibližně $(1-p)^k = 2^{-\frac{m}{n}\log 2} \approx 0.62^{\frac{m}{n}}$

1. Pravděpodobnost, že $h_i(x) \neq h_1(y)$ je $\frac{1}{m}$. Funkcí h_i je k, prvků $x \in S$ je n a náhodné veličiny $h_i(x)$ jsou nezávislé.
2. Z matematické analýzy víme, že $\lim_{m\to\infty}(1 + \frac{a}{m})^m = e^a$. Předpokládáme, že $\frac{n}{m}$ je konstanta, jak později uvidíme, že k je též konstanta. Proto je exponent $\frac{kn}{m}$ konstantní. Z matematické analýzy bychom měli vědět, že chyba v aproximaci je $\mathcal{O}(\frac{1}{m})$.
3. Funkce e^a je rostoucí v a, takže stačí minimalizovat expoment.
4. Ve formálním důkazu bychom našli lokalní optima pomocí derivace a ověřili, že máme globální maximum.

Counting Bloom filtry, Fan et al. 2000 [7]

Cíl

Chtěli bychom v Bloom filtru umět mazat prvky.

Postup

- Na každé pozici v tabulce nebudeme mít jeden bit ale malý čítač
- Při operace INSERTse čítače $h_1(x), \dots, h_k(x)$ zvýší o jedna
- Při operaci DELETEse čítače $h_1(x), \dots, h_k(x)$ sníží o jedna
- Jestliže některý z čítačů $h_1(y), \dots, h_k(y)$ je nulový, pak y není přítomen
- Zvolíme $k = \frac{m}{n} \log 2$
- Jestliže všechny čítače $h_1(y), \dots, h_k(y)$ jsou kladné, pak y není přítomen s pravděpodovností přibližně $0.62^{\frac{m}{n}}$

Jak velký zvolit čítač, aby nedocházelo k přetečení?

- Maximální hodnota čítače je $\Theta(\frac{\log n}{\log \log n})$ s velkou pravděpodobností ①
- Potřebujeme $\Theta(\log \frac{\log n}{\log \log n})$-bitový čítač
- Bez důkazu: Pro 4-bitový čítač je pravděpodobnost přetečení menší než $1.37 \cdot 10^{15} m$

1. Počet přičtených jedniček je $nk = m \log 2$ a tato přičtení jsou náhodně distribuována mezi m přihrádek. Z analýzy hešovaní se separovanými řetězci víme, že když faktor zaplnění je konstantní, pak délka nejdelšího řetězce je $\Theta(\frac{\log n}{\log \log n})$.

Obsah

[1] R Bayer and E McCreight.
Organization and maintenance of large ordered indexes.
Acta Informatica, 1:173–189, 1972.

[2] Burton H Bloom.
Space/time trade-offs in hash coding with allowable errors.
Communications of the ACM, 13(7):422–426, 1970.

[3] Mark R Brown and Robert E Tarjan.
A representation for linear lists with movable fingers.
In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 19–29. ACM, 1978.

[4] Bernard Chazelle.
Lower bounds for orthogonal range searching: I. the reporting case.
Journal of the ACM (JACM), 37(2):200–212, 1990.

[5] Bernard Chazelle.
Lower bounds for orthogonal range searching: part ii. the arithmetic model.
Journal of the ACM (JACM), 37(3):439–463, 1990.

[6] Bernard Chazelle and Leonidas J Guibas.
Fractional cascading: I. a data structuring technique.
Algorithmica, 1(1-4):133–162, 1986.

[7] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder.

Summary cache: a scalable wide-area web cache sharing protocol.
IEEE/ACM Transactions on Networking (TON), 8(3):281–293, 2000.

[8] Michael L Fredman and Robert Endre Tarjan.
Fibonacci heaps and their uses in improved network optimization algorithms.
Journal of the ACM (JACM), 34(3):596–615, 1987.

[9] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran.
Cache-oblivious algorithms.
In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297, 1999.

[10] Leo J Guibas, Edward M McCreight, Michael F Plass, and Janet R Roberts.
A new representation for linear lists.
In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 49–60. ACM, 1977.

[11] Leo J Guibas and Robert Sedgewick.
A dichromatic framework for balanced trees.
In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 8–21. IEEE, 1978.

[12] Scott Huddleston and Kurt Mehlhorn.
A new data structure for representing sorted lists.
Acta informatica, 17(2):157–184, 1982.

[13] Donald B Johnson.
Priority queues with update and finding minimum spanning trees.
Information Processing Letters, 4(3):53–57, 1975.

[14] Donald E. Knuth.
Optimum binary search trees.
Acta informatica, 1(1):14–25, 1971.

[15] Donald Ervin Knuth.
Notes on ”open”addressing.
http://algo.inria.fr/AofA/Research/11-97.html, 1963.

[16] Erkki Mäkinen.
On top-down splaying.
BIT Numerical Mathematics, 27(3):330–339, 1987.

[17] Kurt Mehlhorn.
Sorting presorted files.
In *Theoretical Computer Science 4th GI Conference*, pages 199–212. Springer, 1979.

[18] Jürg Nievergelt and Edward M Reingold.
Binary search trees of bounded balance.
SIAM journal on Computing, 2(1):33–43, 1973.

[19] Anna Pagh, Rasmus Pagh, and Milan Ruzic.

Linear probing with constant independence.
In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 318–327, 2007.

[20] Rasmus Pagh and Flemming Friche Rodler.
Cuckoo hashing.
Journal of Algorithms, 51(2):122–144, 2004.

[21] Mihai Pǎtraşcu and Mikkel Thorup.
On the k-independence required by linear probing and minwise independence.
In *International Colloquium on Automata, Languages, and Programming*, pages 715–726, 2010.

[22] Mihai Pǎtraşcu and Mikkel Thorup.
The power of simple tabulation hashing.
Journal of the ACM (JACM), 59(3):14, 2012.

[23] Jeanette P Schmidt and Alan Siegel.
The spatial complexity of oblivious k-probe hash functions.
SIAM Journal on Computing, 19(5):775–786, 1990.

[24] Jeanette P Schmidt, Alan Siegel, and Aravind Srinivasan.
Chernoff-hoeffding bounds for applications with limited independence.
SIAM Journal on Discrete Mathematics, 8(2):223–250, 1995.

[25] Daniel D Sleator and Robert E Tarjan.

Amortized efficiency of list update and paging rules.
Communications of the ACM, 28(2):202–208, 1985.

[26] Daniel Dominic Sleator and Robert Endre Tarjan.
Self-adjusting binary search trees.
Journal of the ACM (JACM), 32(3):652–686, 1985.

[27] Jean Vuillemin.
A data structure for manipulating priority queues.
Communications of the ACM, 21(4):309–315, 1978.