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 2016/17

Poslední změna 13. února 2017

Základní informace

Kontant

E-mail fink@ktiml.mff.cuni.cz

Homepage https://ktiml.mff.cuni.cz/~fink/

Konzultace Individuální domluva

Podmínky zápočtu

Bude zadaných pět domácích úkolů po 100 bodech a k zápočtu musíte získat minimálně 350 bodů.

Obsah

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.

1. domácí úkol: Externí třídění

Stručné zadání

- Napsat program, který setřídí velký soubor čísel
- Celý vstup se nevejde do paměti RAM
- Všechny datové struktury i algoritmy si musíte naprogramovat sami, tj. nesmíte používat `std::vector`, `std::map`, `std::sort` a podobně
- Počet získaných bodů závisí na době běhu programu
- Programy testujte v Unixové laboratoři na Malé Straně, kde se k ostatním chovejte ohleduplně!
- Termín odevzdání: 30. 10. 2016
- Podrobnosti: `https://ktiml.mff.cuni.cz/~fink/`

Obsah

Amortizovaná analýza

V amortizované analýze je čas potřebný k vykonání posloupnosti operací v datové struktuře průměrován počtem vykonaných operací, např.

	nejhorší případ	amortizovaná složitost
Inkrementace n-bitového čítače	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Dynamického pole	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Metody

- Agregovaná analýza
- Účetní metoda
- Potenciální metoda

Obsah

Inkrementace binárního čítače

Binární čítač

- Máme n-bitový čítač
- 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 — tedy v průměru $k/2$-krát
- i-tý bit od konce se změní každých 2^i operací — tedy v průměru $k/2^i$-krát
- Celkový průměrný počet změn bitů je $\sum_{i=0}^{n} k/2^i \leq k \sum_{i=0}^{\infty} 1/2^i = 2k$

Inkrementace binárního čítače

Účetní metoda

- Změna jednoho bitu stojí jeden žeton a na každou operaci dostaneme dva žetony
- U každého jedničkového bitu si uschováme jeden žeton
- Při inkrementu máme vynulování jedničkových bitů předplaceno
- Oba žetony využijeme na jedinou změnu nulového bitu na jedničku a předplacení vynulování

Potenciální metoda

- Potenciál p_i i-tého bitu je 0, jestliže i-tý bit je nulový, jinak $p_i = 1$
- Potenciál čítače Φ je součet potenciálů všech bitů
- Potenciál před provedením operace je Φ a po provedení je Φ'
- Chceme dokázat: amortizovaný čas = skutečný čas + $(\Phi' - \Phi)$ ①
- Nechť j je vynulovaných jedniček při jedné operaci Increment
- Skutečný čas operace (počet změněných bitů) je $j + 1$
- $\Phi' - \Phi = 1 - j$
- Amortizovaný čas = skutečný čas + $(\Phi' - \Phi) = j + 1 + (1 - j) = 2$

1. 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 ($\Phi' > \Phi$) a naopak při dlouho trvajících operací potenciál neklesne příliš moc.

Inkrementace binárního čítače

Na zamyšlení

- Co přesně znamená celkový průměrný čas?
- Co přesně je amortizovaná složitost?
- Jaká je celková doba na provedení k operací?

Agregovaná složitost

- i-tý bit se při k operací změní nejvýše $\lceil k/2^i \rceil$-krát
- Celkový počet změn je nejvýše $\sum_{i=0}^{n-1} \lceil k/2^i \rceil \leq \sum_{i=0}^{n-1} (1 + k/2^i) \leq n + 2k$

Účetní metoda

- Na začátku dostaneme čítač a na každý jedničkový bit potřebujeme dát žeton
- Celkový počet spotřebovaných žetonů je nejvýše $n + 2k$

Potenciální metoda

- Potenciál po i-té operaci označme Φ_i a Φ_0 počáteční potenciál
- Amortizovaný čas i-té operace = skutečný čas i-té operace + $(\Phi_i - \Phi_{i-1})$
- Celkový čas = $\sum_{i=1}^{k}$ skutečný čas i-té operace =
  $k\cdot$ amortizovaný čas + $\sum_{i=1}^{k}(\Phi_{i-1} - \Phi_i)$ = $k\cdot$ amortizovaný čas + $\Phi_0 - \Phi_k \leq 2k + n$
  ①

1. Sečtení sumy $\sum_{i=1}^{k}(\Phi_{i-1} - \Phi_i) = \Phi_0 - \Phi_k$ se nazývá ”telescopic cancellation”. Poslední nerovnost plyne z faktů, že $\Phi_0 \leq n$ a $\Phi_k \geq 0$.

Obsah

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

Agregovaná analýza: Amortizovaná složitost ①

- 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 ②
- 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 i-té realokaci se kopíruje nejvýše $2k_i$ prvků pro $i \geq 2$
- Po prvé se kopíruje nejvýše $n_0 + k_1$ prvků, kde n_0 je počáteční počet prvků
- Celkový počet zkopírovaných prvků je nejvýše $n_0 + k_1 + \sum_{i \geq 2} 2k_i \leq n_0 + 2k$

1. 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 $\mathcal{O}(1)$.
2. Po realokaci a zkopírování je nové pole z poloviny plné. Musíme tedy přidat n prvků nebo smazat $n/2$ prvků, aby došlo k dalšímu kopírování.

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 lineární interpolaci v ostatních případech
- 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 $+(\Phi' - \Phi) \leq$ amortizovaný počet = 2
- Celkový počet zkopírovaných prvků při k operacích je nejvýše $2k + \Phi_0 - \Phi_k \leq 2k + n_0$
- Celková čas k operací je $\mathcal{O}(n_0 + k)$

1

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

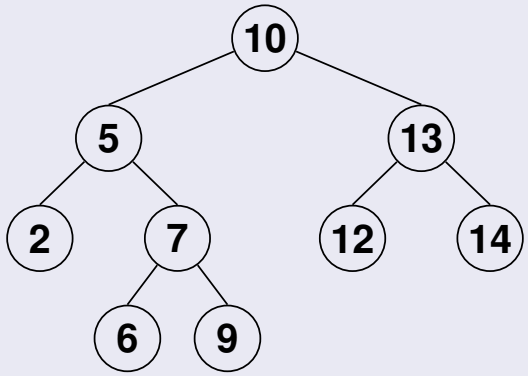
Obsah

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 $n - 1$

Váhově vyvážené stromy: BB[α]-strom

BB[α]-strom (Nievergelt, Reingold [16])

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

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. Do s_u započítáváme i vrchol u.
2. V literatuře můžeme najít různé varianty této podmínky. Podstatné je, aby oba postromy každého vrcholu měli „zhruba“ stejný počet vrcholů.
3. 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 2.2)
- 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 dvěmi operacemi Build podstromu u je $\Omega(s_u)$ operací Insert nebo Delete do postromu 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?

Amortizovaná časová složitost operací Insert a Delete: Potenciální metoda

- V této analýze uvažujeme jen čas na postavení podstromu, zbytek trvá $\mathcal{O}(\log n)$
- Nechť postavení podstromu s s_u prvky vyžaduje nejvýše ds_u instrukcí
- Potenciál $\Phi(u)$ vrcholu u definujeme

$$\Phi(u) = \begin{cases} 0 & \text{pokud } s_{u.left} = s_{u.right} = \frac{s_u}{2} \\ ds_u & \text{pokud } \max\{s_{u.left}, s_{u.right}\} = \alpha s_u \end{cases}$$

  a lineární interpolací v ostatních případech
- Potenciál BB[α]-stromu Φ je součet potenciálů vrcholů
- Při vložení/smazání prvku potenciál $\Phi(u)$ zvýší nejvýše o cd (Cvičení 2.3) ①
- Pro každý vrchol u: Skutečný počet instrukcí ve vrcholu $u + \Phi'(u) - \Phi(u) \leq$
  $\begin{cases} cd & \text{pokud } u \text{ leží na cestě z kořene do vloženého/smazaného prvku} \\ 0 & \text{jinak} \end{cases}$ ②
- Počet instrukcí pro jednu operaci $+ \Phi' - \Phi \leq cdh$, kde h je výška stromu
- $0 \leq \Phi \leq dhn = \mathcal{O}(n \log n)$ ③
- Celkový čas na k operací Insert nebo Delete je $\mathcal{O}((k+n)\log n)$

1. Potenciál $\Phi(u)$ může i klesnout. Konstanta c je závislá jen na parametru α.
2. Pokud u není na cestě z kořene do vloženého/smazaného prvku, tak skutečný čas ve vrcholu u je nula a $\Phi(u) = \Phi'(u)$. Pokud vrcholu u nepřestavíme podstrom, tak skutečný čas je nula potenciál vrcholu u vzroste nejvýše o cd. Na přestavění podstromu vrcholu u potřebujeme ds_u instrukcí a přesně o tolik klesne potenciál vrcholu u.
3. Součet potenciálů všech vrcholů v jedné libovolné vrstvě je nejvýše dn, protože každý vrchol patří do nejvýše jednoho podstromu vrcholu z dané vrstvy. Tudíž potenciál stromu Φ je vždy nejvýše dnh. Též lze nahlédnout, že každý vrchol je započítán v nejvýše h potenciálech vrcholů.

Cvičení:

2.1. Dokažte, že všechny listy v BB[α]-stromu jsou v hloubce $\Theta(\log n)$. Pokuste se určit co přesněji minimální a maximální hloubku listů.

2.2. Počet vrcholů v podstromu vrcholu u po vybalancování je s_u. Najděte minimální počet operací Insert a Delete, který způsobí porušení vyvažovací podmínky ve vrcholu u.

2.3. Najděte přesnou formuli pro potenciál vrcholu v amortizované analýze BB[α]-stromu v potenciální metodě. Dále přesně spočtěte, o kolik se maximálně zvýší potenciál vrcholu při vložení/smazání vrcholu v podstromu.

2.4. Pokud nejprve BB[α]-strom vybudujeme ze setříděného seznamu a poté provedeme k operací, tak celkový čas je $\mathcal{O}(n + k \log n)$, což je pro $k << n$ lepší odhad než $\mathcal{O}((k + n) \log n)$. Je tedy celkový čas na k operací Insert nebo Delete $\mathcal{O}(n + k \log n)$ začínáme-li z libovolného BB[α]-stromu s n listy (bez počátečního vyvážení)?

2.5. Vymyslete pravidla pro rotování v BB[α]-stromech při operacích Insert a Delete tak, aby složitost v nejhorším případě byla $\mathcal{O}(\log n)$. Pro jaké hodnoty parametru α dokážete splnit vyvažovací podmínku BB[α]-stromu?

Obsah

Staticky optimální strom

Cíl

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

Formálně

Máme prvky $x_1, \ldots, x_n$ s váhami $w_1, \ldots, w_n$. 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 3.1)

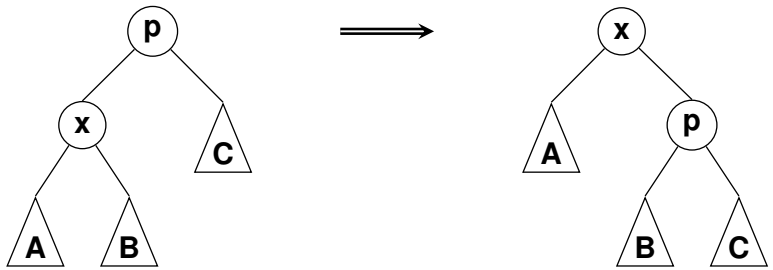
- $\mathcal{O}(n^3)$ – triviálně dynamickým programováním
- $\mathcal{O}(n^2)$ – vylepšené dynamické programování (Knuth [13])

Jak postupovat, když neznáme váhy předem?

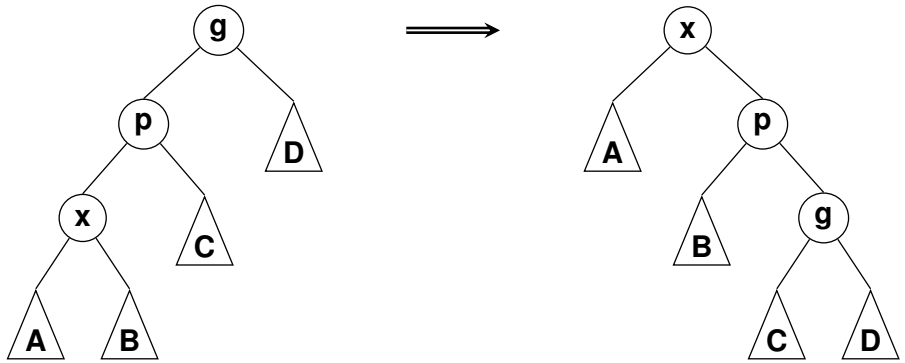
- Pomocí rotací bude udržovat často vyhledávané prvky blízko kořene
- Operací Splay „rotujeme“ zadaný prvek až do kořene
- Operace Find vždy volá Splay na hledaný prvek

Splay strom (Sleator, Tarjan [25]): Operace Splay prvku *x*

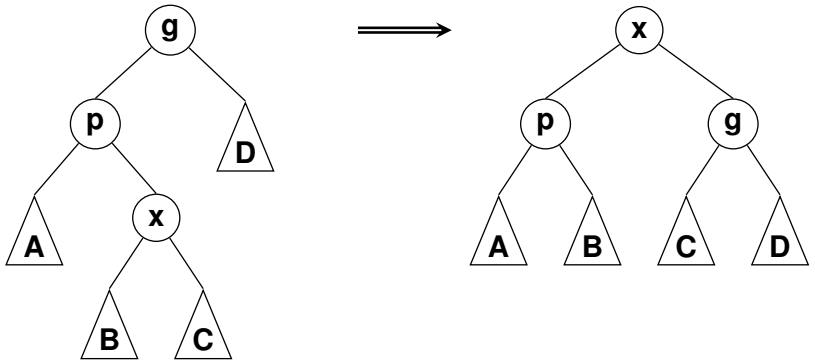
- Zig step: Otec *p* prvku *x* je kořen



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

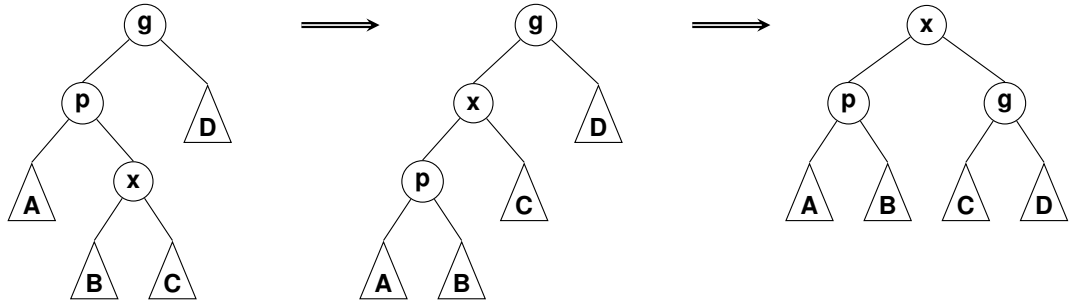


- Zig-zag step: *x* je pravý syn a *p* je levý syn nebo opačně

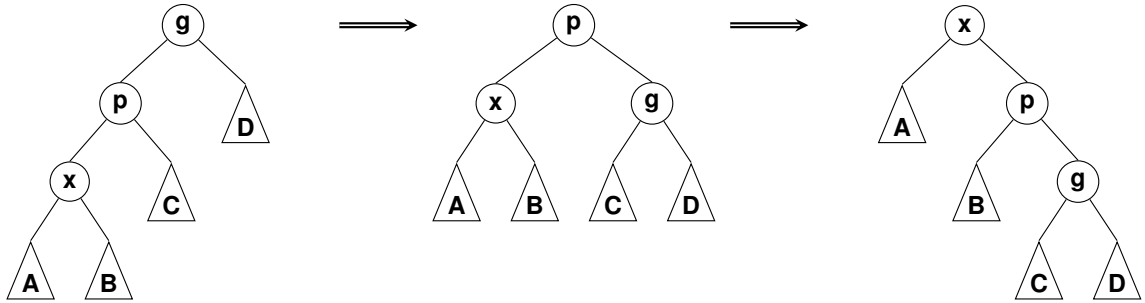


Splay strom: Operace Splay prvku *x*

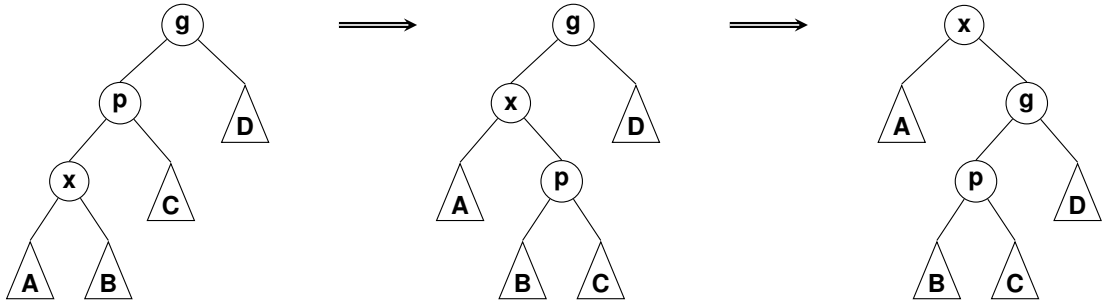
- Zig-zag step jsou pouze dvě (jednoduché) rotace prvku *x* s aktuálním otcem



- Zig-zig step jsou taky dvě rotace,



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



Splay strom: Amortizovaná analýza

Lemma

Jestliže $a + b \leq 1$, pak $\log_2(a) + \log_2(b) \leq -2$.

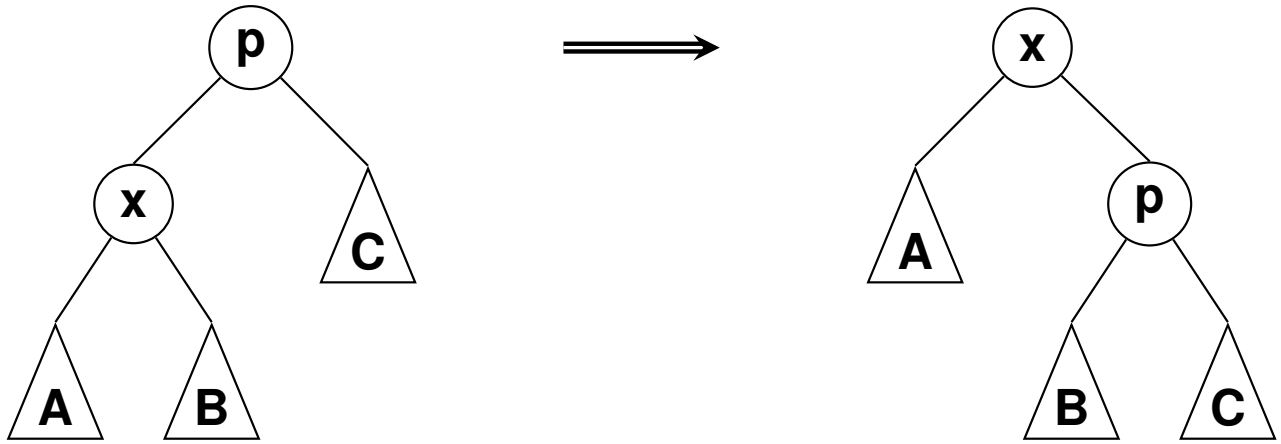
Důkaz

- Platí $4ab = (a+b)^2 - (a-b)^2$
- Z nerovností $(a-b)^2 \geq 0$ a $a+b \leq 1$ plyne $4ab \leq 1$
- Zlogaritmováním dostáváme $\log_2 4 + \log_2 a + \log_2 b \leq 0$

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 potenciál všech vrcholů
- s' a Φ' jsou velikosti a potenciály po jedné rotaci
- Analyzujeme počet jednoduchých rotací a potenciál je banka rotací

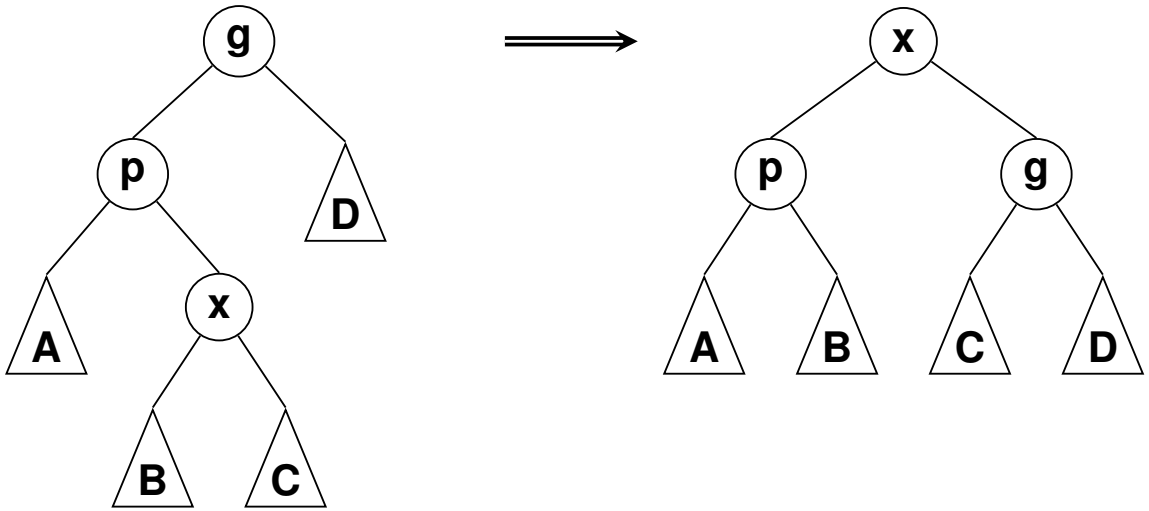
Splay strom: Zig step



Analýza

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

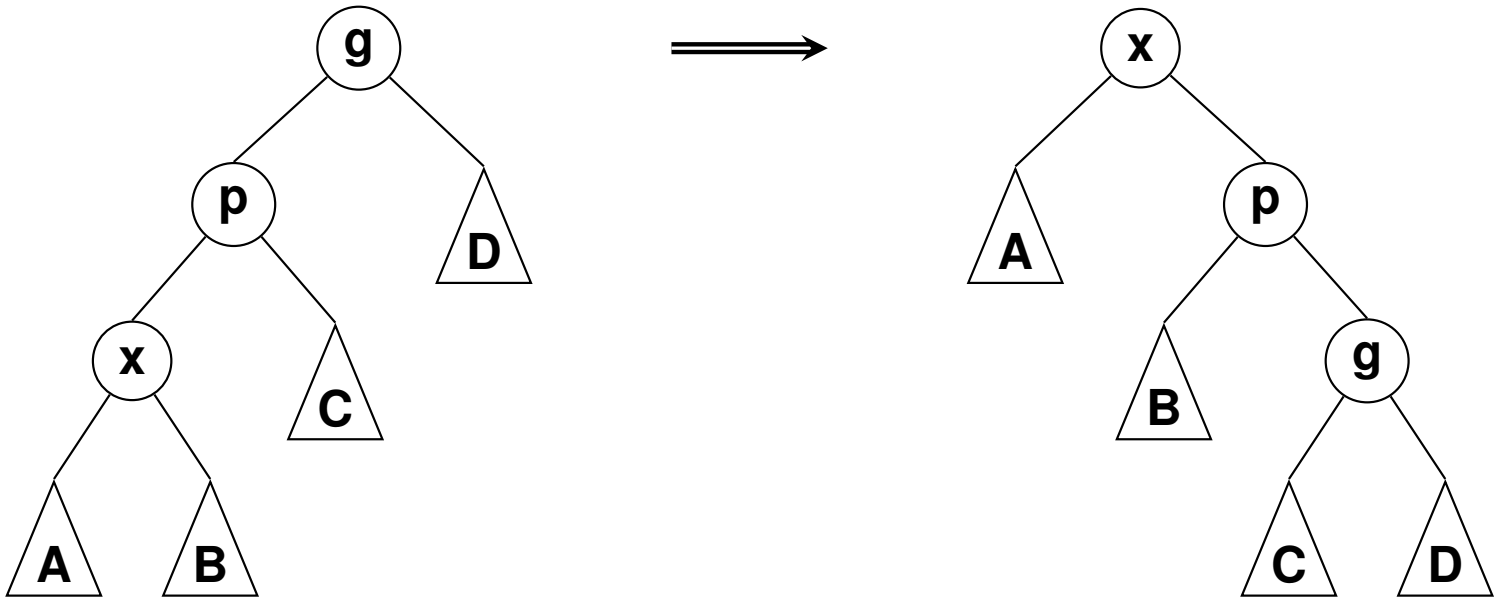
Splay strom: Zig-zag step



Analýza

1. $\Phi'(x) = \Phi(g)$
2. $\Phi(x) < \Phi(p)$
3. $\Phi'(p) + \Phi'(g) \leq 2\Phi'(x) - 2$
   - $s'(p) + s'(g) \leq s'(x)$
   - $\frac{s'(p)}{s'(x)} + \frac{s'(g)}{s'(x)} \leq 1$
   - Použijeme lemma: $a + b \leq 1 \Rightarrow \log_2(a) + \log_2(b) \leq -2$
   - $\log_2 \frac{s'(p)}{s'(x)} + \log_2 \frac{s'(g)}{s'(x)} \leq -2$
   - $\log_2 s'(p) + \log_2 s'(g) \leq 2 \log s'(x) - 2$
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 step



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

- Amortizovaný čas jedné zigzig nebo zigzag rotace:
  skutečný čas $+ \Phi' - \Phi \leq 2 + 3(\Phi'(x) - \Phi(x)) - 2 = 3(\Phi'(x) - \Phi(x))$
- Amortizovaný čas jedné zig rotace:
  skutečný čas $+ \Phi' - \Phi \leq 1 + \Phi'(x) - \Phi(x) \leq 1 + 3(\Phi'(x) - \Phi(x))$
- Nechť Φ_i je potenciál po i-tém kroku (zig, zig-zag nebo zig-zig step)
- Amortizovaný čas (počet jednoduchých rotací) jedné operace Splay:

$$\sum_{i\text{-tý krok}} (\text{skutečný čas} + \Phi_i - \Phi_{i-1}) \leq 1 + \sum_{i\text{-tý krok}} 3(\Phi_i(x) - \Phi_{i-1}(x))$$

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

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

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

Skutečný čas k operací Splay

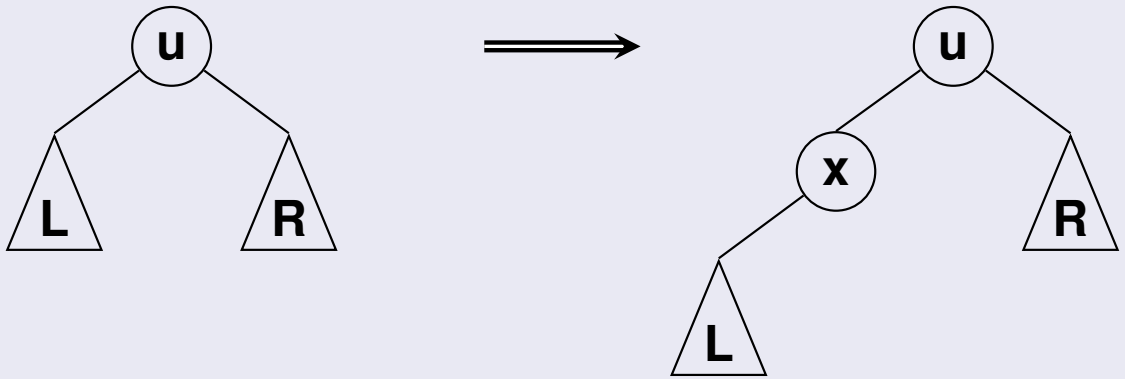
- Potenciál vždy splňuje $0 \leq \Phi \leq n \log_2 n$
- Rozdíl mezi konečným a počátečním potenciálem je nejvýše $n \log_2 n$
- Celkový čas k operací Splay je $\mathcal{O}((n+k)\log n)$

1. 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. Najdeme vrchol u s klíčem, který je nejbližší k x
2. Splay(u)
3. Vložit nový vrchol s prvkem x



Amortizovaná složitost

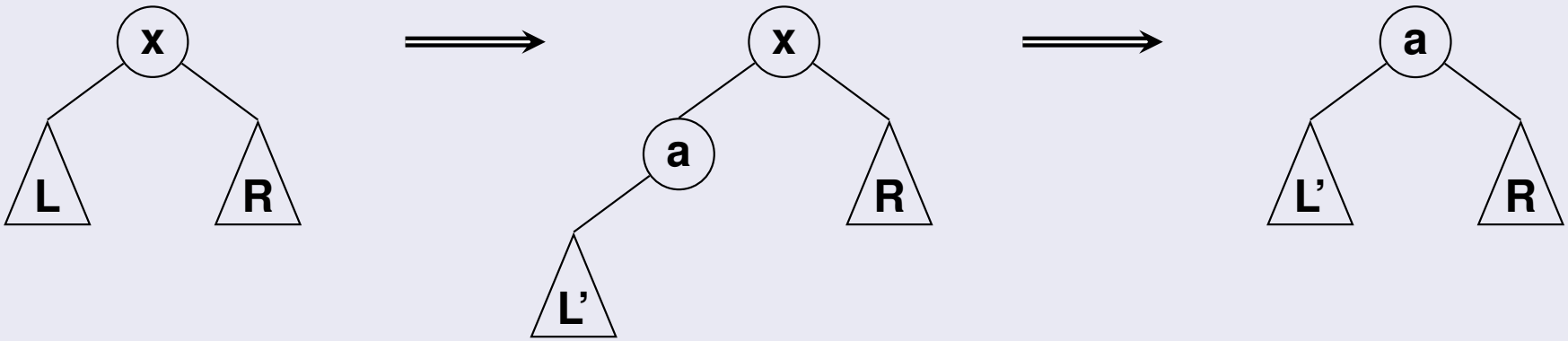
- Operace Find a Splay: $\mathcal{O}(\log n)$
- Vložením nového vrcholu potenciál Φ vzroste nejvýše o $\Phi'(x) + \Phi'(u) \leq 2 \log_2 n$
- Amortizovaná složitost operace Insert je $\mathcal{O}(\log n)$

Splay strom: Operace Delete

Algoritmus

```
1 Splay(x)
2 L ← levý podstrom x
3 if L je prázdný then
4 |   Smazat vrchol x
5 else
6 |   Najít největší prvek a v L
7 |   Splay(a)
8 |   L′ ← levý podstrom a
  |   # a nemá pravého syna
9 |   Sloučit vrcholy x a a
```

Pokud L je neprázdný, tak



2. 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 „zákeřném“ testu
- Termín odevzdání: 13. 11. 2016
- Generátor dat a další podrobnosti: `https://ktiml.mff.cuni.cz/~fink/`

Cvičení:

3.1. Najděte algoritmus, který sestrojí staticky optimální strom v čase $\mathcal{O}(n^3)$.

3.2. Nechť $K[i,j]$ je prvek v kořeni staticky optimálního stromu obsahující pouze prvku $x_i, \ldots, x_j$. Dokažte, že $K[i,j-1] \leq K[i,j] \leq K[i+1,j]$.

3.3. Pomocí nerovnosti z předchozího příkladu najděte algoritmus, který sestrojí staticky optimální strom v čase $\mathcal{O}(n^2)$.

3.4. Najděte posloupnost operací Find, která z libovolného počátečního Splay stromu vytvoří cestu (strom výšky $n-1$).

3.5. Uvažujme naivní splay strom, který využívá jen jednoduché rotace. Najděte posloupnost operací Find, která má v naivním splay stromu složitost $\Omega(nk)$, kde $k \geq n$ je vámi zvolený počet operací Find.

3.6. Mějme ve splay stromu uloženu množinu prvků S a analyzujme vyhledávání podmnožiny $S' \subseteq S$. Nejprve každý prvek S' jednou vyhledáme (samozřejmě s využitím operace Splay). Dokažte, že následné (opakované) vyhledání prvků z množiny S' má amortizovanou složitost $\mathcal{O}(\log |S'|)$ a spočtěte celkovou složitost k vyhledání.

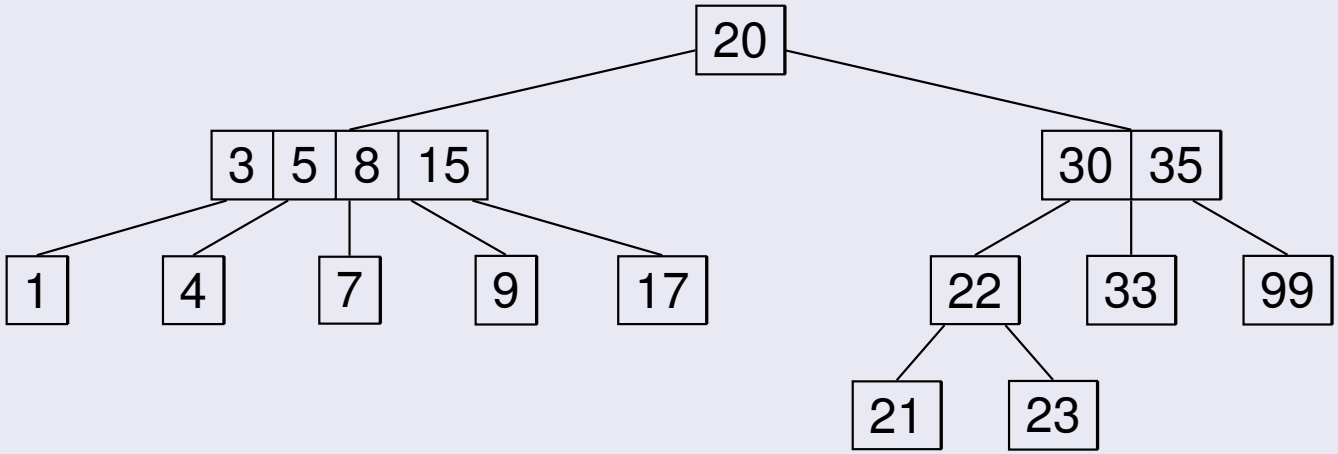
Obsah

Vyhledávací strom

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



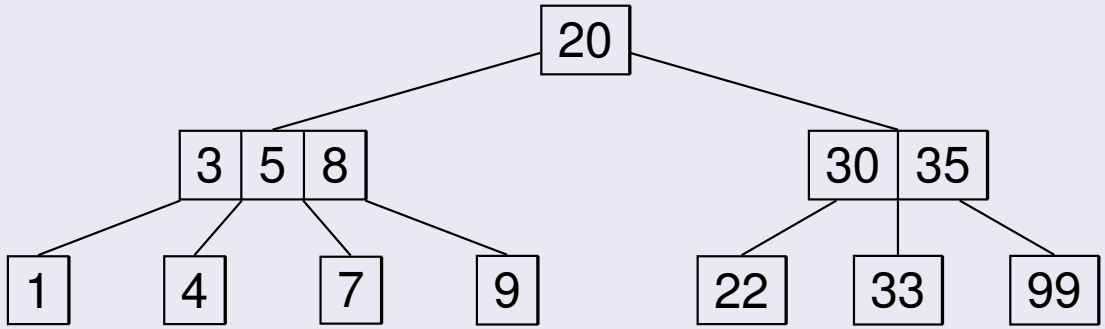
Obsah

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

Vlastnosti

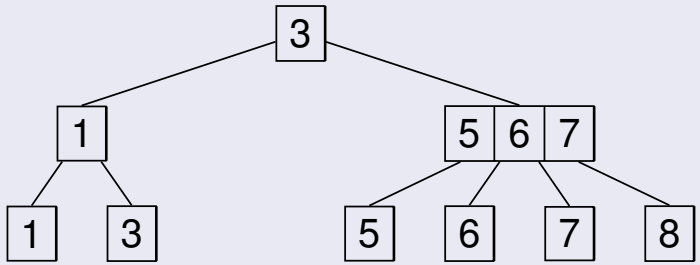
- a, b jsou celá čísla splňující $a \geq 2$ a $b \geq 2a - 1$
- (a,b)-strom je vyhledávací strom
- Všechny vnitřní vrcholy kromě kořene mají alespoň a synů a nejvýše b synů
- Kořen má nejvýše b synů
- Všechny listy jsou ve stejné výšce
- Pro zjednodušení uvažujeme, že prvky jsou jen v listech

Příklad: (2,4)-strom

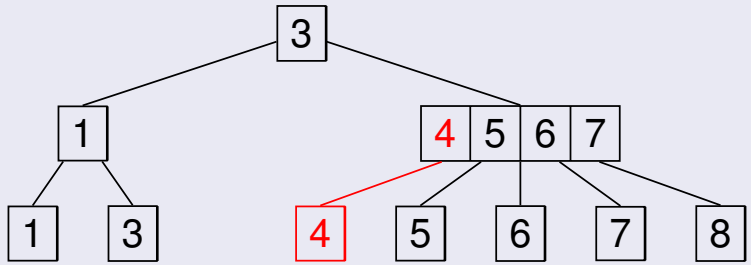


(a,b)-strom: Operace Insert

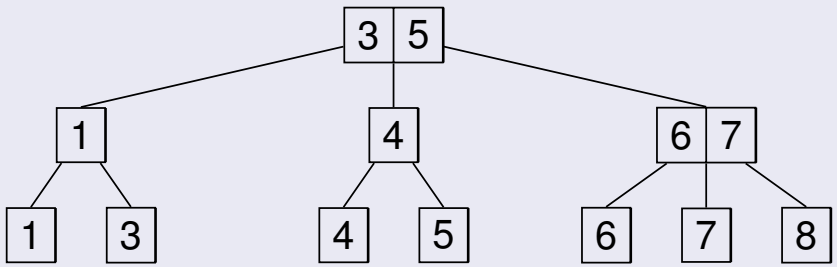
Vložte prvek s klíčem 4 do následujícího (2,4)-stromu



Nejprve najdeme správného otce, jemuž přidáme nový list



Opakovaně rozdělujeme vrchol na dva



(a,b)-strom: Operace Insert

Algoritmus

1 Najít otce v, kterému nový prvek patří
2 Přidat nový list do v
3 **while** $\deg(v) > b$ **do**
    `# Najdeme otce u vrcholu v`
4 **if** v *je kořen* **then**
5 Vytvořit nový kořen u s jediným synem v
6 **else**
7 $u \leftarrow$ otec v
    `# Rozdělíme vrchol v na v and v'`
8 Vytvořit nového syna v' utci u a umístit jej vpravo vedle v
9 Přesunout nejpravějších $\lfloor (b+1)/2 \rfloor$ synů vrcholu v do v'
10 Přesunout nejpravějších $\lfloor (b+1)/2 \rfloor - 1$ klíčů vrcholu v do v'
11 Přesunout poslední klíč vrcholu v do u
12 $v \leftarrow u$

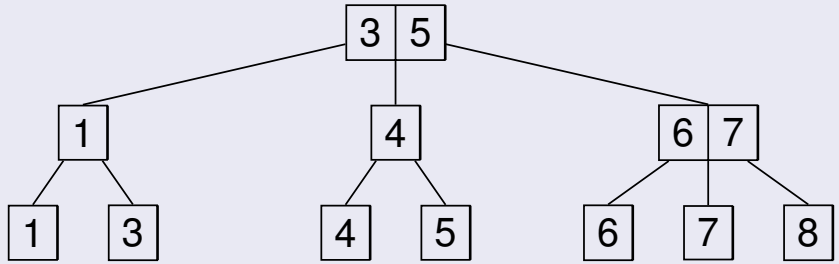
Časová složitost

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

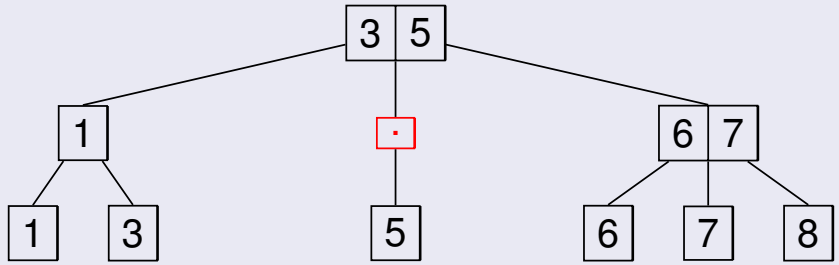
Musíme ještě dokázat, že po provedení všech operací doopravdy dostaneme (a,b)-strom. Ověříme, že rozdělené vrcholy mají alespoň a synů (ostatní požadavky jsou triviální). Rozdělovaný vrchol má na počátku právě $b+1$ synů a počet synů po rozdělení je $\lfloor \frac{b+1}{2} \rfloor$ a $\lceil \frac{b+1}{2}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

(a,b)-strom: Operace Delete

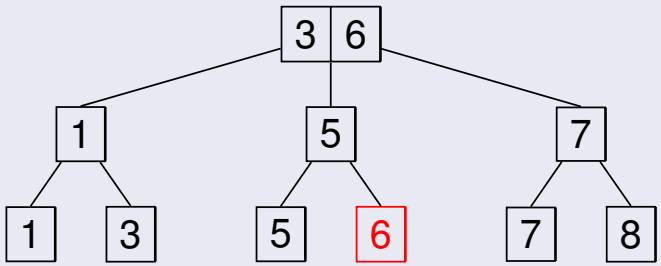
Smažte prvek s klíčem 4 z následujícího (2,4)-stromu

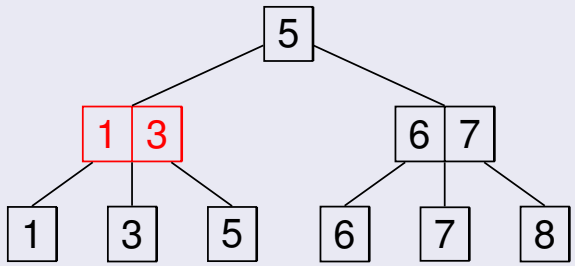


Nalezneme a smažeme list



Přesuneme jedno syna od bratra nebo spojíme vrchol s bratrem





(a,b)-strom: Operace Delete

Algoritmus

```
1  Najít list l obsahující prvek s daným klíčem
2  v ← otec l
3  Smazat l
4  while deg(v) < a & v není kořen do
5      u ← sousední bratr v
6      if deg(u) > a then
7          Přesunout správného syna u pod v ①
8      else
9          Přesunout všechny syny u pod v ②
10         Smazat u
11         if v nemá žádného bratra then
12             Smazat kořen (otec v) a nastavit v jako kořen
13         else
14             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

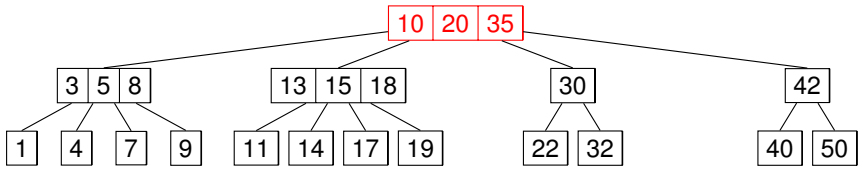
Preventivně rozdělit každý vrchol na cestě od kořene k hledanému listu s b syny na dva vrcholu.

Operace Delete

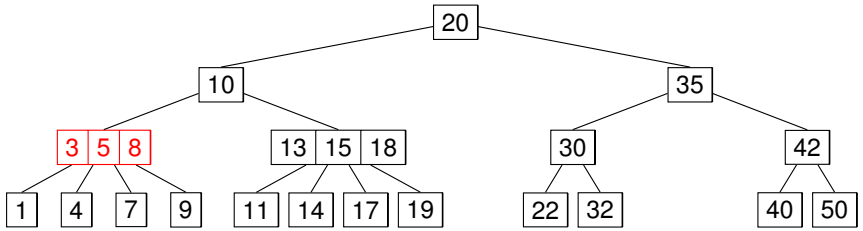
Preventivně sloučit každý vrchol na cestě od kořene k hledanému listu s a syny s bratrem nebo přesunout synovce.

(a,b)-strom: Paralelní přístup: Příklad

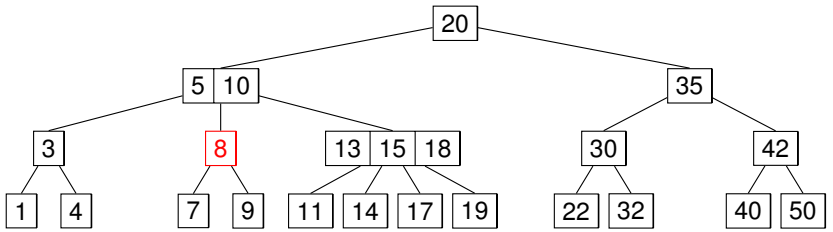
- Vložte prvek s klíčem 6 do následujícího (2,4)-stromu



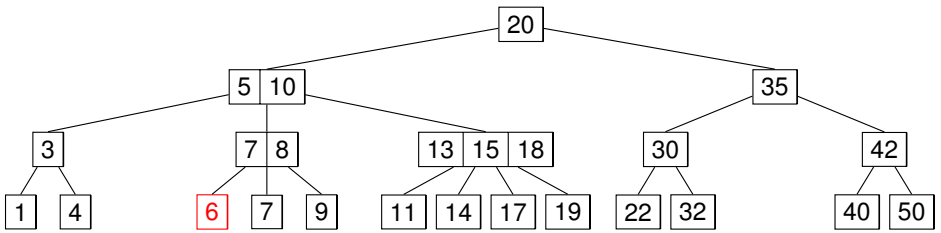
- Nejprve rozdělíme kořen



- 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 [9])

Cíl

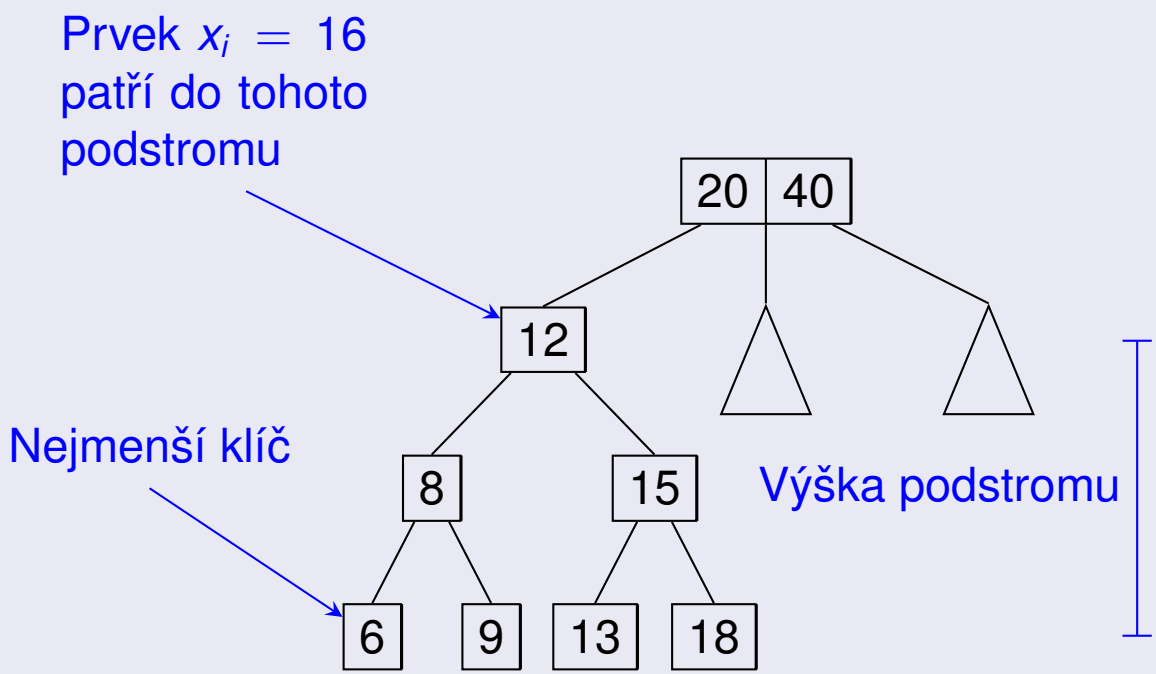
Setřídit „skoro“ setříděné pole

Modifikace (a,b)-stromu

Máme uložený ukazatel na vrchol s nejmenším klíčem

Příklad: Vložte klíč s hodnotou $x_i = 16$

- Začmene 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 $x_1, x_2, \ldots, x_n$

1 $T \leftarrow$ prázdný (a,b)-strom
2 **for** $i \leftarrow n$ **to** *1* `# Prvky procházíme od konce`
3 **do**
    `# Najdeme podstrom, do kterého vložíme` x_i
4 $v \leftarrow$ list s nejmenším klíčem
5 **while** *v není kořen a* x_i *je větší než nejmenší klíč v otci vrcholu v* **do**
6 $v \leftarrow$ otec v
7 Vložíme x_i do podstromu vrcholu v

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

A-sort: Složitost (Brown, Tarjan [2], Mehlhorn [15])

Nerovnost mezi aritmetickým a geometrickým průměrem

Jestliže $a_1, \ldots, a_n$ 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
   $\sum_i \log f_i = \log \prod_i f_i = n \log \sqrt[n]{\prod_i f_i} \leq n \log \frac{\sum_i f_i}{n} = n \log \frac{F}{n}$.
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 $\mathcal{O}(n \log \log n)$

(a,b)-strom: Závěr

Počet modifikovaných vrcholů při operací Insert a Delete (Cvičení 4.8) [11]

- 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+, FAT
- Databáze

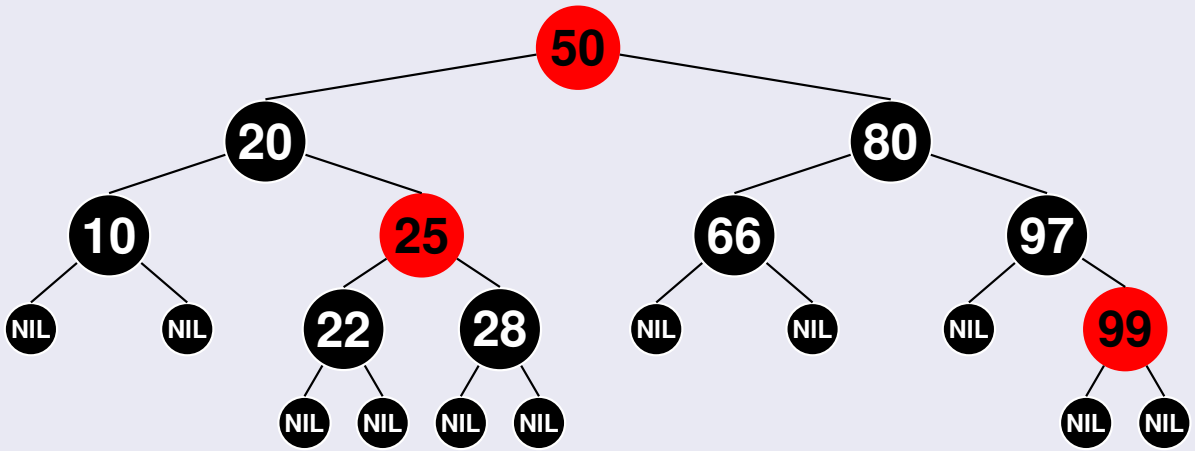
Obsah

Červeno-černé stromy (Guibas, Sedgewick [10])

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

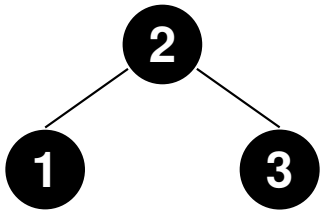
Příklad

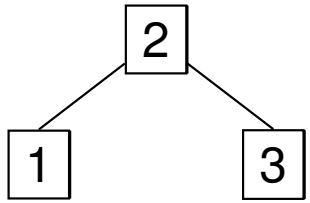


1. Nepovinná podmínka, která jen zjednodušuje operace. V příkladu uvažujeme, že listy jsou reprezentovány NIL/NULL ukazately, a tedy imaginární vrcholy bez prvků.
   Někdy se též vyžaduje, aby kořen byl černý, ale tato podmínka není nutná, protože kořen můžeme vždy přebarvit na černo bez porušení ostatních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

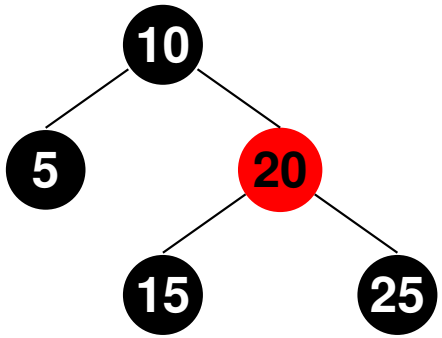
Červeno-černé stromy: Ekvivalence s (2,4)-stromy

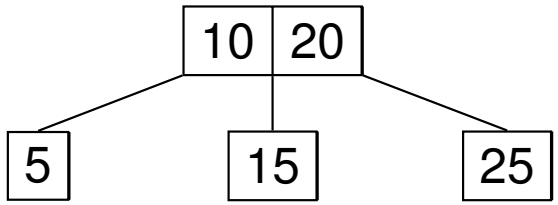
- Vrchol bez červených synů



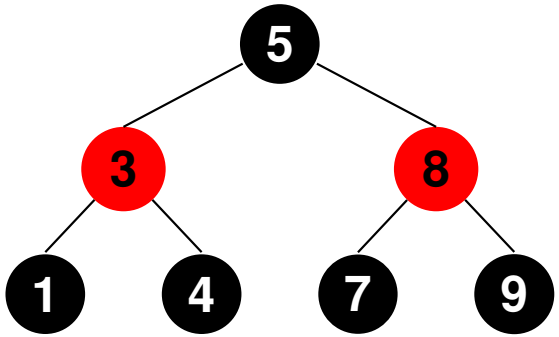


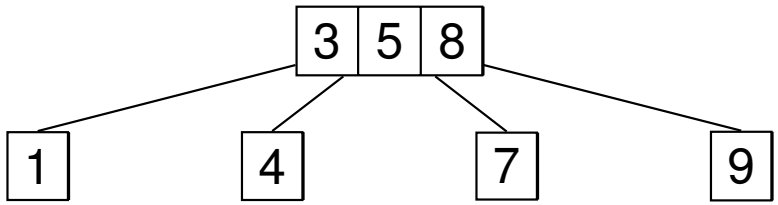
- Vrchol s jedním čevneným synem ①





- Vrchol s dvěma červenými syny



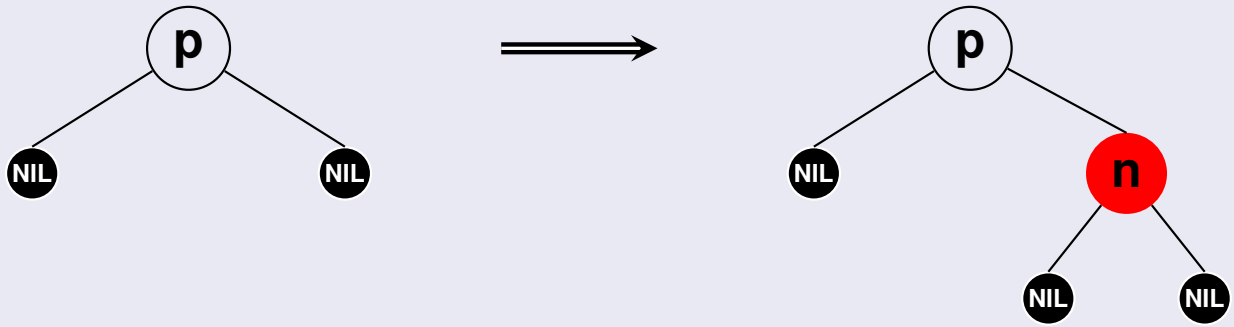


1. Převod mezi červeno-černými stromy a (2,4)-stromy není jednoznačný, protože vrchol (2,4)-stromu se třemi syny a prvky $x < y$ lze převést na černý vrchol červeno-černého stromu s prvkem x a pravým červeným synem y nebo s prvkem y a levým červeným synem x.

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

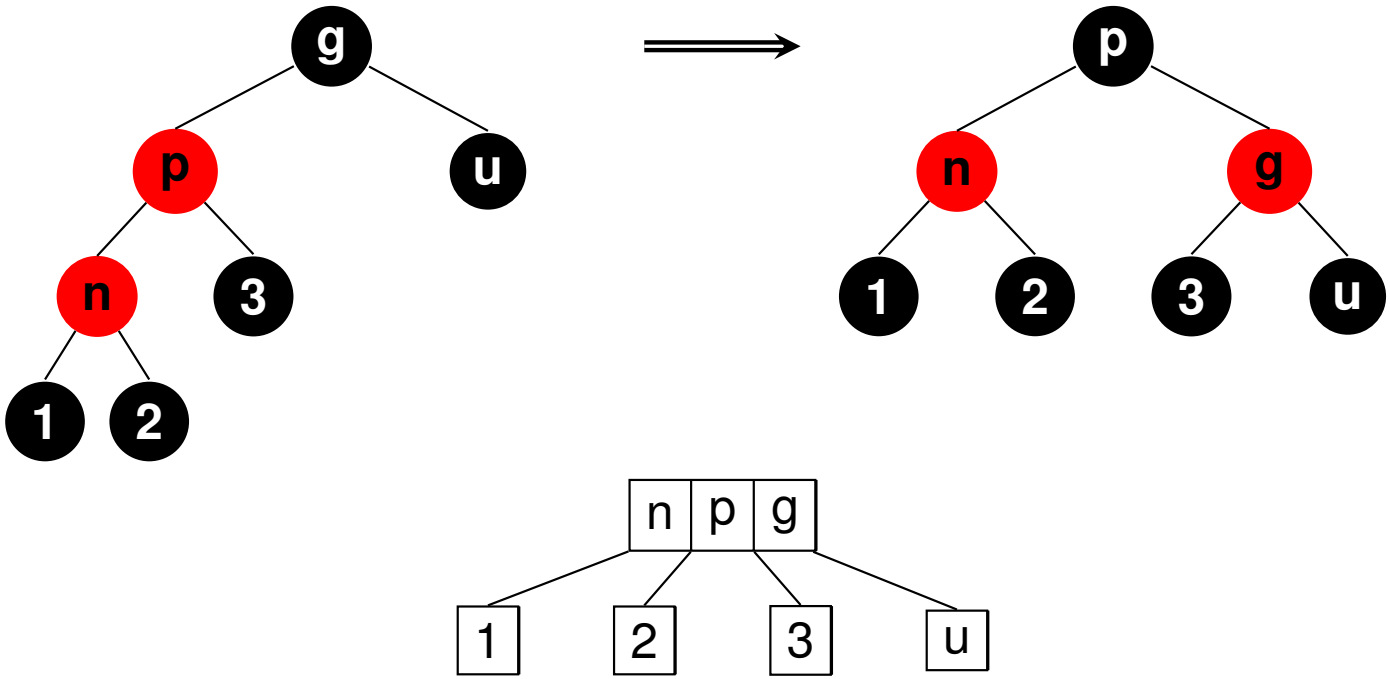
- Vrchol *n* a jeho otec *p* jsou červené vrcholy a toto je jediná porušená podmínka
- Děda *g* vrcholu *n* je černý

Musíme uvažovat tyto případy:

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

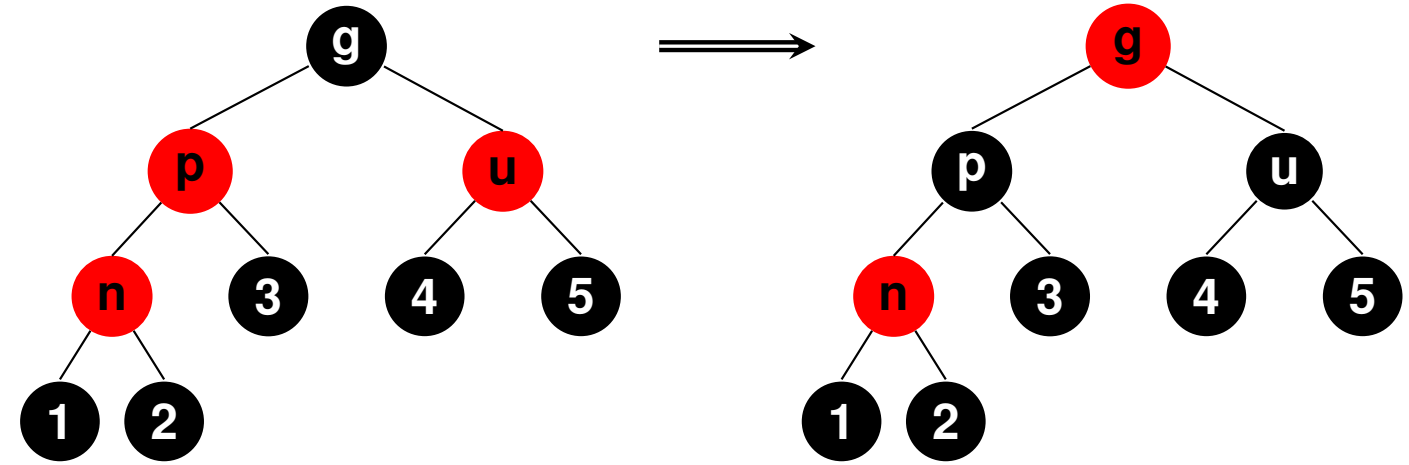
1. S využitím symetrií lze počet případů snížit.

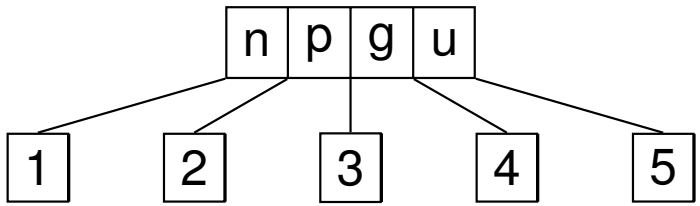
Červeno-černé stromy: Operace Insert, strýc je černý

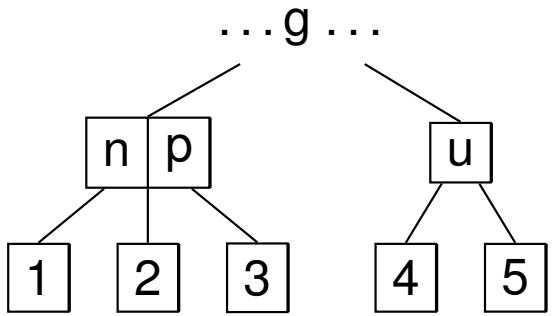


Pořadí prvků v (2,4)-stromu a výsledný červeno-černý strom závisí na tom, zda vrchol *n* je pravým nebo levým synem *p* a zda vrchol *p* je pravým nebo levým synem *g*.

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

- Associativní 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.

Cvičení:

4.1. Jak v (a,b)-stromu (obecném vyhledávacím stromu) najít prvek s nejmenším/největším klíčem? Jak najít prvek s nejbližším klíčem (tj. nejmenší větší klíč nebo největší menší klíč k danému klíči)?

4.2. Upravte (a,b)-strom (obecný vyhledávací strom) tak, aby bylo možné efektivně najít k-tý nejmenší/největší prvek pro dané přirozené k. Dokážete v takto modifikovaném stromu určit pořadí daného prvku?

4.3. V klasické implementaci (a,b)-stromu má každý vrchol alokováno pole velikosti b pro klíče a ukazatele na syny. Toto pole může být až z poloviny nevyužito, což může být významným plýtváním paměti. Pokuste se upravit operace Insert a Delete tak, aby bylo zaručeno efektivnější využití tohoto pole.

4.4. Přesně popište implementaci (a,b)-stromu a zamykání jednotlivých vrcholů v operacích Insert a Delete tak, aby správně fungoval paralelní přístup.

4.5. Jak vytvořit (a,b)-strom ze setříděného pole (co nejrychleji)?

4.6. Existuje asymptoticky rychlejší postup vytvoření (a,b)-stromu z nesetříděného pole než vložení všech prvků?

4.7. Najděte asymptoticky nejrychlejší způsob sjednocení dvou (a,b)-stromů, jestliže jeden (a,b)-strom nemá všechny prvky menší než je nejmenší prvek druhého.

4.8. Dokažte, že v (a,b)-stromu pro $b \geq 2a$ je počet modifikovaných vrcholů $\mathcal{O}(k + l + \log n)$ při l operacích Insert a k operacích Delete. Pro zjednodušení můžete uvažovat (2,4)-strom. Dále ukažte, že podmínka $b \geq 2a$ je nutná.

4.9. Vymyslete operace Insert a Delete v (2,5)-stromu pro paralelní aplikace tak, aby amortizovaný počet modifikovaných vrcholů byl $\mathcal{O}(1)$.

4.10. Vymyslete pravidla pro operaci Delete v červeno-černých stromech pomocí ekvivalence s (2,4)-stromy.

4.11. Upravte operace Insert a Delete v červeno-černém stromu tak, že tyto operace projdou strom jen jednou od kořene k listu (bez zpětného průchodu od listu ke kořeni). Lze tuto modifikaci použít k paralelizaci?

Obsah

Halda

Základní operace

- Insert
- FindMin
- DeleteMin
- DecreaseKey: Snížit hodnotu klíče v daném vrcholu

Haldový invariant

Klíč v každém vrcholu větší nebo roven klíči v otci.

Aplikace

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

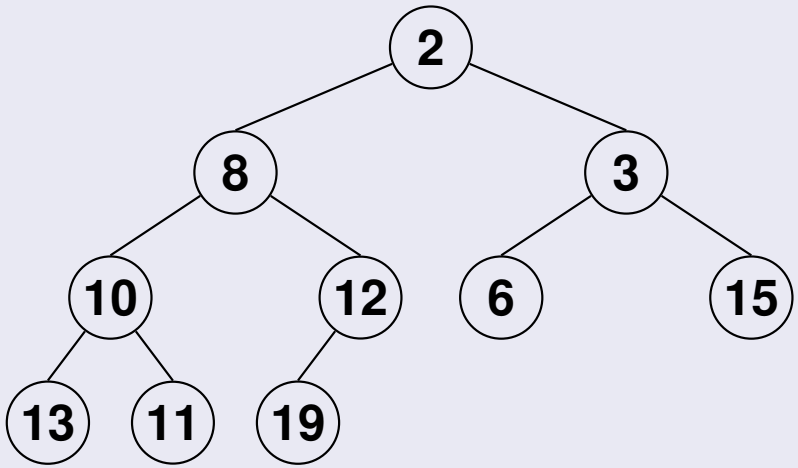
Obsah

d-regulární halda (Johnson [12])

Vlastnosti

- Každý vrchol má nejvýše d synů
- Všechny vrstvy kromě poslední jsou úplně zaplněné
- Poslední hladina je zaplněná zleva

Příklad 2-regulární (binární) haldy



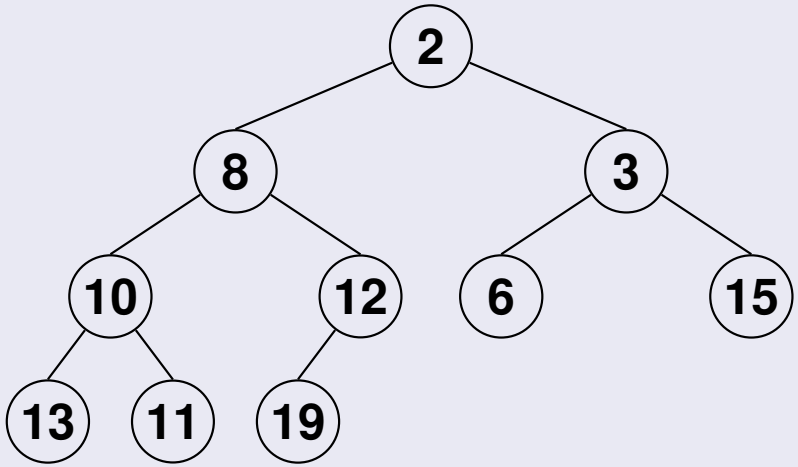
Cvičení 5.1

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 $h \leq \log_d n$. Tudíš výška d-regulární hadly s n prvky je nejvýše $1 + \log_d n$. Najděte formuli udávající přesnou výšku d-regulární haldy.

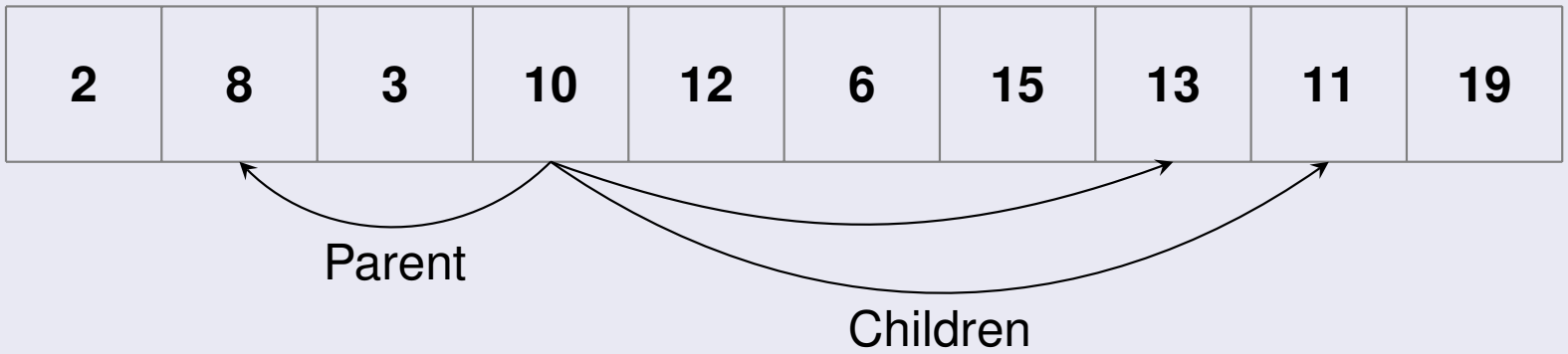
d-regulární halda: Representace

Binární halda uložená ve stromu



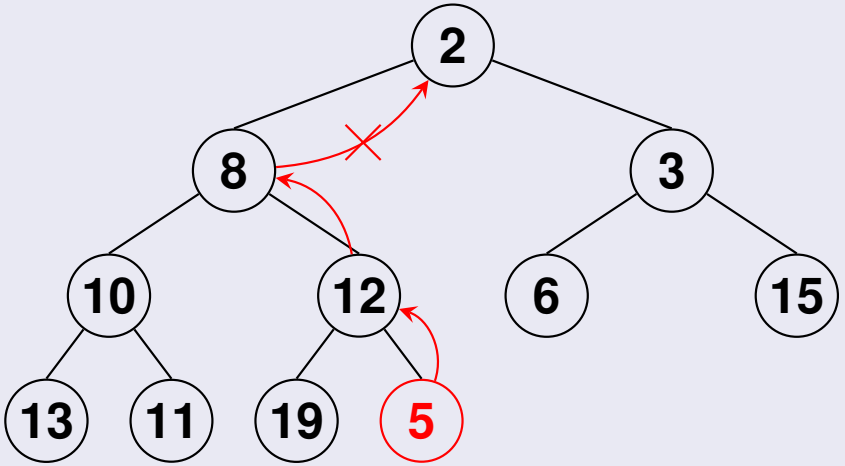
Binární halda uložená v poli

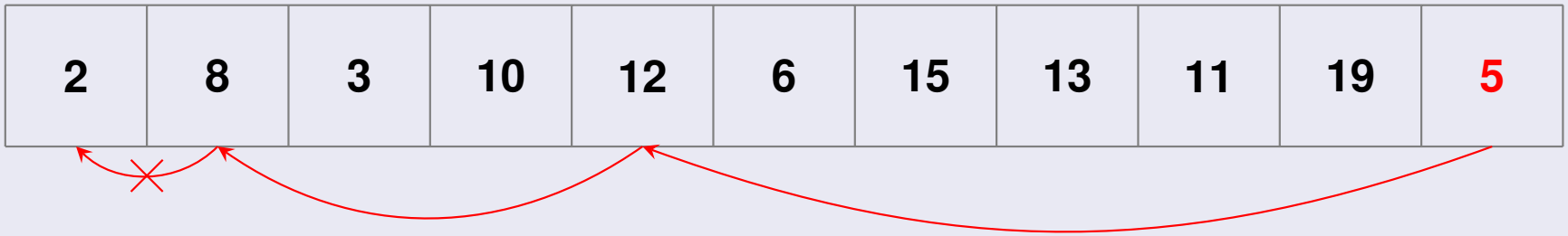
A vrchol na pozici i má otce na pozici $\lfloor (i-1)/2 \rfloor$ a syny na pozici $2i+1$ a $2i+2$:



Cvičení 5.2: Určete pozice otce a synů pro obecnou d-regulární haldu

Příklad: Vložme prvek s klíčem 5





Insert: Algoritmus

Input: Nový prvek s klíčem x

1 $v \leftarrow$ první volný blok v poli
2 Nový prvek uložíme na pozici v
3 **while** *v není kořen a otec p vrcholu v má klíč větší než x* **do**
4 Prohodíme prvky na pozicích v a p
5 $v \leftarrow p$

Operace DecreaseKey

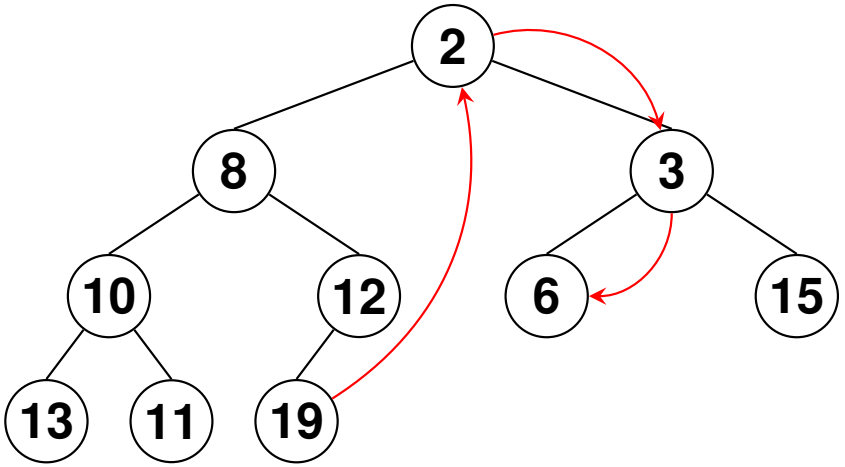
Snižíme hodnotu klíče a pokračujeme podobně jako při operaci Insert

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 hodnost na vstupu.

Operace DeleteMin



Algoritmus

1 Přesuneme poslední prvek do kořene v
2 **while** *Některý se synů vrcholu v má klíč menší než v* **do**
3 $u \leftarrow$ syn vrcholu v s menším klíčem
4 Prohodíme prvky ve vrcholech u a v
5 $v \leftarrow u$

Složitost

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

d-regulární halda: Operace Build

Cíl

Vytvořit haldu z daného pole prvků

Algoritmus

```
1 for r ← poslední pozice to první pozice v poli do
      # Zpracujeme vrchol r podobně jako při operaci Delete
2     v ← r
3     while Některý se synů vrcholu v má klíč menší než v do
4         u ← syn vrcholu v s menším klíčem
5         Prohodíme prvky ve vrcholech u a v
6         v ← u
```

Korektnost

Podstromy všech zpracovaných vrcholů tvoří haldu

Lemma (Cvičení 5.5)

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

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

Binomiální strom

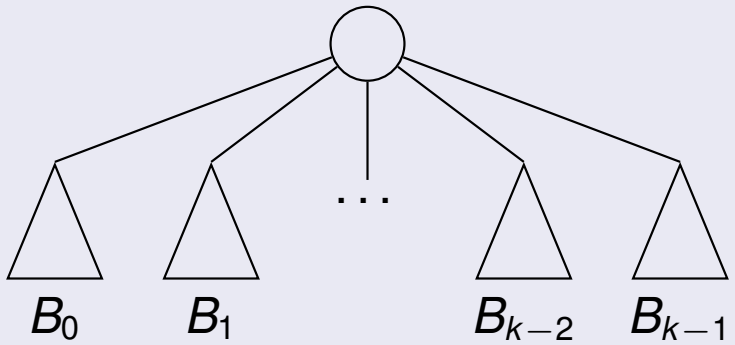
Definice

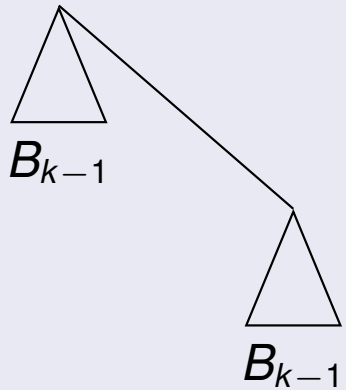
- Binomiální strom B_0 řádu 0 je jeden vrchol
- Binomiální strom B_k řádu $k \geq 1$ má kořen, jehož synové jsou kořeny binomiálních stromů řádu $0, 1, \ldots, k-1$.

Alternativně

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

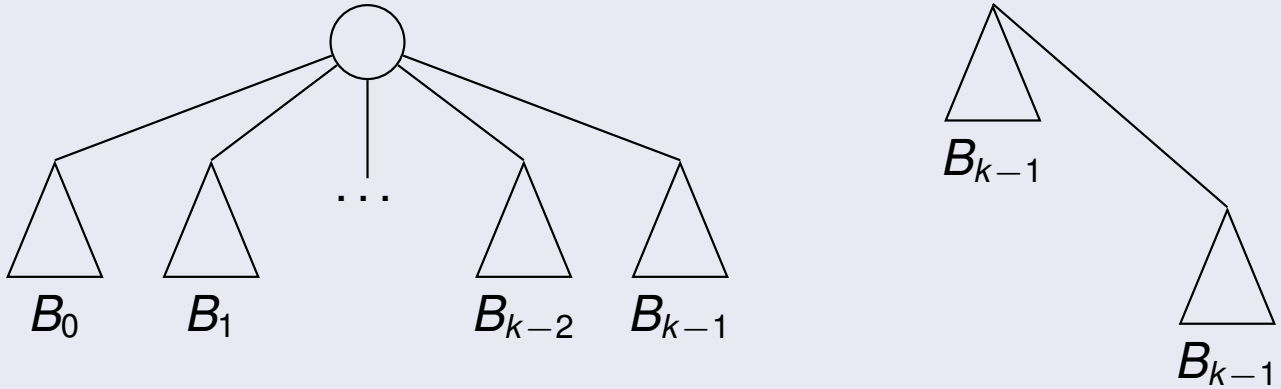
Rekurzivní definice binomiálního stromu



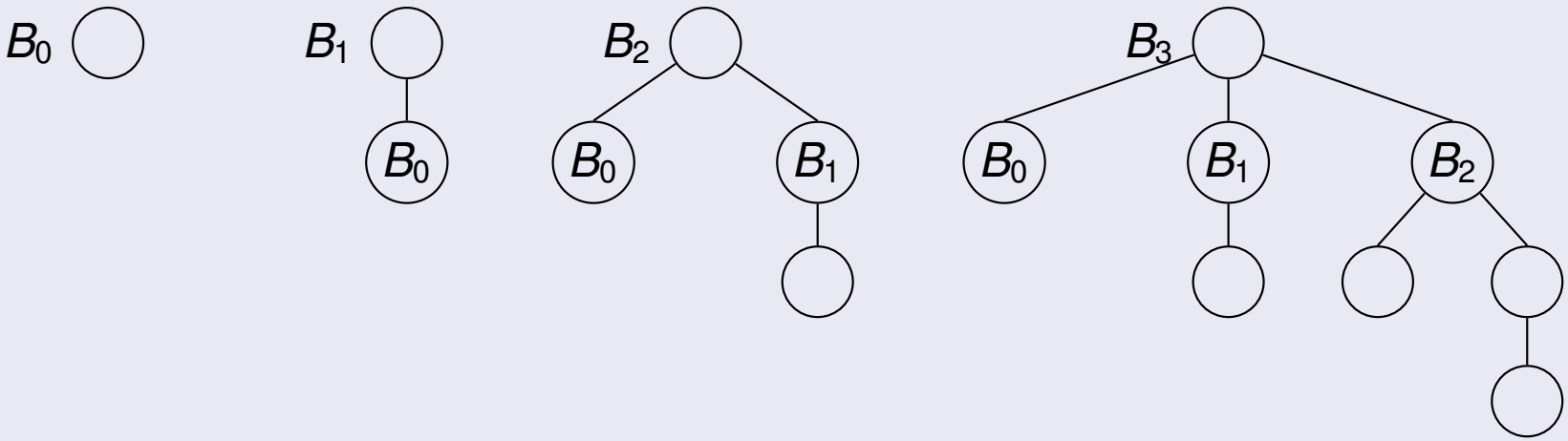


Binomiální strom: Příklad

Rekurzivní definice binomiálního stromu

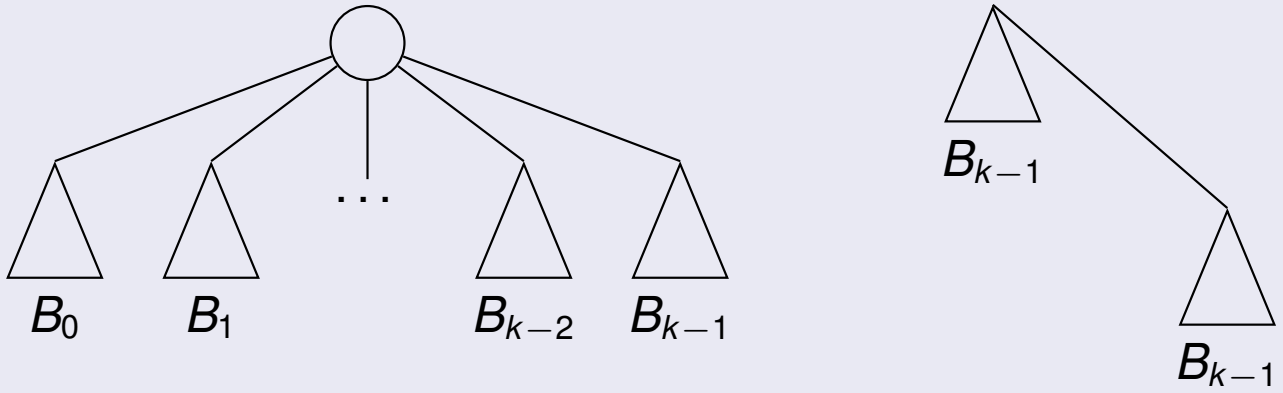


Binomiální stromy řádu 0, 1, 2 a 3



Binomiální strom: Vlastnosti

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

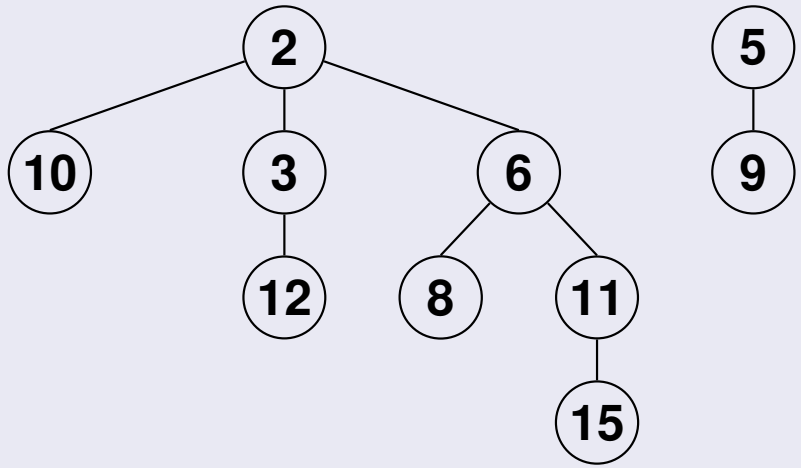
Pozorování

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

Binární číslo n	$=$	1	0	0	1	1	0	1	0
		↓			↓	↓		↓	
Binomiální halda obsahuje:		B_7			B_4	B_3		B_1	

Příklad pro 1010_2 prvků



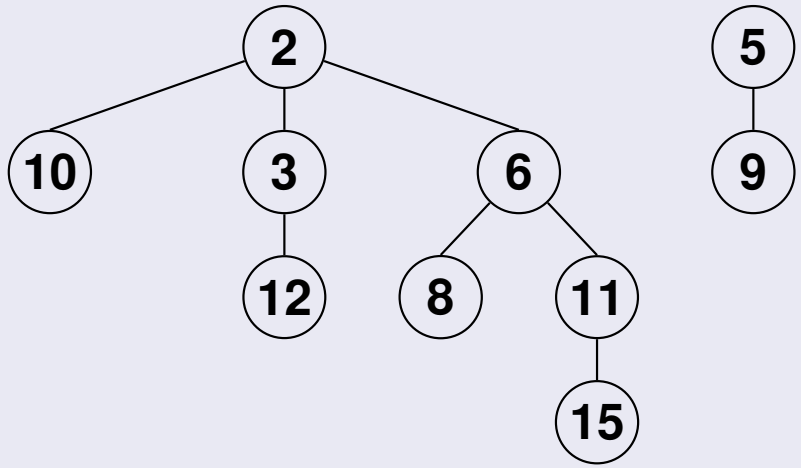
Binomiální halda (Vuillemin [26])

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 klíč 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

Binární číslo n	$=$	1	0	0	1	1	0	1	0
		↓			↓	↓		↓	
Binomiální halda obsahuje:		B_7			B_4	B_3		B_1	

Binomiální halda: Representace

Struktuta pro vrchol binomiálního stromu obsahuje

- prvek (klíč a hodnota),
- 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 minimálním prvek.

Operace FindMin

Triviálně v čase $\mathcal{O}(1)$

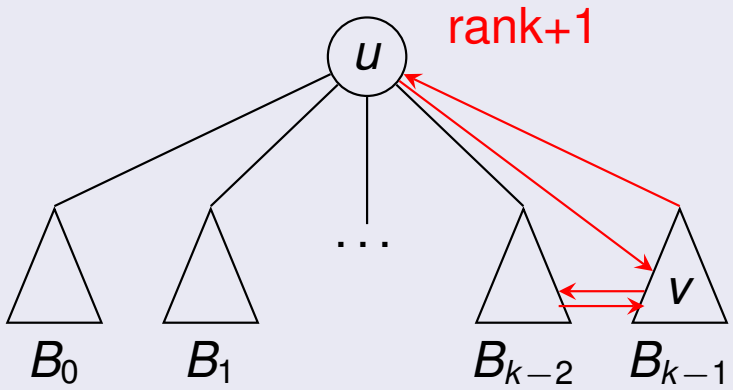
Operace DecreaseKey

Stejně jako v regulární haldě.

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

Binomiální halda: Spojení dvou binomiálních hald

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



Join

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 prvek, 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 Join

- 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, jak je spojujeme
- Použijeme pole indexované řádem stromu k vyhledávání stromů stejného řádu

Algoritmus

1 Inicializujeme pole velikosti $\lceil \log_2(n+1) \rceil$ ukazatelem NIL
2 **for** *pro každý strom h v líné binomiální haldě* **do**
3 $o \leftarrow$ řád stromu h
4 **while** *pole[o]* $\neq$ *NIL* **do**
5 $h \leftarrow$ spojení stromů h a pole[o]
6 pole[o] $\leftarrow$ NIL
7 $o \leftarrow o + 1$
8 pole[o] $\leftarrow h$
9 Pole stromů převedeme na spojový seznam, čímž vytvoříme poctivou binomiální haldu

Cvičení 5.6

Amortizovaná složitost operace DecreaseKey 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
DecreaseKey	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
DeleteMin	$\log n$	$\log n$	$\log n$	n	$\log n$

Cvičení 5.7

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

Další cíl

Zrychlit operaci DecreaseKey

Postup

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

Obsah

Fibonacciho halda (Fredman, Tarjan [7])

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 prvek ⑤
- Smíme smazat kořen stromu ⑥

Operace stejné jako v líné binomiální haldě

Insert, FindMin, DeleteMin

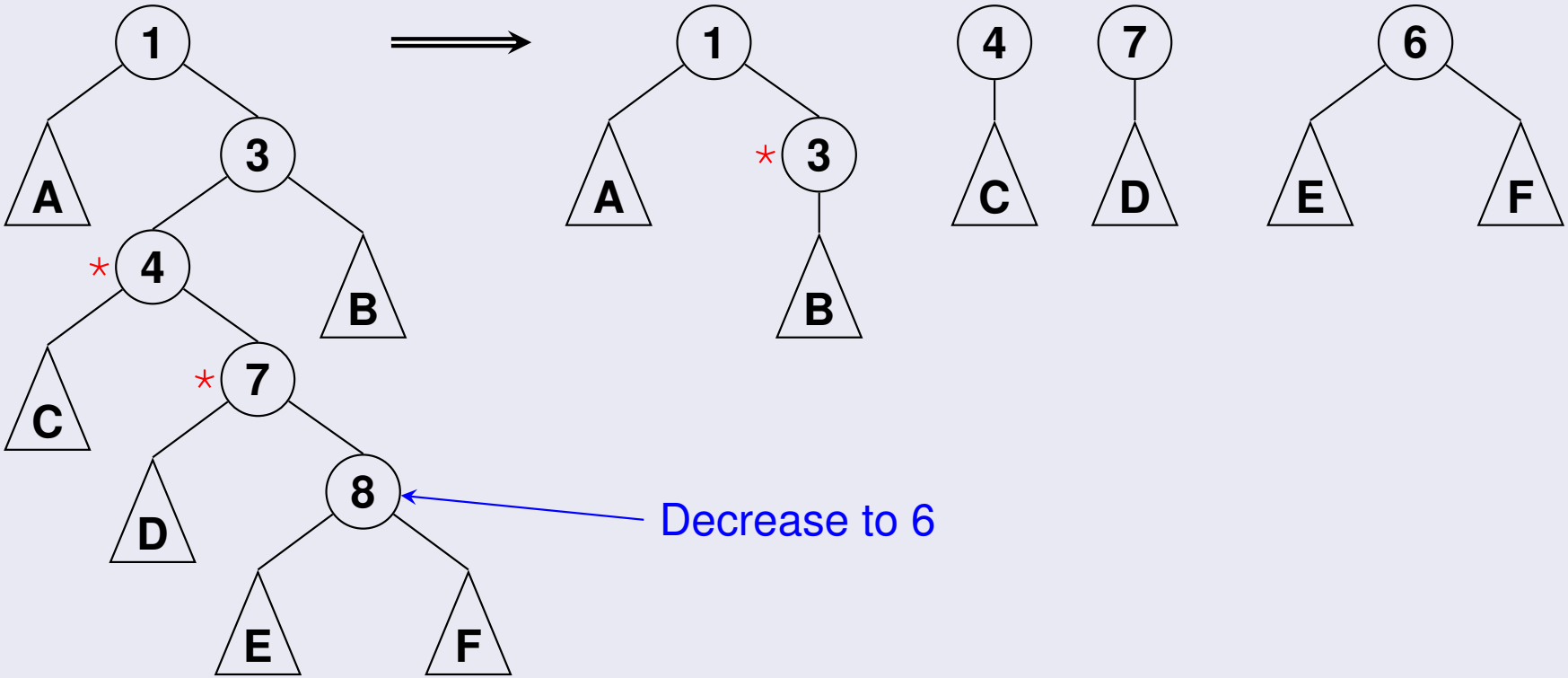
1. Doposud probírané datové struktury mají jasně definovanou strukturu a operace jsou navrženy tak, aby tuto strukturu zachovávaly. Fibonacciho halda je definovaná povolenými operacemi a vlastnosti se odvozují z operací.
2. Podobně jako binomiální halda, ale stromy nemusí být izomorfní s binomiálními stromy.
3. Podobně jako binomiální halda, ale vztahy pro počet vrcholů nebo výšku neplatí. Tvrzení, že podstromy vrcholu řádu k mají řád $0, 1, \ldots, k-1$ 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.

Fibonacciho halda: Operace DecreaseKey

Idea

- Danému vrcholu snížíme hodnotu klíče 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 DecreaseKey

Algoritmus

Input: Vrchol u a nový klíč k

Snížíme klíč vrcholu u
if *u je kořen nebo otec p vrcholu u má klíč nejvýše k* **then**
    return `# Haldový invariant je zachovaný`
$p \leftarrow$ otec vrcholu u
Odznačit vrchol u
Odpojit vrcholu 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 u
    Odznačit vrchol u
    Odpojit vrcholu 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
- Join: 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

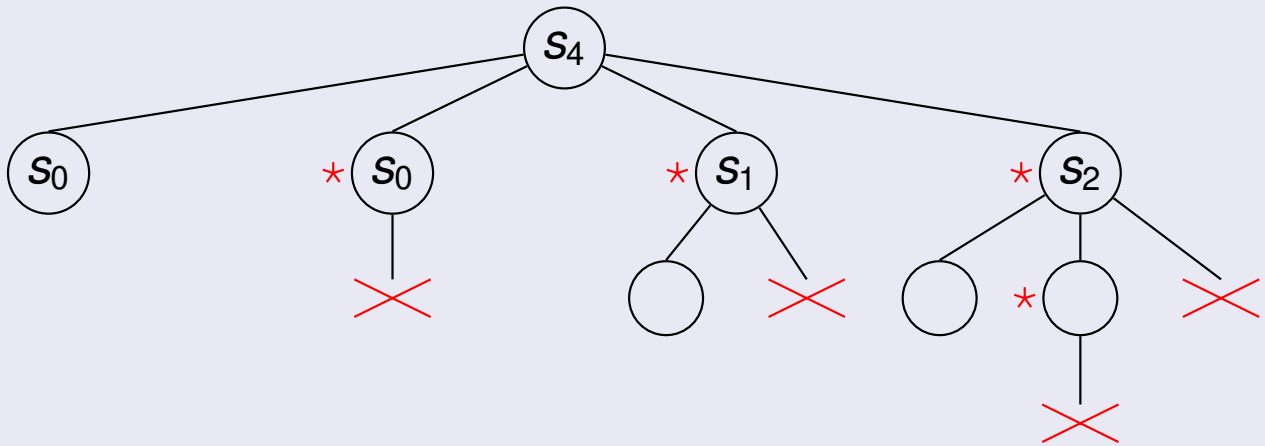
Pro každý vrchol p a jeho i-tého syna s platí, že s má alespoň

- $i - 2$ 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 5.9)

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

Důsledek

Strom řádu k má alespoň $s_k \geq F_{k+2} \geq \left(\frac{1+\sqrt{5}}{2}\right)^{k+2}$ 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. V obecně můžeme mít Fibonacciho halda až n stromů, ale po konsolidaci (součást operace DeleteMin) mají každé dva stromu 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-key: $\mathcal{O}(n)$ (Cvičení 5.10)
- Operace Delete-min: $\mathcal{O}(n)$

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 DecreaseKey

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 $\Phi' - \Phi = 1 - 2$
- Amortizovaná složitost: $\emptyset$

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 DecreaseKey

$\mathcal{O}(1)$

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: $\emptyset$

Ostatní instrukce

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

Amortizovaná složitost operace DeleteMin

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

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
DecreaseKey	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	n	1
DeleteMin	$\log n$	$\log n$	$\log n$	n	$\log n$	n	$\log n$

Cvičení:

5.1. Určete přesně výšku d-regulární haldy s n vrcholy.

5.2. d-regulární haldu podobně jako binární je možné uložit v poli. Najděte vzorce pro výpočet pozice otce a synů pro daný prvek v poli.

5.3. Jak byste zvýšili hodnotu klíče v d-regulární haldě?

5.4. Jak byste smazali prvek v daném vrcholu v d-regulární haldě?

5.5. Pro $d > 1$ dokažte $\sum_{h=0}^{\infty} \frac{h}{d^h} = \frac{d}{(d-1)^2}$.

5.6. Dokažte, že amortizovaná složitost operace DecreaseKey v líné binomiální haldě je $\mathcal{O}(\log n)$. Dále určete celkovou složitost k_I operací Insert, k_M operací DeleteMin a k_D operací DecreseKey.

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

5.8. Jaký je maximální možný počet vrcholů ve stromu řádu k ve Fibonacciho haldě?

5.9. Dokažte, že Fibonacciho čísla splňují $\sum_{i=1}^{k} F_i = F_{k+2} - 1$ a $F_k \geq \left(\frac{1+\sqrt{5}}{2}\right)^k$.

5.10. Jaká je maximální možná výška stromu ve Fibonacciho haldě s n prvky? Může se stát, aby složitost operace DecreaseKey byla $\Omega(n)$?

5.11. Proč Fibonacciho halda povoluje odstranit nejvýše jednoho syna? Uvažujme neoznačující Fibonacciho haldu, která při operaci DecreaseKey pouze odebere daný vrchol od otce, ale neoznačí otce a ani nepokračuje k předkům. Která tvrzení o Fibonacciho haldě přestanou pro neoznačující Fibonacciho haldu platit? Dojde ke zhoršení časových složitostí studovaných operací?

5.12. Změnili by se časové složitost operací, kdybychom ve Fibonacciho haldě dovolili smazat dva syny (nebo libovolný pevný počet synů)?

5.13. Jakou časovou složitost má Dijkstrův algoritmus při použití probíraných hald? Při jakém poměru počtu hran a vrcholů grafu dokážete získat lineární časovou složitost Dijkstrova algoritmu?

Obsah

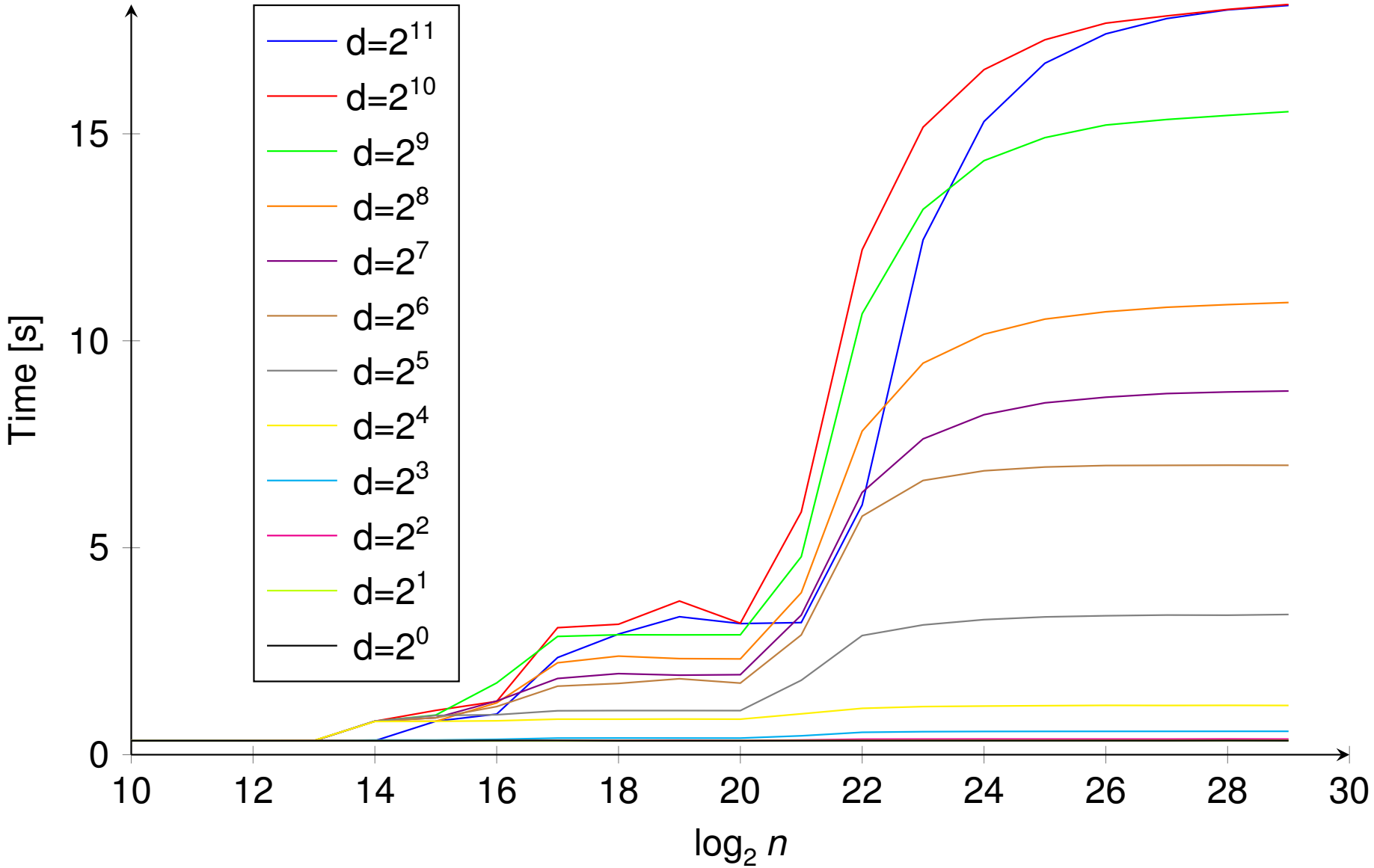
Příklad velikostí a rychlostí různých typů pamětí

	size	speed
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
Internet	∞	10 MB/s

Triviální program

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

Paměťová hierarchie: Triviální program



Cache-oblivious (Frigo, Leiserson, Prokop, Ramachandran, 1999 [8])

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

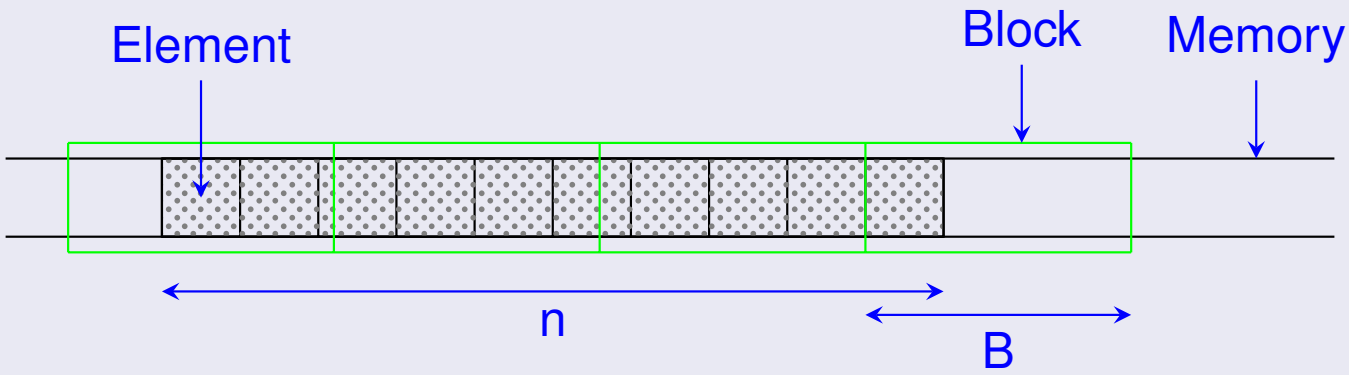
Algoritmus musí efektivně fungovat bez znalostí hodnot M a B. Důsledky:

- 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)

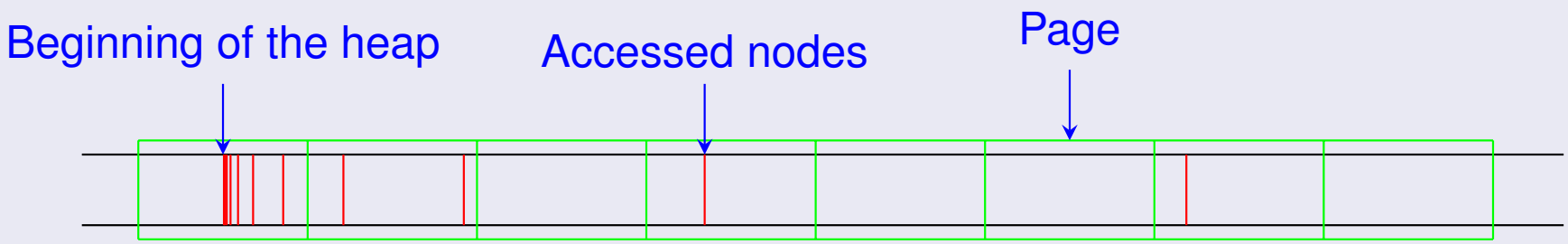


- Minimální možný počet přenesených bloků je $\lceil n/B \rceil$.
- Skutečný počet přenesených bloků je nejvýše $\lceil n/B \rceil + 1$.
- Předpokládáme, že máme k dispozici $\mathcal{O}(1)$ registrů k uložení iterátoru a maxima.

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

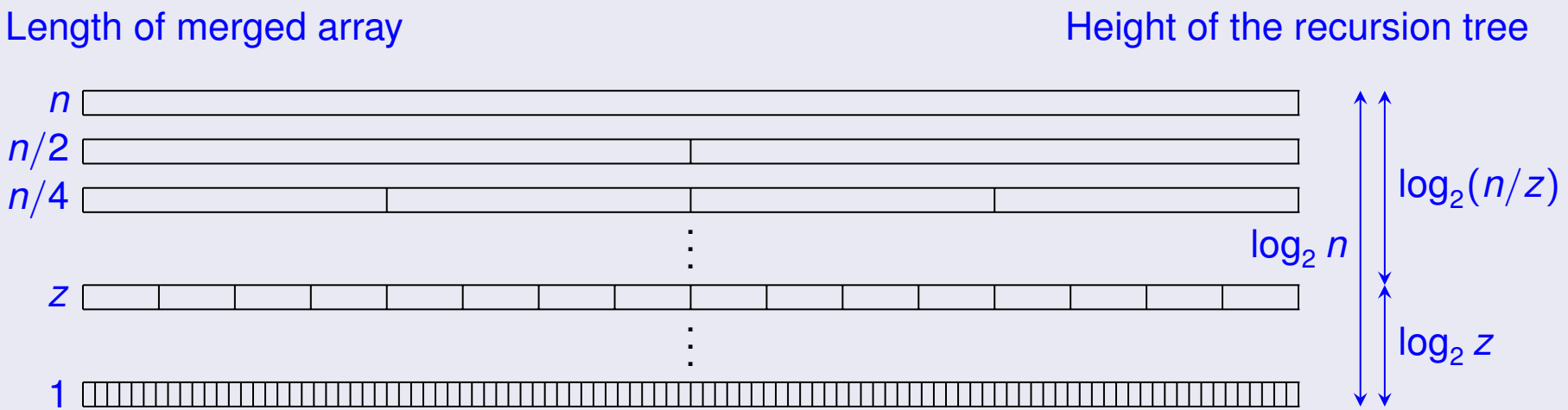
1. Přesněji $\Theta(\max\{1, \log n - \log B\})$. Dále předpokládáme, že $n \geq B$.
2. Pro jednoduchost uvažujeme neúspěšné vyhledávání.

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 poloviního 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 $k \times k$

```
1 for i ← 1 to k do
2     for j ← i + 1 to k do
3         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

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
1 for (i = 0; i < k; i+ = z) do
2     for (j = i; j < k; j+ = z) do
          # Transponujeme submatici začínající na pozici (i, j)
3         for (ii = i; ii < min(k, i + z); ii + +) do
4             for (jj = max(j, ii + 1); jj < min(k, j + z); jj + +) do
5                 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

```
1  Procedure transpose_on_diagonal(A)
2      if Matice A je malá then
3          Transponujeme matici A triviálním postupem
4      else
5          A11, A12, A21, A22 ← souřadnice submatic
6          transpose_on_diagonal(A11)
7          transpose_on_diagonal(A22)
8          transpose_and_swap(A12, A21)
9  Procedure transpose_and_swap(A, B)
10     if Matice A a B jsou malé then
11         Prohodíme a transponujeme matice A a B triviálním postupem
12     else
13         A11, A12, A21, A22, B11, B12, B21, B22 ← souřadnice submatic
14         transpose_and_swap(A11, B11)
15         transpose_and_swap(A12, B21)
16         transpose_and_swap(A21, B12)
17         transpose_and_swap(A22, B22)
```

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.
Počítame 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

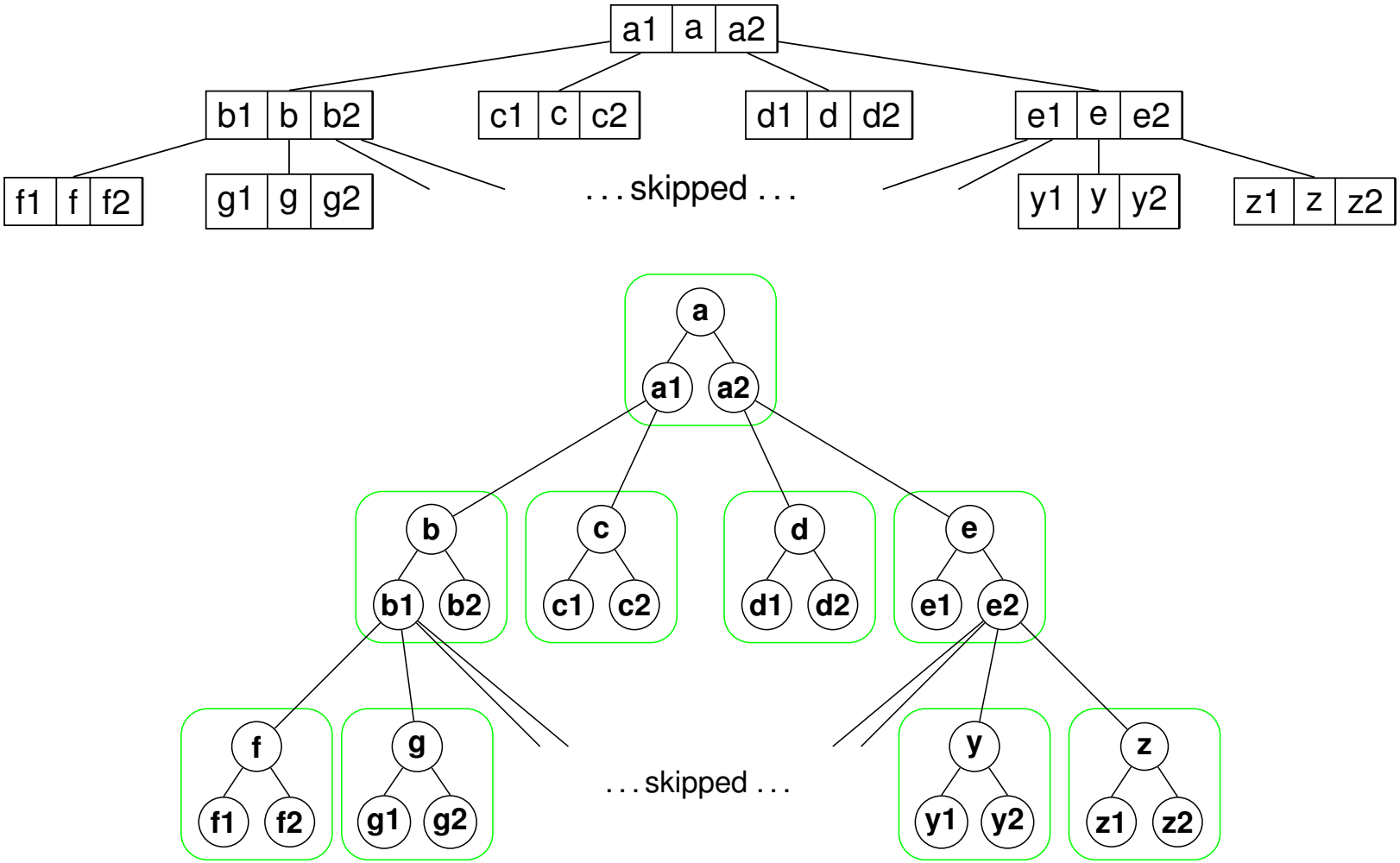
- Výška stromu je $\log_B(n) + \Theta(1)$ ①
- Jeden vrchol je uložen v nejvýše dvou blocích
- Počet načtených bloků je $\Theta(\log_B(n))$ ②
- Nevýhody: cache-aware a chtěli jsme binární strom

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

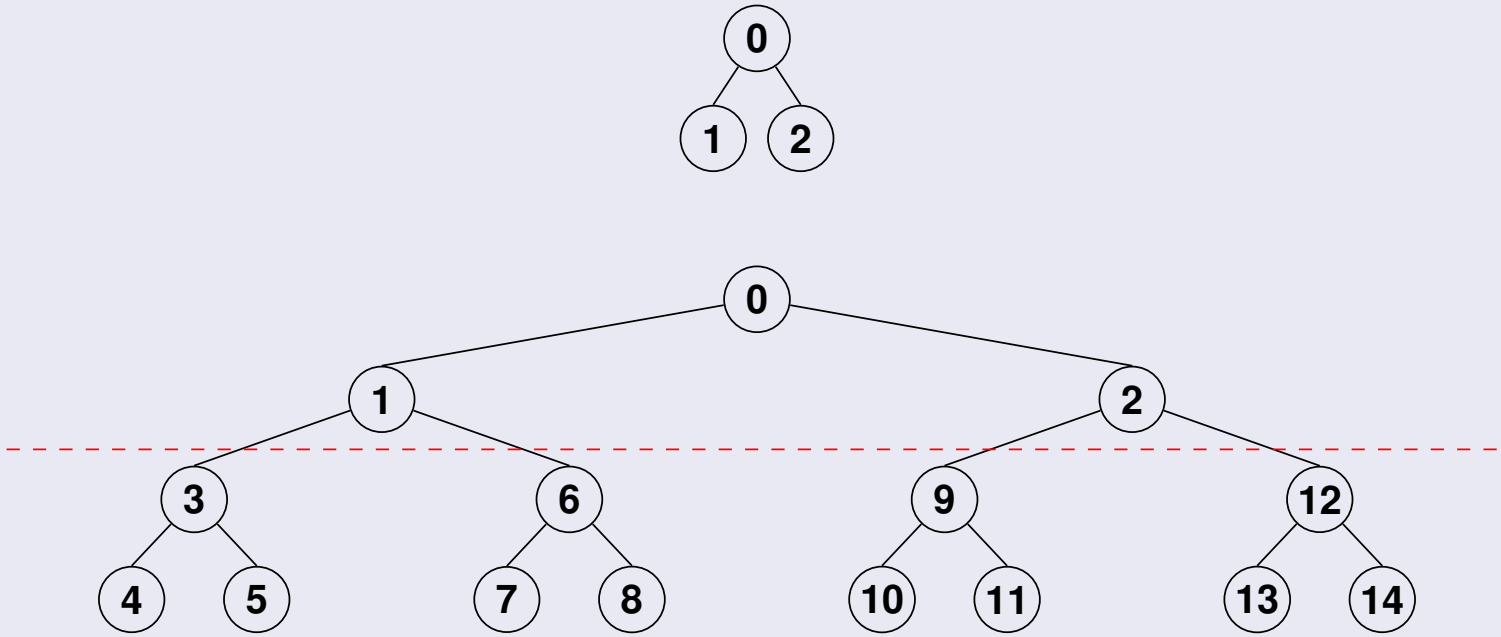


Cache-oblivious analýza: Reprezentace binárních stromů

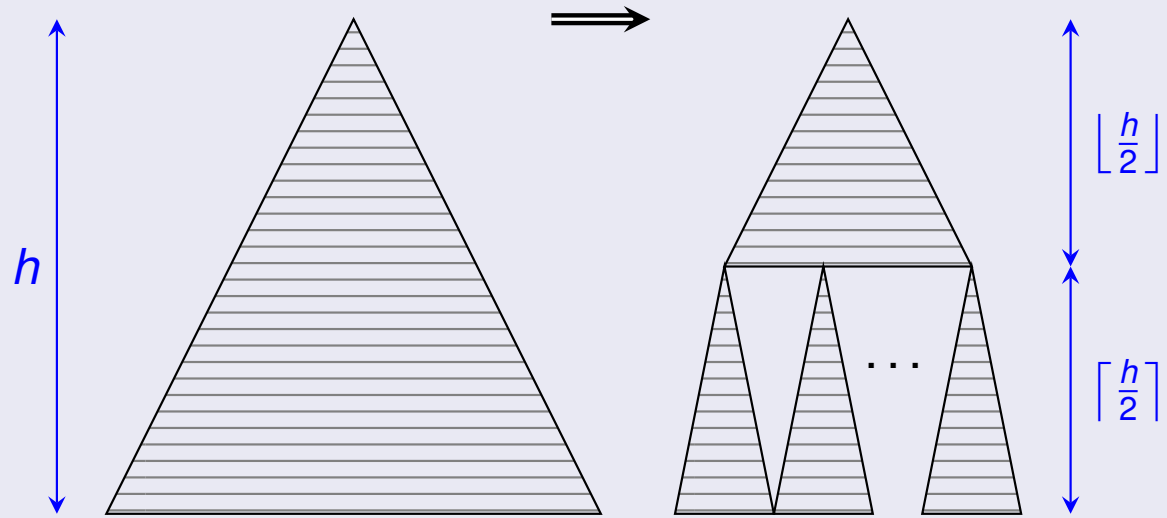
Rekurzivní „bottom-up“ konstrukce van Emde Boas rozložení

- 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



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 [24])

- 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 $s_1, \ldots, s_k$ 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 $F_{\mathrm{LRU}} \leq 2F_{\mathrm{OPT}} + P_{\mathrm{OPT}}$.

Cache-oblivious analýza: Srovnání OPT a LRU strategií

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

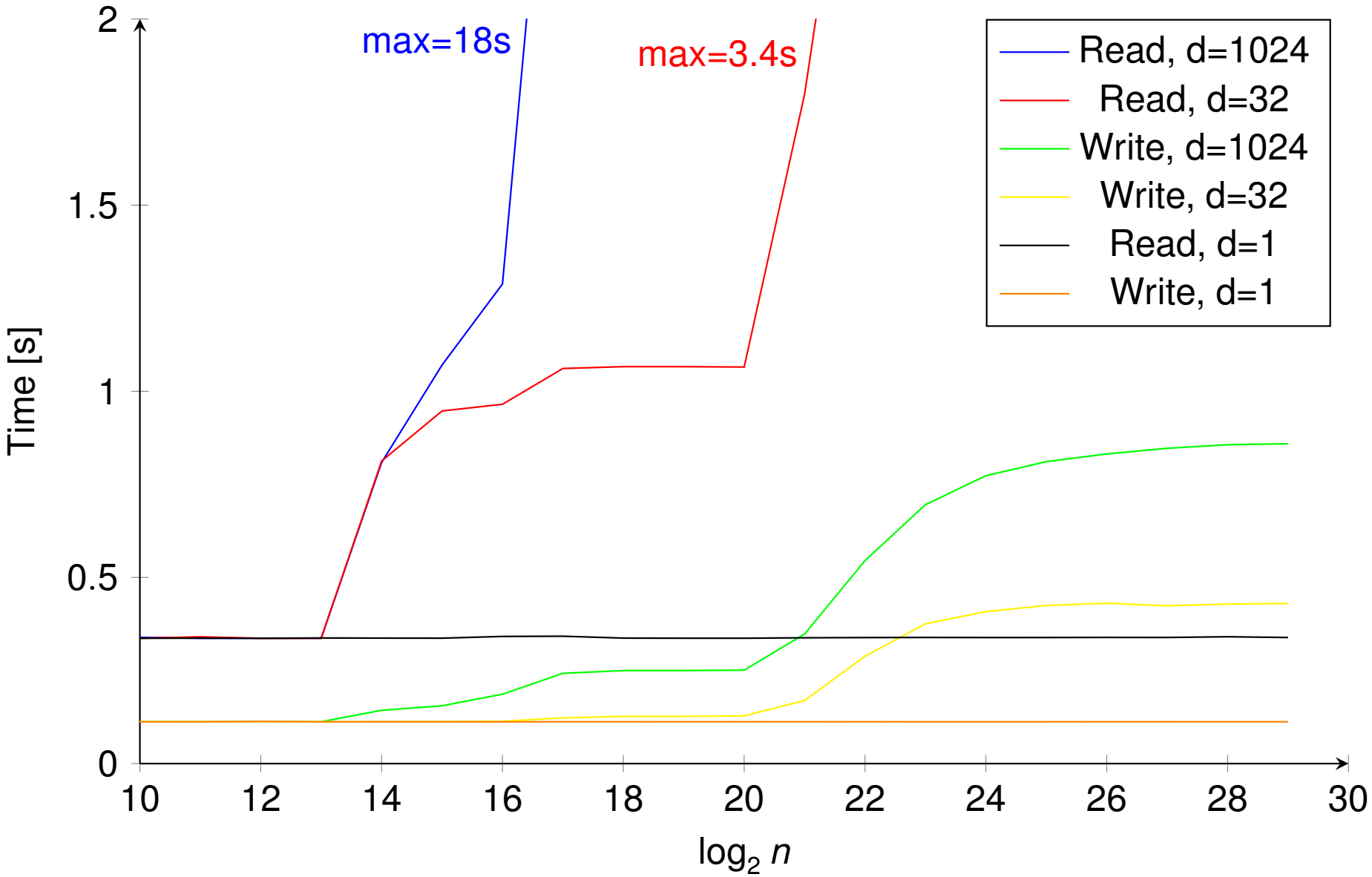
Čtení z paměti

```
# 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

```
mask = (1 << n) − 1
# Měříme dobu průběhu cyklu v závislosti na parametrech n a d
for (j=0; j< 2^28; j++) do
    A[(j*d) & mask] = j # Dokola zapisujeme na d-té pozice
```

Srovnání rychlosti čtení a zápisu z paměti



Pár triků na závěr

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);
```

Cvičení:

6.1. Vymyslete co nejkratší funkci na transpozici matic. Celou transpozici je možné implementovat pomocí jedné rekurzivní funkci, kde vlastní rekurzivní volání je na 5 řádků a dalších 5-10 řádků je na deklarace a zakončení rekurze (v závislosti na zvoleném jazyce). Funkce by měla transponovat i čtvercové matice, jejichž velikost není mocnina dvojky.

6.2. V praxi rekurzivní volání značně zpomaluje transponování matic, a proto zkuste vymyslet postup nevyužívající rekurzi (ani zásobník). Dokázali byste efektivně vygenerovat posloupnost pozic $[i_1, j_1], [i_2, j_2], \ldots, [i_{\binom{k}{2}}, j_{\binom{k}{2}}]$ matice typu $k \times k$, která transponuje matici (zavoláním Swap$(A_{i_l,j_l}, A_{j_l,i_l})$ pro všechna $l = 1, 2, \ldots, \binom{k}{2}$) v cache-oblivious pořadí? Pro zjednodušení můžete předpokládat, že k je mocnina dvojky.

6.3. Podobný problém jako u rekurzivní transpozice matic nastává i u van Emde Boas rozložení. Jak pro danou pozici v poli určit pozice otce a synů v van Emde Boas rozložení? Najděte co nejrychlejší způsob nalezení cesty z daného vrcholu do kořene.

6.4. Proveďte cache-oblivious analýzu algoritmů na násobení dlouhých čísel.

6.5. Analyzujte lineární algoritmus na hledání mediánu.

6.6. Analyzujte následující algoritmy na násobení matic:

- Dle definice
- Nejprve se druhá matice ztransponujte a poté se vynásobí s první
- Rekurzivní dělení na bloky

- Strassen

6.7. Určete střední hodnotu počtu přenesených bloků randomizovaného Quick-sortu.

6.8. k-way Merge-sort rozdělí pole na k částí, které rekurzivně setřídí a slije dohromady. Najděte optimální hodnotu k pro cache-aware k-way Merge-sort.

6.9.

Obsah

Hešovaní

Základní pojmy

- Máme univerzum $U = \{0, 1, \ldots, u-1\}$ všech prvků
- Chceme uložit podmnožinu $S \subseteq U$ velikosti n
- Uložíme S do pole velikosti m pomocí hešovací funkce $h : U \to M$, kde $M = \{0, 1, \ldots, m-1\}$
- 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

c-universální systém

Systém hešovacích funkcí $\mathcal{H}$ je c-universální, pokud náhodně zvolená $h \in \mathcal{H}$ splňuje $P[h(x) = h(y)] \leq \frac{c}{m}$ pro každé $x, y \in U$ a $x \neq y$. ①

Příklad c-universálního systémy (Cvičení 7.5)

- $\mathcal{H} = \{h_a(x) = (ax \bmod p) \bmod m;\ 0 < a < p\}$, kde $p > u$ je prvočíslo

1. Navíc obvykle vyžadujeme, aby hešovací funkci šlo spočítat v čase $\mathcal{O}(1)$ a aby funkci bylo možné popsat $\mathcal{O}(1)$ parametry.

Universální hešování

2-nezávislý systém hešovacích funkcí

Systém hešovacích funkcí $\mathcal{H}$ je 2-nezávislý, pokud náhodně zvolená $h \in \mathcal{H}$ splňuje $P[h(x_1) = z_1 \text{ a } h(x_2) = z_2] = \mathcal{O}\left(\frac{1}{m^2}\right)$ pro každé $x_1, x_2 \in U$ a $x_1 \neq x_2$ a $z_1, z_2 \in M$.

k-nezávislý systém hešovacích funkcí

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

Cvičení 7.7, 7.8, 7.9, 7.10

- k-nezávislý systém hešovacích funkcí je $(k-1)$-nezávislý
- 2-nezávislý systém hešovacích funkcí je c-universální pro nějaké c
- Najděte 1-universální systém, který není 2-nezávislý
- Jestliže $P[h(x_i) = z_i \text{ pro všechna } i = 1, \ldots, k] \leq \frac{1}{m^k}$ pro všechna $z_1, \ldots, z_k \in M$, pak $P[h(x_i) = z_i \text{ pro všechna } i = 1, \ldots, k] = \frac{1}{m^k}$ pro všechna $z_1, \ldots, z_k \in M$

1-nezávislý systém není užitečný

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

Universální hešování: Multiply-mod-prime

Systém Multiply-mod-prime

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

Lemma

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

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

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

Důkaz

- Pro danou dvojici (y_1, y_2) existuje jedinná 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 $\lceil \frac{p}{m} \rceil - 1$ 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 $p(\lceil \frac{p}{m} \rceil - 1) \leq p(\frac{p+m-1}{m} - 1) \leq \frac{p(p-1)}{m}$
- 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

- Zvolme $x_1, x_2, x_3 \in U$ po dvou různé takové, že $x_1 + x_2 = 2x_3$
- Zvolme $z_1, z_2, z_3 \in M$ takové, že $z_1 + z_2 = 2z_3$
- Předpokládejme, že $p = m$ je prvočíslo
- Jestliže $h(x_1) = z_1$ a $h(x_2) = z_2$, pak $h(x_3) = z_3$
  - $2h(x_3) \equiv_p 2(ax_3 + b) \equiv_p (ax_1 + b) + (ax_2 + b) \equiv_p z_1 + z_2 \equiv_p 2z_3$
- Podmíněná pravděpodobnost $P[h(x_3) = z_3 | h(x_1) = z_1 \text{ a } h(x_2) = z_2] = 1$
- $P[h(x_3) = z_3 \text{ a } h(x_1) = z_1 \text{ a } h(x_2) = z_2]$
  $= P[h(x_3) = z_3 | h(x_1) = z_1 \text{ a } h(x_2) = z_2]\ P[h(x_1) = z_1 \text{ a } h(x_2) = z_2]$
  $= P[h(x_1) = z_1 \text{ a } h(x_2) = z_2] = \Theta(\frac{1}{m^2})$, což není $\mathcal{O}(\frac{1}{m^3})$

Systém Poly-mod-prime

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

Cvičení 7.16: k-nezávislost

Systém Poly-mod-prime je k-nezávislý.

Universální hešování: Multiply-shift

Multiply-shift

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

Implementace v C

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

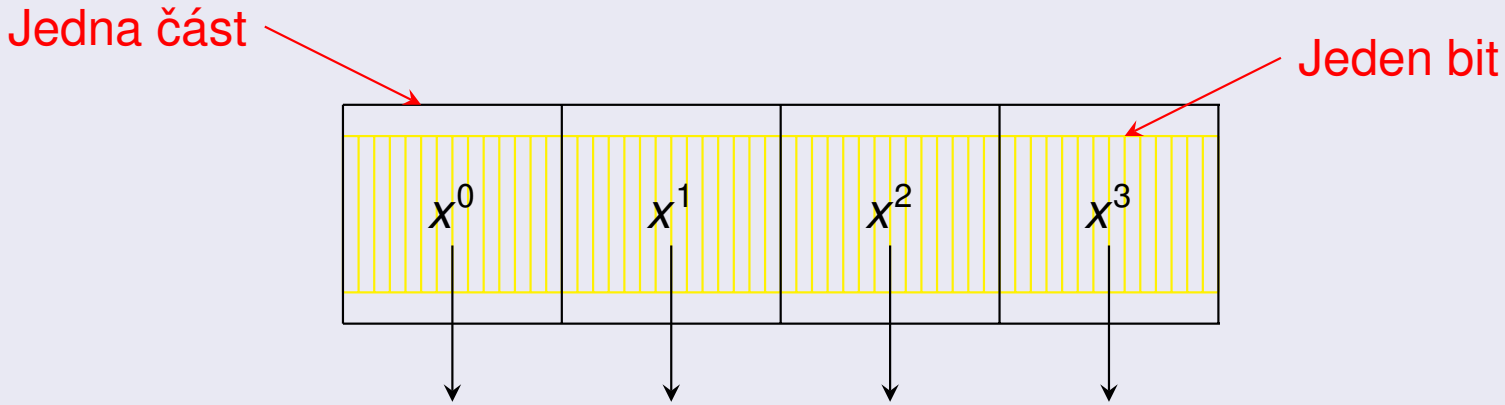
2-nezávislost (bez důkazu)

Pro každé $x_1, x_2 \in [2^w]$, $x_1 \neq x_2$ a $z_1, z_2 \in M$ platí $P[h(x_1) = z_1 \text{ a } h(x_2) = z_2] = \frac{1}{m^2}$.

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 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 $x^0, \ldots, x^{d-1}$ po $\frac{w}{d}$ bitech
- Pro každé $i \in [d]$ vybereme náhodnou hešovací funkci $T_i : [2^{w/d}] \to M$
- Hešovací funkce je $h(x) = T_0(x^0) \oplus \cdots \oplus T_{d-1}(x^{d-1})$

Univerzalita

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

Důkaz 2-nezávislosti (3-nezávislost je Cvičení 7.12)

- 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. $T_i(x_1^j)$ nabývá všech hodnot z M se stejnou pravděpodobností $\frac{1}{m}$ a náhodné proměnné $T_i(x_1^j)$ 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. Jestliže $h(x_1) = h(x_2) = h(x_3) = z$, pak $h(x_4) = z$
   - Z $h(x_1) = h(x_2) = z$ plyne
     $T_0(0) = T_0(x_1^0) = h_0(x_1) \oplus z = h_0(x_2) \oplus z = T_0(x_2^0) = T_0(1)$
   - Z $h(x_1) = h(x_3) = z$ plyne
     $T_1(0) = T_1(x_1^1) = h_1(x_1) \oplus z = h_1(x_3) \oplus z = T_1(x_3^1) = T_1(1)$
   - Z $h(x_1) = h(x_2) = h(x_3) = z$ plyne
     $h(x_4) = h_{0,1}(x_4) \oplus T_0(1) \oplus T_1(1) = h_{0,1}(x_1) \oplus T_0(0) \oplus T_1(0) = h(x_1) = z$,
     kde $h_{0,1}(x) = T_2(x^2) \oplus \cdots \oplus T_{d-1}(x^{d-1})$
3. Podmíněná pravděpodobnost $P[h(x_4) = z | h(x_1) = h(x_2) = h(x_3) = z] = 1$
4. $P[h(x_1) = h(x_2) = h(x_3) = h(x_4) = z]$
   $= P[h(x_4) = z | h(x_1) = h(x_2) = h(x_3) = z] \cdot P[h(x_1) = h(x_2) = h(x_3) = z]$
   $= \frac{1}{m^3}$, což není $\mathcal{O}\left(\frac{1}{m^4}\right)$

Universální hešování: Vektory

Multiply-shift pro vectory pevné délky k

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

Poly-mod-prime pro různě dlouhé řetězce I

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

Poly-mod-prime pro různě dlouhé řetězce II

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

1. $x_0, \ldots, x_d$ 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

Lemma (Cvičení 7.2)

Čekáme-li na událost, která nastane v jednom kroku s pravděpodobností p (nezávisle na ostatních krocích), pak $E[\#\text{ kroků}] = \frac{1}{p}$.

Narozeninový paradox

Pokud n míčů hodíme do $m \geq n$ košů, pak pravděpodobnost, že v každém koši je nejvýše jeden míč, je ①

$$\prod_{i=1}^{n-1} \frac{m-i}{m} \sim e^{-\frac{n^2}{2m}}.$$

- $\prod_{i=1}^{n} \frac{m-i+1}{m} = \prod_{i=1}^{n-1} \left(1 + \frac{-i}{m}\right) \sim \prod_{i=1}^{n-1} e^{-\frac{i}{m}} = e^{-\frac{\sum_{i=1}^{n-1} i}{m}} = e^{-\frac{\binom{n}{2}}{m}} \sim e^{-\frac{n^2}{2m}}$

Důsledek

Pro danou podmnožinu $S \subseteq U$ velikosti n a pro $m = \Theta(n^2)$ lze najít perfektní hešovací funkci do tabulky velikosti m tak, že vyzkoušíme v průměru $\mathcal{O}(1)$ náhodných hešovacích funkcí.

1. Předpokládáme, že se každým míčem trefíme do právě jednoho koše, do každého koše se trefíme se stejnou pravděpodobností a jednotlivé hody jsou nezávislé. Důkaz: i-tý míč padne do prázdného koše s pravděpodobností $\frac{m-i+1}{m}$, takže pravděpodobnost, že v každém koši bude nejvýše jeden míč, je $\prod_{i=1}^{n-1} \frac{m-i}{m}$. Použitím aproximaci prvního řádu funkce $e^x \sim 1 + x$ dostáváme

$$\prod_{i=1}^{n-1} \left(1 + \frac{-i}{m}\right) \sim \prod_{i=1}^{n-1} e^{-\frac{i}{m}} = e^{-\frac{\sum_{i=1}^{n-1} i}{m}} = e^{-\frac{\binom{n}{2}}{m}} \sim e^{-\frac{n^2}{2m}}.$$

FKS statické hešování (Fredman, Komlós, Szemerédi [6])

Cíl: Dvouúrovňové perfektní hešovaní do paměti velikosti $\Theta(n)$

- V první úrovni hešujeme všechny prvky do tabulky velikosti $\Theta(n)$
- Nechť n_j je počet prvků v j-té přihrádce
- V druhé úrovní pro každou přihrádku j sestrojíme perfektní hešovací funkci do tabulky velikosti $\Theta(n_j^2)$
- K volbě hešovací funkce používáme c-universální systém
- Operaci Find máme v konstantním čase výpočtem dvou hešovacích funkcí

Markovova nerovnost

Jestliže X je nezáporná náhodná veličina a $c > 1$, pak $P[X < cE[X]] > \frac{c-1}{c}$.

Nalezení primární hešovací funkce s nejvýše n kolizemi

- $E[\#\text{ kolizí}] = \sum_{\{x,y\}} P[h(x) = h(y)] \leq \binom{n}{2} \frac{c}{m} < \frac{cn^2}{2m} \leq \frac{n}{2}$ pro $m = \lceil cn \rceil$
- $P[\#\text{ kolizí} < n] \geq P[\#\text{ kolizí} < 2E[\#\text{ kolizí}]] \geq \frac{1}{2}$
- Očekávaný počet pokusů na nalezení hešovací funkce s nejvýše n kolizemi je menší než 2
- Paměťová i očekávaná časová složitost je $\mathcal{O}(n)$

FKS statické hešování (Fredman, Komlós, Szemerédi [6])

Cíl: Dvouúrovňové perfektní hešovaní do tabulky velikosti $\Theta(n)$

- V první úrovni hešujeme všechny prvky do tabulky velikosti $\Theta(n)$
- Nechť n_j je počet prvků v j-té přihrádce
- V druhé úrovní pro každou přihrádku j sestrojíme perfektní hešovací funkci do tabulky velikosti $\Theta(n_j^2)$
- K volbě hešovací funkce používáme c-universální systém
- Operaci Find máme v konstantním čase výpočtem dvou hešovacích funkcí

Nalezení sekundárních perfektních hešovacích funkcí

- $E[\#\text{ kolizí v } j\text{-té tabulce}] \leq \frac{cn_j^2}{2m_j} \leq \frac{1}{2}$ pro $m_j = \lceil cn_j^2 \rceil$
- $P[\#\text{ kolizí v } j\text{-té tabulce} < 1] > \frac{1}{2}$
- Součet velikostí sekundárních tabulek je $\mathcal{O}(n)$
  - $\sum_{j=1}^{m} m_j = \sum_{j=1}^{m} \left\lceil cn_j^2 \right\rceil \leq m + c\sum_{j=1}^{m} n_j + 2c\sum_{j=1}^{m} \binom{n_j}{2} =$
    $m + cn + 2c\,(\#\text{ kolizí primární tabulky}) = \mathcal{O}(n)$
- Očekávaná časová složitost nalezení všech sekundárních hešovacích funkcí je $\sum_{j=1}^{m} \mathcal{O}(m_j) = \mathcal{O}(n)$

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.

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 $c > 0$ 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 [23])
- tabulkové hešování (Pătraşcu, Thorup [21])

Očekávaná délka nejdelšího řetězce (Důsledek, Cvičení)

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 $X_1, \ldots, X_n$ 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 $P[\max_j A_j > c\mu] \leq mP[A_1 > c\mu] < me^{-\mu} e^{c\mu - c\mu \log c}$

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

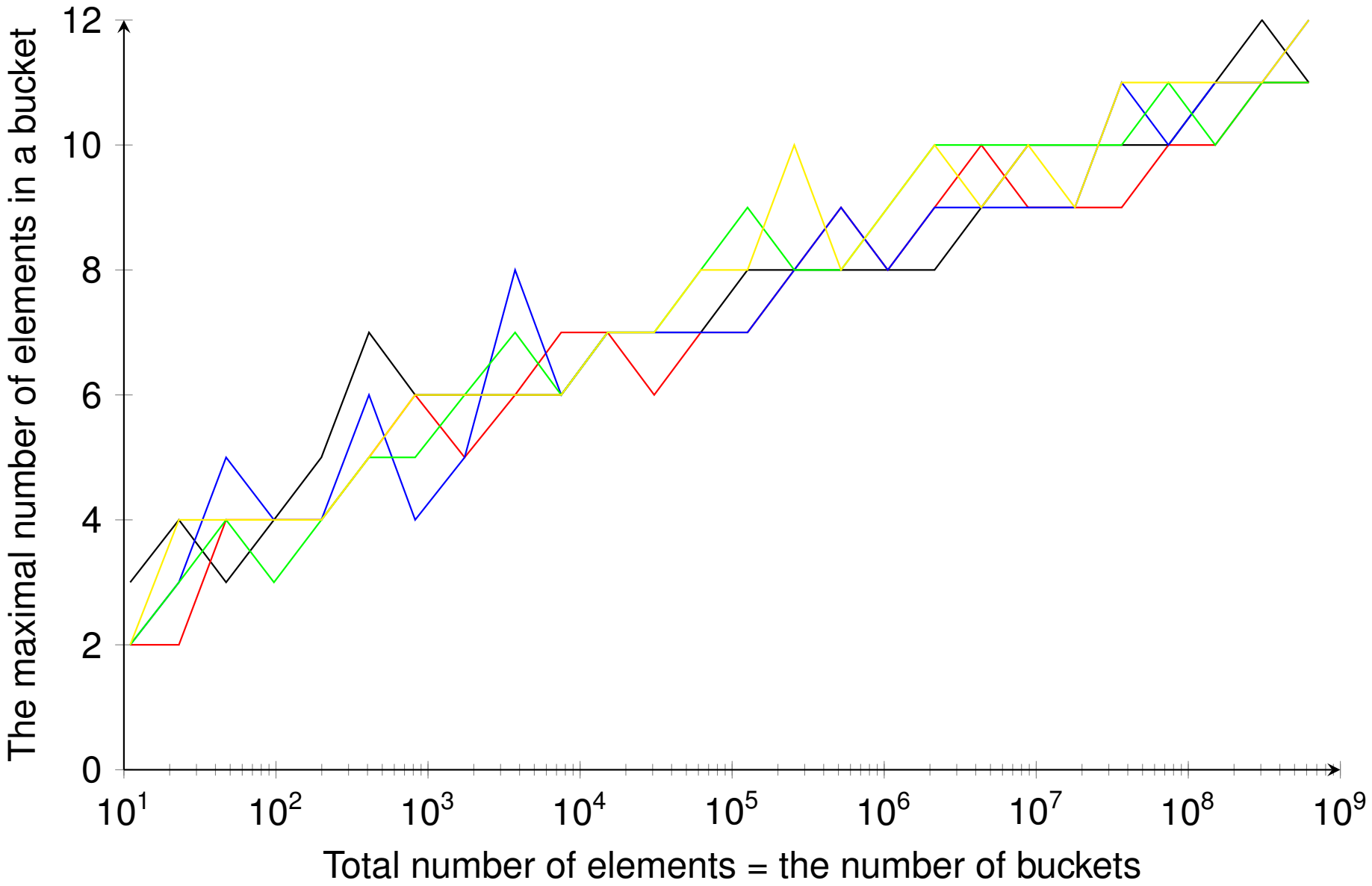
- Nechť $\epsilon > 0$ a $c = (1+\epsilon)\frac{\log n}{\mu \log \log n}$. Tedy $c\mu = (1+\epsilon)\frac{\log n}{\log \log n}$

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

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

Hešování se separovanými řetězci: Nejdelší řetězec: 5 pokusů



Lemma: Počet prvků v $\Theta(\log n)$ přihrádkách

Jestliže $\alpha = \Theta(1)$, systém hešovacích funkcí je úplně náhodný, pak v pevných $d \log n$ přihrádkách T je nejvýše $e\alpha d \log n$ prvků s velkou pravděpodobností, kde $d \geq 0$. ① ②

Důkaz

- Nechť $X_i = 1$, pokud $h(i) \in T$, jinak $X_i = 1$, kde $i \in S$
- Počet prvků v přihrádkách T je $X = \sum_i X_i$
- Očekávaný počet prvků v T je $\mu = E[X] = E[\sum_{j \in T} A_j] = |T|E[A_j] = \alpha d \log n$
- Chernoff: $P[X > c\mu] < \frac{e^{(c-1)\mu}}{c^{c\mu}}$
- $P[X > c\mu] < e^{d\alpha \log n(c-1-c\log c)} = n^{d\alpha(c-1-c\log c)} = n^{-d\alpha}$ pro $c = e$
- $P[X \leq c\mu] \geq 1 - \frac{1}{n^{d\alpha}}$

1. Pravděpodobnost, že množina přihrádek T obsahuje více než $e\alpha d \log n$, se počítá pro náhodnou volbu hešovací funkce $h \in \mathcal{H}$. Proto není možné zvolit zákeřnou množinu přihrádek pro danou hešovací funkci.
2. Zjednodušeně řečeno, $\Theta(\log n)$ obsahuje $\Theta(\log n)$ prvků s velkou pravděpodobností.

Hešování se separovanými řetězci: Amortizovaná složitost hledání

Lemma: Počet prvků v $\Theta(\log n)$ přihrádkách

Jestliže $\alpha = \Theta(1)$, systém hešovacích funkcí je úplně náhodný, pak v pevných $d \log n$ přihrádkách T je nejvýše $e\alpha d \log n$ prvků s velkou pravděpodobností, kde $d \geq 0$.

Lemma (Cvičení)

Jestliže $\alpha = \Theta(1)$ a velikost $S \subseteq U$ je $\mathcal{O}(\log n)$, pak funkce vybraná z úplně náhodného hešovacího systému je perfektní na S s velkou pravděpodobností. ①

Amortizovaná složitost pro hledání $\Omega(\log n)$ prvků (Pătraşcu [19])

Jestliže $\alpha = \Theta(1)$, systém hešovacích funkcí je úplně náhodný a hledáme $\Omega(\log n)$ různých prvků, pak amortizovaná složitost hledání jednoho prvku je $\mathcal{O}(1)$ s velkou pravděpodobností. ②

Amortizovaná složitost pro hledání $\Omega(\log n)$ prvků (Pătraşcu [19])

Jestliže $\alpha = \Theta(1)$, systém hešovacích funkcí je úplně náhodný a máme posloupnost operací Find délky $\Omega(\log n)$, při které můžeme využívat cache $\Theta(\log n)$ prvků, pak amortizovaná složitost hledání jednoho prvku je $\mathcal{O}(1)$ s velkou pravděpodobností. ③ ④

1. $\mathcal{O}(\log n)$ různých prvků je v tabulce velikosti $\Omega(n)$ uloženo v různých přihrádkách s velkou pravděpodobností.
2. Posloupnost hledání rozdělíme na podposloupnosti délky $\Omega(\log n)$. Prvky z každé podposloupnosti jsou uloženy v různých přihrádkách s velkou pravděpodobností (narozeninový paradox), a tyto přihrádky nejvýše $\mathcal{O}(\log n)$ prvků.
3. Jestliže se hledané prvky mohou opakovat, pak si výsledky posledních $\log n$ hledání pamatujeme v cache.
4. Na reálných počítačích by si mohla L1 cache pamatovat prvky z posledních $\log n$ prohledávaných přihrádek.

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

- Lína 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 [14])
- konstantní pro $\log(n)$-nezávislý systém (Schmidt, Siegel, 1990 [22])
- $\mathcal{O}\left(\frac{1}{\epsilon^{\frac{13}{6}}}\right)$ pro 5-nezávislý systém (Pagh, Pagh, Ruzic, 2007 [17])
- $\mathcal{O}(\log n)$ pro 4-nezávislý systém (Pătraşcu, Thorup, 2010 [20]) ①
- $\mathcal{O}\left(\frac{1}{\epsilon^2}\right)$ pro tabulkové hešování (Pătraşcu, Thorup, 2012 [21])

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

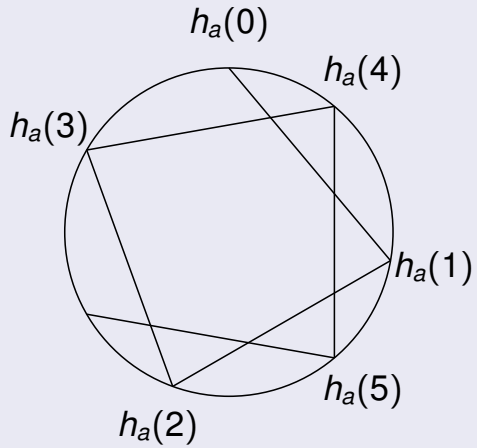
1. Nechť $1 < c < \frac{1}{\alpha}$ a $q = \left(\frac{e^{c-1}}{c^c}\right)^{\alpha}$
   - Platí $0 < q < 1$
2. Nechť $p_t = P[|\{x \in S;\ h(x) \in T\}| = t]$ je pravděpodobnost, že do dané množiny přihrádek T velikosti t je zahešováno t prvků. Pak $p_t < q^t$. ②
   - Nechť X_i je náhodná proměnná indikující, zda prvek i je zaměšován 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 $p'_k < \frac{q^k}{1-q}$. ③
   - $p'_k < \sum_{s=0}^{\infty} p_{s+k} < q^k \sum_{s=0}^{\infty} q^s = \frac{q^k}{1-q}$
4. Očekávaný počet porovnání klíčů je $\sum_{k=0}^{m} k p'_k < \frac{1}{1-q} \sum_{k=0}^{\infty} k q^k = \frac{2-q}{(1-q)^3}$

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 [14] spočítal očekávanou složitost přesně, ale výpočet je náročný.
2. Zde uvažujeme prvky, které hešovací funkce zobrazí do daných přihrádek, a nikoliv prvky, které se do daných přihrádek dostanou vlivem lineárního přidávání.
3. 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?

Kombinace systému Multiply-shift a Lineárního přidávání

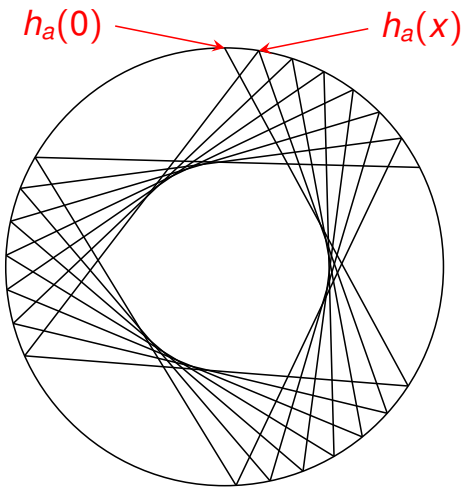
- Multiply-shift: $h_a(x) = (ax \bmod 2^w) >> (w - l)$
- Označíme $h'_a(x) = \frac{ax \bmod 2^w}{2^{w-l}} = \frac{ax}{2^{w-l}} \bmod 2^l$
- Pak $h_a(x) = \lfloor h'_a(x) \rfloor$
- Jaká je očekávaná složitost vložení prvků $S = \{1, \ldots, n\}$?



Vlastnosti

1. Označme $||x|| = \min\{x, 2^l - x\}$, tj. vzdálenost od $h_a(0)$ po kružnici
2. Platí $||h'_a(x) - h'_a(y)|| = ||h'_a(x-1) - h'_a(y-1)|| = ||h'_a(x-y)||$ pro $x \geq y$
3. Platí $||h'_a(ix)|| \leq i||h'_a(x)||$ pro všechna $i \in \{1, \ldots, n\}$
4. Jestliže $||h'_a(x)|| \geq 1$ pro všechna $x \in S$, pak h_a je perfektní na S
5. $P[h_a \text{ není perfektní}] \leq P[\exists x \in S : ||h'_a(x)|| < 1] \leq \sum_{x \in S} P[||h'_a(x)|| < 1] \leq \frac{2n}{m}$
6. $P[h_a \text{ je perfektní na } S] = 1 - P[h_a \text{ není perfektní na } S] \geq 1 - \frac{2n}{m}$

Lineární přidávání: Proč 2-nezávislý systém nestačí?



Kombinace systému Multiply-shift a Lineárního přidávání

- Uvažujme prvek $x \in S$ takový, že $||h'_a(x)|| \leq \frac{1}{2}$
- Prvky $x, 2x, \ldots, kx$ patří do přihrádek $0, 1, \ldots, \lfloor \frac{1}{2}k \rfloor$, kde $k = \lfloor \frac{n}{x} \rfloor$
- Máme nejvýše x skupin po alespoň k prvcích
- Složitost operace Insert je $\Omega(\frac{n}{x})$, platí-li $||h'_a(x)|| \leq \frac{1}{2}$

Lineární přidávání a hešovací systémy (Pătraşcu, Thorup, 2010 [20])

- 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 [18])

Popis

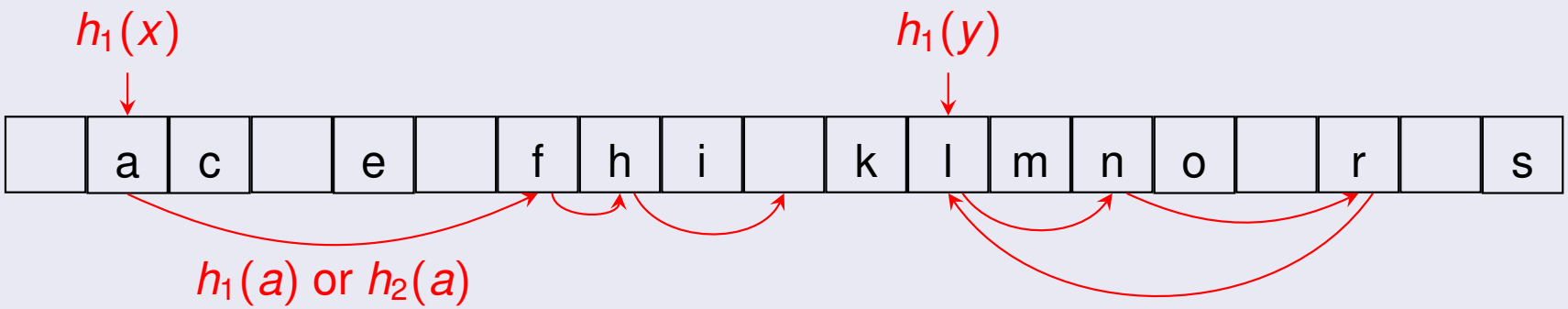
Pro dvě hešovací funkce h_1 a h_2 prvek x musí být uložen v přihrádce $h_1(x)$ nebo $h_2(x)$. V jedné přihrádce může být uložen nejvýše jeden prvek.

Operace Find a Delete

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

Přiklad 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

```
1  pos ← h1(x)
2  for n krát ① do
3      if T[pos] je prázdná then
4          T[pos] ← x
5          return
6      swap(x, T[pos])
7      if pos == h1(x) ② then
8          pos ← h2(x)
9      else
10         pos ← h1(x)
11 rehash()
12 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 [18]

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 [21]

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.

Cvičení:

7.1. Hodíme n míčků do n košů a označíme n_i počet míčů v i-té přihrádce. Dokažte, že $E[\sum_{i=1}^{n} n_i^2] = \Theta(n)$.

7.2. Dokažte, že čekáme-li na událost, která nastane v jednom kroku s pravděpodobností p (nezávisle na ostatních krocích), pak $E[\#\text{ kroků}] = \frac{1}{p}$.

7.3. Dokažte, že pomocí c-universálního hešovacího systému je možné efektivně zkonstruovat (jednoúrovňové) perfektní hešování do tabulky velikosti $m = \Theta(n^2)$.

7.4. Dokažte, že k sestrojení dvouúrovňového perfektního hešování FKS stačí 2-nezávislý hešovací systém.

7.5. Dokažte, že systém $\mathcal{H} = \{h_a(x) = (ax \bmod p) \bmod m;\ 0 < a < p\}$ je c-universální a najděte nejmenší možné c.

7.6. Formálně dokažte, že systém všech náhodných funkcí je 1-universální a k-nezávislý pro všechna k.

7.7. Formálně dokažte, že k-nezávislý systém je $(k-1)$-nezávislý.

7.8. Formálně dokažte, že 2-nezávislý systém je c-universální pro nějaké c.

7.9. Najděte 1-universální systém, který není 2-nezávislý.

7.10. Dokažte, že když $P[h(x_i) = z_i \text{ pro všechna } i = 1, \ldots, k] \leq \frac{1}{m^k}$ pro všechna $z_1, \ldots, z_k \in M$, pak $P[h(x_i) = z_i \text{ pro všechna } i = 1, \ldots, k] = \frac{1}{m^k}$ pro všechna $z_1, \ldots, z_k \in M$.

7.11. Proč je hešovací systém Multiply-mod-prime počítá dvě modula? Předpokládejme, že velikost tabulky m je prvočíslo, a uvažujme hešovací systém funkcí $h_{a,b}(x) = ax + b \bmod m$ pro $a, b \in M$. Je tento systém c-universální pro nějaké c nezávislé na m?

7.12. Dokažte, že tabulkové hešování je 3-nezávislé.

7.13. Rozhodněte, zda systém $\mathcal{H} = \{h_a(x) = (x + a \bmod p) \bmod m;\ 0 \leq a < p\}$ je c-universální pro nějaké c.

7.14. Najděte nejmenší c, pro které je systém $\mathcal{H} = \{h_a(x) = (ax + b \bmod p) \bmod m;\ a, b \in [p]\}$ c-universální.

7.15. Rozhodněte, zda systém $\mathcal{H} = \{h_a(x) = (ax + b \bmod p) \bmod m;\ a, b \in [p], a \neq 0\}$ je 2-nezávislý.

7.16. Dokažte, že systém Poly-mod-prime $\mathcal{H} = \left\{h_{a_0,\ldots,a_{k-1}}(x) = (\sum_{i=0}^{k-1} a_i x^i \bmod p) \bmod m;\ a_0, \ldots, a_{k-1} \in [p]\right\}$ je k-nezávislý.

Obsah

Intervalový dotaz (Range query)

Popis problému

- Daná množina n bodů S v $\mathbb{R}^d$
- Intervalem rozumíme d-dimenzionální obdélník
- 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: $\mathcal{O}(n \log n)$

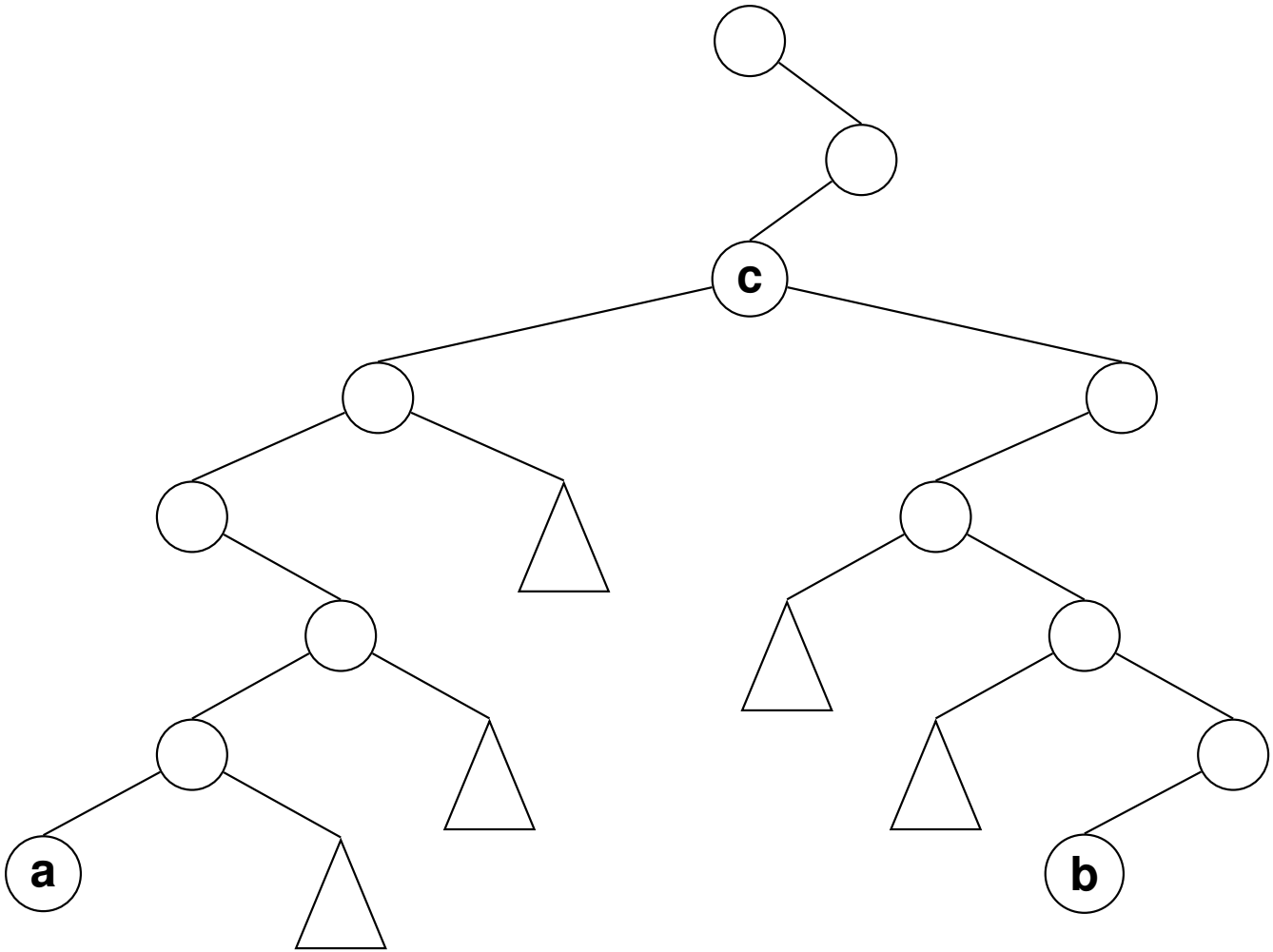
Insert: $\mathcal{O}(\log n)$

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

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

```
1  Procedure Query(vrchol stromu v, interval R)
2      if v je list then
3          Vypiš v, pokud leží v R
4      else if rozdělující nadrovina vrcholu v protíná R then
5          Query(levý syn v, R)
6          Query(pravý syn v, R)
7      else if R je „vlevo“ od rozdělující nadroviny vrcholu v then
8          Query(levý syn v, R)
9      else
10         Query(pravý syn v, R)
```

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

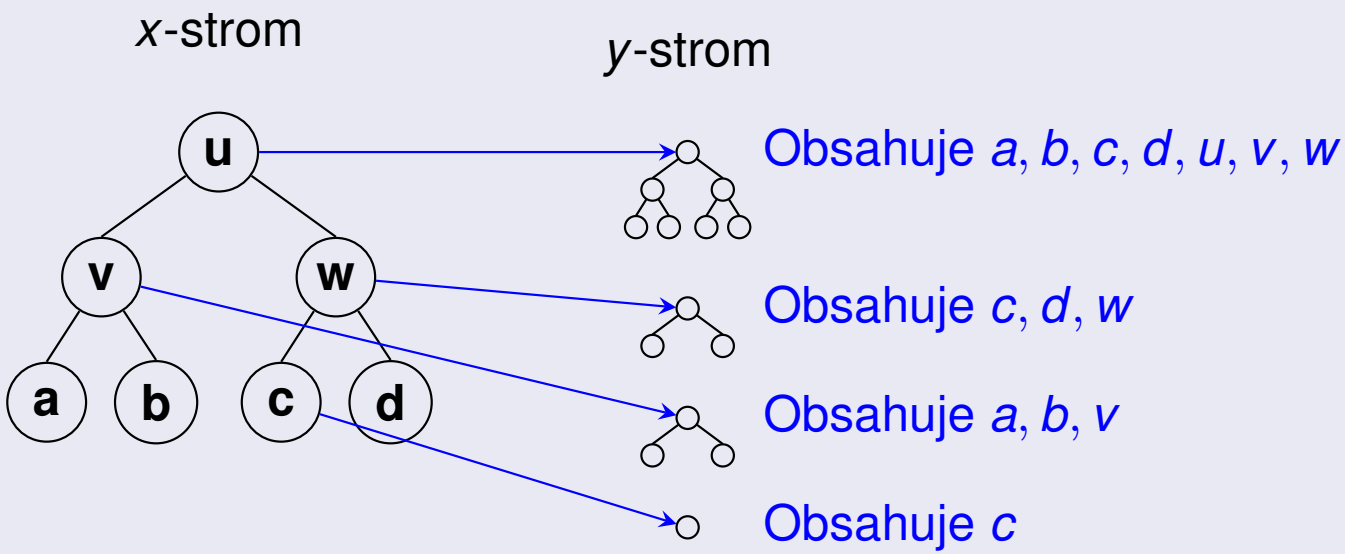
Obsah

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

Konstrukce

- Vybudujeme binární vyhledávací strom podle x-ové souřadnice bodů (x-strom)
- Pak pro každý vrchol v vybudujeme jeden binární vyhledávací strom podle y-ové souřadnice obsahující všechny body v podstromu vrcholu v (y-strom)
- Bodu můžou být uloženy ve všech vrcholech nebo jen v listech

Příklad



A tak dále

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

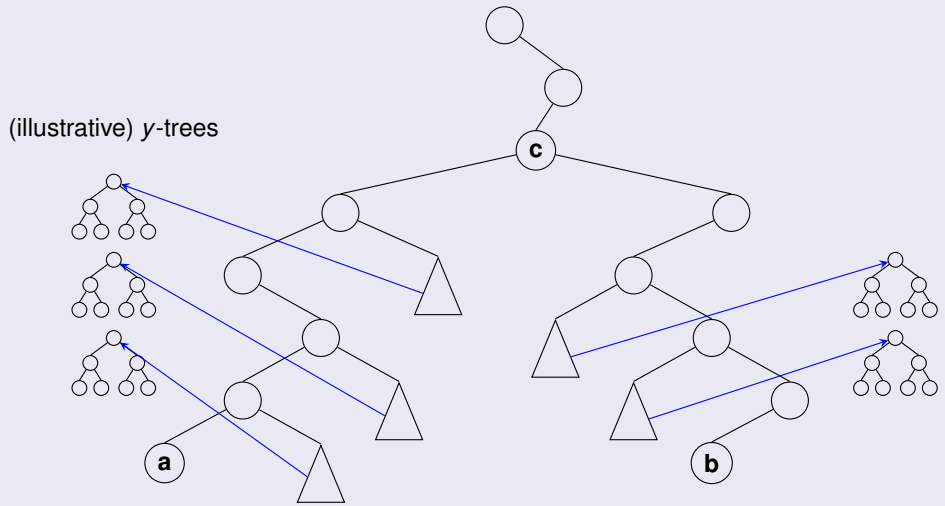
Každá vrstva x-stromu rozkládá body podle x-ové souřadnice. Proto každý bod je uložen v nejvýše jednom y-stromu z každé vrstvy x-stromu.

Paměťová složitost

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

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

Příklad



Složitost

$\mathcal{O}\left(k + \log^2 n\right)$ protože y-ový dotaz je volán v $\mathcal{O}(\log n)$ y-stromech

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

Popis

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

Representace

Struktura vrcholu intervalového stromu obsahuje

key	nadrovina rozdělující prostor mezi syny
left, right	ukazatel na levého a pravého syna
tree	ukazatel na kořen $(i+1)$-stromu

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

Asymptotické velikosti stromů a počty vrcholů, jsou-li stromy jsou vyvážené

	1-stromy	2-stromy	3-stromy	$(d-1)$-stromy	d-stromy
Počet stromů obsahující daný bod	1	$\log n$	$\log^2 n$	$\log^{d-2} n$	$\log^{d-1} n$
Celkový počet vrcholů	n	$n \log n$	$n \log^2 n$	$n \log^{d-2} n$	$n \log^{d-1} n$
Celkový počet stromů	1	n	$n \log n$	$n \log^{d-3} n$	$n \log^{d-2} n$

Paměť

- Indukcí: každý bod je uložený v $\mathcal{O}\left(\log^{i-1} n\right)$ i-stromech
- Celková paměťová složitost je $\mathcal{O}\left(n \log^{d-1} n\right)$

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

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

```
1  Procedure Build(množina bodů M, dimenze stromu d, aktuální souřadnice i)
2      if |M| = 1 then
3          return nový list obsahující jediný vrchol M
4      if i = d then
5          return kořen stromu vytvořený ze setříděného pole
6      v ← nový vrchol
7      v.tree ← Build(M, d, i + 1)
8      v.key ← medián i-tých souřadnic bodů M
9      M_l, M_r ← rozděl M na body mající i-tou souřadnici menší a větší než v.key
10     v.left ← Build(M_l, d, i)
11     v.right ← Build(M_r, d, i)
12     return v
```

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 $\Theta(n \log^{d-1} n)$
- Časová složitost vytvoření všech d-stromů je $\mathcal{O}\left(n \log^{d-1} n\right)$

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

- Počet vrcholů ve všech i-stromech je $\Theta(n \log^{i-1} n)$
- 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

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

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

```
1  Procedure Query(vrchol v, aktuální souřadnice i)
2      if v = NIL then
3          return
4      if v.key ≤ a_i then
5          Query(v.right, i)
6      else if v.key ≥ b_i then
7          Query(v.left, i)
8      else
9          Query_left(v.left, i)
10         Query_right(v.right, i)
11 Procedure Query_left(vrchol v, aktuální souřadnice i)
12     if v = NIL then
13         return
14     if v.key < a_i then
15         Query_left(v.right, i)
16     else
17         Query_left(v.left, i)
18         if i < d then
19             Query(v.right.tree, i + 1)
20         else
21             Vypiš všechny body v podstromu vrcholu v.right
```

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

Složitost

- 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 $\mathcal{O}\left(\log^{i-1} n\right)$
- 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 alespoň 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 porušuje vyvažovací podmínku, tak celý jeho podstrom znovu vytvoříme operací Build (složitost: amortizovaná analýza)

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

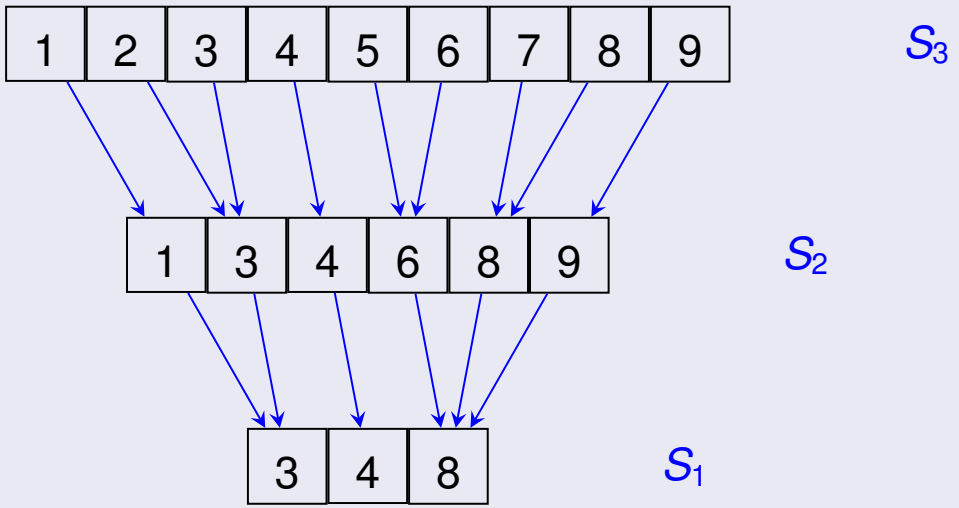
Kaskádování (Fractional cascading)

Motivační problém

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

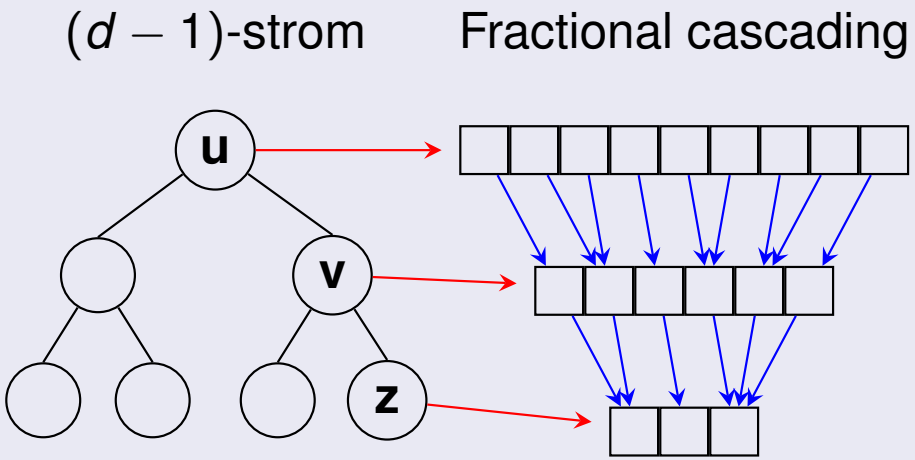
$\mathcal{O}(m + \log n)$

1. Triviálním řešením získáme složitost $\mathcal{O}(m \log n)$, 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: $\mathcal{O}\left(k + \log^{d-1} n\right)$

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

Chazelle [3, 4]

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

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

Chazelle, Guibas [5] pro $d \geq 3$

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

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

Obsah

Interval trees

Vstup

Množina stromů $S = \{I_1, \ldots, I_n\}$, kde $I_i = \langle a_i, b_i \rangle$.

Rekurzivní konstrukce binárního stromu

Nechť

- m je medián koncových bodů $a_1, b_1, \ldots, a_n, b_n$,
- $S_m = \{I_i;\ a_i \leq m \leq b_i\}$ jsou intervaly obsahující m,
- $S_l = \{I_i;\ b_i < m\}$ intervaly menší než m and
- $S_r = \{I_i;\ m < a_i\}$ intervaly větší než m.

Kořen stromu obsahuje

- dvě pole intervalů S_m setříděných podle levého a pravého konce intervalů,
- interval tree intervalů S_l v levém podstromu a
- interval tree intervalů S_r v pravém podstromu. ①

Složitost

- Doba konstrukce $\mathcal{O}(n \log n)$. ②
- Paměť $\mathcal{O}(n)$. ③

1. If S_l or S_r is empty, then there is no left or right child, respectively.
2. There are at most n end-points smaller than m, so S_l contains at most $\frac{n}{2}$ intervals. Therefore, the time complexity satisfies the recurrence formula $T(n) \leq 2T(\frac{n}{2}) + \Theta(n)$.
3. Every interval is stored in exactly one node. If S_m is empty, then n is even and both S_l and S_r contains $\frac{n}{2}$ intervals. There are at most $n - 1$ such nodes. Therefore, the tree has at most $2n - 1$ nodes.

Interval trees: Intervalový dotaz

Popis problému

Pro daný interval $Q = \langle a_q, b_q \rangle$ majít všechny intevaly mající průnik s intervalem Q.

Rekurzivní algoritmus

1 **if** $a_q \leq m \leq b_q$ **then**
2 | Vypíšeme všechny intervaly S_m
3 | Rekurzivně zpracujeme oba podstromu
4 **else if** $b_q < m$ **then**
5 | Pomocí pole intervalů S_m setříděných podle levého konce najdeme všechny intervaly mající průnik s Q
6 | Rekurzivně zpracujeme levý podstrom
7 **else**
8 | Pomocí pole intervalů S_m setříděných podle pravého konce najdeme všechny intervaly mající průnik s Q
9 | Rekurzivně zpracujeme pravý podstrom

Složitost

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

Cvičení:

8.1. Dokažte, že paměťová složitost intervalového stromu je $\Omega(n \log^{d-1})$.

8.2. Zkuste dokázat časovou složitost vytvoření intervalového stromu pomocí Master theorem

8.3. Jak určit počet bodů v daném intervalu?

8.4. Analyzujte složitost operací Insert a Delete pomocí potenciálové metody.

8.5. Jak dlouho trvá postavit intervalový strom s kaskádováním?

8.6. Je možné zkombinovat dynamizaci intervalových stromů pomocí BB[α]-stromů s kaskádováním?

Obsah

R-strom

Základní pojmy

- Obdélníkem v R^n rozumíme n-rozměrný interval, tj. n-rozměrný kvádr, jehož každá hrana je paralelní s některou osou.
- Ohraničujícím obdélníkem množiny $M \subseteq R^n$ rozumíme nejmenší obdélník obsahující S.

Cíl

- Pro grafický software (vektorový editor, CAD, GIS, ...) vytvořit datovou strukturu, která bude umět efektivně vyhledávat objekty splňující různé podmínky podle požadavků uživatele.
- Pro zjednodušení uvažujeme jen ohraničující obdélníky jednotlivých objektů.

R-strom

- Každý vnitřní vrchol (kromě kořene) R-stromu má a až b synů.
- Všechny listy jsou ve stejné hladině.
- Každý objekt (resp. jeho ohraničující obdélník) je uložen jednom listu.
- Každý vnitřní vrchol si pamatuje ohraničující obdélník obsahující všechny objekty ve svém podstromu.

Bodový a intervalový dotaz

- Hledáme objekty ležící v daném bodě nebo mající neprázdný průnik s daným objektem (obdélníkem) O.
- Rekurzivně z vrcholu pokračujeme do synů, jejichž ohraničující obdélník má neprázdný průnik s O.

Hledání nejbližšího objektu k danému bodu

- Vzdálenost bodu p a objektu $O \subseteq \mathbb{R}^n$ je $\text{dist}(p, O) = \min_{x \in O} ||p - x||$.
- Jestliže existuje objekt $O^\star$ a pro ohraničující obdélník O vrcholu u R-stromu platí $\text{dist}(p, O^\star) < \text{dist}(p, O)$, pak v podstromu vrcholu u neexistuje objekt bližší k bodu p než je $O^\star$.

Algoritmus založený na průchodu do hloubky

```
1  Nechť O* je dosud nejbližší nalezený objekt
2  Procedure NearestNeighbour(vrchol u, bod p)
3      if u je list then
4          Zkontroluje, jestli u obsahuje objekt bližší k p než O*
5      else
6          Vytvoříme seznam L všech synů vrcholu u
7          Seznam L setřídíme vhodných třídících kritérií, např. dist(p, O)
8          for v ∈ L do
9              if Obdélník O vrcholu v splňuje podmínky dist(p, O*) ≥ dist(p, O) then
10                 NearestNeighbour(v, p)
```

Algoritmus založený na prioritní frontě vrcholů, které zbývá projít

```
1  Nechť O* je dosud nejbližší nalezený objekt (na začátku libovolný)
2  Prioritní fronta F ← kořen
3  while F je neprázdná do
4      u ← Pop(F)
5      if u je list then
6          if dist(p, O*) > dist(p, O), kde O je objekt vrcholu u then
7              O* := O
8              Z fronty F smažeme všechny vrcholu, pro jejichž obdélník O platí
               dist(p, O*) ≤ dist(p, O)
9      else if dist(p, O*) > dist(p, O), kde O je obdélník vrcholu u then
10         for v syn u do
11             Push(F, v)
```

Varianty

- Nalezení všech dvojic objektů s neprázdným průnikem
- Spatial join: Pro dané dvě množiny objektů R a S najít všechny dvojice objektů $r \in R$ a $s \in S$ s neprázdným průnikem

Algoritmus

```
1 Procedure Intersections(vrchol u, vrchol v)
2     for u′ syn u ① do
3         for v′ syn v do
4             if Obdélníky vrcholů u′ a v′ mají neprázdný průnik then
5                 if u′ a v′ jsou listy then
6                     Vypíšeme objekty ve vrcholech u′ a v′
7                 else
8                     Intersections(u′, v′)
```

1. Jestliže u je list, pak místo for-cyklu uvažujeme jen případ $u' \leftarrow u$. Podobně pro v.

R-strom: Insert

Idea

Nové objekty vkládáme podobně jako do (a,b)-stromu, ale

- v každém vrcholu musíme rozhodnout, do kterého syna pokračovat, a
- musíme rozhodnout, jak rozdělit syny při štěpení.

Tyto volby neovlivňují korektnost struktury, ale efektivitu vyhledávání.

Volba syna

Zvolíme takového syna s obdélníkem O, že po vložení nového objektu musíme O nejméně zvětšit.

Štěpení

Rozdělit vrchol u na vrcholu u_1 a u_2 tak, aby součet velikostí obdélníků v u_1 a u_2 byl nejmenší.

- Vyzkoušet všechny kombinace (může být neefektivní)
- Kvadratické štěpení
- Lineární štěpení

Obsah

Základní pojmy

Semi-dynamizace: Naučit datovou strukturu struktury vkládání nových prvků

Úplná dynamizace: Naučit datovou strukturu i mazání prvků

Persistence: Zapamatování si změn v datové struktuře a vyhledávání podle stavu v daném čase

Deamortizace: Modifikace operace mající amortizovanou složitost tak, že upravená operace garantuje složitost v nejhorším případě

Deamortizace dynamického pole

Cíl

Vkládání prvku do dynamického pole umíme v amortizovaně v konstantním čase. Jak vložit prvek do dynamického pole tak, aby složitost byla konstantní i v nejhorším případě?

Postup

- Na začátku máme n prvků v hlavním poli velikosti $2n$ a alokujeme si prázdné pomocné pole velikosti $4n$
- Při každé operaci Insert zkopírujeme dva prvky do pomocného pole
- Po n operacích Insert jsou všechny prvky zkopírované do pomocného pole, a proto můžeme hlavní pole zrušit, pomocné nazvat hlavním a alokovat nové pomocné pole velikosti $8n$

Složitost

Pamětová složitost $\mathcal{O}(n)$ a časová $\mathcal{O}(1)$ v nejhorším případě. ①

1. Předpokládáme, že alokace a dealokace bloku paměti trvá $\mathcal{O}(1)$.

Dynamizace přebudováním

Jednou za čas přebudujeme celou datovou strukturu

- Aplikace pro datové struktury omezené kapacity (pole, halda, hešovací tabulky)
- Vyčištění datové struktury od smazaných prvků
- Změna vnitřních parametrů

Složitost

- Nechť isložitost přebudování je $T(n)$ a přebudováváme po $\Theta(n)$ operacích
- Přebudování stojí $\mathcal{O}\left(\frac{T(n)}{n}\right)$ amortizovaně na operaci
- Předpokládáme $T(\mathcal{O}(n)) = \mathcal{O}(T(n))$

Částečné přebudování

Například BB[α]-stromy

Obecná semi-dynamizace

Vyhledávací problém

Vyhledávací problém je zobrazení $f : U_Q \times 2^{U_X} \to U_R$, kde

- U_Q je universum dotazů
- U_X je universum prvků (uvažujeme jen konečné podmnožiny prvků)
- U_R je universum výsledků

Rozložitelnost

Vyhledávací problém je rozložitelný, jestliže existuje zobrazení $\sqcup : U_R \times U_R \to U_R$ vyčíslitelná v konstantním čase taková, že pro všechna disjunktní $A, B \subseteq U_X$ a všechny dotazy $q \in U_Q$ platí $f(q, A \cup B) = f(q, A) \sqcup f(q, B)$.

Příklady

- Hledání nejbližšího bodu v $\mathbb{R}^d$ je rozložitelné a máme $U_Q = U_X = U_R = \mathbb{R}^d$
- Rozhodnutí zda bod leží v konvexním obalu bodů je nerozložitelné pro $d \geq 2$

Obecná semi-dynamizace

Uvažujeme statickou datovou strukturu se složitostmi

- $B_S(n)$ - čas na build
- $Q_S(n)$ - čas na dotaz
- $S_S(n)$ - paměť
- Předpokládáme, že $Q_S(n)$, $\frac{B_S(n)}{n}$ a $\frac{S_S(n)}{n}$ jsou neklesající

Konstrukce

- Prvky rozdělíme do bloků $B_0, B_1, \ldots$ takových, že B_i obsahuje 0 nebo 2^i prvků ①
- Pro každý neprázdný blok vytvoříme jednu statickou datovou strukturu ②
- Časová složitost $\sum_{i:B_i \neq \emptyset} B_S(2^i) \leq B_S(n)$ ③
- Prostorová složitost $\sum_{i:B_i \neq \emptyset} S_S(2^i) \leq S_S(n)$

Dotaz

- Dotaz položíme ve všech blocích a výsledky zkombinujeme funkcí $\sqcup$
- Složitost $\log n + \sum_{i:B_i \neq \emptyset} Q_S(2^i) \leq \mathcal{O}(Q_S(n) \log n)$ ④

1. Rozdělení je jednoznačně dáno binárním zápisem čísla n a bloků je nejvýše $\log n + 1$.
2. Prvky nemáme uloženy v jedné datové struktuře, ale máme $\mathcal{O}(\log n)$ instancí datové struktury.
3. $\sum_{i:B_i \neq \emptyset} B_S(2^i) = \sum_{i:B_i \neq \emptyset} 2^i \frac{B_S(2^i)}{2^i} \leq \sum_{i:B_i \neq \emptyset} 2^i \frac{B_S(n)}{n} = n\frac{B_S(n)}{n} = B_S(n)$
4. $\log n + \sum_{i:B_i \neq \emptyset} Q_S(2^i) \leq \log n + \sum_{i:B_i \neq 0} Q_S(n) \leq \mathcal{O}(Q_S(n) \log n)$

Obecná semi-dynamizace

Insert

- Vytvoříme nový blok B_0 s datovou strukturou obsahující jen nový prvek
- Jestliže máme dvakrát blok B_{i-1}, pak je zrušíme a vytvoříme B_i v čase $B_S(2^i)$ ①
- Analýza: B_i vytvoříme jednou za 2^i operací Insert
- Amortizovaná cena vytvoření B_i je $\frac{B_S(2^i)}{2^i} \leq \frac{B_S(n)}{n}$
- Musíme předplácet vytváření všech $B_i \Rightarrow$ amortizovaná složitost je $\mathcal{O}\left(\frac{B_S(n)}{n} \log n\right)$

Deamortizovaná semi-dynamizace

- Rebuild rozložíme mezi více operací a pokaždé provedeme kousek
- Každý blok máme nejvýše třikrát a navíc jeden rozpracovaný
- Jestliže blok B_{i-1} máme dvakrát, tak začneme vytvářet B_i, který vytvoříme po 2^{i-1} krocích
- Při každé operaci Insert provedeme $\frac{B_S(2^i)}{2^{i-1}} \leq 2\frac{B_S(n)}{n}$ instrukcí na vytvoření B_i pro každé $i \leq \log n$
- Složitost operace Insert je $\mathcal{O}\left(\frac{B_S(n)}{n} \log n\right)$ v nejhorším případě

1. Pokud neumíme efektivně vyjmenovat všechny prvky v datové struktuře, pak si navíc pamatujeme seznam prvků v každém bloku. Alternativně můžeme rovnou vytvořit B_i, jestliže před insertem byl B_i nejmenší neprázdný blok.

Úplná dynamizace

Předpoklad

Datová struktura musí umět efektivně prvky hledat a označovat smazané prvky (v čase $M_S(n)$). ①

Jednodušší varianta operace Delete bez binární verze pro operaci Insert

- Pokud máme v datové struktuře víc označených prvků než neoznačených, pak provedeme Rebuild. ②
- Rebuild nastane nejdříve po $\frac{n}{2}$ operacích Delete a trvá $B_S(n)$
- Amortizovaná složitost je $\mathcal{O}\left(M_S(n) + \frac{B_S(n)}{n}\right)$

Kombinace operací Insert a Delete

- Jestliže v bloku B_i je většina prvků označena, pak z B_i vytvoříme B_{i-1} ③
- Každá operace Delete musí předplatit $\frac{B_S(n)}{n}$
- Amortizovaná složitost operace Delete je $\mathcal{O}\left(\log n + M_S(n) + \frac{B_S(n)}{n}\right)$ ④

1. Například hešování s lineárním přidáváním.
2. Maximální poměr počtu označených a neoznačených prvků můžeme nastavit na libovolnou konstantu.
3. Pokud blok B_{i-1} existuje, tak vytvoříme B_i spojením nesmazaných prvků z původního B_i a všech prvků z B_{i-1}.
4. Alternativně můžeme přebudovat celou datovou strukturu, jestliže je většina prvků označena.

Obecné kaskádování (Fractional cascading)

TBD

https://en.wikipedia.org/wiki/Fractional_cascading

http://mj.ucw.cz/vyuka/1516/ds2/: přednáška 20.4.

Obsah

Bloom filtry

Problém

- Máme více dat než se nám vejde do paměti
- Chtěli bychom alespoň umět rozhodnout, zda prvek leží v seznamu
- Povolíme malou chybu odpovědi na vyhledávací dotaz

Bloom filtr

- Jestliže Bloom filtr prohlásí, že daný prvek v seznamu není, pak v seznamu doopravdy nesmí být
- Jestliže Bloom filtr prohlásí, že daný prvek v seznamu je, pak v seznamu není s pravděpodobností nejvýše p
- Pravděpodobnost se nepočítá přes volbu prvků, ale přes volbu hešovací funkce

Bloom filtry: Triviální přístup

Popis

- Máme univerzum $U = \{0, 1, \ldots, u-1\}$ všech prvků
- Chceme uložit podmnožinu $S \subseteq U$ velikosti n
- Použijeme bitové pole B velikosti $m = \left\lceil \frac{n}{p} \right\rceil$
- Uvažujeme 1-universální hešovací funkci $h : U \to [m]$
- Pro každý prvek $x \in S$ nastavíme $B[h(x)] = 1$, ostatní bity B jsou nulové

Operace Find(x)

- Jestliže $B[h(x)] = 0$, pak prvek x v seznamu S určitě není
- Jestliže $B[h(x)] = 1$, pak se může stát, že existuje prvek $y \in S$ takový, že $h(x) = h(y)$
- Pravděpodobnost, že $B[h(x)] = 1$, i když $x \notin S$, je $P[\exists y \in S : \ h(x) = h(y)] \leq p$

Paměť

Potřebujeme $\left\lceil \frac{n}{p} \right\rceil$ bitů, aby pravděpodobnost chybné odpovědi byla nejvýše p.

Bloom filtry: Vylepšení

Vylepšení

- Použijeme k bitových polí $B_1, \ldots, B_k$ velikosti m
- Pro i-té bitové pole vygenerujeme úplně nezávislou hešovací funkci $h_i : U \to [m]$
- Pro každý prvek $x \in S$ a každé i nastavíme $B_i[h_i(x)] = 1$
- Ostatní bity $B_1, \ldots, B_k$ jsou nulové

Operace Find(x)

- Jestliže existuje i takové, že $B_i[h_i(x)] = 0$, pak prvek x v seznamu S určitě není
- Za předpokladu, že $x \notin S$, odhadneme pravděpodobnost, že $B_i[h_i(x)] = 1$ pro všechna i
- $P[\forall i \exists x_i \in S : \ h_i(x) = h_i(y)] \leq \left(\frac{n}{m}\right)^k$
- Volbou $m = \lceil en \rceil$ a $k = \left\lceil \ln \frac{1}{p} \right\rceil$ dostaneme $\left(\frac{n}{m}\right)^k \leq p$

Paměť

Potřebujeme přibližně $1,88 \cdot n \log_2 \frac{1}{p}$ bitů

Bloom filtry (Bloom, 1970)

Bloom, 1970

- Použijeme úplně nezávislé hešovací funkce $h_1, \ldots, h_k : U \to [m]$ a jedno bitové pole B velikosti m
- Pro každý prvek $x \in S$ a pro každou hešovací funkci nastavíme $B[h_i(x)] = 1$
- Ostatní bity B jsou nulové

Operace Find(x)

- Jestliže existuje i takové, že $B[h_i(x)] = 0$, pak prvek x v seznamu S určitě není
- Jestliže $B[h_i(x)] = 1$ pro všechny hešovací funkce h_i, pak odpovíme, že x v seznamu S je, i když odpověď může být špatná

Analýza (bez důkazu)

- Pro dané n a m je pravděpodobnost chyby nejmenší pro $k = \frac{m}{n} \ln 2$
- Chceme-li chybu s pravděpodobností nejvýše p, pak zvolíme $k = \log_2 \frac{1}{p}$ a $m = 1,44 \cdot n \log_2 \frac{1}{p}$

Pagh, Pagh, Rao (2005)

Ukázali variantu Bloom filtrů, které stačí $n \log_2 \frac{1}{p}$ bitů

Počítací filtry (Counting filters: Fan, Cao, Broder, 2000)

Cíl

Chceme umět mazat prvky v Bloom filtru

Popis

- Místo pole bitů B použijeme pole čítačů C
- $C[j]$ je počet dvojic (x, h_i) takových, že $x \in S$ a $h_i(x) = j$
- Find: Odpovíme ano, jestliže $C[h_i(x)] > 0$ pro všechny hešovací funkce h_i
- Insert/Delete: Zvýšíme/Snížíme hodnoty čítačů $C[h_i(x)]$ o jedna pro všechny hešovací funkce h_i
- Musíme zvolit dostatečně velké čítače, aby k přetečení docházelo s dostatečně malou pravděpodobností

Volba velikostí čítačů

- Hodnotu j-tého čítače odhadneme $P[C[j] \geq s] \leq \binom{nk}{s} \frac{1}{m^s} \leq \left(\frac{enk}{sm}\right)^s$
- Volbou $k = \frac{m}{n} \ln 2$ dostáváme $P[\max_j C[j] \geq s] \leq m \left(\frac{e \ln 2}{s}\right)^s$
- Pro 4-bitový čítač dojde k přetečení pro $C[j] = 16$ s pravděpodobností $P[\max_j C[j] \geq 16] \leq 1,37 \cdot 10^{-15} m$

Bloomier filtry (Chazelle, Kilian, Rubinfeld, Tal, 2004)

Cíl

- Chceme si pamatovat r-bitové číslo $v(x)$ pro každý prvek $x \in S$
- Jestliže $x \in S$, pak operace Find musí vrátit $v(x)$
- Jestliže $x \notin S$, pak operace Find může vrátit cokoliv
- Operace Find nemusí umět poznat, jestliže x leží v S

Modifikace kukaččího hešování

- Použijeme pole r-bitových čísel R velikosti m
- a dvě úplně nezávislé hešovací funkce $h_1, h_2 : U \to [m]$
- Operace Find vrátí $R[h_1(x)] \oplus R[h_2(x)]$

Bloomier filtry (Chazelle, Kilian, Rubinfeld, Tal, 2004)

Konstrukce pole R

- Najdeme hešovací funkce h_1 a h_2 takové, že kukaččí graf neobsahuje kružnice
  - Pro $m \geq 2cn$, kde $c > 2$, jsme dokázali, že očekávaný počet pokusů k nalezení takových hešovacích funkcí je $\mathcal{O}(1)$
- V každé komponentě zvolíme jeden vrchol j, kterému nastavíme $R[j] = 0$
- Strom projdeme do hloubky z vrcholu j
- Pro každou hranu odpovídající prvku x, jestliže pozice $R[h_1(x)]$ je již nastavena, pak nastavíme $R[h_2(x)] = R[h_1(x)] \oplus v(x)$

Analýza

- Paměť: $4nr$ bitů (lze vylepšit použitím většího počtu hešovacích funkcí)
- Složitost operace Find: $\mathcal{O}(1)$ v nejhorším případě
- Očekávaná složitost operace Build: $\mathcal{O}(n)$ pokud umíme s prvky U pracovat v konstantním čase
- Paměťová složitost operace Build: $\mathcal{O}(n(r + \log n))$ bitů

Obsah

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

[2] 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.

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

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

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

[6] Michael L Fredman, János Komlós, and Endre Szemerédi.
Storing a sparse table with 0 (1) worst case access time.
Journal of the ACM (JACM), 31(3):538–544, 1984.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

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

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

[17] 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.

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

[19] Mihai Patraşcu.
Better guarantees for chaining and linear probing.
http://infoweekly.blogspot.cz/2010/02/better-guarantees-for-chaining-and.html.
blogspot, February 2, 2010.

[20] Mihai Pătraşcu 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.

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

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

[23] 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.

[24] Daniel D Sleator and Robert E Tarjan.
Amortized efficiency of list update and paging rules.
Communications of the ACM, 28(2):202–208, 1985.

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

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