Datové struktury I

NTIN066

Jirka Fink

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

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

Zimní semestr 2018/19

Poslední změna 17. října 2018

Základní informace

Kontant

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

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

Konzultace Individuální domluva

Cíle předmětu

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

Obsah

Zápočet: Domácí úkoly

Podmínky

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

Nové předměty

Motivace

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

Analýza datových struktur (NTIN105)

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

Nové předměty

Implementace datových struktur (NTIN106)

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

Obsah

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

Literatura

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

Obsah

Amortizovaná analýza

Motivace

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

Metody výpočtu amortizované složitosti

- Agregovaná analýza
- Účetní metoda
- Potenciální metoda

Obsah

Inkrementace binárního čítače: Agregovaná analýza

Binární čítač

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

Agregovaná analýza

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

Inkrementace binárního čítače: Účetní metoda

Účetní metoda

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

Potenciální metoda

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

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

protože $0 \leq \Phi_j \leq n$ ④

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

Definice

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

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

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

Příklad: Inkrementace binárního čítače

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

Obsah

Dynamické pole

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

Intuitivní přístup ②

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

Agregovaná analýza: Celkový čas

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

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

Potenciální metoda

- Uvažujme potenciál

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

  a tyto tři body rozšíříme po částech lineární funkcí
- Explicitně

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

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

1

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

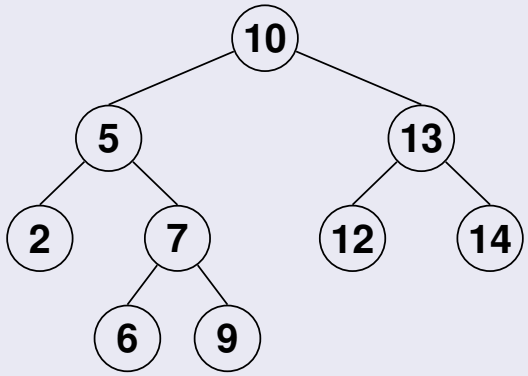
Obsah

Binární vyhledávací strom

Vlastnosti

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

Příklad



Složitost

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

Obsah

Váhově vyvážené stromy: BB[α]-strom

BB[α]-strom (Nievergelt, Reingold [10])

- 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
- Zřejmě musí platit $\frac{1}{2} < \alpha < 1$ 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.

Operace INSERT (DELETE je analogický)

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

Amortizovaná časová složitost operací INSERT a DELETE: Agregovaná metoda

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

1. Při hledání listu pro nový vrchol stačí na cestě od kořene k listu kontrolovat, zda se přidáním vrcholu do podstromu syna neporuší vyvažovací podmínka. Pokud se v nějakém vrcholu podmínka poruší, tak se hledání ukončí a celý podstrom včetně nového prvku znovu vybuduje.
2. Existují pravidla pro rotování BB[α]-stromů, ale ta se nám dnes nehodí.
3. Operace BUILD podstromu vrcholu u trvá $\mathcal{O}(s_u)$ a mezi 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 $\mathcal{O}(\log n)$
- Potenciál vrcholu u definován

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

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

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

Obsah

Staticky optimální strom

Cíl

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

Formálně

Máme prvky $x_1, \ldots, x_n$ 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)

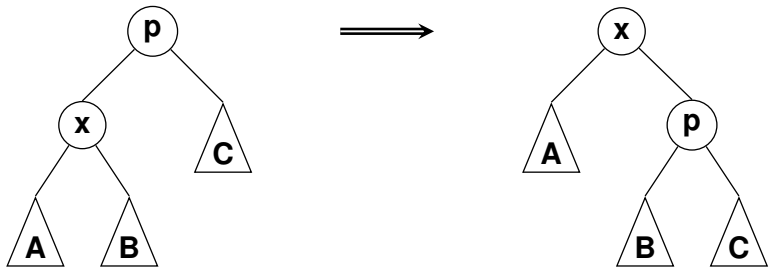
- $\mathcal{O}(n^3)$ – triviálně dynamickým programováním
- $\mathcal{O}(n^2)$ – vylepšené dynamické programování (Knuth [7])

Jak postupovat, když neznáme váhy předem?

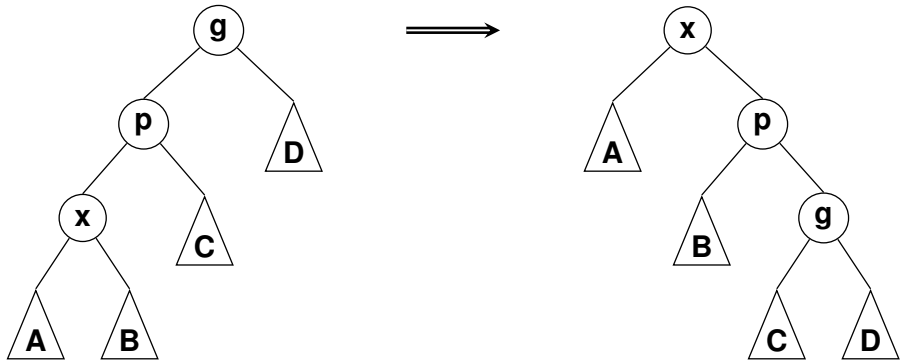
- Pomocí rotací bude udržovat často vyhledávané prvky blízko kořene
- Operací SPLAY „rotujeme“ zadaný prvek až do kořene
- Operace FIND vždy volá SPLAY na hledaný prvek

Splay strom (Sleator, Tarjan [12]): Operace SPLAY prvku *x*

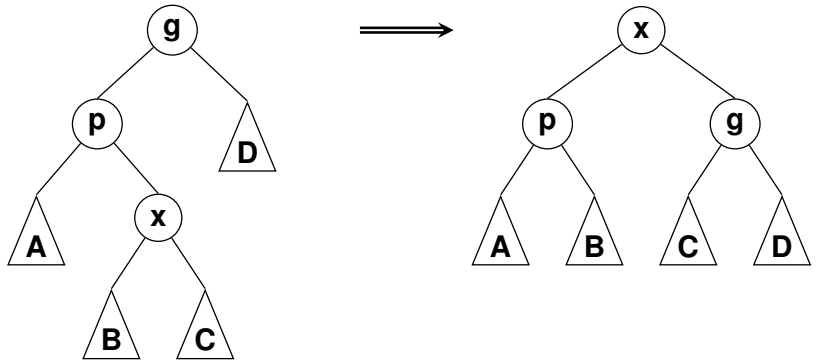
- Zig rotace: Otec *p* prvku *x* je kořen



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



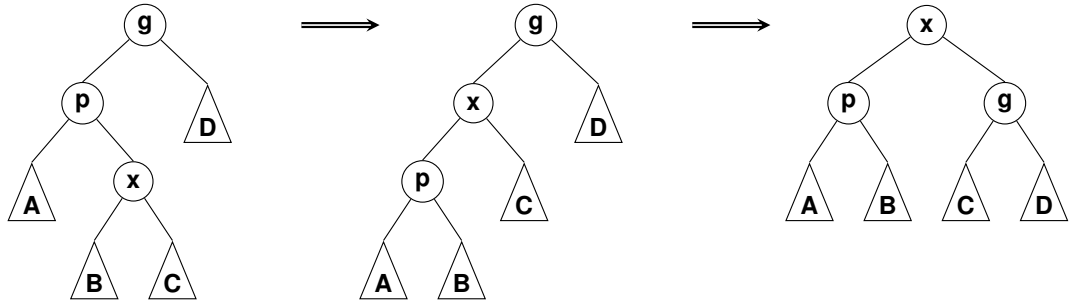
- Zig-zag rotace: *x* je pravý syn a *p* je levý syn nebo opačně



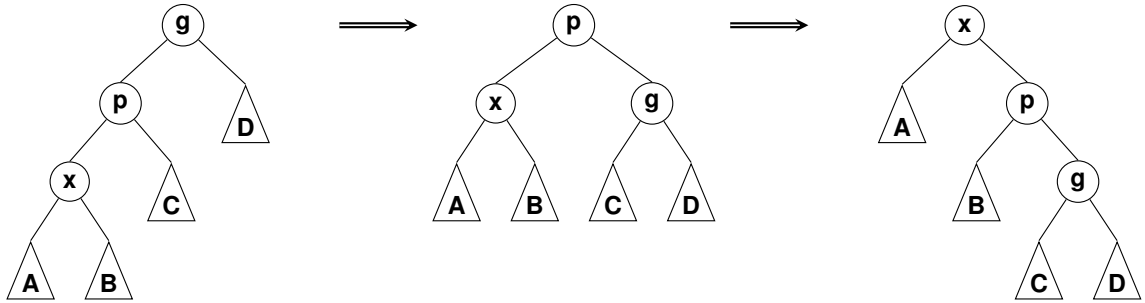
Uvažujeme *buttom-up* verzi, tj. prvek x nejprve najdeme a poté jej postupně rotujeme nahoru, což znamená, že x vždy značí stejný vrchol postupně se přesouvající ke kořeni a ostatní vrcholy stromu jsou sousedé odpovídající dané rotaci.
Existuje též *top-down* verze [8], která vždy rotuje vnuka kořene, jehož podstrom obsahuje prvek x. Tato verze je sice v praxi rychlejší, ale postup a analýza jsou složitější.

Splay strom: Operace SPLAY prvku *x*

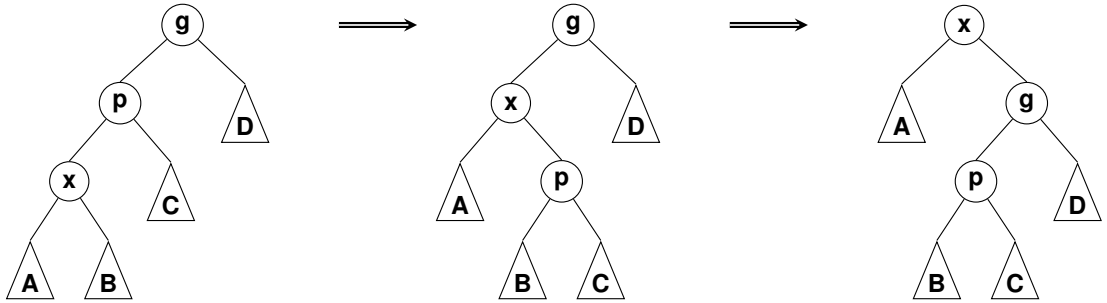
- Zig-zag rotace jsou pouze dvě jednoduché rotace prvku *x* s aktuálním otcem



- Zig-zig rotace jsou taky dvě rotace,



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



Splay strom: Amortizovaná analýza

Lemma

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

Důkaz

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

Značení

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

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

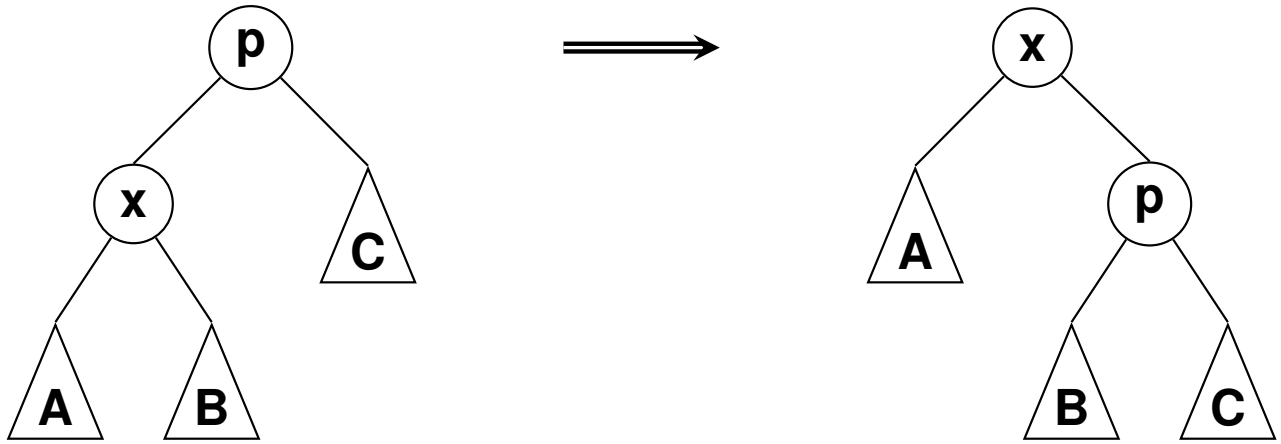
$$f\left(\frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}\right) \leq \frac{\sum_{i=1}^{n} w_i f(x_i)}{\sum_{i=1}^{n} w_i}.$$

Jelikož funkce log je rostoucí a konkávní, dostáváme

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

z čehož plyne znění lemmatu.

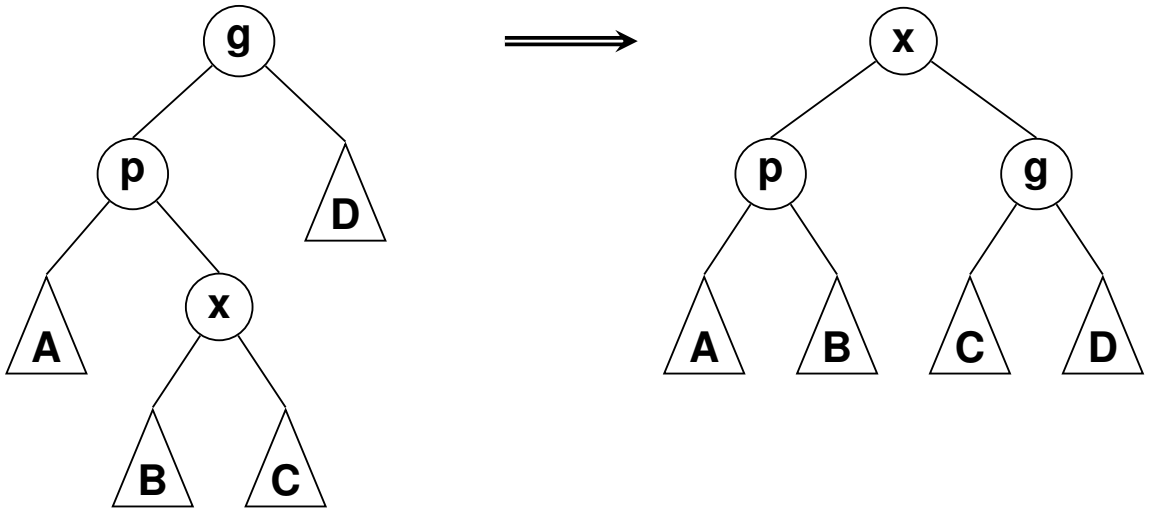
Splay strom: Zig rotace



Analýza

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

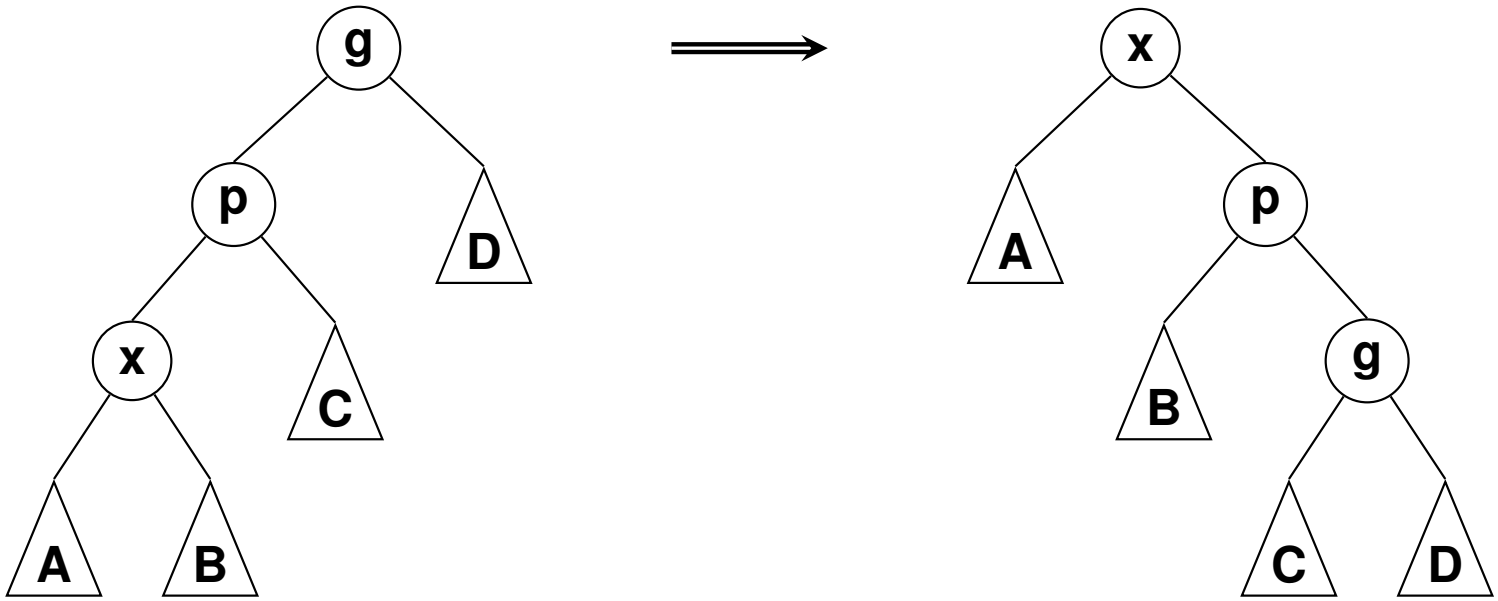
Splay strom: Zig-zag rotace



Analýza

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

Splay strom: Zig-zig rotace



Analýza

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

Splay strom: Analýza

Amortizovaný čas

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

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

$$\leq 1 + 3(\Phi_{\text{konec}}(x) - \Phi_0(x)) \quad ②$$

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

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

Skutečný čas k operací SPLAY

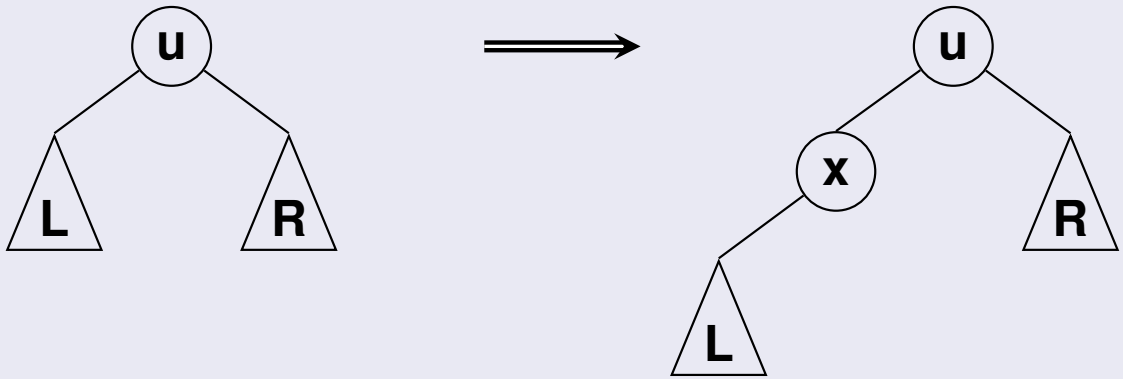
- Potenciál vždy splňuje $0 \leq \Phi \leq n \log_2 n$
- Rozdíl mezi konečným a počátečním potenciálem je nejvýše $n \log_2 n$
- Celkový čas k operací SPLAY je $\mathcal{O}((n + k) \log n)$

1. T značí skutečný čas rotace, což je počet jednoduchých rotací k provedení rotace zig, zigzig nebo zigzag.
2. 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

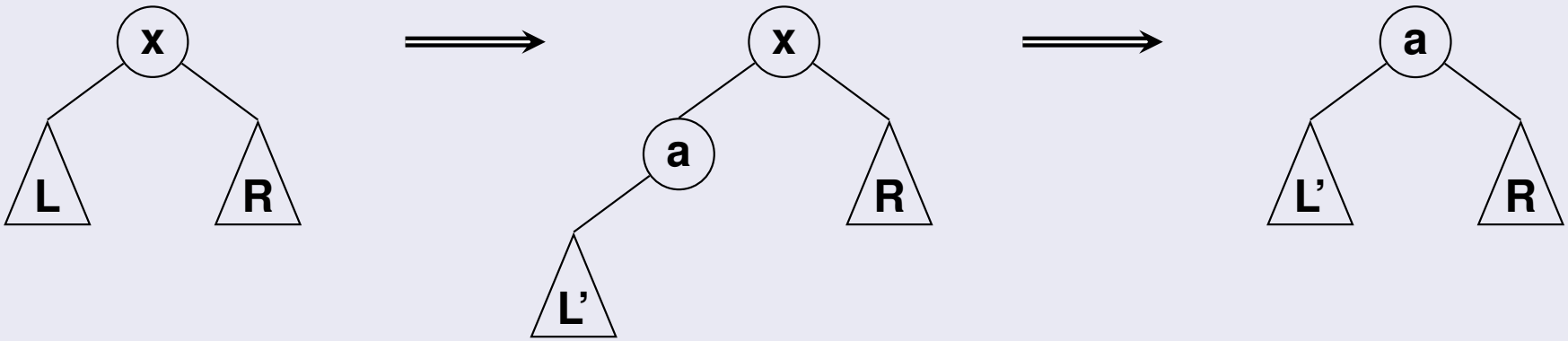
- Operace FIND a SPLAY: $\mathcal{O}(\log n)$
- Vložením nového vrcholu potenciál Φ vzroste nejvýše o $\Phi'(x) + \Phi'(u) \leq 2 \log_2 n$
- Amortizovaná složitost operace INSERT je $\mathcal{O}(\log n)$

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



Další vlastnosti splay stromů

Věta (vyhledávání prvků v rostoucím pořadí)

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

Věta (statická optimalita)

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

Hypotéza (dynamická optimalita)

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

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

Splay stromy: Výhody a aplikace

Výhody a nevýhody Splay stromů

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

Aplikace

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

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

1. domácí úkol: Splay stromy

Stručné zadání

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

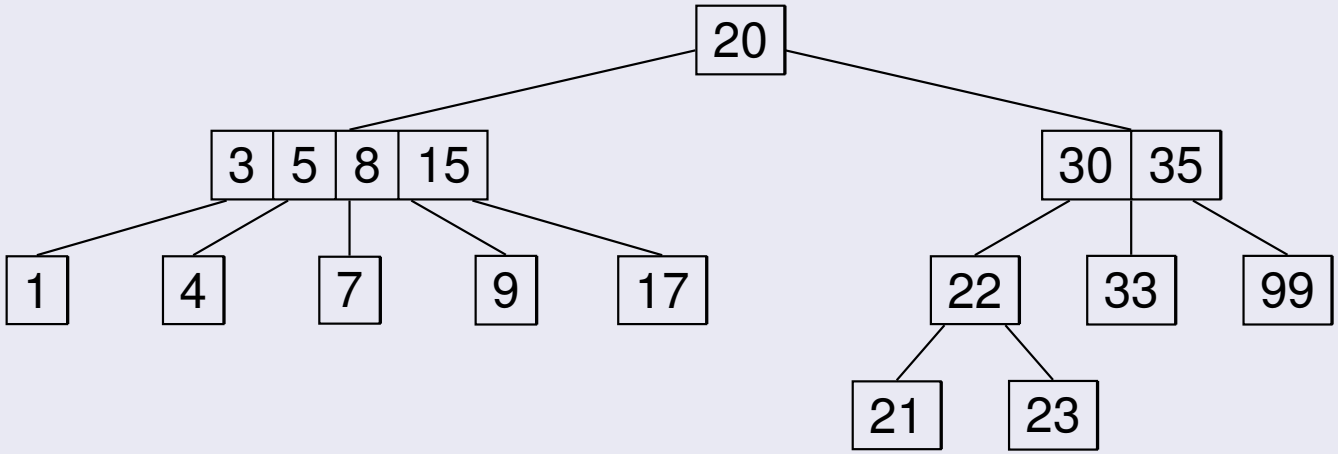
Obsah

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

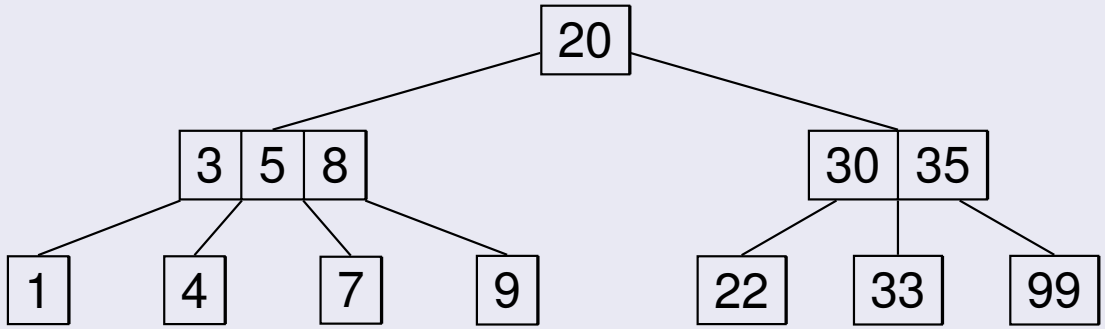


(a,b)-strom (Bayer, McCreight [1])

Vlastnosti

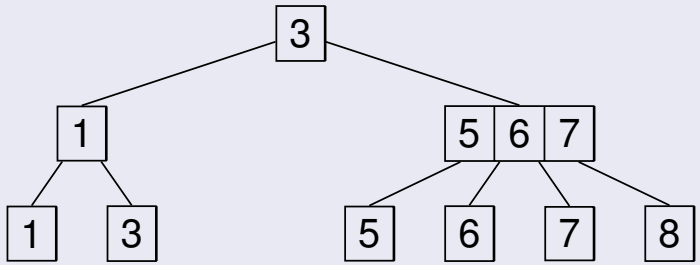
- a, b jsou celá čísla splňující $a \geq 2$ a $b \geq 2a - 1$
- (a,b)-strom je vyhledávací strom
- Všechny vnitřní vrcholy kromě kořene mají alespoň a synů a nejvýše b synů
- Kořen má nejvýše b synů
- Všechny listy jsou ve stejné výšce
- Pro zjednodušení uvažujeme, že prvky jsou jen v listech

Příklad: (2,4)-strom

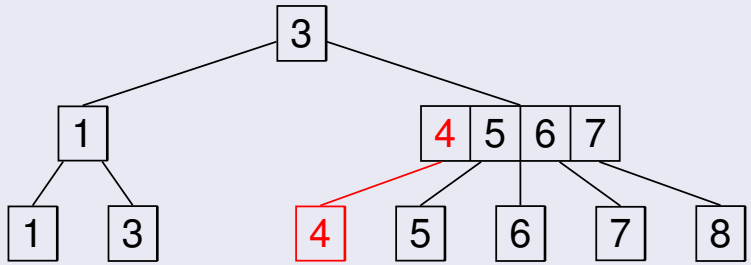


(a,b)-strom: Operace Insert

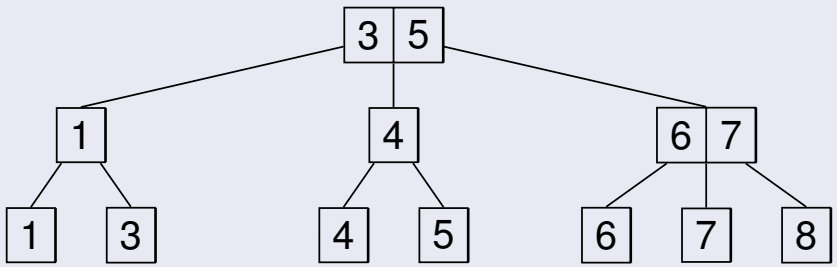
Vložte prvek s klíčem 4 do následujícího (2,4)-stromu



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 $v \leftarrow u$

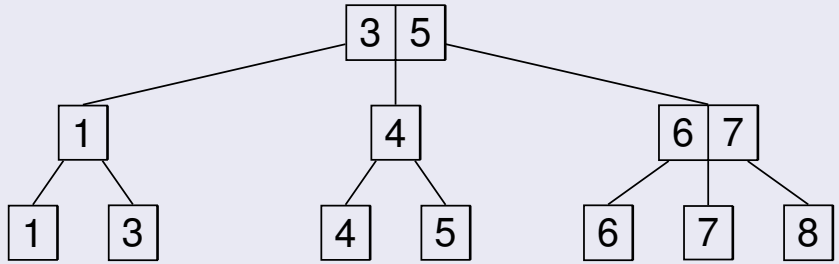
Časová složitost

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

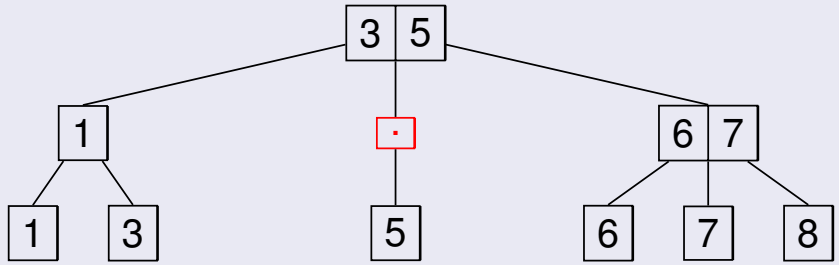
Musíme ještě dokázat, že po provedení všech operací doopravdy dostaneme (a,b)-strom. Ověříme, že rozdělené vrcholy mají alespoň a synů (ostatní požadavky jsou triviální). Rozdělovaný vrchol má na počátku právě $b+1$ synů a počet synů po rozdělení je $\lfloor \frac{b+1}{2} \rfloor$ a $\lceil \frac{b+1}{2}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

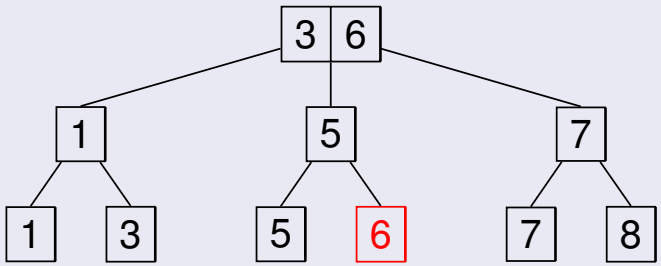
Smažte prvek s klíčem 4 z následujícího (2,4)-stromu

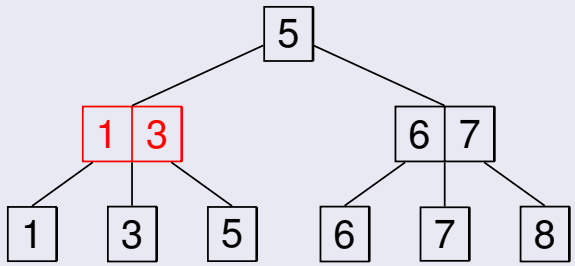


Nalezneme a smažeme list



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

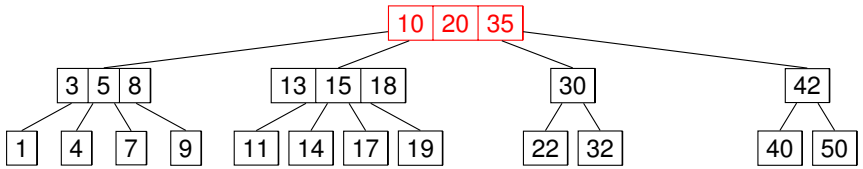
Preventivně rozdělit každý vrchol na cestě od kořene k hledanému listu s b syny na dva vrcholu.

Operace Delete

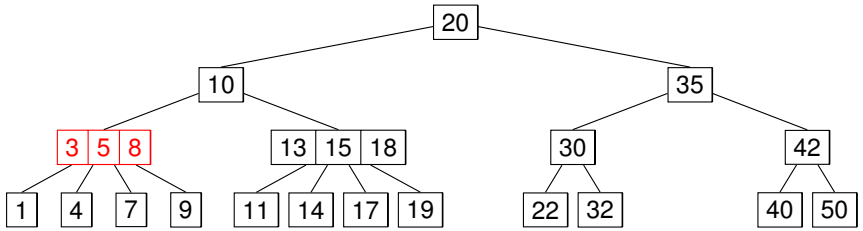
Preventivně sloučit každý vrchol na cestě od kořene k hledanému listu s a syny s bratrem nebo přesunout synovce.

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

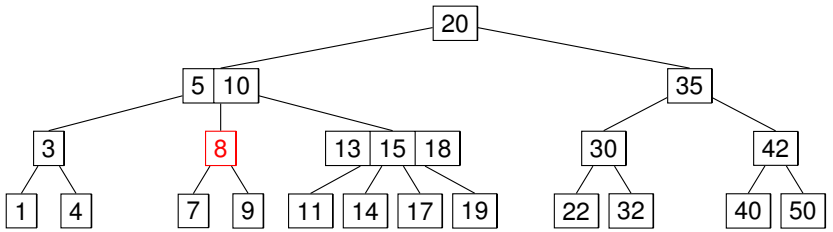
- Vložte prvek s klíčem 6 do následujícího (2,4)-stromu



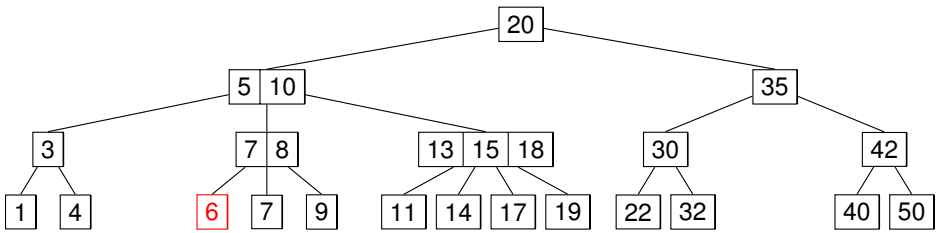
- Nejprve rozdělíme kořen



- 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 [4])

Cíl

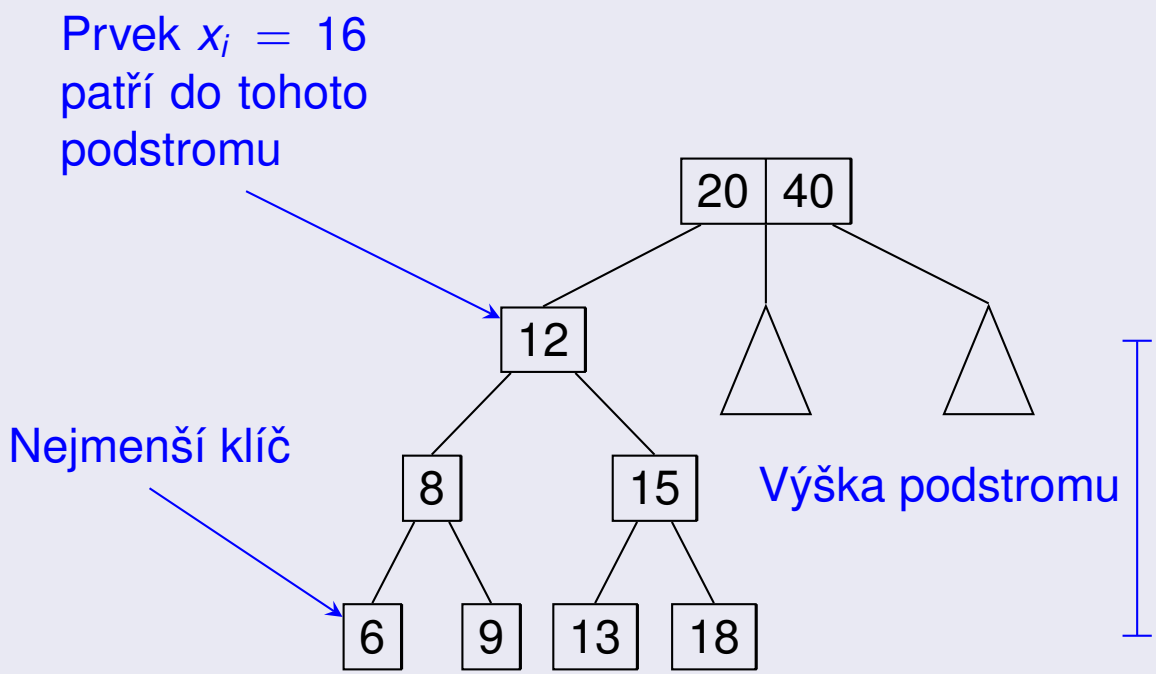
Setřídit „skoro“ setříděné pole

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 [9])

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
   $\sum_i \log f_i = \log \prod_i f_i = n \log \sqrt[n]{\prod_i f_i} \leq n \log \frac{\sum_i f_i}{n} = n \log \frac{F}{n}$. ①
5. Rozdělování vrcholů v průběhu všech operací Insert: $\mathcal{O}(n)$
6. Celková složitost: $\mathcal{O}(n + n \log(F/n))$
7. Složitost v nejhorším případě: $\mathcal{O}(n \log n)$ protože $F \leq \binom{n}{2}$
8. Jestliže $F \leq n \log n$, pak složitost je $\mathcal{O}(n \log \log n)$ ②

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

(a,b)-strom: Závěr

Počet modifikovaných vrcholů při operací Insert a Delete [6]

- 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

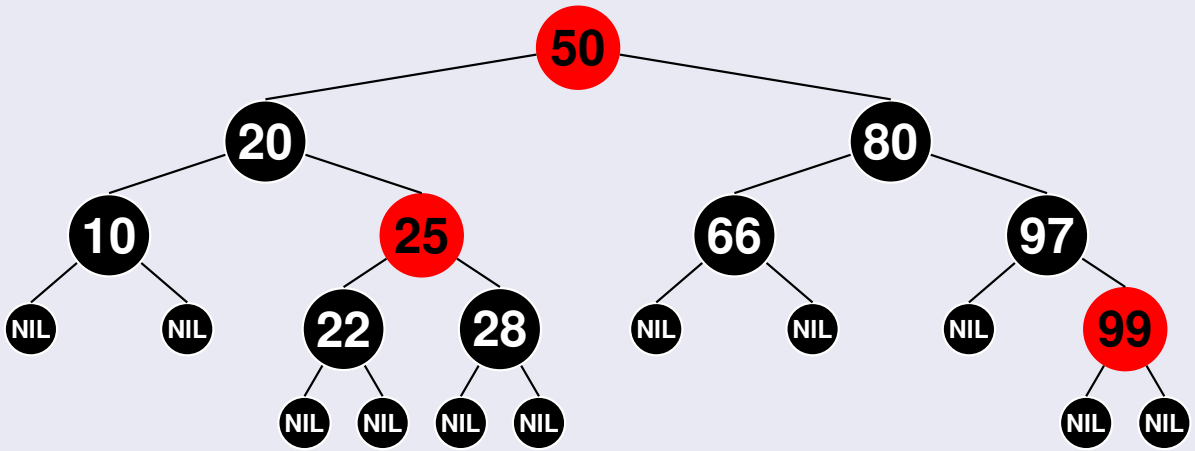
Obsah

Červeno-černé stromy (Guibas, Sedgewick [5])

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

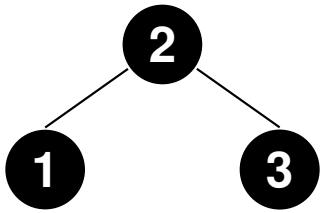
Příklad

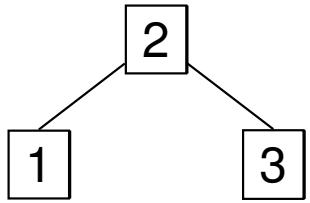


1. Nepovinná podmínka, která jen zjednodušuje operace. V příkladu uvažujeme, že listy jsou reprezentovány NIL/NULL ukazately, a tedy imaginární vrcholy bez prvků.
   Někdy se též vyžaduje, aby kořen byl černý, ale tato podmínka není nutná, protože kořen můžeme vždy přebarvit na černo bez porušení ostatních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

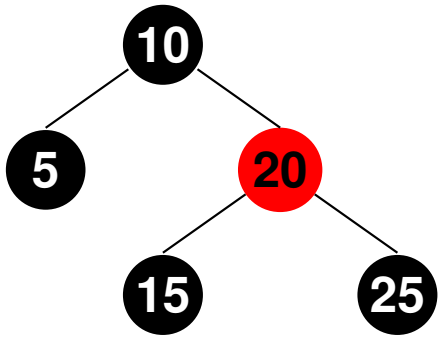
Červeno-černé stromy: Ekvivalence s (2,4)-stromy

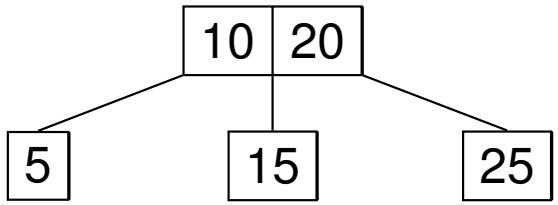
- Vrchol bez červených synů



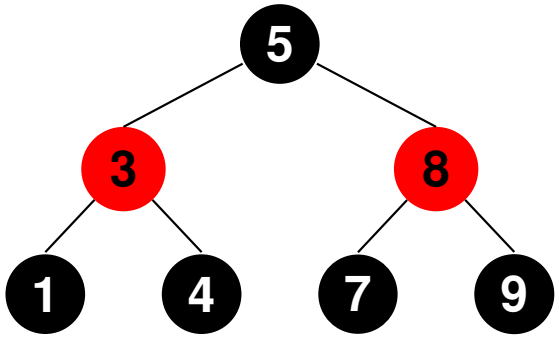


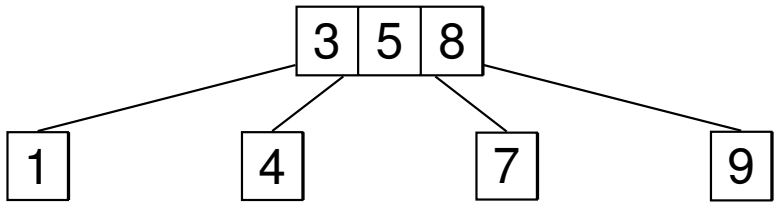
- Vrchol s jedním čevneným synem ①





- Vrchol s dvěma červenými syny



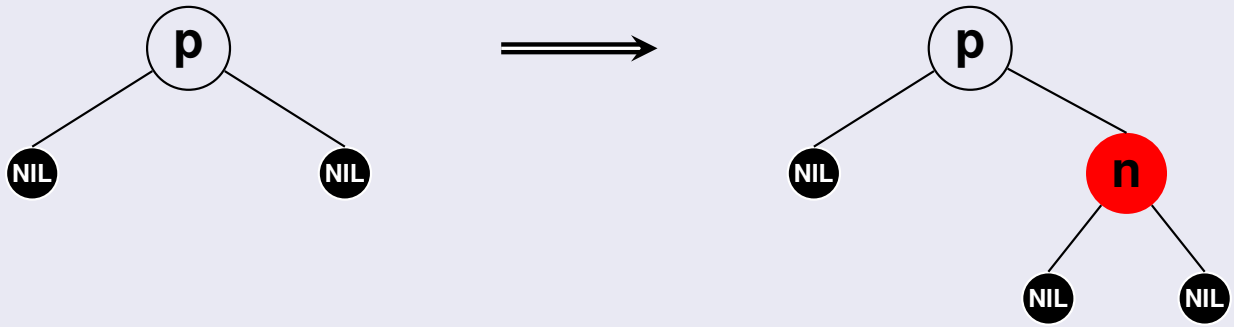


1. Převod mezi červeno-černými stromy a (2,4)-stromy není jednoznačný, protože vrchol (2,4)-stromu se třemi syny a prvky $x < y$ lze převést na černý vrchol červeno-černého stromu s prvkem x a pravým červeným synem y nebo s prvkem y a levým červeným synem x.

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

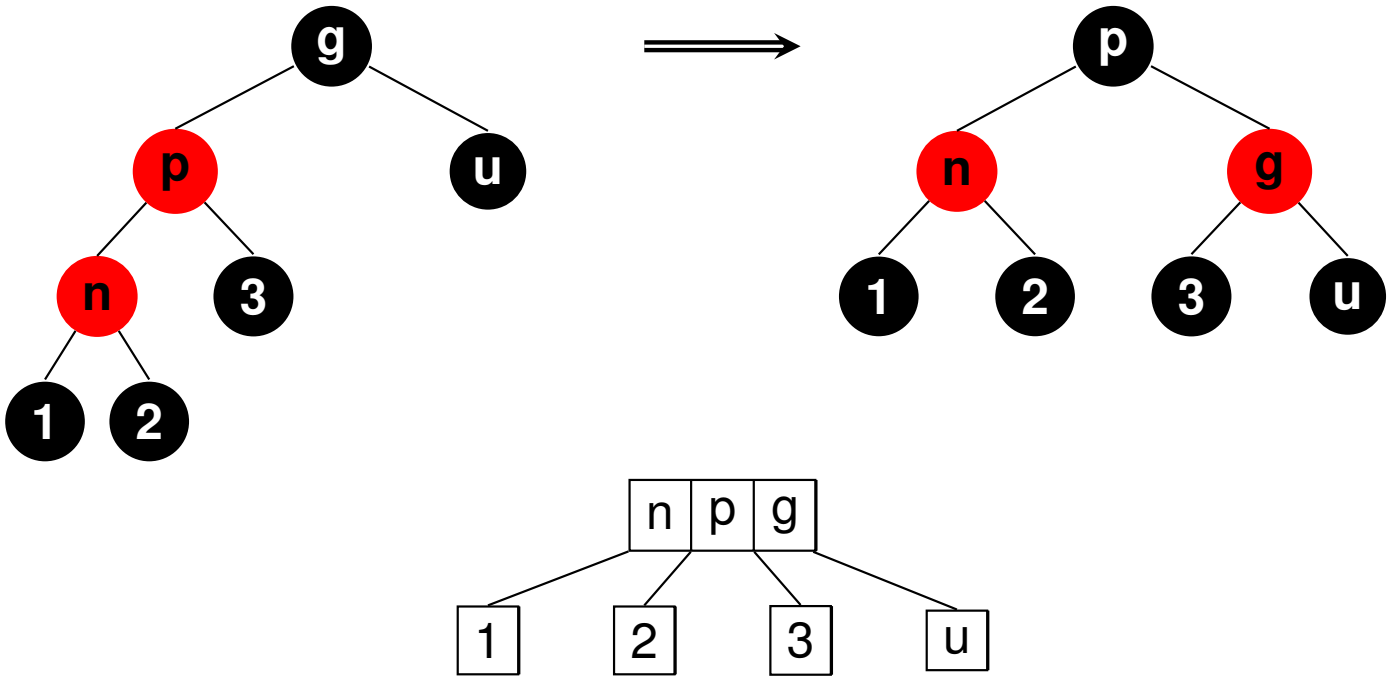
- Vrchol *n* a jeho otec *p* jsou červené vrcholy a toto je jediná porušená podmínka
- Děda *g* vrcholu *n* je černý

Musíme uvažovat tyto případy:

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

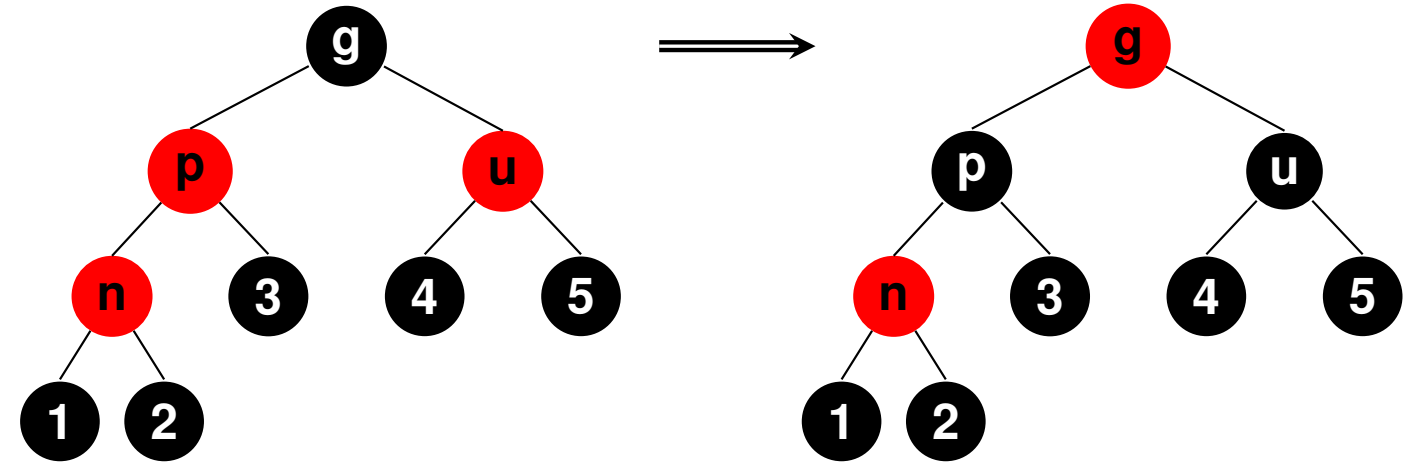
1. S využitím symetrií lze počet případů snížit.

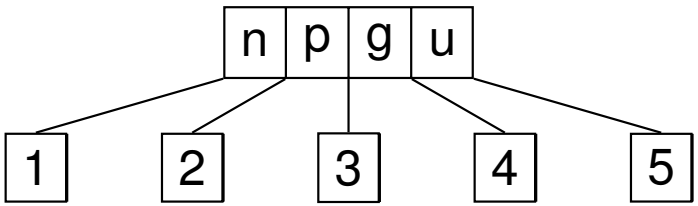
Červeno-černé stromy: Operace Insert, strýc je černý

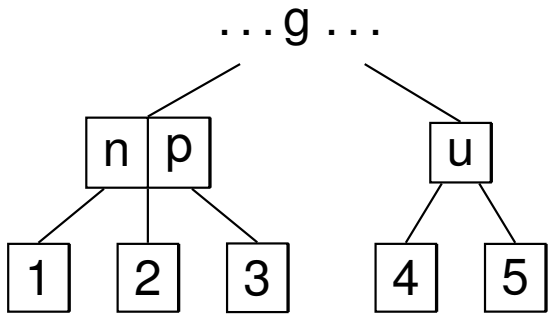


Pořadí prvků v (2,4)-stromu a výsledný červeno-černý strom závisí na tom, zda vrchol *n* je pravým nebo levým synem *p* a zda vrchol *p* je pravým nebo levým synem *g*.

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.

Obsah

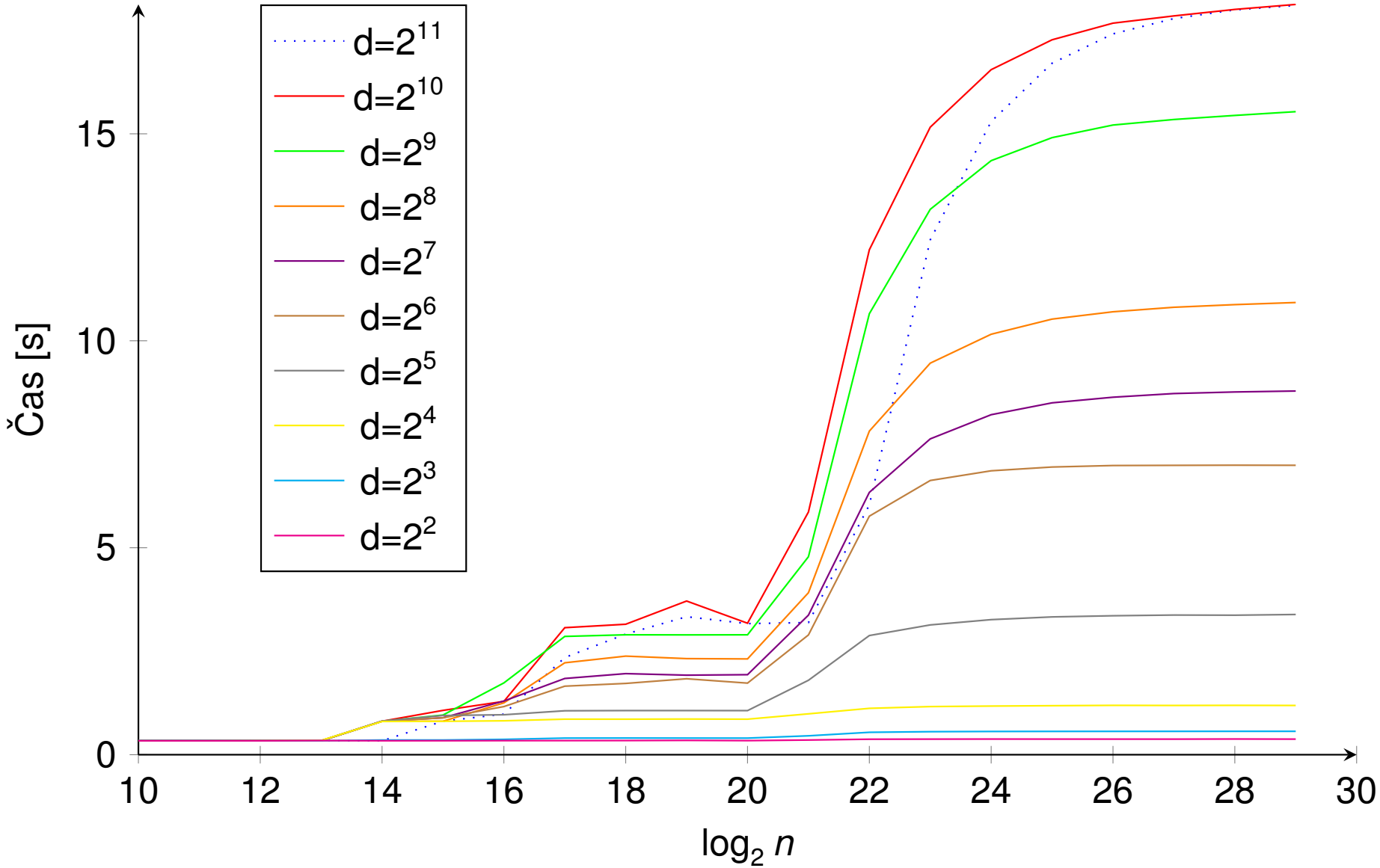
Příklad velikostí a rychlostí různých typů pamětí

	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

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, i=0
# Měříme dobu průběhu cyklu v závislosti na parametrech n a d
# Počet operací je nezávislý na 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 [3])

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).

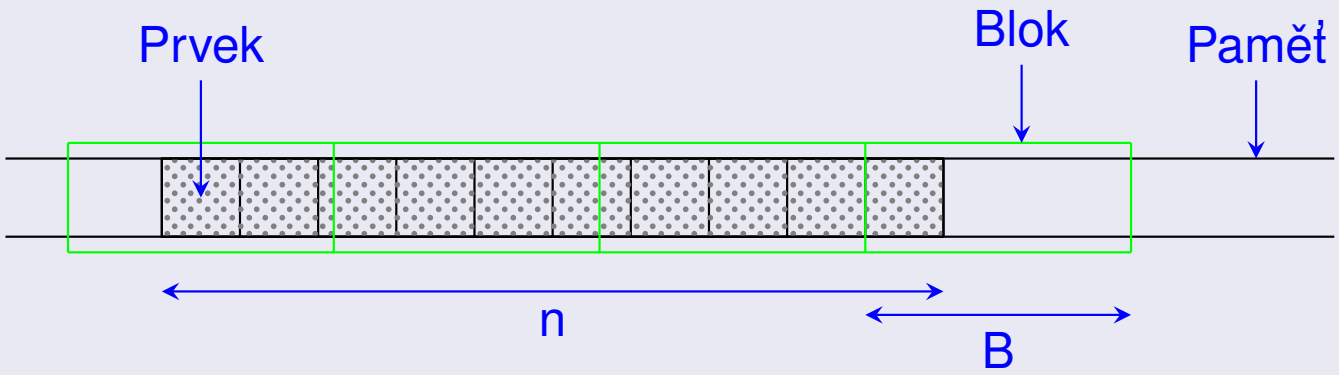
Cache-oblivious algoritmus

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`.

Přečtení souvislého pole (výpočet maxima, součtu a podobně)



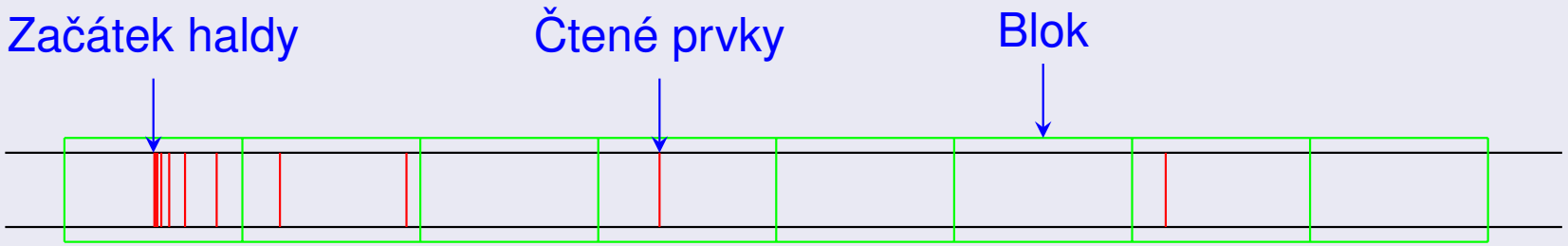
- Minimální možný počet přenesených bloků je $\lceil n/B \rceil$.
- Skutečný počet přenesených bloků je nejvýše $\lceil n/B \rceil + 1$.
- Předpokládáme, že máme k dispozici $\mathcal{O}(1)$ registrů k uložení iterátoru a maxima.

Obrácení pole

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

Cache-oblivious analýza: Binární halda a vyhledávání

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

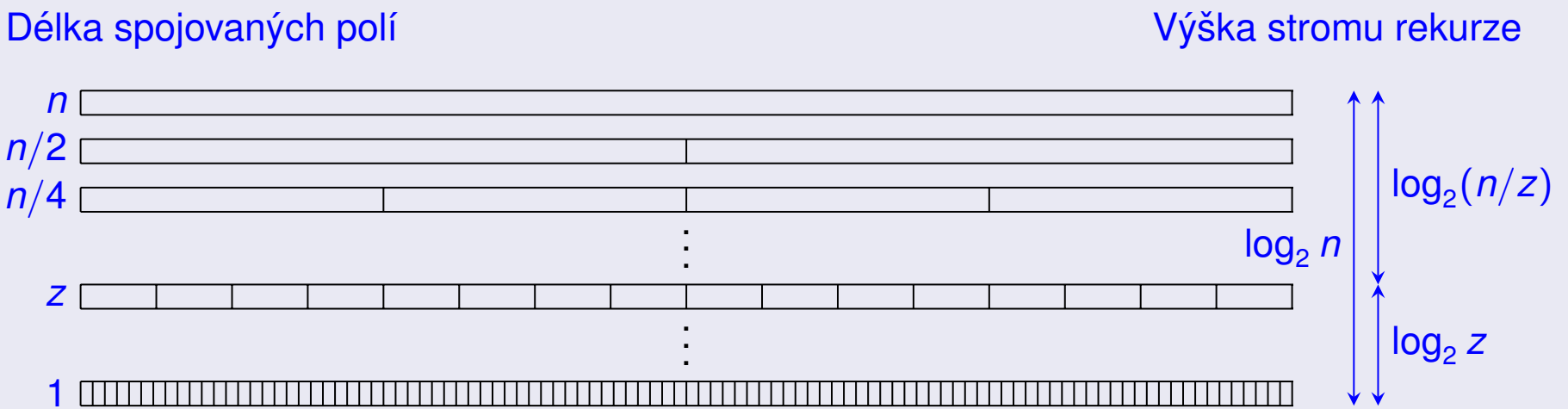
1. Přesněji $\Theta(\max\{1, \log n - \log B\})$. Dále předpokládáme, že $n \geq B$.
2. Pro jednoduchost uvažujeme neúspěšné vyhledávání.

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 $k \times k$

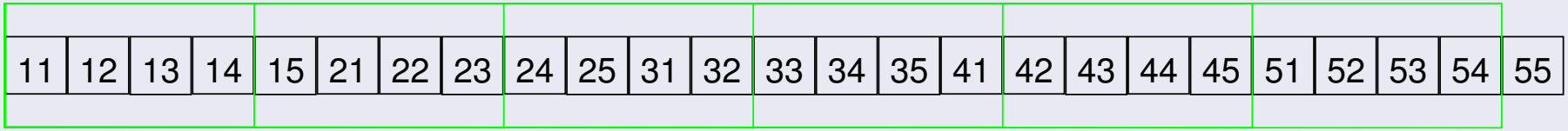
```
1 for i ← 1 to k do
2     for j ← i + 1 to k do
3         Swap(A_ij, A_ji)
```

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



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

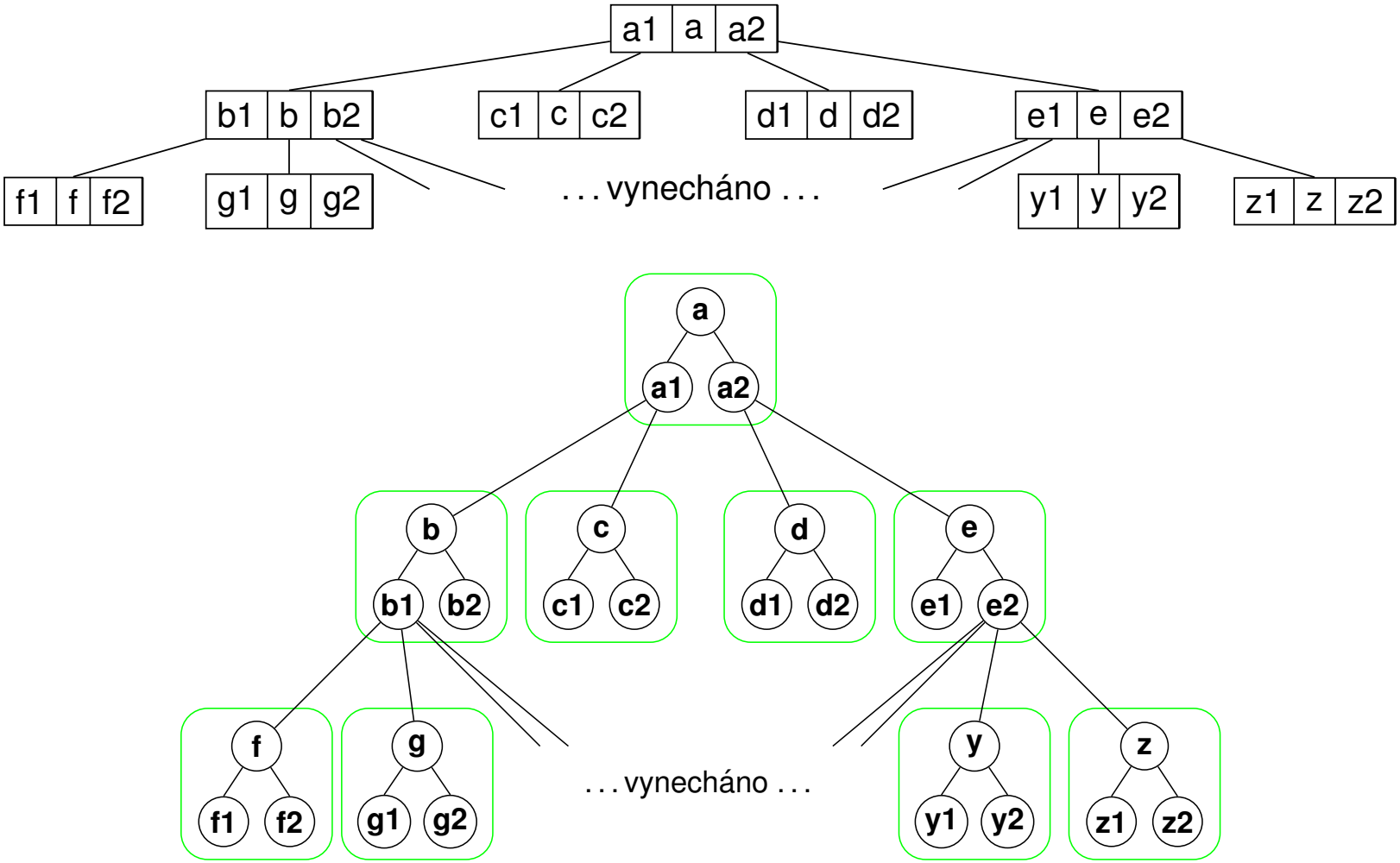
- Výška stromu je $\log_B(n) + \Theta(1)$ ①
- Jeden vrchol je uložen v nejvýše dvou blocích
- Počet načtených bloků je $\Theta(\log_B(n))$ ②
- Nevýhody: cache-aware a chtěli jsme binární strom

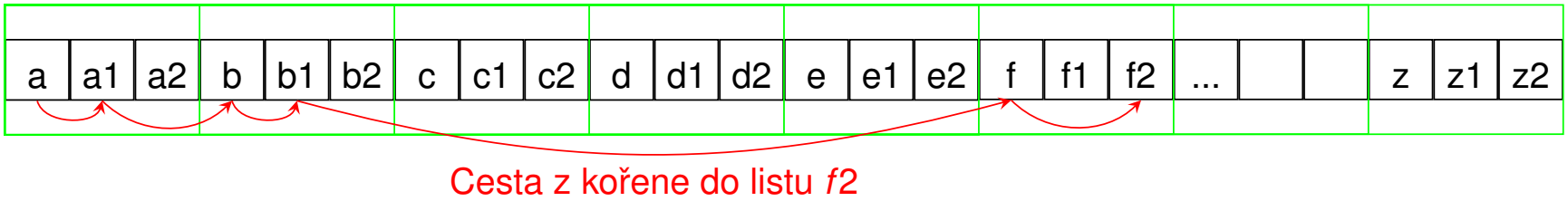
Převedení na binární strom

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



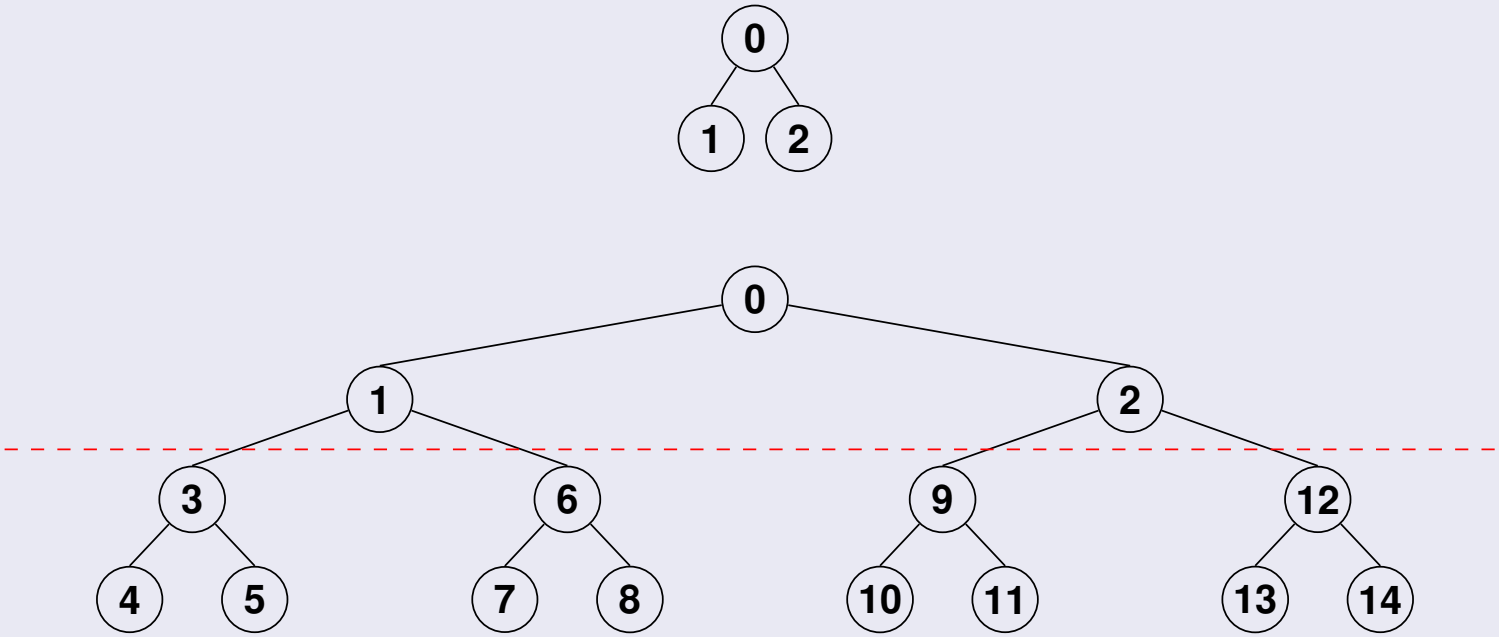


Cache-oblivious analýza: Reprezentace binárních stromů

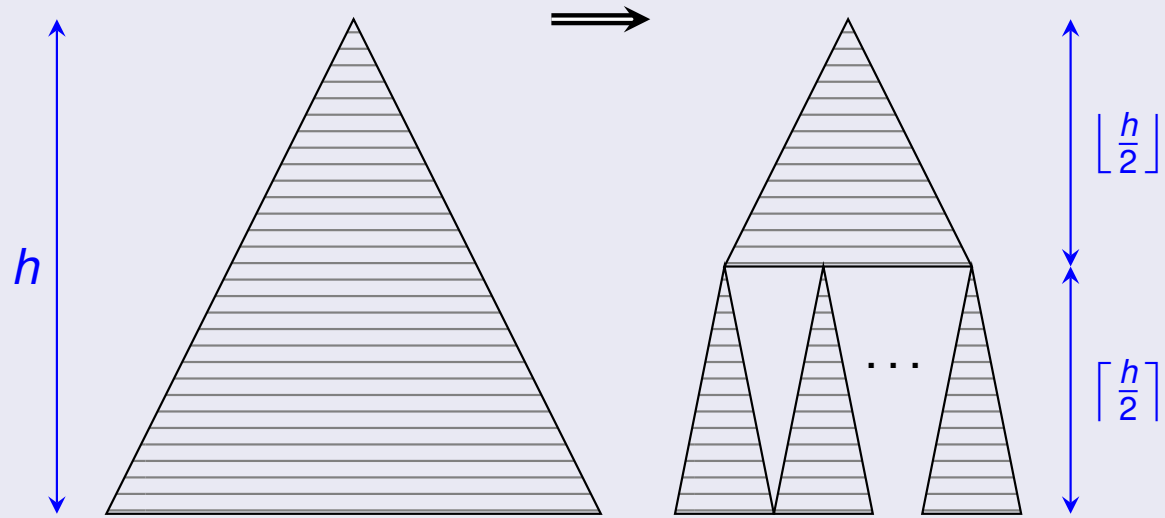
Rekurzivní „bottom-up“ konstrukce van Emde Boas rozložení

- 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 [11])

- 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

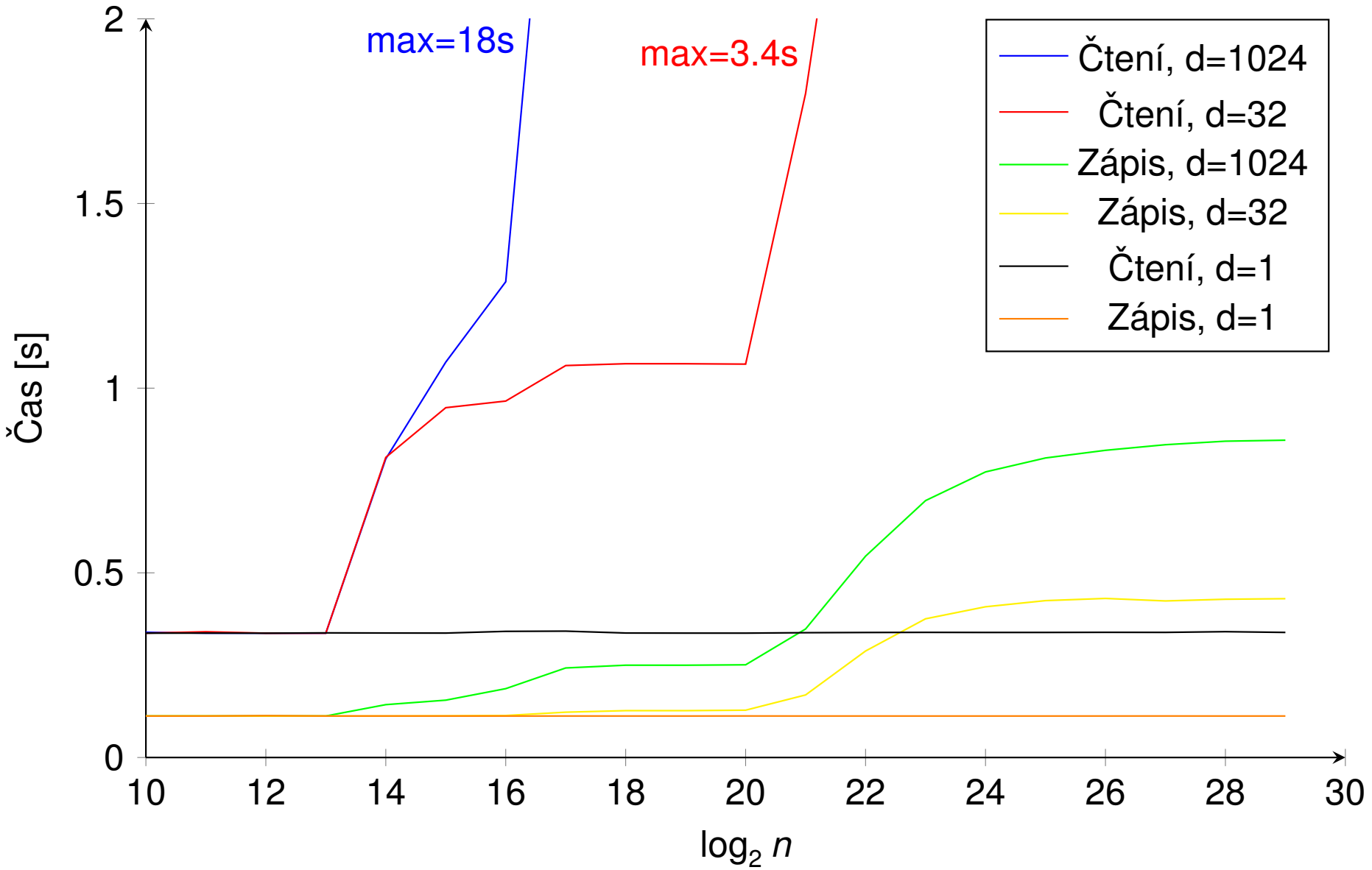
Čtení z paměti

```
# 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

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

Srovnání rychlosti čtení a zápisu z paměti



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);
```

Obsah

Obsah

Obsah

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] 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.

[4] 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.

[5] 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.

[6] Scott Huddleston and Kurt Mehlhorn.
A new data structure for representing sorted lists.

Acta informatica, 17(2):157–184, 1982.

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

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

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

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

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

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