Datové struktury I

1. přednáška: Úvod do amortizované složitosti

Jirka Fink

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

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

Zimní semestr 2022/23

Základní informace

Kontakt

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

Cíle předmětu

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

Literatura

- M. Mareš: Lecture notes on data structures, 2019.
- M. Mareš, T. Valla: Průvodce labyrintem algoritmů, CZ.NIC, 2017.
- 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.
- Záznamy z přednášek Martina Mareše a Petra Gregora (na SISu)

Další moje předměty

Implementace algoritmů a datových struktur (NTIN106)

- Naučit studenty efektivně implementovat datové struktury v rozumném čase bez únavného hledání chyb
- Předmět je praktický — zápočet je za implementaci 3 algoritmů nebo datových struktur
- Rozvrh: napište e-mail s časovými možnostmi

Large Scale Optimization

- Postupy řešení algoritmicky těžkých optimalizačních úloh
- Dva předměty, které se v letním semestru střídají každé dva roky:
  - Exaktní metody (NOPT059)
  - Metaheuristiky (NOPT061)
- Rozvrh: napište e-mail s časovými možnostmi

Bakalářské a diplomové práce

- Různá témata a umělé inteligence a optimalizace
- Zadání po individuální domluvě

Obsah

1. Amortizovaná složitost: binární čítač, dynamické pole
2. Stromy: BB(α)-stromy, splay stromy, (a,b)-stromy
3. Paměťová hierarchie: transpozice matic, merge sort
4. Hešování: volba hešovací funkce a řešení kolizí
5. Bloom filtry
6. Suffixové pole
7. Geometrické datové struktury: intervalové stromy
8. Paralelní datové struktury nepotřebující zámky

Jaké datové struktury znáte?

- Spojový seznam
- Pole
- Fronta, zásobník
- Vyhledávací stromy
- Hešovací tabulky
- Binární halda

Model výpočtu: Random Access Machine

Popis

- Paměť je rozdělená do bloků, které jsou indexovány čísly 1, 2, ...
- V každém bloku je uloženo přirozené číslo velikosti nejvýše $poly(n)$
- Význam hodnoty v bloku: klíč nebo hodnota prvku, adresa a další pomocné proměnné
- Instrukční sada: +, -, *, celočíselné dělení a modulo
- Argumenty instrukcí: konstanta, číslo bloku, blok obsahující číslo bloku

Složitost

- Paměťová: největší index použitého bloku
- Časová: počet instrukcí

Amortizovaná analýza

Motivace

- Uvažujme datovou struktury, která zvládá nějakou operaci většinou velmi rychle
- Ale občas potřebuje reorganizovat svoji vnitřní strukturu, 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“ dobu výpočtu posloupnosti operací, případně celkovou složitost posloupnosti operací

Metody výpočtu amortizované složitosti

- Agregovaná analýza
- Účetní metoda
- Potenciální metoda

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 $T_j + (\Phi_j - \Phi_{j-1}) \geq 2$
- Tedy můžeme zvolit $A = 2$
- Celkový počet změněných bitů při k operacích je

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

protože $0 \leq \Phi_j \leq n$ ④

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

Potenciální metoda

Definice

Potenciál Φ je funkce, která každý stav datové struktury ohodnotí nezáporným reálným číslem. Operace nad datovou strukturou má amortizovanou složitost A, jestliže vykonání libovolné 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 A je 2
- Platí $T \leq A + (\Phi(S) - \Phi(S'))$

Zjednodušený zápis

$\Phi := \Phi(S)$ a $\Phi' := \Phi(S')$

Dynamické pole

Dynamické pole

- Máme zásobník (prvky přidáváme na konec a z konce i mažeme)
- 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

Intuitivní přístup 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 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 $i \geq 2$ ③
- 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 další realokaci.
3. 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 $T + (\Phi' - \Phi) \leq 2$ 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.
$$\Phi' - \Phi = \begin{cases} 2 & \text{pokud přidáváme a } p \leq 2n \\ -2 & \text{pokud mažeme a } p \leq 2n \\ -1 & \text{pokud přidáváme a } p \geq 2n \\ 1 & \text{pokud mažeme a } p \geq 2n \end{cases}$$

2. Jestliže nedojde k realokaci, pak $T = 0$.
Při realokaci musíme nejprve zkopírovat všechny prvky, kterých je $T = \Phi$, čímž potenciál klesne na 0. Poté přidáme/smažeme prvek, čímž potenciál vzroste nejvýše na $\Phi' \leq 2$.

Příští týden

- Líně vyvažované stromy (BB(α)-stromy)
- Samovyvažovací stromy (Splay stromy)

Datové struktury I

2. přednáška: BB[α]-stromy, Splay stromy

Jirka Fink

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

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

Zimní semestr 2022/23

Dynamické pole

Dynamické pole

- Máme zásobník s n prvky uložený v poli velikosti 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
- Jaká je složitost přidávání a mazání prvků?

Intuitivní přístup 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 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 $i \geq 2$ ③
- 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 další realokaci.
3. 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 $T + (\Phi' - \Phi) \leq 2$ 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. $$\Phi' - \Phi = \begin{cases} 2 & \text{pokud přidáváme a } p \leq 2n \\ -2 & \text{pokud mažeme a } p \leq 2n \\ -1 & \text{pokud přidáváme a } p \geq 2n \\ 1 & \text{pokud mažeme a } p \geq 2n \end{cases}$$

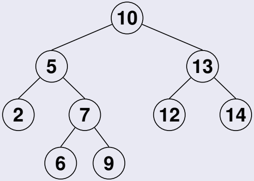
2. Jestliže nedojde k realokaci, pak $T = 0$.
   Při realokaci musíme nejprve zkopírovat všechny prvky, kterých je $T = \Phi$, čímž potenciál klesne na 0. Poté přidáme/smažeme prvek, čímž potenciál vzroste nejvýše na $\Phi' \leq 2$.

Binární vyhledávací strom (BVS)

Definice

- 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, 1973)

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

Výška BB[α]-stromu

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

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

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

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

BB[α]-strom: Operace INSERT a DELETE

Operace INSERT (DELETE je analogický)

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

Amortizovaná složitost operací INSERT a DELETE: Agregovaná metoda

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

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

Amortizovaná 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.

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.

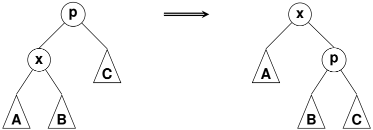
Konstrukce (cvičení)

- $\mathcal{O}(n^3)$ – triviálně dynamickým programováním
- $\mathcal{O}(n^2)$ – vylepšené dynamické programování (Knuth, 1971)

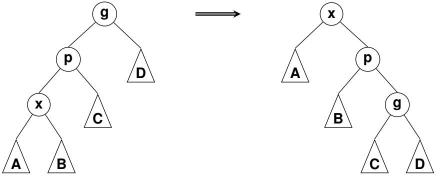
Jak postupovat, když neznáme váhy předem?

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

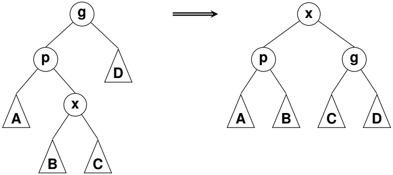
- Zig rotace: Otec p prvku x je kořen



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

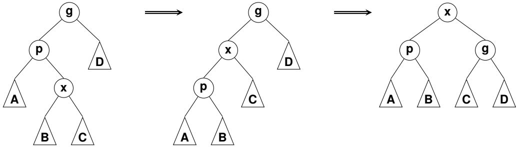


- Zig-zag rotace: x je pravý syn a p je levý syn nebo opačně

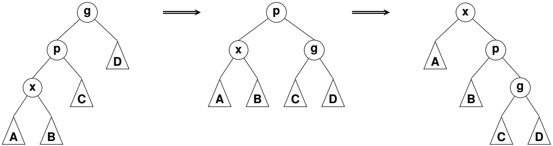


Uvažujeme *buttom-up* verzi, tj. prvek x nejprve najdeme a poté jej postupně rotujeme nahoru, což znamená, že x vždy značí stejný vrchol postupně se přesouvající ke kořeni a ostatní vrcholy stromu jsou sousedé odpovídající dané rotaci.
Existuje též *top-down* verze, která vždy rotuje vnuka kořene, jehož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

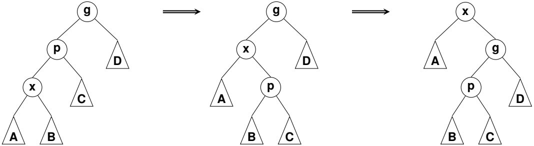
- Zig-zag rotace jsou pouze dvě jednoduché rotace prvku *x* s aktuálním otcem



- Zig-zig rotace jsou taky dvě rotace,



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



Splay strom: Amortizovaná analýza

Lemma

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

Důkaz

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

Značení

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

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

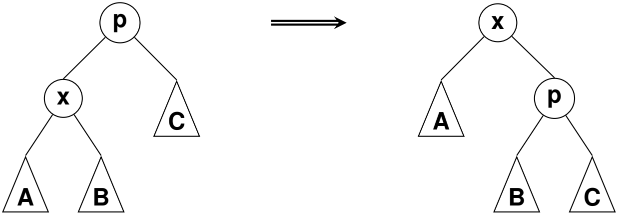
$$f\left(\frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i}\right) \leq \frac{\sum_{i=1}^n w_i f(x_i)}{\sum_{i=1}^n w_i}.$$

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

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

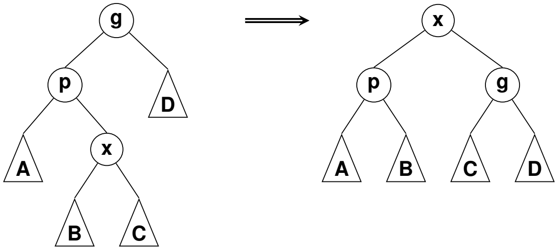
z čehož plyne znění lemmatu.

Splay strom: Zig rotace



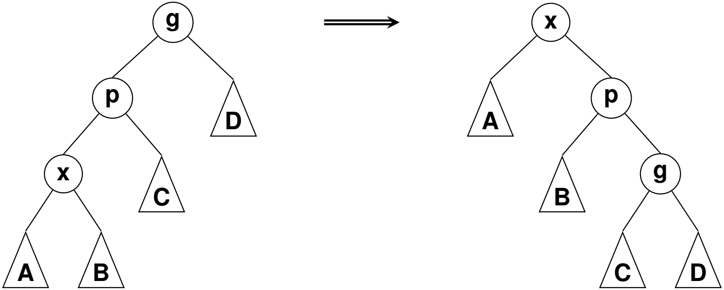
Analýza

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



Analýza

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



Analýza

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

Splay strom: Analýza

Amortizovaný čas ①

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

$$\begin{aligned}\sum_{i\text{-tá rotace}} (T_i + \Phi_i - \Phi_{i-1}) &\leq 1 + \sum_{i\text{-tá rotace}} 3(\Phi_i(x) - \Phi_{i-1}(x)) \\ &\leq 1 + 3(\Phi_{\text{konec}}(x) - \Phi_0(x)) \\ &\leq 1 + 3\log_2 n = \mathcal{O}(\log n)\end{aligned}$$

④

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

Skutečný čas k operací SPLAY

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

1. Časy na nalezení prvku a jeho SPLAY jsou stejné, a proto analyzujeme počet rotací při operaci SPLAY. Neúspěšná operace FIND dojde až k vrcholu mající NULL v ukazateli, na kterém se vyhledávání zastaví. Na tento poslední vrchol je nutné zavolat SPLAY, aby opakovaná neúspěšná vyhledávání měla amortizovanou logaritmickou složitost.
2. T značí skutečný čas rotace, což je počet jednoduchých rotací k provedení rotace zig, zigzig nebo zigzag.
3. Jedním krokem rozumíme provedení jedné zig, zigzag nebo zigzig rotace.
4. 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 a Delete

Postup

1. Vložit/smazat daný vrchol stejně jako v BVS
2. Splay nejhlubšího navštíveného vrcholu ①

Amortizovaná složitost

- Nalezení potřebných vrcholů má stejnou složitost jako Splay: $\mathcal{O}(\log n)$
- Vlastní smazání vrcholu trvá $\mathcal{O}(1)$ a potenciál stromu klesne
- Přidáním listu se potenciál vrcholů $u_1, \ldots, u_h$ na cestě do kořene zvýší o $\sum_{i=1}^{h} \log(s_{u_i} + 1) - \log(s_{u_i}) \leq \log(n)$ ②
- Amortizovaná složitost operací Insert a Deleteje $\mathcal{O}(\log n)$

1. Pokud nezavoláme tento splay, tak celková složitost vkládání/mazání posloupnosti $1, \ldots, n$ je $\Theta(n^2)$.
2. Jestliže u_1 je nový list a u_h je kořen, pak $s_{u_i} + 1 \leq s_{u_{i+1}}$ a tedy $\sum_{i=1}^{h} \log(s_{u_i} + 1) - \log(s_{u_i})$ lze teleskopicky shora odhadnout $\log(s_h) - \log(s_1) = \log n$.

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

Aplikace

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

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

Datové struktury I

3. přednáška: Vlastnosti Splay stromů

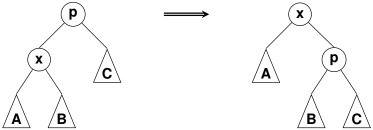
Jirka Fink

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

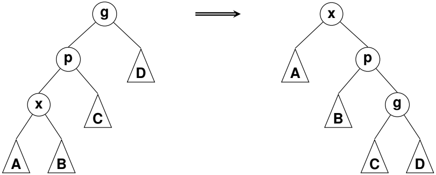
Katedra teoretické informatiky a matematické logiky
Matematicko-fyzikální fakulta
Univerzita Karlova v Praze

Zimní semestr 2022/23

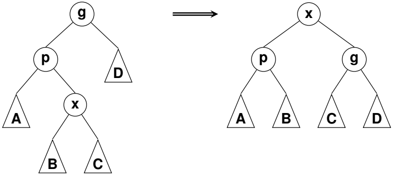
- Zig rotace: Otec *p* prvku *x* je kořen



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

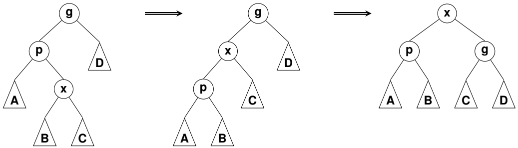


- Zig-zag rotace: *x* je pravý syn a *p* je levý syn nebo opačně

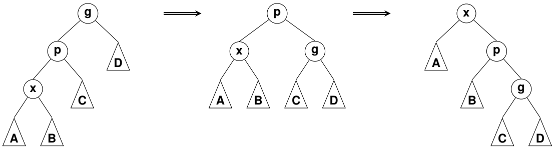


Uvažujeme *buttom-up* verzi, tj. prvek x nejprve najdeme a poté jej postupně rotujeme nahoru, což znamená, že x vždy značí stejný vrchol postupně se přesouvající ke kořeni a ostatní vrcholy stromu jsou sousedé odpovídající dané rotaci.
Existuje též *top-down* verze, která vždy rotuje vnuka kořene, jehož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

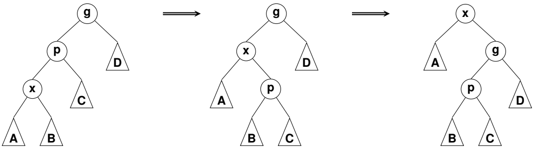
- Zig-zag rotace jsou pouze dvě jednoduché rotace prvku *x* s aktuálním otcem



- Zig-zig rotace jsou taky dvě rotace,



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



Splay strom: Amortizovaná analýza

Lemma

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

Důkaz

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

Značení

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

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

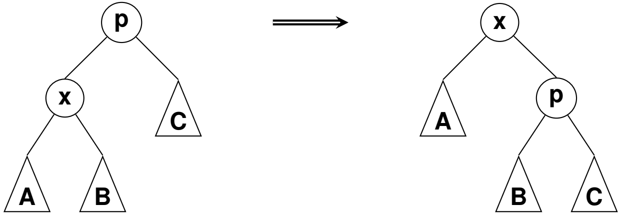
$$f\left(\frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i}\right) \leq \frac{\sum_{i=1}^n w_i f(x_i)}{\sum_{i=1}^n w_i}.$$

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.

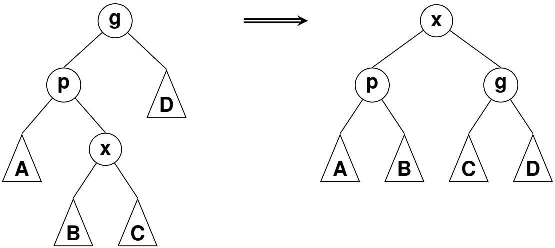
Splay strom: Zig rotace



Analýza

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

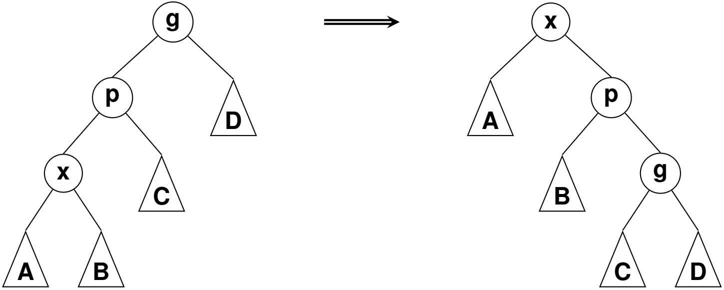
Splay strom: Zig-zag rotace



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

1. $\Phi'(x) = \Phi(g)$
2. $\Phi(x) < \Phi(p)$
3. $\Phi'(p) < \Phi'(x)$
4. $s(x) + s'(g) \leq s'(x)$
5. $\Phi(x) + \Phi'(g) \leq 2\Phi'(x) - 2$
6. $\Phi' - \Phi = \Phi'(g) - \Phi(g) + \Phi'(p) - \Phi(p) + \Phi'(x) - \Phi(x) \leq 3(\Phi'(x) - \Phi(x)) - 2$

Splay strom: Analýza

Amortizovaný čas ①

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

$$
\begin{aligned}
\sum_{i\text{-tá rotace}} (T_i + \Phi_i - \Phi_{i-1}) &\leq 1 + \sum_{i\text{-tá rotace}} 3(\Phi_i(x) - \Phi_{i-1}(x)) \\
&\leq 1 + 3(\Phi_{\text{konec}}(x) - \Phi_0(x)) \\
&\leq 1 + 3\log_2 n = \mathcal{O}(\log n)
\end{aligned}
$$

④

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

Skutečný čas k operací SPLAY

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

1. Časy na nalezení prvku a jeho SPLAY jsou stejné, a proto analyzujeme počet rotací při operaci SPLAY. Neúspěšná operace FIND dojde až k vrcholu mající NULL v ukazateli, na kterém se vyhledávání zastaví. Na tento poslední vrchol je nutné zavolat SPLAY, aby opakovaná neúspěšná vyhledávání měla amortizovanou logaritmickou složitost.
2. T značí skutečný čas rotace, což je počet jednoduchých rotací k provedení rotace zig, zigzig nebo zigzag.
3. Jedním krokem rozumíme provedení jedné zig, zigzag nebo zigzig rotace.
4. 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 a DELETE

Postup

1. Vložit/smazat daný vrchol stejně jako v BVS
2. SPLAY nejhlubšího navštíveného vrcholu ①

Amortizovaná složitost

- Nalezení potřebných vrcholů má stejnou složitost jako SPLAY: $\mathcal{O}(\log n)$
- Vlastní smazání vrcholu trvá $\mathcal{O}(1)$ a potenciál stromu klesne
- Přidáním listu se potenciál vrcholů $u_1, \ldots, u_h$ na cestě do kořene zvýší o $\sum_{i=1}^{h} \log(s(u_i) + 1) - \log(s(u_i)) \leq \log(n)$ ②
- Amortizovaná složitost operací INSERT a DELETE je $\mathcal{O}(\log n)$

1. Pokud nezavoláme tento splay, tak celková složitost vkládání/mazání posloupnosti $1, \ldots, n$ je $\Theta(n^2)$.
2. Jestliže u_1 je nový list a u_h je kořen, pak $s(u_i) + 1 \leq s(u_{i+1})$ a tedy $\sum_{i=1}^{h} \log(s(u_i) + 1) - \log(s(u_i))$ lze teleskopicky shora odhadnout $\log(s(u_h)) - \log(s(u_1)) = \log n$.

Vážená analýza splay stromů

Modifikace

1. Každý prvek x má váhu $w(x) > 0$
2. $s(x)$ je součet vah prvků v podstromu vrcholu x
3. Amortizovaný čas operace SPLAY(x) zůstane nejvýše $1 + 3(\Phi_{\text{konec}}(x) - \Phi_0(x))$ ①
4. Platí $\log(w(x)) \leq \Phi(x) \leq \log(W)$, kde W je součet všech vah
5. Tedy amortizovaný čas operace SPLAY(x) je $\mathcal{O}\left(1 + \log \frac{W}{w(x)}\right)$

Uniformní váhy $w(x) = 1/n$

1. Potenciál může být záporný, ale pokud nepřidáváme prvky, tak $\Phi >= -n \log n$ ②
2. Amortizovaný čas SPLAY je $\mathcal{O}\left(1 + \log \frac{1}{1/n}\right) = \mathcal{O}(\log n)$

Pravděpodobnostní rozložení vyhledání $w(x) = p(x)$

1. Amortizovaný čas SPLAY(x) je $\mathcal{O}\left(1 + \log \frac{1}{p(x)}\right)$
2. Očekávaná amortizovaná složitost SPLAY prvku vybraného z rozložení p je $\mathcal{O}\left(1 + \sum_x p(x) \log \frac{1}{p(x)}\right)$ ③

1. Rozmyslete si, že kroky důkazu do tohoto bodu platí i pro váženou definici $s(x)$.
2. Váhy můžeme vynásobit konstantou tak, aby byly alespoň 1, a pak bude potenciál nezáporný.
3. Shannonova entropie

Srovnání staticky optimálních stromů a splay stromů

Věta (statická optimalita)

Nechť $x_1, \dots, x_k$ je posloupnost vyhledávání prvků z množiny X, kde každý prvek X je vyhledán aspoň jednou. Nechť T je statický strom na X a složitost vyhledání prvků v posloupnosti je f. Pak složitost vyhledání ve splay stromu je $\mathcal{O}(f)$.

Důkaz

1. Nechť $c_T(x)$ je počet navštívených vrcholů při hledání x
2. Pro váhy $w(x) = 3^{-c_T(x)}$ platí $W \leq 1$
3. Amortizovaný čas SPLAY(x) je $\mathcal{O}\left(1 + \log \frac{W}{w(x)}\right) = \mathcal{O}(c_T(x))$
4. Pokles potenciálu v průběhu všech vyhledání je nejvýše $-\sum_x \log(w(x)) \leq \mathcal{O}(f)$ ①

Cvičení

Existuje posloupnost vyhledání, na které je splay strom asymptoticky rychlejší než staticky optimální strom?

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 $\mathcal{O}(n)$. ②

1. Platí $\log(w(x)) \leq \log(s(x)) \leq \log(W) \leq 0$ a proto potenciál může nejvýše klesnout z 0 na $-\sum_x \log(w(x))$.
2. n je opět počet prvků ve stromu a počáteční splay strom může mít prvky rozmístěné libovolně.

Dynamická optimalita

- Jsou dvojité rotace ve splay stromu nejlepší možné?
- Existuje dynamický binární vyhledávací strom, který byl asymptoticky lepší než splay strom?
- Můžeme být lepší než splay strom, kdybychom posloupnost vyhledávání znali dopředu?

Vyhledávání podmnožiny prvků

Věta

Nechť $x_1, \dots, x_k$ je posloupnost vyhledávání prvků z množiny X a nechť z_i je počet různých prvků vyhledaných před předchozím vyhledání prvku x_i. Celková složitost vyhledání prvků $x_1, \dots, x_k$ je $\mathcal{O}\left(n \log n + k + \sum_i \log(1 + z_i)\right)$.

Důkaz

1. Uvažujme pořadí prvků p, které je na počátku libovolné a vyhledaný prvek se vždy posouvá na začátek ①
2. Váha prvku x je $w(x) = \frac{1}{(1+p(x))^2}$ ②
3. Změna pořadí nezmění hodnotu $s(x_i) = W$
4. Ostatním prvkům se pořadí může o 1 zvýšit a tedy $w(x)$ i $\Phi(x)$ snížit
5. Součet vah W je $\mathcal{O}(1)$ ③
6. Amortizovaný čas SPLAY(x) je $\mathcal{O}\left(1 + \log \frac{W}{w(x)}\right) = \mathcal{O}(1 + \log(z_i + 1))$
7. $1/n^2 \leq s(x) \leq \mathcal{O}(1)$ a proto $-2n \log n \leq \Phi \leq \mathcal{O}(n)$
8. Pokles potenciálu v průběhu všech vyhledání je nejvýše $\mathcal{O}(n \log n)$

1. Tj. $z_i \leq p(x)$ a po prvním vyhledání nastává rovnost.
2. Tyto váhy se při každém vyhledání mění, takže budeme muset započítat změnu potenciálu kvůli změnám vah.
3. $\sum_x \frac{1}{(1+p(x))^2} = \sum_{i=1}^n \frac{1}{i^2} = \mathcal{O}(1)$ protože řada konverguje.

Věta

Nechť $x_1, \ldots, x_k$ je posloupnost vyhledávání prvků z množiny X a nechť z_i je počet různých prvků vyhledaných před předchozím vyhledání prvku x_i. Celková složitost vyhledání prvků $x_1, \ldots, x_k$ je $\mathcal{O}\left(n \log n + k + \sum_i \log(1 + z_i)\right)$.

Věta (vyhledávání podmnožiny prvků)

Jestliže provedeme k vyhledání prvků z podmnožiny velikosti m, pak celková složitost je $\mathcal{O}(n \log n + k + k \log m)$

Splay stromy: Výhody a aplikace

Výhody a nevýhody Splay stromů

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

Aplikace

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

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

Příští týden

- (a,b)-stromy
- Souvislost (2,4)-stromů a červeno-černých stromů

Datové struktury I
4. přednáška: (a,b)-stromy

Jirka Fink

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

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

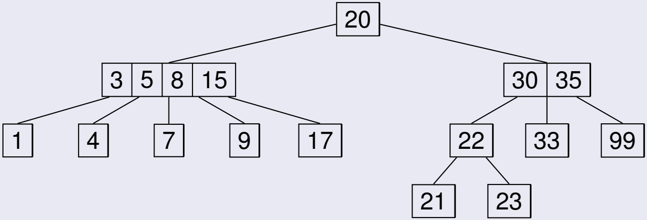
Zimní semestr 2022/23

Vyhledávací strom

Popis

- 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)
- Pokud máme hodnoty jen v listech, pak i-tý klíč vrcholu může být roven největšímu klíči v i-tém podstromu

Příklad: Hodnoty jsou ve všech vrcholech



(a,b)-strom (Bayer, McCreight, 1972)

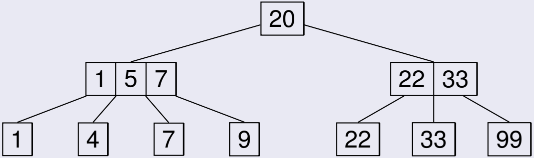
Definice

a, b jsou celá čísla splňující $a \geq 2$ a $b \geq 2a - 1$

- (a,b)-strom je vyhledávací strom
- Všechny vnitřní vrcholy kromě kořene mají alespoň a synů a nejvýše b synů
- Kořen má nejvýše b synů
- Všechny listy jsou ve stejné výšce

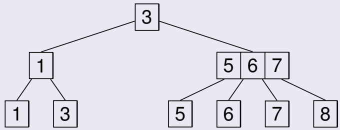
Pro zjednodušení uvažujeme, že prvky jsou jen v listech

Příklad: (2,4)-strom

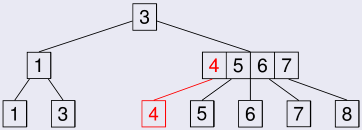


(a,b)-strom: Operace Insert

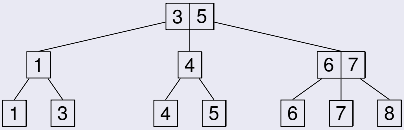
Vložte prvek s klíčem 4 do následujícího (2,4)-stromu



Nejprve najdeme správného otce, jemuž přidáme nový list



Opakovaně rozdělujeme vrchol na dva



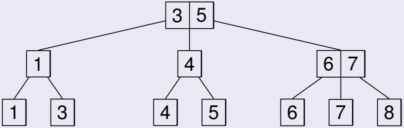
(a,b)-strom: Operace Insert

Algoritmus

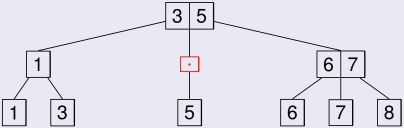
Najít otce v, kterému nový prvek patří
Přidat nový list do v
while $\deg(v) > b$ **do**
  `# Najdeme otce` u `vrcholu` v
  if v *je kořen* **then**
    Vytvořit nový kořen u s jediným synem v
  else
    $u \leftarrow$ otec v
  `# Rozdělíme vrchol` v `na` v `a` v'
  Vytvořit nového syna v' otci u a umístit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

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

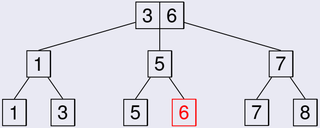
Smažte prvek s klíčem 4 z následujícího (2,4)-stromu

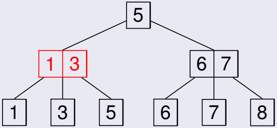


Nalezneme a smažeme list



Přesuneme jedno syna od bratra nebo spojíme vrchol s bratrem





Algoritmus

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

Cíl

- Chceme najít nejlepší hodnoty a, b minimalizující složitost
- Složitost budeme počítat v závislosti na a, b a počtu prvků n

Výška

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

Operace Find

- Nalezení správného syna ve vrcholu: $\mathcal{O}(\log b)$ 1
- Celá operace: $\mathcal{O}(\log b \cdot \log_a n) = \mathcal{O}\left(\log n \frac{\log b}{\log a}\right) = \mathcal{O}(\log n)$ pokud $b = \text{poly}(a)$

Operace Insert a Delete

- Rozdělení nebo sloučení vrcholů: $\mathcal{O}(b)$
- Celá operace: $\mathcal{O}(b \cdot \log_a n) = \mathcal{O}\left(\log n \frac{b}{\log a}\right)$
- Optimální volba $a = 2$, $b = 3$

1. Teoreticky k dosažení nejlepší složitosti použijeme binární půlení, ale pokud hodnota b je malá, tak v praxi je lepší lineárně projít všechny klíče.

(a,b)-strom: Aplikace a volby parametrů a, b

Podobné datové struktury

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

Aplikace

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

Volba parametrů a, b

Hodnotu b volíme tak, aby se jeden vrchol vešel do bloku a $a = \lceil b/2 \rceil$

- 4KB stránka, 32-bitový klíč a ukazatel $\Rightarrow$ (256,511)-strom
- 64B řádka keše $\Rightarrow$ (4,7)-strom

(a,b)-strom: Počet štěpení při operacích INSERT

Věta (počet modifikovaných vrcholů při vytvoření stromu operací INSERT)

Amortizovaný počet štěpení v libovolné posloupnosti operací INSERT začínající na prázdném stromu je $\mathcal{O}(1)$. ①

Důkaz

- 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.

Věta (Huddleston, Mehlhorn, 1982)

Amortizovaný počet štěpení a slučování v libovolné posloupnosti k operací INSERT a DELETE v $(a, 2a)$-stromu je $\mathcal{O}(1)$ a celkový počet je $\mathcal{O}(n+k)$.

Důkaz

- Potenciál vrcholu u závisí na počtu jeho synů takto:

synů	a-1	a	a + 1	...	2a-1	2a	2a+1
$\Phi(u)$	2	1	0	...	0	2	4

- Potenciál stromu je součet potenciálů vrcholů
- Změny potenciálu při štěpení vrcholu u s otcem p jsou:
  - u s potenciálem 4 rozdělíme na dva vrcholy s potenciály 0 a 1
  - Potenciál vrcholu p se zvýší nejvýše o 2
  - Potenciál se sníží alespoň o 1, což zaplatí štěpení
- Změny potenciálu při slučování vrcholů u a u' s otcem p jsou:
  - $\Phi(u) = 2$, $\Phi(u') = 1$ a sloučený vrchol má potenciál 0
  - Potenciál vrcholu p se zvýší nejvýše o 1
  - Potenciál se sníží alespoň o 2, což zaplatí slučování
- Přidání, smazání a přesun vrcholu zvýší potenciál nejvýše o 2, ale tyto operace provádíme nejvýše jednou
- Jelikož $0 \leq \Phi \leq 4n$, celkové snížení potenciálu při všech operacích je $\mathcal{O}(n)$

(a,b)-strom: Balancování shora dolů

Cíl

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

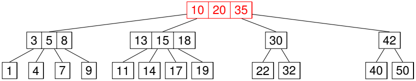
Operace Insert

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

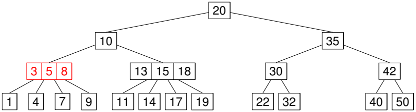
Operace Delete

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

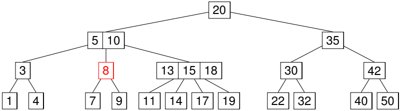
- Vložte prvek s klíčem 6 do následujícího (2,4)-stromu



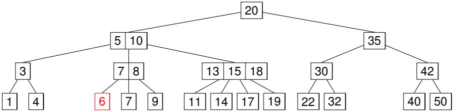
- Nejprve rozdělíme kořen



- Pak pokračujeme do levého syna, který taky rozdělíme



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

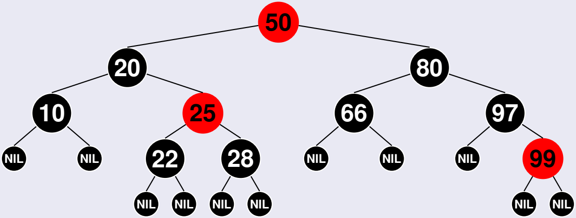


Červeno-černé stromy (Guibas, Sedgewick, 1978)

Definice

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

Příklad

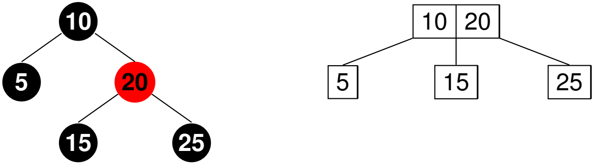


1. Nepovinná podmínka, která jen zjednodušuje operace. V příkladu uvažujeme, že listy jsou reprezentovány NIL/NULL ukazateli, a tedy imaginární vrcholy bez prvků. Někdy se též vyžaduje, aby kořen byl černý, ale tato podmínka není nutná, protože kořen můžeme vždy přebarvit na černo bez porušení ostatních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

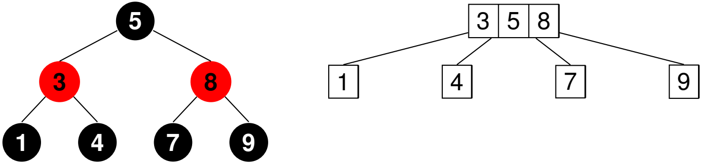
Každému černému vrcholu odpovídá jeden vrchol (2,4)-stromu

- 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: Důsledky ekvivalence s (2,4)-stromy

Vlastnosti

- Výška červeno-černého stromu je nejvýše $2 + 2\log_2 n$
- Časová složitost operací Find, Insert a Delete je $\mathcal{O}(\log n)$
- Pro vkládání a mazání vrcholů existuje varianta shora dolů i zdola nahoru
- Amortizovaný počet změn stromu při balancování zdola nahoru je $\mathcal{O}(1)$

Aplikace

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

A-sort (Guibas, McCreight, Plass, Roberts, 1977)

Cíl

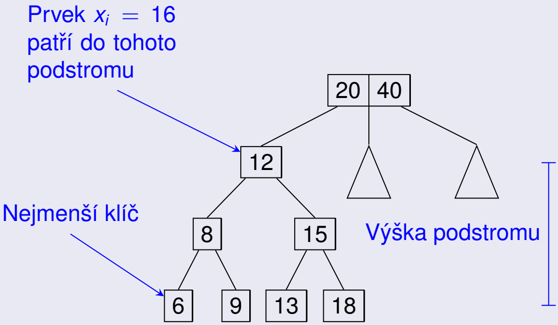
Setřídit „skoro“ setříděné pole

Modifikace (a,b)-stromu

Máme uložený ukazatel na vrchol s nejmenším klíčem

Příklad: Vložte klíč s hodnotou $x_i = 16$

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



A-sort: Algoritmus

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

$T \leftarrow$ prázdný (a,b)-strom

for $i \leftarrow n$ **to** 1 `# Prvky procházíme od konce`

do

  `# Najdeme podstrom, do kterého vložíme` x_i

  $v \leftarrow$ list s nejmenším klíčem

  while *v není kořen a x_i je větší než nejmenší klíč v otci vrcholu v* **do**

    $v \leftarrow$ otec v

  Vložíme x_i do podstromu vrcholu v

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

A-sort: Složitost (Brown, Tarjan, 1978; Mehlhorn, 1979)

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.

Příští týden

Jak navrhovat a analyzovat algoritmy a datové struktury, aby efektivně využívali keše procesorů?

Datové struktury I

5. přednáška: Paměťová hierarchie

Jirka Fink

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

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

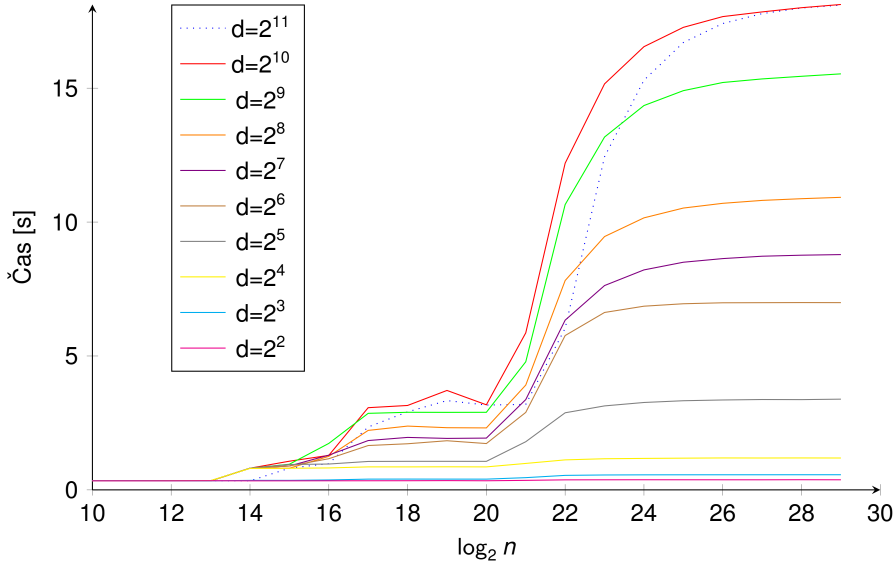
Zimní semestr 2022/23

Příklad velikostí a rychlostí různých typů pamětí

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

Triviální program

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


d=2^11
d=2^10
d=2^9
d=2^8
d=2^7
d=2^6
d=2^5
d=2^4
d=2^3
d=2^2
Čas [s]
0
5
10
15
10
12
14
16
18
20
22
24
26
28
30
log2 n

Cache-oblivious (Frigo, Leiserson, Prokop, Ramachandran, 1999)

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 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).

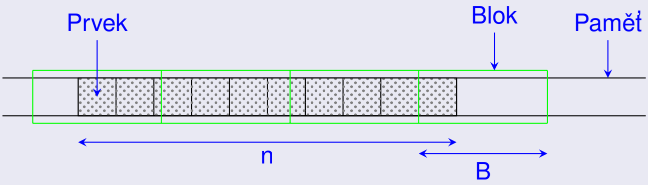
Cache-oblivious algoritmus

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: Reprezentace stromů

Binární vyvážený strom

- Jeden vrchol je uložen v nejvýše 2 blocích ①
- Výška stromu je $\mathcal{O}(\log n)$
- Na cestě z kořene do listu načteme $\mathcal{O}(\log n)$ bloků

(a,b)-strom

- Předpokládejme, že můžeme zvolit $b = \Theta(B)$
- Jeden vrchol je uložen v nejvýše 2 blocích a má $\Theta(B)$ synů
- Výška stromu je $\Theta(\log_B n)$
- Na cestě z kořene do listu načteme $\Theta(1 + \log_B n)$ bloků
- Toto je cache-aware přístup

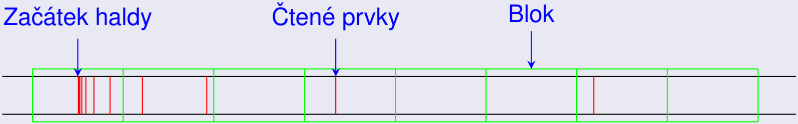
Reprezentace binárního stromu pomocí van Emde Boas

- Cache-oblivious reprezentace
- Na cestě z kořene do listu načteme $\Theta(1 + \log_B n)$ bloků
- Podrobnosti na Datových strukturách II

1. Předpokládáme, že prvky jsou dost malé, aby se vrchol vešel do jednoho bloku a druhý blok máme pro případ, že nemůžeme alokovat paměť tak, aby se paměť pro vrcholy byla zarovnána se začátky bloků.

Cache-oblivious analýza: Haldy

Binární halda v poli: Průchod od listu ke kořeni



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

B-regulární halda v poli: Průchod od listu ke kořeni

- Opět předpokládáme, že do jednoho bloku se vejde B prvků
- Cesta má $\Theta(1 + \log_B n)$ vrcholů
- Na cestě z kořene do listu načteme $\Theta(1 + \log_B n)$ bloků

Cache-oblivious analýza: Binární vyhledávání

Binární vyhledávání

- Uvažujeme neúspěšné vyhledávání, protože úspěšné může skončit dřív
- Porovnáváme $\Theta(\log n)$ prvků s hledaným prvkem
- Posledních $\Theta(\log B)$ prvků je uloženo v nejvýše dvou blocích
- Ostatní prvky jsou uloženy v po dvou různých blocích
- $\Theta(1 + \log \frac{n}{B})$ přenesených bloků

Cache-oblivious analýza: Mergesort

Mergesort

1. Slití polí velikosti n_1 a n_2 potřebuje $\frac{2}{B}(n_1 + n_2) + \mathcal{O}(1)$ přenosů
2. Uvažujme (binární) strom rekurzivního volání
3. Na jedné hladině stromu potřebujeme $\mathcal{O}(n/B + 1)$ přenosů
4. Počet hladin stromu je $\mathcal{O}(\log n)$
5. Celkový počet přenosů je $\mathcal{O}((n/B + 1) \log n)$

k-cestný mergesort

1. Slití polí velikostí $n_1, \ldots, n_k$ potřebuje $\frac{2}{B}(n_1 + \cdots + n_k) + \mathcal{O}(1)$ přenosů
2. Musíme volit $k < P$
3. Uvažujme k-ární strom rekurzivního volání
4. Na jedné hladině stromu potřebujeme $\mathcal{O}(n/B + 1)$ přenosů
5. Počet hladin stromu je $\mathcal{O}(\log_k n)$
6. Celkový počet přenosů je $\mathcal{O}((n/B + 1) \log_k n)$
7. Volbou $k = \Theta(P)$ dostáváme $\mathcal{O}((n/B + 1) \log_P(n))$ přenosů [1]

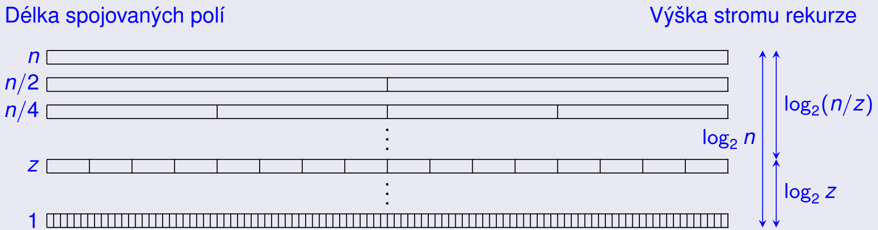
1. Tento algoritmus je cache-aware a tento počet přenosů je teoreticky optimální i v cache-oblivious modelu. Funnelsort je cache-oblivious algoritmus mající tento počet přenosů a časovou složitost $\mathcal{O}(n \log n)$

Cache-oblivious analýza: Mergesort

Případ $n \leq M/2$

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

Schéma



Případ $n > M/2$

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

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

Cache-oblivious analýza: Transpozice matic: Triviální přístup

Strategie pro výměnu stránek v cache

OPT: Optimální off-line algoritmus předpokládající znalost všech přístupů do paměti

FIFO: Z cache smažeme stránku, která je ze všech stránek v cachi nejdelší dobu

LRU: Z cache smažeme stránku, která je ze všech stránek v cachi nejdéle nepoužitá

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

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

Předpoklady

Uvažujeme pouze případ

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

Cache-oblivious analýza: Transpozice matic: Triviální přístup

Příklad: Representace matice 5×5 v paměti

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

LRU a FIFO strategie

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

OPT strategie

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

Cache-oblivious analýza: Transpozice matic: Cache-aware přístup

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

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

Analýza

- Zvolíme $z = B$ a předpokládáme, že $B \leq P$
- K transpozici jedné submatice potřebujeme $\mathcal{O}(z)$ přenosů
- Počet submatic je $\mathcal{O}\left((k/z)^2\right)$
- K transpozici potřebujeme $\mathcal{O}\left(k^2/B\right)$ přenosů, což je optimální
- 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
- Matice A_{12} a A_{21} se prohazují
- Transpozice a prohození matic A_{12} a A_{21} se provádí najednou

Cache-oblivious analýza: Transpozice matic: Rekurzivní transpozice

Procedure `transpose_on_diagonal` (A)

    if *Matice A je malá* **then**

        Transponujeme matici A triviálním postupem

    else

        $A_{11}, A_{12}, A_{21}, A_{22} \leftarrow$ souřadnice submatic

        `transpose_on_diagonal` (A_{11})

        `transpose_on_diagonal` (A_{22})

        `transpose_and_swap` (A_{12}, A_{21})

Procedure `transpose_and_swap` (A, B)

    if *Matice A a B jsou malé* **then**

        Prohodíme a transponujeme matice A a B triviálním postupem

    else

        $A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22} \leftarrow$ souřadnice submatic

        `transpose_and_swap` (A_{11}, B_{11})

        `transpose_and_swap` (A_{12}, B_{21})

        `transpose_and_swap` (A_{21}, B_{12})

        `transpose_and_swap` (A_{22}, B_{22})

- Všimněme si, že matice A a B musí mít posice symetrické podle hlavní diagonály původní matice, a proto ve skutečnosti funkci `transpose_and_swap()` stačí předávat pozice matici A.
- Ve funkci `transpose_on_diagonal` musí být matice A čtvercová a ležet na hlavní diagonále, a proto stačí předávat x-ovou souřadnici a řád matice.
- Funkci `transpose_on_diagonal` stačí předat dva argumenty i a m, aby má transponovat matici velikosti $m \times m$ začínající na pozici (i, i)
- Funkci `transpose_and_swap` stačí předat čtyři argumenty i, j, m, n, aby má transponovat a prohodit matici velikosti $m \times n$ začínající na pozici (i, j) s maticí velikosti $n \times m$ začínající na pozici (j, i)

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.

Datové struktury I

5. přednáška: Transpozice matic

Jirka Fink

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

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

Zimní semestr 2022/23

Cache-oblivious (Frigo, Leiserson, Prokop, Ramachandran, 1999)

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 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: Transpozice matic: Triviální přístup

Strategie pro výměnu stránek v cache

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

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

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

Předpoklady

Uvažujeme pouze případ

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

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-oblivious analýza: Transpozice matic: Cache-aware přístup

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

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

Analýza

- Zvolíme $z = B$ a předpokládáme, že $B \leq P$
- K transpozici jedné submatice potřebujeme $\mathcal{O}(z)$ přenosů
- Počet submatic je $\mathcal{O}\left((k/z)^2\right)$
- K transpozici potřebujeme $\mathcal{O}\left(k^2/B\right)$ přenosů, což je optimální
- 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
- Matice A_{12} a A_{21} se prohazují
- Transpozice a prohození matic A_{12} a A_{21} se provádí najednou

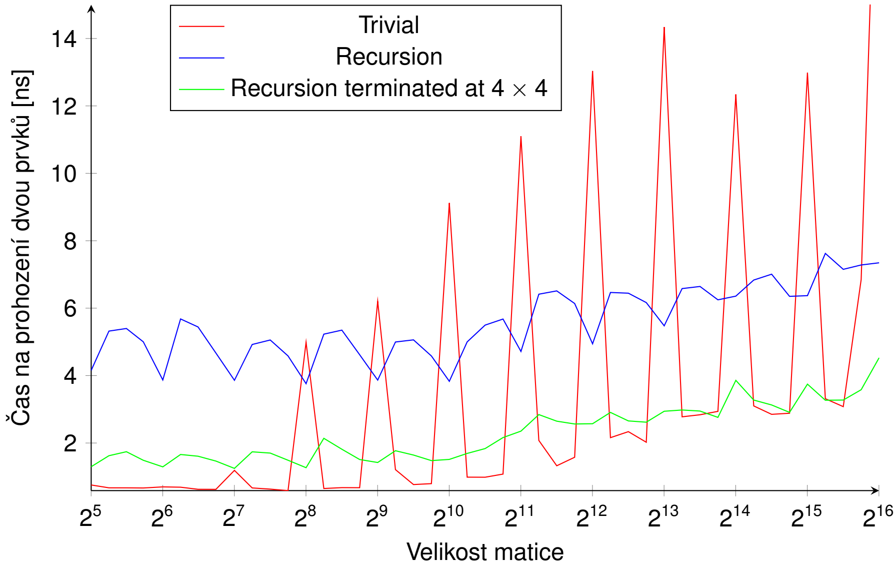
```
Procedure transpose_on_diagonal(A)
    if Matice A je malá then
        Transponujeme matici A triviálním postupem
    else
        A11, A12, A21, A22 ← souřadnice submatic
        transpose_on_diagonal(A11)
        transpose_on_diagonal(A22)
        transpose_and_swap(A12, A21)

Procedure transpose_and_swap(A, B)
    if Matice A a B jsou malé then
        Prohodíme a transponujeme matice A a B triviálním postupem
    else
        A11, A12, A21, A22, B11, B12, B21, B22 ← souřadnice submatic
        transpose_and_swap(A11, B11)
        transpose_and_swap(A12, B21)
        transpose_and_swap(A21, B12)
        transpose_and_swap(A22, B22)
```

- Všimněme si, že matice A a B musí mít posice symetrické podle hlavní diagonály původní matice, a proto ve skutečnosti funkci `transpose_and_swap()` stačí předávat pozice matici A.
- Ve funkci `transpose_on_diagonal` musí být matice A čtvercová a ležet na hlavní diagonále, a proto stačí předávat x-ovou souřadnici a řád matice.
- Funkci `transpose_on_diagonal` stačí předat dva argumenty i a m, aby má transponovat matici velikosti $m \times m$ začínající na pozici (i, i)
- Funkci `transpose_and_swap` stačí předat čtyři argumenty i, j, m, n, aby má transponovat a prohodit matici velikosti $m \times n$ začínající na pozici (i, j) s maticí velikosti $n \times m$ začínající na pozici (j, i)

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.


Trivial
Recursion
Recursion terminated at 4 × 4
Čas na prohození dvou prvků [ns]
14
12
10
8
6
4
2
2^5 2^6 2^7 2^8 2^9 2^10 2^11 2^12 2^13 2^14 2^15 2^16
Velikost matice

Cache-oblivious analýza: Srovnání OPT a LRU strategií

Věta (Sleator, Tarjan, 1985)

- 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 většinou nemá vliv na asymptotický počet přenesených bloků

- Transpozice matic: $\mathcal{O}(n^2/B)$
- Mergesort: $\mathcal{O}\left(\frac{n}{B}\log\frac{n}{M}\right)$
- Funnelsort: $\mathcal{O}\left(\frac{n}{B}\log_P\frac{n}{B}\right)$
- The van Emde Boas layout: $\mathcal{O}(\log_B n)$

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

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

1. Pokud LRU má $f \leq P_{\text{LRU}}$ přenesených bloků v podposloupnosti s, pak OPT přenese alespoň $f - P_{\text{OPT}}$ bloků v podposlopnosti s
   - Kdyby LRU při zpracování s přenesl některý blok dvakrát, tak by přenesl více než P_{OPT} různých bloků
   - Tedy s 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 s nejvýše P_{OPT} bloků v cache a zbylých alespoň $f - P_{\text{OPT}}$ musí načíst
2. Rozdělíme posloupnost $s_1, \dots, s_k$ na podposloupnosti 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

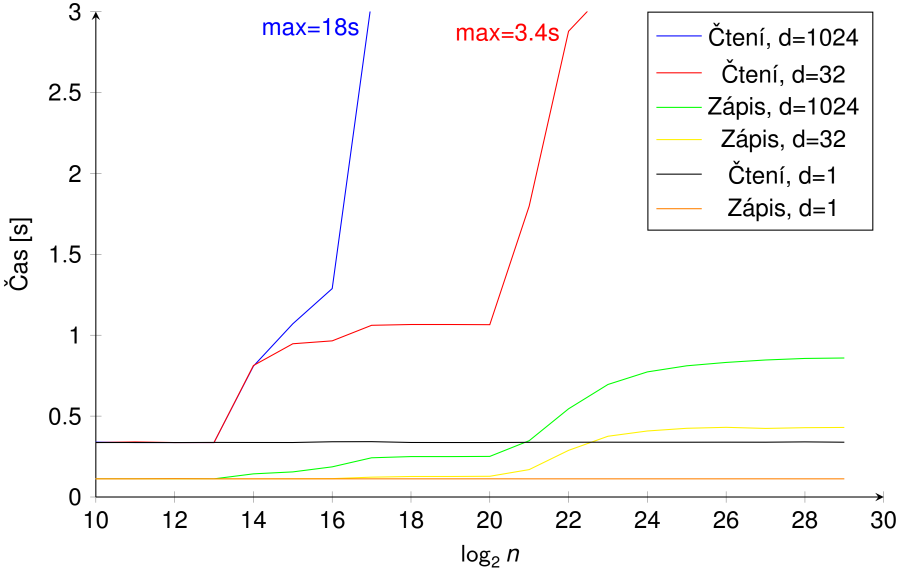
Čtení z paměti

```
# 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);
```

Příští týden

Jak hešovat čísla, řetězce a vektory?

Datové struktury I

7. přednáška: Hešování

Jirka Fink

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

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

Zimní semestr 2022/23

Hešovaní

Základní pojmy

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

Separované řetězce

- Vytvoříme tabulku velikost $m \approx n$ a hešovací funkci $h : U \to M$
- $M[y]$ obsahuje spojový seznam prvků x splňující $h(x) = y$
- INSERT(x): Přidáme prvek x do seznamu $M[h(x)]$
- FIND(x): Projdeme seznam $M[h(x)]$
- DELETE(x): Smažeme prvek ze seznamu $M[h(x)]$

Otázky

- Jak získat hešovací funkci?
- Jaké jsou další způsoby řešení kolizí?

Proč nestačí zvolit jednu triviální funkci?

Funkce $h(x) = x \bmod m$

- Pokud hešujeme náhodná data, tak tato funkce stačí
- Pokud jsou prvky násobky m, pak padnou do stejné přihrádky
- V praxi nikdy nedostaneme dostatečně náhodná data

Pozorování (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. ①

Proč nestačí jedna hešovací funkce?

- Pokud nepřítel zná naši hešovací funkci (open source), tak si může předpočítat kolidující prvky
- Common Vulnerabilities and Exposures:
  - PHP: CVE-2011-4885
  - Ruby: CVE-2011-4815
  - Apache Geronimo: CVE-2011-5034
- Musíme zkonstruovat systém hešovacích funkcí, ze které budeme náhodně vybírat

1. Dirichletův princip (Balls-and-binds): Pokud hodíme mn míčů do m přihrádek (košů), tak v alespoň jedné přihrádce bude alespoň n míčů, které označíme S.

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 (ekvivalentní definice)

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

- počet hešovacích funkcí $h \in \mathcal{H}$ splňujících $h(x) = h(y)$ je nejvýše $\frac{c|\mathcal{H}|}{m}$ pro všechna různá $x, y \in U$
- náhodně zvolená $h \in \mathcal{H}$ splňuje $P[h(x) = h(y)] \leq \frac{c}{m}$ pro každé $x, y \in U$ a $x \neq y$. ② ③

Příklad c-universálního hešovacího systému

- Parametry: p a m, kde $p > u$ je prvočíslo
- Hešovací funkce
$$h_a(x) = (ax \mod p) \mod m$$
je závislá na hodnotě a
- Hešovací systém $\mathcal{H} = \{h_a;\ 0 < a < p\}$ je c-universální
- Hešovací funkce ze systému $\mathcal{H}$ je určena hodnotou a
- Tedy náhodný výběr hešovací funkce z $\mathcal{H}$ je náhodné vygenerování $a \in \{1, \ldots, p-1\}$

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

Kolik máme očekávat kolizí?

Pozorování (Narozeninový paradox)

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

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

a očekávaný počet kolizí je

$$\binom{n}{2}\frac{1}{m} \sim \frac{n^2}{2m}.$$

Důkaz

- $\prod_{i=0}^{n-1} \frac{m-i}{m} = \prod_{i=1}^{n-1} \left(1 + \frac{-i}{m}\right) \sim \prod_{i=1}^{n-1} e^{-\frac{i}{m}} = e^{-\frac{\sum_{i=1}^{n-1} i}{m}} = e^{-\frac{\binom{n}{2}}{m}} \sim e^{-\frac{n^2}{2m}}$ 1
- $E[\#\text{ kolizí}] = \sum_{\{x,y\}} P[h(x) = h(y)] = \binom{n}{2}\frac{1}{m} < \frac{n^2}{2m}$ 2

1. Předpokládáme, že se každým míčem trefíme do právě jednoho koše, do každého koše se trefíme se stejnou pravděpodobností a jednotlivé hody jsou nezávislé. 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

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

2. Pravděpodobnost kolize dvou prvků je $1/m$ a počet dvojic různých prvků je $\binom{n}{2}$. Důkaz plyne z linearity střední hodnoty.

Statické perfektní hešování

Lemma (cvičení)

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

Markovova nerovnost

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

Pozorování: Statické perfektní hešování

Pro danou podmnožinu $S \subseteq U$ velikosti n lze najít perfektní hešovací funkci do tabulky velikosti $m = \Omega(n^2)$ tak, že vyzkoušíme v průměru $\mathcal{O}(1)$ funkcí z c-universálního systému.

Důkaz

- Předpokládejme $m \geq an^2$ a nechť X značí počet kolizí
- $E[X] = \sum_{\{x,y\}} P[h(x) = h(y)] \leq \binom{n}{2}\frac{c}{m} < \frac{n^2}{2}\frac{c}{an^2} = \frac{c}{2a}$
- Markov: $P[X < 1] > P[X < \frac{2a}{c}E[X]] > 1 - \frac{c}{2a}$
- Očekávaný počet pokusů na nalezení perfektní hešovací funkce je nejvýše $\frac{1}{1-c/2a}$

Hešování se separovanými řetězci

Popis

V přihrádce j jsou uloženy všechny prvky $i \in S$ splňující $h(i) = j$.

Implementace

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

Otázka

Je možné zajistit složitost operací FIND, INSERT a DELETE $\mathcal{O}(\log n)$ v nejhorším případě? ①

Platí následující tvrzení?

Jestliže $m = \Theta(n)$ a pro hešovací systém platí, že očekávaný počet prvků v libovolné přihrádce je $\mathcal{O}(1)$, pak očekávaná složitost operací FIND, INSERT a DELETE je $\mathcal{O}(1)$. ②

1. Pro každou přihrádku vytvoříme vyhledávací strom.
2. Pro systém $\mathcal{H} = \{h_a(i) = j; j \in M\}$ a přihrádku $j \in M$ z linearity střední hodnoty platí $E[\{i \in S : h(i) = j\}] = \sum_{i \in S} P[h(i) = j] = \frac{n}{m} = \Theta(1)$, ale všechny prvky jsou ve stejné přihrádce, takže složitost operací je lineární.

Hešování se separovanými řetězci: Složitost

Pozorování

Jestliže $\mathcal{H}$ je c-universální, pak očekávaný počet prvků v přihrádce $h(x)$ pro $x \in U$ je nejvýše $\frac{cn}{m}$.

Důkaz

$E[| \{y \in S : h(x) = h(y)\} |] = \sum_{y \in S} P[h(x) = h(y)] \leq \frac{cn}{m}$

Důsledek

Jestliže $\mathcal{H}$ je c-universální a $m = \Omega(n)$, pak očekávaná složitost operací FIND, INSERT a DELETE je $\mathcal{O}(1)$.

Dynamická velikost tabulky, pokud dopředu neznáme počet prvků

- Tabulku udržujeme ve velikosti $n/4 \leq m \leq n$
- Při překročení mezí tabulku dvakrát zmenšíme/zvětšíme přehešováním všech prvků novou hešovací funkcí
- Amortizovaná očekávaná složitost operací INSERT a DELETE je $\mathcal{O}(1)$

Definice

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

Značení

Nechť A_j je počet prvků v j-té přihrádce.

Věta: Délka nejdelšího řetězce

Pokud $m = \Theta(n)$ 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.

Poznámka

Dokážeme, že $P[\max_j A_j \leq (1+\epsilon)\frac{\log n}{\log \log n}] > 1 - \frac{1}{n^{\frac{\epsilon}{3}}}$ pro všechna $\epsilon > 0$. ①

Důsledek: Očekávaná délka nejdelšího řetězce

Pokud $\alpha = \Theta(1)$ a systém hešovacích funkcí je úplně náhodný, pak očekávaná délka nejdelšího řetězce je $E[\max_{j \in M} A_j] = \Theta(\frac{\log n}{\log \log n})$. ②

1. Nebudeme dokazovat, že $\max_{j \in M} A_j = \Omega(\frac{\log n}{\log \log n})$ s velkou pravděpodobností.
2. Pro $\epsilon = 3$ dostáváme $P[\max_j A_j \leq 4\frac{\log n}{\log \log n}] > 1 - \frac{1}{n}$. Tedy $E[\max_{j \in M} A_j] \leq P[\max_j A_j \leq 4\frac{\log n}{\log \log n}] \cdot 4\frac{\log n}{\log \log n} + P[\max_j A_j > 4\frac{\log n}{\log \log n}] \cdot n \leq 4\frac{\log n}{\log \log n} + 1$.
Důkaz $E[\max_{j \in M} A_j] = \Omega(\frac{\log n}{\log \log n})$ vynecháváme.

Hešování se separovanými řetězci: Nejdelší řetězec

Chernoffův odhad

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

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

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

1. I_{ij} je náhodná proměnná indikující, zda i-tý prvek patří do j-té přihrádky
2. Platí $A_j = \sum_{i \in S} I_{ij}$
3. Mějme $\epsilon > 0$
4. Označme $\mu = E[A_1] = n/m$
5. Dále $c = (1+\epsilon)\frac{\log n}{\mu \log \log n}$
6. Platí $P[\max_j A_j > c\mu] = P[\exists j : A_j > c\mu] \leq \sum_j P[A_j > c\mu] = mP[A_1 > c\mu]$
7. Aplikujeme Chernoffův odhad na proměnné I_{i1} pro $i \in S$
8. Platí $P[\max_j A_j > c\mu] \leq mP[A_1 > c\mu] < m\frac{e^{(c-1)\mu}}{c^{c\mu}} = me^{-\mu} e^{c\mu - c\mu \log c}$

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

- Označili jsme $c = (1+\epsilon)\frac{\log n}{\mu \log\log n}$ a odvozujeme

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

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

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

Více-přihrádkové hešování se separovanými řetězci

2-přihrádkové hešování

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

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

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

k-přihrádkové hešování

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

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

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

Kukačkové hešování (Pagh, Rodler, 2004)

Popis

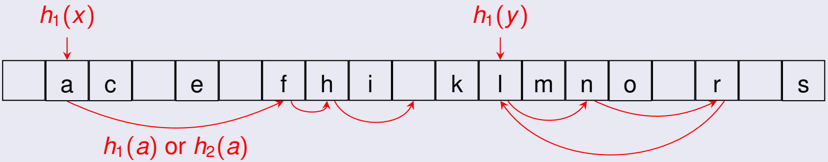
Pro dvě hešovací funkce h_1 a h_2 prvek x musí být uložen v přihrádce $h_1(x)$ nebo $h_2(x)$. V jedné přihrádce může být uložen nejvýše jeden prvek.

Operace Find a Delete

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

Příklad operace Insert

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



Kukačkové hešování: Algoritmus pro operaci Insert

Vložení prvku x do tabulky T

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

Rehash

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

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

Kukačkové hešování: Složitost (bez důkazu)

Tvrzení: Složitost operace Insert bez přehešování

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

Tvrzení: Počet přehešování

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

Věta: Složitost operace Insert včetně přehešování

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

Datové struktury I

8. přednáška: Výběr hešovací funkce

Jirka Fink

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

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

Zimní semestr 2022/23

Základní pojmy

- Značení: $[n] = \{0, \ldots, n-1\}$
- Máme univerzum U všech prvků
- Chceme uložit podmnožinu $S \subseteq U$ velikosti n
- Uložíme S do pole velikosti m pomocí hešovací funkce $h : U \to M$, kde $M = [m]$
- Hešovacím systémem $\mathcal{H}$ rozumíme libovolnou množinu hešovacích funkcí
- Dva prvky $x, y \in S$ kolidují, jestliže $h(x) = h(y)$
- Uvažujeme universum $U = [u]$ pro libovolné $u \in \mathbb{N}$, pokud není uvedeno jinak

c-universální systém (ekvivalentní definice)

Systém hešovacích funkcí $\mathcal{H}$ je c-universální, jestliže pro všechna různá $x, y \in U$ ①

- počet hešovacích funkcí $h \in \mathcal{H}$ splňujících $h(x) = h(y)$ je nejvýše $\frac{c|\mathcal{H}|}{m}$
- náhodně zvolená $h \in \mathcal{H}$ splňuje $P[h(x) = h(y)] \leq \frac{c}{m}$ ② ③

Příklad c-universálního hešovacího systému

- Parametry: p a m, kde $p > u$ je prvočíslo
- Hešovací funkce $h_a(x) = (ax \bmod p) \bmod m$
- Hešovací systém $\mathcal{H} = \{h_a;\ a \in [p]\}$ je c-universální
- Hešovací funkce ze systému $\mathcal{H}$ je určena hodnotou a
- Tedy náhodný výběr hešovací funkce z $\mathcal{H}$ je náhodné vygenerování $a \in [p]$

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

(k,c)-nezávislý hešovací systém

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

Systém hešovacích funkcí $\mathcal{H}$ je $(2, c)$-nezávislý, pokud pro každé $x_1, x_2 \in U$ a $x_1 \neq x_2$ a $z_1, z_2 \in M$

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

(k, c)-nezávislý systém hešovacích funkcí

Nechť $k \in \mathbb{N}$, $K = \{1, \ldots, k\}$ a $c \geq 1$.
Systém hešovacích funkcí $\mathcal{H}$ je (k, c)-nezávislý, pokud náhodně zvolená $h \in \mathcal{H}$ splňuje

$$P[h(x_i) = z_i \, \forall i \in K] \leq \frac{c}{m^k}$$

pro všechna po dvou různá $x_1, \ldots, x_k \in U$ a všechna $z_1, \ldots, z_k \in M$.

k-nezávislý systém hešovacích funkcí

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

Universální a nezávislý hešovací systém

Pozorování

1. (k, c)-nezávislý systém hešovacích funkcí je $(k-1, c)$-nezávislý ①
2. $(2, c)$-nezávislý systém hešovacích funkcí je c-universální ②
3. Existuje 1-universální systém, který není 2-nezávislý ③
4. Pro každý hešovací systém $\mathcal{H}$ a pro všechna $x_1, \ldots, x_k \in U$ existují $z_1, \ldots, z_k \in M$ taková, že $P[h(x_i) = z_i \, \forall i \in K] \geq \frac{1}{m^k}$ ④
5. Jestliže $\mathcal{H}$ je silně k-nezávislý, pak pro po dvou různá $x_1, \ldots, x_k \in U$ a pro $z_1, \ldots, z_k \in M$
   - $P[h(x_i) = z_i \, \forall i \in K] = \frac{1}{m^k}$
   - $P[h(x_k) = z_k | h(x_i) = z_i \, \forall i = 1, \ldots, k-1] = \frac{1}{m}$
6. Jestliže $\mathcal{H}$ je (k, c)-nezávislý, pak $|\mathcal{H}| \geq \frac{m^k}{c}$ a na identifikaci funkce z $|\mathcal{H}|$ potřebujeme alespoň $k \log m - \log c$ bitů ⑤

1-nezávislý systém není užitečný

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

1. $P[h(x_i) = z_i \, \forall i = 1, \ldots, k-1] = P[\exists z_k \in M : \; h(x_i) = z_i \, \forall i \in K] \leq \sum_{z_m \in M} P[h(x_i) = z_i \, \forall i \in K] \leq m \frac{c}{m^k} = \frac{c}{m^{k-1}}$
2. $P[h(x) = h(y)] = P[\exists z \in M : \; h(x) = z \text{ a } h(y) = z] \leq \sum_{z \in M} P[h(x) = z \text{ a } h(y) = z] \leq m \frac{c}{m^2} = \frac{c}{m}$
3. Uvažujme systém $\mathcal{H}$ všech funkcí $h : U \to M$ takových, že $h(0) = 0$ a $h(1) = 1$, t.j. dva prvky mají pevné přihrádky a ostatní prvky náhodné přihrádky. Pak $P[h(x) = h(y)] \leq \frac{1}{m}$, ale $P[h(0) = 0 \text{ a } h(1) = 1] = 1$.
4. Kdyby $P[h(x_i) = z_i \, \forall i \in K] < \frac{1}{m^k}$ pro všechna $z_1, \ldots, z_k \in M$, pak $1 = P[\exists z_1, \ldots, z_k \in M : \; h(x_i) = z_i \, \forall i \in K] \leq \sum_{z_1, \ldots, z_k \in M} P[h(x_i) = z_i \, \forall i \in K] < m^k \frac{1}{m^k} = 1$.
5. Zvolme $h' \in \mathcal{H}$ a $x_1, \ldots, x_n \in U$ různé. Nechť $z_i = h'(x_i)$ a β značí počet $h \in \mathcal{H}$ splňujících $h(x_i) = z_i$ pro všechna $i \in K$. Zřejmě $\beta \geq 1$. Z $P[h(x_i) = z_i \, \forall i \in K] = \frac{\beta}{|\mathcal{H}|} \leq \frac{c}{m^k}$ plyne $|\mathcal{H}| \geq \frac{m^k}{c}$.

Lineární kongruence

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 $(y_1, y_2) \in [p]^2$, kde p je prvočíslo.

Důkaz

Pro danou dvojici (y_1, y_2) existuje jediná dvojice (a, b) splňující rovnice

- Odečtením dostáváme $a(x_1 - x_2) \equiv_p y_1 - y_2$ ①
- V tělese $GF(p) = \mathbb{Z}_p$ dostáváme $a = (y_1 - y_2)(x_1 - x_2)^{-1}$, $b = y_1 - ax_1$

Pozorování

- Nechť $p \geq |U|$ je prvočíslo, kde $U = [u]$
- Uvažujme hešovací funkci $h_{a,b}(x) = ax + b \bmod p$
- Pak systém $\mathcal{H} = \{h_{a,b};\ a, b \in [p]\}$ je $(2, 1)$-nezávislý ②

1. $\equiv_n$ značí rovnost modulo n
2. Důkaz plyne z následujícího lemmatu, protože bijekce zaručuje, že pro různá $x_1, x_2 \in [p]$ a $y_1, y_2 \in [p]$ existuje právě jedna $h \in \mathcal{H}$ taková, že $h(x_1) = y_1$ a $h(x_2) = y_2$.

Skládání modulo m

Lemma

- Nechť systém $\mathcal{H}$ je $(2, c)$-nezávislý z U do $[r]$ a $m \leq r$
- Pak $\mathcal{H} \mod m = \{x \to h(x) \mod m;\ h \in \mathcal{H}\}$ je $2c$-universální a $(2, 4c)$-nezávislý

Důkaz

- Zvolme různá $x_1, x_2 \in U$ a označme $y_1 = h(x_1)$ a $y_2 = h(x_2)$
- $2c$-universálnost:
  - $P[h(x_1) \equiv_m h(x_2)] = \sum_{y_1 \equiv_m y_2} P[h(x_1) = y_1 \text{ a } h(x_2) = y_2]$
  - $P[h(x_1) = y_1 \text{ a } h(x_2) = y_2] \leq c/r^2$
  - Sumu sčítáme přes r hodnot y_1 a $\lceil r/m \rceil$ hodnot y_2
  - $\lceil \frac{r}{m} \rceil \leq \frac{r+m-1}{m} \leq \frac{2r}{m}$
  - Celkem $P[h(x_1) \equiv_m h(x_2)] \leq \frac{c}{r^2} \cdot r \cdot \frac{2r}{m} = \frac{2c}{m}$
- $(2, 4c)$-nezávislost
  - $P[h(x_1) = z_1 \text{ a } h(x_2) = z_2] = \sum_{y_1 \equiv_m z_1 \text{ a } y_2 \equiv_m z_2} P[h(x_1) = y_1 \text{ a } h(x_2) = y_2]$
  - Sumu sčítáme přes nejvýše $\lceil r/m \rceil$ hodnot y_1 a y_2
  - Celkem $P[h(x_1) = z_1 \text{ a } h(x_2) = z_2] \leq \frac{c}{r^2} \cdot \left(\frac{2r}{m}\right)^2 = \frac{4c}{m^2}$

Multiply-mod-prime

Pozorování

- Nechť $p \geq |U|$ je prvočíslo, kde $U = [u]$
- Uvažujme hešovací funkci $h_{a,b}(x) = ax + b \bmod p$
- Pak systém $\mathcal{H} = \{h_{a,b};\ a, b \in [p]\}$ je $(2,1)$-nezávislý ①

Lemma

- Nechť systém $\mathcal{H}$ je $(2,c)$-nezávislý z U do $[r]$ a $r \geq m$
- Pak $\mathcal{H} \bmod p = \{x \to h(x) \bmod p;\ h \in \mathcal{H}\}$ je $2c$-universální a $(2,4c)$-nezávislý

Pozorování: Systém Multiply-mod-prime

- Nechť $p \geq |U| \geq m$ je prvočíslo, kde $U = [u]$
- $h_{a,b}(x) = (ax + b \bmod p) \bmod m$
- $\mathcal{H} = \{h_{a,b};\ a, b \in [p]\}$
- Systém $\mathcal{H}$ je 2-universální a (2,4)-nezávislý, ale není 3-nezávislý

1. Důkaz plyne z následujícího lemmatu, protože bijekce zaručuje, že pro různá $x_1, x_2 \in [p]$ a $y_1, y_2 \in [p]$ existuje právě jedna $h \in \mathcal{H}$ taková, že $h(x_1) = y_1$ a $h(x_2) = y_2$.

Zobecněné modulo m

Lemma

- Nechť systém $\mathcal{H}$ je (k, c)-nezávislý z U do $[r]$ a $r \geq 2km$
- Pak $\mathcal{H} \mod m = \{x \rightarrow h(x) \mod m;\ h \in \mathcal{H}\}$ je $(k, 2c)$-nezávislý

Důkaz

- Zvolme různá $x_1, \dots, x_k \in U$, $z_1, \dots, z_k \in M$ a označme $y_i = h(x_i)$
- $P[h(x_i) \mod m = z_i\ \forall i \in K] = \sum P[h(x_i) = y_i\ \forall i \in K]$
- Sumu sčítáme přes nejvýše $\lceil \frac{r}{m} \rceil \leq \frac{r+m-1}{m}$ hodnot $y_1, \dots, y_k$
- Z odhadu $1 + x \leq e^x$ plyne $\left(\frac{r+m-1}{m}\right)^k \leq \left(1 + \frac{m}{r}\right)^k \leq e^{km/r} \leq e^{1/2} \leq 2$
- $P[h(x_i) \mod m = z_i\ \forall i \in K] \leq \frac{c}{r^k} \cdot \left(\frac{r+m-1}{m}\right)^k \leq \frac{2c}{m^k}$

Poly-mod-prime

Věta z algebry: Jednoznačnost interpolace polynomem

Pro každé těleso T, $k > 1$ celočíselné, po dvou různá $x_1, \dots, x_k \in T$ a $y_1, \dots, y_k \in T$ libovolné existuje právě jeden polynom $p_a(x) = \sum_{i=0}^{k-1} a_i x^i$ stupně $k-1$ s koeficienty $a_0, \dots, a_{k-1} \in T$ takový, že $p(x_i) = y_i$ pro všechna $i \in K$.

Lemma

- Nechť systém $\mathcal{H}$ je (k, c)-nezávislý z U do $[r]$ a $r \geq 2km$
- Pak $\mathcal{H} \mod m = \{x \to h(x) \mod m;\ h \in \mathcal{H}\}$ je $(k, 2c)$-nezávislý

Pozorování: Systém Poly-mod-prime

- Nechť $\mathbb{Z}_p$ je těleso
- Systém $P_k = \{h_a;\ a \in \mathbb{Z}_p^k\}$ je $(k, 1)$-nezávislý, kde $h_a(x) = \sum_{i=0}^{k-1} a_i x^i$ ①
- Systém $P_k \mod m$ je $(k, 2)$-nezávislý pro $p \geq 2km$ ②

1. Důkaz přímo plyne z jednoznačnosti polynomu
2. Jestliže p je prvočíslo, tak hešovací funkci lze zapsat jako $h_a(x) = (\sum_{i=0}^{k-1} a_i x^i \mod p) \mod m$, kde aritmetické operace jsou nad celými čísly.

Datové struktury I

9. přednáška: Výběr hešovací funkce II

Jirka Fink

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

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

Zimní semestr 2022/23

Opakování

c-universální systém (ekvivalentní definice)

Systém hešovacích funkcí $\mathcal{H}$ je c-universální, jestliže pro všechna různá $x, y \in U$ ①

- počet hešovacích funkcí $h \in \mathcal{H}$ splňujících $h(x) = h(y)$ je nejvýše $\frac{c|\mathcal{H}|}{m}$
- náhodně zvolená $h \in \mathcal{H}$ splňuje $P[h(x) = h(y)] \leq \frac{c}{m}$ ② ③

(k, c)-nezávislý systém hešovacích funkcí

Nechť $k \in \mathbb{N}$, $K = \{1, \dots, k\}$ a $c \geq 1$.
Systém hešovacích funkcí $\mathcal{H}$ je (k, c)-nezávislý, pokud náhodně zvolená $h \in \mathcal{H}$ splňuje

$$P[h(x_i) = z_i \, \forall i \in K] \leq \frac{c}{m^k}$$

pro všechna po dvou různá $x_1, \dots, x_k \in U$ a všechna $z_1, \dots, z_k \in M$.

Lemma

- Nechť systém $\mathcal{H}$ je (k, c)-nezávislý z U do $[r]$ a $r \geq 2km$
- Pak $\mathcal{H} \bmod m = \{x \rightarrow h(x) \bmod m;\ h \in \mathcal{H}\}$ je $(k, 2c)$-nezávislý

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

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); }
```

Vlastnosti systému multiply-shift

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

Tabulkové hešování

Tabulkové hešování

- $\oplus$ značí bitový XOR
- Předpokládáme, že $u = 2^w$ a $m = 2^l$ a w je násobek $d \geq 2$
- Bitový zápis čísla $x \in U$ rozdělíme na d částí $x^1, \ldots, x^d$ po $\frac{w}{d}$ bitech
- Pro každé $i = 1, \ldots, d$ vybereme náhodnou hešovací funkci $T_i : [2^{w/d}] \to M$
- Hešovací funkce je $h(x) = T_1(x^1) \oplus \cdots \oplus T_d(x^d)$
- K vygenerování h potřebujeme $d \cdot 2^{w/d}$ náhodných čísel z rozsahu $M = [2^l]$

Ilustrativní příklad

- Uvažujme $w = 12$ a $d = 3$
- Nejprve vygeneruje náhodné funkce $T_1, T_2, T_3 : [2^4] \to M$
- Číslo $x = 101100111001$ rozdělíme na $x^1 = 1011$, $x^2 = 0011$ a $x^3 = 1001$
- Výpočet hešovací funkce je
  $h(x) = T_1(x^1) \oplus T_2(x^2) \oplus T_3(x^3) = T_1(1011) \oplus T_2(0011) \oplus T_3(1001)$

Univerzalita

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

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 $x^1, \dots, x^d$ po $\frac{w}{d}$ bitech
- Pro každé $i = 1, \dots, d$ vybereme náhodnou hešovací funkci $T_i : [2^{w/d}] \to M$
- Hešovací funkce je $h(x) = T_1(x^1) \oplus \cdots \oplus T_d(x^d)$

Univerzalita

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

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

- Mějme dva prvky x_1 a x_2 lišící se v i-tých částech
- Nechť $h_i(x) = T_1(x^1) \oplus \cdots \oplus T_{i-1}(x^{i-1}) \oplus T_{i+1}(x^{i+1}) \oplus \cdots \oplus T_d(x^d)$
- $P[h(x_1) = z_1] = P[h_i(x_1) \oplus T_i(x_1^i) = z_1] = P[T_i(x_1^i) = z_1 \oplus h_i(x_1)] = \frac{1}{m}$ 1
- Náhodné jevy $h(x_1) = z_1$ a $h(x_2) = z_2$ jsou nezávislé
  - Náhodné proměnné $T_i(x_1^i)$ a $T_i(x_2^i)$ jsou nezávislé
  - Náhodné jevy $T_i(x_1^i) = z_1 \oplus h_i(x_1)$ a $T_i(x_2^i) = z_2 \oplus h_i(x_2)$ jsou nezávislé
- $P[h(x_1) = z_1 \text{ a } h(x_2) = z_2] = P[h(x_1) = z_1]P[h(x_2) = z_2] = \frac{1}{m^2}$

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

Tabulkové hešování

Tabulkové hešování není 4-nezávislé

1. Zvolíme prvky x_1, x_2, x_3 a x_4 takové, že
   - části x_1 splňují $x_1^1 = 0$, $x_1^2 = 0$, $x_1^i = 0$ pro $i \geq 3$
   - části x_2 splňují $x_2^1 = 1$, $x_2^2 = 0$, $x_2^i = 0$ pro $i \geq 3$
   - části x_3 splňují $x_3^1 = 0$, $x_3^2 = 1$, $x_3^i = 0$ pro $i \geq 3$
   - části x_4 splňují $x_4^1 = 1$, $x_4^2 = 1$, $x_4^i = 0$ pro $i \geq 3$
2. Platí $h(x_1) \oplus h(x_2) \oplus h(x_3) = h(x_4)$
3. Zvolme libovolná z_1, z_2, z_3 a nechť $z_4 = z_1 \oplus z_2 \oplus z_3$
4. Jestliže $h(x_1) = z_1$, $h(x_2) = z_2$ a $h(x_3) = z_3$, pak platí $h(x_4) = z_4$
5. $P[h(x_1) = z_1 \text{ a } h(x_2) = z_2 \text{ a } h(x_3) = z_3 \text{ a } h(x_4) = z_4] = \frac{1}{m^3} > \frac{c}{m^4}$

Scalar-mod-prime

- Chceme hešovat d-tici $x_1, \dots, x_d \in \mathbb{Z}_p$, kde p je prvočíslo
- $\left\{x_1, \dots, x_d \to \sum_{i=1}^d a_i x_i \mod p;\ a \in \mathbb{Z}_p^d\right\}$ je 1-universální
- $\left\{x_1, \dots, x_d \to b + \sum_{i=1}^d a_i x_i \mod p;\ a \in \mathbb{Z}_p^d, b \in \mathbb{Z}_p\right\}$ je (2,1)-nezávislý
- $\left\{x_1, \dots, x_d \to \left(b + \sum_{i=1}^d a_i x_i \mod p\right) \mod m;\ a \in \mathbb{Z}_p^d, b \in \mathbb{Z}_p\right\}$ je (2,4)-nezávislý

Důkaz 1-universálnosti ①

- Mějme různé $x, y \in \mathbb{Z}_p^d$ a BÚNO předpokládejme, že $x_1 \neq y_1$
- $P[a \cdot x \equiv_p a \cdot y] = P[a \cdot (x - y) \equiv_p 0] = P\left[a_1 \equiv_p \frac{\sum_{i=2}^d a_i(y_i - x_i)}{x_1 - y_1}\right] = 1/p$ ②

1. Pro 2-nezávislost stačí podobně nahlednout, že a_1, b jsou jednoznačně určené.
2. Náhodná proměnná a_1 musí nabývat jednu konkrétní hodnotu, což nastane s pravděpodobností $1/p$.

Hešování řetězců různých délek

Poly-mod-prime pro různě dlouhé řetězce I

- Chceme hešovat řetězec $x_1, \dots, x_d \in \mathbb{Z}_p$, kde p je prvočíslo
- $\left\{x_1, \dots, x_d \to \sum_{i=0}^{d-1} x_{i+1} a^i \bmod p;\ a \in [p]\right\}$ je d-universální
- Dva různé polynomy stupně nejvýše $d-1$ mají nejvýše d společných bodů, takže existuje nejvýše d kolidujících hodnot a.

Poly-mod-prime pro různě dlouhé řetězce II

- Chceme hešovat řetězec $x_1, \dots, x_d \in U$ do M, kde $p \geq m$ je prvočíslo
- $h_{a,b,c}(x_1, \dots, x_d) = \left(b + c \sum_{i=0}^{d-1} x_{i+1} a^i \bmod p\right) \bmod m$
- $\mathcal{H} = \{h_{a,b,c};\ a, b, c \in [p]\}$
- $P[h_{a,b,c}(x_1, \dots, x_d) = h_{a,b,c}(x'_1, \dots, x'_{d'})] \leq \frac{2}{m}$ pro různé řetězce délek $d, d' \leq \frac{p}{m}$.

Bloom filtry (Bloom, 1970)

Cíl

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

Postup

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

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

Cíl

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

Analýza

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

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

Counting Bloom filtry (Fan et al. 2000)

Cíl

Chtěli bychom v Bloom filtru umět mazat prvky.

Postup

- Na každé pozici v tabulce nebudeme mít jeden bit ale malý čítač
- Při operace INSERT se čítače $h_1(x), \dots, h_k(x)$ zvýší o jedna
- Při operaci DELETE se čítače $h_1(x), \dots, h_k(x)$ sníží o jedna
- Jestliže některý z čítačů $h_1(y), \dots, h_k(y)$ je nulový, pak y není přítomen
- Zvolíme $k = \frac{m}{n} \log 2$
- Jestliže všechny čítače $h_1(y), \dots, h_k(y)$ jsou kladné, pak y není přítomen s pravděpodobností přibližně $0.69^{\frac{m}{n}}$

Jak velký zvolit čítač, aby nedocházelo k přetečení?

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

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

Příští týden

Další způsoby řešení kolizí.

Datové struktury I

10. přednáška: Řešení kolizí v hešování

Jirka Fink

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

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

Zimní semestr 2022/23

Hešování se separovanými řetězci

Popis

V přihrádce j jsou uloženy všechny prvky $i \in S$ splňující $h(i) = j$.

Pozorování

Jestliže $\mathcal{H}$ je c-universální, pak očekávaný počet prvků v přihrádce $h(x)$ pro $x \in U$ je nejvýše $\frac{cn}{m}$.

Důsledek

Jestliže $\mathcal{H}$ je c-universální a $m = \Omega(n)$, pak očekávaná složitost operací FIND, INSERT a DELETE je $\mathcal{O}(1)$.

Jaký je maximální počet prvků v jedné přihrádce?

- S velmi malou ale kladnou pravděpodobností padnou všechny prvky do jedné přihrádky
- Jaký je očekávaný počet prvků v nejplnější přihrádce?

Hešování se separovanými řetězci: Nejdelší řetězec

Definice

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

Značení

Nechť A_j je počet prvků v j-té přihrádce.

Věta: Délka nejdelšího řetězce

Pokud $m = \Theta(n)$ 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.

Poznámka

Dokážeme, že $P[\max_j A_j \leq (1+\epsilon)\frac{\log n}{\log \log n}] > 1 - \frac{1}{n^{\frac{\epsilon}{3}}}$ pro všechna $\epsilon > 0$ a dostatečně velká n. 1

Důsledek: Očekávaná délka nejdelšího řetězce

Pokud $\alpha = \Theta(1)$ a systém hešovacích funkcí je úplně náhodný, pak očekávaná délka nejdelšího řetězce je $E[\max_{j \in M} A_j] = \Theta(\frac{\log n}{\log \log n})$. 2

1. Nebudeme dokazovat, že $\max_{j \in M} A_j = \Omega(\frac{\log n}{\log \log n})$ s velkou pravděpodobností.
2. Pro $\epsilon = 3$ dostáváme $P[\max_j A_j \leq 4\frac{\log n}{\log \log n}] > 1 - \frac{1}{n}$. Tedy $E[\max_{j \in M} A_j] \leq P[\max_j A_j \leq 4\frac{\log n}{\log \log n}] \cdot 4\frac{\log n}{\log \log n} + P[\max_j A_j > 4\frac{\log n}{\log \log n}] \cdot n \leq 4\frac{\log n}{\log \log n} + 1$.
   Důkaz $E[\max_{j \in M} A_j] = \Omega(\frac{\log n}{\log \log n})$ vynecháváme.

Hešování se separovanými řetězci: Nejdelší řetězec

Chernoffův odhad

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

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

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

1. I_{ij} je náhodná proměnná indikující, zda i-tý prvek patří do j-té přihrádky
2. Platí $A_j = \sum_{i \in S} I_{ij}$
3. Mějme $\epsilon > 0$
4. Označme $\mu = E[A_1] = n/m$
5. Dále $c = (1+\epsilon)\frac{\log n}{\mu \log\log n}$
6. Platí $P[\max_j A_j > c\mu] = P[\exists j : A_j > c\mu] \leq \sum_j P[A_j > c\mu] = mP[A_1 > c\mu]$
7. Aplikujeme Chernoffův odhad na proměnné I_{i1} pro $i \in S$
8. Platí $P[\max_j A_j > c\mu] \leq mP[A_1 > c\mu] < m\frac{e^{(c-1)\mu}}{c^{c\mu}} = me^{-\mu}e^{c\mu - c\mu \log c}$

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

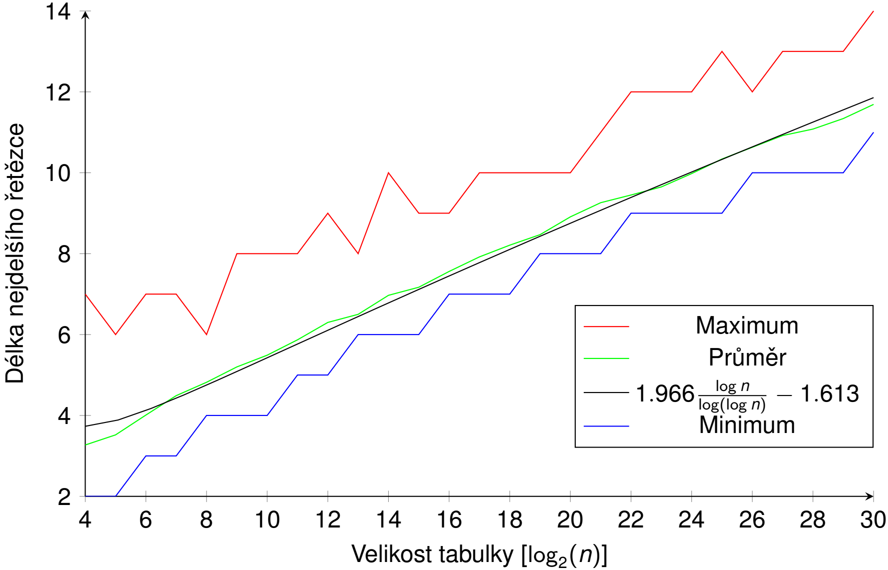
- Označili jsme $c = (1+\epsilon)\frac{\log n}{\mu \log\log n}$ a odvozujeme

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

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

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

Délka nejdelšího řetězce ①



1. Hodíme n míčů do n košů. Kolik míčů je v nejplnějším koši v závislosti na n? Pro každé n provádíme 100 experimentů, ze kterých se díváme na minimum, maximum a průměr. Interpolace funkcí $1.966\frac{\log n}{\log(\log n)} - 1.613$ má chybu (součet čtverců rozdílu) 0.682 a pro funkci $0.466 \log n + 2.254$ je chyba 0.68, takže na ověření správnosti odhadu $\frac{\log n}{\log(\log n)}$ bychom potřebovali zkoušet větší hodnoty n, ale na to už nemáme dost paměti.

Lineární přidávání

Cíl

- Chtěli bychom ušetřit paměť, a tak prvky budeme ukládat přímo do tabulky
- V jedné přihrádce může být jen jeden prvek

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)
- konstantní pro $\log(n)$-nezávislý systém (Schmidt, Siegel, 1990)
- $\mathcal{O}\left(\frac{1}{\epsilon^{\frac{13}{6}}}\right)$ pro 5-nezávislý systém (Pagh, Pagh, Ruzic, 2007)
- $\mathcal{O}(\log n)$ pro 4-nezávislý systém (Pătraşcu, Thorup, 2010) 1
- $\mathcal{O}\left(\frac{1}{\epsilon^2}\right)$ pro tabulkové hešování (Pătraşcu, Thorup, 2012)
- $\mathcal{O}(\log n)$ pro multiply-shift

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

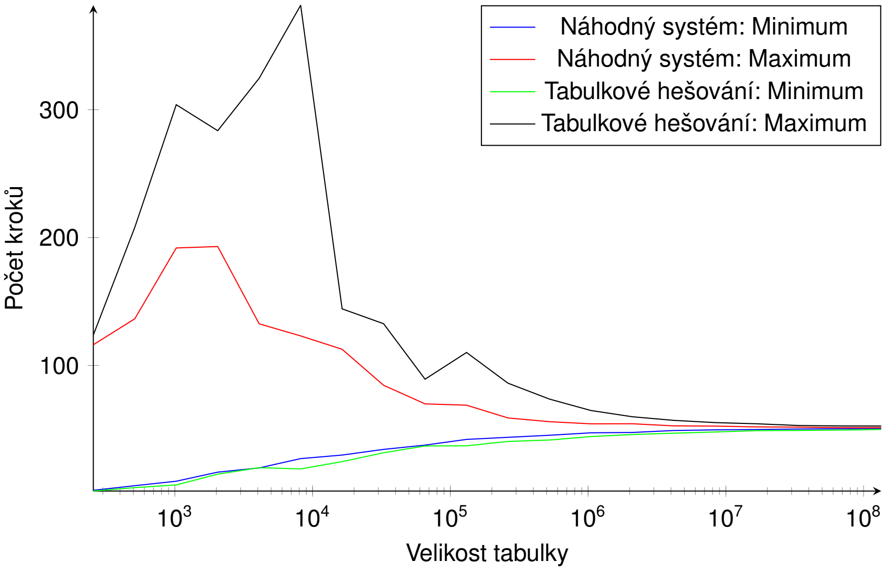
Počet prvků od dané přihrádky do nejbližší volné přihrádky

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

Důkaz

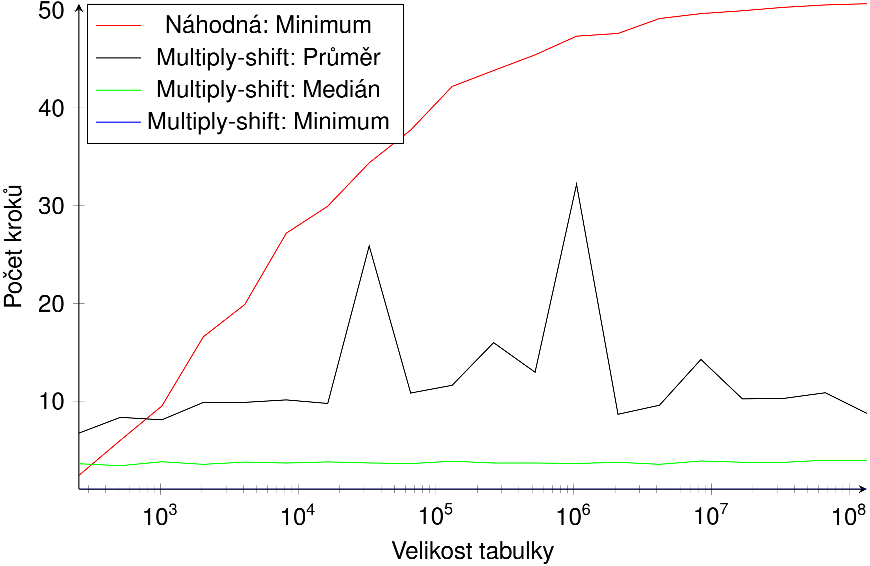
1. Nechť $1 < c < \frac{1}{\alpha}$ a $q = \left(\frac{e^{c-1}}{c^c}\right)^\alpha$
   - Platí $0 < q < 1$ ②
2. Nechť $p_t = P[|\{x \in S;\ h(x) \in T\}| = t]$ je pravděpodobnost, že do dané množiny přihrádek T velikosti t je zahešováno t prvků. Pak $p_t < q^t$. ③
   - Nechť X_i je náhodná proměnná indikující,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

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 spočítal očekávanou složitost přesně, ale výpočet je náročný.
2. Zjevně $q > 0$. Chceme dokázat, že $\frac{e^{c-1}}{c^c} = e^{c-1-c\log c} < 1$. Musíme tedy dokázat, že $c - 1 - c\log c < 0$ pro $c > 1$. Pro $c = 1$ máme $c - 1 - c\log c = 0$, abychom dokázali ostrou nerovnost pro $c > 1$, ukážeme, že funkce $c - 1 - c\log c$ je pro $c > 1$ klesající. Derivace $1 - \log c - 1$ je záporná pro $c > 1$.
3. Zde uvažujeme prvky, které hešovací funkce zobrazí do daných přihrádek, a nikoliv prvky, které se do daných přihrádek dostanou vlivem lineárního přidávání.
4. Tedy přihrádky $b - s$ až $b + k - 1$ jsou obsazeny pro nějaké s. Indexy přihrádek počítáme modulo m.

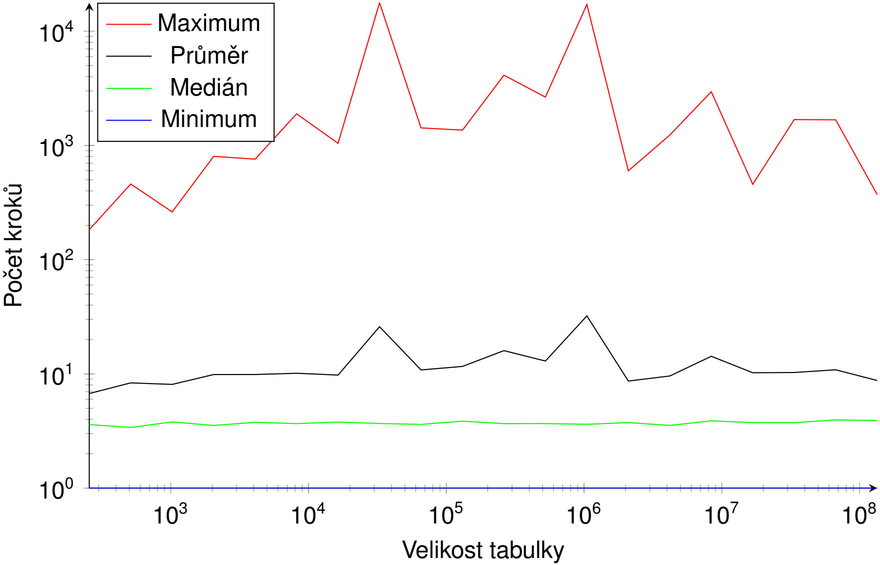

Náhodný systém: Minimum
Náhodný systém: Maximum
Tabulkové hešování: Minimum
Tabulkové hešování: Maximum
Počet kroků
300
200
100
10^3
10^4
10^5
10^6
10^7
10^8
Velikost tabulky

1. Počet kroků při vkládání do tabulky lineárního přidávání při 90% zaplnění. Nejprve vložíme prvky $1, \dots, \lfloor 0.89m \rfloor$ a poté počítáme průměrný počet kroků při vkládání prvků $\lfloor 0.89m + 1 \rfloor, \dots, \lfloor 0.91m \rfloor$. Z jednoho experimentu dostaneme jeden průměrný počet kroků a experiment opakujeme 1000-krát pro různé hešovací funkce. Grafy ukazují statistické údaje těchto experimentů.

Lineární přidávání: Náhodná hešovací funkce a Multiply-shift



Lineární přidávání: Multiply-shift



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.

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 $\mathcal{O}\left(\frac{\log \log n}{\log k}\right)$.

Kukačkové hešování

Popis

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

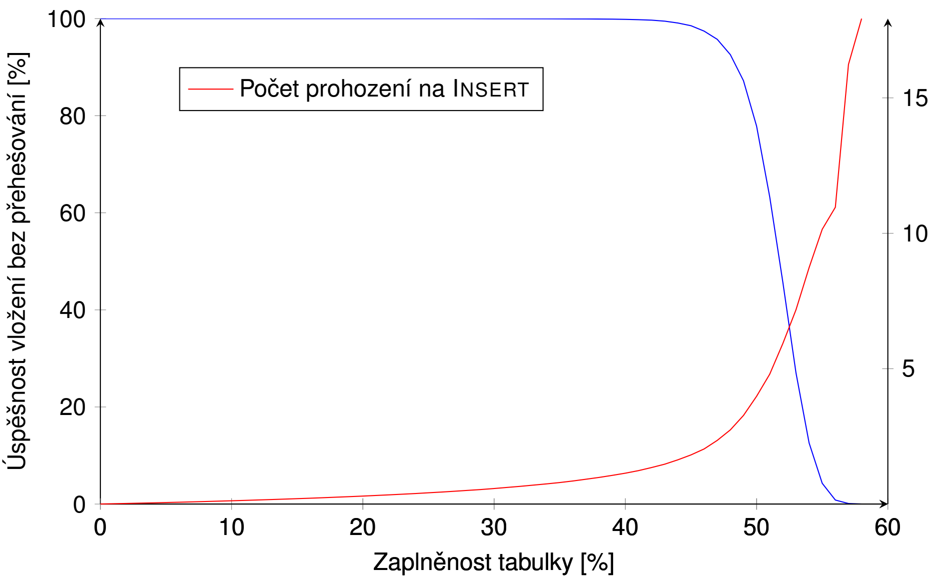
Pagh, Rodler, 2004

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, 2012

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.

Vkládání prvků do kukačky ①



1. Vkládáme prvky do kukaččí tabulky velikosti 10^4 a skončíme, když dojde k zacyklení. Uvažujeme zcela náhodnou hešovací funkci, takže každému prvku jsou přiřazeny dvě náhodné pozice. Opakujeme 10^4-krát s různými pozicemi prvků. Úpěšnost udává, kolika pokusům se podařilo docílit dané zaplněnosti. Druhou křivnou je průměrný počet prohození v operaci INSERT pro danou zaplněnost.

Příští týden

Hledání jehly v kupce sena

Datové struktury I
11. přednáška: Sufixová pole

Jirka Fink

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

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

Zimní semestr 2022/23

Hledání jehly v kupce sena

- Máme daný velmi dlouhý text (seno) a krátký text (jehla)
- Úkolem je najít všechny výskyty jehly v kupce sena
- Příklad: v seně bananas se jehla ana vyskytuje hned dvakrát
- V seně anna se tatáž jehla nevyskytuje vůbec

Algoritmy na hledání jedné jehly ve velmi velkém senu

- Knuth, Morris, Pratt
- Aho, Corasick: Více předem daných jehel
- Robin, Karp: Randomizovaný algoritmus

Opačný přístup

Máme pevně dané seno a chceme hledat různé jehly.

Značení

Abeceda

- Σ je konečná množina znaků (písmen)
- $\Sigma^\star$ je množina konečných posloupností (slov, řetězců) nad Σ
- ε je speciální prázdné slovo
- \$ je speciální znak značící konec slova

Pro slovo $\alpha \in \Sigma$ značíme ①

- $|\alpha|$ délku α
- $\alpha[k]$ je k-tý znak slova α, počítáno od 0 do $|\alpha| - 1$
- $\alpha[k : l]$ je podslovo začínající k-tým znakem a končící těsně před l-tým
- $\alpha[: l]$ je prefix prvních l znaků
- $\alpha[k :]$ je sufix začínající k-tým znakem

Motivace práce se všemi sufixy

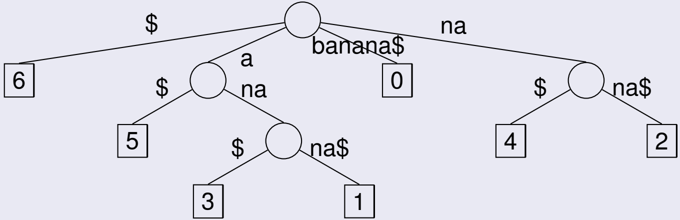
Výskyt jehly znamená, že nějaký sufix sena začíná jehlou

1. Značení je podobné jako v Python.

Definice

(Komprimovaný) sufixový strom je (komprimovaná) trie obsahující všechny sufixy, kde synové vrcholů jsou lexikografickém pořadí. ①

Komprimovaný sufixový strom pro slovo banana$ ②



Čísla v listech značí počáteční pozici sufixu.

Dotazy

- Nalezení všech výskytů jehly v kupce sena ③
- Nalezení nejdelšího podslova s dvěma (více) výskyty ④
- Nalezení nejdelšího společného podslova dvou slov ⑤

1. Hloubkou vrcholu v komprimované trii rozumíme hloubkou odpovídajícího vrcholu v nekomprimované trii, tj. počet písmen na cestě z kořene do vrcholu.
2. K uložení komprimovaného sufixového stromu potřebujeme $\mathcal{O}(|\alpha|)$ paměti, protože každému listu odpovídá sufix a vnitřních vrcholů je méně než listů.
3. Vyhledáme jehlu v sufixovém stromu a listy v podstromu udávají všechny výskyty. Chceme-li znát jen četnost, tak si stačí navíc ve všech vrcholech pamatovat počet listů v podstromu.
4. Vyhledáme nejhlubší vnitřní vrchol, kde hloubkou rozumíme počet písmen na cestě z kořene do vrcholu.
5. Pro slova $\alpha, \beta \in \Sigma^\star$ vytvoříme sufixový strom pro slovo $\alpha\#\beta\$$, kde hledáme nejhlubší vrchol obsahující v podstromu jak sufix obsahující #, tak i neobsahující.

Sufixové pole

- Sufixové pole X udává lexikografické pořadí sufixů daného slova α
  $X[i]$ říká, kde v řetězci začíná i-tý sufix v lexikografickém pořadí
- Rankové pole R je inverzní k X, takže platí $X[R[i]] = i$
  Hodnota $R[i]$ říká, kolikátý v lexikografickém pořadí je sufix $\alpha[i:]$
- $LCP(\alpha, \beta)$ udává délku nejdelšího společného prefixu α a β 1
- Pole společných prefixů (LCP) L udává délku společného prefixu mají sufixy sousedící v lexikografickém pořadí
  Tedy $L[i] = LCP(\alpha[X[i]:], \alpha[X[i+1]:])$

1. LCP = longest common prefix

Sufixové pole: Příklad pro slovo barokoarokoko

i	X[i]	R[i]	L[i]	sufix
0	13	3	0	ε
1	1	1	5	arokoarokoko
2	6	12	0	arokoko
3	0	10	0	barokoarokoko
4	11	5	2	ko
5	4	8	2	koarokoko
6	9	2	0	koko
7	12	13	1	o
8	5	11	1	oarokoko
9	10	6	3	oko
10	3	9	3	okoarokoko
11	8	4	0	okoko
12	2	7	4	rokoarokoko
13	7	0	–	rokoko

- Sufixové pole X udává lexikografické pořadí sufixů daného slova α
- Rankové pole R je inverzní k X, takže platí $X[R[i]] = i$
- Pole společných prefixů LCP $L[i] = LCP(\alpha[X[i]:], \alpha[X[i+1]:])$

Konstrukce sufixového stromu a pole

Motivace sufixového pole

- Sufixové pole potřebuje méně paměti

Cíle

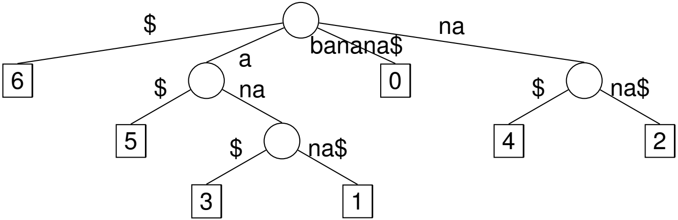
- Vytvořit pole R z X a opačně 1
- Vytvořit sufixový strom z polí X a L a opačně
- Vytvořit pole X a L
- Upravit algoritmy hledání v textu, aby používali pole místo stromu

1. Tento krok je triviální, protože jen vytváříme inverzní permutaci.

Souvislost sufixového stromu a pole

Pozorování

- Sufixové pole udává DFS pořadí listů sufixového stromu
- LCP pole $L[i]$ udává hloubku nejbližšího společného předka listů $X[i]$ a $X[i+1]$



i	X[i]	R[i]	L[i]	suffix
0	6	4	0	ε
1	5	3	1	a
2	3	6	3	ana
3	1	2	0	anana
4	0	5	0	banana
5	4	1	2	na
6	2	0	–	nana

Převod sufixového pole a stromu

Pozorování

- Sufixové pole udává DFS pořadí listů sufixového stromu
- LCP pole $L[i]$ udává hloubku nejbližšího společného předka listů $X[i]$ a $X[i+1]$

Konstrukce sufixového pole a LCP ze stromu

Obě pole vytváříme při průchodu stromu do hloubky

Konstrukce sufixového stromu z pole a LCP

Strom vytváříme průchodem do hloubky

- Nejprve vytvoříme list pro slovo ε
- Po vytvoření listu pro sufix $X[i]$
  - Vynoříme se do vrcholu v hloubce $L[i]$
  - Přidáme list pro sufix $X[i]$ v hloubce $|\text{seno}| - X[i]$

Časová složitost

Složitost těchto převodů je lineární pro komprimovaný sufixový strom

Vytvoření LCP pole ze sefixového pole (Kasai, 2001)

Pozorování

Pro každé slovo α a pro všechna $i = 0, \ldots, |\alpha| - 1$ platí $L[R[i+1]] \geq L[R[i]] - 1$.

Důkaz

- Zřejmé pro $L[R[i]] \leq 1$
- Zřejmé pro $X[R[i+1]+1] = X[R[i]+1]+1$ ①
- Jinak platí $\alpha[i+1:] < \alpha[X[R[i+1]+1]:] < \alpha[X[R[i]+1]:]$ ②
- Platí

$$
\begin{aligned}
L[R[i+1]] = & \quad LCP(\alpha[i+1:], \alpha[X[R[i+1]+1]:]) \\
\geq & \quad LCP(\alpha[i+1:], \alpha[X[R[i]+1]:]) \\
= & \quad LCP(\alpha[i:], \alpha[X[R[i]]:]) - 1 \\
= & \quad L[R[i]] - 1
\end{aligned}
$$

1. i je pozice ve slovu α, $R[i]$ udává pořadí sufixu na pozici i, $X[R[i]+1]$ udává pozici následníka v lexikografickém uspořádání
2. Sufix začínající na pozici $X[R[[i+1]+1]]$ je za sufixem začínající na pozici $i+1$, ale před sufixem na pozici $X[R[[i]+1]]$

Vytvoření LCP pole ze sefixového pole (Kasai, 2001)

Pozorování

Pro každé slovo α a pro všechna $i = 0, \ldots, |\alpha| - 1$ platí $L[R[i+1]] \geq L[R[i]] - 1$.

Algoritmus

$l = 0$
for $i = 0, \ldots, |\alpha| - 1$ **do**
  $l = \max(0, l - 1)$
  $j = X[R[i] + 1]$
  while $i + l < |\alpha|$ *&&* $j + l < |\alpha|$ *&&* $\alpha[i + l] == \alpha[j + l]$ **do**
    $l = l + 1$
  $L[R[i]] = l$

Časová složitost je $\mathcal{O}(|\alpha|)$

- Vnější cyklus má složitost $\mathcal{O}(|\alpha|)$
- Ve vnitřním cyklu se l vždy zvyšuje o 1
  - Hodnata l začíná na 0
  - Hodnota l na konci je nejvýše $\mathcal{O}(|\alpha|)$
  - Vnější cyklus hodnotu l sníží dohromady o nejvýše $\mathcal{O}(|\alpha|)$

Vytvoření pole R zdvojováním

Popis algoritmu

1. $R_k[i]$ je počet j takových, že $\alpha[j : j + k] < \alpha[i : i + k]$ ① ②
2. Cílem je spočítat $R_k[i] = R[i]$ pro nějaké $k \geq |\alpha|$
3. K výpočtu R_1 jen počítáme počet výskytů jednotlivých písmen
4. $\alpha[i : i + 2k] < \alpha[j : j + 2k]$ právě tehdy, když
   $\alpha[i : i + k] < \alpha[j : j + k] \vee (\alpha[i : i + k] = \alpha[j : j + k] \,\&\, \alpha[i + k : i + 2k] < \alpha[j + k : j + 2k])$
5. $R_{2k}[i] < R_{2k}[j]$ právě tehdy, když
   $R_k[i] < R_k[j] \vee (R_k[i] = R_k[j] \,\&\, R_k[i + k] < R_k[j + k])$
6. K získání R_{2k} třídíme dvojice $(R_k[i], R_k[i + k])$ pro $i = 0, \dots, |\alpha| - 1$

Časová složitost

1. Třídění pro získání R_0 máme v čase $\mathcal{O}(|\alpha| \log |\alpha|)$ ③
2. Následuje $\mathcal{O}(\log n)$ přihrádkového třídění
3. Celková složitost je $\mathcal{O}(|\alpha| \log |\alpha|)$
4. Kärkkäinen, Sanders, 2003: Konstrukce v lineárním čase

1. Porovnáváme tedy jen prvních k znaků sufixů.
2. Pokud v zápisu $\alpha[i:j]$ horní index přesahuje délku slova, pak $\alpha[i:j]$ značí jen $\alpha[i:]$.
3. Pokud je $|\Sigma| \leq |\alpha|$, pak můžeme použít přihrádkové třídění a docílit tím času $\mathcal{O}(|\alpha|)$.

Vytvoření sufixového pole zdvojováním

Vytvoříme pole $X[0 \dots n]$ a $R[0 \dots n]$.

`# Báze: vytvoření` R_0

$D = \{(\alpha[i], i);\ i = 0, \dots, n\}$ setříděné lexikograficky

for $j = 0, \dots, n$ **do**

- $X[j] = D[j][1]$
- **if** $j = 0$ *nebo* $D[j][0] \neq D[j-1][0]$ **then**
  - $R[X[j]] = j$
- **else**
  - $R[X[j]] = R[X[j-1]]$

`# Indukční krok: vytvoření` R_{2k} `z` R_k

for $(k = 0; k < n; k = 2k)$ **do**

- $D = \{(R[i], R[i+k], i);\ i = 0, \dots, n\}$ setříděné lexikograficky
- **for** $j = 0, \dots, n$ **do**
  - $X[j] = D[j][2]$
  - **if** $j = 0$ *nebo* $(D[j][0], D[j][1]) \neq (D[j-1][0], D[j-1][1])$ **then**
    - $R[X[j]] = j$
  - **else**
    - $R[X[j]] = R[X[j-1]]$

Použití sufixového pole

- Jak najít nejdelší opakující se podřetězec?
- Jak určit počet různých podřetězců délky k?
- Jak najít jehlu v kupce sena?

Příští týden

Hledání bodů v rovině i více-dimenzionálním prostoru.

Datové struktury I

12. přednáška: Geometrické datové struktury

Jirka Fink

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

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

Zimní semestr 2022/23

Intervalový dotaz (Range query)

Popis problému

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

Aplikace

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

Intervalový dotaz v $\mathbb{R}^1$

Staticky

Body uložíme do pole

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

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

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

k je počet vyjmenovaných bodů

Dynamicky

Body uložíme do vyhledávacího stromu

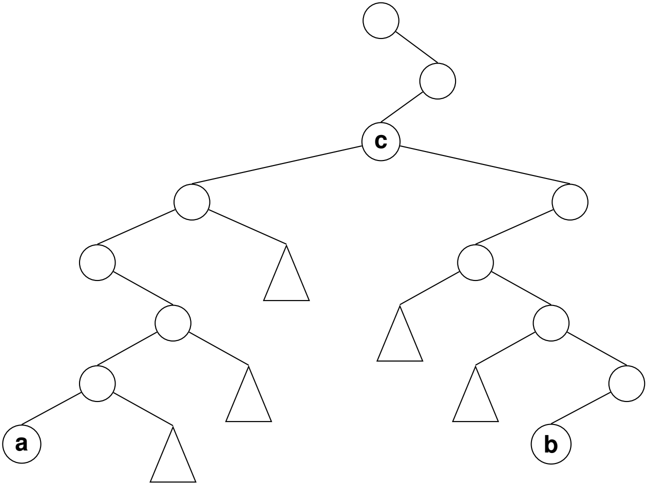
BUILD: $\mathcal{O}(n \log n)$

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

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

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

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

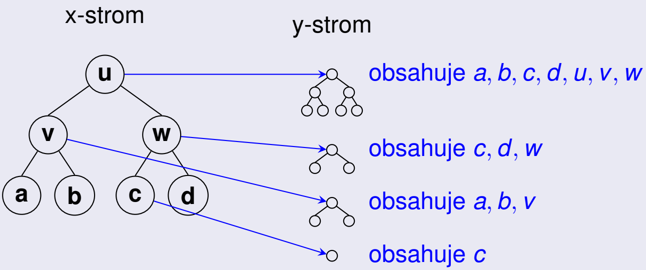


Vrchol *a* je nejmenší prvek v intervalu, *b* je největší prvek v intervalu a *c* je poslední společný vrchol na cestách z kořene do vrcholů *a* a *b*.

Konstrukce

- Vybudujeme binární vyhledávací strom podle x-ové souřadnice bodů (x-strom)
- Nechť S_u je množina bodů v podstromu vrcholu u
- Každý vrchol u vybudujeme jeden binární vyhledávací strom podle y-ové souřadnice obsahující S_u
- V každém vrcholu je uložen jeden bod ① ②

Příklad



atd.

1. Podobně jako ve vyhledávacích stromech můžeme uvažovat dvě varianty: prvky mohou být uloženy ve všech vrcholech nebo jen v listech. V intervalových stromech předpokládáme, že v každém vrcholu je uložen jeden bod.
2. Pozor! Každý bod je uložen v několika vrcholech, takže paměťová složitost není lineární.

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

Kde všude je uložený jeden vrchol?

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.

Předpoklad

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

Paměťová složitost

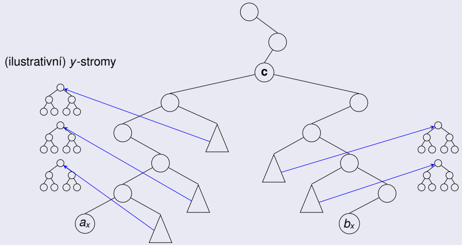
Každý bod je uložen v $\mathcal{O}(\log n)$ y-stromech a celková paměťová složitost je $\mathcal{O}(n \log n)$.

Intervalové stromy v $\mathbb{R}^2$: Dotaz na interval $\langle a_x, b_x \rangle \times \langle a_y, b_y \rangle$

Idea algoritmu vyhledávání

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

Příklad



Složitost dotazu Count

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

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

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

Popis

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

Reprezentace

Struktura vrcholu intervalového stromu obsahuje

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

Poznámka

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

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

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

V kolika vrcholech je uložený bod b?

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

Kolik má intervalový strom i-stromů?

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

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

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

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

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

Složitost jednoho volání funkce Build (bez rekurze)

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

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

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

Vytvoření d-stromů

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

Vytvoření všech i-stromů pro $i = 1, \dots, d-1$ (nepočítaje $(i+1)$-stromy)

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

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

Časová složitost operace BUILD

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

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

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

Složitost operace COUNT

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

Složitost operace QUERY

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

BB[α]-stromy

BB[α]-strom

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

Operace Insert (Delete je analogický)

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

Amortizovaná časová složitost operací Insert a Delete

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

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 přidání/smazání prvků v podstromu 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 $\sum_{i=1}^{d} \mathcal{O}\left(\log^{i-1} n\right)\mathcal{O}(\log n)\mathcal{O}\left(\log^{d-i} n\right) = \mathcal{O}\left(\log^d n\right)$

Další známá vylepšení intervalových stromů

Popsaný postup

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

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

S použitím Fractional cascading

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

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

Chazelle, 1990

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, 1986 (pro $d \geq 3$)

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

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

Popis

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

Algoritmus

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

k-d stromy: Operace QUERY COUNT: Analýza

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

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

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

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

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

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

Po Vánocích

Paralelní programování bez zámků.

Datové struktury I

13. přednáška: Paralelní programování bez zámků

Jirka Fink

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

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

Zimní semestr 2022/23

Přičítání jedničky ke sdílenému čítači

Proč následující postup přičtení jedničky nefunguje, když k čítači přistupuje více vláken najednou?

local_copy $\leftarrow$ shared_counter
local_copy $\leftarrow$ local_copy + 1
shared_counter $\leftarrow$ local_copy

Zámky (mutex) a vzájemné čekání (deadlock)

Zámek

Zabraňuje tomu, aby byly současně vykonávány dva (nebo více) kritické kódy nad stejným sdíleným prostředkem, jako například globální proměnné.

Granularita zámků v datových strukturách

- Jeden globální zámek
- Jeden zámek pro celou instanci datové struktury
- Zámek pro každou přihrádku (např. u hešování)
- Zámek pro každý prvek (např. ve stromu)

Vzájemné čekání (deadlock)

Úspěšné dokončení první akce je podmíněno předchozím dokončením druhé akce, přičemž druhá akce může být dokončena až po dokončení první akce. Příklad:

```
lock(A)
lock(B)
critical section
unlock(B)
unlock(A)
```

```
lock(B)
lock(A)
critical section
unlock(A)
unlock(B)
```

Řešení: Určit globální uspořádání zámků a vždy zamykat v tomto pořadí

Nevýhody zámku

1. Deadlock
2. Spravedlnost: Které vlákno získá uvolněný zámek?
3. Priorizace: Důležitější vlákno by nemělo čekat na pomalejší
4. Výkon: Zámky zpomalují výpočet, zejména při jejich větším využívání
5. Náchylnost na chyby: Při selhání jednoho vlákna může dojít k neuvolnění zámku

Atomické operace

Příklady

- Read and write: Prvek je možné celý najednou načíst a zapsat.
- Exchange: Prohození obsahu atomické a lokální proměnné.
- Test and set bit: Nastaví atomickou binární proměnnou na true a vrátí původní hodnotu.
- Fetch and add: Přičte k atomické proměnné danou hodnotu a vrátí původní hodnotu.
- Compare and swap (CAS): Pro atomický register R a hodnoty a a b nastaví R na b, pokud $R = a$, a vždy vrátí původní hodnotu.
- Load linked and store conditional (LL/SC): LL hodnotu registru R do lokální proměnné L a následná SC zapíše hodnotu z L do R, pokud mezitím nedošlo k jinému přístupu do R, jinak vrátí chybu.

Pořadí vykonávaní operací není dané, ale je konzistentní.

Cvičení

Které z těchto atomických operací lze použít k implementaci zámků?

Zásobník bez zámků

Je tato implementace zásobníku správná?

```
1  Function push(node)
2      while true do
3          h ← head
4          node.next ← h
5          if CAS(head, h, node) == h then
6              return
```

```
7  Function pop()
8      while true do
9          h ← head
10         n ← h.next
11         if CAS(head, h, n) == h then
12             return h
```

Problém Livelock

Sice máme jistotu, že vždy alespoň jedno uspěje, ale teoreticky může jiné vlákno donekonečna cyklit.

Problém ABA

První vlákno je přerušeno mezi řádky 10 a 11, kdy druhé vlákno provede

```
A ← pop
B ← pop
push(A)
```

Zásobník bez zámků

Je tato implementace zásobníku správná?

```
Function push(node)
    while true do
        h ← head
        node.next ← h
        if CAS(head, h, node) == h then
            return
```

```
Function pop()
    while true do
        h ← head
        n ← h.next
        if CAS(head, h, n) == h then
            return h
```

Řešení problému ABA

- Použít LL/CS místo CAS
- Použít Wide CAS (Double CAS): pracuje s dvojicí sousedních buněk paměti
  V zásobníku máme za ukazatelem head uložení timestamp a testujeme

```
(h,t) ← (head,timestamp)
node.next ← h
if CAS((head,timestamp), (h,t), (node,t+1)) == (h,t) then
    return
```

Zásobník bez zámků

Je tato implementace zásobníku správná?

```
1  Function push(node)
2      while true do
3          h ← head
4          node.next ← h
5          if CAS(head, h, node) == h then
6              return
```

```
7  Function pop()
8      while true do
9          h ← head
10         n ← h.next
11         if CAS(head, h, n) == h then
12             return h
```

Problém dealokace paměti

První vlákno je přerušeno mezi řádky 9 a 10, kdy druhé vlákno prvek odebere z fronty a dealokuje jej.

Dealokace paměti

Globální synchronizace

Ukládáme ukládáme uvolněné bloky paměti do seznamu a v bezpečném okamžiku paměť uvolníme.

Reference counting

Budeme počítat reference, ale může dojít dealokaci mezi získáním ukazatele na prvek a zvýšením čítače.

```
1  Function pop()
2      while true do
3          h ← head
4          Increment h.ref_count
5          if h ≠ head then
6              Decrement h.ref_cnt and retry the loop
7          n ← h.next
8          if CAS(head, h, n) == h then
9              Decrement h.ref_cnt and return h
10         Decrement h.ref_cnt
```

Předpokládáme, že uvolněná paměť bude v budoucnu využita ke stejnému typu objektu, protože na řádce 4 zvyšujeme čítač nějakého prvku.

Hazardní ukazatele

Každé vlákno má jeden/seznam hazardních ukazatelů.

```
Function pop()
    while true do
        h ← head
        hp ← h
        if h ≠ head then
            Retry the loop
        n ← h.next
        if CAS(head, h, n) == h then
            hp ← NULL
            return h
```

Při uvolnění paměti si hazardní ukazatele uložíme do vyhledávací struktury a projdeme seznam bloků k uvolnění.

Problém producenta a spotřebitele

Fronta využívaná dvěma vlákny

- Chceme frontu implementovanou pomocí spojového seznamu
- Jedno vlákno producenta přidává prvku do fronty
- Jedno vlákno spotřebitele prvky vybírá
- Je následující postup správný?

Producent

node $\leftarrow$ new node
node.data $\leftarrow$ produced data
node.next $\leftarrow$ NULL
last.next $\leftarrow$ node
last $\leftarrow$ node

Spotřebitel

if *first.next* $\neq$ *NULL* **then**
    previous $\leftarrow$ first
    first $\leftarrow$ first.next
    delete previous
    consume first.data

Garance paralelních datových struktur

1. Blocking: Operace může čekat neomezeně dlouho
2. Obstruction-free: Operace určitě doběhne, když jsou ostatní vlákna zastavena
3. Lock-free: Některé vlákno musí doběhnou, i když pro jiné může nastat live-lock.
4. Wait-free: Všechna vlákna doběhnou v konečném čase, tj. nemůže nastat live-lock.
5. Bounded wait-free: Všechna vlákna doběhnou v garantovaném čase.