KAPITOLA 1

Cache-oblivius algoritmy a dátové štruktúry

Intro text - obsah kapitoly (alg/ds, analyza, ...)

1.1 Základné algoritmy

Na demonštráciu *cache-oblivious* algoritmov a ich analýzy v *external-memory* modeli použijeme jednoduchý algoritmus, ktorý počíta agregačnú funkciu nad hodnotami uloženými v poli.

1.1.1 Popis algoritmu

Majme pole A veľkosti $|A| = N$ a označme jeho prvky $A = \{a_1, \ldots, a_N\} \in X^N$. Chceme vypočítať hodnotu $f_g(A)$, kde $g : X \times Y \to Y$ je agregačná funkcia, $g_0 \in Y$ je počiatočná hodnota a $f_g : X^\infty \to Y$ je rozšírenie agregačnej funkcie definované následovne:

$$f_g(\{a_1, \ldots, a_k\}) = g(a_k, f(\{a_1, \ldots, a_{k-1}\}))$$
$$f_g(\emptyset) = g_0$$

Spravna notacia pre x infinity?

Túto funkciu je možné implementovať jednoducho ako jeden cyklus. Schematickú verziu implementácie uvádzame v algoritme 2.1.

Algoritmus 1.1 Implementácia agregačnej funkcie f_g

```
1: function AGGREGATE(g, g_0, A)
2:     y ← g_0
3:     for i ← 1, ..., |A| do
4:         y ← g(A[i], y)
5:     return y
```

Tento algoritmus s použitím vhodnej funkcie g a hodnoty g_0 je možné použiť na rôzne, často užitočné výpočty, ako napríklad maximum, minimum, suma a podobne:

$$g^{\max}(x,y) = \max(x,y) \qquad g_0^{\max} = -\infty$$
$$g^{\text{sum}}(x,y) = x + y \qquad g_0^{\text{sum}} = 0$$

1.1.2 Analýza zložitosti

Časová analýza

Klasická časová analýza tohto algoritmu je triviálna ak uvažujeme *RAM model*. Keďže prístup ku každému prvku `A[i]` zaberie konštantný čas a za predpokladu, že čas na výpočet funkcie g, T_g, je nezávislý na vstupe, bude výsledný čas na výpočet tejto funkcie

$$T(N) = \mathcal{O}(1) + N[\mathcal{O}(1) + T_g + \mathcal{O}(1)] = T_g \cdot \mathcal{O}(N)$$

Pamäťová analýza

V prípade *cache-aware* algoritmu by sme pole A mali uložené v $\lceil \frac{N}{B} \rceil$ blokoch veľkosti B. Pri výpočte by sme postupne tieto bloky načítali do cache a pracovali s nimi. V rámci jedného bloku počas výpočtu nedochádza k pamäťovým presunom. Zároveň stačí každý prvok spracovať raz a teda celkový počet pamäťových operácií bude presne rovný počtu blokov, $\lceil \frac{N}{B} \rceil$. Tento algoritmus však požaduje znalosť parametra B a explicitný presun blokov.

Jednoducho však vieme dosiahnuť (takmer) rovnakú zložitosť aj v prípade *cache-oblivious* algoritmu 2.1, ktorý žiadne parametre pamäte zjavne nevyužíva a nepozná. Budeme predpokladať, že pole A je uložené v súvislom úseku pamäte - to je možné dosiahnuť aj bez znalosti parametrov pamäte. Zvyšok algoritmu prebieha rovnako ako v predchádzajúcom prípade. Každý blok obsahujúci nejaký prvok poľa A bude teda presunutý do cache práve raz, a žiadne iné presuny nenastanú. Ostáva zistiť, koľko takých blokov môže byť.

Keďže nepoznáme veľkosti blokov v pamäti, nevieme pri ukladaní prvkov poľa zaručiť zarovnanie so začiatkom bloku. V najhoršom prípade uložíme do prvého bloku iba jeden prvok. Potom bude nasledovať $\lfloor \frac{N}{B} \rfloor$ plných blokov a nakoniec ešte najviac jeden blok, ktorý opäť nie je plný. Spolu máme teda $\lfloor \frac{N}{B} \rfloor + 2$ blokov.

Pokiaľ $\lfloor \frac{N}{B} \rfloor < \lceil \frac{N}{B} \rceil$ máme spolu najviac $\lceil \frac{N}{B} \rceil + 1$ blokov. V opačnom prípade B delí N, teda v prvom a poslednom bloku je spolu presne B prvkov a medzi nimi sa nachádza najviac $\frac{N-B}{B} = \frac{N}{B} - 1$ plných. Teda blokov je vždy najviac $\lceil \frac{N}{B} \rceil + 1$.

Zostrojili sme teda *cache-oblivious* algoritmus s asymptoticky rovnakou zložitosťou $\mathcal{O}(\frac{N}{B})$ ako optimálny *cache-aware* algoritmus, ktorého implementácia je však jednoduchšia, keďže nemusí explicitne spravovať presun blokov do cache.

1.2 Vyhľadávacie stromy

intro ...

lowerbound

1.2.1 *Cache-aware* riešenie

V prípade, že poznáme veľkosť blokov B v cache, môžeme problém vyhľadávacích stromov riešiť B-stromom s vetvením $\Theta(B)$. Každý vrchol teda vieme načítať s použitím $\mathcal{O}(1)$ pamäťových presunov. Výška takého B-stromu, ktorý má N listov, bude $\mathcal{O}(\log_B N)$. Celkovo teda vyhľadávanie v tomto strome vykoná $\mathcal{O}(\log_B N)$ pamäťových presunov. To zodpovedá dolnej hranici vo vete . dokaz

1.2.2 Naivné *cache-oblivious* riešenie

Predtým ako popíšeme efektívne *cache-oblivious* riešenie, pozrime sa na klasický binárny vyhľadávací strom. Jednoduchý a častý spôsob ako usporiadať uzly binárneho stromu v pamäti je nasledovný. Koreň uložíme na pozíciu 1. Ľavého a pravého potomka vrchola na pozícii x uložíme na pozície $2x$ a $2x+1$. Otec vrcholu x bude na pozícii $\lfloor \frac{x}{2} \rfloor$. Príklad takto uloženého stromu je na obrázku 2.2(a).

Výhodou tohto usporiadania sú implicitné vzťahy medzi vrcholmi. Na udržiavanie stromu stačí jednorozmerné pole kľúčov. Na prechod medzi nimi môžeme použiť triviálne funkcie uvedene v . algoritmu

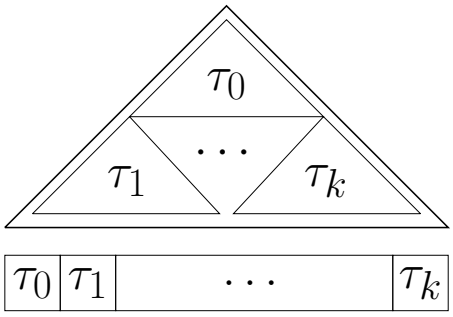
Nevýhodou je však vysoký počet pamäťových presunov pri vyhľadávaní. Výška tohto stromu je $\mathcal{O}(\log N)$. Pri načítaní vrcholu na pozícii x sa v rovnakom bloku nachádzajú vrcholy na pozíciach

$$x-k, \ldots, x-1, x, x+1, \ldots, x+l$$

kde $k + l < B$. Pri ďalšom kroku vyhľadávania budeme potrebovať vrchol $2x$ alebo $2x+1$ a teda nás pozície menšie ako x nezaujímajú. V najlepšom prípade teda bude $k = 0$ a $l = B-1$. Aby sa v tomto intervale nachádzali požadované vrcholy musí platiť

$$2x + 1 \leq x + l = x + B - 1$$
$$x \leq B - 2$$

To znamená, že pre pozície $x > B - 2$ už bude potrebný pamäťový presun pre každý vrchol. Vrchol s pozíciou $B-2$ bude mať hĺbku $\mathcal{O}(\log B)$ a teda počet vrcholov na ceste z koreňa do listu, ktorých pozície v pamäti sú väčšie ako $B-2$ bude $\Omega(\log N - \log B)$. Pre každý z nich je potrebné vykonať pamäťový presun a teda vyhľadávanie v takto usporiadanom binárnom strome vykoná $\Omega(\log \frac{N}{B})$ pamäťových presunov, čo je horšie ako pri *cache-aware* B-strome. porovnat fnc?



Obrázok 1.1: Schematické znázornenie rekurzívneho delenia pri van Emde Boas usporiadaní. Podstromy $\tau_0, \ldots, \tau_k$ sa uložia do súvislého pola.

1.2.3 Statický *cache-oblivious* vyhľadávací strom

Problémom predošlého riešenia je neefektívne usporiadanie v pamäti - pri prístupe ku vrcholu sa spolu s ním v rovnakom bloku nachádzajú vrcholy, ktoré nie sú pre ďalší priebeh algoritmu podstatné.

Riešením je takzvané *van Emde Boas usporiadanie* (*van Emde Boas layout*, nazvané podla van Emde Boas stromov s podobnou myšlienkou), ktoré funguje následovne. Uvažujme úplný binárny strom výšky h. Ak $h = 1$ tak máme iba jeden vrchol v a výstupom usporiadania bude (v).

Pre $h > 1$ rozdelíme vstupný strom na podstrom τ_0 výšky $\frac{h}{2}$, ktorého koreňom je koreň pôvodného stromu. Zostanú nám podstromy $\tau_1, \ldots, \tau_k$, ktorých korene sú potomkovia listov τ_0 a listy sú listy vstupného stromu. Rekurzívne ich uložíme do van Emde Boas usporiadania a následne uložíme za seba, výstupom teda bude $(\tau_0, \tau_1, \ldots, \tau_k)$. Schéma tohto delenia je na obrázku 2.1 a príklad takto usporiadaného stromu na obrázku 2.2(b).

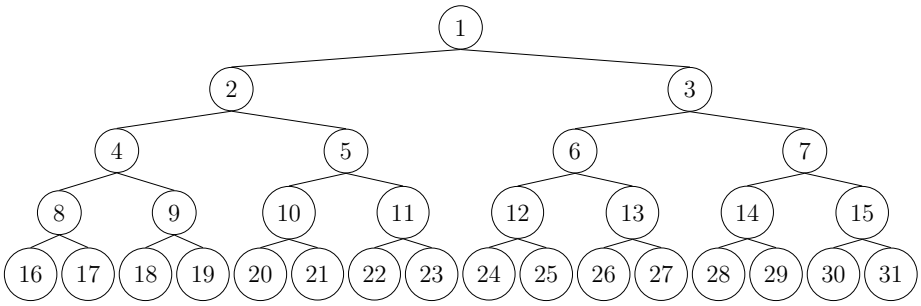
Tieto podstromy majú veľkosť $\Theta(\sqrt{N})$ kde N je veľkosť vstupného stromu, keďže ich výška je $\frac{h}{2} = \frac{1}{2} \lg N = \lg \sqrt{N}$.

Vyhľadávanie

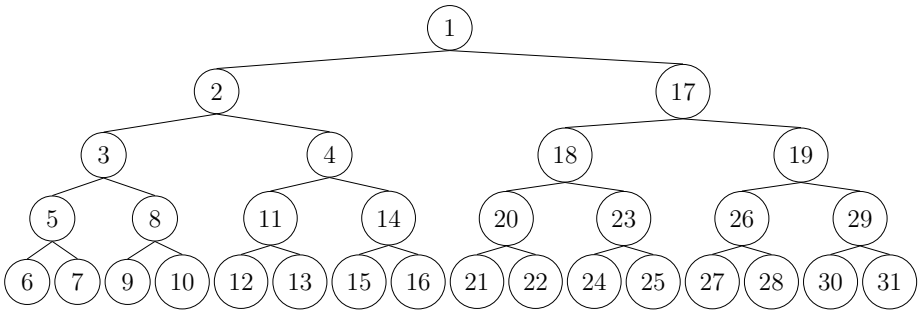
pointers vs implicit indexing

Pri analýze vyhľadávania sa pozrime na také podstromy predošlého delenia, že ich veľkosť je $\Theta(B)$. Ďalšie delenie a preusporiadanie je už zbytočné, no to *cache-oblivious* algoritmus nemá ako vedieť. Keďže ale po rekurzívnom volaní získame len iné usporiadanie, ktoré uložíme v súvislom úseku pamäte, bude stále možné tento podstrom načítať v $\mathcal{O}(1)$ blokoch.

Majme teda vyhľadávací strom zložený z takýchto podstromov, ktorých veľkosť je medzi $\Omega(\sqrt{B})$ a $\mathcal{O}(B)$. Ich výška je teda $\Omega(\log B)$. Pri strome výšky $\mathcal{O}(\log N)$ teda prejdeme cez $\mathcal{O}(\frac{\log N}{\log B})$ takých podstromov a každý vyžaduje konštantný počet pamäťových presunov a spolu sa ich teda vykoná $\mathcal{O}(\log_B N)$, čo zodpovedá spodnej



(a) Klasické usporiadanie



(b) van Emde Boas usporiadanie

Obrázok 1.2: Porovnanie klasického a van Emde Boas usporiadania na úplnom binárnom strome výšky 5. Čísla vo vrcholoch určujú poradie v pamäti.

hranici tohto problému.

Máme teda vyhľadávanie, ktoré je rovnako ako pri *cache-aware* B-stromoch optimálne. Problémom tejto dátovej štruktúry je však nemožnosť efektívne vkladať či odoberať prvky - pri každej zmene by bolo potrebné strom preusporiadať. Máme teda *cache-oblivious* ekvivalent statických *cache-aware* B-stromov.

Na úpravu tohto statického stromu tak, aby efektívne zvládal operácie pridávania a odoberania, budeme potrebovať pomocnú dátovú štruktúru, ktorú popíšeme v nasledovnej sekcii.

1.3 Usporiadané pole

obrazky

Problémom *údržby usporiadaného poľa* (z anglického *ordered-file maintenance*) budeme volať problém spočívajúci v udržiavaní zoradenej postupnosti prvkov, do ktorej možno pridávať nové prvky medzi ľubovolné dva existujúce a tiež prvky odstraňovať. Dátovou štruktúrou, ktorá tento problém rieši efektívne je *štruktúra zhustenej pamäte* (*packed-memory structure*). Táto štruktúra udržiava prvky v súvislom poli veľkosti $\mathcal{O}(N)$ s *medzerami* medzi prvkami veľkosti $\mathcal{O}(1)$. Vďaka tomu bude načítanie K po sebe idúcich prvkov vyžadovať $\mathcal{O}(\frac{K}{B})$ pamäťových presunov.

1.3.1 Popis štruktúry

Celá dátová štruktúra pozostáva z jedného poľa veľkosti $T = 2^k$. To (pomyselne) rozdelíme na *bloky* veľkosti $S = 2^l$ tak, že $S = \Theta(\log N)$. Počet blokov tak bude tiež mocnina dvoch.

Nad týmito blokmi zostrojíme (imaginárny) úplný binárny strom, ktorého listy sú bloky udržiavaného poľa. *Hĺbkou* vrchola označíme jeho vzdialenosť od koreňa, pričom koreň má hĺbku 0 a listy majú hĺbku $d = k - l$.

1.3.2 Definície

Kapacitou vrchola v, $c(v)$, označíme počet položiek (aj prázdnych, teda aj s medzerami) poľa patriacich do blokov v podstrome začínajúcom v tomto vrchole. Kapacita listov bude teda S, ich rodičov $2S$ a kapacita koreňa bude T. Podobne budeme počet neprázdnych položiek v podstrome vrcholu v volať *obsadnosť* a značiť $o(v)$.

Ďale *hustotou*, $0 \leq d(v) \leq 1$, označíme $d(v) = \frac{o(v)}{c(v)}$. Zvoľme ľubovolné konštanty

$$0 < \rho_d < \rho_0 < \tau_o < \tau_d < 1$$

a definujme pre vrchol s hĺbkou k *dolnú* a *hornú hranicu hustoty* ρ_k a τ_k tak, že dostaneme postupnosť hraníc pre všetky hĺbky, pričom platí $(\rho_i, \tau_i) \subset (\rho_{i+1}, \tau_{i+1})$ a teda sa tieto intervaly smerom od listov ku koreňu zmenšujú:

$$\rho_k = \rho_0 + \frac{k}{d}(\rho_d - \rho_0) \quad \tau_k = \tau_0 - \frac{k}{d}(\tau_0 - \tau_d)$$

$$0 < \rho_d < \rho_{d-1} < \cdots < \rho_0 < \tau_0 < \tau_1 < \cdots < \tau_d < 1$$

Napokon, vrchol v hĺbky k je v *hraniciach* hustoty ak platí $\rho_k \leq d(v) \leq \tau_k$.

1.3.3 Operácie

pseudocode?

Vkladanie

Implementácia operácie vkladania sa skladá z niekoľkých krokov. Najskôr zistíme, do ktorého bloku v spadá pozícia, na ktorú vkladáme. Pozrieme sa, či je tento blok v hraniciach hustoty. Ak áno tak platí $d(v) < 1$ a teda $o(v) < c(v)$, čiže v tomto bloku je voľné miesto. Môžeme teda zapísať novú hodnotu do tohto bloku, pričom môže byť potrebné hodnoty v bloku popresúvať, avšak zmení sa najviac S pozícii.

V opačnom prípade je tento blok mimo hraníc hustoty. Budeme postupovať hore v strome dovtedy, kým nenájdeme vrchol v hraniciach. Keďže strom je iba pomyselný, budeme túto operáciu realizovať pomocou dvoch súčasných prechodov k okrajom poľa.

Počas tohto prechodu si udržiavame počet neprázdnych a všetkých pozícii a zastavíme v momente, keď hustota dosiahne požadované hranice.

Po nájdení takéhoto vrchola v hraniciach rovnomerne rozdelíme všetky hodnoty v blokoch prislúchajúcich danému podstromu. Keďže intervaly pre hranice sa smerom ku listom iba rozširujú budú po tomto popresúvaní všetky vrcholy tohto podstromu v hraniciach hustoty a teda aj požadovaný blok bude obsahovať aspoň jednu prázdnu pozíciu. Môžeme teda novú hodnotu vložiť ako v prvom kroku.

Ak nenájdeme taký vrchol, ktorého hustota by bola v hraniciach, a teda aj koreň je mimo hraníc, je táto štruktúra príliš plná. V takom prípade zostrojíme nové pole dvojnásobnej veľkosti a všetky existujúce položky, s pridanou novou, rovnomerne rozmiestnime do nového poľa.

Odstraňovanie

Operácia odstraňovania prebieha analogicky. Ako prvé požadovanú položku odstránime z prislúchajúceho bloku. Ak je tento blok aj naďalej v hraniciach hustoty tak skončíme, inak postupujeme nahor v strome, kým nenájdeme vrchol v hraniciach. Následne rovnomerne prerozdelíme položky blokoch daného podstromu.

Pokiaľ taký vrchol nenájdeme, je pole príliš prázdne a zostrojíme nové polovičnej veľkosti a rovnomerne do neho rozmiestnime zostávajúce položky pôvodného.

1.3.4 Analýza

Pri vložení aj odstraňovaní sa upraví súvislý interval I, ktorý sa skladá z niekoľkých blokov. Nech pri nejakej operácii došlo k prerozdeleniu prvkov v blokoch prislúchajúcich podstromu vrchola u. Teda pred týmto prerozdelením bol vrchol u v hĺbke k v hraniciach hustoty ($\rho_k \leq d(u) \leq \tau_k$) ale nejaký jeho potomok v nebol.

Po prerozdelení budú všetky vrcholy v danom podstrome v hraniciach hustoty, avšak nie len v svojich ale aj v hraniciach pre hĺbku k, ktoré sú tesnejšie. Bude teda platiť $\rho_k \leq d(v) \leq \tau_k$. Najmenší počet operácii vloženia, q, potrebný na to, aby bol vrchol v opäť mimo hraníc je

$$\frac{o(v)}{c(v)} = d(v) \leq \tau_k \qquad \frac{o(v)+q}{c(v)} > \tau_{k+1}$$

$$o(v) \leq \tau_k c(v) \qquad o(v)+q > \tau_{k+1}c(v)$$

$$q > (\tau_{k+1} - \tau_k)c(v)$$

Podobne pre prekročenie dolnej hranice je potrebných aspoň $(\rho_k - \rho_{k+1})c(v)$ operácii odstránenia.

Pri úprave intervalu blokov v podstrome vrcholu u je potrebné upraviť najviac $c(u)$ položiek, avšak táto situácia nastane pokiaľ sa potomok v ocitne mimo hraníc hustoty.

Priemerná veľkosť intervalu, ktorý treba preusporiadať pri vložení do podintervalu prislúchajúceho vrcholu v teda bude

$$\frac{c(u)}{(\tau_{k+1} - \tau_k)c(v)} = \frac{2c(v)}{(\tau_{k+1} - \tau_k)c(v)} = \frac{2}{\tau_{k+1} - \tau_k} = \frac{2d}{\tau_d - \tau_0} = \mathcal{O}(\log T)$$

keďže τ_d a τ_0 sú konštanty a výška stromu d s T listami je $d = \Theta(\log T)$. Podobným spôsobom dostaneme rovnaký odhad pre odstraňovanie.

Pri vkladaní a odstraňovaní prvku ovplyvníme najviac d podintervalov - tie, ktoré prislúchajú vrcholom na ceste z daného listu (bloku) do koreňa. Spolu teda bude veľkosť upraveného intervalu v priemernom prípade $\mathcal{O}(\log^2 T)$.

worstcase bounds? conjenctured lowerbound?

Táto dátová štruktúra teda udržiava N usporiadaných prvkov v poli veľkosti $\mathcal{O}(N)$ a podporuje operácie vkladania a odstraňovania, ktoré upravujú súvislý interval priemernej veľkosti $\mathcal{O}(\log^2 N)$ a je ich teda možné realizovať pomocou $\mathcal{O}(\frac{\log^2 N}{B})$ pamäťových presunov.