Úvod

Pri tvorbe efektívnych algoritmov sa najčastejšie zaujímame o časovú zložitosť a prípadne aj pamäťovú zložitosť. Dôležitým faktorom však môže byť aj množstvo pamäťových operácií. Tie majú priamy vplyv na výslednú časovú zložitosť avšak pri asymptotickej analýze sa často považujú za operáciu vykonateľnú v konštantnom čase. V prípadoch keď pracujeme s veľkým objemom dát, ktoré sú uložené na médiu s veľkou kapacitou ale nízkou prístupovou rýchlosťou, môže byť výhodné minimalizovať množstvo zápisov a čítaní z tohto média.

Klasickým prístupom v týchto prípadoch boli takzvané *cache-aware* algoritmy, ktoré potrebujú poznať presné parametre pamäťového systému pre dosiahnutie požadovanej efektivity. V tejto práci sa budeme venovať *cache-oblivious* algoritmom, ktoré sú asymptoticky rovnako efektívne ako *cache-aware*, avšak bez nutnosti poznať parametre pamäte.

Okrem samozrejmej výhody, kedy nám stačí jedna univerzálna implementácia algoritmu pre ľubovolné množstvo rôznych systémov, prinášajú tieto algoritmy aj iné zlepšenia. V takmer každom systéme je pamäť hierarchická, zložená z viacerých úrovní, pričom každá je väčšia ale pomalšia ako predošlá. Pri *cache-aware* by pre optimálne využitie každej z úrovní pamäte bolo potrebné poznať parametre každej úrovne, a rekurzívne vnoriť do seba mnoho inštancií, každú optimalizovanú pre jednu úroveň. No v *cache-oblivious* nám stačí jedna inštancia - keďže nepozná parametre žiadnej úrovne, bude rovnako dobre fungovať na každej z nich.

Súčasťou práce je vysvetliť problematiku *cache-oblivious* pamäťového modelu, uviesť prehlad algoritmov a dátových štruktúr a vytvoriť vizualizácie vybraných z nich. Vizualizácie sú implementované ako súčasť programu *Alg-Vis*. Cieľom týchto vizualizácií je umožniť užívateľom experimentovať s implementáciou týchto algoritmov a dátových štruktúr, sledovať ich prácu krok po kroku a uľahčiť ich porozumeniu.a

KAPITOLA 1

Pamäťový model

Pri časovej analýze algoritmov sa zvyčajne používa takzvaný *RAM model* (skratka z anglického *Random-Access Machine*, stroj s náhodným prístupom k pamäti), v ktorom sa predpokladá možnosť pristupovať k ľubovolnému úseku pamäte v konštantnom čase. To znamená, že vo výslednej asymptotickej analýze počítame len počet vykonaných operácií.

V skutočnosti však moderné počítače využívajú niekoľko úrovňovú pamäťovú hierarchiu [2]. Tá sa typicky skladá z registrov a troch úrovní *cache* (vyrovnávacej pamäte) priamo na procesore, následne z hlavnej operačnej pamäte a disku. V tomto poradí referencie sú tieto úrovne zoradené od najrýchlejšej a najmenšej (odozva rádovo 1 ns, kapacita 64 KiB) až po najpomalšiu ale najväčšiu (odozva od 100 µs po 10 ms podľa typu[1], kapacita rádovo 1 TiB). Približné hodnoty pre všetky úrovne sú v tabuľke 1.1.

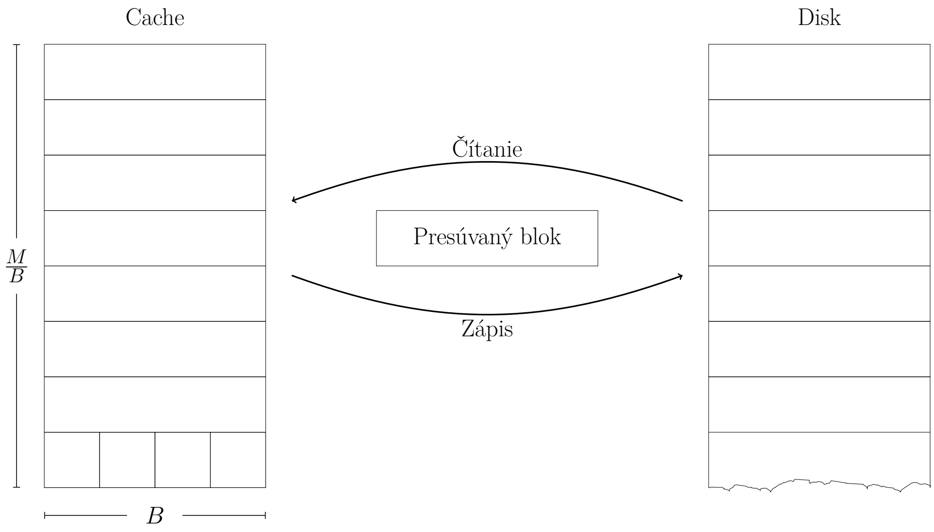
Dôsledkom tejto hierarchie je závislosť výslednej rýchlosti algoritmu od jeho prístupov k pamäti. Operácie, ktoré využívajú dáta uložene na disku potrvajú dlhšie ako tie, ktoré využívajú iba dáta v registroch. V skutočnosti sa tieto dáta postupne presunú z disku do hlavnej pamäte, do *cache* na procesore a napokon do registrov. Následne sa

Tabuľka 1.1: Približné parametre rôznych úrovní pamäte. Hodnoty troch úrovní *cache* na procesore (L1, L2 a L3) sú pre mikroarchitektúru Intel Haswell. [4]

Úroveň	**Veľkosť**	**Odozva**
L1	64 KiB	4clk[1] ≈ 1 ns
L2	256 KiB	11clk[1] ≈ 4 ns
L3	2–20 MiB	36clk[1] ≈ 12 ns
RAM	≈ 8 GiB	≈ 100 ns
Disk	≈ 1 TiB	≈ 0.1–10 ms

[1] Počet cyklov procesora, uvedené časové aproximácie pri 3 GHz.

[1]Klasické pevné disky (HDD) alebo disky bez pohyblivých častí (SSD)



Obrázok 1.1: External-memory model

môže požadovaná operácia vykonať rovnako rýchlo ako v druhom prípade, avšak nejaký čas bol algoritmus nečinný a čakalo sa na presun dát. Pre všetky susedné dvojice pamäťových úrovní teda slúži tá menšia a rýchlejšia ako vyrovnávacia pamäť pre tú väčšiu a pomalšiu.

1.1 External-memory model

Spôsobom ako zohľadniť tieto skutočnosti pri analýze algoritmov je takzvaný *external-memory model* (model externej pamäte), nazývaný tiež *I/O model* alebo *cache-aware model* [1]. Ten popisuje pamäť skladajúcu sa z dvoch častí (obrázok 1.1), ktoré voláme *cache* a *disk*.

Všetky výpočty prebiehajú nad dátami v *cache*, ktorá má obmedzenú veľkosť. Ostatné dáta sú uložené na disku neobmedzenej veľkosti, no nemôžeme s nimi priamo manipulovať a je ich potrebné najskôr preniesť do *cache*. V samotnej analýze algoritmov potom počítame počet týchto prenosov z disku do *cache* a naopak.

Samotné presuny sú realizované v *blokoch* pamäte veľkosti B. Disk aj *cache* sa skladajú z takýchto blokov a za jednu operáciu považujeme presun jedného bloku medzi nimi. *Cache* má obmedzenú veľkosť M a skladá sa teda z $\frac{M}{B}$ blokov.

1.1.1 Cache-aware algoritmy

Pokiaľ poznáme parametre B a M, môžeme skonštruovať algoritmus, ktorý bude túto dvojicu pamätí využívať efektívne. Takéto algoritmy voláme *cache-aware* (uvedomujúci si *cache*). Súčasťou tohto algoritmu by bolo explicitné spravovanie presunov pamäte - je potrebné riešiť čítanie blokov z disku a ich umiestňovanie do *cache*, nahrádzanie blokov v *cache* pri zaplnení a spätný zápis blokov na disk.

Tento model tiež popisuje len dve úrovne pamäte a teda funguje efektívne len pre danú susednú dvojicu, pre ktorú ho na základe znalosti parametrov optimalizujeme. V moderných systémoch ale máme takýchto dvojíc niekoľko. Keby sme poznali parametre pre všetky tieto dvojice môžeme zovšeobecniť tento model pre viac úrovní a explicitne riešiť presun blokov medzi nimi. Takto sa však samotná správa pamäte potenciálne stáva komplikovanejším problémom ako pôvodný algoritmus.

1.2 Cache-oblivious model

Riešením týchto problémov je *cache-oblivious model* (na *cache* nedbajúci), v ktorom uvažujeme rovnakú dvoj-úrovňovú pamäť zloženú z disku a *cache* [3, 5]. Na rozdiel od *cache-aware* modelu však algoritmus nepozná parametre B a M. Pokiaľ sa nám napriek tomu podarí navrhnúť algoritmus, ktorý vykonáva (asymptoticky) rovnaký počet pamäťových presunov ako *cache-aware* algoritmus, bude bežať efektívne pre ľubovolné takéto parametre.

Takéto algoritmy majú na rozdiel od *cache-aware* algoritmov v *external-memory* modely mnohé výhody. Samotná implementácia algoritmu nemôže explicitne riešiť presun blokov pokiaľ nepozná veľkosť bloku ani koľko blokov môže do *cache* uložiť. Táto úloha zostane ponechaná na nižšiu vrstvu (operačný systém resp. hardware v prípade *cache* na procesore) - algoritmus bude pristupovať k pamäti priamo bez ohľadu na to, či sa nachádza v *cache* alebo nie a v prípade potreby prebehnú nutné prenosy na nižšej úrovni (z pohľadu algoritmu) automaticky.

Ďalšou výhodou je automatická optimalizácia pre dané parametre. V prípade *cache-aware* algoritmov môže byť problémom získať presné hodnoty týchto parametrov a potrebné pri ich zmene upraviť algoritmus. Vývoj algoritmu, ktorý bude fungovať na rozličných architektúrach, môže byť problematický.

V neposlednom rade bude takýto *cache-oblivious* algoritmus efektívny medzi každou dvojicou susedných úrovní. Vzhľadom na to, že hodnoty parametrov nepozná, musí pre ľubovolnú takú dvojicu pracovať rovnako efektívne.

1.2.1 Správa pamäte

V momente keď sa *cache-oblivious* algoritmus pokúsi o vykonanie operácie, ktorá potrebuje dáta mimo cache je potrebné ich najskôr z disku skopírovať. V prípade, že je v cache voľný blok tak je možné presunúť dáta bez nutnosti nahradenia. V opačnom prípade je však potrebné uvoľniť miesto tým, že sa vyberie blok z cache, v prípade, že bol upravený sa jeho obsah zapíše späť na disk a následne sa požadovaný blok z disku zapíše na jeho miesto v cache. Tento proces sa nazýva výmena stránok (*page replacement*), a algoritmus rozhodujúci, ktorý blok z cache odstrániť, voláme stratégia výmeny stránok (*page-replacement strategy*).

Ak by sa táto stratégia správala tak, že vždy odstráni blok, ktorý bude potrebný v ďalšom kroku algoritmu, tak by bolo zakaždým presúvať bloky medzi diskom a cache. To by znamenalo, že počet blokov, s ktorými v cache môžeme pracovať je $M/B = 1$. Ďalším problémom je *asociatívnosť* cache - z praktických dôvodov je často možné daný blok z disku uložiť len na niekoľko pozícií v cache. Inak by bolo potrebné ukladať spolu s každým blokom jeho plnú adresu na disku, čo by redukovalo celkový počet blokov, ktoré sa do cache zmestia. Znížením asociativity je možné ukladať iba časť adresy, pričom zvyšok je implicitne určený pozíciou v cache. V prípade nízkej asociativity však môžu opäť nastať situácie, kedy je algoritmus schopný využiť iba malý počet blokov v cache.

Tieto problémy *cache-oblivious* model obchádza predpokladom ideálnej cache - cache, ktorá je plne asociatívna (každý blok disku je možné uložiť v každom bloku cache) a používa optimálnu stratégiu výmeny stránok, ktorá vždy odstráni blok, ktorý bude potrebný najneskôr. Prvý predpoklad je síce v reálnych systémoch nepraktický, no z teoretického hľadiska je v poriadku. Druhý predpoklad je však nerealizovateľný, keďže by stratégia výmeny stránok musela predpovedať budúce kroky algoritmu. Nasledovné lemy však ukazujú, že bez týchto predpokladov na reálnom systéme s nízkou asociativitou a jednoduchou stratégiou výmeny stránok sa algoritmus zhorší len o konštantný faktor.

Lema 1.2.1. *Algoritmus, ktorý v ideálnej cache veľkosti M s blokmi veľkosti B vykoná T pamäťových operácií, vykoná najviac $2T$ pamäťových operácií v cache veľkosti $2M$ s blokmi veľkosti B pri použití stratégie* LRU *alebo* FIFO*. [3, Lemma 12]*

Lema 1.2.2. *Plne asociatívna cache veľkosti M sa dá simulovať s použitím $\mathcal{O}(M)$ pamäte tak, že prístup ku každému bloku v cache zaberie v priemernom prípade $\mathcal{O}(1)$ času. [3, Lemma 16]*

Stratégia *LRU* (least recently used) vyberá vždy blok, ktorý bol najdlhšie nepoužitý. Implementácia vyžaduje udržiavať si ku každému bloku počítadlo, ktoré sa pri prístupe nastaví na nulu a pri prístupe k iným blokom zvýši o jedna. Pri potrebe uvoľniť miesto v cache vyberieme blok s najväčšou hodnotou počítadla - ten, ku ktorému najdlhšie nebol prístup.

Stratégia *FIFO* (first in, first out) je ešte jednoduchšia - bloky sú udržiavame zoradené podla poradia ich vloženia do cache. Keď vyberáme blok na odstránenie tak vezmeme ten, ktorý bol pridaný najskôr a teda je na začiatku tohto usporiadania.

KAPITOLA 2

Cache-oblivius algoritmy a dátové štruktúry

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

2.1 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.

2.1.1 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:

Spravna notacia pre x infinity?

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

$$f_g(\emptyset) = g_0$$

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

Algoritmus 2.1 Implementácia agregačnej funkcie f_g

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

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

2.1.2 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.

2.2 Vyhľadávacie stromy

intro ...

lowerbound

2.2.1 *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

2.2.2 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 algoritme 2.2.2.

Algoritmus 2.2 Algoritmy pre získanie pozícií ľavého syna, pravého syna a rodiča vrcholu na pozícii x

```
function LEFT(x)
    return 2x
```

```
function RIGHT(x)
    return 2x + 1
```

```
function PARENT(x)
    return ⌊x/2⌋
```

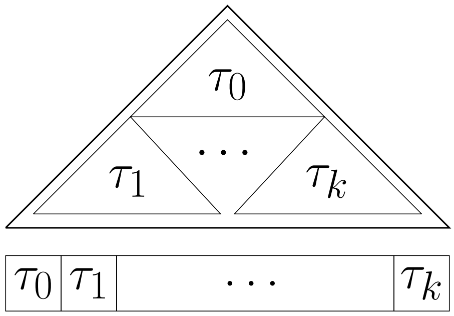
Nevýhodou je však vysoký počet pamäťových presunov pri vyhľadávaní. Výška tohto stromu je $\mathcal{O}(\log N)$. Pri načítaní vrcholu na pozícii x sa v rovnakom bloku nachádzajú vrcholy na pozíciach

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

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

$$\begin{aligned} 2x+1 &\leq x+l = x+B-1 \\ x &\leq B-2 \end{aligned}$$

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



Obrázok 2.1: Schematické znázornenie rekurzívneho delenia pri *van Emde Boas* usporiadaní. Podstromy $\tau_0, \ldots, \tau_k$ sa uložia do súvislého pola.

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?

2.2.3 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).

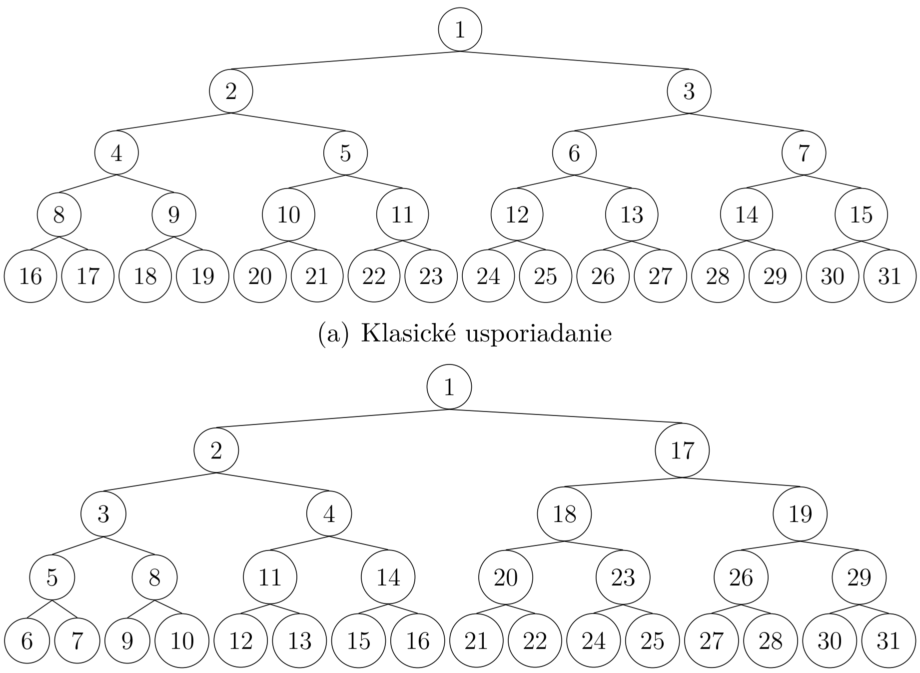
Pre $h > 1$ rozdelíme vstupný strom na podstrom τ_0 výšky $\frac{h}{2}$, ktorého koreňom je koreň pôvodného stromu. Zostanú nám podstromy $\tau_1, \ldots, \tau_k$, ktorých korene sú potomkovia listov τ_0 a listy sú listy vstupného stromu. Rekurzívne ich uložíme do *van Emde Boas* usporiadania a následne 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



(a) Klasické usporiadanie

(b) *van Emde Boas* usporiadanie

Obrázok 2.2: Porovnanie klasického a *van Emde Boas* usporiadania na úplnom binárnom strome výšky 5. Čísla vo vrcholoch určujú poradie v pamäti.

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 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.

2.3 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.

2.3.1 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 $d = k - l$.

2.3.2 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 $\rho_k \leq d(v) \leq \tau_k$.

2.3.3 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.

2.3.4 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

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

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

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

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?

amortizacia double/half rebuildu

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.

2.4 Dynamický B-strom

nazov? nie moc bstrom ked b=2... lepsie dictionary/tree?

intro, historia, ...

2.4.1 *cache-aware* riešenie

V prípade *cache-aware* modelu použijeme opäť B-strom s vetvením $Theta(B)$ ako v časti 2.2.1. Rovnako ako pre vyhľadávanie bude na vkladanie potrebných $\mathcal{O}(\log_B N)$ pamäťových presunov. odvodeni

2.4.2 Popis *cache-oblivious* štruktúry

Táto dátová štruktúra vznikne zložením predchádzajúcich dvoch - statického stromu (2.2.3) a usporiadaného poľa (2.3.1) - jednoduchým spôsobom. Majme usporiadané pole veľkosti $\Theta(N)$. Keďže počet položiek v usporiadanom poli je mocnina dvoch, môžeme nad ním vybudovať statický vyhľadávací strom uložený vo *van Emde Boas* usporiadaní. Listy tohto stromu budú bijektívne spárované s položkami v usporiadanom poli.

Kľúče, ktoré táto štruktúra obsahuje, sú uložené v usporiadanom poli (s medzerami). Listy statického stromu obsahujú v svojich kľúčoch rovnakú hodnotu ako k ním prislúchajúce položky usporiadaného pola. Medzeru reprezentujeme hodnotou, ktorá je pri použitom usporiadaní najmenšia. Ostatné vrcholy stromu obsahujú ako kľúč maximum z kľúčov svojich synov.

2.4.3 Vyhľadávanie

Vyhľadávanie v tejto štruktúre prebieha jednoducho. Začínajúc od koreňa, porovnáme hľadaný kľúč s kľúčom v ľavom synovi. Pokiaľ je hľadaný väčší, bude v pravom podstrome (keďže maximum z celého ľavého podstromu je práve kľúč ľavého syna), ktorý rekurzívne prehľadáme. V opačnom prípade prehľadáme ľavý podstrom. Keď dosiahneme list, porovnáme jeho kľúč s hľadaným. Ak sa zhodujú tak sme požadovanú položku našli, inak sa v štruktúre nenachádza.

dokaz ze tam nie je

Analýza

Toto prehľadávanie prechádza cestu od koreňa k listu v strome hĺbky $\mathcal{O}(\log N)$ uloženom vo *van Emde Boas* usporiadaní a teda rovnako ako v časti 2.2.3 vykoná $\mathcal{O}(\log_B N)$ pamäťových presunov.

2.4.4 Vkladanie

Zaujímavejšou operáciou je vkladanie, ktoré v pôvodnom statickom strome nebolo možné. Prvým krokom je nájdenie pozície, na ktorú tento kľúč patrí. To dosiahneme podobne ako v predošlej sekcii. Budeme ale hľadať predchádzajúci a nasledujúci kľúč. Tým nájdeme v usporiadanom poli dvojicu pozícii, medzi ktoré chceme vložiť novú hodnotu.

Vloženie do usporiadaného pola zmení súvislý interval veľkosti K. Následne bude potrebné aktualizovať kľúče v statickom strome, aby opäť obsahovali maximum zo svojich synov. Túto aktualizáciu dosiahneme takzvaným *post-order* prechodom, kedy sa vrchol aktualizuje až po tom, čo boli aktualizovaný jeho synovia. Tým máme zaručené, že po aktualizácii vrchol obsahuje výslednú, korektnú hodnotu. Implementácia takéhoto prechodu je naznačená v algoritme 2.3.

2.4.5 Analýza vkladania

Túto analýzu rozdelíme na tri časti, v ktorých sa postupne pozrieme na rôzne úrovne v statickom strome, ktoré treba po vložení do usporiadaného pola aktualizovať.

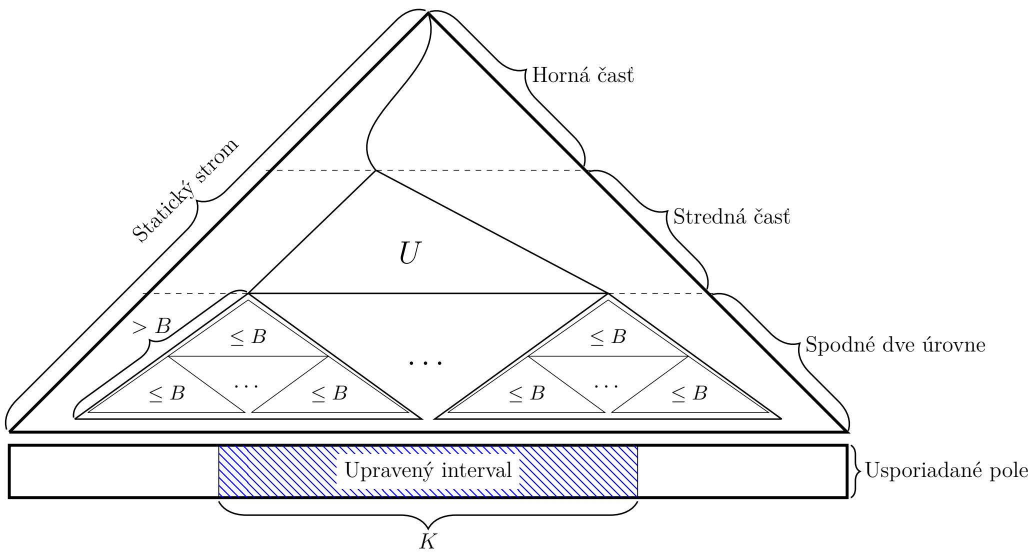
Algoritmus 2.3 Implementácia *post-order* prechodu na aktualizáciu statického stromu

function UPDATE(v, I) ▷ v je vrchol stromu, ktorý práve aktualizujeme
▷ I je zmenený interval v usporiadanom poli

  if v.podstrom $\cap I = \emptyset$ **then** ▷ pokiaľ sa zmena tohto vrcholu
    return ▷ určite nedotkla, tak ho môžeme preskočiť

  if v je list **then**
    v.kľuč $\leftarrow$ hodnota z usporiadaného poľa
  else
    UPDATE(v.ľavý) ▷ najskôr aktualizujeme ľavého
    UPDATE(v.pravý) ▷ a pravého syna

    v.kľúč $\leftarrow \max(v.\text{ľavý.kľúč}, v.\text{pravý.kľúč})$ ▷ až potom tento vrchol



Obrázok 2.3:
caption

Spodné dve úrovne

Uvažujme najskôr ako v časti 2.2.3 také podstromy *van Emde Boas* delenia, ktoré sa ako prvé zmestia do bloku *cache* a teda ich veľkosť je medzi $\sqrt{B}$ a B. Pozrime sa na spodné dve úrovne takýchto podstromov. Listy spodnej teda budú listy celého statického strom a korene spodnej spodnej úrovne budú synovia listov hornej úrovne. Zvyšok stromu pokračuje nad týmito úrovňami a vrátime sa k nemu neskôr.

Podstromy v tomto delení sa zmestia do najviac dvoch blokov a vieme ich teda načítať pomocou $\mathcal{O}(1)$ pamäťových presunov. Označme podstromy, ktoré vznikly rekurzívnym *van Emde Boas* delením z podstromu T veľkosti viac než B, rovnako, ako v časti 2.2.3: $\tau_0, \tau_1, \ldots, \tau_k$. Tu τ_0 je podstrom v hornej úrovni a $\tau_1, \ldots, \tau_k$ sú podstromy v spodnej úrovni, ktorých korene sú potomkovia listov τ_0.

Pri *post-order* prechode cez podstrom T budeme prejdeme najskôr cez ľavých synov od koreňa τ_0 cez τ_1 až do listu. Následne bude *post-order* prechod prebiehať v podstrome τ_1, až kým nebudú všetky jeho vrcholy a napokon aj koreň aktualizované. Potom sa vrátime do τ_0 a k τ_1 už pristupovať nebudeme ale prejdeme nasledovný podstrom, τ_2. Takto postupne aktualizujeme všetky podstromy, pričom postupnosť navštívených podstromov bude

$$\tau_0, \tau_1, \tau_0, \tau_2, \ldots, \tau_0, \tau_i, \tau_0, \ldots, \tau_k, \tau_0$$

Vidíme, že pokiaľ dokážeme udržať v *cache* aspoň dva také podstromy, teda za predpokladu $M \geq 4B$, sme schopný takýto *post-order* prechod zrealizovať pomocou $\mathcal{O}(1)$ pamäťových operácii pre každý podstrom veľkosti $\leq B$. Takéto podstromy majú $\mathcal{O}(B)$ listov, pričom nimi potrebujeme pokryť interval veľkosti K (podstromy, ktorých žiaden list neleží v upravenom intervale nie je potrebné aktualizovať) a teda celkový počet podstromov na spodných dvoch úrovniach, ktoré potrebujeme načítať a upraviť je $\mathcal{O}(\frac{K}{B})$ a stačí nám na to $\mathcal{O}(\frac{K}{B})$ pamäťových presunov.

Stredná úroveň - najbližší spoločný predok

Označme ako $T_1, \ldots, T_l$ tie stromy veľkosti $> B$ obsahujúce postromy veľkosti $\leq B$ na spodných dvoch úrovniach, ktoré boli v predošlej časti aktualizované. Platí teda $l = \mathcal{O}(\frac{K}{B})$. Označme U taký podstrom statického stromu, ktorého koreň je najbližší spoločný predok koreňov $T_1, \ldots, T_l$. Ak je tento koreň v hĺbke h tak U obsahuje všetky vrcholy hĺbky $\geq h$, ktoré treba aktualizovať - tie mimo U nie sú predkovia upravených vrcholov a teda hodnoty kľúčov ich synov neboli v prvej časti zmenené.

Počet vrcholov U je najviac dvojnásobok počtu listov a teda $|U| = \mathcal{O}(l) = \mathcal{O}(\frac{K}{B})$. Keďže pri *post-order* prechode navštívime každý vrchol $\mathcal{O}(1)$ krát (najprv pri prechode z koreňa ku listom, pričom následne rekurzívne prejdeme ľavého a pravého syna, a potom pri návrate z rekurzie) a teda celkovo budeme potrebovať $\mathcal{O}(\frac{K}{B})$ pamäťových presunov na aktualizáciu U.

Horná úroveň - cesta z najbližšieho spoločného predka do koreňa

Posledná množina vrcholov, ktoré je potrebné aktualizovať je cesta z koreňa U do koreňa statického stromu. Dĺžka tejto cesty je obmedzená výškou stromu $\mathcal{O}(\log N)$ a pri aktualizovaní prechádzame cez jej vrcholy postupne. Podobne ako pri vyhľadávaní (časť 2.4.3) bude potrebných $\mathcal{O}(\log_B N)$ pamäťových presunov.

Výsledná zložitosť

Sčítaním nasledovných zložitostí počtu pamäťových operácii

Nájdenie pozície v usporiadanom poli:	$\mathcal{O}(\log_B N)$
Vloženie do usporiadaného poľa:	$\mathcal{O}(\dfrac{K}{B})$
Aktualizácia spodných dvoch úrovní:	$\mathcal{O}(\dfrac{K}{B})$
Aktualizácia strednej úrovne:	$\mathcal{O}(\dfrac{K}{B})$
Aktualizácia hornej úrovne:	$\mathcal{O}(\log_B N)$

dostávame celkovú zložitosť $\mathcal{O}(\log_B N + \frac{K}{B})$. Keďže amortizovaná veľkosť upraveného intervalu je $K = \mathcal{O}(\log^2 N)$ dostávame výslednú amortizovanú zložitosť počtu pamäťových operácii pri vkladaní: $\mathcal{O}(\log_B N + \frac{\log^2 N}{B})$.

2.4.6 Odstraňovanie

Algoritmus odstraňovania je analogický, po nájdení prvku v usporiadanom poli ho odstránime, čím sa zmení súvislý amortizovanej veľkosti $\mathcal{O}(\log^2 N)$. Následne aktualizujeme kľúče v statickom poli rovnako ako pri vkladaní. Výsledná zložitosť bude taktiež rovnaká.

2.5 Vylepšený dynamický B-strom

Oproti *cache-aware* B-stromom ma pri vkladaní *cache-oblivious* dynamický strom popísaný v predchádzajúcej sekcii naviac $\mathcal{O}(\frac{\log^2 N}{B})$ pamäťových presunov. Odstrániť ho môžeme jednoduchou modifikáciou.

Vezmeme N kľúčov a rozdelíme ich do $\Theta(\frac{N}{\log N})$ blokov veľkosti $\Theta(\log N)$. Minimum z každej skupiny použijeme ako kľúče v predchádzajúcej dátovej štruktúre, ktorej veľkosť bude $\Theta(\frac{N}{\log N})$.

2.5.1 Vyhľadávanie

Najskôr vyhľadáme požadovaný blok v B-strome, čo zaberie $\mathcal{O}(\log_B \frac{N}{\log N}) = \mathcal{O}(\log_B N - \log_B \log N) = \mathcal{O}(\log_B N)$ pamäťových presunov. Následne prejdeme daný blok celý a nájdeme v ňom požadovaný kľúč. To vyžaduje $\mathcal{O}(\frac{\log N}{B})$ pamäťových presunov - zanedbateľné voči hľadaniu v B-strome - a spolu teda vyhľadávanie vyžaduje rovnako veľa pamäťových presunov ako neupravený B-strom: $\mathcal{O}(\log_B N)$.

popis najdenia bloku - min/max left

2.5.2 Vkladanie a odstraňovanie

Po nájdení požadovaného bloku vykonáme vloženie prípadne odstránenie z neho kompletným prepísaním, čo vyžaduje $\mathcal{O}(\frac{\log N}{B})$ presunov. Zároveň budeme udržiavať veľkosti blokov medzi $\frac{1}{4}\log N$ a $\log N$. Príliš malé skupiny môžeme spojiť do väčších a veľké rozdeliť na niekoľko menších. Skupina prekročí svoje hranice najskôr po $\Omega(\log N)$ operáciách. V takom prípade po rozdelení alebo spojení bude potrebné vykonať vloženie alebo odstránenie z B-stromu. Vydelením dostaneme amortizovanú zložitosť vkladania a odstraňovania v takto upravenej štruktúre: $\mathcal{O}(\log_B N)$.

Opäť sme teda dosiahli rovnaký počet operácií ako ekvivalentná *cache-aware* dátová štruktúra, avšak tentokrát v amortizovane.

jedine amortizovane je OF, to sa da worstcase, potom bude aj toto worstcase

Literatúra

[1] Aggarwal, Alok ; Vitter, Jeffrey u. a.: The input/output complexity of sorting and related problems. In: *Communications of the ACM* 31 (1988), Nr. 9, S. 1116–1127

[2] Drepper, Ulrich: What every programmer should know about memory. In: *Red Hat, Inc* 11 (2007)

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

[4] Intel Corporation: *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2014 (248966-029)

[5] Prokop, Harald: *Cache-oblivious algorithms*, Massachusetts Institute of Technology, Diss., 1999