

Cache-oblivious algoritmy a dátové štruktúry

V tejto kapitole popíšeme niekoľko *cache-oblivious* algoritmov a dátových štruktúr, spolu s ich pamäťovou analýzou v *cache-oblivious* modeli. Zároveň uvedieme ekvivalentnú *cache-aware* dátovú štruktúru a vzájomne ich porovnáme.

1.1 Základné algoritmy

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

1.1.1 Popis algoritmu

Majme pole A veľkosti $|A| = N$ a označme jeho prvky $A = \{a_1, \dots, a_N\} \in X^N$. Chceme vypočítať hodnotu $f_g(A)$, kde $g : X \times Y \rightarrow Y$ je agregáčná funkcia, $g_0 \in Y$ je počiatočná hodnota a f_g je rozšírenie agregáčnej funkcie definované nasledovne:

$$\begin{aligned} f_g(\{a_1, \dots, a_k\}) &= g(a_k, f(\{a_1, \dots, a_{k-1}\})) \\ f_g(\emptyset) &= g_0 \end{aligned}$$

Túto funkciu je možné implementovať jednoducho ako jeden cyklus. Schematickú verziu implementácie uvádzame v algoritme 1.1.

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:

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

Algoritmus 1.1 Implementácia agregáčnej funkcie f_g

```

1: function AGGREGATE( $g, g_0, A$ )
2:    $y \leftarrow g_0$ 
3:   for  $i \leftarrow 1, \dots, |A|$  do
4:      $y \leftarrow g(A[i], y)$ 
5:   return  $y$ 

```

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 T_g na výpočet funkcie g je nezávislý na vstupe, bude výsledný čas na výpočet tejto funkcie

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

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 1.1, 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 zaradiť 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 jedno-

duchšia, keďže nemusí explicitne spravovať presun blokov do cache. Parameter B sme použili len počas analýzy, v samotnom návrhu algoritmu nie.

1.2 Vyhľadávanie v statickej množine

Častým problémom v informatike je nutnosť rýchlo a efektívne vyhľadávať prvok v nejakej statickej množine prvkov. V tejto sekcii popíšeme niekoľko algoritmov a dátových štruktúr, ktoré tento problém riešia. Všetky majú v *RAM* modeli časovú zložitosť $\mathcal{O}(\log N)$, kde N je počet prvkov v prehľadávanej množine, no v *cache-aware* a *cache-oblivious* modeloch sa budú líšiť počtom pamäťových presunov, ktoré vykonajú.

1.2.1 Binárne vyhľadávanie

Pozrime sa najskôr na binárne vyhľadávanie, ktoré je asi najznámejšie. V algoritme 1.2 uvádzame implementáciu, ktorá nájde prvok K v usporiadanom poli $A[0, \dots, N-1]$, pre ktoré platí $\forall i, j; 0 \leq i < j < N : A[i] \leq A[j]$.

Algoritmus 1.2 Implementácia binárneho vyhľadávania

```

1: function FIND( $A, K$ )
2:    $left \leftarrow 0$ 
3:    $right \leftarrow N$ 
4:   while  $left < right$  do
5:      $mid \leftarrow \lfloor \frac{left+right}{2} \rfloor$ 
6:     if  $A[mid] = K$  then
7:       return  $mid$ 
8:     else if  $A[mid] > K$  then
9:        $right \leftarrow mid$ 
10:    else
11:       $left \leftarrow mid + 1$ 
12:  return  $K \notin A$ 

```

Vidíme, že táto implementácia binárneho vyhľadávania nikde nepožaduje znalosť parametrov B ani M a teda ide o *cache-oblivious* algoritmus, tak ako v prípade agregáčného algoritmu 1.1.

Analýza zložitosti

Pri každej iterácii zmenšíme oblasť, ktorú treba preskúmať na polovicu. Počet iterácií a celková zložitosť bude teda riešenie rekurentného vzťahu

$$T(N) = T(N/2) + \mathcal{O}(1)$$

V prípade časovej analýzy je bázou rekurencie $T(1) = \mathcal{O}(1)$ a výsledná časová zložitosť bude $T(N) = \mathcal{O}(\log N)$.

Pri pamäťovej analýze bude bázou $T(B) = \mathcal{O}(1)$, pretože po zredukovaní prehľadávaného intervalu na veľkosť B vieme všetky potenciálne prvky načítať v $\mathcal{O}(1)$ blokoch a k ďalším presunom už nedôjde. Riešením a celkovou pamäťovo zložitostou teda bude $T(N) = \mathcal{O}(\log N - \log B) = \mathcal{O}(\log \frac{N}{B})$.

Ako však uvidíme v ďalšej sekcii, toto riešenie nie je optimálne.

1.2.2 *Cache-aware* riešenie

V prípade, že poznáme veľkosť blokov B v cache, môžeme problém vyhľadávacích stromov riešiť B-stromom [2] 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.

Toto je lepší výsledok ako dosahuje binárne vyhľadávanie. Stále však zostáva otázka, či neexistuje ešte efektívnejšie riešenie. Pozrime sa teda na spodnú hranicu tohto problému.

1.2.3 Spodná hranica pre vyhľadávanie

Ukážeme odvodenie minimálneho počtu pamäťových presunov rovnako ako v [6] technikou informačnej zložitosti. Na nájdenie daného prvku v množine obsahujúcej N položiek potrebujeme získať $\lg(2N + 1)$ informačných bitov reprezentujúcich pozíciu tohto prvku – N možných existujúcich prvkov a $N + 1$ pozícií medzi nimi (a na okrajoch) pre neexistujúci prvok. V každom pamäťovom presune načítame jeden blok veľkosti B , ktorý môže obsahovať najviac $\lg(2B + 1)$ bitov informácie – hľadaný prvok je jeden z týchto B prvkov alebo patrí v usporiadaní niekam medzi ne. V najlepšom prípade je teda potrebných aspoň

$$\frac{\lg(2N + 1)}{\lg(2B + 1)} \geq \frac{\lg N}{\lg B} = \Omega(\log_B N)$$

pamäťových presunov.

Vidíme teda, že na rozdiel od *cache-oblivious* binárneho vyhľadávania, *cache-aware* B-strom je asymptoticky optimálnym riešením tohto problému. Chceli by sme teraz nájsť *cache-oblivious* dátovú štruktúru, ktorá tiež dosahuje túto hranicu.

1.2.4 Naivné *cache-oblivious* riešenie

Predtým ako popíšeme efektívne *cache-oblivious* riešenie, pozrime sa na klasický binárny vyhľadávací strom. Asi najjednoduchší spôsob, ako usporiadať uzly (statického) binárneho stromu v pamäti, je nasledovný. Koreň uložíme na pozíciu 1. Ľavého a pravého potomka vrcholu na pozícii x uložíme na pozície $2x$ a $2x + 1$. Otec vrcholu x

bude teda na pozícii $\lfloor \frac{x}{2} \rfloor$. Toto usporiadanie voláme *BFS usporiadanie* (z anglického *breadth-first search* – prehľadávanie do šírky), keďže pozície vrcholov sa zvyšujú v rovnakom poradí ako by prebiehalo prehľadávanie tohto stromu do šírky. Príklad takto uloženého stromu je na obrázku 1.2(a).

Výhodou tohto usporiadania sú implicitné vzťahy medzi vrcholmi. Na udržiavanie stromu stačí jednorozmerné pole kľúčov. Na prechod medzi nimi môžeme použiť triviálne funkcie uvedené v algoritme 1.3.

Algoritmus 1.3 Funkcie pre získanie pozícií ľavého syna, pravého syna a rodiča vrcholu na pozícii x

1: function LEFT(x)	1: function RIGHT(x)	1: function PARENT(x)
2: return $2x$	2: return $2x + 1$	2: return $\lfloor \frac{x}{2} \rfloor$

Nevýhodou je však vysoký počet pamäťových presunov pri vyhľadávaní. Výška tohto stromu je¹ $\mathcal{O}(\log N)$. Pri načítaní vrcholu na pozícii x sa v rovnakom bloku nachádzajú vrcholy na pozíciách

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

kde $k + l = B - 1$. 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

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

$$x \leq B - 2$$

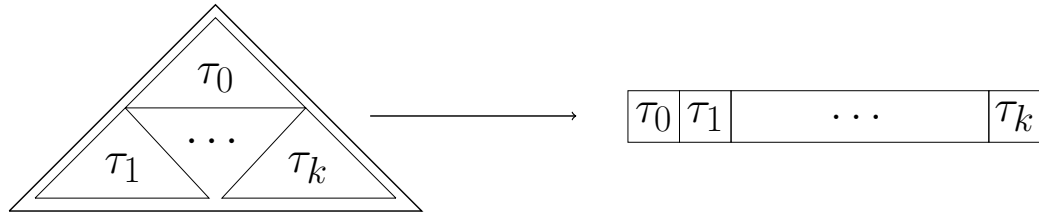
To znamená, že pre pozície $x > B - 2$ už bude potrebný pamäťový presun pre každý vrchol. Vrchol s pozíciou $B - 2$ bude mať hĺbku $\mathcal{O}(\log B)$ a teda počet vrcholov na ceste z koreňa do listu, ktorých pozície v pamäti sú väčšie ako $B - 2$ bude $\Omega(\log N - \log B)$. Pre každý z nich je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

1.2.5 Vyhľadávací strom vo *van Emde Boasovom* usporiadaní

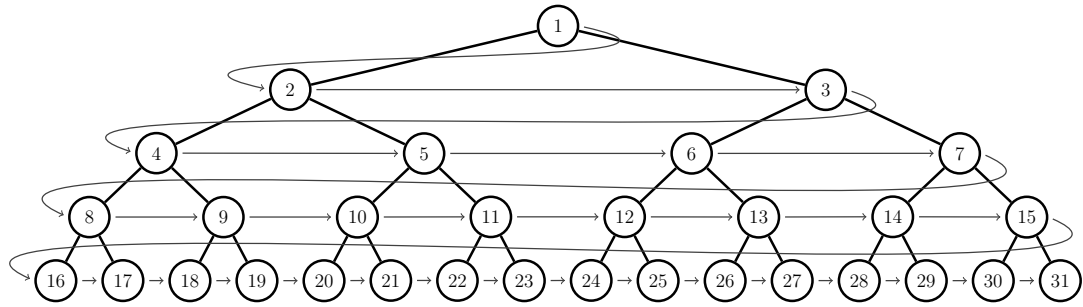
Problémom predošlého riešenia je neefektívne usporiadanie v pamäti - pri prístupe ku vrcholu sa spolu s ním v rovnakom bloku nachádzajú vrcholy, ktoré nie sú pre ďalší priebeh algoritmu podstatné.

Riešením je takzvané *van Emde Boasovo usporiadanie* (*van Emde Boas layout*, nazvané podľa *van Emde Boasových* stromov s podobnou myšlienkou), popísané v

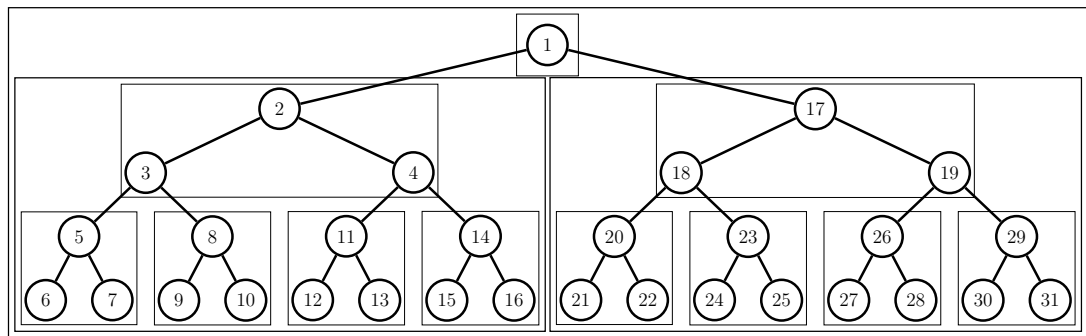
¹Nejde o vyvažovaný binárny strom a teda by výška mohla byť až $\mathcal{O}(N)$. Keďže však pracujeme so statickými dátami, môžeme ich vopred usporiadať a vyrobiť vyvážený statický strom



Obrázok 1.1: Schematické znázornenie rekurzívneho delenia pri *van Emde Boasovom* usporiadaní. Podstromy τ_0, \dots, τ_k sa uložia do súvislého úseku pamäte.



(a) Klasické usporiadanie



(b) *van Emde Boasovo* usporiadanie

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

[3, 6, 15] ktoré funguje následovne. Uvažujme úplný binárny strom výšky h . Ak $h = 1$ tak máme iba jeden vrchol v a výstupom usporiadania bude poradie (v) .

Pre $h > 1$ označme $m = \max_{i \in \mathbb{N}} \{2^i \mid 2^i < h\}$, teda najväčšiu mocninu dvoch menšiu ako h . Rozdelíme vstupný strom na podstrom τ_0 výšky $h - m$, ktorého koreňom je koreň pôvodného stromu. Zostanú nám podstromy τ_1, \dots, τ_k výšky m , ktorých korene sú potomkovia listov τ_0 a listy sú listy vstupného stromu. Všetky tieto podstromy majú približne polovičnú výšku a teda ich veľkosť je $\Theta(\sqrt{N})$ kde N je veľkosť vstupného stromu, keďže $\frac{h}{2} = \frac{1}{2} \lg N = \lg \sqrt{N}$. Rekurzívne ich uložíme do *van Emde Boasovho* usporiadania a následne uložíme za seba, výstupom teda bude poradie $(\tau_0, \tau_1, \dots, \tau_k)$. Schéma tohto delenia je na obrázku 1.1 a príklad takto usporiadaného stromu na obrázku 1.2(b).

Prechod stromom v pamäti

Na rozdiel od klasického *BFS* usporiadania nie je pri takomto usporiadaní v pamäti triviálne zistiť, na akej pozícii sa nachádzajú potomkovia alebo rodič aktuálneho vrcholu. Jedným možným riešením je spolu s každým vrcholom ukladať aj ukazovateľ na tieto relevantné vrcholy. Tým znížime počet vrcholov, ktoré sa zmestia do jedného bloku o konštantný násobok – v prípade, že je kľúč rovnako veľký ako ukazovateľ, bude počet vrcholov v jednom bloku štvrtina pôvodného počtu.

Iným riešením je vnútorne pracovať s *BFS* usporiadaním a prechádzať medzi pozíciami funkciami uvedenými v algoritme 1.3. Pri prístupe k vrcholu túto *BFS* pozíciu prevedieme na ekvivalentnú pozíciu vo *van Emde Boasovom* usporiadaní. Pri strome s N vrcholmi je možné túto konverziu realizovať v čase $\mathcal{O}(\log \log N)$ [11].

Vyhľadávanie

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 $\Theta(\log B)$. 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. Spolu sa ich teda vykoná $\mathcal{O}(\log_B N)$, čo zodpovedá spodnej hranici tohto problému.

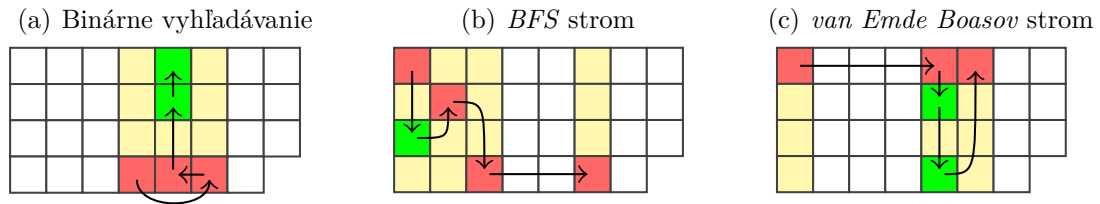
Máme teda vyhľadávanie, ktoré je rovnako ako pri *cache-aware* B-stromoch optimálne. Problémom tejto dátovej štruktúry je však nemožnosť efektívne vkladať či odoberať prvky – pri každej zmene by bolo potrebné strom preusporiadať. Dostávame tak *cache-oblivious* ekvivalent statických *cache-aware* B-stromov.

Na úpravu tohto statického stromu tak, aby efektívne zvládal operácie pridávania a odoberania, budeme potrebovať pomocnú dátovú štruktúru, ktorú popíšeme v sekcii 1.3.

1.2.6 Rozdiely v prístupe ku pamäti

Rozdiely medzi binárnym vyhľadávaním a statickými vyhľadávacími stromami v *BFS* a *van Emde Boas* usporiadaniach sú vizuálne znázornené na obrázkoch 1.4 a 1.5 na strane 9. Pozrime sa však najskôr na menší obrázok 1.3, na ktorom vysvetlíme princíp tejto vizualizácie.

Jednotlivé políčka sú položky poľa, pričom v pamäti sú usporiadané v poradí zhora nadol a zľava doprava. Samotné kľúče (vo vrchoch stromov) sú však usporiadané inak



Obrázok 1.3: Porovnanie prístupov ku pamäti v rôznych štruktúrach pri vyhľadávaní hodnoty 17 medzi 31 prvkami

a preto sú tieto prístupy inak efektívne. Vo všetkých prípadoch vyhľadáваме prvok v množine $\{1, \dots, 2^h - 1\}$, čo je počet vrcholov úplného binárneho stromu výšky h . Farby označujú stav políčka v simulovanej *cache* s veľkosťou bloku $B = 4$. Bloky sú v zarovnané tak, že jeden stĺpec predstavuje jeden blok a teda sa pri načítaní ľubovôle políčka spolu s ním načíta celý príslušný stĺpec.

Červená farba reprezentuje prvky, ktoré sa v momente prístupu nenachádzali v *cache* a bolo potrebné ich načítať (*cache miss*). Zelená naopak označuje prvky, ktoré už v *cache* boli (*cache hit*) – načítali sa spolu s nejakým červeným prvkom v rovnakom bloku. Ostatné prvky, ktoré sa načítali ako súčasť bloku ale neboli použité, sú označené žltou farbou. V tejto simulácii je počet blokov v *cache* ($\frac{M}{B}$) pre jednoduchosť neobmedzený.

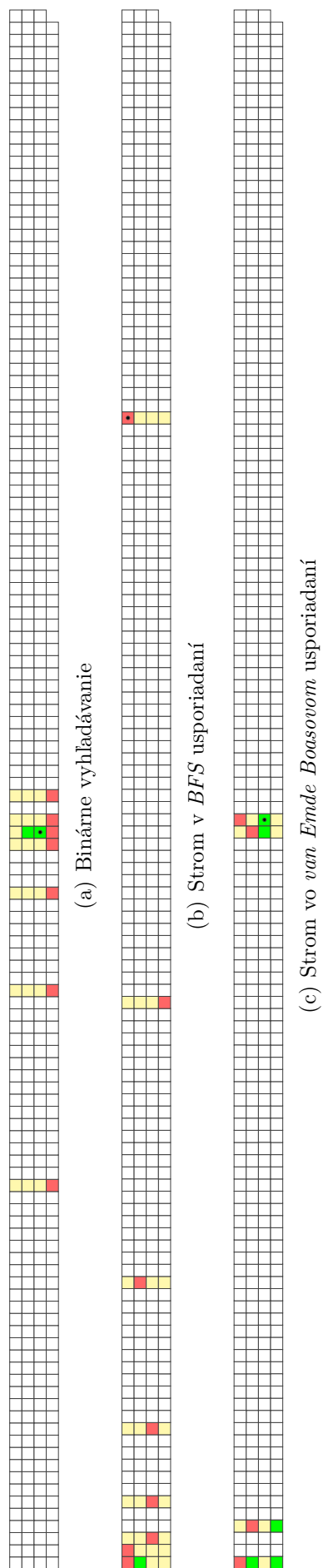
V malom obrázku 1.3 (výška stromu $h = 5$) šípky určujú poradie, v akom vyhľadávanie pristupovalo k jednotlivým prvkom. Vo veľkých obrázkoch 1.4 a 1.5 (výška stromu $h = 9$) sú šípky pre prehľadnosť vynechané a políčko s bodkou označuje pozíciu, na ktorej bol napokon požadovaný prvok nájdený. Rozdiely medzi týmito tromi prístupmi k vyhľadávaniu je lepšie vidieť práve na väčších obrázkoch.

Keďže všetky hľadané hodnoty boli zvolené ako listy týchto vyhľadávacích stromov, pristúpili operácie vyhľadávania vo všetkých troch prípadoch práve k h prvkom. Rozdiel je však v pomere počtu červených (ktoré v *cache* neboli a bolo ich potrebné načítať) a zelených (na ktoré nebol potrebný pamäťový presun) prvkov.

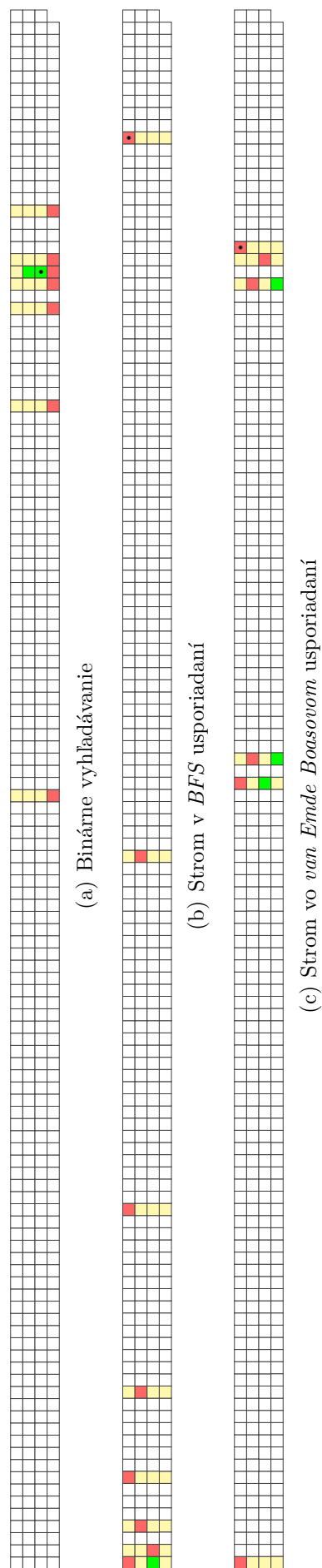
Ako ľahko vidieť, binárne vyhľadávanie začne v strede a presúva sa medzi prvkami, ktoré sú od seba čoraz menej vzdialené až do momentu, kedy už je prehľadávaný interval dostatočne malý na to, aby sa zmestil do bloku.

Naopak strom v *BFS* usporiadaní robí stále väčšie a väčšie *skoky* a porovnávané prvky sa do jedného bloku zmestia iba na začiatku.

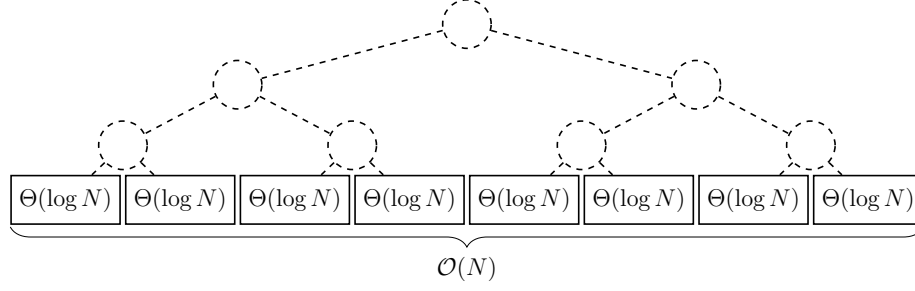
Avšak strom vo *van Emde Boasovom* usporiadaní potom čo pristúpi k nejakému prvku v ďalších krokoch pristupuje k prvkom, ktoré sú v pamäti blízko a veľký *skok*, ktorý pristúpi mimo *cache*, vykonáva menej často. Vďaka tomu dosahuje v oboch príkladoch najlepší počet zásahov do *cache*, teda najviac zelených políčok.



Obrázok 1.4: Porovnávanie prístupov ku pamäti v rôznych štruktúrach pri vyhľadávaní hodnoty 243 medzi 511 prvkami



Obrázok 1.5: Porovnávanie prístupov ku pamäti v rôznych štruktúrach pri vyhľadávaní hodnoty 427 medzi 511 prvkami



Obrázok 1.6: Usporiadané pole veľkosti N . Strom nad blokmi je len imaginárny a nezostrojuje sa v pamäti.

1.3 Usporiadané pole

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 ľubovoľné 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

Popíšeme dátovú štruktúru z [3] s malými úpravami. 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 (obrázok 1.6), 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 $o(v)$. Ďalej *hustotou*, $0 \leq d(v) \leq 1$, označíme $d(v) = \frac{o(v)}{c(v)}$.

Zvoľme ľubovoľné konštanty $\rho_0, \tau_0, \rho_d, \tau_d$ spĺňajúce

$$0 < \rho_d < \rho_0 < \tau_0 < \tau_d < 1$$

Z týchto konštánt definujeme pre vrchol s hĺbkou k *dolnú* a *hornú hranicu hustoty* (ρ_k a τ_k) následovne:

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

Dostaneme tak 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:

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

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

1.3.3 Operácie

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 $S = \Theta(\log N)$ pozícií.

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 počítadlá neprázdnych a všetkých pozícií 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 vkladani 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 v hĺbke k . Teda pred týmto prerozdelením bol vrchol u v hraniciach hustoty ($\rho_k \leq d(u) \leq \tau_k$) ale nejaký jeho potomok v nebol.

Po prerozdelení budú všetky vrcholy v danom podstromu 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

$$\begin{aligned} \frac{o(v)}{c(v)} = d(v) &\leq \tau_k & \frac{o(v) + q}{c(v)} &> \tau_{k+1} \\ o(v) &\leq \tau_k c(v) & o(v) + q &> \tau_{k+1} c(v) \\ q &> (\tau_{k+1} - \tau_k) c(v) \end{aligned}$$

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

Pri úprave intervalu blokov v podstromu vrcholu u je potrebné upraviť najviac $c(u)$ položiek, avšak táto situácia nastane až keď sa potomok v ocitne znovu 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 úplného binárneho stromu d s T listami je $d = \Theta(\log T)$. Podobným spôsobom dostaneme rovnaký odhad pre odstraňovanie.

Pri vkladani 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 $\mathcal{O}(\log^2 T)$.

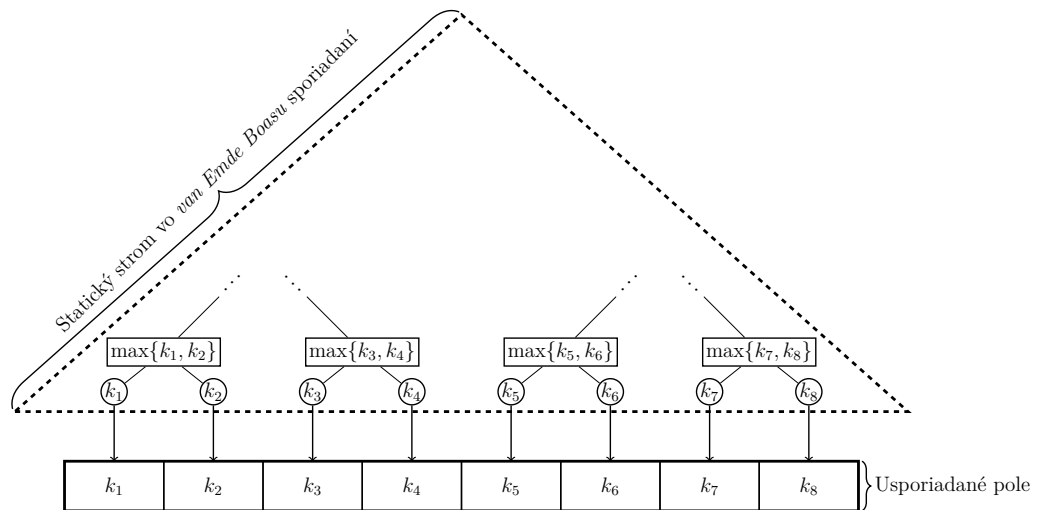
worstcase bounds? conjectured lowerbound?

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.

1.4 Dynamický B-strom

V tejto sekcií popíšeme dynamickú verziu *cache-oblivious* vyhľadávacieho stromu. Prvá verzia tejto štruktúry pochádza z [3, 4], bola avšak značne komplikovaná. Preto



Obrázok 1.7: Dynamický strom, ktorý vznikne spojením statického stromu vo *van Emde Boasovom* usporiadaní a usporiadaného pola. Šípky znázorňujú spárovanie listov a položiek pola.

použijeme zjednodušenú verziu z [5], ktorá dosahuje rovnaké výsledky ale jej popis a analýza sú podstatne jednoduchšie.

1.4.1 *Cache-aware* riešenie

V prípade *cache-aware* modelu použijeme opäť B-strom s vetvením $\Theta(B)$ ako v časti 1.2.2. Rovnako ako pre vyhľadávanie bude na vkladanie potrebných $\mathcal{O}(\log_B N)$ pamäťových presunov.

1.4.2 Popis *cache-oblivious* štruktúry

Táto dátová štruktúra vznikne zložením predchádzajúcich dvoch - statického stromu (1.2.5) a usporiadaného pola (1.3.1) - jednoduchým spôsobom. Majme usporiadané pole veľkosti $\Theta(N)$. Keďže počet položiek v usporiadanom poli je mocnina dvoch, môžeme nad ním vybudovať úplný binárny statický vyhľadávací strom uložený vo *van Emde Boasovom* usporiadaní. Listy tohto stromu budú bijektívne spárované s položkami v usporiadanom poli (obrázok 1.7).

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

1.4.3 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.

Analýza

Toto prehľadávanie prechádza cestu od koreňa k listu v strome hĺbky $\mathcal{O}(\log N)$ uloženom vo *van Emde Boasovom* usporiadaní a teda rovnako ako v časti 1.2.5 vykoná $\mathcal{O}(\log_B N)$ pamäťových presunov.

1.4.4 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 pozícií, 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 1.4.

Algoritmus 1.4 Implementácia *post-order* prechodu na aktualizáciu statického stromu

```

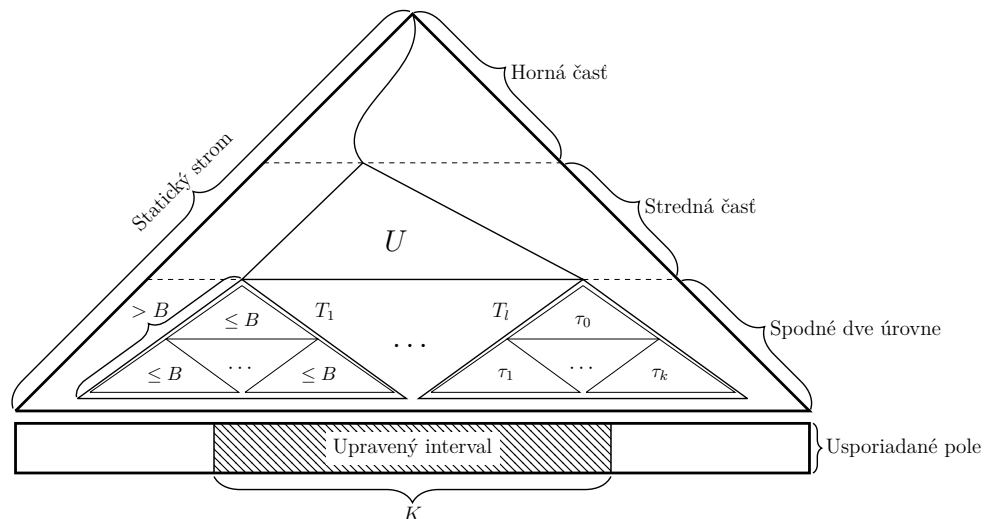
1: function UPDATE( $v, I$ )                                ▷  $v$  je vrchol stromu, ktorý práve aktualizujeme
                                                         ▷  $I$  je zmenený interval v usporiadanom poli

2:   if  $v.\text{podstrom} \cap I = \emptyset$  then                ▷ pokiaľ sa zmena tohto vrcholu
3:     return                                              ▷ určite nedotkla, tak ho môžeme preskočiť

4:   if  $v$  je list then
5:      $v.\text{kľúč} \leftarrow$  hodnota z usporiadaného pola
6:   else
7:     UPDATE( $v.\text{ľavý}$ )                                    ▷ najskôr aktualizujeme ľavého
8:     UPDATE( $v.\text{pravý}$ )                                    ▷ a pravého syna

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

```



Obrázok 1.8: Rozdelenie statického stromu na tri časti pri analýze počtu pamäťových presunov.

1.4.5 Analýza vkladania

Túto analýzu rozdelíme na tri časti (obrázok 1.8), v ktorých sa postupne pozrieme na rôzne úrovne v statickom strome, ktoré treba po vložení do usporiadaného pola aktualizovať.

Spodné dve úrovne

Uvažujme najskôr ako v časti 1.2.5 také podstromy *van Emde Boasovho* 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 Boasovým* delením z podstromu T veľkosti viac než B , rovnako, ako v časti 1.2.5: $\tau_0, \tau_1, \dots, \tau_k$. Tu τ_0 je podstrom v hornej úrovni a τ_1, \dots, τ_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

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

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 operácií 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 časť - najbližší spoločný predok

Označme ako T_1, \dots, 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, \dots, 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 potomkov neboli v prvej časti zmenené, alebo už boli aktualizované ako súčasť nejakého podstromu T_i na spodnej úrovni.

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 rekurzíe) a teda celkovo budeme potrebovať $\mathcal{O}(\frac{K}{B})$ pamäťových presunov na aktualizáciu U .

Horná časť - 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 1.4.3) bude potrebných $\mathcal{O}(\log_B N)$ pamäťových presunov.

Výsledná zložitosť

Sčítaním nasledovných zložítostí počtu pamäťových operácií

Nájdenie pozície v usporiadanom poli:	$\mathcal{O}(\log_B N)$
Vloženie do usporiadaného poľa:	$\mathcal{O}(\frac{K}{B})$
Aktualizácia spodných dvoch úrovní:	$\mathcal{O}(\frac{K}{B})$
Aktualizácia strednej úrovne:	$\mathcal{O}(\frac{K}{B})$
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 operácií pri vkladaní: $\mathcal{O}(\log_B N + \frac{\log^2 N}{B})$.

1.4.6 Odstraňovanie

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

double/half

1.5 Vylepšený dynamický B-strom

Any problem in computer science
can be solved with another level of
indirection.

David Wheeler

Oproti *cache-aware* B-stromom ma pri vkladaní *cache-oblivious* dynamický strom popísaný v predchádzajúcej sekcii navyše $\mathcal{O}(\frac{\log^2 N}{B})$ pamäťových presunov. Tento člen môžeme odstrániť 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 $\Theta(\frac{N}{\log N})$.

1.5.1 Vyhľadávanie

Najskôr vyhľadáme požadovaný blok v B-strome, čo zaberie $\mathcal{O}(\log_B \frac{N}{\log N}) = \mathcal{O}(\log_B N)$ 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: $\mathcal{O}(\log_B N)$.

1.5.2 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

popis
naj-
denia
blo-
ku -
min/max
left
child,
...

alebo odstránenie z B-stromu. Vydelením dostávame amortizovanú zložitosť vkladania a odstraňovania:

$$\mathcal{O}\left(\frac{\log_B N + \frac{\log^2 N}{B}}{\log N}\right) = \mathcal{O}\left(\frac{\log N}{B}\right)$$

Najskôr však musíme nájsť blok, ktorý budeme aktualizovať. Teda výsledná zložitosť bude $\mathcal{O}(\log_B N)$ rovnako ako pri *cache-aware* B-strome. Opäť sme teda dosiahli rovnaký počet operácií ako ekvivalentná *cache-aware* dátová štruktúra, avšak tentokrát nie v najhoršom ale amortizovanom prípade.

KAPITOLA 2

Vizualizácie

An algorithm must be seen
to be believed.

The Art of Computer Programming

DONALD E. KNUTH

Cielom tejto práce je nielen popísať *cache-oblivious* pamäťový model a rôzne dátové štruktúry v ňom, ale aj vytvoriť ich vizualizácie. Tie majú slúžiť na edukačné účely pre študentov (a učiteľov) a pomáhať pri pochopení ich fungovania.

Výsledkom práce sú vizualizácie demonštrujúce dátové štruktúry popísané v predchádzajúcich sekciách: *van Emde Boasovo* usporiadanie (sekcia 1.2.5) v statickom binárnom vyhľadávacom strome, usporiadané pole (1.3) a dynamický B-strom (1.4). Súčasťou je tiež simulácia *cache* (sekcia ??) s možnosťou voľby parametrov B a M - veľkosť bloku a celková veľkosť.

Existujúce? nič nie je...

2.1 Gnarley trees

Tieto vizualizácie sú implementované ako rozšírenie programu *Gnarley trees*, ktorý vznikol ako súčasť bakalárskej práce Jakuba Kováča [13]. Tento nástroj na vizualizáciu (prevažne stromových) dátových štruktúr bol následne v bakalárskych prácach [12, 14, 16] a ročníkových projektoch rozšírený o mnohé ďalšie dátové štruktúry a v súčasnosti podporuje desiatky štruktúr, ako napríklad červeno-čierne, sufixové a intervalové stromy, *union-find*, haldy a mnohé ďalšie.

2.1.1 Spustenie

Súčasťou práce je priložené CD obsahujúce zdrojový kód tohto programu a tiež skompilovanú verziu. Aplikácia na spustenie požaduje *JVM*¹. Spustenie je možné vykonať spustením súborov `start.sh` prípadne `start.bat`, podľa operačného systému. Dostupné sú tiež individuálne vizualizácie vo forme *Java appletov*, ktoré je možné spustiť otvorením súboru `index.html` v preferovanom internetovom prehliadači.

Obsah tejto CD prílohy je tiež dostupný na .

[url](#)

2.1.2 Funkcionalita

Program umožňuje užívateľom zobrazovať tieto štruktúry a manipulovať s nimi. Všetky operácie sú rozložené na malé, jednoduché kroky a každý je vysvetlený, keď sa vykonáva. Je možné posúvať sa po krokoch dopredu, ale aj vracieť sa dozadu, a teda sa dá kedykoľvek vrátiť až k počiatočnému stavu. Toto je dôležité pri experimentovaní s danou štruktúrou, kedy dve rôzne operácie (alebo jedna operácia s dvoma rôznymi parametrami) spôsobia rôzne správanie a výsledky. Užívateľ má takto možnosť jednoducho sa po vykonaní prvej operácie vrátiť do predošlého stavu a preskúmať správanie druhej z nich.

Celý program je taktiež dvojjazyčný - je možné prepnúť medzi angličtinou a slovenčinou, čo umožňuje širšie použitie týchto vizualizácií.

2.1.3 Prehľad programu

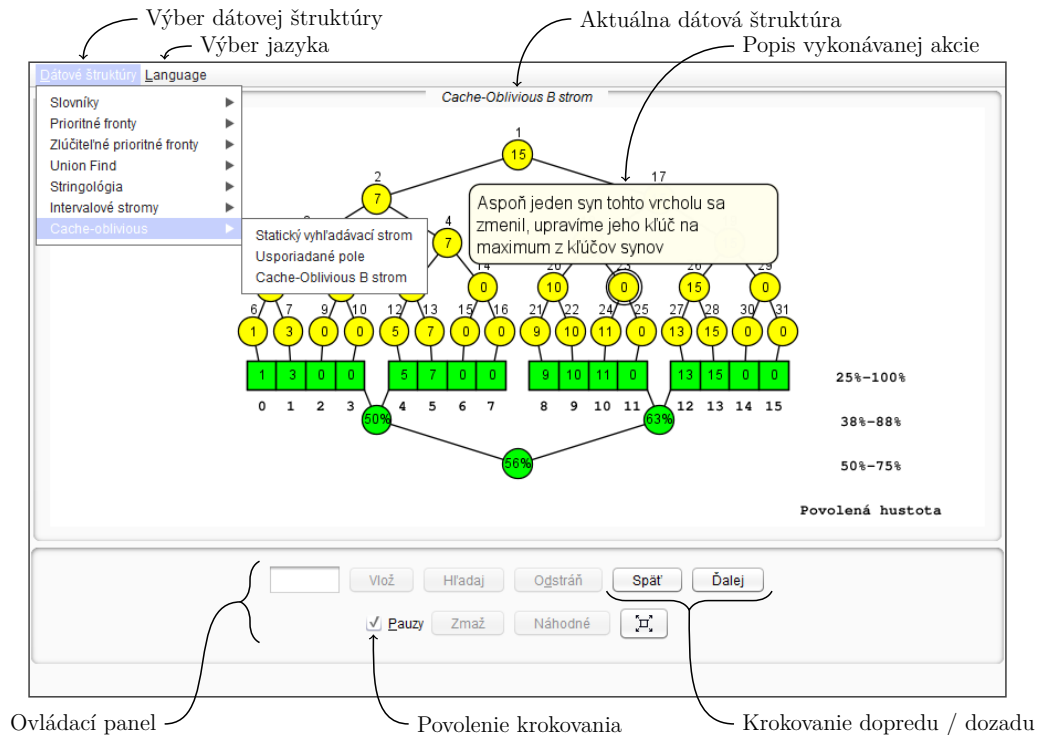
Program sa skladá z troch hlavných častí (obrázok ??). Najvrchnejšia časť okna tvorí hlavné menu, v ktorej môžeme voliť dátové štruktúry a prepínať jazyk rozhrania. Dátové štruktúry sú pre prehľadnosť rozdelené do niekoľkých kategórií. Tie popísané a implementované v tejto práci sa nachádzajú v kategórii **Cache-oblivious**.

V spodnej časti okna sa nachádza ovládací panel, ktorý obsahuje vstupné pole pre hodnotu, ktorú chceme vyhľadať alebo vložiť a tlačidlá na vykonanie týchto akcií. Ďalej obsahuje tlačidlá na prechod do ďalšieho kroku a návrat do predchádzajúceho stavu s možnosťou toto krokovanie vypnúť.

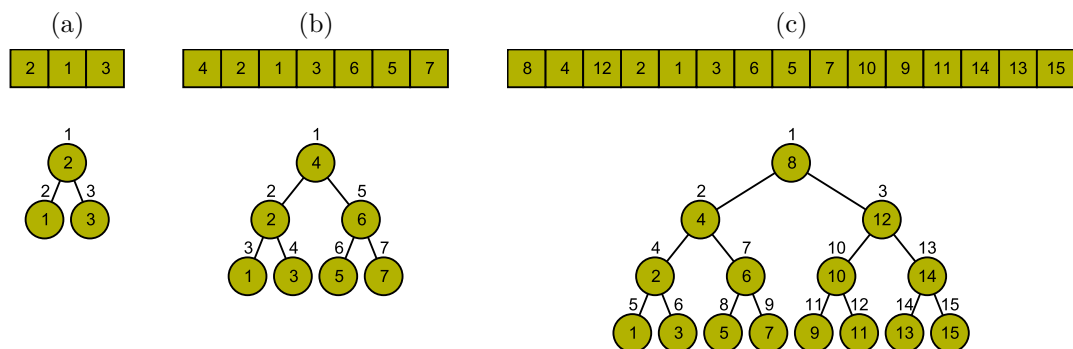
Najväčšia, prostredná časť okna zobrazuje vizualizáciu aktuálnej dátovej štruktúry. Toto zobrazenie je vektorové a je možné ho posúvať, približovať a oddaľovať. Zároveň sa tu zobrazujú informácie o aktuálne vykonávanej akcii (ak je povolené krokovanie) a ďalšie vizualizačné prvky ako šípky alebo význačné vrcholy.

Podrobnejší popis užívateľského rozhrania a návod na používanie sa nachádza v bakalárskej práci Jakuba Kováča [13] v siedmej kapitole.

¹*Java Virtual Machine*, dostupné na <https://www.java.com/en/download/>



Obrázok 2.1: Užívateľské rozhranie počas operácie vkladania kľúča 10 do dynamického *cache-oblivious* B-stromu (sekcia 1.4).

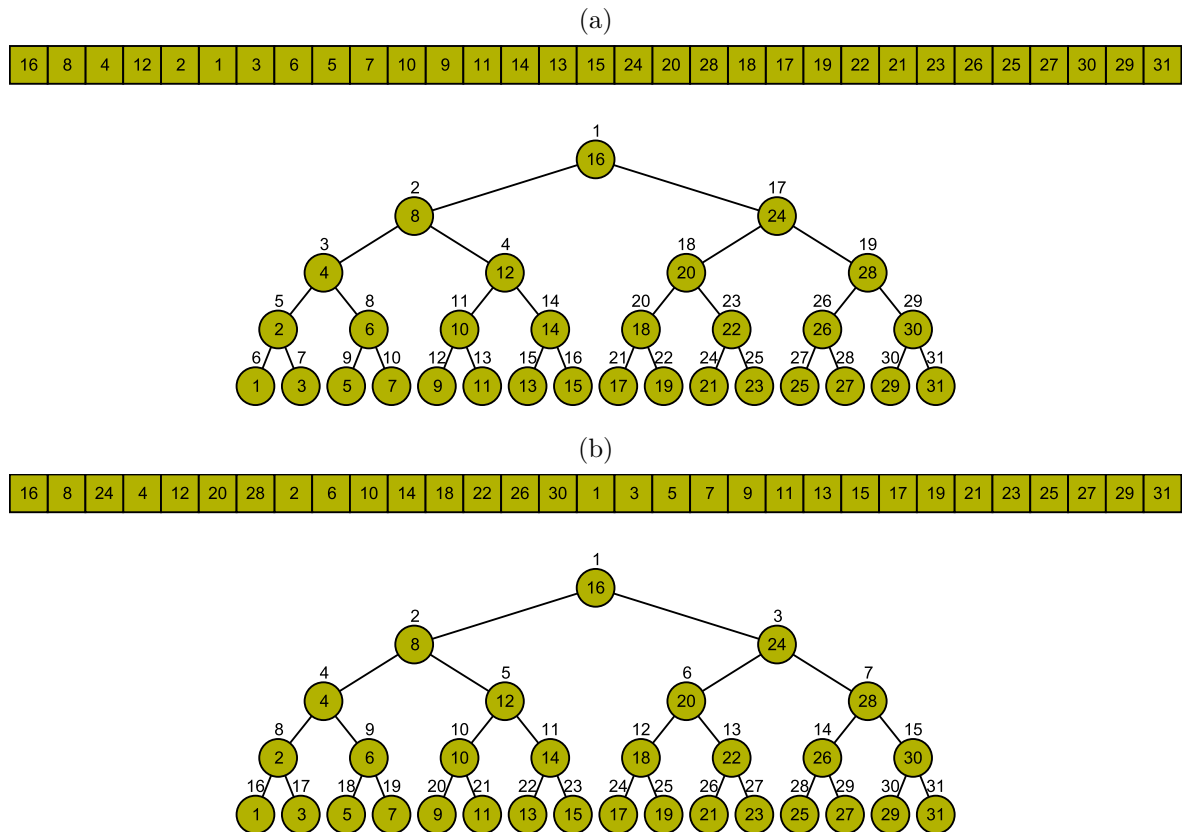


Obrázok 2.2: Statické stromy rôznych veľkostí (výšok) vo *van Emde Boasovom* usporiadaní.

2.2 Statický strom

Najjednoduchšou dátovou štruktúrou je statický vyhľadávací strom. Implementovali sme vytvorenie tohto stromu a jeho uloženie v pamäti. Je možné strom zväčšiť alebo zmenšiť podľa preferencií - ukážky stromov rôznych veľkostí sú na obrázku ???. Čísla vo vnútorných vrcholoch sú kľúče, čísla nad vrcholom určujú jeho pozíciu v pamäti. Obdĺžnik nad stromom reprezentuje uloženie tohto stromu v poli. Vnútné čísla sú opäť kľúče, pričom sú zoradené podľa svojich pozícií zľava (pozícia 1) doprava.

Medzi uložením vo *van Emde Boasovom* usporiadaní a klasickým *BFS* usporiadaním (ako v časti 1.2.4) je možné prepínať. Zmenia sa pritom čísla udávajúce pozície



Obrázok 2.3: Rozdiel medzi *BFS* a *van Emde Boasovým* usporiadaním na strome výšky 5. Kľúče zostávajú rovnaké, líšia sa však ich pozície v pamäti reprezentované malými číslami nad vrcholmi a polom nad koreňom.

vrcholov v pamäti a ich poradie v poli nad stromom. Rozdiel medzi týmito dvoma usporiadaniami vidieť na obrázku ???. Pozície sa zhodujú s obrázkom 1.2(b).

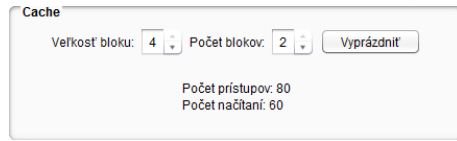
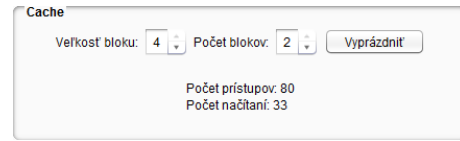
2.2.1 Simulácia *cache*

Porovnanie týchto dvoch usporiadaní je rozšírené o simuláciu *cache*. Užívateľ si môže zvoliť parametre *cache* - počet vrcholov B , ktoré sa zmestia do jedného bloku a počet blokov $\frac{M}{B}$ v *cache*. Tiež je možné *cache* kedykoľvek vyprázdniť – odstrániť z nej všetky načítané bloky. Simulácia na výmenu stránok používa stratégiu *FIFO*, ktorá je popísaná v časti ???.

Táto simulácia zároveň počíta počet prístupov k vrcholom pri vyhľadávaní a počet presunutých blokov do *cache*. V najhoršom prípade by tieto dve čísla boli rovnaké (ak treba každý vrchol načítať osobitne) avšak pri *cache* s blokmi veľkosti $B > 1$ a s *van Emde Boasovým* usporiadaním dochádza k podstatnému zlepšeniu - ušetreniu počtu presunutých blokov.

To môžeme vidieť na jednoduchom príklade, kedy v strome výšky 5 postupne vyhľadáme všetkých 16 kľúčov, ktoré sa nachádzajú v listoch tohto stromu. V oboch usporiadaniach bude počet prístupov rovnaký, avšak počet načítaní blokov z disku

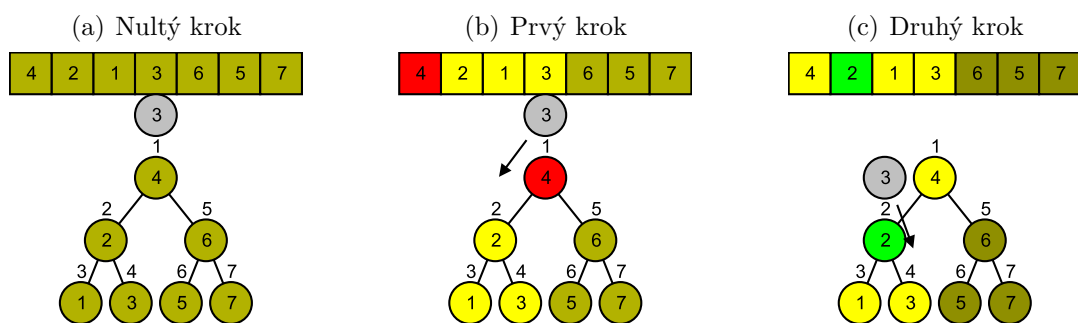
do *cache* je v tomto príklade pri *BFS* usporiadaní takmer dvakrát väčší ako pri *van Emde Boasovom* usporiadaní. Obrázok ?? ukazuje stav týchto štatistík po nájdení posledného listu.

(a) Strom v *BFS* usporiadaní(b) Strom vo *van Emde Boasovom* usporiadaní

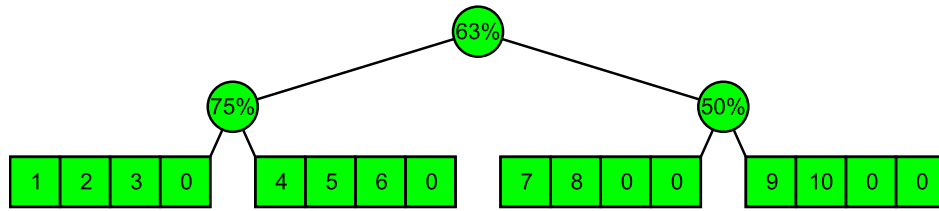
Obrázok 2.4: Porovnanie štatistík simulovanej *cache* po prechode listov $1, 3, \dots, 31$ stromu výšky 5. Parametre *cache* sú v oboch prípadoch rovnaké ($B = 4$, $M = 2B = 8$), rozdiel je však v usporiadaní v pamäti.

Ako vizualizácia prítomnosti v *cache* slúži farba - vrcholy a položky poľa obsahujúce kľúče majú svetlejšiu farbu pozadia v prípade, že je daný blok v *cache* a tmavšiu ak je mimo. V strome je vďaka tomu ľahko vidieť, ktorá časť je načítaná a je možné ňou prechádzať bez ďalších presunov. V prípade *van Emde Boasovho* usporiadania pôjde prevažne o časť podstromu aktuálne porovnávaného vrcholu, avšak pri klasickom usporiadaní to budú práve vrcholy mimo tohto podstromu, o ktorých už vieme, že nie sú pri vyhľadávaní potrebné.

Pri krokování vyhľadávania je pri prístupe k vrcholu tiež použitá zelená alebo červená farba na jeho dočasné zafarbenie (podobne ako v časti 1.2.6) podľa toho, či sa v danom momente v *cache* nachádzal (*cache hit*) alebo nie (*cache miss*). Ukážka tohto zafarbovania je na obrázku ??.



Obrázok 2.5: Simulácia *cache* počas vyhľadávania kľúča 3. Pred prvým prístupom je *cache* prázdna. V prvom kroku načítame koreň, ktorý je označený červenou farbou (*cache miss*), keďže nebol v *cache*. Spolu s ním sa v jednom bloku načítali ďalšie vrcholy, ktoré sú označené svetlejšou farbou. V druhom kroku je vrchol s kľúčom 2 označený zelenou farbou (*cache hit*), keďže už bol do *cache* načítaný v predchádzajúcom kroku.



Obrázok 2.6: Usporiadané pole obsahujúce hodnoty 1 až 10. Všetky vrcholy majú hustotu v hraniach hustoty. Hodnoty 0 reprezentujú prázdne pozície.

2.3 Usporiadané pole

Ďalšou implementovanou vizualizáciou je usporiadané pole (obrázok ??), ktoré bolo popísané v časti 1.3. Bloky, na ktoré je toto pole imaginárne rozdelené sú znázornené tým, že sú od seba oddelené medzerou. Taktiež sa zobrazuje imaginárny strom nad týmito blokmi, ktorým sa pri vkladaní prechádza. Hodnoty vo vrcholoch reprezentujú hustotu (v percentách) intervalu v príslušnom podstromi. Farby (zelená alebo červená) vrcholov indikujú, či sa hustota daného vrcholu nachádza v hraniciach.

Táto vizualizácia podporuje vkladanie hodnoty na ľubovoľnú pozíciu v poli. V prípade, že sa pole preplní, automaticky sa vytvorí nové, dvojnásobne väčšie. Po vložení hodnoty je tiež vyznačený interval, ktorý sa zmenil.

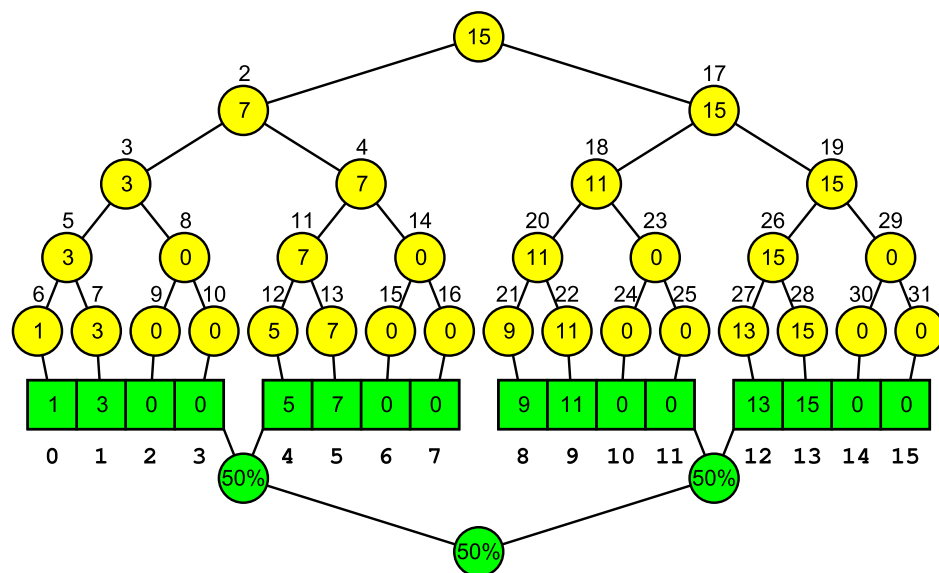
2.4 Dynamický strom

Vizualizácia dynamického stromu (časť 1.4) vznikla spojením predchádzajúcich dvoch vizualizácií (obrázok ??). Horná časť je statický strom vo *van Emde Boasovom* usporiadaní (obrázok ??), dolná je usporiadané pole (obrázok ??), pričom strom hustôt je tu preklopený nadol, aby sa neprekrýval so statickým stromom. Hrany medzi nimi spájajú listy stromu s prvkami usporiadaného pola.

V tomto strome je možné vyhľadávať a vkladať do neho nové kľúče. Pri vkladaní prebehne najskôr vloženie do usporiadaného pola rovnako, ako pri jeho samostatnej vizualizácii. Následne sa aktualizujú kľúče statického stromu. V prípade zdvojnásobenia veľkosti statického pola sa vytvorí nový, väčší statický strom.

2.5 Implementácia

?

Obrázok 2.7: Vizualizácia dynamického *cache-oblivious* B-stromu.

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] BAYER, Rudolf: Binary B-trees for Virtual Memory. In: *Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. New York, NY, USA : ACM, 1971 (SIGFIDET '71), 219–235
- [3] BENDER, Michael A. ; DEMAINE, Erik D. ; FARACH-COLTON, Martin: Cache-Oblivious B-Trees. In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*. Redondo Beach, California, November 12–14 2000, S. 399–409
- [4] BENDER, Michael A. ; DEMAINE, Erik D. ; FARACH-COLTON, Martin: Cache-Oblivious B-Trees. In: *SIAM Journal on Computing* 35 (2005), Nr. 2, S. 341–358
- [5] BENDER, Michael A. ; DUAN, Ziyang ; IACONO, John ; WU, Jing: A locality-preserving cache-oblivious dynamic dictionary. In: *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms* Society for Industrial and Applied Mathematics, 2002, S. 29–38
- [6] DEMAINE, Erik D.: Cache-Oblivious Algorithms and Data Structures. In: *Lecture Notes from the EEF Summer School on Massive Data Sets*. BRICS, University of Aarhus, Denmark, June 27–July 1 2002
- [7] DREPPER, Ulrich: What every programmer should know about memory. In: *Red Hat, Inc* 11 (2007)
- [8] 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
- [9] INTEL CORPORATION: *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2014 (248966-029)

- [10] INTEL CORPORATION: *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2014 (325462-050US)
- [11] KASHEFF, Zardosht: *Cache-oblivious dynamic search trees*, Massachusetts Institute of Technology, Diss., 2004
- [12] KOTRLOVÁ, Katarína: *Vizualizácia háld a intervalových stromov*, Univerzita Komenského v Bratislave, bakalárska práca, 2012
- [13] KOVÁČ, Jakub: *Vyhľadávacie stromy a ich vizualizácia*, Univerzita Komenského v Bratislave, bakalárska práca, 2007
- [14] LUKČA, Pavol: *Perzistentné dátové štruktúry a ich vizualizácia*, Univerzita Komenského v Bratislave, bakalárska práca, 2013
- [15] PROKOP, Harald: *Cache-oblivious algorithms*, Massachusetts Institute of Technology, Diss., 1999
- [16] TOMKOVIČ, Viktor: *Vizualizácia stromových dátových štruktúr*, Univerzita Komenského v Bratislave, bakalárska práca, 2012