Úvod

Pri tvorbe efektívnych algoritmov sa najčastejšie zaujímame o časovú zložitosť a prípadne aj pamäťovú zložitosť. To dosahujeme analýzou v takzvanom *RAM* modeli, v ktorom je prístup k dátam v pamäti považovaný 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, však tento model prestáva vystihovať realitu.

V skutočnosti však nemusí ísť priamo o veľmi veľké objemy dát a pomalé médiá. Rozdiely v prístupových rýchlostiach medzi vyrovnávacou pamäťou na procesore a hlavnou operačnou pamäťou sú dostatočne veľké na to, aby sa nám v praxi oplatilo minimalizovať počet presunov aj medzi nimi. Ich veľkosť je pritom rádovo len niekoľko kilobajtov, čo je častokrát podstatne menej ako objem spracúvaných dát a nemôžeme teda predpokladať, že k týmto pomalým prístupom nedôjde.

Dôležitým faktorom teda môže byť aj počet týchto pamäťových operácií. Tie majú priamy vplyv na výslednú časovú zložitosť, a preto je výhodné minimalizovať množstvo zápisov a čítaní z pomalého úložiska a snažiť sa využívať práve rýchlu vyrovnávaciu pamäť.

Klasickým prístupom v týchto prípadoch sú takzvané *cache-aware* algoritmy, ktoré poznajú presné parametre pamäťového systému pre dosiahnutie požadovanej efektivity. Následkom toho môže byť viazanosť na konkrétny systém či náročnosť a komplikovanosť implementácie. V tejto práci sa budeme venovať *cache-oblivious* algoritmom a dátovým štruktúram, ktoré tieto parametre nepoznajú, no napriek tomu sú v istých prípadoch asymptoticky rovnako efektívne ako ich *cache-aware* ekvivalenty.

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 tá 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 prípade *cache-oblivious* algoritmov a dátových štruktúr nám stačí jedna inštancia – keďže nevyžaduje poznať parametre žiadnej úrovne, bude rovnako efektívna bez ohľadu na ich hodnoty a teda rovnako efektívna na každej úrovni súčasne.

V tejto práci vysvetľujeme problematiku analýzy počtu pamäťových operácií, popisujeme *cache-oblivious* pamäťový model, porovnávame ho s *cache-aware* modelom a uvádzame prehľad vybraných algoritmov a dátových štruktúr. Pre popísané štruktúry uvádzame analýzu ich správania v *cache-oblivious* modeli.

Hlavným výsledkom práce je implementácia vizualizácií týchto dátových štruktúr. Vizualizácie sú implementované ako rozšírenie programu *Gnarley trees*, ktorý vznikol ako bakalárska práca Jakuba Kováča a neskôr bol rozšírený o ďalšiu funkcionalitu v ročníkových projektoch a bakalárskych prácach. Tento program sme rozšírili o podporu pre simuláciu vyrovnávacej pamäte a pridali sme vizualizácie niekoľkých *cache-oblivious* dátových štruktúr. Cieľom je umožniť užívateľom experimentovať s týmito štruktúrami, sledovať ich správanie krok po kroku. Tieto kroky sú podrobne vysvetlené, čo uľahčuje porozumeniu ich fungovania.

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) [2], v ktorom sa predpokladá možnosť pristupovať k ľubovoľnému 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 [11]. 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 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. Ako vidieť, rozdiel v prístupovej dobe k rôznym dátam môže byť až 10 000 000-násobný. Naskytuje sa teda otázka, či je užitočné a presné hovoriť o konštantnom čase.

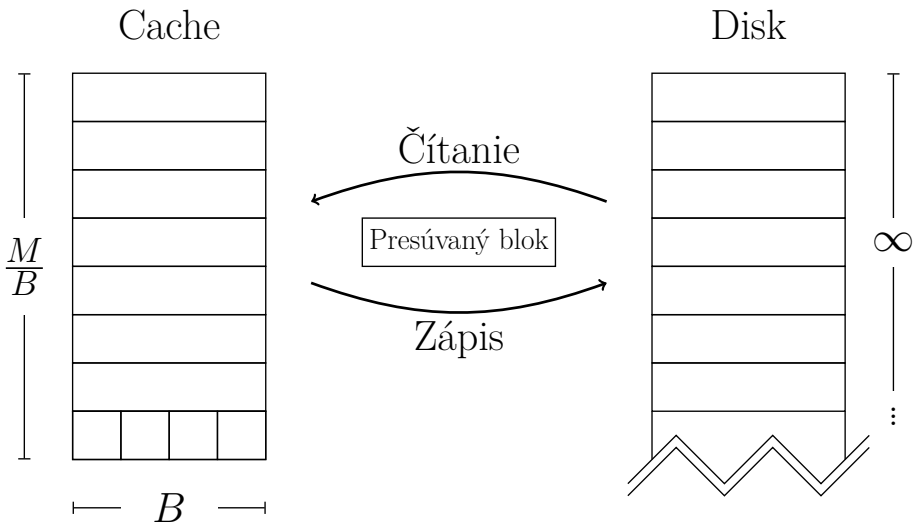
Dôsledkom tejto hierarchie je závislosť výslednej rýchlosti algoritmu od jeho prístupov k pamäti. Operácie, ktoré využívajú dáta uložené na disku potrvajú dlhšie ako tie,

Tabuľka 1.1: Približné parametre rôznych úrovní pamäte. Hodnoty troch úrovní *cache* na procesore (L1, L2 a L3) uvádzame pre mikroarchitektúru Intel Haswell. [13, 14]

Úroveň	**Veľkosť**	**Odozva**		**Asociativita**	**Veľkosť bloku**
L1	64 KiB[1]	4clk[2]	≈ 1 ns	8	64 B
L2	256 KiB[1]	11clk[2]	≈ 4 ns	8	64 B
L3	2–20 MiB	36clk[2]	≈ 12 ns		64 B
RAM	≈ 8 GiB		≈ 100 ns		16 B
Disk	≈ 1 TiB		≈ 0.1–10 ms		4 KiB–2 MiB

[1] Hodnota pre jedno jadro procesora.
[2] 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

ktoré využívajú iba dáta v registroch. Pri prístupe k dátam na disku sa v skutočnosti tieto dáta postupne presunú z disku do hlavnej pamäte, do *cache* na procesore a napokon do registrov. Toto zabezpečí, že ich opakované použitie, pokiaľ nebudú dovtedy z *cache* odstránené, bude rýchlejšie. 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

Jedným zo spôsobov, 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.

Prístup k dátam, ktoré sa práve nachádzajú v *cache* voláme *cache hit* (zásah *cache*). Naopak prístup, ktorý vyžaduje dáta najskôr presunúť z disku do *cache* voláme *cache miss* (minutie *cache*). Tieto 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. Takýto algoritmus voláme *cache-aware* (uvedomu-

júci si *cache*). Súčasťou tohto algoritmu by bolo 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. Toto nemusí byť explicitnou súčasťou algoritmu a môže byť riešené na inej úrovni.

Tento model 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 tieto algoritmy optimalizovať pre všetky susedné dvojice a ich parametre. Stále však zostáva problémom viazanosť algoritmu na tieto parametre a pri ich zmene prestáva byť optimálny.

1.2 Cache-oblivious model

Druhým spôsobom, ktorý zohľadňuje nekonštantný prístupový čas k údajom v pamäti, je takzvaný *cache-oblivious model* (na *cache* nedbajúci), v ktorom uvažujeme rovnakú dvoj-úrovňovú pamäť zloženú z disku a *cache* [12, 20]. 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.

Výhodou oproti *cache-aware* algoritmom 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 značne problematický.

Ďalšou výhodou je, že takýto *cache-oblivious* algoritmus bude (rovnako) efektívny medzi každou dvojicou susedných úrovní. Vzhľadom na to, že hodnoty parametrov nepozná, bude pre ľubovolnú takú dvojicu pracovať rovnako efektívne ako pre každú inú.

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, je možné presunúť dáta bez nutnosti nahradenia. V opačnom prípade je však potrebné uvoľniť miesto tým, že sa vyberie blok z *cache* (ak bol tento blok upravený, najskôr sa jeho obsah zapíše späť na disk), ktorý bude následne prepísaný požadovaným blokom. Tento proces sa nazýva výmena stránok (*page replacement*), a algoritmus rozhodujúci, ktorý blok z cache odstrániť, voláme stratégia výmeny stránok (*page-replacement strategy*). Dve základné stratégie výmeny stránok sú *LRU* a *FIFO*.

Stratégia *LRU* (least recently used – najdlhšie nepoužité) 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 – prvé dnu, prvé von) je ešte jednoduchšia – bloky udržiavame zoradené podľa poradia, v akom sme ich vložili do *cache*. Keď vyberáme blok na odstránenie, vezmeme ten, ktorý bol pridaný najskôr.

Problémy

Ak by sa táto stratégia správala tak, že vždy odstráni blok, ktorý bude potrebný v ďalšom kroku algoritmu, tak by bolo možné efektívne pracovať len s jedným blokom v *cache*, napriek tomu, že by sa ich do *cache* zmestilo viac. Vzhľadom na to, že analýza množstva *cache-oblivious* algoritmov predpokladá istý minimálny počet blokov, ktorý sa zmestí do *cache*, bol by tento predpoklad nenaplnený. To by mohlo spôsobiť, že by algoritmus vykonal viac pamäťových presunov ako táto analýza predpovedala.

Ďalším problémom je takzvaná *asociatívnosť* cache – počet pozícií v *cache*, na ktoré môžeme daný blok uložiť, ktorý je z praktických dôvodov často obmedzený. 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*. V moderných systémoch (tabuľka 1.1) sa asociativita pohybuje okolo 8, čo znamená, že daný blok je v *cache* možné umiestniť len na 8 z $\frac{M}{B}$ pozícií.

Ideálna *cache*

Tieto problémy *cache-oblivious* model obchádza predpokladom ideálnej *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 nerealizovateľný, keďže by stratégia výmeny stránok musela predpovedať budúce kroky algoritmu. Nasledovné lemy však ukazujú, že aj 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. *[12, 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. [12, Lemma 16]*

1.3 Prehľad výsledkov v cache-oblivious modeli

Pre širší rozhľad v tejto problematike uvádzame v tabuľke 1.2 na strane 8 zoznam niekoľkých bežných problémov. Ku každému uvádzame ako príklad najlepšie známe *cache-aware* a *cache-oblivious* algoritmy a dátové štruktúry, ktoré ten problém riešia, spolu s asymptotickou analýzou počtu pamäťových presunov.

V tejto práci sa budeme venovať prevažne problému vyhľadávacích stromov, konkrétne B-stromom. V nasledujúcej kapitole uvedieme ich fungovanie a analýzu počtu pamäťových presunov v *external-memory* modeli. V tretej kapitole popíšeme vizualizácie, ktoré sme pre tieto štruktúry vytvorili.

Tabuľka 1.2: Prehľad výsledkov *cache-aware* a *cache-oblivious* algoritmov a dátových štruktúr pre rôzne problémy. Odhady udávajú počet pamäťových operácií, pričom M a B sú parametre *cache*. Skratka *amort.* označuje amortizovanú zložitosť.

Vyhľadávacie stromy	**Vyhľadávanie**	**Vkladanie**	**Prechod**[1]	
cache-aware B-strom	$\mathcal{O}(\log_B N)$	$\mathcal{O}(\log_B N)$	$\mathcal{O}(\frac{K}{B})$	[5, 24]
cache-oblivious B-strom	$\mathcal{O}(\log_B N)$	$\mathcal{O}(\log_B N + \frac{\log^2 N}{B})$ *amort.*	$\mathcal{O}(\frac{K}{B})$	[6, 7, 8], časť 2.4
cache-oblivious B-strom	$\mathcal{O}(\log_B N)$	$\mathcal{O}(\log_B N)$ *amort.*	$\mathcal{O}(\frac{K}{\min\{B, \log N\}})$	[6, 7, 8], časť 2.5
Usporiadaná postupnosť	**Vkladanie**	**Prechod**[1]		
cache-aware spájaný zoznam	$\mathcal{O}(1)$ *amort.*	$\mathcal{O}(\frac{K}{B})$		[10, 19]
cache-oblivious usporiadané pole	$\mathcal{O}(\frac{\log^2 N}{B})$	$\mathcal{O}(\frac{K}{B})$		[6, 9], časť 2.3
Triedenie				
cache-aware mergesort	$\mathcal{O}(\frac{N}{B} \log_{M/B} \frac{N}{B})$			[10, 24]
cache-oblivious funnelsort	$\mathcal{O}(\frac{N}{B} \log_{M/B} \frac{N}{B})$			[4, 10]
Prioritné fronty	**Vkladanie**	**Výber minima**		
cache-aware buffer tree	$\mathcal{O}(\frac{1}{B} \log_{M/B} \frac{N}{B})$ *amort.*	$\mathcal{O}(\frac{1}{B} \log_{M/B} \frac{N}{B})$ *amort.*		[3]
cache-oblivious prioritná fronta	$\mathcal{O}(\frac{1}{B} \log_{M/B} \frac{N}{B})$ *amort.*	$\mathcal{O}(\frac{1}{B} \log_{M/B} \frac{N}{B})$ *amort.*		[4, 10]

[1] Prechod K po sebe idúcich prvkov

KAPITOLA 2

Cache-oblivius 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. Najskôr sa však pozrieme na jeden jednoduchý príklad, ako túto analýzu riešime.

2.1 Základný algoritmus

Na demonštráciu *cache-oblivious* algoritmov a ich analýzy v *external-memory* modeli použijeme jednoduchý algoritmus, ktorý počíta agregačnú funkciu nad hodnotami uloženými v poli. Uvažujeme pole $A = \{a_1, \ldots, a_n\}$ a danú funkciu g s počiatočnou hodnotou g_0. Chceme vypočítať hodnotu $f_g(A)$, kde f_g je rozšírenie agregačnej funkcie definované následovne:

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

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

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

Algoritmus 2.1 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
```

2.1.1 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 je nezávislý na vstupe, bude výsledný čas na výpočet tejto funkcie lineárny od veľkosti poľa N, teda $T(N) = \mathcal{O}(N)$.

Analýza počtu pamäťových presunov

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 zarovnať pole v pamäti tak, aby zabralo najmenší potrebný počet 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. Parameter B sme použili len počas analýzy, v samotnom návrhu algoritmu nie.

2.2 Vyhľadávanie v statickej množine

Častým problémom v informatike je nutnosť rýchlo a efektívne vyhľadávať prvok v nejakej statickej množine prvkov. Tento problém by šiel riešiť hašovaním, budeme však uvažovať usporiadanú množinu, v ktorej chceme vedieť efektívne pristúpiť k predchádzajúcemu a nesledujúcemu prvku. 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.

Tento rozdiel je spôsobený iným usporiadaním prvkov v pamäti. Keďže majú všetky uvedené prístupy logaritmickú časovú zložitosť bude počet presunutých blokov najviac $\mathcal{O}(\log N)$. V tomto odhade však nevystupujú žiadne parametre *cache* a teda ich zväčšenie tento algoritmus nemusí urýchliť. Ukážeme ale dátovú štruktúru, ktorá dosahuje podstatne lepší výsledok $\mathcal{O}(\log_B N)$ a bude nám tiež slúžiť ako základ pri riešení dynamickej verzie tohto problému.

2.2.1 Binárne vyhľadávanie

Ako prvé popíšeme binárne vyhľadávanie, ktoré je asi najznámejšie. V algoritme 2.2 uvádzame implementáciu, ktorá nájde pozíciu prvku K v usporiadanom poli $\texttt{A}[0, \ldots, N-1]$, pre ktoré platí $\forall i, j;\ 0 \leq i < j < N : \texttt{A}[i] \leq \texttt{A}[j]$.

Algoritmus 2.2 Implementácia binárneho vyhľadávania

```
1:  function FIND(A, K)
2:      left ← 0
3:      right ← N
4:      while left < right do
5:          mid ← ⌊(left+right)/2⌋
6:          if A[mid] = K then
7:              return mid
8:          else if A[mid] > K then
9:              right ← mid
10:         else
11:             left ← mid + 1
12:     return K ∉ 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 agregačného algoritmu 2.1.

Analýza zložitosti

Pri každej iterácií 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 zložitosťou 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.

2.2.2 *Cache-aware* riešenie

V prípade, že poznáme veľkosť blokov B v cache, môžeme problém vyhľadávacích stromov riešiť B-stromom [5] 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.

2.2.3 Spodná hranica pre vyhľadávanie

Pri vyhľadávaní potrebujeme pristúpiť k aspoň $\Omega(\log N)$ prvkom. Tie by v ideálnom prípade boli uložené v $\Omega(\frac{\log N}{B})$ blokoch, čo tvorí spodnú hranica počtu pamäťových presunov. Tento odhad je ale príliš optimistický – ako ukážeme odvodením väčšej spodnej hranice, nie je možné ho dosiahnuť.

Toto odvodenie spravíme rovnako ako v [10] 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

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

pamäťových presunov. Tento odhad je na rozdiel od predchádzajúceho dosiahnuteľný, napríklad práve *cache-aware* B-stromami.

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

2.2.4 Naivné *cache-oblivious* riešenie

Predtým ako popíšeme efektívne *cache-oblivious* riešenie, sa pozrime 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 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 2.3.

Algoritmus 2.3 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 ⌊x/2⌋
```

Nevýhodou je však vysoký počet pamäťových presunov pri vyhľadávaní. Výška tohto stromu je[1] $\mathcal{O}(\log N)$. Pri načítaní vrcholu na pozícii x sa v rovnakom bloku nachádzajú vrcholy na pozíciach

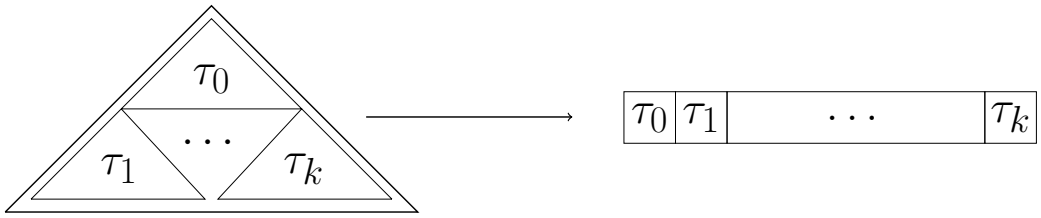
$$x-k, \ldots, x-1, x, x+1, \ldots, x+\ell$$

kde $k+\ell = B-1$. 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 $\ell = B-1$. Aby sa v tomto intervale nachádzali požadované vrcholy, musí platiť

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

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

[1] Nejde 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 2.1: Schematické znázornenie rekurzívneho delenia pri *van Emde Boasovom* usporiadaní. Podstromy $\tau_0, \ldots, \tau_k$ sa uložia do súvislého úseku pamäte.

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 aspoň $\Omega(\log \frac{N}{B})$ pamäťových presunov, čo je rovnako ako binárne vyhľadávanie horšie v porovnaní s *cache-aware* B-stromom.

2.2.5 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.

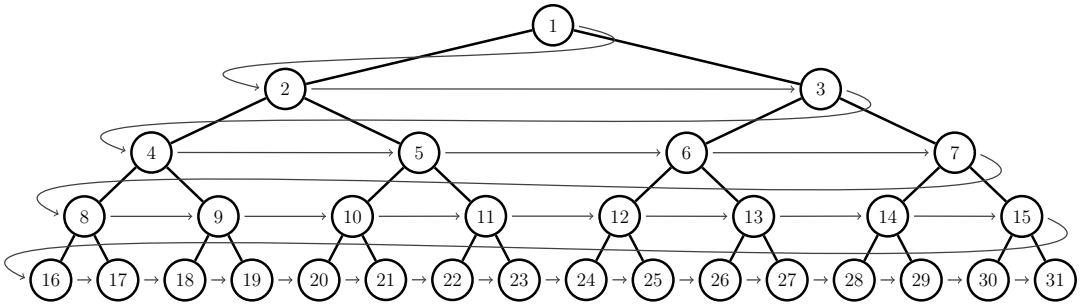
Možným riešením je takzvané *van Emde Boasovo usporiadanie* (*van Emde Boas layout*, nazvané podla *van Emde Boasových* stromov s podobnou myšlienkou), ktoré popísali Bender et al. [6, 7]. Toto usporiadanie vznikne rekurzívnym delením, ktorého schéma je na obrázku 2.1 a príklad takto uloženého stromu na obrázku 2.2(b).

Uvažujme úplný binárny strom výšky h. Ak $h = 1$ tak máme iba jeden vrchol v a výstupom usporiadania bude poradie (v).

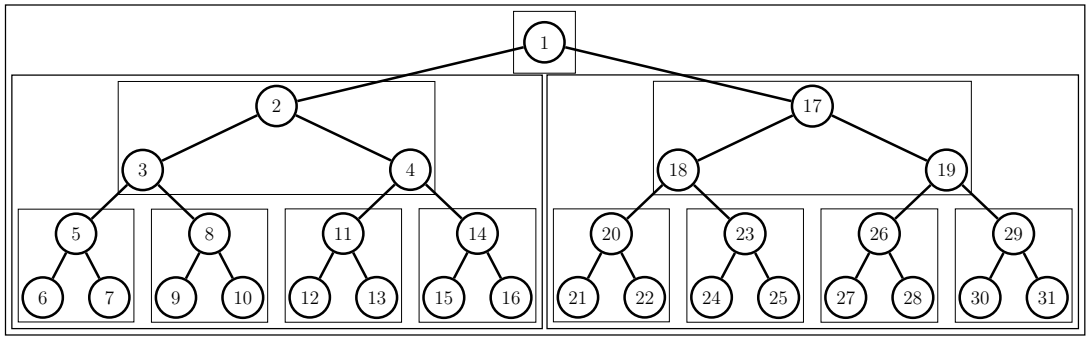
Pre $h > 1$ označme $m = \max_{i \in \mathbb{N}}\{2^i \mid 2^i < h\}$, teda najväčšiu mocninu dvoch menšiu ako h. Rozdelíme vstupný strom na podstrom τ_0 výšky $h - m$, ktorého koreňom je koreň pôvodného stromu. Zostanú nám podstromy $\tau_1, \ldots, \tau_k$ výšky m (obrázok 2.1). Korene týchto podstromov sú potomkovia listov τ_0 a ich z 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 $(\tau_0, \tau_1, \ldots, \tau_k)$.

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.



(a) Klasické usporiadanie



(b) *van Emde Boasovo* usporiadanie

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

Iným riešením je vnútorne pracovať s *BFS* usporiadaním a prechádzať medzi pozíciami funkciami uvedenými v algoritme 2.3. 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 $\mathcal{O}(\log \log N)$ [15].

Vyhľadávanie

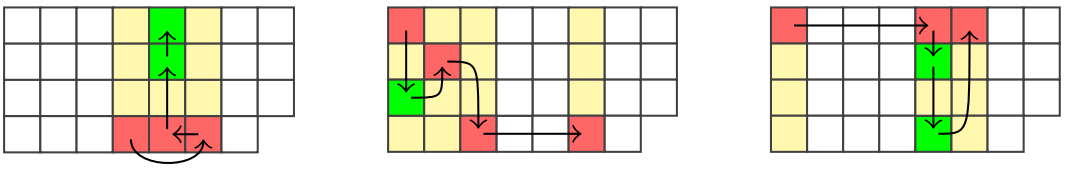
Pri analýze vyhľadávania sa pozrime na také podstromy predošlého delenia, že ich veľkosť je $\Theta(B)$. Ďalšie delenie a preusporiadanie je už zbytočné, no to *cache-oblivious* algoritmus nemá ako vedieť. Keďže ale po rekurzívnom volaní získame len iné usporiadanie, ktoré uložíme v súvislom úseku pamäte,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

2.2.6 Rozdiely v prístupe ku pamäti

Rozdiely medzi binárnym vyhľadávaním a statickými vyhľadávacími stromami v *BFS* a *van Emde Boas* usporiadaniach vizuálne znázorňujú obrázky 2.4 a 2.5 na strane 18. Pozrime sa však najskôr na menší obrázok 2.3, na ktorom vysvetlíme princíp tejto vizualizácie.

(a) Binárne vyhľadávanie (b) *BFS* strom (c) *van Emde Boasov* strom



Obrázok 2.3: Porovnávanie prístupov ku pamäti v rôznych štruktúrach pri vyhľadávaní hodnoty 17 medzi 31 prvkami

Jednotlivé políčka sú položky poľa, pričom v pamäti sú usporiadané v poradí zhora nadol a zľava doprava. Samotné kľúče (vo vrcholoch stromov) sú však usporiadané inak a práve preto sú tieto prístupy inak efektívne. Vo všetkých prípadoch vyhľadávame prvok v množine $\{1, \ldots, 2^h - 1\}$, čo je počet vrcholov úplného binárneho stromu výšky h. Farby označujú stav políčka v simulovanej *cache* s veľkosťou bloku $B = 4$. Bloky sú zarovnané tak, že jeden stĺpec predstavuje jeden blok a teda sa pri načítaní ľubovolného 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 2.3 (výška stromu $h = 5$) šípky určujú poradie, v akom vyhľadávanie pristupovalo k jednotlivým prvkom. Vo veľkých obrázkoch 2.4 a 2.5 (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 troma 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.

2.2.7 Dynamická verzia vyhľadávacieho stromu

V časti 2.2.5 sme popísali štruktúru, ktorá dokáže vyhľadávať v usporiadanej statickej množine optimálne, rovnako ako *cache-aware* B-stromy. 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 odstraňovania, budeme potrebovať pomocnú dátovú štruktúru, ktorú popíšeme v následovnej sekcii.

2.3 Usporiadané pole

Problémom *údržby usporiadaného poľa* (z anglického *ordered-file maintenance*) budeme volať problém spočívajúci v udržiavaní zoradenej postupnosti N prvkov vo forme súvislého poľa veľkosti $\mathcal{O}(N)$. V tomto poli teda môžu byť *medzery* veľkosti najviac $\mathcal{O}(1)$, vďaka čomu bude načítanie K po sebe idúcich prvkov vyžadovať $\mathcal{O}(\frac{K}{B})$ pamäťových presunov. Do tejto postupnosti musí byť možné efektívne vkladať prvky na ľubovoľnú danú pozíciu a tiež odstraňovať existujúce prvky.

Dátovou štruktúrou, ktorá tento problém rieši efektívne je *štruktúra zhustenej pamäte* (*packed-memory structure*). Myšlienka tejto štruktúry spočíva v zostrojení binárneho stromu nad prvkami tohto poľa a udržiavaní *hustoty* – podielu plných a všetkých pozícií – pre každý vrchol v konštantných hraniciach. Tým zaručíme, že štruktúra nebude ani príliš plná, čo by spôsobovalo pri vkladaní nutnosť presúvať veľké množstvo existujúcich prvkov na uvoľnenie miesta, a ani príliš prázdna, čo by spôsobovalo pomalé prechádzanie kvôli veľkým medzerám.

2.3.1 Popis štruktúry

Budeme používať verziu z [6] s malými úpravami. Celá dátová štruktúra pozostáva z jedného poľa veľkosti $T = 2^h$. To (pomyselne) rozdelíme na *bloky* veľkosti $S = \Theta(\log N)$, tak aby $S = 2^\ell$. Počet blokov tak bude tiež mocnina dvoch.

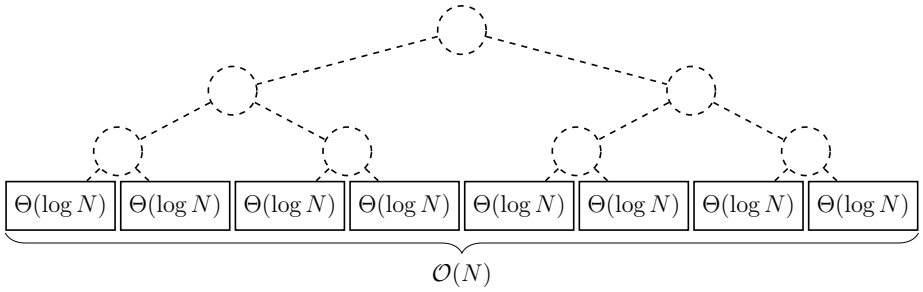
Nad týmito blokmi zostrojíme (imaginárny) úplný binárny strom (obrázok 2.6). *Hĺbkou* vrcholu označíme jeho vzdialenosť od koreňa, pričom koreň má hĺbku 0 a listy

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

Obrázok 2.4: Porovnávanie prístupov ku pamäti v rôznych štruktúrach pri vyhľadávaní hodnoty 243 medzi 511 prvkami

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

Obrázok 2.5: Porovnávanie prístupov ku pamäti v rôznych štruktúrach pri vyhľadávaní hodnoty 427 medzi 511 prvkami



Obrázok 2.6: Usporiadané pole veľkosti N. Strom nad blokmi je len imaginárny a nezostrojuje sa v pamäti.

majú hĺbku $d = h - \ell$. Na popis operácií budeme potrebovať definície *hustoty* a *hraníc hustoty*, ktoré uvedieme v nasledovnej časti.

2.3.2 Definícia hustoty

Kapacitou vrcholu v, $c(v)$, označíme počet položiek (aj prázdnych) poľa patriacich do blokov v podstrome, ktorého koreň je tento vrchol. 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 $o(v)$.

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:

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

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:

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

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

2.3.3 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 $S = \Theta(\log N)$ 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 lineárnych prechodov k okrajom poľa. Počas tohto prechodu si udržiavame počítadlá neprázdnych a všetkých pozícii, ktoré inicializujeme podľa tohto bloku. Ďalej postupujeme následovne.

V prípade, že je hustota (podiel počítadiel) podstromu reprezentujúceho práve prejdený interval mimo hraníc hustoty, posunieme sa v pomyselnom strome nahor. To dosiahneme prechodom doľava alebo doprava o práve toľko pozícií ako je veľkosť už prejdeného intervalu. Tento postup ukončíme v momente, keď hustota dosiahne požadované hranice.

Po nájdení takéhoto vrcholu v hraniciach rovnomerne rozdelíme všetky hodnoty v blokoch prislúchajúcich danému podstromu. Keďže intervaly pre hranice sa smerom k 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 prvky 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 vkladaní 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 vrcholu 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 podstrome v hraniciach hustoty, avšak nie len v svojich, ale keďže sme prerozdelili podstrom vrcholu u, tak 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 až keď sa potomok v ocitne znovu mimo hraníc hustoty. Amortizovaná 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 d úplného binárneho stromu 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 amortizovaná veľkosť upraveného intervalu $\mathcal{O}(\log^2 T)$.

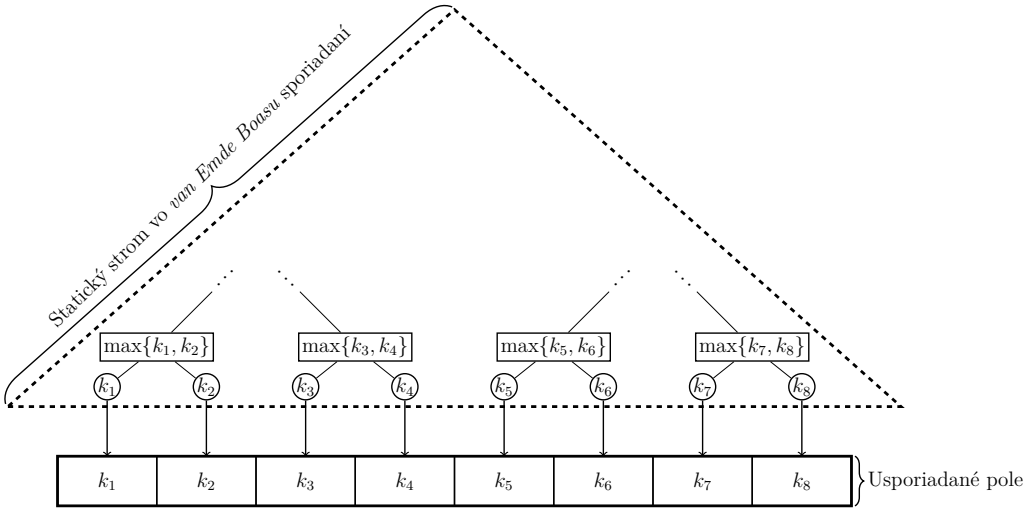
V prípade, že je aj koreň mimo hraníc hustoty, je potrebné vytvoriť pole dvojnásobnej veľkosti. Do spôsobí zmenu intervalu veľkosti $\mathcal{O}(N)$, avšak najbližších N operácií vkladania môže prebehnúť bez nutnosti zväčšovať pole. Preto je amortizovaná veľkosť týmto spôsobom upraveného intervalu iba $\mathcal{O}(1)$ a podobne aj pre odstraňovanie.

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

Existujú tiež verzie tejto štruktúry [9], ktoré dosahujú rovnaký počet pamäťových operácií nie v amortizovanom, ale najhoršom prípade. V tejto práci sa im však nebudeme venovať.

2.4 Dynamický B-strom

V tejto sekcií popíšeme dynamickú verziu *cache-oblivious* vyhľadávacieho stromu. Prvá verzia tejto štruktúry, ktorú navrhli Bender, Demaine et al. v [6] a neskôr podrobne popísali v [7], bola však značne komplikovaná a preto uvedieme zjednodušenú verziu. Tú navrhli Bender, Duan et al. v [8] a dosahuje rovnaké výsledky, ale jej popis a analýza sú podstatne jednoduchšie.



Obrázok 2.7: Dynamický strom, ktorý vznikne spojením usporiadaného poľa a statického stromu vo *van Emde Boasovom* usporiadaní. Šipky znázorňujú spárovanie listov a položiek pola.

2.4.1 *Cache-aware* riešenie

V prípade *cache-aware* modelu môžeme na riešenie tohto problému opäť použiť B-strom s vetvením $\Theta(B)$ ako v časti 2.2.2. Rovnako ako pre vyhľadávanie bude na vkladanie potrebných $\mathcal{O}(\log_B N)$ pamäťových presunov.

2.4.2 Popis *cache-oblivious* štruktúry

Túto dátovú štruktúru vytvoríme zložením predchádzajúcich dvoch – statického stromu (2.2.5) 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 úplný binárny statický vyhľadávací strom uložený vo *van Emde Boasovom* usporiadaní. Listy tohto stromu budú naviazané na prvky usporiadaného poľa (pozri obrázok 2.7).

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 poľa. Medzeru reprezentujeme hodnotou, ktorá je pri použitom usporiadaní najmenšia (v prípade číselných kľúčov $-\infty$). 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 prvok 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, našli sme požadovanú položku a v opačnom prípade 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 2.2.5, vykoná $\mathcal{O}(\log_B N)$ pamäťových presunov.

2.4.4 Vkladanie

Zaujímavejšou operáciou je vkladanie, ktoré v pôvodnom statickom strome nebolo (efektívne) 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.

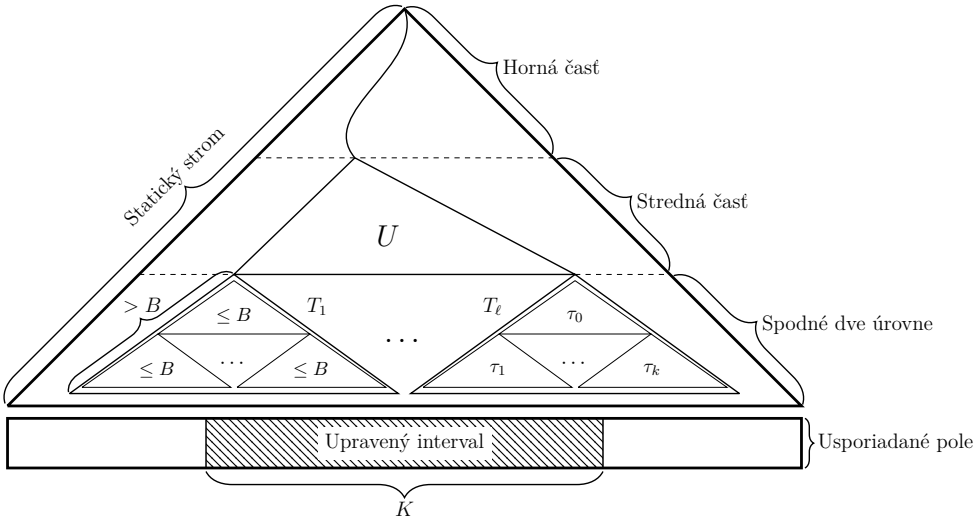
Vloženie do usporiadaného pola zmení súvislý interval I. Následne bude potrebné aktualizovať kľúče v statickom strome, aby opäť obsahovali maximum zo svojich synov. Túto aktualizáciu dosiahneme takzvaným *post-order* prechodom, kedy sa vrchol aktualizuje až po tom, čo boli aktualizovaní jeho synovia. Tým máme zaručené, že po aktualizácii vrchol obsahuje výslednú, korektnú hodnotu. Implementácia takéhoto prechodu je naznačená v algoritme 2.4.

Algoritmus 2.4 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.podstrom ∩ I = ∅ then          ▷ pokiaľ sa zmena tohto vrcholu
3:         return                          ▷ určite nedotkla, tak ho môžeme preskočiť

4:     if v je list then
5:         v.kľuč ← hodnota z usporiadaného poľa
6:     else
7:         UPDATE(v.ľavý)                  ▷ najskôr aktualizujeme ľavého
8:         UPDATE(v.pravý)                 ▷ a pravého syna

9:         v.kľúč ← max(v.ľavý.kľúč, v.pravý.kľúč)   ▷ až potom tento vrchol
```

2.4.5 Analýza vkladania

Túto analýzu rozdelíme na tri časti (obrázok 2.8), v ktorých sa postupne pozrieme na rôzne úrovne v statickom strome, ktoré treba po vložení do usporiadaného pola aktualizovať. Budeme predpokladať, že veľkosť zmeneného intervalu I v usporiadanom



Obrázok 2.8: Rozdelenie statického stromu na tri časti pri analýze počtu pamäťových presunov.

poli je K. Pre jednotlivé úrovne spočítame počet potrebných pamäťových presunov a nakoniec ich sčítaním získame výslednú zložitosť operácie vkladania.

Spodné dve úrovne

Uvažujme najskôr ako v časti 2.2.5 také podstromy *van Emde Boasovho* delenia, ktorých veľkost je medzi $\sqrt{B}$ a B, a teda sa zmestia do najviac dvoch blokov v *cache*. Pozrime sa na spodné dve úrovne takýchto podstromov (na obrázku 2.8 reprezentované najmenšími trojuholníkmi). Listy spodnej úrovne budú listy celého statického stromu 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. Tieto podstromy vznikli rekurzívnym *van Emde Boasovým* delením z podstromu T veľkosti viac než B. Označme ich, rovnako, ako v časti 2.2.5, $\tau_0, \tau_1, \ldots, \tau_k$, pričom 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 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. Až potom sa vrátime do τ_0 a k τ_1 už viac 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 časť – najbližší spoločný predok

Označme ako $T_1, \ldots, T_\ell$ tie stromy, ktoré obsahujú aktualizované podstromy veľkosti najviac B na spodných dvoch úrovniach. Platí teda $\ell = \mathcal{O}(\frac{K}{B})$. Označme U taký podstrom statického stromu, ktorého koreň je najbližší spoločný predok koreňov $T_1, \ldots, T_\ell$. Ak je tento koreň v hĺbke h tak U obsahuje všetky vrcholy hĺbky aspoň 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 $|U| = \mathcal{O}(\frac{K}{B})$. 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 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 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:	$\mathcal{O}(\frac{K}{B})$
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 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 interval 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

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

Oproti *cache-aware* B-stromom má pri vkladaní *cache-oblivious* dynamický strom popísaný v predchádzajúcej sekcii naviac $\mathcal{O}(\frac{\log^2 N}{B})$ pamäťových presunov. V prípade, že $B = \Theta(\log N \log\log N)$, bude tento člen zanedbateľný a dosiahneme rovnaký výsledok ako *cache-aware* B-stromy. Jednoduchou modifikáciou však môžeme tento člen môžeme odstrániť bez nutnosti tohto predpokladu.

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 $\mathcal{O}(\log_B \frac{N}{\log N}) = \mathcal{O}(\log_B N)$ 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: $\mathcal{O}(\log_B N)$.

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 dostávame amortizovanú zložitosť vkladania a odstraňovania:

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

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.

2.5.3 Prechod

Nevýhodou tejto modifikácie je zhoršenie počtu pamäťových presunov potrebných na prechod po sebe idúcich kľúčov. V prípade pôvodného *cache-oblivious* B-stromu sú kľúče uložené v súvislom úseku pamäte – usporiadanom poli veľkosti $\mathcal{O}(N)$. Je teda možné načítať K po sebe idúcich kľúčov s použitím $\mathcal{O}(\frac{K}{B})$ pamäťových presunov.

Po tejto zmene však v jednom bloku B-stromu načítame najviac $\mathcal{O}(\log N)$ kľúčov. Ďalšie sú uložené v nasledovnom bloku, ktorý však nemusí byť v pamäti na nasledovnej pozícii. Preto v prípade, keď $B = \Omega(\log N)$ a vieme jeden takýto blok načítať v $\mathcal{O}(1)$ pamäťových presunoch, môže byť potrebných až $\mathcal{O}(\frac{K}{\log N})$ pamäťových presunov – toľko, koľko je blokov. V prípade, keď $B = \mathcal{O}(\log N)$ načítame v jednom pamäťovom presune najviac B kľúčov a bude teda potrebných $\mathcal{O}(\frac{K}{B})$ pamäťových presunov ako v prípade pôvodného *cache-oblivious* B-stromu.

Spolu teda dostávame, že na prechod K po sebe idúcich kľúčov potrebujeme

$$\mathcal{O}(\frac{K}{\min\{B, \log N\}})$$

pamäťových presunov.

KAPITOLA 3

Vizualizácie

An algorithm must be seen
to be believed.

The Art of Computer Programming
Donald E. Knuth

Cieľom 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 2.2.5) v statickom binárnom vyhľadávacom strome, usporiadané pole (2.3) a dynamický B-strom (2.4). Súčasťou je tiež simulácia *cache* (sekcia 1.1) s možnosťou voľby parametrov B a M – veľkosť bloku a celková veľkosť.

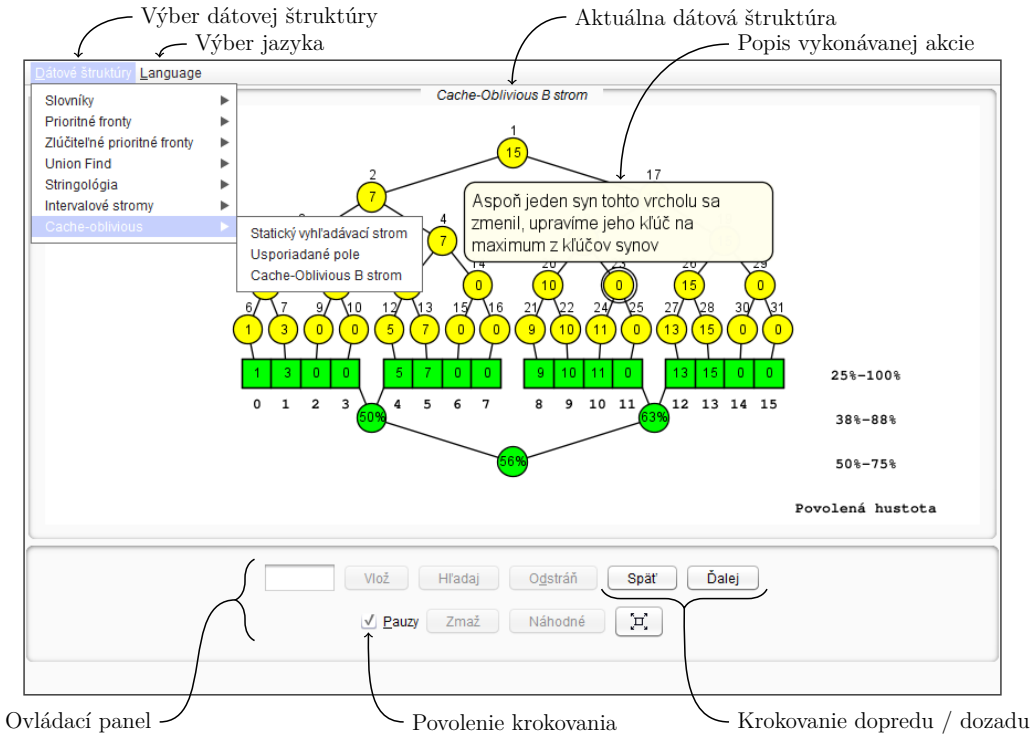
3.1 Gnarley trees

Tieto vizualizácie sú implementované ako rozšírenie programu *Gnarley trees*, ktorý vznikol ako súčasť bakalárskej práca Jakuba Kováča [17]. Tento nástroj na vizualizáciu (prevažne stromových) dátových štruktúr bol následne v bakalárskych prácach [16, 18, 23] a ročníkových projektoch rozšírený o mnohé ďalšie dátové štruktúry a v súčastnosti podporuje desiatky štruktúr, ako napríklad červeno-čierne, sufixové a intervalové stromy, *union-find*, haldy a mnohé ďalšie.

3.1.1 Spustenie

Súčasťou práce je priložené CD obsahujúce kompletný zdrojový kód [22] tohto programu a tiež spustiteľnú verziu [21]. Aplikácia požaduje *JVM*[1] a je ju možné spustiť pomocou

[1] *Java Virtual Machine*, dostupné na `https://www.java.com/en/download/`



Obrázok 3.1: Užívateľské rozhranie počas operácie vkladania kľúča 10 do dynamického *cache-oblivious* B-stromu (sekcia 2.4).

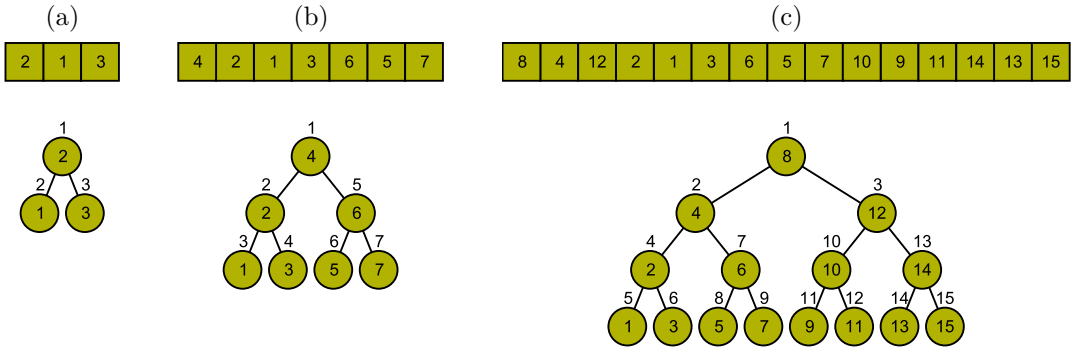
súborov `start.sh` prípadne `start.bat`, podla operačného systému. Dostupné sú tiež individuálne vizualizácie vo forme *Java appletov*, ktoré je možné spustiť otvorením súboru `index.html` v preferovanom internetovom prehliadači.

Obsah tejto CD prílohy je tiež dostupný na `http://school.lacop.net/thesis/cd-contents.zip`.

3.1.2 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 vracať 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.



Obrázok 3.2: Statické stromy rôznych veľkostí (výšok) vo *van Emde Boasovom* usporiadaní.

3.1.3 Prehľad programu

Program sa skladá z troch hlavných častí (obrázok 3.1). 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 akcií (ak je povolené krokovanie) a ďalšie vizualizačné prvky ako šípky alebo význačné vrcholy.

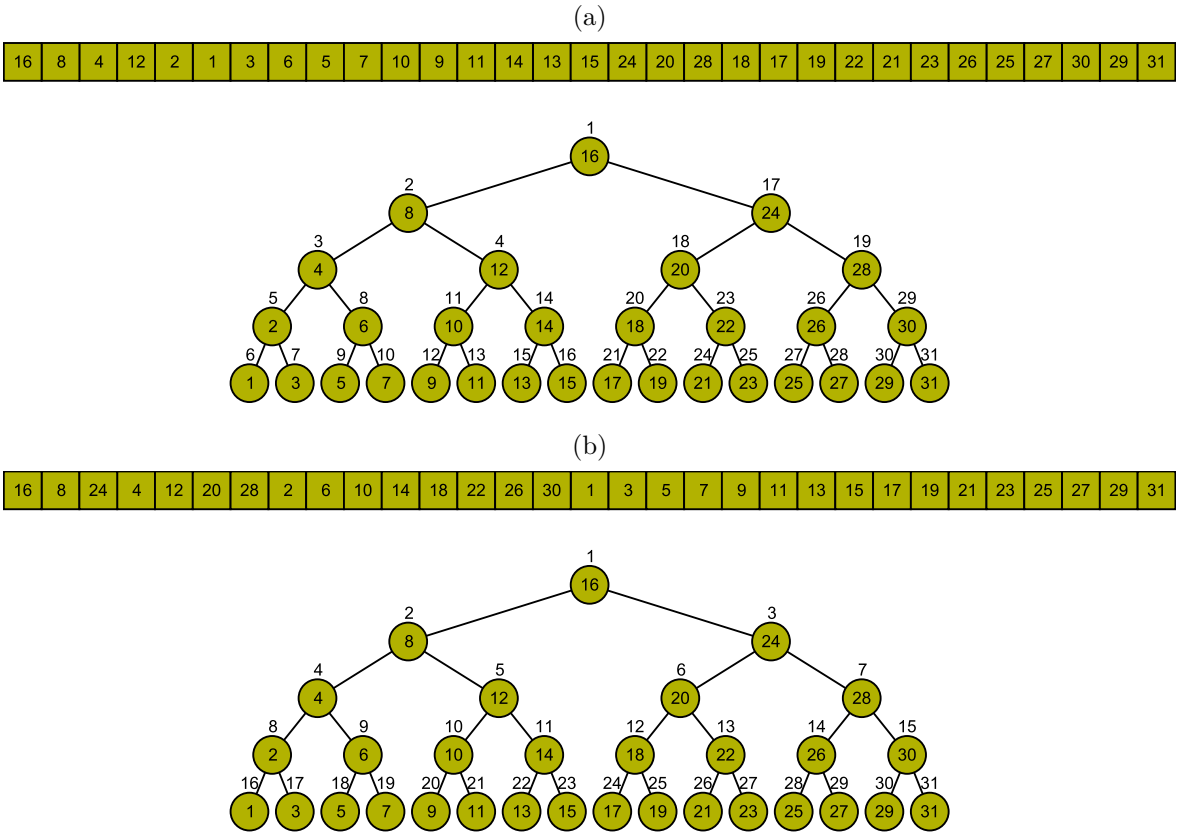
Podrobnejší popis užívateľského rozhrania a návod na používanie sa nachádza v bakalárskej práci Jakuba Kováča [17] v siedmej kapitole.

3.2 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 preferencii – ukážky stromov rôznych veľkostí sú na obrázku 3.2.

Čísla vo vnútorných vrcholoch sú kľúče, čísla nad vrcholom určujú jeho pozíciu v pamäti. Uloženie tohto stromu v pamäti, vo forme súvislého poľa, je zobrazené nad koreňom stromu. Vnútorné čísla poľa sú opäť kľúče, pričom sú zoradené podľa svojich pozícii zľava (pozícia 1) doprava.

Medzi uložením vo *van Emde Boasovom* usporiadaní a klasickom *BFS* usporiadaní (ako v časti 2.2.4) je možné prepínať. Zmenia sa pritom čísla udávajúce pozície



Obrázok 3.3: 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 poľom nad koreňom.

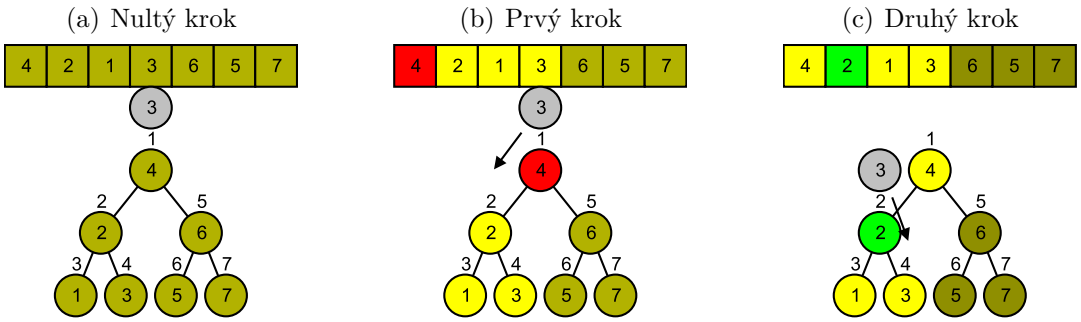
vrcholov v pamäti a ich poradie v poli nad stromom. Rozdiel medzi týmito dvoma usporiadaniami vidieť na obrázku 3.3.

3.2.1 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 1.2.1.

Táto simulácia zároveň počíta počet pamäťových prístupov k vrcholom pri vyhľadávaní. Prvé počítadlo udáva počet všetkých prístupov a správa sa rovnako pri *van Emde Boasovom* aj *BFS* usporiadaní a tiež pre ľubovolné parametre *cache*. Druhé udáva počet prístupov, pri ktorých bolo potrebné načítanie z disku (*cache miss*) a pri zmene parametrov či usporiadania sa bude správať inak. Cieľom *cache-oblivious* algoritmov je minimalizovať práve počet takých presunov, ktoré počíta toto druhé počítadlo.

Na vizualizáciu 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



Obrázok 3.4: 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.

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 krokovaní vyhľadávania je pri prístupe k vrcholu tiež použitá zelená alebo červená farba na jeho dočasné zafarbenie (podobne ako v časti 2.2.6) podľa toho, či sa v danom momente v *cache* nachádzal (*cache hit*) alebo nie (*cache miss*). Ukážky tohto zafarbovania sú na obrázkoch 3.4 a 3.5.

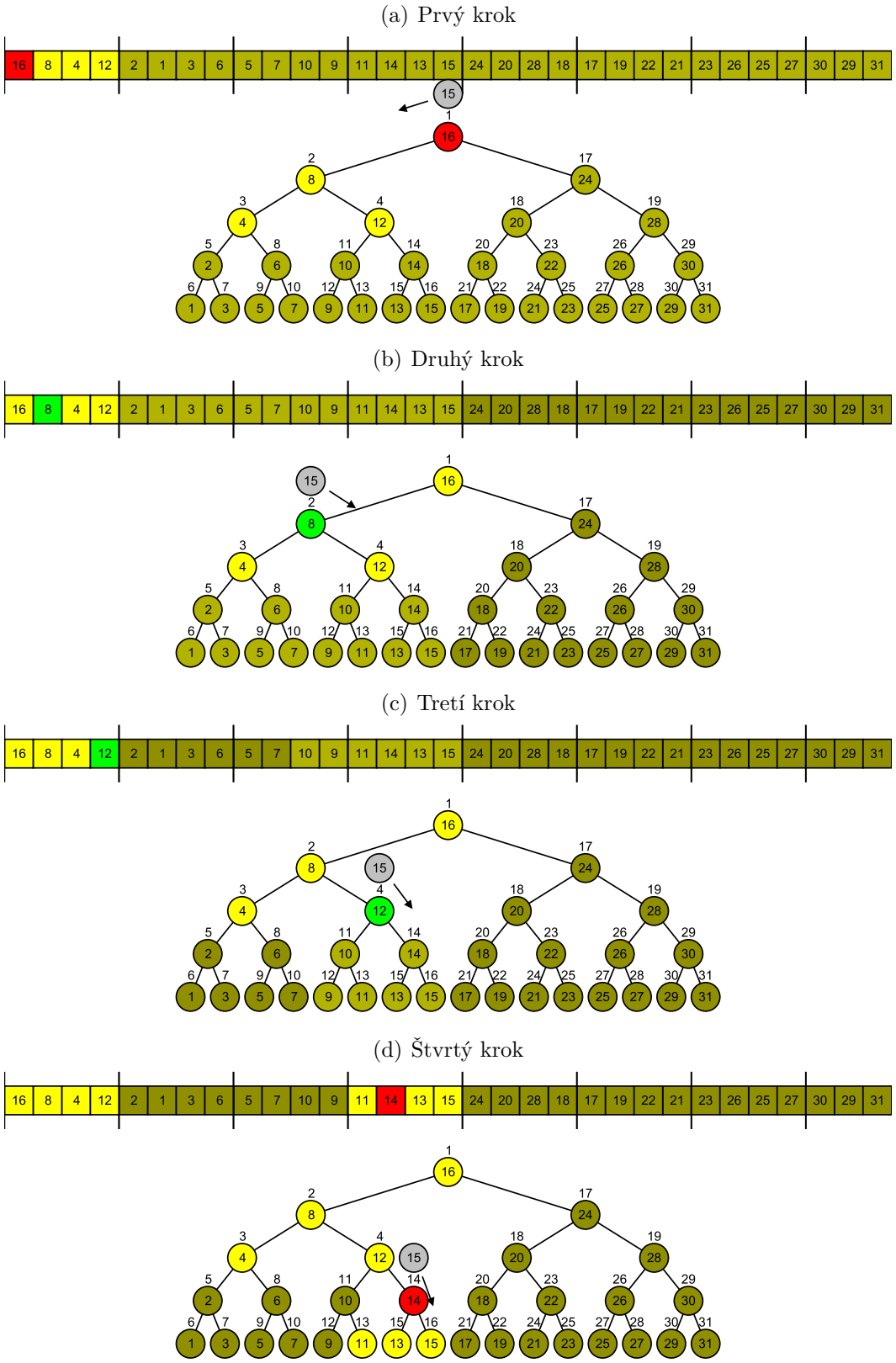
3.3 Usporiadané pole

Ďalšou implementovanou vizualizáciou je usporiadané pole (obrázok 3.6), ktoré bolo popísané v časti 2.3. 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 podstrome. Farby (zelená alebo červená) vrcholov indikujú, či sa hustota daného vrcholu nachádza v hraniciach.

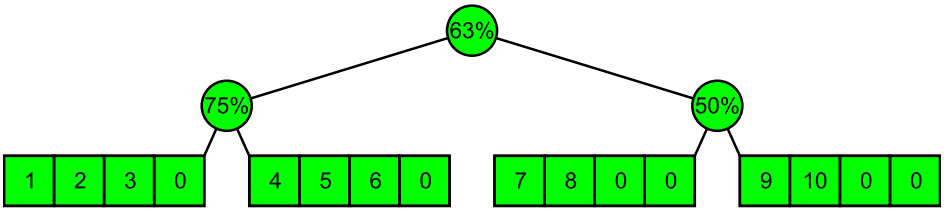
Táto vizualizácia podporuje vkladanie hodnoty na ľubovolnú pozíciu v poli. V prípade, že sa pole preplní, automaticky sa vytvorí nové, dvojnásobne väčšie. Po vložení hodnoty je tiež vyznačený interval, ktorý sa zmenil.

3.4 Dynamický strom

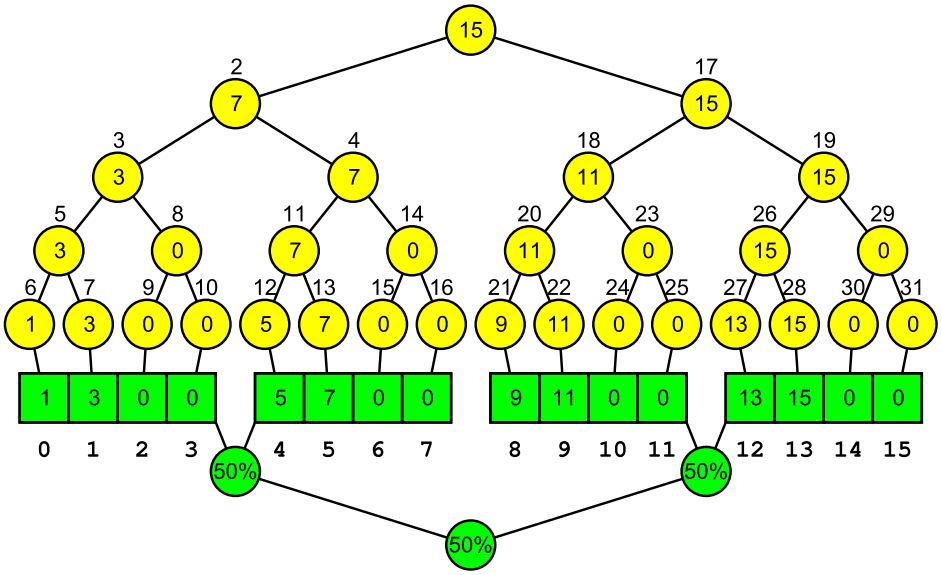
Vizualizácia dynamického stromu (časť 2.4) vznikla spojením predchádzajúcich dvoch vizualizácií (obrázok 3.7). Horná časť je statický strom vo *van Emde Boasovom* uspo-



Obrázok 3.5: Simulácia *cache* počas vyhľadávania kľúča 15. Pred prvým prístupom je cache prázdna. Po načítaní bloku, ktorý obsahuje kľúče $\{16, 8, 4, 12\}$ v prvom kroku, nedochádza v druhom a treťom kroku k pamäťovým presunom. Najbližší kľúč mimo *cache* (*cache miss*) je až v štvrtom kroku.



Obrázok 3.6: Usporiadané pole obsahujúce hodnoty 1 až 10. Všetky vrcholy majú hustotu v hraniach hustoty. Hodnoty 0 reprezentujú prázdne pozície.



Obrázok 3.7: Vizualizácia dynamického *cache-oblivious* B-stromu.

riadaní (obrázok 3.2), dolná je usporiadané pole (obrázok 3.6), 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 poľa rovnako, ako pri jeho samostatnej vizualizácii. Následne sa aktualizujú kľúče statického stromu. V prípade zdvojnásobenia veľkosti statického poľa sa vytvorí nový, väčší statický strom.

Záver

V tejto práci sme uviedli problematiku návrhu a analýzy algoritmov a dátových štruktúr v pamäťových modeloch akými sú *cache-aware* a *cache-oblivious* modely, v ktorých na rozdiel od klasického *RAM* modelu nepovažujeme každý prístup k dátam za operáciu vykonateľnú v konštantnom čase. Prínos týchto modelov sme podložili prístupovými dobami k rôznym pamätiam v reálnych systémoch, ktoré sa líšia o niekoľko rádov.

Ďalej sme sa zamerali na *cache-oblivious* model, ktorý je podstatne novší a menej známy ako *cache-aware* model. Jeho hlavným prínosom je automatická prenositeľnosť medzi systémami s rôznymi parametrami pamäte, automatické fungovanie na každej úrovni pamäťovej hierarchie a jednoduchšia implementácia. Uviedli sme niekoľko *cache-oblivious* algoritmov a dátových štruktúr, ktoré sú rovnako alebo porovnateľne efektívne ako ich *cache-aware* ekvivalenty.

Hlavným výsledkom práce sú však vizualizácie vybraných *cache-oblivious* dátových štruktúr, ktoré boli implementované ako rozšírenie vizualizačného programu *Gnarley trees*. Tie môžu slúžiť ako učebný nástroj pre zjednodušenie a urýchlenie porozumenia týmto štruktúram.

V budúcnosti by sme chceli tieto vizualizácie rozšíriť o ďalšie *cache-oblivious* dátové štruktúry, ako napríklad prioritné fronty, stromy s textovými kľúčmi, spájané zoznamy a tiež o *cache-oblivious* triediace algoritmy.