Zadanie 2 - Analýza Dátových Štruktúr

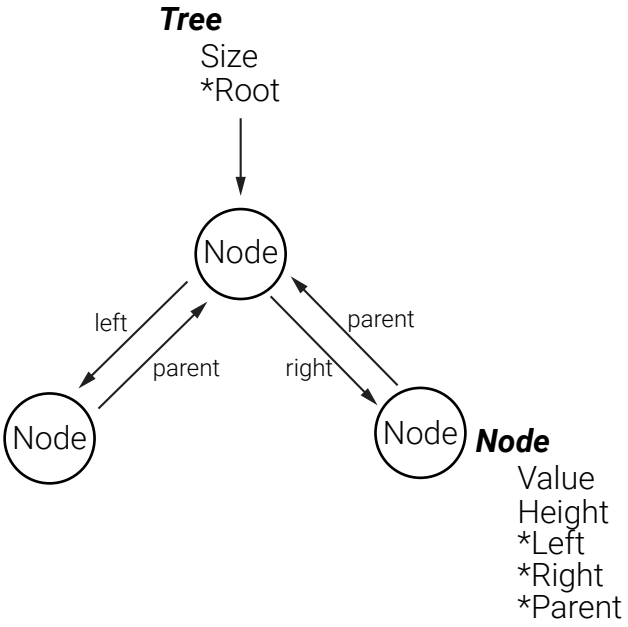
Dátové Štruktúry a Algoritmy

Dávid Silady

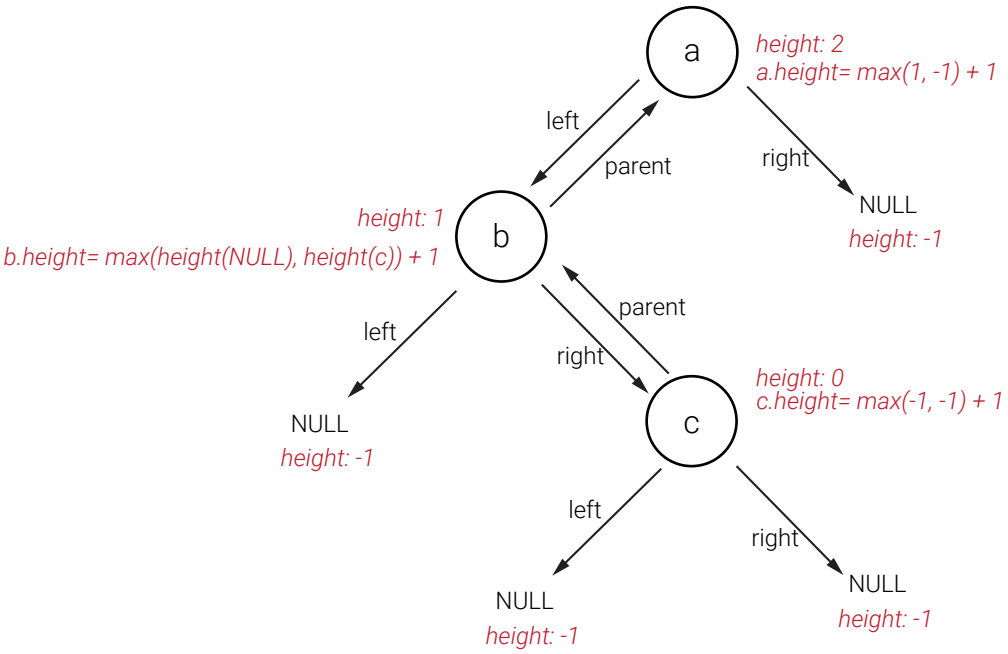
ZS2019/2020

1. Vlastná implementácia AVL stromu

Diagram

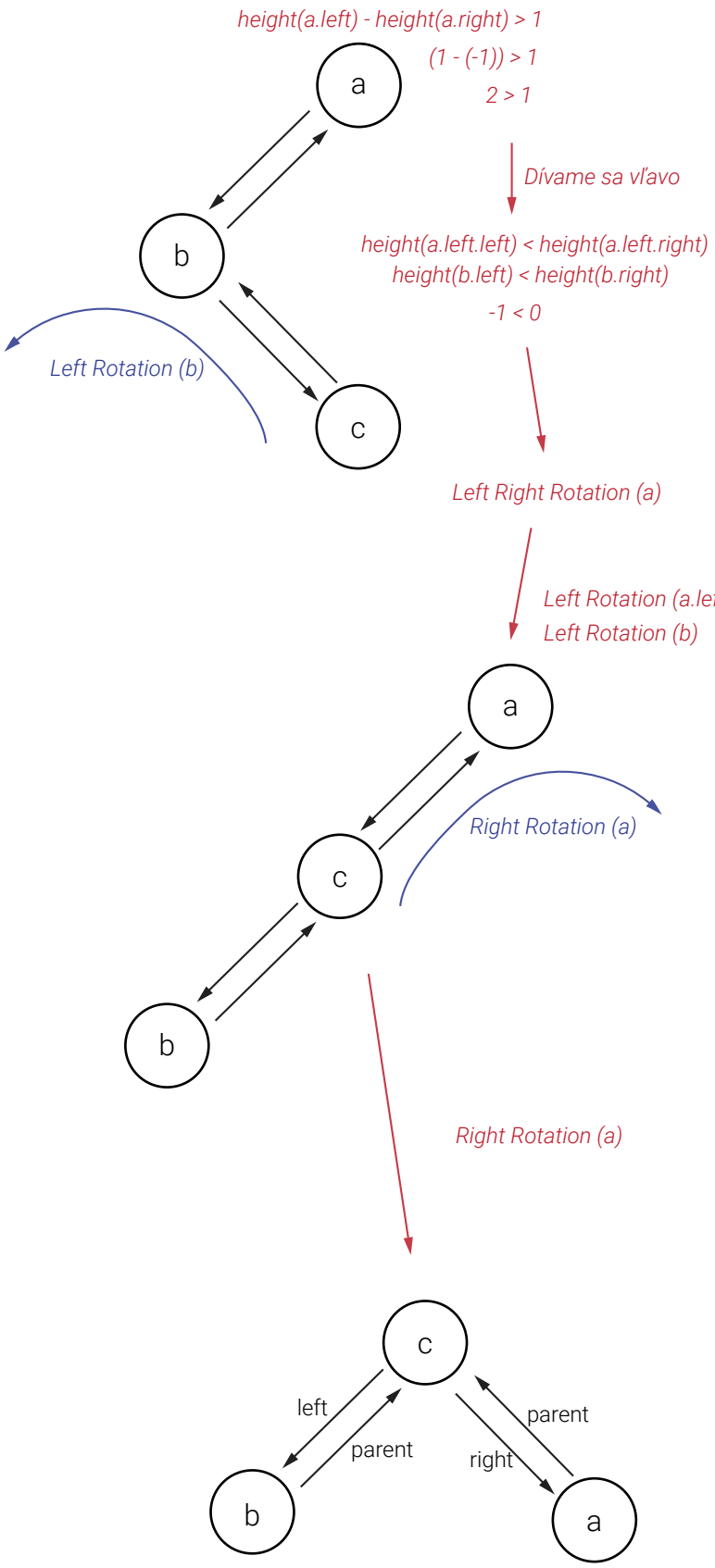


Výpočet výšky



height= max(height(left), height(right)) + 1
height(NULL) = -1

Vyvažovanie



Ľavé a pravé rotácie funguju analogicky rovnako.

Výška sa zisťuje vždy pri vkladaní na konci rekurzie. Výška sa tiež obnovuje pri rotáciách[0]. Ak by sme výšku získavali vždy rekurzívne, vkladanie by sa spomalilo na mieru, kde je nepoužiteľné pri veľkosti väčšej cez desať tisíc.

K rotáciám dochádza ak je rozdieľ výšok väčší ako 2 alebo menší ako -2 a podľa toho zistí, či rotovať v pravom podstrome alebo ľavom.
Následne sa pozrie do správneho podstromu a zistí, ktorý z jeho uzol má väčšiu výšku. Podľa toho vykoná buď jednoduchú alebo dvojitú rotáciu.

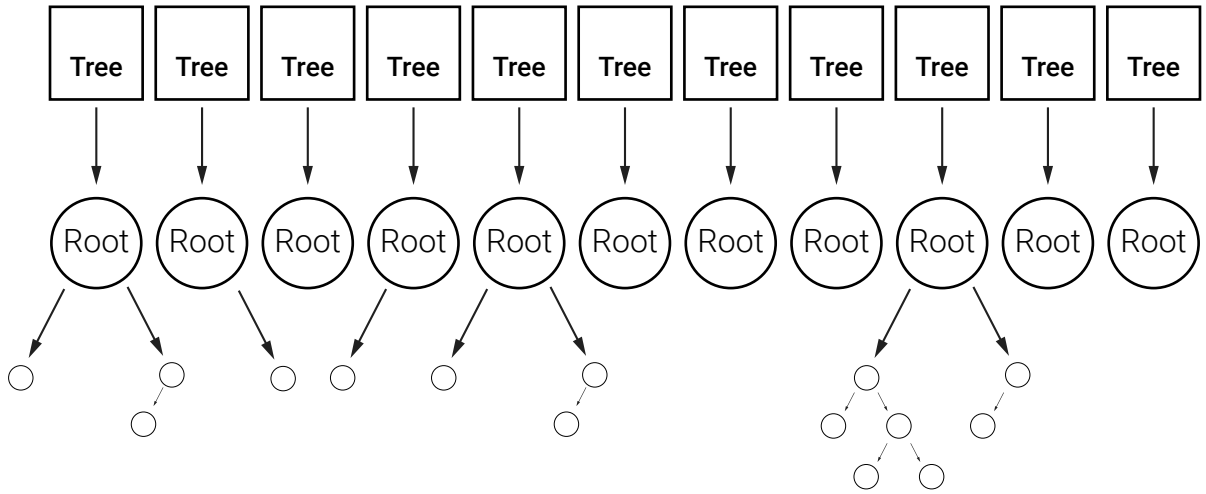
Ukazovateľ na rodiča je tu užitočný najmä pri pripájaní podstromov[1] najmä pri jednoduchých rotáciách, kde si ho do funkcie nemusíme vkladať a zisťovať inými spôsobmi. Tiež sa nám takto nestratí root. A pomôže pri debugovaní.

```
Node *left_right_rotation(Node *top) {
   top->left = left_rotation(top->left);
   return right_rotation(top);
}
```

```
Node *left_rotation(Node *top) {
   Node *parent = top->parent;
   Node *new_top = top->right;
   top->right= new_top->left;

   if (new_top->left != NULL) {
      new_top->left->parent = top;
   }

   new_top->left = top;
   top->parent = new_top;
   new_top->parent = parent;

   if (parent != NULL) {   [1]
      if (parent->left == top) {
         parent->left = new_top;
      } else {
         parent->right = new_top;
      }
   }

   update_height(new_top);   [0]
   return new_top;
}
```

2. Vlastná implementácia hashovacej tabuľky

Diagram



Vo vlastnej implementácii stromu sa kľúč rovná i vkladanej hodnote, čo znamená, že môžme miesto zoznamov využiť už zostrojený AVL strom.

Celá hash tabuľka sa teda skladá z ukazovateľov na jednotlivé stromy. Tie ukazujú na svoje korene a vkladá sa do nich teda veľmi jednoducho:

Hash funkcia vykoná jednoduché modulo veľkosťou tabuľky, čím zistíme, do ktorého stromu budeme vkladať.
Následne do stromu v indexe danom našou jednoduchou hash fukniou vyvoláme funkciu insert( ) z nášho, už implementovaného, stromu.

Znamená to, že zložitoť vyhľadávania je pri najhoršom možnom vstupe iba O(log(n)), nie O(n).

Vkladanie je na druhú stranu pomalšie, keďže nevkladáme na prvé miesto ako by to bolo pri spájanom zozname.

Zväčšovanie tabuľky môže byť teda menej časté, napríklad pri faktore zaplnenia 5.0, na rozdiel od lineárnej (0.6) alebo klasickému zreťazeniu (1.0 - 1.5). Pri zväčšení jednoducho prejde každým stromom a každý prvok pridá do novej tabuľky stromov.

Do stromu zasiahneme iba pri zväčšovaní, kde musíme manuálne prejsť cez každý uzol.[2]

O zvyšnú optimalizáciu sa stará strom.

Hash
size
num_elements
load factor
Tree **table

```
void hash_refill(Hash *hash, Node *node) {
   if (node == NULL)
      return;

   hash_refill(hash, node->left); [2]
   hash_refill(hash, node->right);

   hash_add(hash, node->value);
   free(node);
}

void resize(Hash *hash, int new_size) {
   Tree **old_table = hash->table;
   int old_size = hash->size;

   hash->size = new_size;
   hash->table = new_hash_table(hash->size);

   hash->loadFactor = 0;
   for (int i = 0; i < old_size; ++i) {
      hash_refill(hash, old_table[i]->root);
   }

   free(old_table);
}
```

3. Prevzatá implementácia RB stromu

https://github.com/prasanthmadhavan/Red-Black-Tree

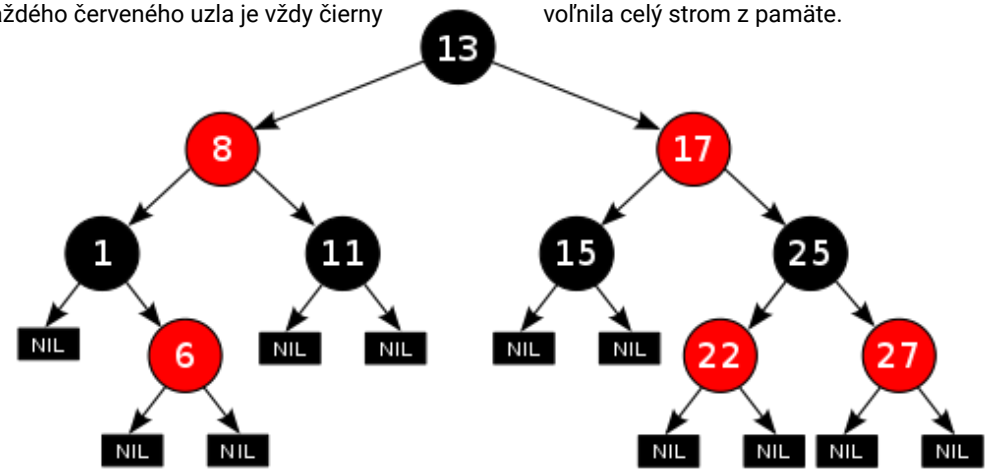
Na rozdieľ od vlastného AVL stromu, prevzatý je generický, čo znamenalo, že som musel dodefinovať funkciu na porovnávanie.
Použil som v šak funkciu z príkladu používania daného stromu.

Rovnako ako v mojom strome, i tu sú použíté dve štruktúry pre strom a uzoľ ako i tentoraz nutný ukazovateľ na rodiča.

Koreň stromu je vždy čierny.
Listy nemajú uloženú žiadnu hodnotu, sú reprezentované cez NULL, tiež sú vždy považované za čierne.
Každý červený uzol má dve deti, obe čierne.
Rodič každého červeného uzla je vždy čierny

Táto implementácia má rýchle vkladanie do veľkosti vstupu 10 000. Neskôr sa už značne spomalí.
Kvôli neprehľadnosti kódu som však nebol schopný identifikovať, kde pri vkladaní je vyvolaná zbytočne náročná funkcia.
Je možné, že problém je vo využití iteratívneho a nie rekurzívneho algoritmu. Rekurzívne je spravené prefarbovanie stromu - funkcia má štyri vrstvy, kde z tretej môže preskočiť do prvej, čo môže byť dôvodom spomalenia celého vkladania.
Iteratívny je však využitý i pri vyhľadávaní, ktoré je ale rýchle.

Rovnako chýba funkcia destroy( ), ktorá by vymazala/uvoľnila celý strom z pamäte.

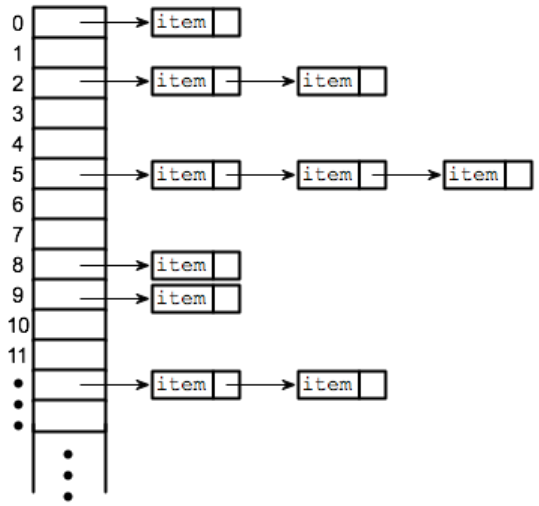


Zdroj: https://bit.ly/2pXBwOQ

4. Prevzatá implementácia hashovacej tabuľky

https://github.com/skeeto/hashtab

Implemntácia zreťazením, s lineárnymi spájanými zoznamami.
Tabuľka sa automaticky nezväčšuje, iba pri manuálnom zavolaní funkcie na zväčšenie.
Testy boli však vykonané bez tejto funkcie z dôvodu jednoduchosti implementácie do testov.
Vyhľadávanie bolo teda pomalé, asi rovnako ako vkladanie.
Vkladanie je pomalé pretože pri vkladaní do spájaného zozamu sa nevkladá na začiatok ale až na jeho koniec, čo znamená, že v každom prípade bola zložitosť vloženia O(n).



Zdroj: https://bit.ly/2qGmaOG

Testovanie

Testovanie pozostávalo z dvoch hlavných testov:

1. Pole náhodných hodnôt
2. Pole po sebe idúcich hodnôt

Čas sa vždy zapíše po každých 10 000 vložených prvkoch. Zapisuje sa do dvoch .log súborov, kde sa zapíše počet vložených hodnôt, čas od posledného zápisu a celkový čas od začiatku vkladania.
Nakoniec sa hodnoty všetkých opakovaných testov spriemerujú pre objektívnejší výsledok a zapíšu sa to test_average.log.

Každý test pozostáva z vloženia všetkých prvkov do danej štruktúry a následného vyhľadania všetkých prvkov v štruktúre.
Pri vyhľadávaní som pre nízke hodnoty zaznamenal vždy iba celkový čas.

```
Consecutive Array:
    0. inserted | Split time: 0.000000 | Total time elapsed: 0.000000
10000. inserted | Split time: 0.031000 | Total time elapsed: 0.031000
20000. inserted | Split time: 0.187000 | Total time elapsed: 0.218000
30000. inserted | Split time: 0.313000 | Total time elapsed: 0.531000
40000. inserted | Split time: 0.531000 | Total time elapsed: 1.062000
50000. inserted | Split time: 0.875000 | Total time elapsed: 1.937000
60000. inserted | Split time: 1.235000 | Total time elapsed: 3.172000
70000. inserted | Split time: 1.500000 | Total time elapsed: 4.672000
80000. inserted | Split time: 1.874000 | Total time elapsed: 6.546000
90000. inserted | Split time: 2.375000 | Total time elapsed: 8.921000
Time elapsed (Stolen Hash Insert): 11.796000
Time elapsed (Stolen Hash Find): 11.578000
```

Vkladanie náhodných	*10 000*	*40 000*	*70 000*	*100 000*
Cudzí Hash	0.036779	0.328828	0.798656	1.195500
Cudzí Strom	2.077980	12.975260	20.381549	24.805865
Môj Hash	0.013309	0.015719	0.031313	0.032813
Môj Strom	0.002000	0.017758	0.041188	0.058293
Môj BVS Strom	0.000937	0.013865		
Vyhľadávanie náhodných				
Cudzí Hash	0.000000	0.000000	0.000000	1.155113
Cudzí Strom	0.000000	0.000000	0.000000	0.031299
Môj Hash	0.000000	0.000000	0.000000	0.001063
Môj Strom	0.000000	0.000000	0.000000	0.021840
Môj BVS Strom	0.000000	0.028916		
Vkladanie po sebe idúcich				
Cudzí Hash	0.045062	1.152484	4.858732	10.977000
Cudzí Strom	2.080639	36.140816	122.189463	270.431275
Môj Hash	0.000059	0.015736	0.031578	0.050523
Môj Strom	0.000680	0.031330	0.048594	0.078623
Môj BVS Strom	0.330320	6.472504		
Vyhľadávanie po sebe idúcich				
Cudzí Hash	0.000000	0.000000	0.000000	12.328371
Cudzí Strom	0.000000	0.000000	0.000000	0.020658
Môj Hash	0.000000	0.000000	0.000000	0.003992
Môj Strom	0.000000	0.000000	0.000000	0.016000
Môj BVS Strom	0.000000	25.157953		

Záver

Cudzí hash

Ako môžme pozorovať, prebratý hash sa kvôli nezväčšovaniu výrazne spomalí pri väčších číslach, kde zároveň rastie aj jeho čas jedného korku (10 000 vložení).
Problém je jednoznačne vo vkladaní na koniec spájaného zoznamu.
Následné vyhľadávanie je zas spomalené najmä kvôli spájanému zoznamu, kde je najhorší prípad O(n).

Cudzí strom

Ako môžme pozorovať, prebratý hash sa kvôli nezväčšovaniu výrazne spomalí pri väčších číslach, kde zároveň rastie aj jeho čas jedného korku (10 000 vložení).
Problém je jednoznačne vo vkladaní na koniec spájaného zoznamu.
Tiež si môžme všimnúť výraný nárast pri vkladaní po sebe idúcich čísel, kde sa musí konštantne vyrovnávať a prefarbovať.
Následné vyhľadávanie je zas spomalené najmä kvôli spájanému zoznamu, kde je najhorší prípad O(n).

Vlastný hash

Vlastný hash je konštantne najrýchlejší, či sa to už týka vyhľadávania alebo vkladania. Je to spôsobené konkrétnou implementáciu na mieru (vkladali sme iba celé kladné čísla, čo nám umožnilo použiť jednoduchú hashovaciu fukciu a stromy namiesto spájaných zoznamov).

Vlastný strom

Pri vlastnom strome sa dá badať ľahký časový nárast pri vkladaní po sebe idúcich.
Veľká optimalizačná chyba bola pôvodne zisťovanie výšky rekurzívne bez ukladania hodnoty. Vtedy by dosahoval tisíc-násobky aktuálnych časov.

Vlastný nevyvážený strom

Pri vlastnom BVS je badateľné, že vkladanie náhodného pola je rýchlejšie ako pri vyváženom.
Následné vyhľadávanie je potom stále porovnateľne rýchle, i keď o trochu pomalšie.
Pri po sebe idúcom poli je však vidno dramatický časový nárast, kde čas vyhľadávania je niekoľkonásobne vyšší ako čas vloženia.
Tiež treba spomenúť, že kvôli priestorovej zložitosti nebolo možné uložiť do nevyváženého stromu viac ako 40 000 prvkov, inak by nám pretiekol stack.