Slovenská Technická Univerzita v Bratislave

Fakulta informatiky a informačných technológií

Vyhľadávanie v dynamických množinách

Zadanie 2

**Predmet:** Dátové štruktúry a algoritmy

**Obdobie:** Letný semester 2019/2020

**Cvičiaci:** Ing. Dominik Macko , PhD.

**Študent:** Bc. František Gič

# Obsah

[Obsah 2](#_Toc37619708)

[Binárne vyhľadávacie stromy 3](#_Toc37619709)

[AVL strom 3](#_Toc37619710)

[Implementácia 3](#_Toc37619711)

[Dátová štruktúra 3](#_Toc37619712)

[Vloženie prvku 4](#_Toc37619713)

[Hľadanie prvku 4](#_Toc37619714)

[Testovanie 5](#_Toc37619715)

[Test 1 5](#_Toc37619716)

[Test 2: Pravá rotácia (nevyváženosť LL) 7](#_Toc37619717)

[Test 3: Ľavá rotácia (nevyváženosť RR) 8](#_Toc37619718)

[Test 4: Rotácia LR 9](#_Toc37619719)

[Test 5: Rotácia RL 10](#_Toc37619720)

[Červeno-čierny strom 11](#_Toc37619721)

[Implementácia 11](#_Toc37619722)

[Porovnanie 11](#_Toc37619723)

[Hashovacie tabuľky 14](#_Toc37619724)

[Otvorené adresovanie - Kvadratické skúšanie 14](#_Toc37619725)

[Implementácia 14](#_Toc37619726)

[Dátová štruktúra 14](#_Toc37619727)

[Vloženie prvku 14](#_Toc37619728)

[Hľadanie prvku 15](#_Toc37619729)

[Zväčšenie tabuľky 15](#_Toc37619730)

[Test 16](#_Toc37619731)

[Reťazenie 17](#_Toc37619732)

[Implementácia 17](#_Toc37619733)

[Porovnanie 18](#_Toc37619734)

[Zhodnotenie 20](#_Toc37619735)

[Referencie 21](#_Toc37619736)

# Binárne vyhľadávacie stromy

V prípade binárnych vyhľadávacích stromov som si pre vlastnú implementáciu vybral algoritmus AVL na vyvažovanie binárneho vyhľadávacieho stromu a prevzatou implementáciou bola implementácia Červeno-čierneho stromu.

## AVL strom

AVL strom je algoritmus samo vyvažovania na princípe kontroly koeficientu vyváženia (balance) pod stromu. V prípade vyvažovania stromu sa po každom vložení prvku do stromu kontroluje každý prejdený uzol stromu a vypočítava sa neho koeficient vyváženia. Pokiaľ je absolútna hodnota koeficientu väčšia ako 1 (v správnej implementácií rovná 2 - nikdy nebude väčšia), tak nad tým daným uzlom musia prebehnúť rotácie.

Typy rotácie poznáme štyri, a aj keď sú pomenované rôznymi spôsobmi zakaždým ide o tie isté rotácie. V niektorých daných zdrojoch pomenúvajú autori rotácie podľa smeru hodinových ručičiek, niekde podľa strán (vľavo, vpravo), a v zdrojoch z ktorých som čerpal (Bari, 2020) boli uvedené podľa zdroja nevyváženia. Konkrétne sa jedná o LL, RR, LR a RL Rotácie. Daná rotácia sa teda nazýva podľa pôvodu nevyváženosti. V prípade LL rotácie ide teda o zdroj nevyváženosti v ľavom potomkovi ľavého podstromu, a preto nad aktuálnym uzlom vykonáme LL rotáciu (v mojej implementácií ide o funkciu rot\_ll).

### 

### Implementácia

#### Dátová štruktúra

V mojej implementácií som použil dátovú štruktúru uzlu s názvom Node, ktorá obsahuje dátovú časť - int data, výšku podstromu int height, ktorá sa počíta ako maximum z výšok pravého a ľavého podstromu, a smerníky na pravý a ľavý podstrom - struct Node\* left, struct Node\* right.

#### Vloženie prvku

Samotné vloženie prvku do binárneho prvku by sme mohli opísať nasledovným psuedokódom:

insert(node,data)

if node is null

create\_node()

return

if data < node.data

insert(node.left,data)

else

insert(node.right,data)

perform\_rotations(balance(node),balance(node.left),balance(node.right))

return node

Vloženie prvku do AVL stromu je podobné ako pri klasickom, neváženom strome, obohatené je však o dané rotácie, ktoré sa odohrávajú nad daným podstromom a modifikujú ho. Vloženie prvku je rekurzívna funkcia vracajúca vždy daný podstrom do ktorého sa vnárame.

#### Hľadanie prvku

Vyhľadávanie prvku je jednoduchá rekurzívna funkcia v časovej zložitosti O(log n). Vyhľadáva vo vetvách stromu podľa daného dáta, ktoré je hľadané a traverzuje stromom, pokiaľ ho nenájde alebo nenarazí na list stromu.

search(subtree, data)

if subtree == NULL return NULL

if subtree.data == data return subtree

if data > subtree.data

return search(subtree.right)

else

return search(subtree.left)

### Testovanie

Pre testovanie správnosti mojej implementácie AVL stromu som pripravil päť rozličných testov.

#### Test 1

Všeobecný test, ktorý testuje vkladanie, hľadanie, existenciu prvkov, duplikáciu prvkov a ich správnu výšku.

V tomto teste ukážeme aj testovací kód, pri zvyšných testoch budú len výsledky z konzole.

Takisto meriame aj čas v sekundách na konci programu.

V prípade neúspechu niektorej z overovacích funkcií should by program skončil s exit kódom 1.

A screen shot of a computer

Description automatically generated

Obrázok 1: Ukážka testovacej sady

A picture containing clock, drawing

Description automatically generated

Obrázok 2: Ukážka diagramu stromu a rotácie pri vložení a vytvorení nevyváženia (Použije sa rotácia LR - nevyváženosť je v pravom potomkovi (20) ľavého podstromu (10)

A screenshot of a cell phone

Description automatically generated

Obrázok 3: Výsledok testu, In order traverzovanie stromom, výpis jednotlivých uzlov a listov, hĺbky, koeficientu vyváženosti a počet uzlov.  
Poznámka: Výšku stromu číslujeme od 1 (root == 1)

#### Test 2: Pravá rotácia (nevyváženosť LL)

Test zameraný konkrétne na správne riešenie rotácie LL.

A close up of a map

Description automatically generated

Obrázok 4: Diagram vloženia a simulácia LL rotácie   
(Zdroj: <https://static.javatpoint.com/ds/images/ll-rotation-in-avl-tree-solution.png>)

A close up of text on a black background

Description automatically generated

Obrázok 5: LL rotácia

#### Test 3: Ľavá rotácia (nevyváženosť RR)

Test zameraný konkrétne na správne riešenie rotácie RR.

A close up of a device

Description automatically generated

Obrázok 6: Diagram vloženia a simulácia RR rotácie   
(Zdroj: <https://static.javatpoint.com/ds/images/rr-rotation-in-avl-tree-solution.png>)

A close up of text on a black background

Description automatically generated

Obrázok 7: RR rotácia

#### Test 4: Rotácia LR

Test zameraný konkrétne na správne riešenie rotácie LR.

A close up of a map

Description automatically generated

Obrázok 8: Diagram vloženia a simulácia LR rotácie   
(Zdroj: <https://static.javatpoint.com/ds/images/lr-rotation-in-avl-tree-solution.png>)

A close up of text on a black background

Description automatically generated

Obrázok 9: LR rotácia

#### Test 5: Rotácia RL

Test zameraný konkrétne na správne riešenie rotácie RL.

A close up of a map

Description automatically generated

Obrázok 10: Diagram vloženia simulácia RL rotácie  
(Zdroj: <https://static.javatpoint.com/ds/images/rl-rotation-in-avl-tree-solution.png>)

A close up of text on a black background

Description automatically generated

Obrázok 11: RL rotácia

## Červeno-čierny strom

### Implementácia

Implementácia červeno-čierneho stromu v tomto zadaní je prevzatá z portálu Programiz.com: <https://www.programiz.com/dsa/insertion-in-a-red-black-tree>.

## Porovnanie

Oba algoritmy sú z rovnakého druhu – sú to algoritmy na vyvažovanie binárneho stromu. Ich teoretická časová zložitosť je rovnaká – vkladanie aj vyhľadávanie je O(log n). Pamäťová zložitosť oboch algoritmov je O(n).

Medzi rozdiely algoritmov patrí však časová zložitosť z praktického hľadiska. AVL stromy sú viac vyváženejšie a pre to vyhľadávanie v reálnom čase je rýchlejšie ako pri červeno-čiernom strome. Tieto stromy bývajú teda viac využívané v štruktúrach ako sú databázy, kde je potrebné rýchle získanie dát. Naopak, vkladanie dát je v reálnom čase pomalšie, pretože AVL strom rebalancuje strom pri každom vložení.

Z pamäťového hľadiska je červeno čierny strom jednoduchšie ukladať, pretože na uloženie informácie nám stačí pri najmenšom bit alebo char, kým na výšku stromu v každom uzle pri AVL strome potrebujeme minimálne integer.

Pre porovnanie, že červeno-čierny strom nemusí byť vždy vyvážený demonštrujme v oboch algoritmoch nasledujúci vstup: 2,1,4,5,9,6,7. (University of Maryland, Baltimore County)

A picture containing clock

Description automatically generated

Obrázok 12: Demonštrácia nevyváženosti stromu pri červeno-čiernom strome  
(Zdroj: <https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>)

A close up of a logo

Description automatically generated

Obrázok 13: Rovnaký vstup, AVL strom  
(Zdroj: <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>)

A screenshot of a cell phone

Description automatically generated

Obrázok 14: Čas funkcie získania čísla 6 AVL stromu pri danej konfigurácií

A picture containing meter

Description automatically generated

Obrázok 15: : Čas funkcie získania čísla 6 červeno-čierneho stromu pri danej konfigurácií

Pre reálneho porovnanie času potrebné na vloženie prvku a demonštráciu nášho tvrdenia sme vytvorili nasledovný test, kde vkladáme 10000 prvkov.

A screenshot of a cell phone

Description automatically generated

Obrázok 16: Čas potrebný na vloženie 10 tisíc prvkov do AVL stromu

A screenshot of a cell phone

Description automatically generated

Obrázok 17: Čas potrebný na vloženie 10 tisíc prvkov do červeno-čierneho stromu

# Hashovacie tabuľky

V prípade hashovacích tabuliek som si pre vlastnú implementáciu vybral algoritmus riešenia kolízií otvoreného adresovania, konkrétne kvadratické skúšanie (quadratic probing) a prevzatou implementáciou algoritmu reťazenia.

## Otvorené adresovanie - Kvadratické skúšanie

Kvadratické skúšanie je jednou z metód riešenia kolízií v hashovaní. Je to nadstavba lineárneho skúšania so snahou zamedziť primárnemu clustrovaniu hodnôt v hashovacej tabuľke.

### Implementácia

#### Dátová štruktúra

Hashovaciu tabuľku sme uložili do štruktúry obsahujúcej celkovú veľkosť a počet kľúčov – int size, int keys. Táto štruktúra taktiež obsahuje samotnú tabuľku table, reprezentovanú dynamickým poľom typu int.

Dôvodom prečo je nutné si uchovávať aktuálny počet a veľkosť tabuľky je výpočet faktoru naplnenia.

#### Vloženie prvku

Samotné vloženie prvku je pomerne jednoduché a skúsime ho vysvetliť na nasledovnom pseudokóde.

Insert(table,key)

If (load\_factor(keys+1) > 0.5)

Rehash(table)

index = hash\_fn(key);  
 if (table[index] != FREE)  
 index = quadratic\_probe(key)  
  
 table[index] = key;

Vkladanie prvku sa uskutoční získaním indexu z hashovacej funkcie. Vo vlastnej implementácií používame jednoduchú hashovaciu funkciu fn(k) = k mod SIZE, kde SIZE je veľkosť hashovacej tabuľky.

Samotné riešenie kolízií tkvie v tom, že pokiaľ nie je daný index voľný, vyskúša sa iné miesto v tabuľke. Rozdiel medzi lineárnym a kvadratickým skúšaním je vo výbere miesta. Kým lineárne skúšanie skúša voľnosť prvkov hashovacej tabuľky za daným prvotným indexom, kvadratické skúšanie pripočítava ku kľúču druhú mocninu iterátoru skúšania, čo celé predelí so zvyškom veľkosťou tabuľky. Takto zamedzuje už vyššie spomenutému primárnemu clustrovaniu – to je prípad, keď sa v tabuľke zhromažďujú hodnoty v okolí nejakého kľúča.

Zámerne sme vynechali vysvetlenie prvého kroku pseudokódu. V prípade hashovacích tabuliek s otvoreným adresovaním je práve faktor naplnenia veľkou premennou v prípade časovej zložitosti algoritmu. Faktor naplnenia λ = počet prvkov / veľkosť tabuľky. Faktor naplnenia udržovaný pod hodnotou 0.5 nám zabezpečuje, aby bola časová zložitosť hľadania prvku v tabuľke konštantná.

V prípade teda, že by faktor naplnenia bol vyšší po najbližšom vkladaní, musíme zväčšiť zdvojnásobiť veľkosť tabuľky a znova zahashovať prvky. Dôvodom prečo sa musia znova zahashovať prvky je, že hashovacia funkcia je závislá na veľkosti tabuľky, a pri zmenení veľkosti tabuľky by dané prvky už neboli získateľne novou hashovaciou funkciou.

#### Hľadanie prvku

Samotné hľadanie prvku je v kvadratickom skúšaní skoro totožné s vkladaním prvku. Vynecháva sa iba prípad zväčšovania a rehashovania tabuľky.

Hľadá sa hashovaciou funkciou kľúč a ak sa daná hodnota nenachádza na danom indexe, tak sa skúšajú všetky lokácie pripočítania druhej mocniny iterátoru vydelené veľkosťou tabuľky.

V prípade, ak však algoritmus narazí na index tabuľky ktorý je prázdny, môžeme s určitosťou povedať že daný kľúč sa v tabuľke nenachádza.

#### Zväčšenie tabuľky

Zväčšenie tabuľky je operáciou vytvorenia novej tabuľky s dvojnásobnou veľkosťou predchádzajúcej a znova zahashovanie jednotlivých prvkov starej tabuľky s hashovaciou funkciou novej.

### Test

Pre demonštráciu správnosti riešenia sme vytvorili jednoduchý test pre vkladanie a hľadanie prvkov v hashovacej tabuľke a správnosť jej zväčšovania.

A screenshot of a video game

Description automatically generated

Obrázok 18: Kód testu vkladania prvkov do hashovacej tabuľky s kvadratickým skúšaním

A screenshot of a cell phone

Description automatically generated

Obrázok 19: Test vkladania prvkov do hashovacej tabuľky s kvadratickým skúšaním

A picture containing computer

Description automatically generated

Obrázok 20: Zväčšená tabuľka

Obrázok 20 demonštruje zdvojnásobenie veľkosti tabuľky po vložení 6teho kľúču do tabuľky s inicializovanou veľkosťou 10 po prekročení limitu faktoru naplnenia.

## Reťazenie

### Implementácia

Implementácia hashovacej tabuľky s algoritmom riešenia kolízií pomocou separátneho reťazenia v tomto zadaní je prevzatá z portálu Log2Base2.com: <https://www.log2base2.com/algorithms/searching/open-hashing.html>

## Porovnanie

Oba algoritmy riešenia kolízií majú rovnakú teoretickú časovú i pamäťovú zložitosť.

Ohľadom pamäťovej zložitosti, je to O(n). V praxi pri ukladaní numerických dát ako bolo v tomto cvičení demonštrované je však lepšie z pamäťového hľadiska použiť pole typu int miest lineárneho zoznamu.

Časová zložitosť oboch algoritmov je veľmi závislá na faktore naplnenia λ. Priemerná časová zložitosť prehľadávania s úspešným koncom je O(1 + λ/2). V najlepšom prípade (hodnota existuje priamo na kľúči) je O(1), v najhoršom prípade (traverzovanie zoznamom až na koniec alebo nenájdenie prvku) je to O(1+λ).

V porovnaní samotných algoritmov, sú to rozdielne algoritmy – tzv. Closed hashingu a open hashingu a rozdielne aj pracujú.

Algoritmy open hashingu, ako je aj separate chaining ukladajú kľúče aj v tabuľke, aj mimo tabuľky (zoznam), čo zabraňuje napríklad možnosti cachovania, ktorú closed hashing (probing, double hashing) umožňuje.

Nevýhodou v reťazeniu je, že obsahuje prázdne „vedierka“, ktoré nikdy využité nebudú, ak na ne neukazujú žiadne kľúče. V prípade closed hashingu to tak nie je, prvky hashovacej tabuľky sú využívané, aj keď na ne žiadny kľúč neukazuje (priamo).

Problémom skúšania – hlavne lineárneho je nemožnosť reálneho vymazania prvku z tabuľky – Vytvorila by sa tak medzera medzi jednotlivými prvkami. Dalo by sa to riešiť buď časovo náročným presúvaním zvyšku clustra, alebo označením daného „vedierka“ ako vymazané, čo by neprerušilo cyklus vyhľadávania.

Posledný z testov na porovnanie tvrdenia týchto dvoch rozličných algoritmov sme spravili test vkladania prvku do hashovacej tabuľky, konkrétne tisíc prvkov a odmerali čas.

Tento test nám však príde trošku nešťastný a to z dôvodu, že hashovacia tabuľka získaná z internetu s algoritmom reťazenia mala fixnú veľkosť a nemala implementované zväčšovanie. A takú so zväčšovaním sa nám nepodarilo nájsť.

Pravdepodobne z toho dôvodu nebude tento test objektívny, i keď na ňom vidíme že vkladanie tisícky prvkov do tabuľky s reťazením trvalo niekoľkonásobok viac času ako samotnému kvadratickému skúšaniu. Problémom však je že v prípade kvadratického skúšania zväčšovanie tabuľky dosiahlo až veľkosť 2559 prázdnych, nevyužitých „vedierok“.

A picture containing computer, keyboard

Description automatically generated

Obrázok 21: Test vkladania tisíc prvkov do tabuľky s reťazením

A screen shot of a computer

Description automatically generated

Obrázok 22: Test vkladania tisíc prvkov do tabuľky s kvadratickým skúšaním

# Zhodnotenie

Toto cvičenie ma naučilo veľa hlavne o hashovacích tabuľkách, ich typy implementácií, ktoré som bol nútený si viac naštudovať na internete, aby som neskôr mohol implementovať, čo ma veľmi bavilo v následnej implementácií - či už binárnych vyhľadávacích stromoch alebo už spomínaných hashovacích tabuľkách.

Nakoľko prevažne pracujem vo vyšších programovacích jazykoch v ktorých sú tieto štruktúry naimplementované, je veľmi dobré vedieť základy, ako tieto štruktúry fungujú a nemusím sa na ne teda už pozerať ako na "black-box".

# Referencie

BARI, A. *AVL Tree - Insertion and Rotations.* [cit. 12.04.2020]. Dostupné na internete: <https://www.youtube.com/watch?v=jDM6_TnYIqE>

BARI, A. *Hashing Technique – Simplified*. [cit. 12.04.2020]. Dostupné na internete:

<https://www.youtube.com/watch?v=mFY0J5W8Udk>

GATEVIDYALAY.COM. *Separate chaining vs. Open Addressing.* [cit. 12.04.2020]. Dostupné na internete: <https://www.gatevidyalay.com/tag/difference-between-linear-probing-and-quadratic-probing/>

LOG2BASE2.COM. *Open Hashing.* [cit. 12.04.2020]. Dostupné na internete: <https://www.log2base2.com/algorithms/searching/open-hashing.html>

STEPIK.ORG. *Hash Table and Hash Map.* [cit. 12.04.2020]. Dostupné na internete:

<https://stepik.org/lesson/31445/step/7>

UNIVERSITY OF MARYLAND, BALTIMORE COUNTY. *Red Black Tree Practice Problems.* [cit. 12.04.2020]. Dostupné na internete:

<https://www.csee.umbc.edu/courses/undergraduate/341/spring04/hood/notes/red_black/>

UNIVERSITY OF SAN FRANCISCO. *Visualization of Red/Black Tree.* [cit. 12.04.2020]. Dostupné na internete: <https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

UNIVERSITY OF SAN FRANCISCO. *Visualization of AVL Tree.* [cit. 12.04.2020]. Dostupné na internete: <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>