Fakulta informatiky a informačných technológií STU v Bratislave Ilkovičova 2, 842 16 Bratislava 4

Tibor Dulovec

**Zadanie 2 – Vyhľadávanie v dynamických množinách**

DSA LS 2020/21  
Cvičiaci P. Lehoczký  
Pondelok 11:00 – 12:50

**Obsah**

[Opis zadania 3](#_Toc68985837)

[Obsiahnuté implementácie 3](#_Toc68985838)

[Zdroje prevzatých implementácií 3](#_Toc68985839)

[Splay Tree 3](#_Toc68985840)

[Hashtable: Open addressing 3](#_Toc68985841)

[AVL stromy 4](#_Toc68985842)

[Funkcie AVL tried 4](#_Toc68985843)

[Pridávanie hodnoty: addItem() 4](#_Toc68985844)

[Vyváženie uzla rebalanced() 4](#_Toc68985845)

[Vyhľadávanie hodnoty: findItem() 5](#_Toc68985846)

[Testovacie funkcia pre výpis stromu: printTree() 5](#_Toc68985847)

[Splay Tree 6](#_Toc68985848)

[Hashtable 7](#_Toc68985849)

[Chaining 7](#_Toc68985850)

[Pridanie prvku: insert() 7](#_Toc68985851)

[Vyhľadávanie prvku: get() 7](#_Toc68985852)

[Zväčšenie tabuľky: resizeTable() 7](#_Toc68985853)

[Hashovacia funkcia: hash() 8](#_Toc68985854)

# Opis zadania

Zadanie bolo zamerané na vyhľadávanie v dynamických množinách. V zadaní som obsiahol štyri implementácie. Moje implementácie sú AVL stromy a hash tabuľka s kolíziami riešenými cez chaining. Zadanie je naprogramované v jazyku Java.

## Obsiahnuté implementácie

* AVL stromy
* Splay Tree
* Hashtable: Chaining
* Hashtable: Open addressing

# Zdroje prevzatých implementácií

Obe prevzaté implementácie boli upravené, pre správne fungovanie s mojím kódom.

## Splay Tree

<https://algorithmtutor.com/Data-Structures/Tree/Splay-Trees/>

## Hashtable: Open addressing

<https://www.algolist.net/Data_structures/Hash_table/Open_addressing>

# AVL stromy

AVL strom je prvý vynájdený samovyvažovací binárny vyhľadávací strom. Princíp jeho vyvažovanie je v bilancií každého uzla. Balancia je rozdiel výšok pravého a ľavého uzlu, ktorá sa prepočítava zakaždým po pridaní nového uzla. Vďaka rovnomernému vyváženiu je následné vyhľadávanie prvkov efektívnejšie.

Časová zložitosť vyhľadávania a vkladania je **O(log(n))**.

Funkčnosť AVL stromov je obsiahnutá v triede „AVL“. Pre správne fungovanie tejto triedy je potrebná aj trieda „AVLNode“, ktorá symbolizuje dane uzly do AVL stromu.

## Funkcie AVL tried

### Pridávanie hodnoty: addItem()

Za pomocou tejto funkcie pridávame nové uzly to stromu. Táto funkcia je **rekurzívna**. Ak sa dostaneme na miesto v uzly, ktoré ešte nie je obsadené, tak vložený objekt uložíme na jej miesto. Ale ak sa na tom mieste už nejaký objekt nachádza, tak na ten objekt(uzol) znova zavoláme funkciu **addItem**. Potom, ako sa pridá nový uzol, všetky uzly, ktoré boli týmto ovplyvnené vykonajú funkciu triedy AVLNode **rebalanced().** Týmto spôsobom vždy keď pridáme novú hodnotu, tak sa vyvážia všetky uzly, cez ktoré táto hodnota prešla.

### Vyváženie uzla rebalanced()

Funkcia je pre triedu AVLNode. Jej úlohou je vyvážiť strom, aby bol viac efektívny a cez návratovú hodnotu vrátiť vyvážený uzol.

Trieda AVLNode má okrem iného vlastnosti „**depth**“ a „**balance**“, ktoré predstavujú hĺbku a balanciu uzla. Hĺbka znač, na akej výške je. Napríklad 1 znamená, že pod ňou nič nie je a 3 znamená, že na jednej strane je uzol s výškou 2.

Na začiatku funkcie rebalanced() sa zavolá funkcia **getBiggerItem()**, ktorá vráti prvok s väčšou hĺbkou z daného uzla. Buď ľavý alebo pravý a podľa neho sa zvýši hĺbka uzla.

Ďalej sa nanovo vypočíta balancia. Tu vypočítame rozdielom hĺbok vnútorných uzlov.   
Nasledujú 4 if podmienky, ktoré podľa balancie rozhodujú či a akú rotáciu potrebuje uzol. V prípade že je balancia väčšia ako jedna. Vykoná sa **pravá rotácia**. Ak je menšia, vykoná sa **ľavá rotácia**.

Tieto rotácie prebiehajú tak, že sa zmení hodnota hĺbky a daný uzol sa vymení s pravým alebo ľavým. Záleží od toho, či je potrebná pravá alebo ľavá rotácia.  
Ľavá a pravá rotácia sa vykonáva v podstate rovnako. Rozdeľujú sa iba, ktoré strany sa kde premiestnia.

Existujú ale určite scenáre, kedy sa môže stať, žeby tieto rotácie nefungovali správne a otáčali sa nesprávne. V týchto prípadoch je potrebné vykonať **right-left** alebo **left-right rotáciu.**

V prípade left-right rotácie overujeme, či balancia ľavej strany uzla je menšia ako 0. V prípade right-left rotácie overujeme pravú stranu. A to či je balancia väčšia ako 0. Ak áno, vykonáme prehodenie uzlov a následne spravíme left alebo right rotáciu. Záleží od toho, ktorá rotácia bola vykonaná pred tým.

Po vykonaní rotácií, sa vráti vyvážený uzol a celý strom sa aktualizuje

### Vyhľadávanie hodnoty: findItem()

Funkcia na vyhľadávanie prvku je veľmi podobná ako pri vkladaní. Tiež je rekurzívna a rovnakým spôsobom prechádza prvkami. Rozdiel ale je, že funkcia sa ukončí, keď sa nájde zhoda s hľadaným menom a menom v uzli. Potrebné atribúty pre túto funkciu sú hľadaný výraz a strom v ktorom treba hľadať. V základe sa funkcia volá so stromom, ktorý je uložený v parametri „root“.

### Testovacie funkcia pre výpis stromu: printTree()

Táto funkcia je určená iba pre a vývojárske účely. Slúži na vypísanie kódu. Výpis obsahuje každý uzol. Meno (klúč) záznamu a v zátvorke je balancia a hĺbka uzla. Za šípkov sú uzly, ktoré sú na ľavej a na pravej strane. Celý výpis je ukončený výpisom, koľko uzlov strom obsahuje.  
 

# Splay Tree

Implementáciu som prevzal zo stránky [algorithmtutor.com](https://algorithmtutor.com/Data-Structures/Tree/Splay-Trees/). Má podobné výhody ako AVL strom. Je samo vyvažujúci a da sa podobne vizuálne znázorniť. Ale uzly sa presúvajú používaním. To znamená, že keď pristúpime k nejakému prvku, tak prvok sa posunie vyššie v strome a do budúcna bude rýchlejšie dostupný.

Vzhľadom k tomu, jeho časová zložitosť nemusí byť vždy rovnaká. Môže byť od **O(log(n))** až po **O(n)**

Vkladanie a hľadanie je štandardné ako pri ostatných vyvážených stromov. Rozdiel nastane po vyhľadaní prvku. Vtedy sa uzol otáča k koreňu stromu. Vďaka tomu, ak budeme v blízkej dobe opakovane vyhľadávať tento prvok, tak bude rýchlejšie prístupný a prvky, ktoré sa vôbec nevyhľadávajú budú hlboko v strome a nebudú negatívne ovplyvňovať čas vyhľadávania. Tieto rotácie sa nazývajú **Zig Zag rotácie.** V určitých scenároch, podobne ako pri AVL stromoch, je potrebná **Zig-zig** **rotácia**.

Výhoda použitia tejto implementácia je, ak sa často pristupuje k rovnaký Napríklad pre cash pamäť. Výrazným časovým problémom sa to stáva, keď sa k prvkom pristupujú veľmi náhodne.

# Hashtable

Hashtable je dynamická množina, ktorá je veľmi efektívna vďaka časovej náročnosti pri vyhľadávaní. Spája **kľuč** s hodnotami a preto ak sa chceme dostať k hodnote, stačí poznať kľuč, ktorý môže byť napríklad meno, a okamžite dostaneme prvok, ktorý sme vyhľadávali. Tak isto je výhodná rýchlosť pri pridávaní.  
Časová efektívnosť pridávania aj hľadania je **O(1)**

Časová efektívnosť sa ale môže mierne zvýšiť v scenároch, ktoré sa nazývajú **kolízie**. Stanú sa vtedy, keď po zhashovaní kľúču vyjde rovnaká hodnota. Práve preto sa snažíme dosiahnuť, aby bol hash čo najviac unikátny. No nie vždy to je možné a preto na riešenie kolízií existuje niekoľko spôsobov. Jeden z najrozšírenejších je **Chaining** (reťazenie).

## Chaining

Chaining je riešenie kolízií pre hashovacie tabuľky. V prípade, že kľúč po zhashovaní bude mať hodnotu, ktorá sa už opakovala. Prvok sa zreťazí k tomu prvku.

### Pridanie prvku: insert()

Parametre pre túto funkciu sú kľuč a objekt, ktorý chceme pridať. Na začiatku funkcie sa vygeneruje index podľa hash funkcie do ktorej vložíme kľúč.

Ak na tomto indexe nič nie je, vložíme tam daný objekt. Ak sa tam ale už niečo nachádza, je potrebné nový objekt zreťaziť.

Uzol pre Hashovanie s kolíziami riešenými cez chaining je definovaný podľa triedy „**ChainingNode**“. V prípade, že je potrebné prvky reťaziť, tak sa ukladajú do prvého prvku na danom indexe. Prvok má vlastnosť ArrayList, v ktorej sú uložené všetky zreťazené prvky. Následne sa do neho pridáva alebo ním prechádza, ak je potrebné niektorý prvok nájsť.

### Vyhľadávanie prvku: get()

Parameter pre túto funkciu je kľúč. Tento kľúč sa rovnakou funkciou, ako pri pridávaní, použije na vygenerovanie indexu. A v časovej náročnosti O(1) sa v poli prvkov vyberie. Ak na tomto mieste nič nie je, znamená to že bol zadaný nesprávny kľúč a vráti sa hodnota null. Ak sa nachádza, overí sa či sa meno prvku zhoduje s kľúčom. Ak áno, funkcia tento prvok vráti.

Ak sa ale na danom mieste prvok nachádza a meno sa stále nezhoduje s kľúčom, môže to znamenať, že prvok môže byť v reťazení a preto sa celý ArrayList prehľadá a skúsi nájsť zhodu.

V ideálnom prípade bude zreťazenie minimálne. No čím viac dát na čím menšie pole prvkov, tak tým menej bude vznikať jedinečných indexov a zreťazenie bude narastať. Funkcia „**resizeTable**“ slúži nato, aby v prípade že sa počet voľných indexov bude výrazne zmenšovať, tak celá hash tabuľka sa zväčší.

### Zväčšenie tabuľky: resizeTable()

Pri každom pridávaní nového prvku na miesto indexu, sa do hodnoty „usedIndex“ zaznačí využitie nového indexu. Ak voľných indexov je menej ako polovica, tak sa zavolá funkcia resizeTable. Táto funkcia vytvorí nový zoznam prvkov do ktorého podľa nového hashovania priradí každý jeden prvok. Na konci funkcie táto nová tabuľka, s novou veľkosťou, nahradí tu starú.

Výhoda tejto funkcie je tá, že v prípade, keby sa do implementovala funkcia na vymazávanie prvkov, tak jej upravenie na zmenšenie tabuľky by bolo veľmi jednoduché

### Hashovacia funkcia: hash()

Pôvodne bola táto funkcia obsiahlejšia aby generovala dostatočne jedinečný index. Nakoniec som ju nahradil jednoduchou funkciou „hashCode“, ktorá je v jazyku Java dostupná pre String. Efektívnosť na veľkosti 300 tisíc bola identická, tak som ju touto Java funkciou nahradil.

## Open addressing

Táto implementácia bola prezvaná z webstránky: [algolist.net](https://www.algolist.net/Data_structures/Hash_table/Open_addressing)

Open addressing je ďalší spôsob riešenia kolízií. Nevzniká tú žiadne reťazenie a všetky prvky sú uložené v poli prvkov. Vďaka tomu je táto implementácia jednoduchšia na zhotovenie. V prípade, žeby všetky vygenerované indexy boli unikátne tak by bola aj časovo efektívnejšia. To ale nevieme zaručiť a tak nárastom prvkov pri tejto implementácií sa časová náročnosť zhoršuje.

# Testovanie

Testované boli primárne časové zložitosti. Každá implementácia mala odmeranú dobu pridávania a dobu vyhľadávania každého jedného prvku. Vyhľadával som každý jeden prvok, a nie len zopár určených, z dôvodu, či sa úspešne všetky prvky pridané. V súbore „main“ sú všetky testy vložené.

Záznamy na testovanie sú importované zo súborov vo formáte „csv“. Tie som vygeneroval pomocou online stránky. Projekt obsahuje 7 týchto súborov, ktoré sú rozlišujú počtom záznamov. Od 50 do 300 000.

Pomocou funkcie „**getItemFromCsvFile**“ sa všetky tieto záznamy nahrajú do ArrayListu „**importedItems**“. Tieto záznamy potom používajú všetky testovacie funkcie rôznych implementácií. Používajú sa rovnaké záznamy pre presné porovnávanie časovej efektívnosti. 

V hlavnej main funkcií sa následne spustia všetky testovacie funkcie. Z dôvodu prehľadnosti sú veľmi podobné názvom aj telom funkcie.

## Testovacia funkcia InsertItems()

Každá implementácia ma vlastnú funkciu pre automatické vloženie všetkých prvkov do pamäte. Na začiatku funkcie sa uloží stopa času, ktorá sa porovná s časom po vykonaní funkcie. Tento časový rozdiel sa vypíše aj s údajom koľko celkovo údajov bolo do pamäte vložených.

## Testovacia funkcia SearchItems()

Každá implementácia ma vlastnú funkciu ktorá vyhľadá všetky prvky, ktoré mali byť uložené do pamäte. Na začiatku a na konci funkcie sa tak isto uložia stopy času, ktorých rozdiel sa potom vypíše po vyhľadaní všetkých prvkov. K času sa vypíše aj koľko prvkov z koľkých bolo nájdených. V ideálnom prípade to budú vždy dva rovnaké čísla.