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](#_Toc68976350)

[Obsiahnuté implementácie 3](#_Toc68976351)

[AVL stromy 4](#_Toc68976352)

[Funkcie AVL tried 4](#_Toc68976353)

[Pridávanie hodnoty: addItem() 4](#_Toc68976354)

[Vyváženie uzla rebalanced() 4](#_Toc68976355)

[Vyhľadávanie hodnoty: findItem() 5](#_Toc68976356)

[Testovacie funkcia pre výpis stromu: printTree() 5](#_Toc68976357)

# 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.