**FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ SLOVENSKÁ TECHNICKÁ UNIVERZITA**

lkovičova 2, 842 16 Bratislava 4

2022/2023

Dátové štruktúry a algoritmy

Zadanie č.1

# Úvod

Účelom tohto článku je zdokumentovať moju prácu na 1. zadaní v predmete Dátové štruktúry a algoritmy. Zdrojový kód som písal v jazyku C++. V prvom semestri som sa naučil jazyk C, no C++ má napríklad knižnicu <list>, typ ‘auto‘ a niekoľko ďalších vecí, ktoré mi pri vypracovávaní tohto zadania zjednodušili život, a navyše som sa naučil nový programovací jazyk.

Štruktúra tejto dokumentácie je nasledovná: v prvej sekcii je popísaný zdrojový kód pre implementáciu dátových štruktúr. V druhej sekcii sa venujem testovaniu časovej náročnosti mojich algoritmov a na záver rozoberám výsledky testovania.

Table of Contents

[Úvod 2](#_Toc130039430)

[Zdrojové kódy 3](#_Toc130039431)

[AVL Strom 3](#_Toc130039432)

[Balansovanie 4](#_Toc130039433)

[Insert, Search, Delete 6](#_Toc130039434)

[Výpis a main funkcia 8](#_Toc130039435)

[Splay strom 11](#_Toc130039436)

# Zdrojové kódy

## AVL Strom

Ako prvú dátovú štruktúru som si vybral AVL strom na základe odporúčania kamarátov z vyšších ročníkov. Definoval som si štruktúru Node (uzol stromu) v ktorej sa uchovávajú vstupné dáta. Aby som sa naučil pracovať s objektovou paradigmou, ktorá bola pre mňa nová, implementoval som ho v triede – AVLTree, kde som jednotlivé funkcie zadefinoval ako metódy tejto triedy.

Text

Description automatically generated

V hlavičke sú zahrnuté knižnice:

* <iostream> na prácu s terminálom – input-output
* <algorithm> na rôzne funkcie, použil som iba max()
* <fstream> na prácu s textovými súbormi – tie som použil na testovanie
* <string> na jednoduchšiu prácu s reťazcami
* <chrono> na účely merania času

Štruktúra uzla obsahuje:

* Int key – uchováva číselnú hodnotu uzla
* Int height – uchováva výšku uzla v strome (súčasť algoritmu na vyvažovanie)
* String name – uchováva extra informácie asociované s číselnou hodnotou key
* Ukazovatele na dieťa s menšou hodnotou a dieťa s väčšou hodnotou

V triede som ďalej definoval konštruktor **AVLTree()** , ktorý nastaví hodnotu pomocného pointera root (koreň) na NULL a deštruktor **~AVLTree()**, ktorý volá funkciu **destructTree(\*\*node)**. Deštrukciu stromu som vyriešil takýmto spôsobom preto, lebo najjednoduchšie sa to celé premaže rekurzívne, no deštruktor (ani konštruktor) by nemal mať rekurziu.

### Balansovanie

Na účely vyvažovania stromu som si zadefinoval pomocné funkcie **getHeight(\*node)**, **getBalance(\*node)** a **updateHeight(\*node)**, tieto sú ďalej využité vo funkcii **balanceNode(\*node)**.

Funkcia **getHeight(\*node)** je veľmi jednoduchá, no veľa krát sa používa a preto je jednoduchšie si ju zadefinovať zvlášť. Ak uzol neexistuje (\*node == NULL), vráti 0, inak vráti jeho výšku. Týmto spôsobom sa vyhýbam možnému segmentation faultu.

Funkcia **getBalance(\*node)** vráti 0, ak uzol neexistuje (\*node == NULL), inak vypočíta a vráti balans daného uzla: *getHeight(ľavé dieťa) mínus getHeight(pravé dieťa).*

Funkcia **updateHeight(\*node)** sa volá po vložení alebo vymazaní uzla a slúži na aktualizáciu výšky daného uzla. Funkcia **max(...)** vráti maximálnu hodnotu z výšok detí a k tomu sa pripočíta +1.

Text

Description automatically generatedText

Description automatically generated

Vyššie spomenuté funkcie majú ako som už spomenul uplatnenie vo funkcii **balanceNode(\*node).** V tejto funkcii sa na základe hodnoty balansu *(int bf)* volajú jednotlivé rotácie:

* Ak je balans uzla > 1 a balans jeho menšieho dieťaťa je < 0,

zavolá sa najprv **leftrotate(\*node)** a potom **rightRotate(\*node)**

* Ak je balans uzla > 1 a balans jeho menšieho diťaťa je >= 0,

zavolá sa iba **rightRotate(\*node)**

* Ak je balans uzla < -1 a balans jeho väčšieho dieťaťa je > 0,

zavolá sa najprv **rightRotate(\*node)** a potom **leftRotate(\*node)**

* Ak je balans uzla < -1 a balans jeho väčšieho dieťaťa je <= 0,

zavolá sa iba **leftRotate(\*node)**

Text

Description automatically generatedTieto rotácie sú jadrom balansovania pomocou AVL algoritmu, a zabezpečujú fakt, že v AVL strome nikdy nie je niektorá vetva dlhšia (vyššia) o viac ako jeden uzol od druhej. Vďaka tomu sú AVL stromy kompletne vybalansované, narozdiel od mnohých iných stromov, ktoré sú iba približne vybalansované.

Funkcia **leftRotate(\*node)** dostáva ukazovateľ na uzol *(node),* podľa ktorého chceme strom otočiť doľava. Do pomocnej premennej *(temp)* uloží väčšie dieťa tohto uzla, a do ďalšej pomocnej premennej *(temp2)* uloží menšie dieťa štruktúry uloženej v temp. Tieto kroky sú nevyhnutné na to, aby sa zabránilo strácaniu detí alebo porušeniu vlastností stromu. Ďalej sa zamenia temp za pôvodný uzol, aktualizuje sa výška uzlov, ktoré sa vymieňali a vracia sa ukazovateľ temp.

Funkcia **rightRotate(\*node)** robí analogický opak funkcie ľavého otáčania. Praktický príklad týchto rotácii je uvedený na obrázkoch nižšie.

Diagram

Description automatically generatedDiagram

Description automatically generated

### Insert, Search, Delete

Trieda AVLTree má ďalej metódy:

* **addNode()** – na vkladanie uzlov
* **findNode()** – na vyhľadávanie uzlov
* **deleteNode()** – na mazanie uzlov

Funkcia **addNode(\*node, ....)** funguje rekurzívne. Postupne hľadá kam má vložiť zadané dáta podľa hodnoty value *(key)* a akonáhle sa posunie na miesto stromu, kde nie je uzol, tak ho vyrobí a vloží na to miesto. Ak zadaná hodnota v strome už je, funkcia vráti pôvodný strom (neurobí teda nič). Ak sa nejaký prvok do stromu pridal, zavolajú sa vyššie spomenuté funkcie **updateHeight(\*node)** a  **balanceNode(\*node).**

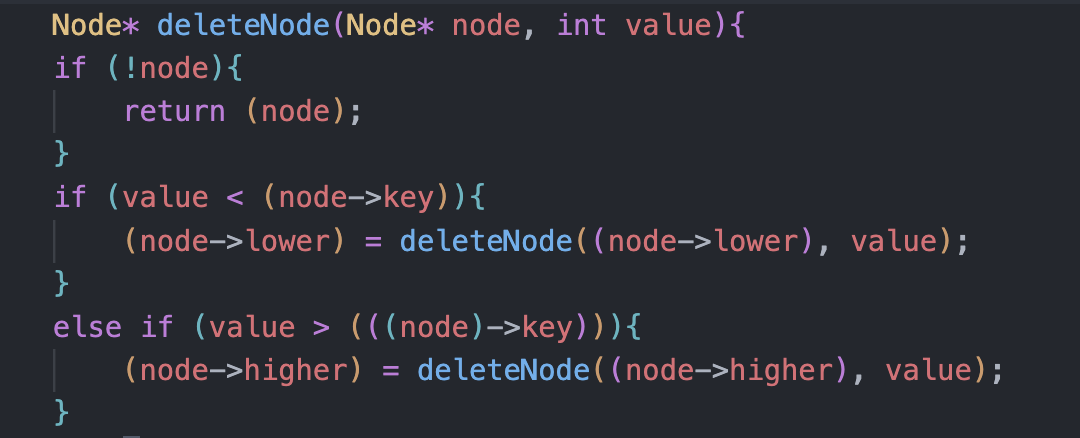
Text

Description automatically generated

**Text

Description automatically generated**Funkcia **findNode(\*node, value)** funguje tiež rekurzívne. Postupne prechádza strom podľa zadanej hodnoty, ktorú má nájsť a ak nájde uzol s danou hodnotou, vráti dáta uchované v tomto uzle, ináč vráti správu o tom že uzol nebol nájdený.

Funkcia **deleteNode(\*node)** najprv rekurzívne nájde uzol, ktorý sa má vymazať podľa zadanej hodnoty, podobne ako pri hľadaní vo funkcii **findNode(\*node, value)**,s tým rozdielom, že ak sa uzol danej hodnoty v strome nenáchádza, nič sa nestane.



Následne, môžu nastať štyri situácie. Uzol, ktorý sa má vymazať má:

* žiadne dieťa
* iba menšie alebo iba väčšie dieťa
* obidve deti

Funkcia rozpozná, o akú situáciu ide a vyrieši ich nasledovne:

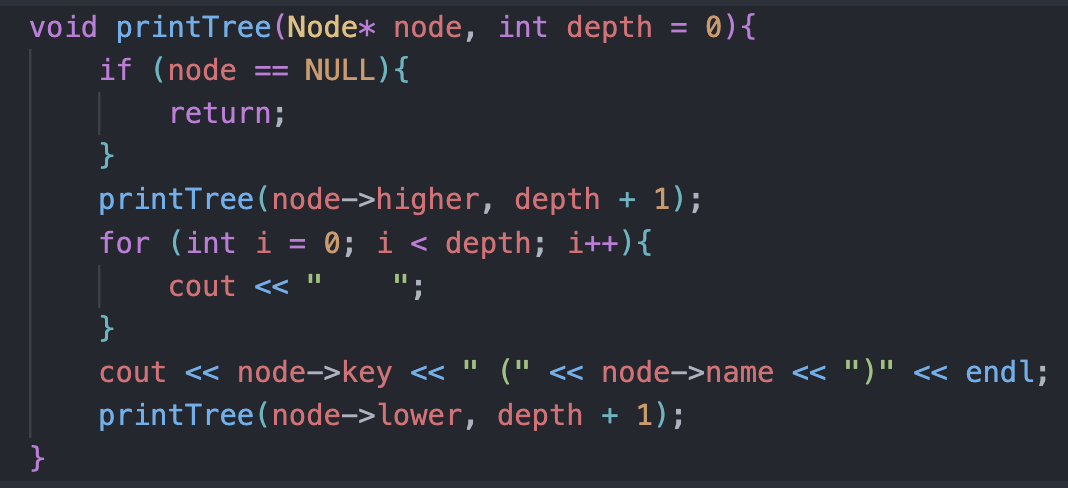
* ak nemá žiadne dieťa, jednoducho ho vymaže
* ak má iba jedno dieťa, ukazovateľ na uzol sa posunie na svoje dieťa a pôvodný uzol sa vymaže
* ak má obe deti, funkcia nájde najmenší uzol z pravého podstromu pôvodného uzla *(temp)*, nahradí hodnotu pôvodného uzla touto hodnotou *(temp->key)* a následne rekurzívne vymaže duplikátnu hodnotu.

Text

Description automatically generatedPo vymazaní sa samozrejme opäť zavolajú funckie **updateHeight(\*node)** a **balanceNode(\*node),** aby sa zaistilo, že strom je aj naďalej vyvážený.

### Výpis a main funkcia

Poslednou metódou v triede AVLTree je **printTree(\*node, depth),** ktorá jednoducho rekurzívne vypíše uzly stromu od koreňa do terminálu v smere zprava doľava. Ak je strom prázdny, funkcia neurobí nič.



Na začiatku mainu mám deklarovaných zopár premenných, pomocou ktorých sa vykonávajú zmeny v strome, ukazovateľ na objekt triedy AVLTree a  ukazovatele na súbory rôznych veľkostí s náhodne vygenerovanými číslami a reťazcami pre účely testovania.

A screenshot of a computer

Description automatically generated with medium confidence

Ďalej mám v maine už len veľký switch(), ktorý na základe užívateľom zadaného vstupu rozhoduje o tom, či sa bude so stromom pracovať manuálne alebo prebehne automatický test.

Text

Description automatically generated

Samozrejme, že užívateľ má možnosť program aj vypnúť, ak stlačí ‘q’.

V prípade že stlačí čokoľvek iné ako zadané hodnoty, program len vypíše správu s inštrukciami na používanie.

Ak užívateľ stlačí ‘m’, spustí sa ďalší switch(), v pomocou ktorého môže používateľ po jednom buď pridávať uzly do stromu, vyhľadávať ich, mazať ich alebo strom vypísať do terminálu. Samozrejme, že aj z tohto miesta má užívateľ možnosť sa vrátiť naspäť na výber medzi manuálnym používaním a automatickým testom. Na začiatku, ako aj po zadaní hociakého neakceptovaného vstupu program vypíše správu s inštrukciami.

Ak užívateľ stlačí ‘a‘, spustí sa switch(), v ktorom si môže užívateľ vybrať, na akej veľkej sade uzlov sa strom automaticky otestuje. Test prebehne tak, že sa najprv zadaný počet uzlov vloží do stromu, potom sa po jednom vyhľadajú a nakoniec sa po jednom vymažú. Program následne vypíše správu o čase trvania jednotlivých operácii ako aj celkovom čase, za ktorý to celé prebehlo. Výpis správy s inštrukciami ako aj možnosť vrátenia sa tu fungujú rovnako ako pri manuálnom testovaní.

Tieto switche sú na príliš veľa riadkov na to, aby som ich mohol nejako rozumne celé ukázať tu, no vždy vypíšu užívateľovi na začiatku možnosti ovládania a sú pomerne jednoduché na používanie. Pre celistvosť tejto dokumentácie je ešte nižšie uvedená prvá možnosť z automatického testovania, ktorá je vzorom pre všetky ostatné.

Text

Description automatically generated

Záverečné poznámky:

* Časy meraní sú uvedené v mikrosekundách, pri sadách 1M a 10M sú však uvedené v milisekundách, nakoľko desatinné miesta sú pri týchto možnostiach zanedbateľné.
* Podľa nastavenia compileru môže po spustení programu užívateľa privítať až krásnych 56 warningov, všetky sa však sťahujú na zmienený typ ‘auto’, ktorý je použitý pri deklaráciach časových premenných pri automatickom testovaní. Tento typ je totiž súčasťou jazyka C++ až od verzie 11 a vyššie, dnes sa už však všeobecne používa verzia 17, takže tieto warningy sú viacmenej bezpredmetné – hovoria len o tom, že program je nekompatibilný so zastaranou verziou jazyka.
* Ak sa použije možnosť q vo výbere m, tj. je možnosť, že používateľ manuálne pridal do stromu nejaké uzly a zabudol ich odtiaľ pred odchodom korektne vymazať, zavolá sa funkcia destructTree(), ktorá to urobí za neho. Preistotu sa volá aj po stlačení q vo výbere a, aj keď kód automatických testov všetky uzly vymazáva.

## Splay strom

Ako druhý algoritmus na balansovanie binárneho vyhľadávacieho stromu som si vybral Splay, kvôli jeho zaujímavej charaktristike, ktorá ho v mojich očiach robí dobrým kandidátom na použitie v mnohých prípadoch.

V hlavičke sú zahrnuté rovnaké knižnice ako pri AVL strome, konštruktor, deštruktor aj funkcia destructTree() sú tiež totožné.

Štruktúra Node je taktiež veľmi podobná, s tým rozdielom, že pri Splay strome som sa rozhodol uchovávať si aj ukazovateľ na rodiča uzla. Týmto krokom som si zjednodušil implementáciu funkcie splayNode(), keďže rekurzívnu som si túto funkciu nevedel predstaviť.

Text

Description automatically generated

### Balansovanie

Na balansovanie Splay stromu slúži funkcia splayNode(\*myNode, \*rootNode), ktorá na vstupe dostáva ukazovateľ na uzol, ktorý má byť „vynesený“ *(mynode)* a ukazovateľ na uzol, ktorého dieťaťom sa má stať *(rootNode),* tj. pozíciu kam sa má *myNode* „vyniesť“. Pomocou rotácii potom „vynesie“ zadaný uzol na zadanú pozíciu v strome, najčastejšie na koreň – ak má byť uzol „vynesený“ na koreň, destinácia *rootNode* je nastavená na nullptr, keďže koreň ako jediný nemá rodiča.

Funkcia prebieha, pokým rodič daného uzla nie je ten ktorý bol zadaný. Najprv sa vždy overí či náhodou nie je rodič rodiča uzla jeho finálnou destináciou, v tom prípade totiž stačí jediná posledná rotácia:

* Ak je uzol napravo od svojej destinácie tak sa zrotuje doľava
* Ak je uzol naľavo od svojej destinácie tak sa zrotuje doprava

Ak nie, môžu nastať štyri situácie:

* Uzol je menší od svojho rodiča a jeho rodič je menší od svojho rodiča, treba vykonať dve pravé rotácie
* Uzol je väčší od svojho rodiča a jeho rodič je väčší od svojho rodiča, treba vykonať dve ľavé rotácie
* Uzol je menší od svojho rodiča a jeho rodič je väčší od svojho rodiča, treba vykonať najprv pravú rotáciu a potom ľavú rotáciu
* Uzol je väčší od svojho rodiča a jeho rodič je menší od svojho rodiča, treba vykonať najprv ľavú rotáciu a potom pravú rotáciu

Text

Description automatically generatedTieto rotácie sa inak nazývajú aj zig, zag, zig-zig, zig-zag,...

Rotácie, ktoré sú vo funkcii splayNode() volané, sú podobné tým z AVL stromu, no tieto zahŕňajú aj zmeny ukazovateľov na rodičov prvkov, ktoré sa rotujú, keďže v tomto prípade je aj táto premenná v hre.

Text

Description automatically generatedText

Description automatically generated

### Insert, Search, Delete

Podobne ako predtým, aj trieda SplayTree má tieto metódy:

* addNode()
* findNode()
* deleteNode()

Funkcia addNode() už však nie je rekurzívna ale využíva while cyklus a pomocnú funkciu newNode(), ktorá jednoducho vytvorí nový uzol stromu, uloží do neho zadané hodnoty a ukazovatele nastaví na nullptr.

AddNode() najprv pozrie, či koreň stromu je prázdny. Ak áno, pomocou newNode() vytvorí uzol a vráti ho ako koreň. Ak nie, presunie sa do while cyklu, v ktorom sa presúva po strome, kým nenájde vhodné miesto kam nový uzol vložiť. Pri tom nastávajú tri situácie:

* uzol s danou hodnotou už v strome existuje
* momentálne objavený uzol má väčšiu hodnotu ako je zadávaná
* momentálne objavený uzol má menšiu hodnotu ako je zadávaná

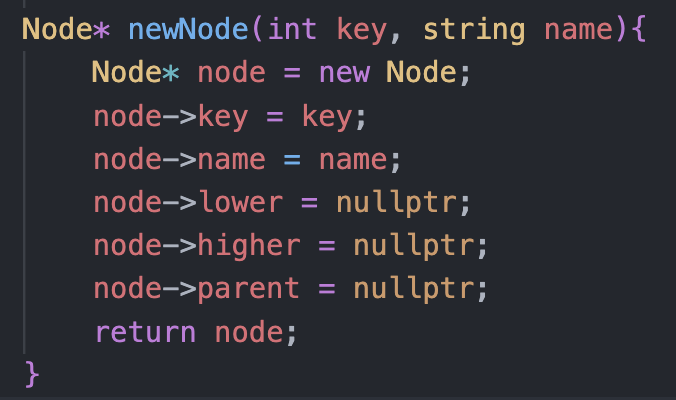
Ak je uzol s danou hodnotou už v strome, zavolá sa splayNode() a tam to končí.

Ak má momentálny uzol väčšiu alebo menšiu hodnotu ako je zadávaná, môžu nastať dve totožné situácie:

* príslušné dieťa uzla je voľné
* príslušné dieťa nie je voľné

Ak je ukazovateľ na väčšie alebo menšie dieťa voľný, najprv sa pomocou newNode() na príslušné miesto pridá nový uzol a potom sa zavolá splayNode().

Ak ukzovateľ na príslušné dieťa nie je voľný, pomocný ukazovateľ na prehľadávanie sa jednoducho posunie na príslušnú pozíciu a cyklus sa opakuje, kým sa buď neobjaví uzol sa danou hodnotou alebo sa nepridá nový uzol.





Funkcia findNode() je totožná s tou z AVL stromu uvedenou vyššie. Jediným rozdielom je, že po úspešnom nájdení uzla sa volá funkcia splayNode(), ktorá hľadaný uzol „vynesie“ na koreň stromu.

Text

Description automatically generated

Funkcia deleteNode() najprv „vynesie“ zadný uzol na koreň stromu pomocu findNode(). Ak sa v strome nenachádza uzol so zadanou hodnotou, nič sa nestane.

Text

Description automatically generated

Ak sa však nachádza, tj. bol vynesený na koreň stromu, môžu nastať tieto situácie:

* koreň stromu nemá žiadne deti, tj. je jediným uzlom stromu
* koreň stromu má buď iba ľavé alebo iba pravé dieťa
* koreň stromu má aj pravé aj ľavé dieťa

Ak nemá žiadne deti, stačí ho jednoducho vymazať, ak má iba jedno dieťa, riešenie je podobne jednoduché – do pomocnej premennej sa uloží uzol na ktorý ukazuje koreň, ukazovateľ na koreň sa posunie na svoje dieťa a uzol, ktorý je uložený v dočasnej premennej sa vymaže.

Ak má koreň obidve deti, postup je podobný ako pri mazaní uzla, ktorý má obe deti z AVL stromu. Tam sa najpv hľadal najmenší uzol z pravého podstromu uzla, v tomto prípade sa deje analogický opak – hľadá sa najväčší uzol z ľavého podstromu. Tieto prístupy majú v princípe rovnaký výsledok, v oboch prípadoch ide o nájdenie uzla, ktorého hodnota v strome vie plniť rovnakú úlohu ako hodnota toho, ktorý sa má vymazať. V každom strome som použil iný prístup na podčiarknutie tejto skutočnosti.

Po nájdení najväčšieho uzla z ľavého podstromu a uložení tohto uzla do pomocnej premennej *temp* môžu nastať dve situácie:

* najväčší uzol ľavého podstromu je dieťa uzla, ktorý sa má vymazať
* najväčší uzol ľavého podstromu nie je dieťa uzla, ktorý sa má vymazať

Pri prvej možnosti sa pravý podstrom uzla, ktorý sa má vymazať presunie na pravú stranu uzla temp. Uzol ktorý sa má vymazať sa potom uloží do ďalšej pomocnej premennej *temp2,* ukazovateľ na koreň sa posunie na svoje menšie dieťa, uzol uložený v premennej temp2 sa vymaže a funkcia vracia nový koreň.

Pri druhej možnosti sa najprv musí ešte zavolať funkcia splayNode(), ktorá vynesie uzol temp na pozíciu ľavého dieťaťa koreňa stromu, zvyšok vyzerá rovnako.