Adam Vrabeľ AIS ID: 116327

FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ SLOVENSKÁ TECHNICKÁ UNIVERZITA
Ilkovičova 2, 842 16 Bratislava 4

2022/ 2023

Dátové štruktúry a algoritmy

Zadanie č.1

Cvičiaci: **Ing. Martin Komák, PhD.**
Čas cvičení: 16:00

Vypracoval: **Adam Vrabeľ**
AIS ID: 116327

Obsah

Samovyvažovací binárny strom

Úvod

V tomto zadaní bolo mojou úlohou implementovať dve vlastné implementácie binárnych vyhľadávacích stromov (BVS), s rozdielnym algoritmom na vyvažovanie a operácie insert, search a delete. Vybral som si AVL a RedBlack Tree.

AVL TREE

V mojej implemetácii AVL stromu využívam triedu Node, ktorá reprezentuje jeden prvok stromu. Tento Node má atribúty **data (hodnota podľa ktorej je každý Node uložený, vyhľadaný alebo vymazaný)**, **height (hodnota znázorňuja výšku prvku v strome)** a ukazovatele **left, right, ancestor (ukazovateľ na rodiča)**.
V celom strome sú prvky v ľavom podstrome vždy menšie od rodiča a v pravom podstrome vždy väčšie od rodiča.
String meno_a_priezvisko je predprípravou, aby sa tento AVL strom dal využiť pri uložení napríklad študentov, kde **data** budú ich identifikačné čísla.

```
public class Node {
    public int data;
    public int height; // VÝŠKA NODEu je vzdialenosť k listu
    public Node ancestor = null;   //REFERENCIA NA PREDCHODCU (parenta)
    public Node left = null;
    public Node right = null;


    private String meno_a_priezvisko;     // DOPLNKOVÉ DÁTA
    public String getMeno_a_priezvisko() {
        return meno_a_priezvisko;
    }
    public void setMeno_a_priezvisko(String meno_a_priezvisko) {
        this.meno_a_priezvisko = meno_a_priezvisko;
    }

    //public int balanceFactor;
    // CONSTRUCTOR
    public Node(int data){
        this.data = data;
        height = 0; // novo pridaný prvok ma vždy výšku 0
        ancestor = null;
        left = null;
        right = null;
        setMeno_a_priezvisko(null);
    }
```

AVL strom je vyvážený binárny strom, čo znamená, že všetky cesty z koreňa k listom majú rovnakú dĺžku alebo sa líšia maximálne o jedna. Toto zabezpečuje rýchly prístup k dátam, pretože časová zložitosť vyhľadávania, vkladania a odstraňovania v AVL stromoch je O(log n), kde n je počet uzlov v strome.

Vkladanie

Vkladanie realizujem iteratívnou formou.
V AVL strome sa prvok (Node) najskôr pridá klasicky ako v binárnom vyhľadavacom strome. Ak strom ešte nemá hlavný prvok (root Node) tak sa hodnota priradí do rootu, ak root prvok už existuje tak sa vykonáva posúvanie ukazovateľa (ak je pridávana hodnota menšia ako aktuálny prvok idem vľavo, inak vpravo, ak je hodnota rovnaká, prvok sa nepridá), takto postupujem až pokial nedojdem na posledný prvok (ukazovateľ vľavo alebo vpravo je null) a na nájdené miesto pridám nový prvok.

Po pridaní prvku sa prepočítajú výšky všetkých dotknutých prvkov smerom od listov nahor ku koreňu od pridaneho prvku pomocou ancestorov. Ostatné výšky nieje potrebne prepočítavať, pretože sa nezmenili, zbytočne by sme algoritmus spomalili. Prepočítanie výšky realizuje funkcia **updateHeight()** a kontrolu vyvažovania realizuje funkcia **controlBalanceFator() .** Ak je to potrebné, strom sa vyváži pomocou rotácii.

Vyvažovanie

Vyvažovanie v AVL strome je proces, ktorý zabezpečuje, že strom zostane vyvážený, teda že rozdiel výšok ľavého a pravého podstromu každého Node bude maximálne 1. Ak by sme strom nevyvažovali, jeho výška by sa mohla príliš zvýšiť a časová zložitosť operácií ako vkladanie, vyhľadávanie a mazanie by sa mohla v najhoršom prípade zhoršiť na O(n), kde n je počet uzlov v strome.
Výška stromu je vzdialenosť daného uzla (Node) k listom. List je Node ktorého ukazovatele left a right sú null. Výška nulového Nodeu je -1.

```
// VRATI VÝŠKU, NEPOČÍTA JU
private int height(Node n){      // VRÁTI VÝŠKU NODEu alebo -1 pre null
    if(n == null){
        return -1;
    }
    else{
        return n.height;
    }
}
private void updateHeight(Node n){
    if (n != null){
        n.height = 1 + Math.max( height(n.left), height(n.right) );
    }
}
```

Po pridaní prvku sa volá metóda **updateHeight** ktorá vezme maximum výšky ľavého alebo pravého podstromu a pripočíta +1, to zabezpečí korektný výpočet výšky.

Následne v každom Node sa kontroluje vyváženosť (BalanceFactor). Tento BalanceFactor nám hovorí či je Node vyvážený alebo nie a ak nieje vyvážený tak nám napovie aj na ktorej strane je viac prvkov. Takto postupujeme až po koreň stromu.

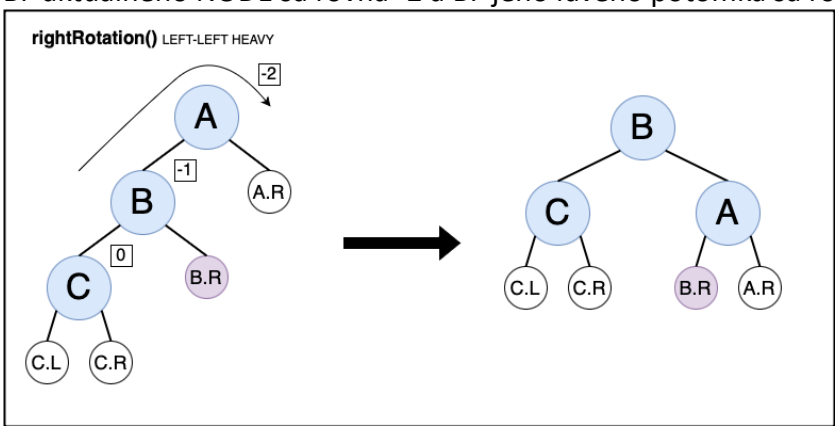
BalanceFactor (BF) v mojej implementácii sa vypočíta:
BF = výška pravého podstromu – výška ľavého podstromu.

Akceptované hodnoty BF sú {-1, 0, 1}, ak BF má iné hodnoty, je potrebne daný node vyvážiť.

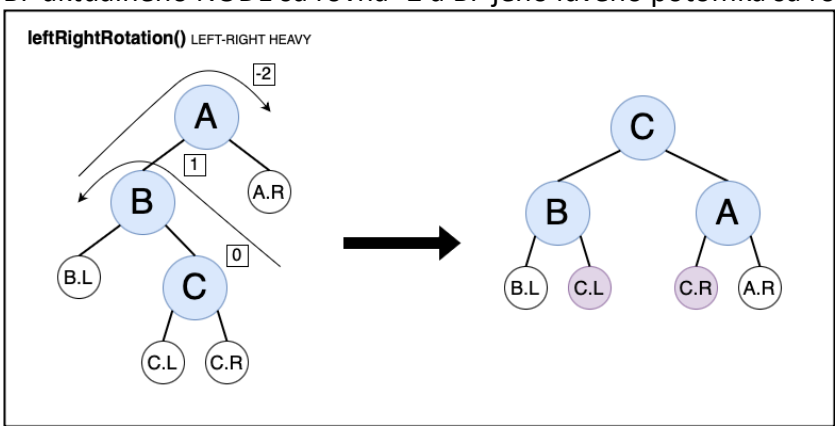
Funkcia controlBalanceFactor() zistí či je potrebné NODE vyvážiť, ak áno, zavolá sa funkcia doRebalance() ktorá zistí o aký typ nevyváženia ide a tá zavola jednotlivé rotácie.

Pri kontrole vyváženosti každého NODEu môžu nastať 4 situácie: LEFT-LEFT, LEFT-RIGHT, RIGHT-LEFT a RIGHT-RIGHT HEAVY

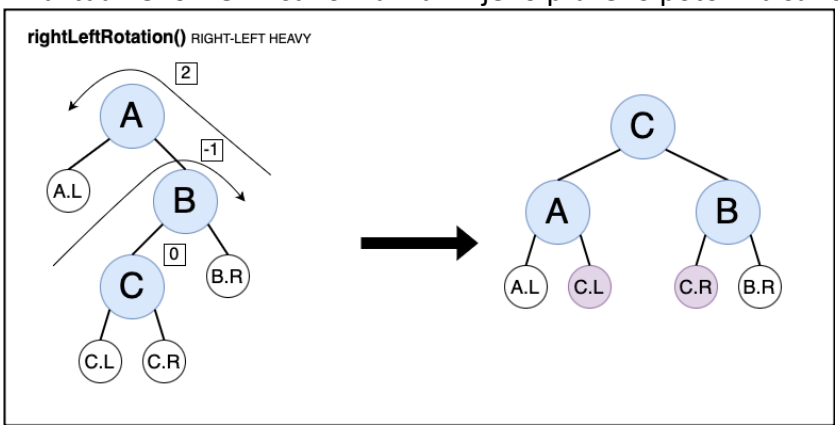
- **LEFT-LEFT HEAVY** robím rightRotation()
  BF aktuálneho NODE sa rovná -2 a BF jeho ľavého potomka sa rovná -1



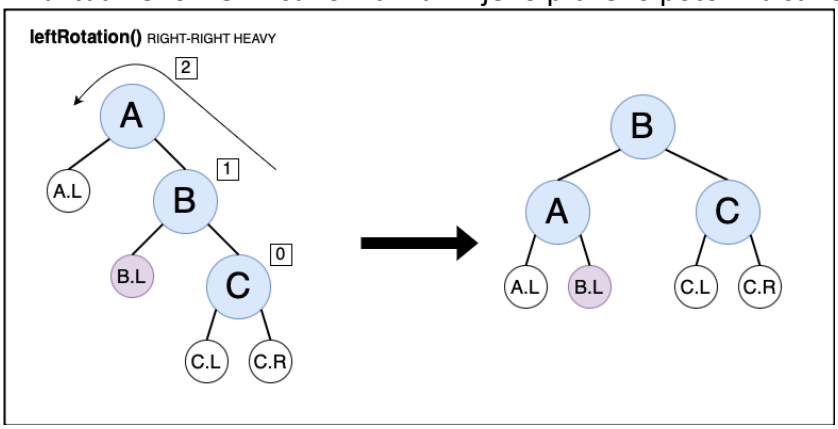
- **LEFT-RIGHT HEAVY** robím leftRightRotation()
  BF aktuálneho NODE sa rovná -2 a BF jeho ľavého potomka sa rovná 1



- **RIGHT-LEFT HEAVY** robím RightLeftRotation()
  BF aktuálneho NODE sa rovná 2 a BF jeho pravého potomka sa rovná -1



- **RIGHT-RIGHT HEAVY** robím leftRotation()
  BF aktuálneho NODE sa rovná 2 a BF jeho pravého potomka sa rovná 1



Pri každej rotácii vždy pracujeme s 3 prvkami podľa typu nevyváženosti a ich deťmi, ak nejaké majú. Do funkcií leftRotation(), rightRotation, leftRightRotation() a rightLeftRotation() sa posiela nevyvážený aktuálny NODE (local root), ktorý nespĺňa podmienku: BF = {-1, 0, 1} a tieto funkcie vratia nový local root Node, kde sú korektne vykonané rotácie a napojené deti a ancestory.

Pre rotácie leftRightRotation a rightLeftRotation som sa rozhodol vytvoriť samostatné funkcie kvôli prehľadosti a lepšiemu porozumeniu kódu, no stačilo by, že sa vykoná najskor rotácia vľavo a potom vpravo (respektíve vpravo a potom vľavo).

Časová náročnosť vyvažovania je O(n), kde n je počet vykonaných rotácií.

Vymazanie

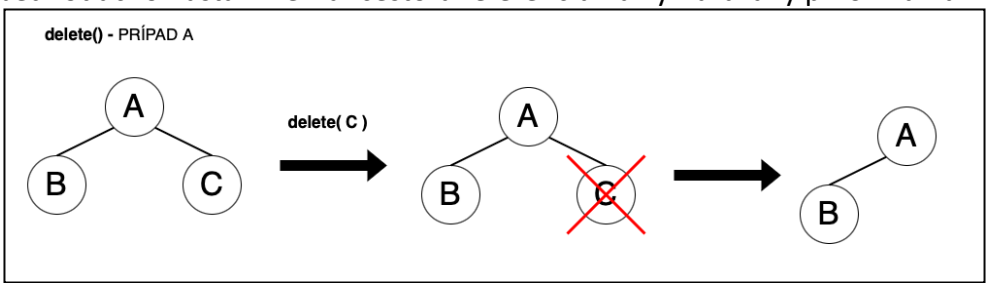
Funkcia delete() zabezpečuje vymazanie prvku. Ako prvé je potrebne zistiť či sa prvok v BVS nachádza (hľadanie vykona funkcia searchNode()).
Moju logiku vymazania mám rozdelenú na následovné časti:

- vymazávam root
- vymazávam prvok (ktorý nieje root)

V oboch prípadoch môže mať vymazávaný prvok buď žiadne deti (PRÍPAD A), jedno dieťa vľavo (PRÍPAD B1), jedno dieťa vpravo (PRÍPAD B2) alebo dieťa aj vpravo aj vľavo (PRÍPAD C).

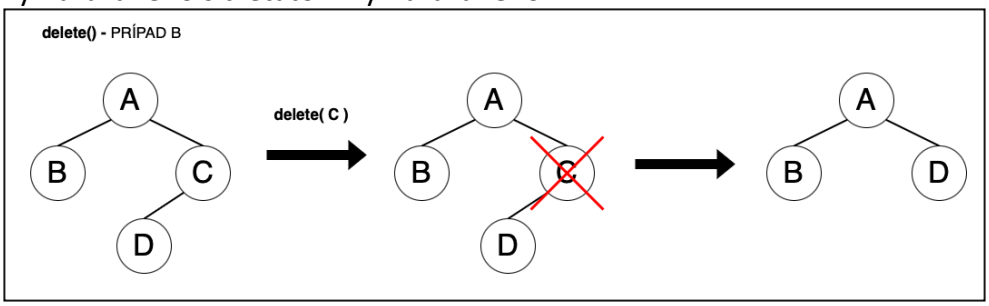
PRÍPAD A – ODSTRAŇOVANIE NODE BEZ DETÍ (LIST)

Jednoducho nastavíme z ancestora referenciu na vymazávaný prvok na null.



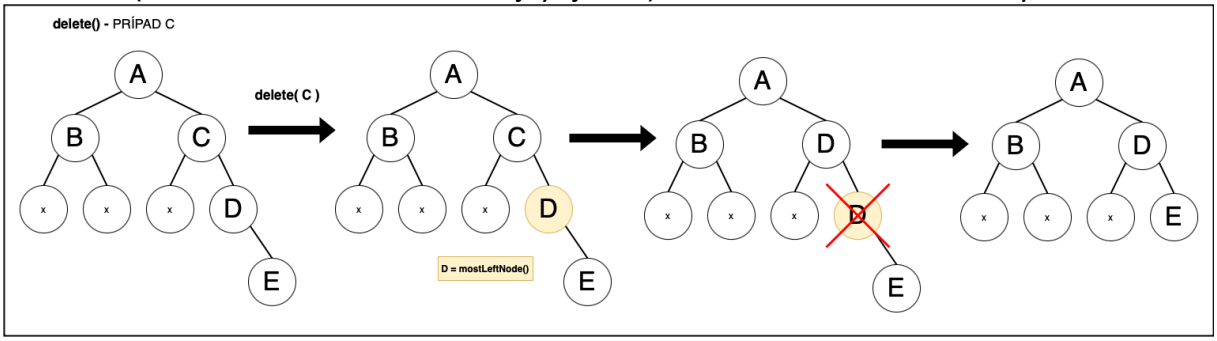
PRÍPAD B1, B2 – ODSTRÁNENIE NODE S JEDNÝM DIEŤAŤOM

Ak chceme vymazať Node s práve jedným dieťaťom, jednoducho prepojíme nadradený prvok vymazávaného s dieťaťom vymazavaného.



PRÍPAD C – ODSTRÁNENIE NODE S DVOMA DETMI

V tomto prípade nájdem najmenší prvok v pravom podstrome vymazavaného Node pomocou funkcie **mostLeftChild()**. Dáta z mostLeftChild uložím do Node ktorý vymazávam a následne vymažem mostLeftChild. Tento node nemôže mať ľavé dieťa. Takto môže nastať iba prípad A alebo B (nemá žiadne deti alebo nanajvýš jedno). Takéto mazanie už vieme vykonať.



Po vymazaní vykonám prepočítanie výšky a kontrolu BF smerom ku koreňu (iteratívne, prechádzaním cez ancestorov), ak je to potrebné strom vyvážim.

Vymazávanie je časovo náročne a to O(log n) v najhoršom prípade.

Vyhľadávanie

Funkcia search() nám zabezpečí výpis na konzolu a volá funkciu searchNode(), ktorá hľadá zadaný prvok.

Funkcia searchNode() má za úlohu vyhľadať prvok v BVS a vrátiť ukazovateľ, v prípade, že sa prvok v strome nenachádza, funkcia vráti NULL. Funkcia jednoducho prehľadáva strom od koreňa a zisťuje či je hľadaná hodnota väčšia alebo menšia ako hodnota momentálneho uzla.
Vyhľadávanie v najhoršom prípade je O(log n), kde n je počet prvkov v strome.

RED-BLACK TREE

Ako druhú implemetáciu som si vybral RedBlack Tree. Je to samovyvažovací binárny strom. V mojej implementácii RED-BLACK stromu využívam triedu RedBlackNode, ktorá reprezentuje jeden prvok stromu. Tento RedBlackNode má atribúty **data** (hodnota podľa ktorej je každý Node uložený, vyhľadaný alebo vymazaný), **color** (farba daného Node) a **ukazovatele left, right, ancestor** (ukazovateľ na rodiča).
String **meno_a_priezvisko** je predprípravou, aby sa tento RED-BLACK strom dal využiť pri uložení napríklad študentov, kde **data** budú ich identifikačné čísla.

Tento strom musí spĺňať podmienky:

- každý Node je buď ČERVENÝ alebo ČIERNY
- koreň a null prvky sú vždy čierne
- ak je Node ČERVENÝ, jeho deti musia byť ČIERNE
- všetky cesty od Node k null potomkom obsahujú rovnaký počet čiernych NODEOV (čierna výška stromu)

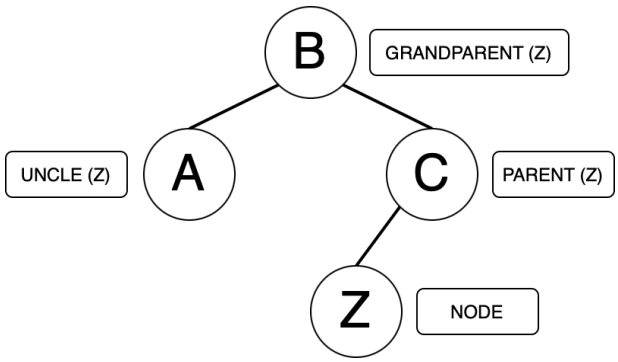
Vo vyváženom RED-BLACK strome je časová zložitosť vyhľadávania, vkladania a odstraňovania v AVL stromoch je O(log n), kde n je počet uzlov v strome.

Vkladanie

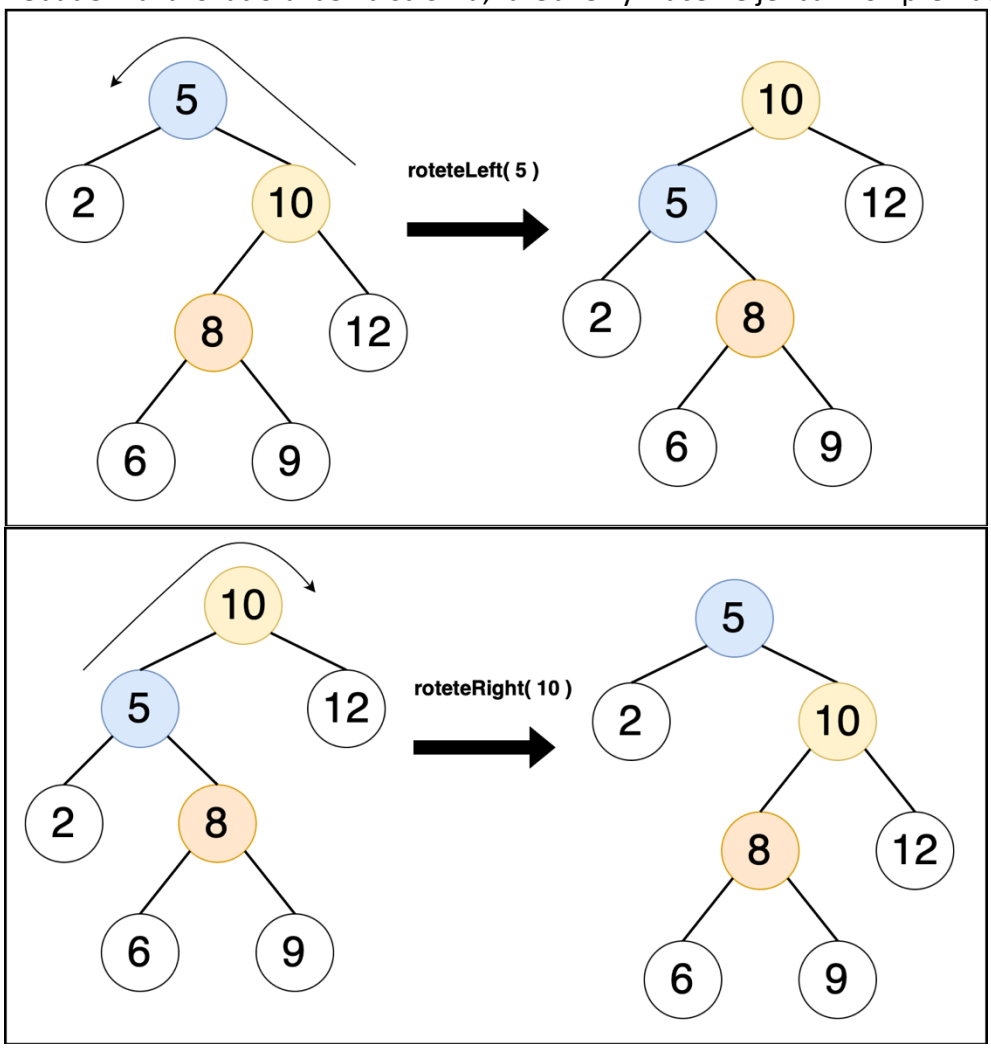
Vkladanie realizujem iteratívnou formou.
V RED-BLACK strome sa prvok (RedBlackNode) najskôr pridá klasicky ako v binárnom vyhľadavacom strome. Ak strom ešte nemá hlavný prvok (root RedBlackNode) tak sa hodnota priradí do rootu, ak root prvok už existuje tak sa vykonáva posúvanie ukazovateľa (ak je pridávana hodnota menšia ako aktuálny prvok idem vľavo, inak vpravo, ak je hodnota rovnaká, prvok sa nepridá), takto postupujem až pokial nedôjdem na posledný prvok (ukazovateľ vľavo alebo vpravo je null) a na nájdené miesto pridám nový prvok.
Po pridaní nového prvku sa volá funkcia **controlRedBlack()**, ktorá kotroluje či neboli porušené pravidlá RED-BLACK stromu, ak je to potrebné, napraví ich.

Vysvetlenie pojmov:

- **ancestor** = rodič (parent) prvku Z
- **grandparent** = rodič rodiča prvku Z = ancestor.ancestor (parent.parent)
- **uncle** = strýko prvku = dieťa grandparenta na opačnej strane ako novy node Z



V RED-BLACK strome poznáme 2 typy rotácií: **rotateLeft** a **rotateRight** (pre túto ukážku nebudem uvažovať ofarbenia stromu, farebné vyznačenie je iba kvôli prehľadnosti)

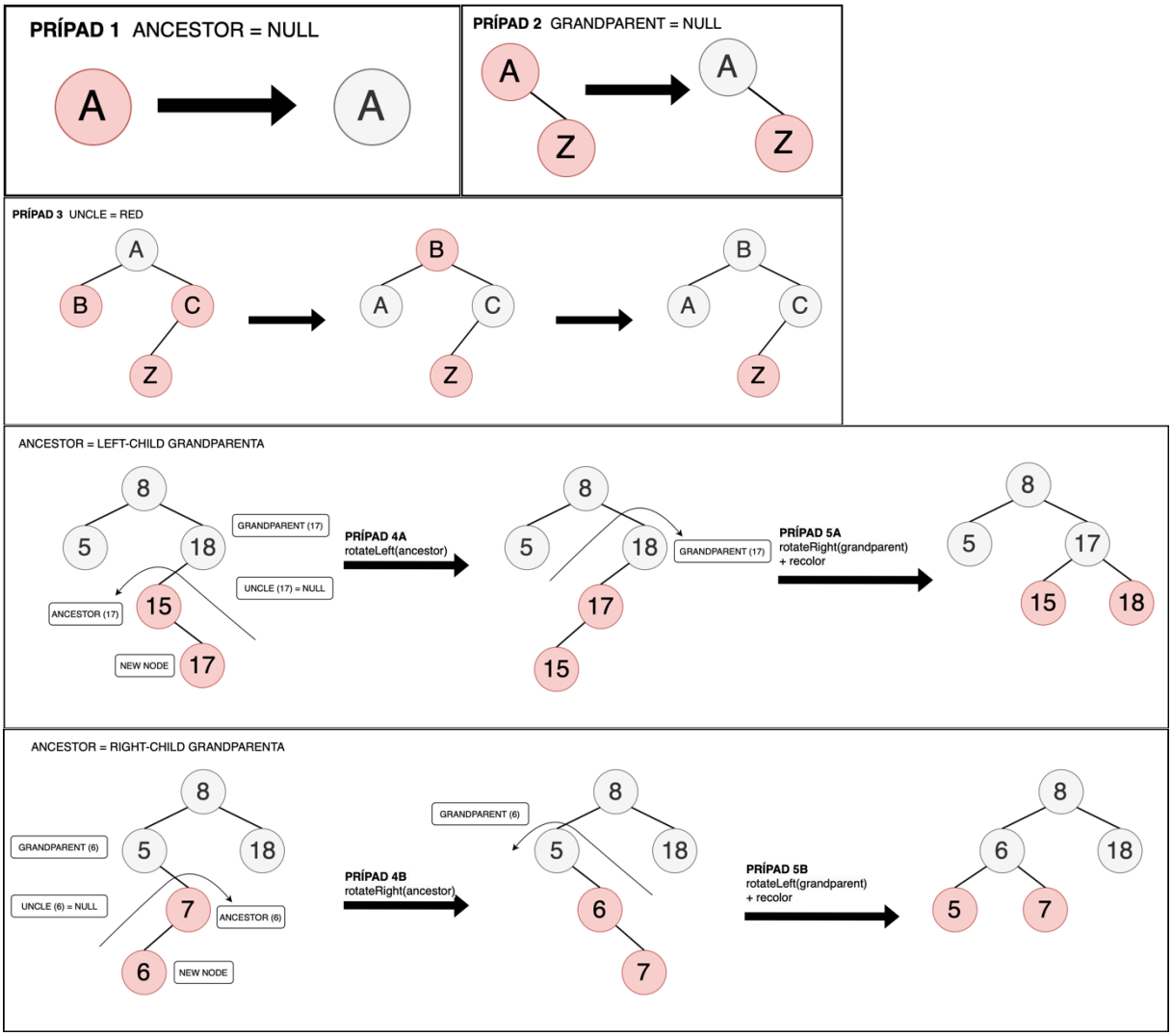


Po pridaní prvku sa prvok zafarbí na ČERVENO.
A volá sa funkcia controlRedBlack, ktorá rotáciami a ofarbeniami opraví porušenie pravidiel v strome.

Vyvažovanie

Vyvažovanie kontroluje funkcia **controlRedBlack()**. Pri pridávaní nového prvku môžu nastať tieto prípady narušenia pravidiel:

- **PRÍPAD 1: sme v roote**, prefarbíme root na čierno
- **PRÍPAD 2: ancestor (parent) = root**, root prefarbím na čierno
- **PRÍPAD 3: uncle = RED**, prefarbim **ancestor, grandparent, uncle**, **controlRedBlack**(grandparent) – môžu nastať porušenia pravidiel
- **PRÍPAD 4: uncle = BLACK** (TRIANGLE), rotujem **ancestor** (parent) do opačnej strany ako je uložený Node
- **PRÍPAD 5: uncle = BLACK** (LINE), rotujem **grandparent** do opačnej strany ako je uložený Node + **prefarbenie ancestor** (parent) a **grandparent**



Vymazanie

Funkcia **deleteNode()** zabezpečuje vymazanie prvku. Ako prvé je potrebne zistiť či sa prvok v BVS nachádza (hľadanie vykona funkcia searchNode()).

Najskôr postupujeme podobne ako vo vymazávaní pri AVL strome:

- ak Node, ktorý odstráňujeme **nemá deti**, jednoducho ho odstránime
- ak Node, ktorý odstráňujeme **má práve jedného potomka**, Node odstránime a prepojíme ancestora s potomkom
- ak Node, ktorý odstráňujem **má dvoch potomkov**, nájdem najmenší prvok v pravom podstrome vymazavaného Node pomocou funkcie mostLeftChild(). Dáta z mostLeftChild uložím do Node ktorý vymazávam a následne vymažem mostLeftChild. Tento node, nemôže mať ľavé dieťa. Takto môže nastať prípad, že nemá žiadne deti alebo nanajvýš jedno. Takéto mazanie už vieme vykonať.

Po vymazaní prvku je potrebné skontrolovať pravidlá stromu, v prípade potreby ich opraviť rotáciami a prefarbeniami.
Aby som to dosiahol, musím si zapamätať farbu odstráneného uzla a ktorý uzol som posunul vyššie.

- ak je **vymazaný uzol červený**, nemôžeme porušiť žiadne pravidlo.
- ak je však **vymazaný uzol čierny**, je zaručené, že sme porušili pravidlo a je potrebné skontrolovať a opraviť porušenia, o to sa stará funkcia **fixAfterDelete()**

Súrodenec (SIBLING) je druhé dieťa ancestora.
Vonkajší synovec je dieťa súrodenca (na „vonkajšej“ strane stromu).

Možné prípady aké môžu nastať:

- **PRÍPAD 1:** sme v roote, koniec rekurzie
- **PRÍPAD 2:** súrodenec je červený
- **PRÍPAD 3**: súrodenec je čierny a má dve čierne deti, ancestor je červený
- **PRÍPAD 4**: súrodenec je čierny a má dve čierne deti, ancestor je čierny
- **PRÍPAD 5**: súrodenec je čierny a má aspoň jedno červené dieťa, „vonkajší synovec“ je čierny
- **PRÍPAD 6**: súrodenec je čierny a má aspoň jedno červené dieťa, „vonkajší synovec“ je červený

Vyriešenie týchto prípadov nám zabezpečí, že strom bude aj naďalej vyvážený a budú splnené všetky vlastnosti RED-BLACK stromu.

Vyhľadávanie

Funkcia **search()** nám zabezpečí výpis na konzolu a volá funkciu searchNode(), ktorá hľadá zadaný prvok.

Funkcia **searchNode()** má za úlohu vyhľadať prvok v BVS a vrátiť ukazovateľ, v prípade, že sa prvok v strome nenachádza, funkcia vráti NULL. Funkcia jednoducho prehľadáva strom od koreňa a zisťuje či je hľadaná hodnota väčšia alebo menšia ako hodnota momentálneho uzla.
Vyhľadávanie v najhoršom prípade je O(log n), kde n je počet prvkov v strome.

Hashovanie

Úvod

Ako zadanie sme mali vytvoriť dve vlastné implementácie hašovacej tabuľky s riešením kolízií podľa vlastného výberu. Vybral som si riešenie kolízii formou **linear probing (otvorené adresovanie)** a **chaining (uzavreté adresovanie).**

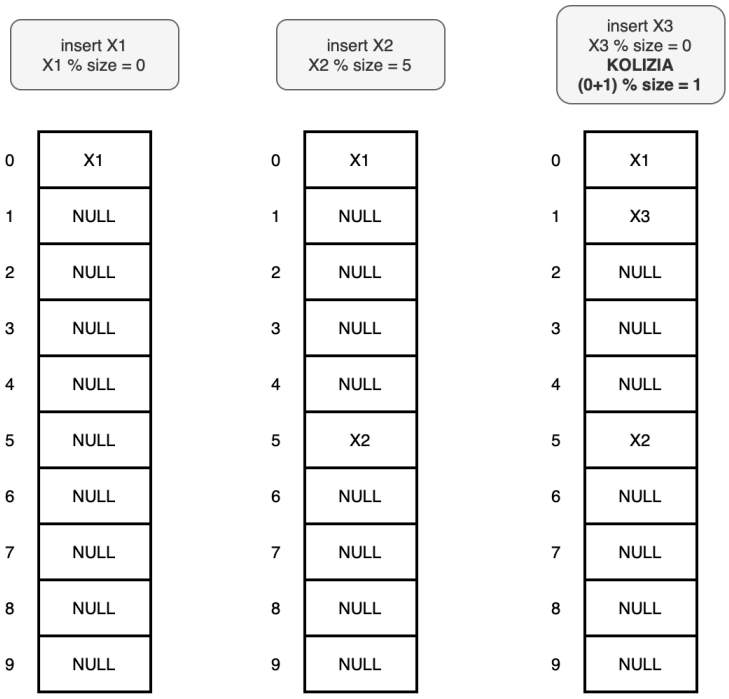
Hašovacia tabuľka je dátova štruktúra, ktorá asociuje kľúče s hodnotami. Pracuje vďaka transformácii kľúča hashovacou funkciou na hash číslo (index), ktoré tabuľka používa na nájdenie požadovanej hodnoty.

Hashovacie tabuľky sú veľmi efektívne, čo sa týka časovej náročnosti pre operácie vkladanie, hľadanie a vymazávanie. V ideálnom prípade sú tieto operácie vyriešené v konštantnom čase O(1), ale skutočná časová zložitosť závisí od týchto faktorov:

- kvalita hashovacej funkcie (pri hashovacej funkcii je želané, aby vytvorila čo najmenej kolízií)
- počet vzniknutých kolízií
- veľkosť tabuľky (nieje dobré, ak je tabuľka príliš plná alebo príliš prázdna)
- použitý algoritmus na riešenie kolízií

Open adressing (linear probing) hash

V tejto implementácií sú prvky s rovnakým indexom ukladané priamo do hashovacej tabuľky (ak nastane kolízia uloží sa na najbližší voľný index).



Trieda Item nesie pár dôležitých informácii o vkladanom prvku. Jeho **hodnotu (value)** vo forme Stringu, **kľúč** ktorý sa vypočíta funkciou getNumberFromString(). Ak pošleme kľúč do hashovacej funkcie, získame **index** v poli, kam máme uložiť/ hľadať daný prvok.

Pri hashovacej tabuľke nás zaujíma veľkosť **tabuľky (size)**, **počet vložených prvkov (filled)** a samotné **pole Itemov.**
String **meno_a_priezvisko** je predprípravou, aby sa táto hashovacia tabuľka dala využiť pri uložení napríklad študentov, kde value budú ich mená.

```
class Item {

    public String value;

    int key;      // VYSLEDOK V HASH FUNKCII

    int vek_osoby; // DOPLNKOVÉ DÁTA
    public int getVek_osoby() {
        return vek_osoby;
    }
    public void setVek_osoby(int vek_osoby) {
        this.vek_osoby = vek_osoby;
    }

    public Item(String value, int key){
        this.value = value;
        this.key = key;
    }
```

Hash Metóda

Ako hashovaciu metódu som zvolil výpočet **kľúč % veľkost.** Je to jedna z najpoužívanejších hashovacích funkcií. Výpočet modulo je rýchly a jednoznačný. Táto funkcia vždy zabezpečí, že nám vráti index, ktorý patrí do tabuľky. Pre naše účely testovania postačuje.

Kľúč zo Stringu si vyrábam pomocou java funkcie hashCode() a absolútnej hodnoty.

FillFactor

Po každom vložení táto funkcia prepočíta faktor naplnenia pomocou údajov z hashovacej tabuľky **filled/size**. Faktor naplnenia udržiavam podľa odporúčaní z prednášok pod 0.5. V prípade, že faktor naplnenia presiahne túto hodnotu, tabuľku zväčším dvojnásobne.

Vkladanie

Parameter tejto funkcie je String, ktorý vkladáme. Funkcia vytvorí kľúč (key) cez funkciu getNumberFromString. Tento kľúč je odoslaný do hashovacej funkcie, ktorá nám vráti miesto pre uloženie Itemu v poli. Na prázdne miesto vloží Item.
Očakávaná časová zložitosť funkcie je O(1), keďže pomocou indexu z hash() priamo pristúpim k miestu vloženia.
Ak miesto nieje voľné nastane kolízia.
V tejto implemetácii riešim kolízie formou **LINEAR PROBING.** Pri narazení na kolíziu (prvok so želaným indexom je v tabuľke už obsadený) prehľadávam tabuľku dopredu (posúvam index o 1). Na prvé voľné miesto vložím prvok. Ak sa narazí na koniec tabuľky, prehľadávanie pokračuje od začiatku tabuľky.
Najhorší prípad časovej zložitosti je O(n), kde n je počet prvkov, o ktoré je nutné iterovať. (až pokiaľ nenarazím na null)

Pri nutnosti zväčšovania tabuľky je nutné prejsť celou tabuľkou a prehashovať všetky prvky, zložitosť resize() je O(n), kde n je veľkosť tabuľky.

Taktiež je ošetrená aj duplicita prvkov, rovnaký String je možné vložiť práve raz. Po každom vložení inkrementujem počet vložených prvkov (filled) a kontrolujem fillFactor.

Voľné miesto: ak pole s daným indexom == null alebo value=="#DELETED#"

Nevýhodou lineárneho probingu je to, že pri veľmi zaplnenej tabuľke môže dôjsť k mnohým kolíziám a prehľadávanie môže byť veľmi pomalé. Tento problém som vyriešil jednoducho, kontrolou faktoru naplnenia tabuľky (fillFactor), ktorý som opisoval vyššie.

Vymazávanie

Pre vymazanie prvku je potrebné zadať jeho value (String value) podľa ktorého sa prvok do tabuľky vložil. Funkcia searchItem nájde v tabuľke vymazávaný prvok, ak taký prvok v tabuľke existuje, zmení jeho value na „#DELETED# a zmenší počítadlo naplnenia (filled).
Pri vymazávaní taktiež kontrolujem faktor naplnenia, ak klesne pod 0.2 tak tabuľku zmenším o polovicu a prehashujem prvky rovnako ako pri zväčšovaní.
Číslo 0.2 som zvolil z dôvodu, že ak by sme príliž často zmenšovali tabuľku, tak by stačilo pridať málo prvkov a opäť by sa zväčšovala, čo by značne ovplyvnilo časovú efektivitu operácií ako insert a delete.

Pri vymazaní prvku bez nutnosti zmenšovania tabuľky je očakávaná časová zložitosť O(1), keďže stači iba zmeniť value. Pri nutnosti zmenšovania tabuľky je nutné prejsť celou tabuľkou a prehashovať všetky prvky, **zložitosť resize() je O(n), kde n je veľkosť tabuľky.**

Vyhľadávanie

Funkcia search() nám zabezpečí výpis na konzolu a volá funkciu searchItem(), ktorá hľadá zadaný prvok.

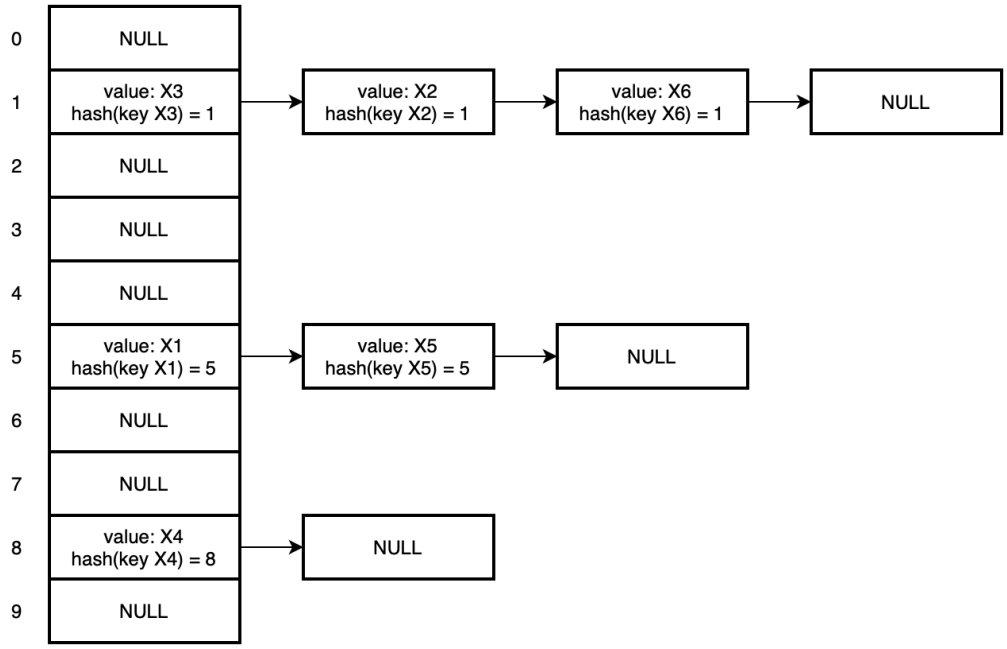
Funkcia searchItem() má za úlohu vyhľadať prvok v hashovacej tabuľke a vrátiť ukazovateľ, v prípade, že sa prvok v tabuľke nenachádza, funkcia vráti NULL.
Funkcia si vypočíta key z value ktorú dostane (ak key nemá dopočíta si ho z value), následne použitím hashovacej funkcie prevedie key na index a hľadá prvok.
Očakávaná časová zložitosť O(1), keďže pomocou indexu z hash() priamo pristúpime k miestu v tabuľke a zistíme či sa tam hľadaný prvok nachádza.

Ak na hľadanom indexe nieje null no prvok nieje rovnaký ako hľadaný, tak postupujem v hľadaní vždy o 1 index dopredu až pokiaľ nenarazím na null.
V tomto prípade najhorší prípad, ktorý môže nastať je O(n). Kde n je počet prvkov až po najbližší null.

Ak bolo hľadanie úspešne tak vrátim referenciu na aktuálny index, inak null.

Closed adressing (chaining) hash

V tejto implementácii sú prvky s rovnakým indexom ukladané do spojeného zoznamu. Prvky v tomto spájanom zozname majú rovnaký hash.



Trieda Item nesie pár dôležitých informácii o vkladanom prvku. Jeho **hodnotu (value)** vo forme Stringu, **kľúč** ktorý sa vypočíta funkciou getNumberFromString(). Ak pošleme kľúč do hashovacej funkcie, získame **index** v poli, kam máme uložiť/ hľadať daný prvok. **Ukazovatele before a next**, ktoré slúžia pri kolíziach, kde vytváram spájany zoznam. Pri hashovacej tabuľke nás zaujíma **veľkosť tabuľky (size)**, **počet vložených prvkov (filled)** a samotné **pole Itemov**.
Číslo **vek_osoby** je predprípravou, aby sa táto hashovacia tabuľka dala využiť pri uložení osôb podľa veku.

Hash Metóda

Ako hashovaciu metódu som zvolil výpočet **kľúč % veľkost.** Je to jedna z najpoužívanejších hashovacích funkcií. Výpočet modulo je rýchly a jednoznačný. Táto funkcia vždy zabezpečí, že nám vráti index, ktorý patrí do tabuľky. Pre naše účely testovania postačuje.

Kľúč zo Stringu si vyrábam pomocou java funkcie hashCode() a absolútnej hodnoty.

FillFactor

Po každom vložení táto funkcia prepočíta faktor naplnenia pomocou údajov z hashovacej tabuľky **filled/size**. Faktor naplnenia udržiavam podľa odporúčaní z prednášok pod 0.5. V prípade, že faktor naplnenia presiahne túto hodnotu, tabuľku zväčším dvojnásobne.

Vkladanie

Parameter tejto funkcie je String, ktorý vkladáme. Funkcia vytvorí kľúč (key) cez funkciu getNumberFromString. Tento kľúč je odoslaný do hashovacej funkcie, ktorá nám vráti miesto pre uloženie Itemu v poli. Na prázdne miesto vloží Item. Ak miesto nieje voľné nastane kolízia. V tejto implemetácii riešim kolízie formou **CHAINING.** Pri narazení na kolíziu (prvok so želaným indexom je v tabuľke už obsadený) na danom indexe vytvorím spájaný zoznam. Taktiež je ošetrená aj duplicita prvkov, rovnaký String je možné vložiť práve raz. Po každom vložení inkrementujem počet vložených prvkov (filled) a kontrolujem fillFactor.

Toto riešenie kolízií by malo byť veľmi efektívne pri nastavení naplnenia 0.5. No bude náročnejšie na pamäť nakoľko pri kolízii vždy vytvorím miesto pre nový prvok.

Očakávaná časová zložitosť funkcie je O(1), keďže pomocou indexu z hash() priamo pristúpim k miestu vloženia. V prípade obsadenosti miesta a následne nutnej iterácie cez spájaný zoznam je najhorší prípad časovej zložitosti O(n), kde n je počet prvkov v spájanom zozname.

Pri nutnosti zväčšovania tabuľky je nutné prejsť celou tabuľkou a prehashovať všetky prvky, zložitosť resize() je O(n), kde n je veľkosť tabuľky.

Vymazávanie

Pre vymazanie prvku je potrebné zadať jeho value (String value) podľa ktorého sa prvok do tabuľky vkladal. Funkcia searchItem nájde v tabuľke vymazávaný prvok, ak taký prvok v tabuľke existuje tak môžu nastať tieto prípady:

- vymazáva sa prvok ktorý je jediný na danom indexe
- vymazáva sa prvok na začiatku spájaného zoznamu
- vymazáva sa prvok v spájanom zozname
- vymazáva sa prvok na konci spájaného zoznamu

Po korektnom odstránení prvku zmenším počítadlo naplnenia (filled).

Pri vymazávaní taktiež kontrolujem faktor naplnenia, ak klesne pod 0.2 tak tabuľku zmenším o polovicu a prehashujem prvky rovnako ako pri zväčšovaní.
Číslo 0.2 som zvolil z dôvodu, ak by sme príliž často zmenšovali tabuľku, tak by stačilo pridať málo prvkov a opäť by sa zväčšovala, čo by značne ovplyvnilo časovú efektivitu operácií ako insert a delete.

Pri vymazaní prvku bez nutnosti zmenšovania tabuľky je očakávaná časová zložitosť O(n), kde n je počet prvkov v spájanom zozname. Pri nutnosti zmenšovania tabuľky je nutné prejsť celou tabuľkou a prehashovať všetky prvky, zložitosť resize() je O(n), kde n je veľkosť tabuľky.

Vyhľadávanie

Funkcia search() nám zabezpečí výpis na konzolu a volá funkciu searchItem(), ktorá hľadá zadaný prvok.

Funkcia searchItem() má za úlohu vyhľadať prvok v hashovacej tabuľke a vrátiť ukazovateľ, v prípade, že sa prvok v tabuľke nenachádza, funkcia vráti NULL.
Funkcia si vypočíta key z value ktorú dostane (ak key nemá, dopočíta si ho z value), následne použitím hashovacej funkcie prevedie key na index a hľadá prvok.
Očakávaná časová zložitosť je O(1), keďže pomocou indexu z hash() priamo pristúpime k miestu v tabuľke a zistíme či sa tam hľadaný prvok nachádza.
Ak na hľadanom indexe nieje null no prvok nieje rovnaký ako hľadaný, tak prehľadávam spájaný zoznam. **V tomto prípade najhorší prípad, ktorý môže nastať je O(n). Kde n je počet prvkov v spájanom zozname.**
Ak bolo hľadanie úspešne tak vrátim referenciu na aktuálny prvok, inak null.

TESTOVANIE

Testy pri všetkých typoch dátových štruktúr boli vykonáné v intervaloch 10K, 100K a 1M prvkov pre každú metódu. V kóde je možné zmeniť premennú **testNumber** a takto nastaviť interval pre testovanie.
Testovanie bolo vykonané v IntelliJ IDEA, na počítači s APPLE M1 PRO čipom a 16GB RAM. V metódach nevypisujem žiadne hlášky, nakoľko to ovplyvňuje čas testovania.

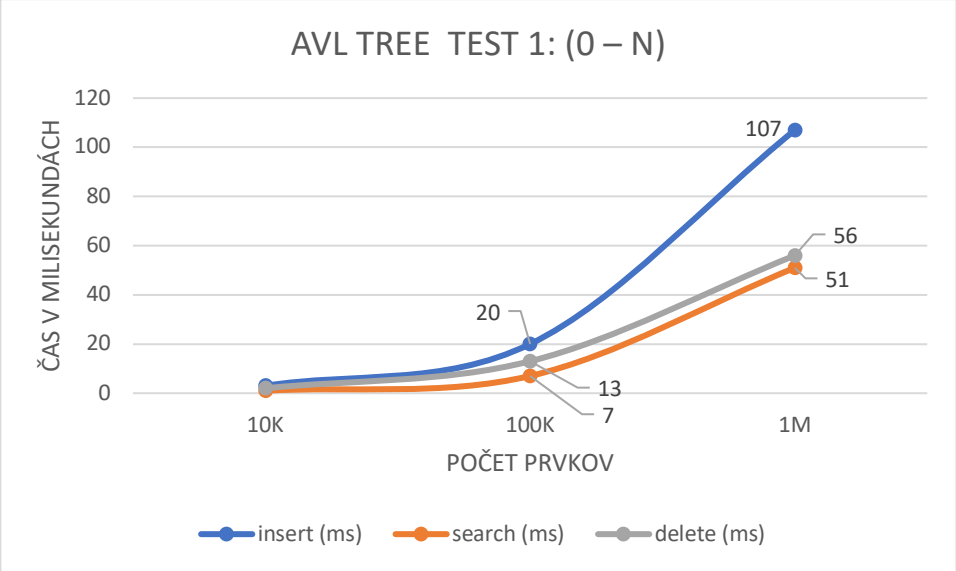
PRE AVL a RED-BLACK STROMY mám 3 typy testovania (rôzne poradie prvkov).
TEST 1: prvky od 0 do N
TEST 2: prvky od N do 0
TEST 3: vygenerujem si pole o veľkosti N náhodných prvkov
Pre každý test sa odmeria metóda insert, search a delete pre všetky prvky. Pre TEST 3 sa rovnaké pole použije pre všetky metódy.

V oboch HASHOVACÍCH tabuľkách si vopred vygenerujem pole náhodných Stringov aby mi to neovplyvnilo čas testovania. Následne vykonám vloženie, vyhľadanie a vymazanie celého poľa náhodne vygenerovaných stringov. Pre každú funkciu meriam čas osobitne.

AVL TEST

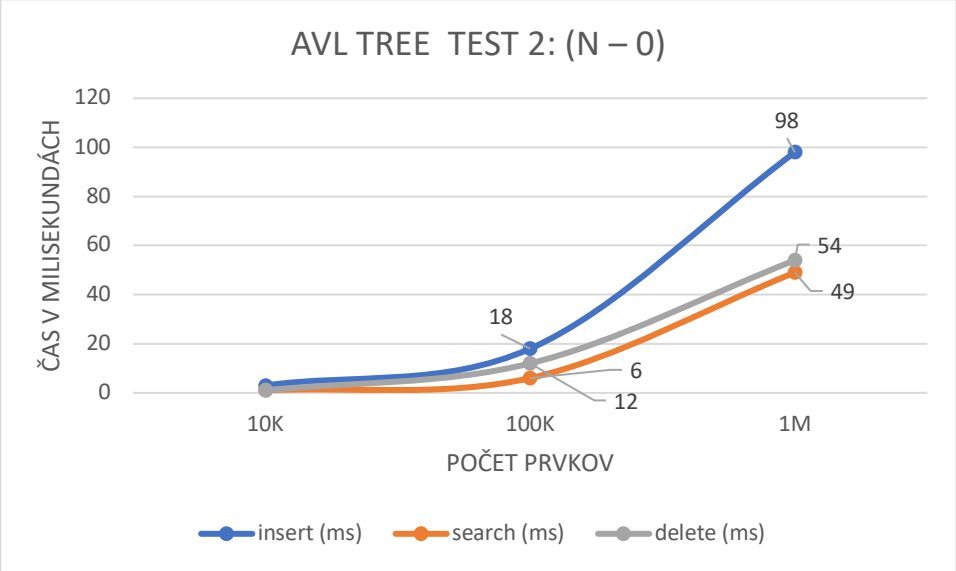
AVL TREE TEST 1: (0 – N)

POČET PRVKOV	**10K**	**100K**	**1M**
insert (ms)	3	20	107
search (ms)	1	7	51
delete (ms)	2	13	56



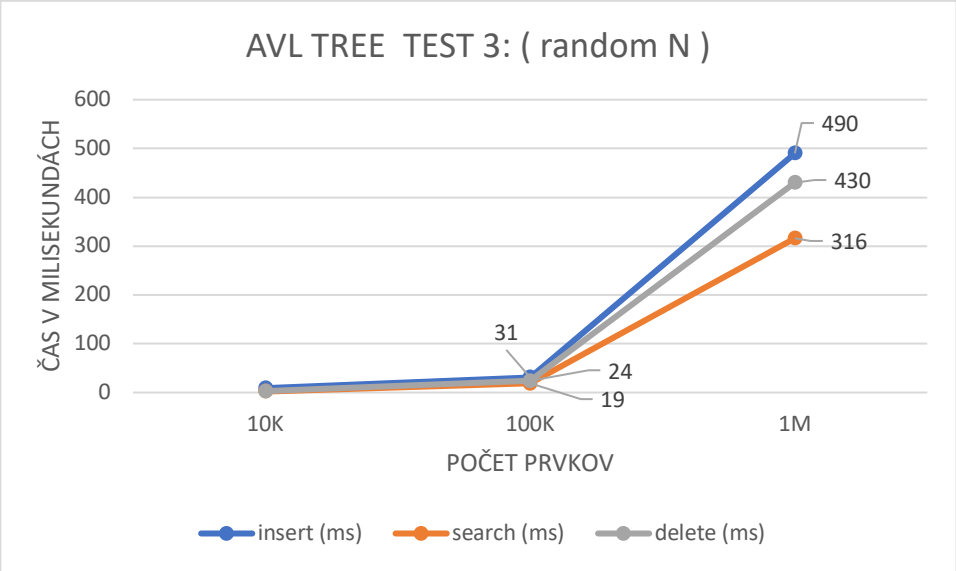
AVL TREE TEST 2: (N – 0)

POČET PRVKOV	**10K**	**100K**	**1M**
insert (ms)	3	18	98
search (ms)	1	6	49
delete (ms)	1	12	54



AVL TREE TEST 3: (random N)

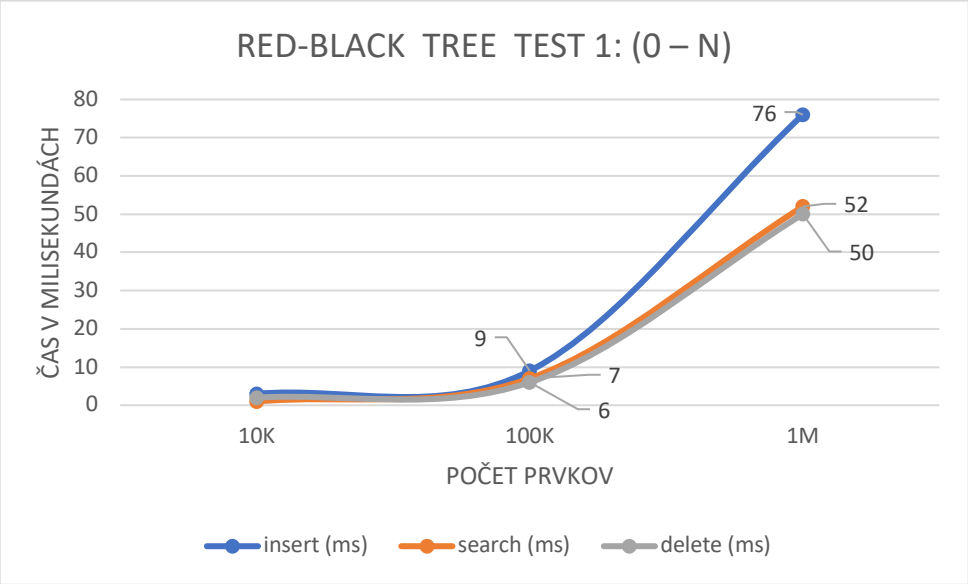
POČET PRVKOV	**10K**	**100K**	**1M**
insert (ms)	9	31	490
search (ms)	2	19	316
delete (ms)	3	24	430



RED–BLACK TEST

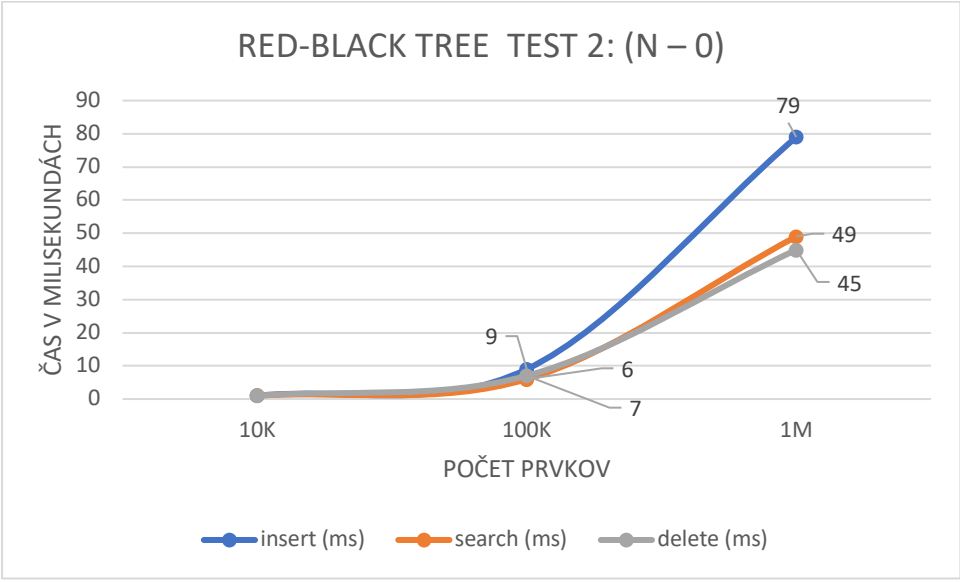
RED-BLACK TREE TEST 1: (0 – N)

POČET PRVKOV	**10K**	**100K**	**1M**
insert (ms)	3	9	76
search (ms)	1	7	52
delete (ms)	2	6	50



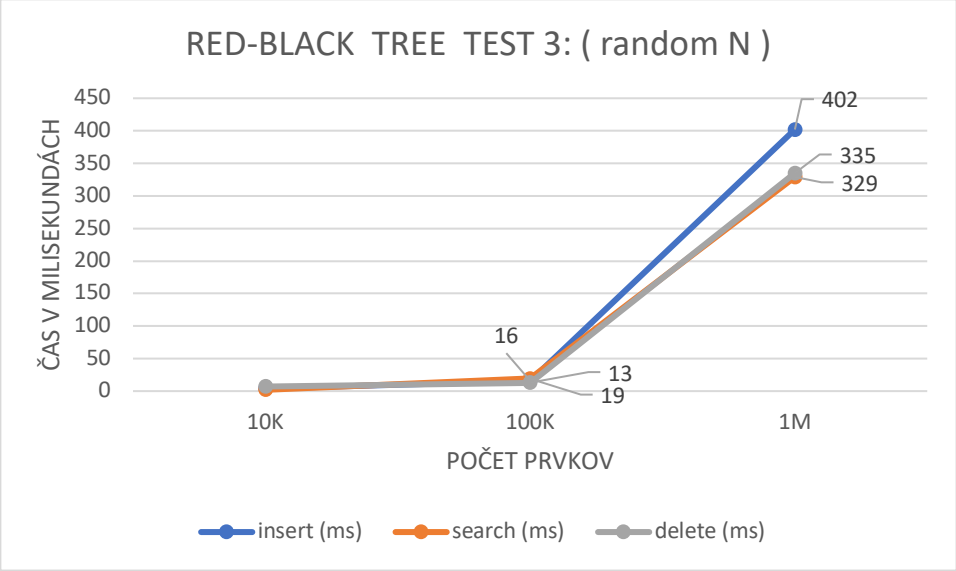
RED-BLACK TREE TEST 2: (N – 0)

POČET PRVKOV	**10K**	**100K**	**1M**
insert (ms)	1	9	79
search (ms)	1	6	49
delete (ms)	1	7	45



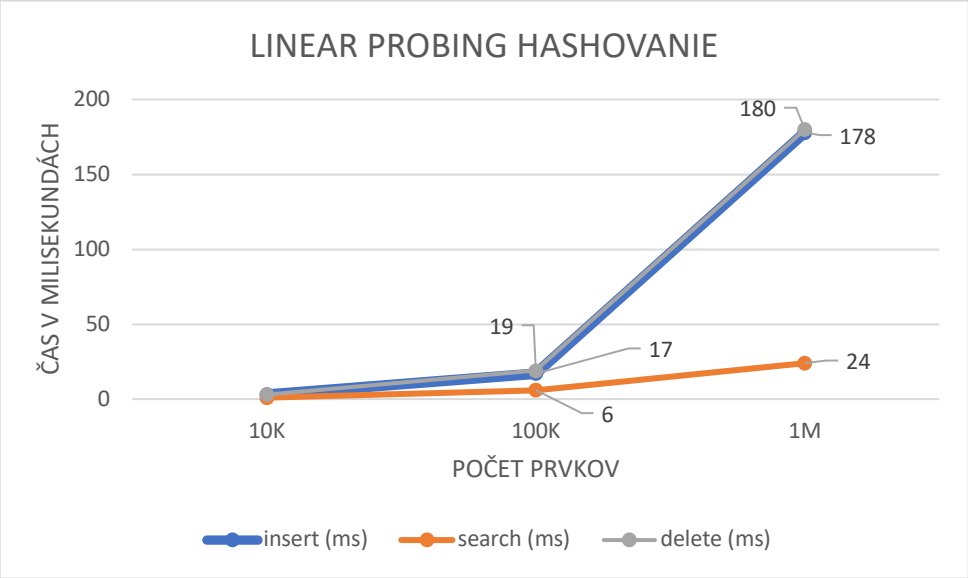
RED-BLACK TREE TEST 3: (random N)

POČET PRVKOV	**10K**	**100K**	**1M**
insert (ms)	4	16	402
search (ms)	2	19	329
delete (ms)	7	13	335



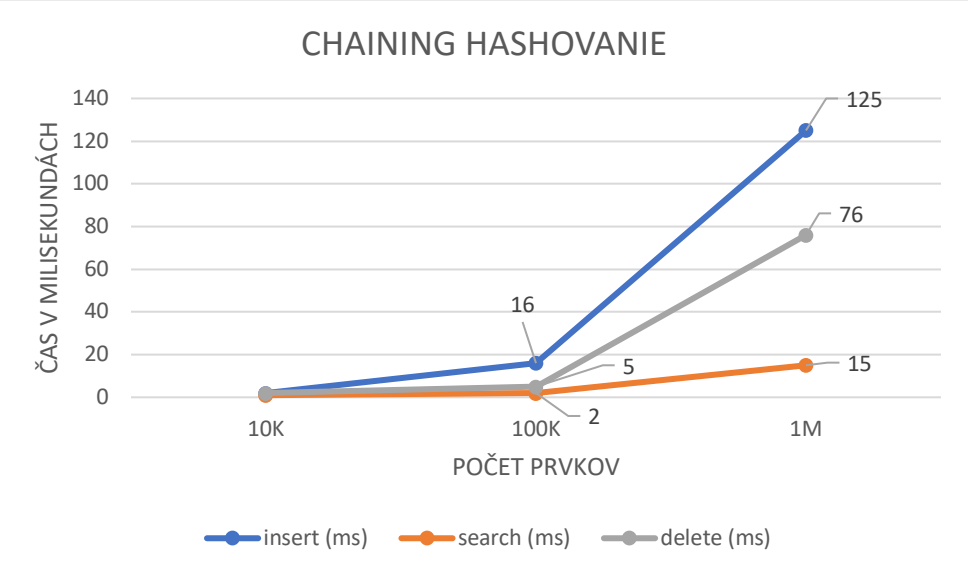
LINEAR PROBING HASHOVANIE

POČET PRVKOV	10K	100K	1M
insert (ms)	3	17	178
search (ms)	1	6	24
delete (ms)	3	19	180



CHAINING HASHOVANIE

POČET PRVKOV	10K	100K	1M
insert (ms)	2	16	125
search (ms)	1	2	15
delete (ms)	2	5	76



Mojou úlohou bolo implementovať 4 typy dátových štruktúr (AVL strom, R-B strom, 2 hashovacie tabuľky) z hľadiska efektivity operácií insert, search, delete v rozličných situáciách.

Výsledky testov možno vidieť v tabuľkách vyššie. Časy sú priemerom z 5 testovaní a sú uvedené v milisekundách.

ČASY PRI AVL A RED-BLACK STROMOCH sú pri testoch 1 a 2 približne rovnaké pre všetky operácie, no pri teste 3 (náhodné čísla) čas všetkých operácií je o niečo vyšší.

Zväčšujúcim sa počtom prvkov je RED-BLACK strom o niečo rýchlejší ako AVL strom.

Hashovacia tabuľka s riešením kolízií chaining je rýchlejšia oproti linear probing, no je náročnejšia na pamäť, keďže pri každej kolízii je potrebne vytvoriť miesto na nový prvok v spájanom zozname.
Obe tabuľky sú veľmi rýchle, ak udržiavam faktor naplnenia pod 0.5, zvačšovaním faktoru naplnenia sa výkonnosť oboch tabuliek exponenciálne zhoršuje.
Hashovacie tabuľky veľmi ovplyvňuje nastavenie faktoru naplnenia a samotné prehashovanie prvkov pri zväčšení tabuľky, nakoľko je potrebné prejsť celú starú tabuľku a prehashovať všetky prvky do novej. Ak sa tento proces deje vtedy ako má, aj tabuľky sú veľmi efektívne.

V mojom testovaní pri 1M prvkoch je najlepšie použiť chaining hashovaciu tabuľku, no všetko závisí od toho, na čo chcem tieto dátové štruktúry použiť a aké mám možnosti (vypočtový výkon/ veľkosť úložiska).

Rozdiely v časoch boli stále v rámci milisekúnd, čo nepovažujem za nejak závažné, preto si myslím že je jedno akú z implementácií zvolím.