
SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Vyhľadávanie v dynamických množinách

ZADANIE 1

Mária Matušisková
DSA
Cvičiaci: Ing. Martin Komák, PhD. (Po 16:00)

AID ID: 116248
2023

Vyhľadávanie v dynamických množinách

ZADANIE 1

Študijný program: Informatika, konverzný

Študijný odbor: 9.2.1 Informatika

Školiace pracovisko: Fakulta informatiky a informačných technológií

2023

Mária Matušisková

Zadanie 1 – Vyhľadávanie v dynamických množinách

Existuje veľké množstvo algoritmov, určených na efektívne vyhľadávanie prvkov v dynamických množinách: binárne vyhľadávacie stromy, viaceré prístupy k ich vyvažovaniu, hašovacie tabuľky a viaceré prístupy k riešeniu kolízí. Rôzne algoritmy sú vhodné pre rôzne situácie podľa charakteru spracovaných údajov, distribúcie hodnôt, vykonávaným operáciám, a pod. V tomto zadaní máte za úlohu implementovať a porovnať tieto prístupy.

Vašou úlohou v rámci tohto zadania je implementovať a následne porovnať 4 implementácie dátových štruktúr z hľadiska efektivity operácií **insert**, **delete** a **search** v rozličných situáciach:

- (3 body) Vlastnú implementáciu binárneho vyhľadávacieho stromu (BVS) s ľubovoľným algoritmom na vyvažovanie, napr. AVL, Červeno-Čierne stromy, (2,3), (2,3,4) stromy, Splay stromy, ...
- (3 body) Druhú vlastnú implementáciu BVS s iným algoritmom na vyvažovanie ako v predchádzajúcim bode.
- (3 body) Vlastnú implementáciu hašovacej tabuľky s riešením kolízí podľa vlastného výberu. Treba implementovať aj prispôsobenie veľkosti hašovacej tabuľky.
- (3 body) Druhú vlastnú implementáciu hašovacej tabuľky s riešením kolízí iným spôsobom ako v predchádzajúcim bode. Treba implementovať aj prispôsobenie veľkosti hašovacej tabuľky.

Za samotné implementácie podľa vyššie uvedených bodov môžete získať celkovo 12 bodov. Každú implementáciu odovzdáte v jednom samostatnom zdrojovom súbore (v prípade, že chcete odovzdať všetky štyri, tak odovzdáte ich v štyroch súboroch). Nie je povolené prevziať cudzí zdrojový kód. **Pre úspešné odovzdanie musíte zrealizovať aspoň dve z vyššie uvedených implementácií.** Správnosť overte testovaním – porovnaním s inými implementáciami.

V technickej dokumentácii je vašou úlohou zdokumentovať všetky implementované dátové štruktúry a uviesť podrobné scenáre testovania, na základe ktorých ste zistili, v akých situáciach sú ktoré z týchto implementácií efektívnejšie. **Vyžaduje to tiež odovzdanie programu**, ktorý slúži na testovanie a odmeranie efektívnosti týchto implementácií ako jedného samostatného zdrojového súboru (obsahuje funkciu **main**). **Bez testovacieho programu, a teda bez úspešného porovnania aspoň dvoch implementácií bude riešenie hodnotené 0 bodmi**. Za dokumentáciu, testovanie a dosiahnuté výsledky (identifikované vhodné situácie) môžete získať najviac 8 bodov. Hodnotí sa hlavne kvalita testovania a spracovania výsledkov v dokumentácii. Samozrejmosťou by mali byť tabuľky a grafy s výsledkami, v ktorých sa porovnáva výkonnosť (rýchlosť) jednotlivých riešení. Keďže výsledky závisia nielen od implementácie riešenia, ale aj od testovacích scenárov (postupnosti operácií **insert**, **delete** a **search**), je dôležité vyskúšať a zdokumentovať rôzne scenáre.

Celkovo môžete získať 20 bodov, **minimálna požiadavka je 8 bodov**.

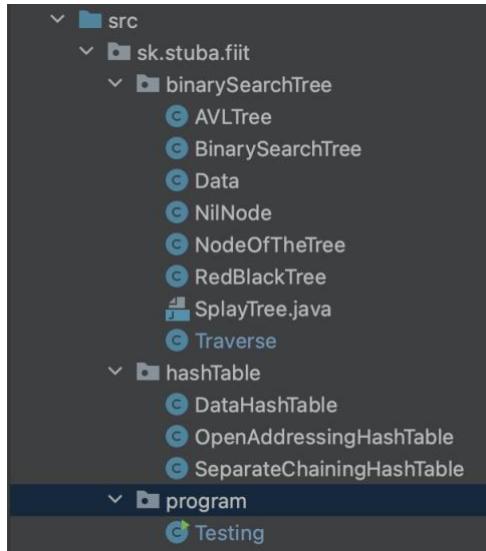
Riešenie zadania sa odovzdáva do miesta odovzdania v AIS do stanoveného termínu (oneskorené odovzdanie je prípustné len vo veľmi vážnych prípadoch, ako napr. choroba, o možnosti odovzdať zadanie oneskorené rozhodne cvičiaci). Odovzdáva sa jeden **zip** archív, ktorý obsahuje jednotlivé zdrojové súbory s implementáciami, testovací súbor a jeden súbor s dokumentáciou vo formáte **pdf**.

Obsah

FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ	1
2023 Mária Matušisková	2
1 ÚVOD PROJEKTU.....	4
2 BINÁRNE VYHĽADÁVACIE STROMY	5
2.1 DÁTA	5
2.2 UZLY BINÁRNYCH VYHĽADÁVACÍCH STROMOV	7
2.3 BINÁRNY NEVYVÁŽENÝ VYHĽADÁVACÍ STROM.....	8
2.3.1 Implementácia kódu	8
2.4 ADEISON-VELSKYOV A LANDISOV STROM	11
2.4.1 Implementácia kódu	11
2.5 ČERVENO-ČIERNY STROM	15
2.5.1 Implementácia kódu	15
2.6 TESTOVANIE BINÁRNYCH VYHĽADÁVACÍCH STROMOV.....	21
2.6.1 Testovací scenár I.....	21
2.6.2 Testovací scenár II.....	24
2.6.3 Testovací scenár III.....	26
3 HAŠOVACIE TABUĽKY	30
3.1 DÁTA	30
3.2 REŽAŽENIE	31
3.2.1 Implementácia kódu	32
3.3 OTVORENÉ ADRESOVANIE.....	34
3.3.1 Implementácia kódu	34
3.4 TESTOVANIE HAŠOVACÍCH TABULIEK	37
3.4.1 Testovací scenár I.....	37
3.4.2 Testovací scenár II.....	42
3.4.3 Testovací scenár III.....	44
4 ZÁVER	46
ZDROJE	47

1 Úvod projektu

Projekt bol realizovaný vo vývojárskom prostredí IntelliJ IDEA v programovacom jazyku Java na počítači MacBook Pro (13-inch, M1, 2020). Boli použité niektoré metódy, ktoré budú spomenuté v tejto dokumentácii nižšie z knižnice `java.util.*`. Osnova projektu je rozdelená do pod balíkov v hlavnej doméne v reverznom tvare `sk.stuba.fiit.*`.



Obrázok 1, Ukážka rozloženia balíkov v projekte

Na [obrázku 1](#) je ukázané ako sú jednotlivé súbory rozdelené do troch časti. Obsahom vrchného balíka sú **binárne vyhľadávacie stromy** s algoritmami na vyvažovanie ako **AVL** a **Červeno-Čierny strom**. Triedy **Data** a **Traverse** sú takzvaným doplnkom pre použitie stromov. (**Splay** iba implementácia kódu)

Nadchádzajúci balík sa dotýka druhej časti projektu, hashovacie tabuľky. V projekte sa používajú tabuľky so zvolenými algoritmami na riešenie ich kolízii. Prvým z nich je **zretazenie**, následne **lineárne otvorené adresovanie**. Trieda **DataHashTable** je ako v predchádzajúcim balíku doplnkom týchto tabuľiek.

Posledná zložka je samotný **program**, kde testujeme funkčnosť vybraných algoritmov.

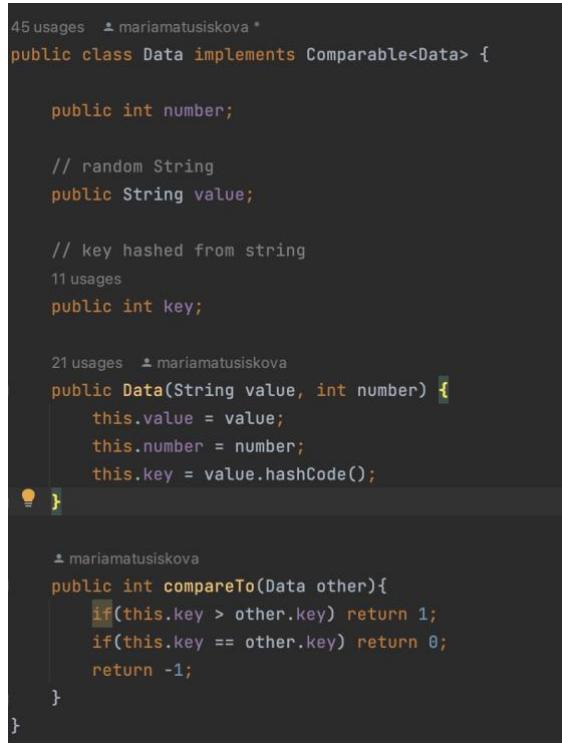
2 Binárne vyhľadávacie stromy

Binárny vyhľadávací strom je dátová štruktúra, ktorá ukladá, vyhľadáva a vymazáva pridelené dátá rôzneho typu. Používa sa hlavne kvôli časovej zložitosti, dokáže rýchlejšie vyhľadávať prvky v porovnaní s obyčajným binárnym stromom alebo tabuľkami. Avšak časová komplexita sa dá ešte znížiť a to vďaka algoritmom na vyvažovanie. Niektoré z ich budú ukázané v tejto kapitole.

Základom takého stromu je koreň po anglicky *root*, ktorý nemá rodiča, ale môže mať potomkov (*children*). Potomkovia môžu byť aj pod stromami ak obsahujú ďalších nasledovníkov. V strome sa orientujeme pomocou ukazovateľov na ľavého alebo pravého potomka.

2.1 Dáta

Dátovým typom pre prácu so binárnymi stromami je vytvorená trieda **Data**. V nej sa nachádza viacero druhov údajov. Generujú sa náhodne v testovacom programe a následne sa posielajú na spracovanie do vybraných stromov. V [obrázku 2](#) je znázornené s akými dátami sa v projekte zaoberáme.



```
45 usages  ± mariamatusiskova *
public class Data implements Comparable<Data> {

    public int number;

    // random String
    public String value;

    // key hashed from string
    11 usages
    public int key;

    21 usages  ± mariamatusiskova
    public Data(String value, int number) {
        this.value = value;
        this.number = number;
        this.key = value.hashCode();
    }

    ± mariamatusiskova
    public int compareTo(Data other){
        if(this.key > other.key) return 1;
        if(this.key == other.key) return 0;
        return -1;
    }
}
```

Obrázok 2, trieda Data

Každé vygenerované dátá sú unikátné za použitia metódy *hashCode()* z knižnice *java.util.**, ktorá vytvorí originálny klíč a potom pri vyvažovaní ho porovná pomocou funkcie *compareTo()*. Podľa nej sa určuje ako sa strom usporiada, teda dátá si samé určia prioritu vyvažovania. Ak by predsa nastali duplicity, tak sú riešené v použitých algoritnoch.

Do konštruktora sa posiela náhodne vygenerovaný reťazec znakov v kombinácii malých a veľkých písmen a čísel od 0 po 9. Dĺžka reťazca je konštantná veľkosť 5 a generuje sa pomocou konštruktorov a tried z balíčka `java.util.*`. Konkrétnie sa používajú inštancie **StringBuilder** a **Random** s ich metódami `append()`, `toString()` a `random()`.

([obrázok 3](#))

```
30 usages  ↲ mariamatusiskova
public String generateRandomString() {

    String chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

    Random ran = new Random();

    // make a string from chars of array --> append()
    StringBuilder sb = new StringBuilder( capacity: 5);

    for (int i = 0; i < 5; i++) {
        sb.append(chars.charAt(ran.nextInt(chars.length())));
    }

    // toString()
    return sb.toString();
}
```

Obrázok 3, generovanie náhodného reťazca

Okrem reťazca, konštruktor obsahuje aj parameter typu **int**, kde sa rovnako ako pri náhodne vygenerovanom reťazci posiela náhodne vygenerované číslo od 1 po 1000.

([obrázok 4](#))

```
30 usages  ↲ mariamatusiskova
int generateRandomNumber() {

    return 1 + (int)(Math.random() * 1000);
}
```

Obrázok 4, generovanie náhodného čísla

Zobrazenie samotného generovania dát([obrázok 5](#)):

```
1 usage  ↲ mariamatusiskova
public List<Data> generateData() {
    List<Data> data = new ArrayList<>();
    for (int i = 0; i < 200; i++) {
        data.add(new Data(generateRandomString(), generateRandomNumber()));
    }
    return data;
}
```

Obrázok 5, generovanie náhodných dát

2.2 Uzly binárnych vyhľadávacích stromov

Uzly sú hlavnou súčasťou grafu stromov. Pomocou nich sa posúvame na ľavého a pravého potomka, ukladáme do nich dátu. Okrem toho algoritmy v tomto projekte potrebujú ešte aj niečo navyše. **AVL strom** vyžaduje výšku pre správne rotácie a **Červeno-čierny strom** zas fiktívneho rodiča a farbu.

([obrázok 6](#))

```
// Node for storing values and keeping a reference to each child
60 usages 1 inheritor ▲ mariamatusiskova
public class NodeOfTheTree {

    // BinarySearchTree

    public Data data;
    59 usages
    public NodeOfTheTree right;
    63 usages
    public NodeOfTheTree left;

    // AVL Tree
    4 usages
    public int height;

    // Splay Tree (There was attempt of Red-Black Tree)
    // we need a parent node to balance the tree (because the colours of the tree depend on the parent)
    47 usages
    public NodeOfTheTree parent;
    39 usages
    public boolean color;
```

Obrázok 6, ukážka referencii na orientovanie sa v grafe

Volanie konštruktora na vloženie dát ([obrázok 7](#)):

```
// default constructor
no usages ▲ mariamatusiskova
NodeOfTheTree() {
    this.data = null;
    this.right = this.left = null;
    this.height = 0;
    this.color = false;
    this.parent = null;
}

// add a new node
3 usages ▲ mariamatusiskova
public NodeOfTheTree(Data data) {
    this.data = data;
    this.right = this.left = null;
    this.height = 0;
    this.color = false;
    this.parent = null;
}
```

Obrázok 7, konštruktory uzlov

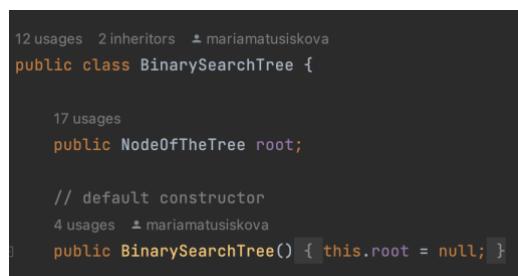
2.3 Binárny nevyvážený vyhľadávací strom

Binárny nevyvážený vyhľadávací strom je orientovaný graf s takzvanými uzlami, ktoré obsahujú dátu. Podľa pravidla má každý uzol najviac dvoch potomkov, ľavý a pravý. Ľavostranný potomok obsahuje vždy menšie hodnoty ako pravý.

Má časovú zložitosť v priemere **O(log n)**, v najhorších prípadoch **O(n)**. O(n) nastane vtedy keď vkladáme prvky postupne s približnou hodnotou. Je efektívny hlavne vo vyhľadávaní a vymazávaní dát. Nezapĺňa až tak úložisko pamäti ako obyčajné pole dát.

2.3.1 Implementácia kódu

Úvod do binárneho vyhľadávacieho stromu ([obrázok 8](#)):



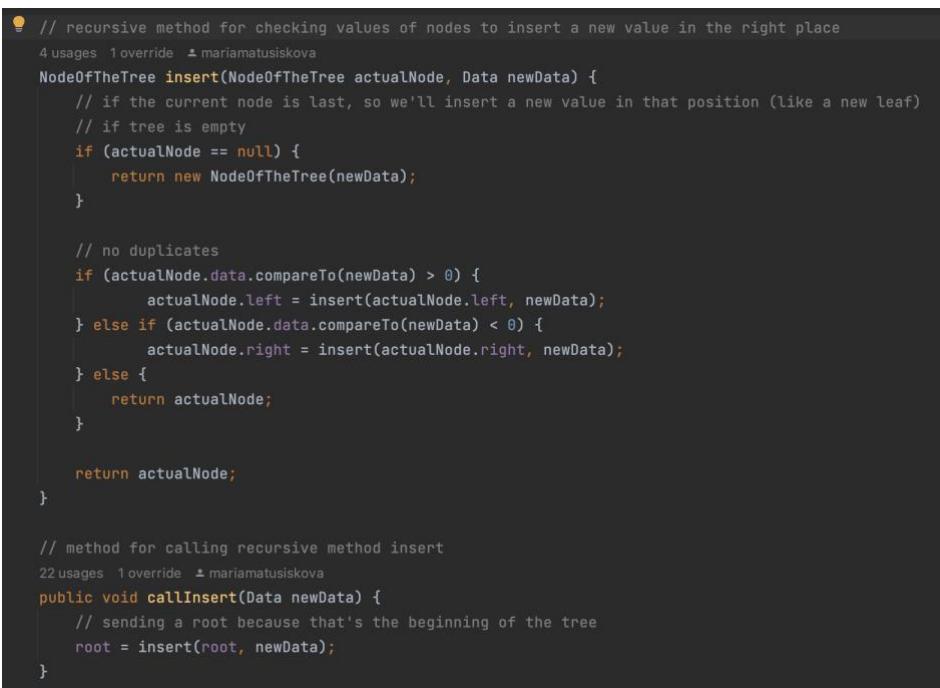
```
12 usages 2 inheritors ▲ mariamatusiskova
public class BinarySearchTree {

    17 usages
    public NodeOfTheTree root;

    // default constructor
    4 usages ▲ mariamatusiskova
    public BinarySearchTree() { this.root = null; }
```

Obrázok 8, konštruktor BVS a koreň stromu

Metóda *insert()* (vložiť) je riešená **rekurzívnou formou**. Vstup je hned' ošetrený, kde sa kontroluje či sa neposielajú dátu s hodnotou *null*. A z testovacieho programu ju voláme pomocou metódy *callInsert()*, kde vkladáme dátu. ([obrázok 9](#))



```
● // recursive method for checking values of nodes to insert a new value in the right place
4 usages 1 override ▲ mariamatusiskova
NodeOfTheTree insert(NodeOfTheTree actualNode, Data newData) {
    // if the current node is last, so we'll insert a new value in that position (like a new leaf)
    // if tree is empty
    if (actualNode == null) {
        return new NodeOfTheTree(newData);
    }

    // no duplicates
    if (actualNode.data.compareTo(newData) > 0) {
        actualNode.left = insert(actualNode.left, newData);
    } else if (actualNode.data.compareTo(newData) < 0) {
        actualNode.right = insert(actualNode.right, newData);
    } else {
        return actualNode;
    }

    return actualNode;
}

// method for calling recursive method insert
22 usages 1 override ▲ mariamatusiskova
public void callInsert(Data newData) {
    // sending a root because that's the beginning of the tree
    root = insert(root, newData);
}
```

Obrázok 9, metóda *insert()* BVS

Metóda `search()` (hľadat') s návratovou hodnotou **boolean** je riešená cez **while** cyklus, čo je efektívnejšia forma, ktorá je potrebná najmä pri hľadaní. Z testovacieho programu ju voláme pomocou funkciou `callSearch()` s vloženými dátami. Vstup je takisto ošetrený. ([obrázok 10](#))

```
1 usage  ↗ mariamatusiskova
boolean search(NodeOfTheTree actualNode, Data searchData) {

    if (actualNode == null) {
        return false;
    }

    while (actualNode != null) {
        if (actualNode.data.key == searchData.key) {
            return true;
        } else if (actualNode.data.key > searchData.key) {
            actualNode = actualNode.left;
        } else if (actualNode.data.key < searchData.key) {
            actualNode = actualNode.right;
        }
    }
    return false;
}

// method for calling recursive method search
8 usages  ↗ mariamatusiskova
public void callSearch(Data searchData) {
    // sending a root because that's the beginning of the tree
    search(root, searchData);
}
```

Obrázok 10, metóda `search()` BVS

Metóda `delete()` (vymazať) je riešená **rekurzívou formou**, tu by bol efektívnejší while cyklus pre časovú zložitosť. Pri delení nastávajú tri možnosti, že strom nemá žiadneho potomka, vtedy uzol jednoducho vymažeme. Druhá možnosť je, že strom má iba jedného potomka, tu sa rozlišuje či je to pravého alebo ľavého. Potom sa uzol jednoducho prepíše. Posledná a najnáročnejšia možnosť nastane, keď rodič má dvoch potomkov, zavolá sa funkcia `findMinimum()`, ktorá nájde najmenšiu hodnotu a prepíše rodiča, pravý uzol sa tiež prepíše. Túto metódu voláme metódou `callDelete()` z programu. ([obrázok 11](#))

```
5 usages 1 override ▾ mariamatusiskova *
NodeOfTheTree delete(NodeOfTheTree actualNode, Data deleteData) {
    if (actualNode == null) {
        return null;
    }

    if(deleteData.compareTo(actualNode.data) < 0){
        actualNode.left = delete(actualNode.left, deleteData);
    }
    else if(deleteData.compareTo(actualNode.data) > 0){
        actualNode.right = delete(actualNode.right, deleteData);
    }
    else{
        //one child or no child
        if(actualNode.left == null){
            return actualNode.right;
        }
        //one child
        else if(actualNode.right == null){
            return actualNode.left;
        }

        //has two children
        actualNode = findMinimum(actualNode.right);

        actualNode.right = delete(actualNode.right, deleteData);
    }

    return actualNode;
}
```

Obrázok 11, metóda `delete()` BVS

Ako funguje hľadanie minimálnej hodnoty stromu pravej strany pod stromu:
[\(obrázok 12\)](#)

```
2 usages ▾ mariamatusiskova
NodeOfTheTree findMinimum(NodeOfTheTree actualNode) {
    while (actualNode.left != null) {
        actualNode = actualNode.left;
    }

    return actualNode;
}

5 usages 1 override ▾ mariamatusiskova
public void callDelete(Data deleteData) {
    // sending a root because that's the beginning of the tree
    root = delete(root, deleteData);
}
```

Obrázok 12, metóda `findMinimum()` a `callDelete()`

2.4 Adeison-Velskyov a Landisov strom

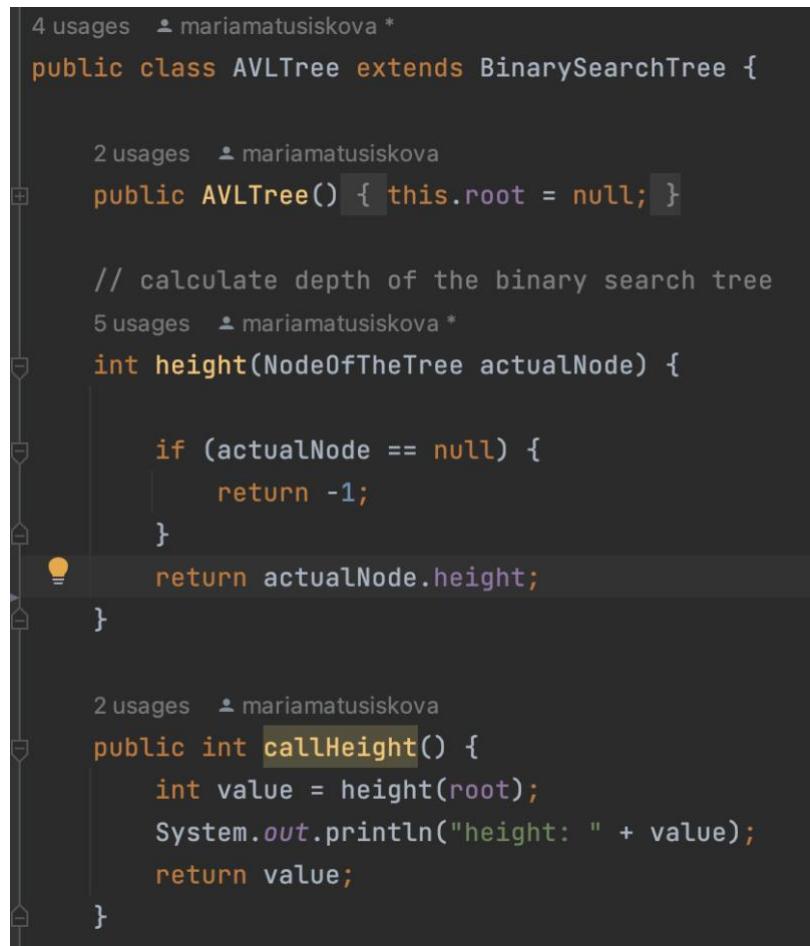
AVL je samovyvažovací strom u ktorého rozdiel medzi hĺbkami/výškami ľavého a pravého **nemôže byť väčší ako 1**. Inak povedané ide tu o výškový balans stromu. Zakaždým keď sa pridáva alebo vymazáva prvk AVL kontroluje výšku a podľa toho rotuje uzlami v grafe, aby sa strom vyvážil.

Časová zložitosť tohto algoritmu je **O(log n)**, vďaka tomu sa dá vyhnúť problému, ktorý môže pomerne ľahko nastaviť u nevyváženom binárnom vyhľadávacom strome. Ten nastáva najmä pri vkladaní podobných prvkov a tak sa výška zvýši aj o viac než 1, ako už bolo uvedené v predošej podkapitole.

V tomto projekte AVL dedí od nevyváženého binárneho vyhľadávacieho stromu, keďže má podobnú štruktúru, ale do kódu sa ešte pridali rotácie a kalkulovanie výšky stromu.

2.4.1 Implementácia kódu

Úvod do AVL stromu znázorňuje [obrázok 13](#). Metóda *callHeight()* slúži iba na testovacie účely pre kontrolu výšky.



The screenshot shows a code editor with Java code for an AVL tree. The code includes a constructor, a height calculation method, and a callHeight() method for testing. The code is annotated with tool tips and highlights from an IDE like IntelliJ IDEA.

```
4 usages  ↳ mariamatusiskova *
public class AVLTree extends BinarySearchTree {

    2 usages  ↳ mariamatusiskova
    public AVLTree() { this.root = null; }

    // calculate depth of the binary search tree
    5 usages  ↳ mariamatusiskova *
    int height(NodeOfTheTree actualNode) {

        if (actualNode == null) {
            return -1;
        }
        return actualNode.height;
    }

    2 usages  ↳ mariamatusiskova
    public int callHeight() {
        int value = height(root);
        System.out.println("height: " + value);
        return value;
    }
}
```

Obrázok 13, výška AVL stromu a jeho koreň

Metóda *insert()* (pridať) je zdedená od BVS a doplnená o ďalší kód na sledovanie výšky stromu a jeho rotácie. ([obrázok 14](#))

```
4 usages  ↗ mariamatusiskova
NodeOfTheTree insert(NodeOfTheTree actualNode, Data newData) {
    actualNode = super.insert(actualNode, newData);

    if (actualNode.data == newData) {
        return actualNode;
    }

    updateHeight(actualNode);
    return rebalance(actualNode);
}
```

Obrázok 14, metóda *insert()* AVL

Metóda *search()* (hladat') je zdedená od BVS. ([obrázok 10](#))

Metóda *delete()* (vymazat') je zdedená od BVS a doplnená o ďalší kód na sledovanie výšky stromu a jeho rotácie. ([obrázok 15](#))

```
5 usages  ↗ mariamatusiskova
NodeOfTheTree delete(NodeOfTheTree actualNode, Data deleteData) {
    actualNode = super.delete(actualNode, deleteData);

    if (actualNode == null) {
        return null;
    }

    updateHeight(actualNode);
    return rebalance(actualNode);
}
```

Obrázok 15, metóda *delete()* AVL

Metóda `rebalance()` volaná pri `insert()` a `delete()`. Kontroluje výšku stromu a podľa toho vyvažuje binárny vyhľadávací strom. Ak je výška ľavého a pravého pod stromu nevyvážená, tak zavolá jednu z rotácií, ktoré strom vybalansujú. ([obrázok 16](#))

```
2 usages  ↗ mariamatusiskova
NodeOfTheTree rebalance(NodeOfTheTree node) {

    if (node == null) {
        return null;
    }

    if (balanceFactor(node) < -1) {
        if (balanceFactor(node.left) <= 0) {
            node = leftRotation(node);
        } else if (balanceFactor(node.left) > 0) {
            node = leftRightRotation(node);
        }
    } else if (balanceFactor(node) > 1) {
        if (balanceFactor(node.right) >= 0) {
            node = rightRotation(node);
        } else if (balanceFactor(node) < 0) {
            node = rightLeftRotation(node);
        }
    }
}

return node;
}
```

Obrázok 16, metóda `rebalance()` AVL

Metódy `balanceFactor()` a `updateHeight()` ([obrázok 17](#)):

```
6 usages  ↗ mariamatusiskova
void updateHeight(NodeOfTheTree node) { node.height = 1 + Math.max(height(node.left), height(node.right)); }

6 usages  ↗ mariamatusiskova
int balanceFactor(NodeOfTheTree node) { return height(node.right) - height(node.left); }
```

Obrázok 17, metódy `balanceFactor()` a `updateHeight()`

Metódy na rotácie:

- *rightRotation()* ([obrázok 18](#))
- *leftRotation()* ([obrázok 19](#))
- *rightLeftRotation()* ([obrázok 20](#))
- *leftRightRotation()* ([obrázok 20](#))

```
3 usages ▾ mariamatusiskova
NodeOfTheTree rightRotation(NodeOfTheTree actualNode) {

    if (actualNode.left != null) {

        // new parent
        NodeOfTheTree newParent = actualNode.left;

        // the left child will be changed from the original parent
        actualNode.left = newParent.right;

        if (newParent.right != null) {
            // original parent will be right child of the new parent
            newParent.right = actualNode;
        }

        updateHeight(actualNode);
        updateHeight(newParent);

        return newParent;
    } else {
        return actualNode;
    }
}
```

Obrázok 18, metóda *rightRotation()*

```
3 usages ▾ mariamatusiskova
NodeOfTheTree leftRotation(NodeOfTheTree actualNode) {

    if (actualNode.right != null) {

        // new parent
        NodeOfTheTree newParent = actualNode.right;

        // the right child will be changed from the original parent
        actualNode.right = newParent.left;

        if (newParent.left != null) {
            // original parent will be right child of the new parent
            newParent.left = actualNode;
        }

        updateHeight(actualNode);
        updateHeight(newParent);

        return newParent;
    } else {
        return actualNode;
    }
}
```

Obrázok 19, metóda *leftRotation()*

```

1 usage  ▲ mariamatusiskova
NodeOfTheTree rightLeftRotation(NodeOfTheTree actualNode) {

    actualNode.right = rightRotation(actualNode.right);
    return leftRotation(actualNode);
}

1 usage  ▲ mariamatusiskova
NodeOfTheTree leftRightRotation(NodeOfTheTree actualNode) {

    actualNode.left = leftRotation(actualNode.left);
    return rightRotation(actualNode);
}

```

Obrázok 20, metódy `rightLeftRotation()` a `rightRightRotation()`

2.5 Červeno-čierny strom

Červeno-čierny strom je samovyvažovací a jeho pomenovanie znamená, že každý uzol stromu má buď čiernu alebo červenú farbu podľa pravidla, že farba z jedného z potomkov červeného rodiča musí byť čierna. Teda rotuje a vyvažuje na základe farieb uzlov.

Časová zložitosť je **O(log n)**, podobne ako pri AVL stromoch s rozdielom, že celkovo je rýchlejší pretože robí menej rotácií.

V tomto projekte tento strom dedí od nevyváženého binárneho vyhľadávacieho stromu, keďže má podobnú štruktúru, avšak väčšinu funkcií má vlastných.

2.5.1 Implementácia kódu

V tomto algoritme sa používa pomocný nulový uzol, kvôli špecifickým rotáciám. Je to dôležité, pretože null uzly majú byť čierne, len tak rotácia prebehne korektnie.

```

3 usages  ▲ mariamatusiskova
public class NilNode extends NodeOfTheTree {
    1 usage
    private final boolean BLACK = true;

    2 usages  ▲ mariamatusiskova
    NilNode() {
        super(new Data( value: "", number: 0));
        this.color = BLACK;
    }
}

```

Obrázok 21, trieda `NilNode` (pomocný nulový uzol)

Na [obrázku 21](#) je znázornený nil uzol, ktorý dedí od jeho základného rodiča, ktorý je používaný v stromoch ([obrázok 2](#)).

Úvod do červeno-čierneho stromu, sú tam konštanty RED a BLACK, ktoré sa používajú na farbenie uzlov ([obrázok 22](#)):

```
package sk.stuba.fiit.binarySearchTree;

4 usages ▲ mariamatusiskova
public class RedBlackTree extends BinarySearchTree {

    11 usages
    private final boolean RED = false;
    22 usages
    private final boolean BLACK = true;

    2 usages ▲ mariamatusiskova
    public RedBlackTree() { this.root = null; }
```

Obrázok 22, konštanty RBT a jeho koreň

Metóda *insert()* sa skladá z troch častí:

- *callInsert()*
 - ([obrázok 23](#))
 - úvodna metóda na volanie insertu v testovacom programe
- *insertRBT()*
 - ([obrázok 25](#))
 - vloženie uzlu do stromu
 - v prvej časti sa nastaví ukazovateľ na voľné miesto, pravidlo RBT je, že každý nový uzol je zafarbený na červeno
 - po nastavení ukazovateľa sa uzol vloží za správnych podmienok
 - po vložení sa kontrolujú farby v strome
- *insertFixup()* – kontrola farieb v strome a vyvažovanie
 - ďalším pravidlo RBT je, že koreň stromu má byť vždy čierny (kontrola podmienky) ([obrázok 24](#))
 - kontrola prarodiča, rovnako ako u podmienky vyššie ([obrázok 24](#))
 - kontrola farby vedľajšieho uzla rodiča ([obrázok 26](#))
 - kontrola prarodiča s rodičom a ich potomkami po kontrole farieb, následná rotácia za splnenia podmienok a znova prefarbovanie uzlov, keďže farby nesedeli a z toho dôvodu nastala rotácia ([obrázok 27](#))

```
22 usages ▲ mariamatusiskova
public void callInsert(Data newData) {
    // sending a root because that's the beginning of the tree
    insertRBT(root, newData);
}
```

Obrázok 23, metóda *callInsert()*

```

public void insertRBT(NodeOfTheTree actualNode, Data newData) {

    // store data into node
    NodeOfTheTree newNode = new NodeOfTheTree(newData);
    NodeOfTheTree parent = null;

    // if tree is not empty
    while (actualNode != null) {
        parent = actualNode;
        if (actualNode.data.compareTo(newData) > 0) {
            actualNode = actualNode.left;
        } else if (actualNode.data.compareTo(newData) < 0) {
            actualNode = actualNode.right;
        } else {
            return;
        }
    }

    newNode.color = RED;

    // if added node is a root
    if (parent == null) {
        root = newNode;
    } else if (parent.data.compareTo(newNode.data) > 0) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
    newNode.parent = parent;

    insertFixup(newNode);
}

```

Obrázok 25, metóda `insertRBT()`

```

2 usages  ± mariamatusiskova
private void insertFixup(NodeOfTheTree newNode) {

    NodeOfTheTree parent = newNode.parent;

    // check if parent is null, pointer is in the end of the tree
    if (parent == null) {
        // newNode is a root
        newNode.color = BLACK;
        return;
    }

    // case when parent is already black
    if (parent.color == BLACK) {
        return;
    }

    // case when parent is red:
    NodeOfTheTree grandparent = parent.parent;

    // if grandparent is null then again parent is a root
    if (grandparent == null) {
        parent.color = BLACK;
        return;
    }
}

```

Obrázok 24, metóda `insertFixup()`

```

// check if uncle is black or nor
NodeOfTheTree uncle = null;

// grandparent.left = uncle
if (parent.parent.left == parent) {
    uncle = parent.parent.right;
// grandparent.right = uncle
} else if (parent.parent.right == parent) {
    uncle = parent.parent.left;
}

// case if uncle is red
if (uncle != null && uncle.color == RED) {

    /* before:           after:
       G = B             G = R
       / \              / \
      U = R   P = R      U = B   P = B
    */

    parent.color = BLACK;
    grandparent.color = RED;
    uncle.color = BLACK;

    // grandparent is red, but we don't know if it is a root or not and the rule is, that root have to be black
    insertFixup(grandparent);
}

/*
condition:
  G
  / \
  P
*/

```

Obrázok 26, metóda `insertFixup()`

```

    } else if (parent == grandparent.left) {
        /*
         condition:
            G
            / \
            P
            / \
            NewNode
        */
        if (newNode == parent.right) {
            leftRotation(parent);
            parent = newNode;
            /*
             after:
                G
                / \
                NewNode
                / \
                P
            */
        }
    }
    /*
     case:
        G
        / \
        NewNode
        / \
        P
    */
}
}

```

rightRotation(grandparent);
 /*
 after:
 NewNode
 / \
 G P
 */
 parent.color = BLACK;
 grandparent.color = RED;
/*
condition:
 G
 / \
 P
*/
} else {
 // symmetrical
 if (newNode == parent.left) {
 rightRotation(parent);
 parent = newNode;
 }
 leftRotation(grandparent);
 parent.color = BLACK;
 grandparent.color = RED;
}
}

Obrázok 27, metóda `insertFixup()`

Metóda `search()` (hl'adat') je zdedená od BVS. ([obrázok 10](#))

Metóda `delete()` (vymazat') sa skladá z troch časti“

- `callDelete()`
 - ([obrázok 28](#))
 - úvodná metóda na volanie delete v testovacom programe
- `deleteRBT()`
 - ([obrázok 29](#), [obrázok 30](#), [obrázok 31](#))
 - hl'adanie uzlu na vymazanie
 - ak sa nenašiel vrátenie naspäť do programu
 - ak sa našiel, vymazanie v troch prípadoch mazania (žiadny potomok, jeden potomok, dva potomkovia), riešia sa tam aj farby jednotlivých uzlov
- `deleteFixup()`
 - ([obrázok 32](#), [obrázok 33](#), [obrázok 34](#))
 - kontrola farieb a potrebná rotácia na vyváženie podľa pravidiel RBT

```

5 usages  ± mariamatusiskova
public void callDelete(Data deleteData) {
    // sending a root because that's the beginning of the tree
    deleteRBT(root, deleteData);
}

```

Obrázok 28, metóda *callDelete()*

```

1 usage  ± mariamatusiskova
public void deleteRBT(NodeOfTheTree actualNode, Data deleteData) {

    NodeOfTheTree deleteNode;
    boolean deleteColor;

    // search node to delete
    while (actualNode != null && actualNode.data != deleteData) {

        if (actualNode.data.compareTo(deleteData) > 0) {
            actualNode = actualNode.left;
        } else {
            actualNode = actualNode.right;
        }
    }

    // not found
    if (actualNode == null) {
        return;
    }

    // no children or only right child
    if (actualNode.left == null || actualNode.right == null) {

        if (actualNode.left != null){
            transplant(actualNode.parent, actualNode, actualNode.left);
            deleteNode = actualNode.left;
        } else if (actualNode.right != null) {
            transplant(actualNode.parent, actualNode, actualNode.right);
            deleteNode = actualNode.right;
        }
    }
}

```

Obrázok 29, metóda *deleteRBT()*

```

// no children, two options: 1) actualNode is red, then remove it; 2) actualNode is black, replace it with NIL node, then fix RBT
} else {
    NodeOfTheTree alternativeNode = actualNode.color == BLACK ? new NilNode() : null;
    transplant(actualNode.parent, actualNode, alternativeNode);
    deleteNode = alternativeNode;
}

deleteColor = actualNode.color;
// two children
} else {

    NodeOfTheTree successorNode = findMinimum(actualNode.right);
    actualNode.data = successorNode.data;

    // if it is child of deleteNode
    if (successorNode.left != null) {
        transplant(successorNode.parent, successorNode, successorNode.left);
        deleteNode = successorNode.left;
    } else if (successorNode.right != null) {
        transplant(successorNode.parent, successorNode, successorNode.right);
        deleteNode = successorNode.right;
    }
    // no children, two options: 1) actualNode is red, then remove it; 2) actualNode is black, replace it with NIL node, then fix RBT
} else {
    NodeOfTheTree newChild = successorNode.color == BLACK ? new NilNode() : null;
    transplant(successorNode.parent, successorNode, newChild);
    deleteNode = newChild;
}

deleteColor = successorNode.color;
}

```

Obrázok 30, metóda *deleteRBT()*

```

        if (deleteColor == BLACK) {
            deleteFixup(deleteNode);

            // remove NIL node
            if (deleteNode.getClass() == NilNode.class) {
                transplant(deleteNode.parent, deleteNode, newChild: null);
            }
        }
    }
}

```

Obrázok 31, metóda `deleteRBT()`

<pre> // recursive method 2 usages + mariamatusiskova private void deleteFixup(NodeOfTheTree deleteNode) { NodeOfTheTree uncle; if (deleteNode == root) { deleteNode.color = BLACK; return; } if (deleteNode == deleteNode.parent.left) { uncle = deleteNode.parent.right; } else if (deleteNode == deleteNode.parent.right) { uncle = deleteNode.parent.left; } else { uncle = null; } if (uncle == null) { return; } } </pre>	<pre> // uncle is red, needs to be black if (uncle.color == RED) { uncle.color = BLACK; deleteNode.parent.color = RED; // fix RBT if (deleteNode == deleteNode.parent.left) { leftRotation(deleteNode.parent); } else { rightRotation(deleteNode.parent); } // uncle update if (deleteNode == deleteNode.parent.left) { uncle = deleteNode.parent.right; } else if (deleteNode == deleteNode.parent.right) uncle = deleteNode.parent.left; } else { uncle = null; } } </pre>
---	---

Obrázok 32, metóda `deleteFixup()`

<pre> // black uncle and two black children if ((uncle.left == null uncle.left.color == BLACK) && (uncle.right == null uncle.right.color == BLACK)) { uncle.color = RED; // case: red parent if (deleteNode.parent.color == RED) { deleteNode.parent.color = BLACK; // case: black parent } else { deleteFixup(deleteNode.parent); } } // black uncle and one or more red children } else { if ((deleteNode == deleteNode.parent.left) && (uncle.right == null uncle.right.color == BLACK)) { uncle.left.color = BLACK; uncle.color = RED; rightRotation(uncle); uncle = deleteNode.parent.right; } else if (!(deleteNode == deleteNode.parent.left) && (uncle.left == null uncle.left.color == BLACK)) { uncle.right.color = BLACK; uncle.color = RED; leftRotation(uncle); uncle = deleteNode.parent.left; } uncle.color = deleteNode.parent.color; deleteNode.parent.color = BLACK; } </pre>

Obrázok 33, metóda `deleteFixup()`

```

        if (deleteNode == deleteNode.parent.left) {
            uncle.right.color = BLACK;
            leftRotation(deleteNode.parent);
        } else {
            uncle.left.color = BLACK;
            rightRotation(deleteNode.parent);
        }
    }
}

```

Obrázok 34, metóda `deleteFixup()`

2.6 Testovanie binárnych vyhľadávacích stromov

V testovacom programe sa pracovalo s kardinalitou – 1 000, 10 000, 100 000, 1 000 000, 10 000 000. Dáta boli náhodne vygenerované, viac je o tom napísané v [podkapitole 2.1](#).

2.6.1 Testovací scenár I.

V tomto scenári sa testuje každá metóda individuálne. Do `ArrayList`, ktorý je použitý k knižnici `java.util.*` sa vložili náhodné hodnoty a potom sa zvlášť u každej metódy testovali. Sledovalo sa ich správanie pri rôznych mohutnostiach, teda časová komplexita, ktorá sa spriemerovala na jeden prvok. Výsledok tohto testu by mal mať časovú zložitosť v priemere **O(log n)** a v najhoršom prípade **O(n)**.

2.6.1.1 Implementácia kódu

Čas sa začal merať od volania funkcie po jej koniec, následne sa spriemeroval.
[\(Obrázok 35\)](#)

```

// ##### Binary search tree, Adelson-Velsky Landis Tree, Red-Black Tree --> testing #####
3 usages  ↗ mariamatusikova
public void testTreeInsertDeleteSearchIndividually(BinarySearchTree tree, int numberOfNodes, List<Data> data) {
    final int LOOP = data.size();

    for (int i = 0; i < numberOfNodes; i++) {
        tree.callInsert(new Data(generateRandomString(), generateRandomNumber()));
    }

    // insert testing
    startTime = System.nanoTime();
    for (int i = 0; i < LOOP; i++) {
        tree.callInsert(data.get(i));
    }
    endTime = System.nanoTime();
    duration = endTime - startTime;
    System.out.println("Time taken to insert " + i + " element from " + (numberOfNodes+1) + ":" + duration + " nanoseconds ");
    System.out.println("Average time of insert element: " + duration/LOOP + " nanoseconds");
    System.out.println();
}

```

```

// search testing
startTime = System.nanoTime();
for (int i = 0; i < LOOP; i++) {
    tree.callSearch(data.get(i));
}
endTime = System.nanoTime();
duration = endTime - startTime;
System.out.println("Time taken to search " + 1 + " element from " + (numberOfNodes+1) + ":" + duration + " nanoseconds");
System.out.println("Average time of search element: " + duration/LOOP + " nanoseconds");
System.out.println();

// delete testing
startTime = System.nanoTime();
for (int i = 0; i < LOOP; i++) {
    tree.callDelete(data.get(i));
}
endTime = System.nanoTime();
duration = endTime - startTime;
System.out.println("Time taken to delete " + 1 + " element from " + (numberOfNodes+1) + ":" + duration + " nanoseconds");
System.out.println("Average time of delete element: " + duration/LOOP + " nanoseconds");
System.out.println();
}

```

Obrázok 35, metóda `testTreeInsertDeleteSearchIndividually()`

Volanie testu v maine ([obrázok 36](#)):

```

System.out.println("***** BST - testing methods individually *****");
test.testTreeInsertDeleteSearchIndividually(bst, numberOfNodes_10000, data);
System.out.println("***** AVL - testing methods individually *****");
test.testTreeInsertDeleteSearchIndividually(avlTree, numberOfNodes_10000, data);
System.out.println("***** RBT - testing methods individually *****");
test.testTreeInsertDeleteSearchIndividually(redBlackTree, numberOfNodes_10000, data);

```

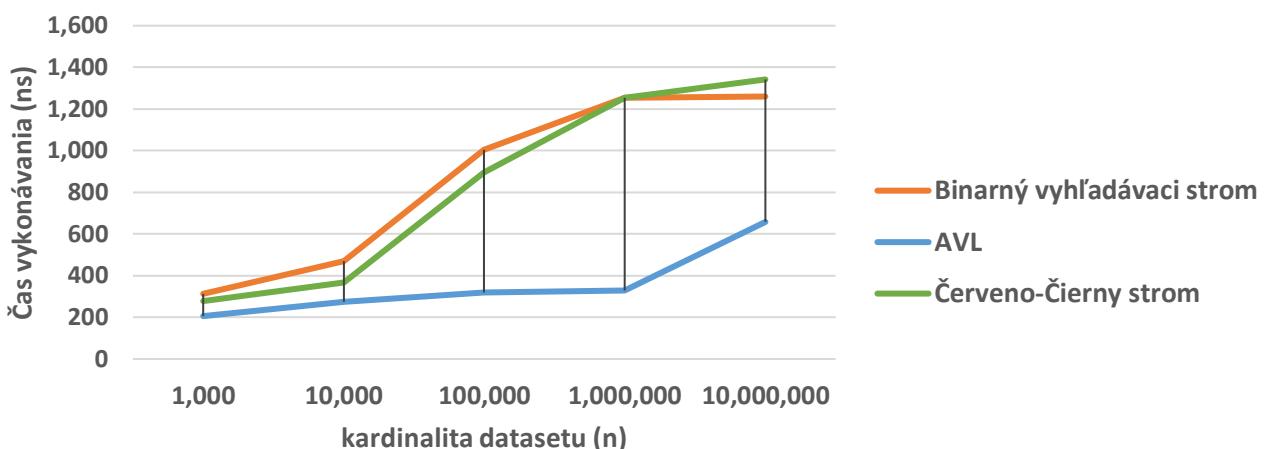
Obrázok 36, volanie v maine()

2.6.1.2 Grafy a štatistika

Metóda `insert()`, testovalo sa 200 vložených dát a spriemerovalo sa to na jeden element výpočtom celkového času trvania **/200** v nanosekundách.

Metóda <code>insert()</code>			
n	Binárny vyhľadávací strom	AVL	Červeno-Čierny strom
1000	313	206	278
10000	469	273	366
100000	1004	319	894
1000000	1255	330	1255
10000000	1260	658	1342

Stromy - metóda insert()

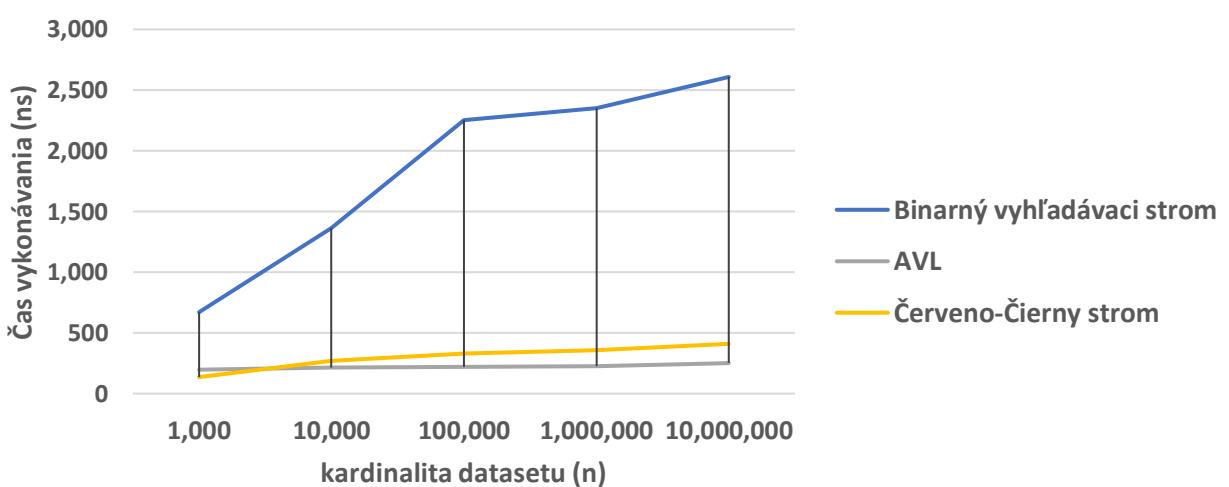


Z grafu je vidieť, že najrýchlejšie vkladá dáta AVL algoritmus na vyváženie a to aj napriek neustálym rotáciám. Nevyvážený binárny vyhľadávací strom v porovnaní s červeno-čiernym stromom majú len malé rozdiely.

Metóda *search()*, testovalo sa 200 vložených dát a spriemerovalo sa to na jeden element výpočtom **celkového času trvania /200** v nanosekundách.

Metóda search()			
n	Binárny vyhľadávací strom	AVL	Červeno-Čierny strom
1000	671	197	136
10000	1360	217	269
100000	2252	219	328
1000000	2351	227	358
10000000	2608	251	410

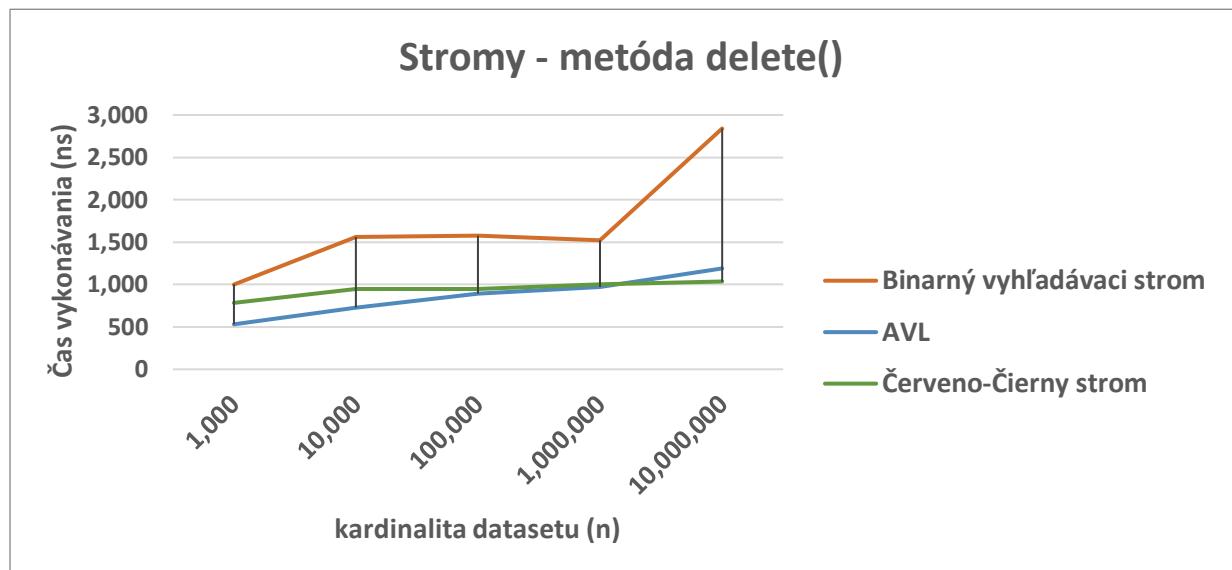
Stromy - metóda search()



Pri metóde `search()` je viditeľné ako nevyvážený strom vie byť neefektívny pri takomto rozmere dát. Zatiaľ čo pri rotáciach je tá časová komplexita omnoho lepšia. Najrýchlejšie vyhľadávanie je pri AVL algoritme.

Metóda `delete()`, testovalo sa 200 vložených dát a spriemerovalo sa to na jeden element výpočtom **celkového času trvania /200** v nanosekundách.

Metóda <code>delete()</code>				
n	Binárny vyhľadávací strom	AVL	Červeno-Čierny strom	
1000	1000	531	784	
10000	1560	730	947	
100000	1573	890	948	
1000000	1523	969	1005	
10000000	2841	1190	1036	



V poslednom grafe sa ukázal Červeno-čierny strom ako najefektívnejší pri metóde `delete()` pri väčšej mohutnosti dát, pri menšej je to AVL strom. Najmenej účinný je opäť nevyvážený strom.

2.6.2 Testovací scenár II.

Druhý testovací scenár sa zaobrá celkovou časovou zložitosťou metód `insert()` → `search()` → `delete()`. Sleduje sa ich správanie pri rôznych kardinalitách dát, keď sa po ich vložení vloží do nich ešte plus jeden prvok a ten sa práve testuje a časuje. Pri menšom počte dát by mal byť najrýchlejší Červeno-čierny strom a pri väčšom AVL. Tak sa aj v praxi využívajú. Algoritmická zložitosť by sa mala rovnati **O(1)**.

2.6.2.1 Implementácia kódu

Čas sa stopuje od vloženia jedného prvku po jeho vymazanie. ([obrázok 37](#))

```
3 usages  mariamatusiskova
public void testWholeTree(BinarySearchTree tree, int numberOfNodes) {

    for (int i = 0; i < numberOfNodes; i++) {
        tree.callInsert(new Data(generateRandomString(), generateRandomNumber()));
    }

    startTime = System.nanoTime();

    tree.callInsert(specific);
    tree.callSearch(specific);
    tree.callDelete(specific);

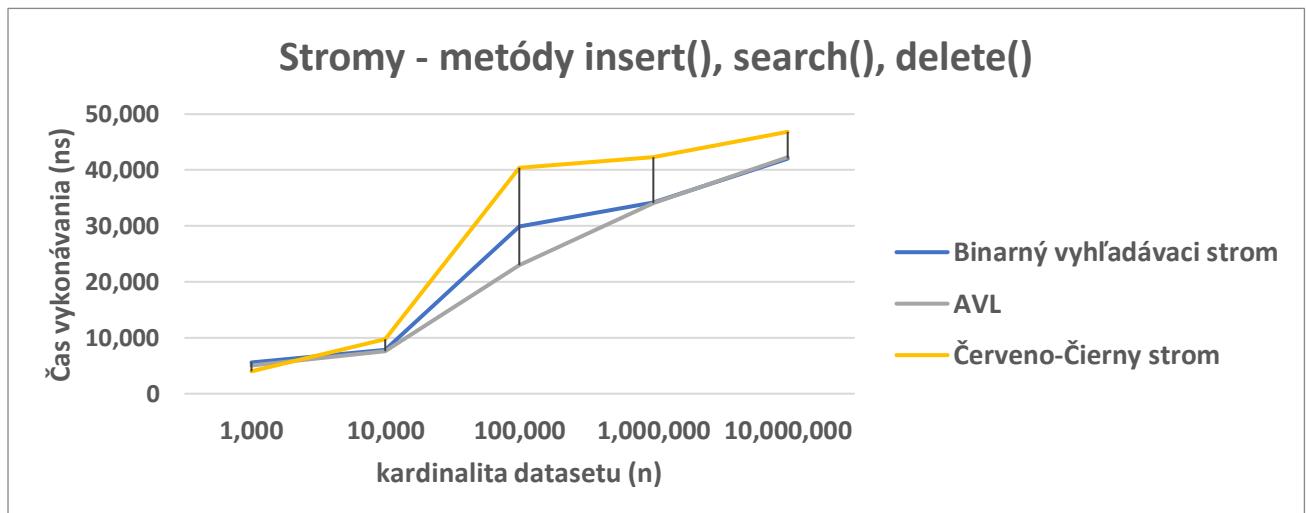
    endTime = System.nanoTime();
    duration = endTime - startTime;
    System.out.println("Time taken to tree with " + 1 + " element from " + (numberOfNodes+1) + ":" + duration + " nanoseconds");
    System.out.println();
}
```

Obrázok 37, metóda `testWholeTree()`

2.6.2.2 Grafy a štatistiká

Metódy `insert() → search() → delete()`, testoval sa jeden prvok pri rôznych mohutnostiach.

insert() + search() + delete()			
n	Binarný vyhľadávací strom	AVL	Červeno-Čierny strom
1000	5586	5042	4000
10000	7792	7542	9792
100000	29958	23000	40417
1000000	34250	34083	42291
10000000	42042	42291	46834



Prekvapivo Červeno-čierny strom má najhoršiu celkovú časovú komplexitu, avšak pri menších dátach najlepšiu, ale nie príliš. Najlepšie je na tom AVL strom, ktorý má najmenšie narastajúce výkyvy pri narastajúcim objeme dát.

2.6.3 Testovací scenár III.

Tento scenár slúži iba na testovanie funkčnosti algoritmu a duplicit vstupov, aby sa nestalo, že v stromoch by sa nachádzal rovnaký údaj viackrát alebo by celý program prestal fungovať. Taktiež tento test slúži na vizualizáciu usporiadania dát v stromoch.

2.6.3.1 Trieda Traverse

Trieda Traverse slúži na usporiadanie výpisu stromov. Sú známe tri spôsoby:

- Inorder ([obrázok 38](#), [obrázok 39](#))
- Preorder ([obrázok 40](#), [obrázok 41](#))
- Posorder ([obrázok 42](#), [obrázok 43](#))

```
// ##### inorder tree walk --> visiting left subtree, then the root, at the end the right subtree
// recursive method
3 usages  ± mariamatusiskova
void inorder(NodeOfTheTree actualNode, int space) {

    if (actualNode != null) {

        inorder(actualNode.left, space: space + 1);

        if (space != 0) {
            for (int i = 0; i < space; i++) {
                System.out.print("\t\t\t");
            }
            System.out.println("----- " + actualNode.data.value + " | " + actualNode.data.number);
        } else {
            // root
            System.out.println("INORDER ROOT: " + actualNode.data.value + " | " + actualNode.data.number);
        }

        // System.out.println(actualNode.data.value + " | " + actualNode.data.number);

        inorder(actualNode.right, space: space + 1);
    }
}

6 usages  ± mariamatusiskova
public void callInorder(BinarySearchTree tree) {
    System.out.println();
    inorder(tree.root, space: 0);
    System.out.println();
    System.out.println();
}
```

Obrázok 38, metóda `inorder()`

```
|----- ATYKD | 459
INORDER ROOT: OjGpD | 378
|----- Y77r6 | 45
|----- Xr7Yk | 190
|----- cPjBT | 876
|----- sPjEX | 887
```

Obrázok 39, výpis metódy inorder()

```
// ### preorder tree walk --> visiting root, then the left subtree, at the end the right subtree
// recursive method
3 usages ▲ mariamatusiskova
void preorder(NodeOfTheTree actualNode, int space) {

    if (actualNode != null) {

        if (space != 0) {
            for (int i = 0; i < space; i++) {
                System.out.print("\t\t\t");
            }
            System.out.println("----- " + actualNode.data.value + " | " + actualNode.data.number);
        } else {
            // root
            System.out.println("PREORDER ROOT: " + actualNode.data.value + " | " + actualNode.data.number);
        }

        preorder(actualNode.left, space: space + 1);
        preorder(actualNode.right, space: space + 1);
    }
}

no usages ▲ mariamatusiskova
public void callPreorder(BinarySearchTree tree) {
    System.out.println();
    preorder(tree.root, space: 0);
    System.out.println();
    System.out.println();
}
```

Obrázok 40, metóda preorder()

```
PREORDER ROOT: OjGpD | 378
|----- ATYKD | 459
|----- cPjBT | 876
|----- Y77r6 | 45
|----- Xr7Yk | 190
|----- sPjEX | 887
```

Obrázok 41, výpis metódy preorder()

```

// ### postorder tree walk --> visiting the left subtree, then the right subtree, the root node at the end
// recursive method
3 usages  ± mariamatusiskova
void postorder(NodeOfTheTree actualNode, int space) {

    if (actualNode != null) {

        postorder(actualNode.left, space: space + 1);
        postorder(actualNode.right, space: space + 1);

        if (space != 0) {
            for (int i = 0; i < space; i++) {
                System.out.print("\t\t\t\t");
            }
            System.out.println("----- " + actualNode.data.value + " | " + actualNode.data.number);
        } else {
            // root
            System.out.println("POSTORDER ROOT: " + actualNode.data.value + " | " + actualNode.data.number);
        }
    }
}

1 usage  ± mariamatusiskova
public void callPostorder(BinarySearchTree tree) {
    System.out.println();
    postorder(tree.root, space: 0);
    System.out.println();
    System.out.println();
}

```

Obrázok 42, metóda `postorder()`

```

| ----- ATYKD | 459
| ----- Xr7Yk | 190
| ----- Y77r6 | 45
| ----- sPjEX | 887
| ----- cPjBT | 876
POSTORDER ROOT: 0jGpD | 378

```

Obrázok 43, výpis metódy `postorder`

2.6.3.2 Implementácia kódu

```
public void testIfTreesAreWorkingCorrect() {  
  
    Traverse tr = new Traverse();  
  
    System.out.println();  
    System.out.println("***** BST *****");  
    System.out.println();  
  
    BinarySearchTree bst = new BinarySearchTree();  
  
    bst.callInsert(new Data(generateRandomString(), generateRandomNumber()));  
    Data specificBST = new Data(generateRandomString(), generateRandomNumber());  
    bst.callInsert(specific);  
    bst.callInsert(new Data(generateRandomString(), generateRandomNumber()));  
    for (int i = 0; i < 3; i++) {  
        bst.callInsert(new Data(generateRandomString(), generateRandomNumber()));  
    }  
  
    bst.callSearch(specificBST);  
  
    tr.callInorder(bst);  
    tr.callPreorder(bst);  
    tr.callPostorder(bst);  
  
    bst.callDelete(specificBST);  
    bst.callSearch(specificBST);  
  
    tr.callInorder(bst);  
    // tr.callPreorder(bst);  
    // tr.callPostorder(bst);
```

Obrázok 44, testovanie BVS

```
System.out.println();  
System.out.println("***** AVL *****");  
System.out.println();  
  
avl.callInsert(new Data(generateRandomString(), generateRandomNumber()));  
Data specificAVL = new Data(generateRandomString(), generateRandomNumber());  
avl.callInsert(specificAVL);  
avl.callInsert(new Data(generateRandomString(), generateRandomNumber()));  
avl.callInsert(specificAVL);  
avl.callInsert(new Data(generateRandomString(), generateRandomNumber()));  
avl.callInsert(new Data(generateRandomString(), generateRandomNumber()));  
avl.callInsert(new Data(generateRandomString(), generateRandomNumber()));  
  
avl.callSearch(specificAVL);  
  
avl.callHeight();  
  
tr.callInorder(avl);  
// tr.callPreorder(avl);  
// tr.callPostorder(avl);  
  
avl.callDelete(specificAVL);  
avl.callSearch(specificAVL);  
  
tr.callInorder(avl);  
// tr.callPreorder(avl);  
// tr.callPostorder(avl);  
  
avl.callHeight();
```

Obrázok 45, tetovanie AVL

```

System.out.println();
System.out.println("***** RBT *****");
System.out.println();

rbt.callInsert(new Data(generateRandomString(), generateRandomNumber()));
Data specificRBT = new Data(generateRandomString(), generateRandomNumber());
rbt.callInsert(specificRBT);
rbt.callInsert(new Data(generateRandomString(), generateRandomNumber()));
rbt.callInsert(specificRBT);
rbt.callInsert(new Data(generateRandomString(), generateRandomNumber()));
rbt.callInsert(new Data(generateRandomString(), generateRandomNumber()));
rbt.callInsert(new Data(generateRandomString(), generateRandomNumber()));

rbt.callSearch(specificRBT);

tr.callInorder(rbt);
// tr.callPreorder(rbt);
// tr.callPostorder(rbt);

rbt.callDelete(specificRBT);
rbt.callSearch(specificRBT);

tr.callInorder(rbt);
// tr.callPreorder(rbt);
// tr.callPostorder(rbt);

}

```

Obrázok 46, testovanie RBT

3 Hašovacie tabuľky

Hašovacia tabuľka je dátová štruktúra, ktorá vyhľadáva páru kľúču alebo hodnote. Mapuje hašované kľúče na vygenerovaných indexoch, tam ich aj ukladá.

V tomto projekte sa riešilo odhašovanie takto: **(kľúč & 0x7fffffff) % (veľkosť tabuľky - 1)**. Vysvetlenie spomenutého kódu je, že najskôr prevediem bitový posun kľúča, tým zaistíme, že heš bude mať len kladné hodnoty, následne pomocou modula sa vypočíta nový index. Minus jeden je tam preto, že je o efektívnejšia voľba, ak do tabuľky posielame len párske veľkosti celkovej kapacity tabuľky. Samotný kľúč je zahašovaný v dátach tabuľky. ([obrazok 47](#))

Časová zložitosť v priemere je **O(1)**, v najhorších prípadoch **O(n)**.

3.1 Dáta

Dátovým typom pre prácu s hašovacími tabuľkami je vytvorená trieda **DataHashTable**. V nej sa nachádza viacero druhov údajov. Generujú sa náhodne v testovacom programme a následne sa posielajú na spracovanie do vybraných stromov. V [obrázku 47](#) je znázornené s akými dátami sa v projekte zaoberáme.

```
package sk.stuba.fiit.hashTable;

4.2 usages  ↗ mariamatusiskova
public class DataHashTable implements Comparable<DataHashTable> {

    // random String
    public String value;

    // key hashed from string
    37 usages
    public int key;

    11 usages  ↗ mariamatusiskova
    public DataHashTable(int key, String value) {
        this.value = value;
        this.key = key;
    }

    ↗ mariamatusiskova
    public int compareTo(DataHashTable other){
        if(this.key > other.key) return 1;
        if(this.key == other.key) return 0;
        return -1;
    }
}
```

Obrázok 47, znázornenie triedy DataHashTable

Pričom dáta sa náhodne generujú v testovacom programe. Používajú sa rovnaké funkcie ako pri generovaní údajov do stromov. S tým, že náhodne vygenerované číslo sa priamo hešuje. ([obrázok 3](#), [obrázok 4](#))

3.2 Reťazenie

Reťazenie rieši kolízie hašovacích tabuliek, tie nastávajú vtedy keď sa viacero kľúčov mapuje na ten istý index. Na vyriešenie tohto problému sa používa Linked List z `java.util.*`, ale môže sa využiť aj obyčajné pole v poli. No Linked List je v tomto prípade efektívnejší, aj čo sa týka časovej komplexity.

Ako náhle nastane kolízia, prvok s rovnakým indexom sa pridá na koniec prepojeného zoznamu v tom indexe. Teda na indexe je ako keby ďalšie pole určené na tieto dátu.

Časová zložitosť v priemere je **$O(1 + \alpha)$** , kde α je faktor zatiaženia (pomer počtu prvkov v hašovacej tabuľke k počtu slotov), najhorší prípad, ktorý môže nastať je stále **$O(n)$** .

V zadaní je riešené aj zväčšovanie a zmenšovanie tabuľky podľa pomeru naplnenia, horná hranica je $0.7f$, kde f znamená len zaokrúhlenie na tri desatinné miesta a 0.7 znázorňuje 70% zaplnenia tabuľky. Spodná hranica je $0.3f$.

3.2.1 Implementácia kódu

Úvod do oddelovacej zretazenej tabuľky. V konštruktore sa nastavuje celková kapacita hašovacej tabuľky. ([obrázok 48](#))

```
15 usages ▲ mariamatusiskova
public class SeparateChainingHashTable {

    // dynamic array of objects
    18 usages
    public ArrayList<LinkedList<DataHashTable>> buckets = new ArrayList<>();
    4 usages
    public int sentSizeOfTable;
    9 usages
    public int tableSize;
    6 usages
    public int countNodes;

    // default constructor
    6 usages ▲ mariamatusiskova
    public SeparateChainingHashTable(int tableSize) {

        this.sentSizeOfTable = tableSize;
        this.tableSize = tableSize;
        this.countNodes = 0;

        for (int i = 0; i < tableSize; i++) {
            buckets.add(new LinkedList<>());
        }
    }
}
```

Obrázok 48, premenné a konštruktor zretazenia

Metóda *insert()* (hl'adat') pridáva dátá do tabuľky a kontroluje jej veľkosť (či je väčšina tabuľky už zaplnená), duplicita dát je ošetrená hned' na začiatku funkcie.

([obrázok 49](#))

```
8 usages ▲ mariamatusiskova
public void insert(DataHashTable data) {

    // hexadecimal number provides only positive numbers
    int index = (data.key & 0xffffffff) % (tableSize-1);

    // no duplicity
    for (int i = 0; i < buckets.get(index).size(); i++) {
        if (data.key == buckets.get(index).get(i).key) {
            buckets.get(index).get(i).value = data.value;
            return;
        }
    }

    buckets.get(index).add(data);
    countNodes++;

    if (((float) countNodes) / tableSize > 0.7f) {
        resize( sizeOfTable: this.tableSize*2);
    }
}
```

Obrázok 49. metóda *insert()* separate chaining

Metóda `search()` (hľadat') je typu **boolean**, je pomerne jednoduchá a pomocou **for** cyklu prehľadáva spájaný zoznam pokiaľ nenájde hľadanú hodnotu, inak vracia **false**. ([obrázok 50](#))

```
4 usages ▲ mariamatusiskova
public boolean search(int key) {

    int index = (key & 0x7fffffff) % (tableSize-1);

    if (buckets.get(index) == null) {
        return false;
    }

    for (int i = 0; i < buckets.get(index).size(); i++) {
        DataHashTable currentData = buckets.get(index).get(i);
        if (key == buckets.get(index).get(i).key) {
            return true;
        }
    }

    return false;
}
```

Obrázok 50, metóda `search()` separate chaining

Metóda `delete()` (vymazat') je podobná funkcie `search`, ale bez návratovej hodnoty. Po vymazaní odpočítava **počet vložených dát**, pomocou tejto premennej vieme kontrolovať nadmerný priestor v tabuľke. ([obrázok 51](#))

```
3 usages ▲ mariamatusiskova
public void delete(int key) {

    int index = (key & 0x7fffffff) % (tableSize-1);

    if (buckets.get(index) == null) {
        return;
    }

    for (int i = 0; i < buckets.get(index).size(); i++) {
        if (key == buckets.get(index).get(i).key) {
            buckets.get(index).removeFirstOccurrence(buckets.get(index).get(i));
            countNodes--;
        }
    }

    if (((float) countNodes) / tableSize < 0.3f) {
        resize( sizeOfTable: this.tableSize/2 );
    }
}
```

Obrázok 51, metóda `delete()` separate chaining

Metóda `resize()` (zmeniť veľkosť) po vstupe bud' kapacitu tabuľky zväčší alebo zmenší. Na tento účel sa vytvorí nové pole, zaplní sa a prepíše to pôvodné. ([obrázok 52](#))

```
2 usages ▲ mariamatusiskova
void resize(int sizeOfTable) {

    ArrayList<LinkedList<DataHashTable>> newTable = new ArrayList<>();

    // create empty table of new size
    for (int i = 0; i < sizeOfTable; i++) {
        newTable.add(new LinkedList<>());
    }

    ArrayList<LinkedList<DataHashTable>> oldTable = buckets;
    this.buckets = newTable;
    this.tableSize = sizeOfTable;
    this.countNodes = 0;

    for (LinkedList<DataHashTable> bucket : oldTable) {
        for (DataHashTable data : bucket) {
            insert(data);
        }
    }
}
```

Obrázok 52, metoda `resize()` separate chaining

3.3 Otvorené adresovanie

Lineárne otvorené adresovanie je ďalší algoritmus na kolízie hašovacích tabuľiek. Ked' dôjde ku kolízii, otvorené adresovanie vyhľadá najbližšiu pozíciu na vloženie prvku do tabuľky a to zvyšovaním indexu o pevnú hodnotu.

Toto riešenie kolízie nie je najefektívnejšie, pretože pri rovnakých indexoch môže dochádzať ku dlhším sekvenciám a to môže ovplyvniť rýchlosť výkonu. Na predĺženie tohto scenáru sa dá použiť algoritmus dvojté hašovanie. Treba, ale podotknúť, že pri práci s menšími dátami, toto riešenie úplne stačí.

Časová komplexita v priemere je **O(1)** za rovnomeného vkladania dát, v najhoršom prípade môže stúpnuť na **O(n)**.

Podobne ako u zreťazenia, menenie veľkosti tabuľky funguje na rovnakom princípe. Hlavným rozdielom je, že pri tomto algoritme sa už nepoužíva spájaný zoznam, ale `ArrayList`.

3.3.1 Implementácia kódu

Úvod do lineárneho otvoreného adresovania. V konštruktore sa nastavuje kapacita tabuľky. Je tam aj premenná **deleted**, ktorá slúži na zlepšenie časovej efektivity kódu. ([obrázok 53](#))

```

15 usages  ↗ mariamatusiskova
public class OpenAddressingHashTable {

    17 usages
    public ArrayList<DataHashTable> buckets = new ArrayList<>();
    12 usages
    public int tableSize;
    4 usages
    public int sentSizeOfTable;
    6 usages
    public int countNodes;
    3 usages
    public DataHashTable deleted = new DataHashTable(-1, "deleted");

    // default constructor
    6 usages  ↗ mariamatusiskova
    public OpenAddressingHashTable(int tableSize) {

        this.sentSizeOfTable = tableSize;
        this.tableSize = tableSize;
        this.countNodes = 0;

        for (int i = 0; i < tableSize; i++) {
            buckets.add(null);
        }
    }
}

```

Obrázok 53, premenné a konštruktor linear probing

Metóda `insert()` (hl'adat') vkladá nový prvok do tabuľky, ak sa index nachádza na prvku `deleted`, automaticky sa prepíše na nové dáta. Kontroluje sa aj kapacita tabuľky. ([obrázok 54](#))

```

8 usages  ↗ mariamatusiskova
public void insert(DataHashTable data) {

    if (data == null) {
        return;
    }

    int index = (data.key & 0xffffffff) % (tableSize-1);

    while (buckets.get(index) != null && buckets.get(index).key != data.key) {
        if (buckets.get(index).key == deleted.key) {
            break;
        }
        index = (index + 1) % (tableSize-1);
    }

    if (buckets.get(index) == null || buckets.get(index).key == deleted.key) {
        countNodes++;
    }
    buckets.set(index, data);

    if (((float)countNodes)/tableSize > 0.7f) {
        resize( sizeOfTable: this.tableSize*2);
    }
}

```

Obrázok 54, metóda `insert()` linear probing

Metóda `search()` (hl'adat') typu návratovej hodnoty **boolean**. Hľadá pomocou cyklu **while** a posúva sa vždy **o jeden prvok** pokým nenájde hľadanú hodnotu alebo neprejde celú tabuľku. ([obrázok 55](#))

```
4 usages  ↗ mariamatusiskova
public boolean search(int key) {

    int index = (key & 0xffffffff) % (tableSize-1);

    while (buckets.get(index) != null) {

        if (buckets.get(index).key == key) {
            DataHashTable currentNode = buckets.get(index);
            return true;
        }
        index = (index + 1) % (tableSize-1);
    }

    return false;
}
```

Obrázok 55. metóda `search()` linear probing

Metóda `delete()` (vymazat') prvok, ktorý sa vymaže sa prepíše na **deleted**, urýchľuje to vkladanie nových dát. ([obrázok 56](#))

```
3 usages  ↗ mariamatusiskova
public void delete(int key) {

    int index = (key & 0xffffffff) % (tableSize-1);

    while (buckets.get(index) != null) {

        if (buckets.get(index).key == key) {
            buckets.set(index, deleted);
            countNodes--;
        }
        index = (index + 1) % (tableSize-1);
    }

    if (((float)countNodes)/tableSize < 0.3f) {
        resize( sizeOfTable: this.tableSize/2);
    }
}
```

Obrázok 56, metóda `delete()` linear probing

Metóda `resize()` zväčší alebo zmenší tabuľku a prepíše potrebné hodnoty. Pomocou funkcie `insert()` sa vložia elementy naspäť už do novej tabuľky. ([obrázok 57](#))

```
2 usages  mariamatusiskova
private void resize(int sizeOfTable) {

    ArrayList<DataHashTable> newBuckets = new ArrayList<>();
    for (int i = 0; i < sizeOfTable; i++) {
        newBuckets.add(null);
    }
    ArrayList<DataHashTable> oldBuckets = buckets;

    this.buckets = newBuckets;
    this.tableSize = sizeOfTable;
    this.countNodes = 0;

    for (DataHashTable data : oldBuckets) {
        insert(data);
    }
}
```

Obrázok 57, metóda `resize()` linear probing

3.4 Testovanie hašovacích tabuliek

V testovacom programe sa pracovalo s kardinalitou – 1 000, 10 000, 100 000, 1 000 000, 10 000 000. Dáta boli náhodne vygenerované, viac je o tom napísané v [podkapitole 3.1](#).

3.4.1 Testovací scenár I.

V tomto scenári sa testuje každá metóda individuálne. Do `ArrayListu`, ktorý je použitý k knižnici `java.util.*` sa vložili náhodné hodnoty a potom sa zvlášť u každej metódy testovali. Sledovalo sa ich správanie pri rôznych mohutnostiach, teda časová komplexita, ktorá sa spriemerovala na jeden prvok. Výsledok tohto testu by mal mať časovú zložitosť v priemere **O(1)** a v najhoršom prípade **O(n)**.

3.4.1.1 Implementácia kódu

Čas sa začal merat' od volania funkcie po jej koniec, následne sa spriemeroval. Tabuľky s riešeniami na kolízie sa testovali osobitnými metódami.

- Test na reťazenie ([obrázok 58](#))
- Test na otvorené adresovanie ([obrázok 59](#))

```
1 usage  ± mariamatusiskova
public void testSeparateChainingHashTableInsertDeleteSearchIndividually(SeparateChainingHashTable table, int numberofNodes, List<DataHashTable> datahash) {

    final int LOOP = datahash.size();

    for (int i = 0; i < (table.sizeOfTable-1); i++) {
        table.insert(new DataHashTable(generateRandomNumber(), generateRandomString()));
    }

    // insert testing
    startTime = System.nanoTime();
    for (int i = 0; i < LOOP; i++) {
        table.insert(datahash.get(i));
    }
    endTime = System.nanoTime();
    duration = endTime - startTime;

    System.out.println("Time taken to insert " + 1 + " element from " + (numberofNodes+1) + ": " + duration + " nanoseconds");
    System.out.println("Average time of insert element: " + duration/LOOP + " nanoseconds");
    System.out.println();

    // search testing
    startTime = System.nanoTime();
    for (int i = 0; i < LOOP; i++) {
        table.search(datahash.get(i).key);
    }
    endTime = System.nanoTime();
    duration = endTime - startTime;
    System.out.println("Time taken to search " + 1 + " element from " + (numberofNodes+1) + ": " + duration + " nanoseconds");
    System.out.println("Average time of search element: " + duration/LOOP + " nanoseconds");
    System.out.println();

    // delete testing
    startTime = System.nanoTime();
    for (int i = 0; i < LOOP; i++) {
        table.delete(datahash.get(i).key);
    }
    endTime = System.nanoTime();
    duration = endTime - startTime;
    System.out.println("Time taken to delete " + 1 + " element from " + (numberofNodes+1) + ": " + duration + " nanoseconds");
    System.out.println("Average time of delete element: " + duration/LOOP + " nanoseconds");
    System.out.println();
}
```

Obrázok 58. metóda `testSeparateChainingHashTableInsertDeleteSearchIndividually()`

```

1 usage  ~ mariamatusiskova
public void testOpenAddressingHashTableInsertDeleteSearchIndividually(OpenAddressingHashTable table, int numberOfNodes, List<DataHashTable> datahash) {

    final int LOOP = datahash.size();

    for (int i = 0; i < (table.sentSizeOfTable-1); i++) {
        table.insert(new DataHashTable(generateRandomNumber(), generateRandomString()));
    }

    // insert testing
    startTime = System.nanoTime();
    for (int i = 0; i < LOOP; i++) {
        table.insert(datahash.get(i));
    }
    endTime = System.nanoTime();
    duration = endTime - startTime;
    System.out.println("Time taken to insert " + 1 + " element from " + (numberOfNodes+1) + ": " + duration + " nanoseconds");
    System.out.println("Average time of insert element: " + duration/LOOP + " nanoseconds");
    System.out.println();

    // search testing
    startTime = System.nanoTime();
    for (int i = 0; i < LOOP; i++) {
        table.search(datahash.get(i).key);
    }
    endTime = System.nanoTime();
    duration = endTime - startTime;
    System.out.println("Time taken to search " + 1 + " element from " + (numberOfNodes+1) + ": " + duration + " nanoseconds");
    System.out.println("Average time of search element: " + duration/LOOP + " nanoseconds");
    System.out.println();

    // delete testing
    startTime = System.nanoTime();
    for (int i = 0; i < LOOP; i++) {
        table.delete(datahash.get(i).key);
    }
    endTime = System.nanoTime();
    duration = endTime - startTime;
    System.out.println("Time taken to delete " + 1 + " element from " + (numberOfNodes+1) + ": " + duration + " nanoseconds");
    System.out.println("Average time of delete element: " + duration/LOOP + " nanoseconds");
    System.out.println();
}
}

```

Obrázok 59. metóda `testOpenAddressingHashTableInsertDeleteSearchIndividually()`

Volanie testu vo funkcií `main()` ([obrázok 60](#))

```

SeparateChainingHashTable separateChainingHashTable_10000 = new SeparateChainingHashTable(numberOfNodes_10000);
OpenAddressingHashTable openAddressingHashTable_10000 = new OpenAddressingHashTable(numberOfNodes_10000);

System.out.println("***** Separate Chaining - testing methods individually *****");
test.testSeparateChainingHashTableInsertDeleteSearchIndividually(separateChainingHashTable_1000, numberOfNodes_1000, datahash);
test.testSeparateChainingHashTableInsertDeleteSearchIndividually(separateChainingHashTable_10000, numberOfNodes_10000, datahash);

System.out.println("***** Linear Probing - testing methods individually *****");
test.testOpenAddressingHashTableInsertDeleteSearchIndividually(openAddressingHashTable_1000, numberOfNodes_1000, datahash);
test.testOpenAddressingHashTableInsertDeleteSearchIndividually(openAddressingHashTable_10000, numberOfNodes_10000, datahash);

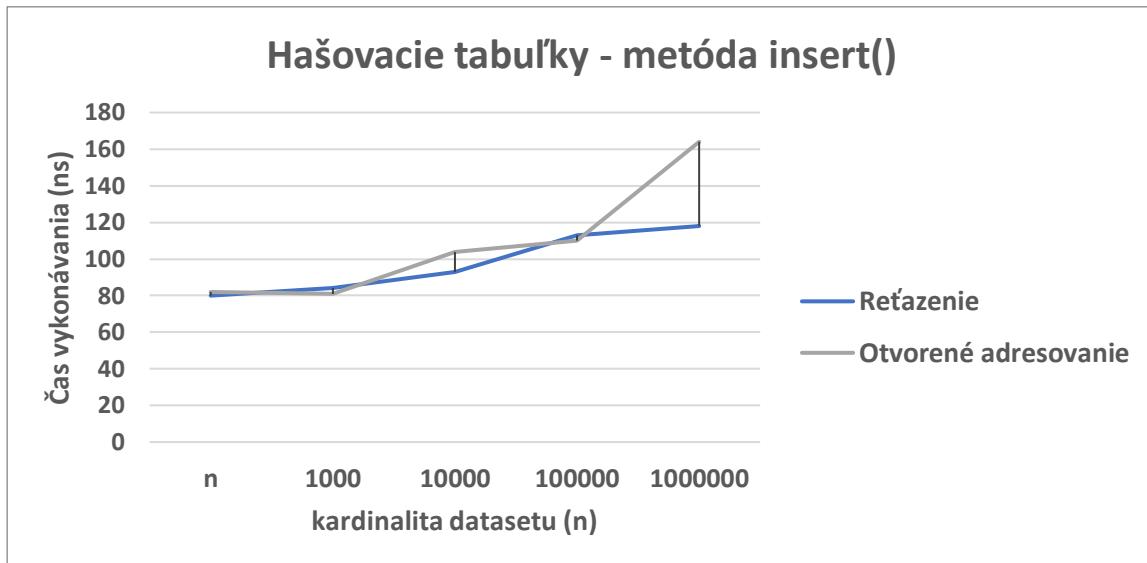
```

Obrázok 60, testovací program

3.4.1.2 Grafy a štatistika

Metóda *insert()*, testovalo sa 200 vložených dát a spriemerovalo sa to na jeden element výpočtom celkového času trvania /200 v nanosekundách.

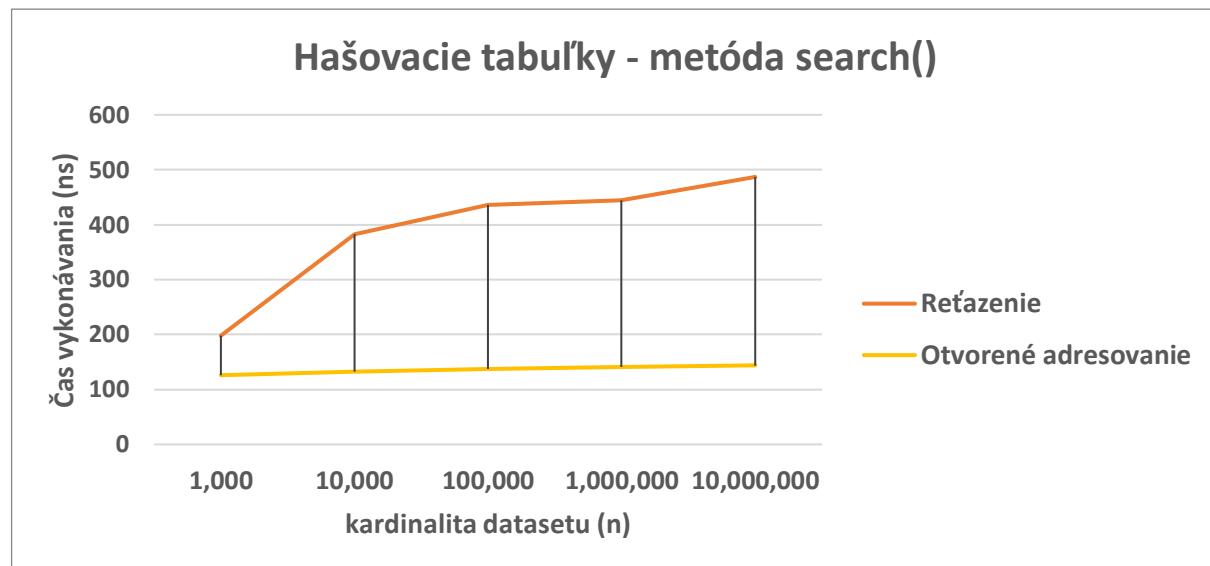
Metóda <i>insert()</i>		
n	Ret'azenie	Otvorené adresovanie
1000	80	82
10000	84	81
100000	93	104
1000000	113	110
10000000	118	164



Z výhodnotenie testu *insert()* metódy je viditeľné, že reťazenie pri zvyšovaní mohutnosti rastie pomerne súvisle. Pri otvorenom adresovaní sú menšie výkyvy, to môže byť už spomínaným problém so zhľukovaním.

Metóda `search()`, testovalo sa 200 vložených dát a spriemerovalo sa to na jeden element výpočtom **celkového času trvania /200** v nanosekundách.

Metóda <code>search()</code>		
n	Ret'azenie	Otvorené adresovanie
1000	198	126
10000	382	133
100000	436	137
1000000	445	141
10000000	487	144



Tu je naopak otvorená adresovanie omnoho efektívnejšie oproti ret'azenu. Dôvodom môže byť, že pri zret'azení program prehľadáva link list, kým nedosiahne jeho koniec, pri takom množstve dát môže nastáť vysoké zaťaženie.

Metóda `delete()`, testovalo sa 200 vložených dát a spriemerovalo sa to na jeden element výpočtom **celkového času trvania /200** v nanosekundách.

Metóda <code>delete()</code>		
n	Ret'azenie	Otvorené adresovanie
1000	401	346
10000	16300	16213
100000	92187	76665
1000000	187225	123640
10000000	1861111	507947



Od 1 000 do 1 000 000 sú grafy pomerne súvislé, ale pri vyššej mohutnosti je zrejmé, že efektívnejšie riešenie je otvorené adresovanie napriek zhľukovaniu.

3.4.2 Testovací scenár II.

Druhý testovací scenár sa zaoberá celkovou časovou zložitosťou metód `insert() → search() → delete()`. Sleduje sa ich správanie pri rôznych kardinalitách dát, keď sa po ich vložení vloží do nich ešte plus jeden prvok a ten sa práve testuje a časuje. Algoritmická zložitosť by sa mala rovnať **O(1)**.

3.4.2.1 Implementácia kódu

Testovanie sa vŕahuje na jeden prvok, riešenie kolízii sa testuje v osobitných metódach, ale s rovnakým prvkom. ([obrázok 61](#), [obrázok 62](#))

```
1 usage  ± mariamatusiskova
public void testWholeSeparateChaining(SeparateChainingHashTable table, int numberofNodes) {
    for (int i = 0; i < (table.sentSizeOfTable-1); i++) {
        table.insert(new DataHashTable(generateRandomNumber(), generateRandomString()));
    }

    startTime = System.nanoTime();
    table.insert(specificHashTable);
    table.search(specificHashTable.key);
    table.delete(specificHashTable.key);

    endTime = System.nanoTime();
    duration = endTime - startTime;
    System.out.println("Time taken to separate chaining hash table with " + 1 + " element from " + (numberofNodes+1) + ":" + duration + " nanoseconds");
    System.out.println();
}
```

Obrázok 61, metóda testWholeSeparateChainig()

```

1 usage  ~ mariamatusiskova
public void testWholeLinearProbing(OpenAddressingHashTable table, int numberOfNodes) {

    for (int i = 0; i < (table.sentSizeOfTable-1); i++) {
        table.insert(new DataHashTable(generateRandomNumber(), generateRandomString()));
    }

    startTime = System.nanoTime();

    table.insert(specificHashTable);
    table.search(specificHashTable.key);
    table.delete(specificHashTable.key);

    endTime = System.nanoTime();
    duration = endTime - startTime;
    System.out.println("Time taken to linear probing hash table one " + 1 + " element from " + (numberOfNodes+1) + ":" + duration + " nanoseconds");
    System.out.println();
}

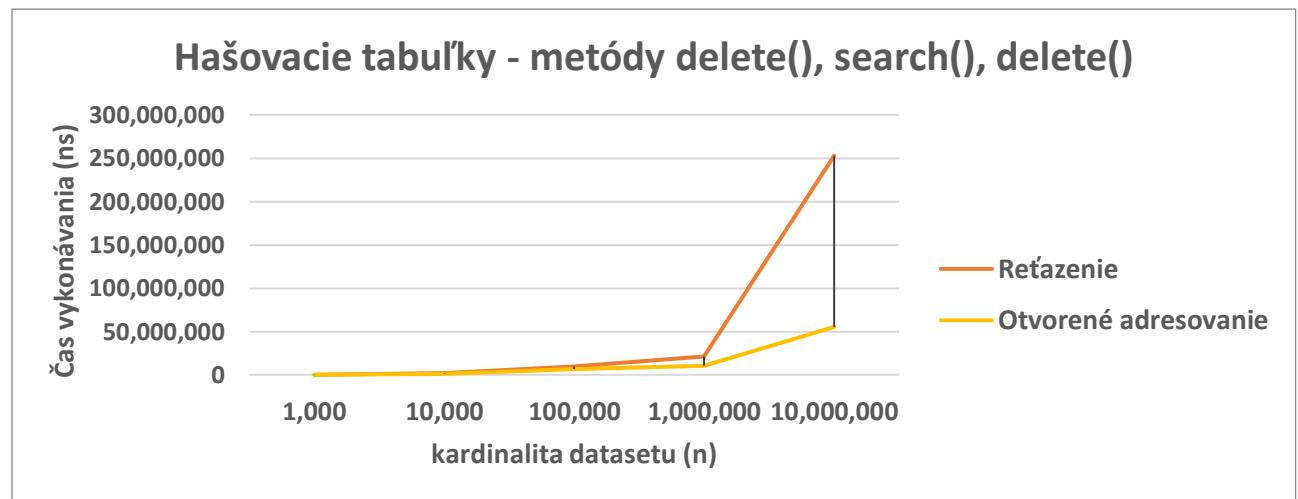
```

Obrázok 62, metóda `test wholeLinearProbing()`

3.4.2.2 Grafy a štatistiky

Metódy `insert() → search() → delete()`, testoval sa jeden prvok pri rôznych mohutnostiach.

insert() + search() + delete()		
n	Ret'azenie	Otvorené adresovanie
1000	8709	329
10000	1974875	948125
100000	9852708	6308417
1000000	21562417	10357208
10000000	252875708	55234084



Z grafu vyplýva, že otvorené adresovanie je efektívnejšie ako zret'azenie.

3.4.3 Testovací scenár III.

Tento scenár slúži iba na testovanie funkčnosti algoritmu a duplícít vstupov, aby sa nestalo, že v hašovacích tabuľkách by sa nachádzal rovnaký údaj viackrát alebo by celý program prestal fungovať. Taktiež tento test slúži na vizualizáciu usporiadania dát v tabuľkách, najmä ak sa zavolá aj funkcia *resize()*.

3.4.3.1 Implementácia kódu

Test na reťazenie. ([obrátok 63](#))

```
1 usage  ~ mariamatusiskova
public void testIfHashTablesAreWorkingCorrect() {

    System.out.println();
    System.out.println("***** Separate Chaining *****");
    System.out.println();

    SeparateChainingHashTable chaining = new SeparateChainingHashTable( tableSize: 10);

    for (int i = 0; i < (chaining.sizeOfTable-1); i++) {
        chaining.insert(new DataHashTable(generateRandomNumber(), generateRandomString()));
    }

    DataHashTable separateChaining = new DataHashTable(generateRandomNumber(), generateRandomString());
    chaining.insert(separateChaining);
    chaining.insert(separateChaining);

    int j = 0;
    System.out.println();
    for (LinkedList<DataHashTable> bucket : chaining.buckets) {
        System.out.print("Bucket " + j + ": ");
        for (DataHashTable dataOfTable : bucket) {
            System.out.print(dataOfTable.value + " | " + dataOfTable.key + " | " + " ");
        }
        System.out.println();
        j++;
    }

    System.out.println();

    System.out.println();

    chaining.search(separateChaining.key);

    chaining.delete(separateChaining.key);

    chaining.search(separateChaining.key);

    int k = 0;
    System.out.println();
    for (LinkedList<DataHashTable> bucket : chaining.buckets) {
        System.out.print("Bucket " + k + ": ");
        for (DataHashTable dataOfTable : bucket) {
            System.out.print(dataOfTable.value + " | " + dataOfTable.key);
        }
        System.out.println();
        k++;
    }
}
```

Obrázok 63, metóda na testovanie korektnosti hašovacích tabuľiek - reťazenie

Test na lineárne otvorené adresovanie. ([obrázok 64](#))

```
System.out.println();
System.out.println("***** Open Addressing *****");
System.out.println();

OpenAddressingHashTable openAddressingHashTable = new OpenAddressingHashTable( tableSize: 10);

for (int i = 0; i < (openAddressingHashTable.sentSizeOfTable-1); i++) {
    openAddressingHashTable.insert(new DataHashTable(generateRandomNumber(), generateRandomString()));
}

DataHashTable linearProbing = new DataHashTable(generateRandomNumber(), generateRandomString());
openAddressingHashTable.insert(linearProbing);
openAddressingHashTable.insert(linearProbing);

int m = 0;
for (DataHashTable dataOfTable : openAddressingHashTable.buckets) {
    System.out.print("Bucket " + m + ": ");
    if(dataOfTable == null){
        System.out.print("null");
    } else{
        System.out.print(dataOfTable.value + " | " + dataOfTable.key + " | " + " ");
    }
    System.out.println();
    m++;
}

System.out.println();

openAddressingHashTable.search(linearProbing.key);

openAddressingHashTable.delete(linearProbing.key);

openAddressingHashTable.search(linearProbing.key);

int n = 0;
for (DataHashTable dataOfTable : openAddressingHashTable.buckets) {
    System.out.print("Bucket " + n + ": ");
    if(dataOfTable == null){
        System.out.print("null");
    } else {
        System.out.print(dataOfTable.value + " | " + dataOfTable.key + " | " + " ");
    }
    System.out.println();
    n++;
}

}
```

Obrázok 64, metóda na testovanie korektnosti hašovacích tabuľiek – otvorené adresovanie

4 Záver

Po preskúmaní vybraných algoritmov sa zistilo, ktoré štruktúry sú vhodné na rôzne metódy. Podľa toho na čo sa chce program najviac orientovať.

Pri stromoch. Je celkovo najlepší AVL strom, následne červeno-čierny strom a ako posledný nevyvážený binárny vyhľadávací strom. Z grafu bolo vidieť, že AVL strom narastal čo najviac súvisle, kolízie vznikali pri BVS. Celkovo najpomalší bol červeno-čierny strom, avšak pri testovaní jednotlivých metód, mal časy pomerne dobré.

V hašovacích tabuľkách sa ako efektívnejšie riešenie kolízii prejavilo otvorené adresovanie. Reťazenie vynikalo iba pri funkcií *insert()*.

Projekt bol realizovaný v jazyku Java, je možné, že v iných jazykoch ako napríklad C sa môže časová efektivita meniť. Pretože knižnice tam môžu byť implementované inak.

Zdroje

Binárny vyhľadávací strom

- [https://sd.blackball.lv/library/Introduction_to_Algorithms_Third_Edition_\(2009\).pdf](https://sd.blackball.lv/library/Introduction_to_Algorithms_Third_Edition_(2009).pdf)
- <https://www.geeksforgeeks.org/deletion-in-binary-search-tree/>
- <https://www.baeldung.com/java-binary-tree>
- <https://www.youtube.com/watch?v=ytfEFJAhMgo>

AVL strom

- <https://www.happycoders.eu/algorithms/avl-tree-java/>

Červeno-čierny strom

- <https://www.happycoders.eu/algorithms/red-black-tree-java/>

Dáta stromu

- <https://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec16/lec16-15.html>

Prechádzanie stromov

- <https://stackoverflow.com/questions/4965335/how-to-print-binary-tree-diagram-in-java>

Dáta v tabuľkách

- <https://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec16/lec16-15.html>

Otvorené adresovanie

- <https://www.geeksforgeeks.org/implementing-hash-table-open-addressing-linear-probing-cpp/>

Ret'azenie

- [https://sd.blackball.lv/library/Introduction_to_Algorithms_Third_Edition_\(2009\).pdf](https://sd.blackball.lv/library/Introduction_to_Algorithms_Third_Edition_(2009).pdf)
- <https://medium.com/omarelgabrys-blog/hash-tables-2fec6870207f>
- Adam Gábor, AIS ID 116174, DSA documentation