Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Dátové štruktúry a algoritmy

# Zadanie č.1 – Dynamické množiny

Akademický rok 2021/2022

**Meno:** Ján Ágh
**Cvičiaci:** Ing. Lukáš Kohútka, PhD.

**Dátum:** 20.3.2022
**Počet strán:** 18

# Obsah

# 1 Úvod

Cieľom prvého zadania z predmetu Dátové štruktúry a algoritmy bolo navrhnúť vlastnú implementáciu binárneho vyhľadávacieho stromu s dvoma rôznymi vyvažovacími algoritmami a hashovaciu tabuľku s dvoma odlišnými spôsobmi riešenia kolízií, porovnať ich z hľadiska efektivity operácií insert, delete a search a vyhodnotiť získané informácie. Na implementáciu uvedených dátových štruktúr bol použitý programovací jazyk Java 17.0.2 v kombinácií s prostredím Eclipse IDE.

# 2 Binárne vyhľadávacie stromy

Binárny vyhľadávací strom je populárna dátová štruktúra pozostávajúca z uzlov a spojeniami medzi nimi, pričom ľavý podstrom vybraného uzla vždy obsahuje uzly s prístupovým kľúčom menším ako kľúč uzla, pravý podstrom obsahuje uzly s prístupovým kľúčom väčším ako kľúč uzla a zároveň ľavý aj pravý podstrom musia byť binárne vyhľadávacie stromy. Ak binárny vyhľadávací strom neobsahuje žiadnen algoritmus vyvažovania, časová zložitosť operácií vyhľadávania, vkladania a mazania uzlov je v najlepšom prípade O(log n), v najhoršom O(n).

## 2.1 AVL strom

AVL strom je špeciálna verzia binárneho vyhľadávacieho stromu so samovyvažovacím algoritmom. Bol vynájdený sovietskymi vynálezcami a považuje sa za prvý algoritmus svojho druhu. Jeho základnou vlastnosťou je výšková vyváženosť – pre každý uzol musí platiť, že koeficient jeho vyváženia (rozdiel výšky jeho ľavého a pravého podstromu) musí patriť do intervalu <-1, 1>. V prípade, ak táto podmienka nie je splnená, dochádza k vyváženiu stromu jednou alebo dvoma rotáciami. K rotácii dochádza ihneď po zistení nevyváženosti, t.j. koeficient vyváženia by nikdy nemal dosahovať hodnoty menšie ako -2 alebo väčšie ako 2. Z tohto dôvodu je časová zložitosť operácií vyhľadávania, vkladania aj mazania uzlov v najlepšom aj najhoršom prípade O(log n).

### *2.1.1 Vlastná implementácia*

Moja implementácia AVL stromu pozostáva z dvoch tried. Trieda AVLTree obsahuje hlavné metódy insert, delete a search a iné pomocné metódy, ako napríklad realizácia rotácií alebo hľadanie najväčšieho prvku v danom podstrome. Druhá trieda s názvom AVLNode predstavuje uzol stromu a z tohto dôvodu slúži na uloženie unikátneho kľúča spolu s údajmi, pričom taktiež obsahuje odkazy na svoj ľavý a pravý podstrom a rodičovský uzol.

**Insert**

Úlohou funkcionality insert je vloženie nového uzla do stromu. Skladá sa z dvoch metód, verejnej a privátnej. Po zavolaní verejnej metódy (Obr. 2.1) sa najprv zistí, či sa v strome už nachádzajú nejaké uzly. Ak je strom prázdny, z nového uzla sa stane koreň, v opačnom prípade sa uzol odovzdá privátnej funkcii (Obr 2.2), ktorej hlavnou úlohou je nájsť správne miesto, kam môže byť daný uzol vložený.

```
public void insert(int ID, String name) {
    if(this.mainNode != null) this.applyBalance(this.insertNewNode(ID, name, this.mainNode), ID, false);
    else this.mainNode = new TreeNode(ID, name);
}
```

*Obr. 2.1 Metóda insert triedy AVLTree*

Hľadanie správneho miesta je riešené iteratívne a funguje podľa všeobecných pravidiel binárnych vyhľadávacích stromov. Porovnáva sa kľúč nového uzla s kľúčom práve analyzovaného uzla. V prípade, ak je kľúč nového uzla väčší, posunieme sa do pravého podstromu, ak je menší, do ľavého podstromu. Táto aktivita pokračuje dovtedy, kým nenájdeme vhodné prázdne miesto.

```
public TreeNode insertNewNode(int ID, String name, TreeNode node) {
    while(node != null) {
        if(ID > node.getID()) {
            if(node.getRight() == null) {
                node.setRight(new TreeNode(ID, name));
                return node.getRight();
            }
            node = node.getRight(); continue;
        }
        else if(ID < node.getID()) {
            if(node.getLeft() == null) {
                node.setLeft(new TreeNode(ID, name));
                return node.getLeft();
            }
            node = node.getLeft(); continue;
        }
        node = null;
    }
    return node;
}
```

*Obr. 2.2 Metóda insertNewNode abstraktnej triedy BinarySearchTree*

Po vložení nového prvku je na rade zistiť, či je strom stále výškovo vyvážený. Na túto činnosť slúži metóda applyBalance, ktorá iteratívne prechádza z najnižšie položeného uzla / listu (v tomto prípade uzla, ktorý bol práve vložený do stromu) postupne až do koreňa, pričom v každom prejdenom uzle aktualizuje výšku a vypočíta koeficient vyváženosti. Ak sa objaví nedostatok pri vyvážení, zistí typ nedostatku a napraví ho vhodnými rotáciami uzlov. Používajú sa štyri druhy rotácií, a to rotácia doprava, doľava a ich permutácie (najprv doľava, potom doprava a najprv doprava, potom doľava). Rotácie doprava a doľava sú realizované samostatnými metódami, zvyšné rotácie volaním týchto dvoch metód v správnom poradí. Viac informácií ohľadom rotácií je poskytnutých na strane 5 tejto dokumentácie.

**Search**

Pri AVL strome je funkcionalita vyhľadávania implementovaná rovnako, ako v prípade základného binárneho vyhľadávacieho stromu. Metóda search dostane ako vstup kľúč uzla, ktorý hľadáme, a odkaz na uzol, od ktorého má začať vyhľadávanie (vo väčšine prípadov ide o koreň stromu). Následne iteratívne prechádza uzly, pričom ak je kľúč hľadaného uzla väčší, ako kľúč súčasného uzla, posunie sa do pravého podstromu, v opačnom prípade do ľavého. Vyhľadávanie skončí buď po nájdení hľadaného uzla alebo dosiahnutí dna stromu.

```
public TreeNode search(int ID, TreeNode node) {
    while(node != null && node.getID() != ID) {
        if(node.getID() > ID) node = node.getLeft();
        else if(node.getID() < ID) node = node.getRight();
    }
    return node;
}
```

*Obr. 2.3 Metóda search triedy AVLTree*

## Delete

Funkcionalita delete slúži na vymazanie daného uzla zo stromu s predpokladom, že sa v ňom nachádza. Po zavolaní metódy (Obr. 2.4) sa najprv pomocou metódy search uskutoční prehľadávanie stromu s cieľom nájsť uzol, ktorý má byť vymazaný. Ak je daný uzol nájdený, musí sa zistiť jeho stav. Ak ide o list (uzol nemá ľavý ani pravý podstrom), jednoducho ho odstránime, pričom toto platí aj v prípade koreňa. V prípade, že uzol má práve jedného potomka (buď ľavý, alebo pravý podstrom), musí sa zabezpečiť správne prepojenie tohto podstromu s rodičovským uzlom daného uzla, ktorý chceme vymazať. Situácia, keď má uzol ľavého aj pravého potomka, je mierne zložitejšia. Najprv sa v ľavom podstrome pomocou metódy searchMax nájde uzol s najväčším kľúčom (vždy to bude list) a jeho obsah sa prekopíruje (vrátane kľúča) do vymazávaného uzla. Následne sa rekurzívne zavolá metóda delete opäť, ale tentokrát s cieľom vymazať uzol, z ktorého sme kopírovali dáta. Po úspešnej operácii odstránenia sa podobne, ako v prípade funkcionality insert, zisťuje výškové vyváženie stromu a v prípade potreby sa aplikujú potrebné rotácie.

```
public TreeNode deleteNode(int ID, TreeNode node) {

    TreeNode delete = this.search(ID, node == null ? this.mainNode : node);

    if(delete == null) return null;

        TreeNode parent = delete.getParent();
        delete.setParent(null);

        if(delete.getLeft() == null && delete.getRight() == null) {         // no children
            if(parent != null) this.bypassNode(parent, delete, null);
            else this.mainNode = null;
        }
        else if(delete.getLeft() == null) {                 // right child, no left child
            if(parent != null) this.bypassNode(parent, delete, delete.getRight());
            else {
                this.mainNode = delete.getRight();
                this.mainNode.setParent(null);
            }
            delete.setRight(null);
        }
        else if(delete.getRight() == null) {                // left child, no right child
            if(parent != null) this.bypassNode(parent, delete, delete.getLeft());
            else {
                this.mainNode = delete.getLeft();
                this.mainNode.setParent(null);
            }
            delete.setLeft(null);
        }
        else {                                                          // 2 children
            TreeNode max = this.searchMax(delete.getLeft());
            delete.copyNode(max);
            this.deleteNode(max.getID(), delete.getLeft());
            delete.setParent(parent);
        }
    return parent;
}
```

*Obr. 2.4 Metóda deleteNode triedy AVLTree*

### Koeficient vyváženosti a rotácie

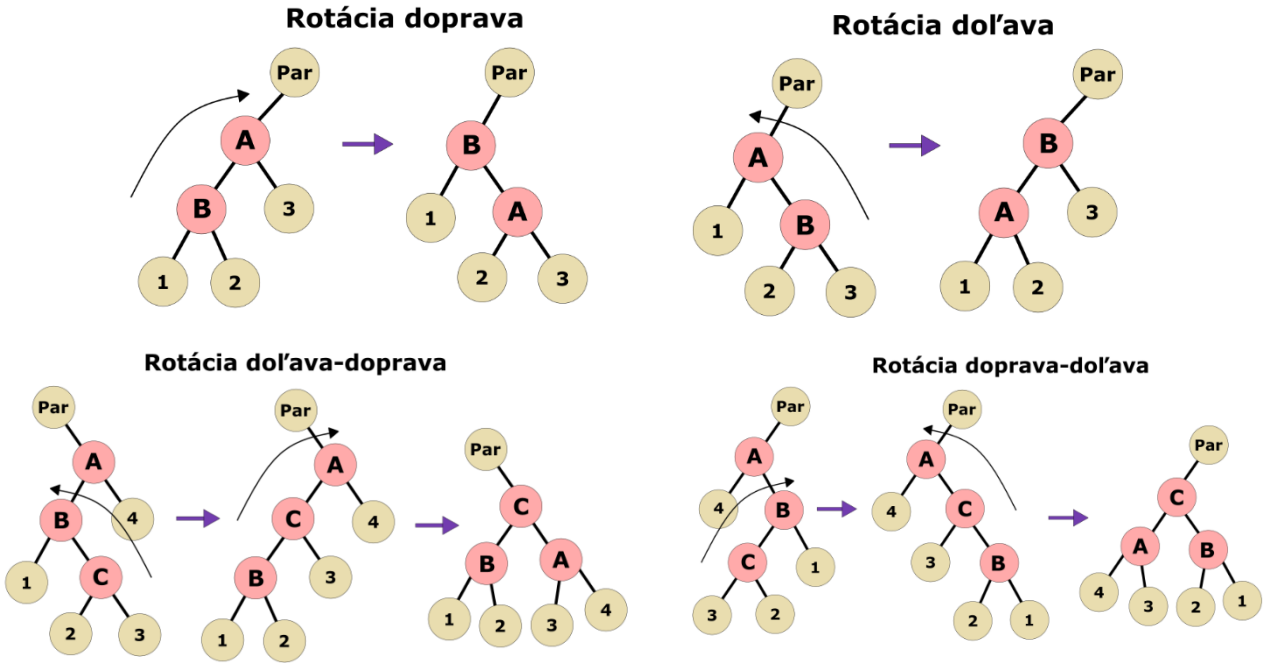
Obr. 2.5 obsahuje implementáciu rotácií doľava a doprava. Základom rotácií je správne prepojenie rotovaných uzlov, ich potomkov a rodičov.

```
public void leftRotate(TreeNode node) {
    TreeNode right = node.getRight();
    TreeNode par = node.getParent();
    TreeNode lChild = right.getLeft();

    if(par != null && par.getRight() == node)
        par.setRight(right);
    else if(par != null && par.getLeft() == node)
        par.setLeft(right);
    else right.setParent(par);
    node.setRight(lChild);
    right.setLeft(node);

    if(this.mainNode == node) this.mainNode = right;

    node.setHeight(this.getMaxHeight(node));
    right.setHeight(this.getMaxHeight(right));
}
```

```
public void rightRotate(TreeNode node) {
    TreeNode left = node.getLeft();
    TreeNode par = node.getParent();
    TreeNode rChild = left.getRight();

    if(par != null && par.getLeft() == node)
        par.setLeft(left);
    else if(par != null && par.getRight() == node)
        par.setRight(left);
    else left.setParent(par);
    node.setLeft(rChild);
    left.setRight(node);

    if(this.mainNode == node) this.mainNode = left;

    node.setHeight(this.getMaxHeight(node));
    left.setHeight(this.getMaxHeight(left));
}
```

*Obr. 2.5 Metódy leftRotate a rightRotate triedy AVLTree*

Ako už bolo spomenuté, v AVL strome sa realizujú štyri rôzne druhy rotácií podľa toho, na akých pozíciách sa nachádzajú okolité uzly pri uzle, ktorý nie je výškovo vyvážený. Tieto rotácie sú znázornené na Obr. 2.6. Komplikovanejšie rotácie sú realizované poskladaním viacerých jednoduchších.




*Obr. 2.6 Štyri druhy rotácií používané pri AVL stromoch*

## 2.2 Splay strom

Splay strom je typ binárneho vyhľadávacieho stromu, kde sa nedávno sprístupnené uzly ukladajú neďaleko koreňa stromu. Pracuje s myšlienkou, podľa ktorej každý súbor dát obsahuje istú skupinu údajov, ku ktorým sa pristupuje častejšie, ako k zvyšným údajom. Prvky tejto skupiny údajov budú uložené vo vyšších vrstvách stromu pomocou procedúry nazvanej „splaying“ (aplikovanie rotácií na daný uzol dovtedy, kým sa z neho nestane koreň stromu) a preto prístup k nim môže mať aj menšiu ako logaritmickú zložitosť. Na druhej strane, splay strom nepoužíva striktný vyvažovací algoritmus a preto sa časová zložitosť operácií vyhľadávania, vkladania a mazania uzlov pohybuje v priemere medzi O(log n) a O(n).

### *2.2.1 Vlastná implementácia*

Implementácia Splay stromu pozostáva, rovnako ako v prípade AVL stromu, z dvoch tried, pričom hlavná trieda SplayTree obsahuje dôležité metódy insert, delete, search a splay a trieda SplayNode predstavuje jeden uzol stromu, oproti AVL stromu sa tu však nenachádzajú žiadne premenné ani metódy zaoberajúce sa výškou stromu alebo koeficientom vyváženosti.

**Insert**

Priebeh a realizácia funkcionality insert (Obr. 2.7) je prakticky identická ako v prípade AVL stromu. Obe riešenia sa skladajú z verejnej a privátnej metódy, obe riešenia postupujú podľa základných pravidiel binárnych vyhľadávacích stromov a vkladajú nový uzol na dno stromu.

```
public void insert(int ID, String name) {
    if(this.mainNode != null) this.splay(this.insertNewNode(ID, name, this.mainNode));
    else this.mainNode = new TreeNode(ID, name);
}

public TreeNode insertNewNode(int ID, String name, TreeNode node) {
    while(node != null) {
        if(ID > node.getID()) {
            if(node.getRight() == null) {
                node.setRight(new TreeNode(ID, name));
                return node.getRight();
            }
            node = node.getRight(); continue;
        }
        else if(ID < node.getID()) {
            if(node.getLeft() == null) {
                node.setLeft(new TreeNode(ID, name));
                return node.getLeft();
            }
            node = node.getLeft(); continue;
        }
        node = null;
    }
    return node;
}
```

*Obr. 2.7 Metóda insert spolu s metódou insertNewNode triedy SplayTree*

Zmena nastane po uskutočnení vloženia uzla. Kým pri AVL strome v tomto bode došlo k overeniu vyváženosti stromu, v prípade Splay stromu sa odkaz na vložený uzol odovzdá metóde splay (Obr. 2.8). Úlohou tejto špeciálnej metódy je posúvať vložený prvok z dna stromu vyššie pomocou rotácií dovtedy, kým sa z neho nestane koreň stromu (pod pojmom rotácie sa rozumejú rovnaké otáčania doprava a doľava, aké obsahuje AVL strom). Používa sa tu pať druhov rotácií, ktoré sú pomenované nasledovne: Cik (jedna rotácia buď doľava, alebo doprava), Cik-Cik (dve po sebe idúce rotácie doprava), Cak-Cak (dve po sebe idúce rotácie doľava), Cik-Cak (jedna rotácia doľava a jedna doprava) a Cak-Cik (jedna rotácia doprava a jedna doľava).

```
public TreeNode splay(TreeNode node) {
    if(node == null) return null;
    while(this.mainNode != node && node.getParent() != null) {
        TreeNode parent = node.getParent();
        TreeNode gParent = (parent == null ? null : parent.getParent());

        if(gParent != null && parent != null) {
            if(gParent.getLeft() == parent) {
                if(parent.getLeft() == node) {
                    this.rightRotate(gParent);
                    this.rightRotate(parent);
                }
                else {
                    this.leftRotate(parent);
                    this.rightRotate(gParent);
                }
            }
            else {
                if(parent.getLeft() == node) {
                    this.rightRotate(parent);
                    this.leftRotate(gParent);
                }
                else {
                    this.leftRotate(gParent);
                    this.leftRotate(parent);
                }
            }
        }
        else if(parent != null) {
            if(parent.getLeft() == node) this.rightRotate(parent);
            else this.leftRotate(parent);
        }
    }
    return node;
}
```

*Obr. 2.8 Metóda splay triedy SplayTree*

## Search

Operácia search v sebe zahŕňa vyhľadanie daného uzla (Obr. 2.9) spolu s použitím metódy splay na posunutie uzla do koreňa stromu. Zaujímavosťou je, že v prípade, ak sa hľadaný uzol nenachádza v strome, do koreňa bude posunutý posledný prehľadaný uzol. Tým je zabezpečená neustála obmena uzlov nachádzajúcich sa blízko koreňa stromu. V kóde je táto skutočnosť ošetrená tak, že pri každej iterácii program overuje, či má práve prehľadávaný uzol potomka na tej strane, kam sa snažíme posunúť. Ak nemá potomka, znamená to, že sa hľadaný uzol nenachádza v strome a dojde k automatickému posunutiu súčasného uzla do koreňa použitím metódy splay.

```
public TreeNode search(int ID, TreeNode node) {
    if(node == null) return node;
    while(node.getID() != ID) {
        if(node.getID() > ID) {
            if(node.getLeft() == null) break;
            else node = node.getLeft();
        }
        else if(node.getID() < ID) {
            if(node.getRight() == null) break;
            else node = node.getRight();
        }
    }
    return this.splay(node);
}
```

*Obr. 2.9 Metóda search triedy SplayTree*

### Delete

Vymazávanie uzlov (Obr. 2.10) v Splay strome funguje unikátnym spôsobom. V prvom kroku sa zavolaním metódy search vyhľadá daný uzol a posunie sa do koreňa. Ak sa daný uzol nenájde, do koreňa sa posunie posledný prehľadaný uzol a metóda sa ukončí. Naopak, ak sa uzol nájde, po jehu presunutí do koreňa dojde k jeho odstráneniu a tým k dočasnému rozdeleniu stromu na dve disjunktné časti.

```
public void delete(int ID, TreeNode node) {

    TreeNode del = this.search(ID, node);

    if(del == null || del.getID() != ID) return;

    TreeNode left = del.getLeft();
    TreeNode right = del.getRight();

    if(left != null) {
        del.setLeft(null);
        left.setParent(null);
    }
    if(right != null) {
        del.setRight(null);
        right.setParent(null);
    }
    if(left == null) {
        this.mainNode = right; return;
    }
    else {
        TreeNode max = this.searchMax(left);
        this.splay(max);
        max.setRight(right);
        this.mainNode = max;
    }
}
```

*Obr. 2.10 Metóda delete triedy SplayTree*

V druhom kroku sa metóda musí rozhodnúť, ktorý potomok zmazaného uzla bude novým koreňom celého stromu. V prípade, ak ľavý potomok neexistuje, pravý potomok sa automaticky stáva novým koreňom stromu. Iná situácia nastane, ak ľavý potomok existuje. V takom prípade je potreba nájsť uzol s najväčším kľúčom v ľavom podstrome a tento uzol pomocou metódy splay presunúť na vrch ľavého podstromu (nakonci procedúry sa z neho stane nový koreň celého stromu). Po presunutí už iba stačí spraviť z pravého potomka originálneho koreňa pravého potomka nového koreňa a strom bude opäť spojený do jedného celku.

# 3 Hašovacia tabuľka

Hašovacia tabuľka je dátová štruktúra využívajúca lineárne pole na ukladanie dvojíc kľúč – údaj, pričom pod pojmom kľúč sa rozumie unikátna postupnosť symbolov, na základe ktorých dokážeme jednoznačne identifikovať a získať hľadaný údaj. Jej základom je hašovacia funkcia slúžiaca na generovanie hašu z daného unikátneho kľúča, pričom tento haš udáva miesto v poli, kam môžeme uložiť novú dvojicu kľúč – údaj alebo kde sa nachádza hľadaný údaj. Hašovacie tabuľky sa rozšírene používajú na tvorbu asociatívnych polí alebo na implementáciu rýchlejších druhov pamätí primárne preto, lebo prístup k údajom sa dá uskutočniť jednoznačne rýchlejšie ako v prípade iných dátových štruktúr (priemerná zložitosť iba O(1)) a rýchlosť vyhľadávania je nezávislá od počtu údajov v tabuľke. Problémy však môžu nastať, keď hašovacia funkcia vygeneruje rovnaký haš pre dva odlišné kľúče. Z tohto dôvodu je dôležité riešenie kolízií.

## 3.1 Hašovacia tabuľka s reťazením

Reťazenie je jeden z hlavných spôsobov riešenia kolízií v hašovacích tabuľkách. Jeho princíp spočíva v tom, že prvky poľa pozostávajú zo spájaných zoznamov. Ak nastane kolízia, nová dvojica kľúč – údaj sa jednoducho uloží na koniec spájaného zoznamu, ktorý sa nachádza na danom mieste v poli. Hlavnou výhodou tohto riešenia je, že sa tabuľka nikdy úplne „nenaplní“ a kvôli tomu zmenu veľkosti tabuľky netreba vykonávať tak často, ako pri iných riešeniach.

### *3.1.1 Vlastná implementácia*

Riešenie pozostáva z dvoch tried, ChainTable a TableElement. Ako to už z názvu vyplýva, prvá menovaná trieda obsahuje implementáciu dôležitých funkcionalít hašovacej tabuľky a druhá menovaná predstavuje jednu dvojicu kľúč – údaj, ktorá bude uložená do tabuľky. Kľúčom v našom prípade bude postupnosť siedmych číslic (dokážeme takto vygenerovať 10 000 000 rôznych kľúčov), údajom meno istého človeka.

**Insert**

Prvým krokom funkcionality insert (Obr. 3.1) je transformovanie kľúča pomocou metódy hash (pre viac informácií viď. stranu 12) na súradnicu, ktorá určuje, kam by mal byť prvok uložený. Následne je potrebné prehľadať pole s cieľom zistiť, či sa už identický záznam v ňom nachádza. Ak áno, metóda skončí. Ak nie, ďalším krokom je zistiť, či je dané miesto v poli prázdne, alebo už obsahuje prvok / prvky. V prípade, ak už obsahuje aspoň jeden prvok, na danom mieste v poli sa vytvorí spájaný zoznam a nový záznam bude uložený na jeho koniec.

```
public void insert(String IDCardNum, String name, TableElement[] table) {

    int pos = this.hash(IDCardNum);
    if(this.search(IDCardNum, table, pos) != null) return;

    if(table[pos] == null) table[pos] = new TableElement(name, IDCardNum);
    else {
        TableElement current = table[pos];
        while(current.getNext() != null) current = current.getNext();
        current.setNext(new TableElement(name, IDCardNum));
    }
    if(table != this.hashTable) return;
    this.entries++;
    if((this.entries / this.tableSize) >= 0.5) this.resizeTable();
}
```

*Obr. 3.1 Metóda insert*

### Search

Funkcionalita search (Obr. 3.2) je realizovaná veľmi jednoducho. Najprv, ako pri každej inej metóde, je potrebné pomocou metódy hash zistiť pozíciu hľadaného prvku. Potom, po presunutí sa na zistenú pozíciu v poli nasleduje porovnávanie prvkov v spájanom zozname s hľadaným prvkom. Ak dojdeme na koniec spájaného zoznamu a prvok sme stále nenašli, nenachádza sa v tabuľke a vyhľadávanie sa môže skončiť.

```
public TableElement search(String IDCardNum, TableElement[] table, int p) {

    int pos = (p == -1 ? this.hash(IDCardNum) : p);
    TableElement current = table[pos];

    while(current != null && !current.getID().equals(IDCardNum)) current = current.getNext();
    return current;
}
```

*Obr. 3.2 Metóda search triedy ChainTable*

### Delete

Pri funkcionalite delete (Obr. 3.3) bolo dôležité zohľadniť skutočnosť, že prvky budú s najväčšou pravdepodobnosťou súčasťou spájaných zoznamov. Z tohto dôvodu po nájdení miesta v poli, kde sa prvok môže nachádzať, je na rade zistiť jeho pozíciu v spájanom zozname. Ak sa nachádza na jeho začiatku, stačí ho z poľa odstrániť a na jeho miesto vložiť nasledujúci prvok v zozname. Ak sa nachádza na konci zoznamu, situácia je ešte jednoduchšia – musíme odstrániť iba referenciu naň z predchádzajúceho prvku. V prípade, ak má predchodcu aj nasledovníka, úlohou metódy je prepojiť tieto dva záznamy a referencia naň sa automaticky odstráni.

```
public void delete(String IDCardNum) {

    int pos = this.hash(IDCardNum);
    TableElement current = this.hashTable[pos], parent = null;

    while(current != null && !current.getID().equals(IDCardNum)) {
        parent = current;
        current = current.getNext();
    }
    if(current == null) return;

    if(this.hashTable[pos] == current) {
        if(current.getNext() == null) this.hashTable[pos] = null;
        else {
            this.hashTable[pos] = current.getNext();
            current.setNext(null);
        }
    }
    else if(parent != null && current.getNext() != null) {
        parent.setNext(current.getNext());
        current.setNext(null);
    }
    else parent.setNext(null);
    this.entries--;
}
```

*Obr. 3.3 Metóda delete triedy ChainTable*

### Metóda hash a zmena veľkosti tabuľky

Metóda implementujúca hašovaciu funkcionalitu je najdôležitejšou súčasťou každej hašovacej tabuľky. Ona totižto určí, na ktoré miesto v poli budú ukladané prvky. Vstupný kľúč musí byť vo formáte reťazca (v našom prípade bude vstupný reťazec pozostávať vždy z číslic). Tento kľúč je vovnútri metódy rozložený na jednotlivé znaky. Nasleduje polynóm výpočtu indexu, v ktorom dané znaky kľúča vystupujú vo forme svojich ASCII hodnôt. Posledným krokom je zistenie zvyšku po delení súčasnou veľkosťou poľa (aby sa predišlo indexovaniu poľa mimo jeho hranice). Identickú hašovaciu metódu (Obr. 3.4) využíva aj tabuľka s priamym adresovaním.

```
public int hash(String key) {

    long sum = 0;

    for(int i = 0; i < key.length(); i++) sum += ((int) key.charAt(i)) * (sum + 1) + 31 * i;
    return Math.abs((int)(23 * sum + 197) % this.tableSize);
}
```

*Obr. 3.4 Metóda hash použitá v oboch tabuľkách*

Každá správna hašovacia tabuľka má zároveň implementovanú aj funkcionalitu zameranú na zmenu veľkosti pre prípad, ak by súčasná veľkosť nepostačovala. Okrem navýšenia kapacity je zmena veľkosti dôležitá aj z pohľadu rýchlosti operácií insert, delete a search. Čím je tabuľka preplnenejšia, tým pomalšie sú spomenuté operácie. Z tohto dôvodu sa odporúča navýšiť kapacitu tabuľky vždy, keď sa zaplní polovica prístupného miesta. Keďže veľkosť tabuľky priamo vystupuje ako premenná vrámci hašovacej metódy, po navýšení kapacity je potrebné všetky už uložené dvojice kľúč – údaj hašovať znova (s veľkou pravdepodobnosťou sa zmení ich index uloženia). Na Obr. 3.5 sa nachádza metóda zmeny veľkosti hašovacej tabuľky použitá v tabuľke s reťazením.

```
public void resizeTable() {
    this.tableSize *= 2;
    TableElement[] newTable = new TableElement[this.tableSize];

    for(int i = 0; i < this.hashTable.length; i++) {
        if(this.hashTable[i] != null) {
            while(this.hashTable[i] != null) {
                String ID = this.hashTable[i].getID();
                String name = this.hashTable[i].getName();
                this.insert(ID, name, newTable);
                this.hashTable[i] = this.hashTable[i].getNext();
            }
        }
    }
    this.hashTable = newTable;
}
```

*Obr. 3.5 Metóda resizeTable triedy ChainTable*

## 3.2 Hašovacia tabuľka s otvoreným adresovaním

Riešenie kolízií otvoreným adresovaním sa vyznačuje tým, že sa pod každým indexom poľa môže nachádzať maximálne jedna dvojica kľúč – údaj. Ak dojde ku kolízii, nová dvojica kľúč – údaj bude vložená do prvého prázdneho miesta v poli napravo od indexu, kde nastala kolízia (tento postup sa nazýva „lineárne pokusy“), prípadne môže byť hľadanie prázdneho miesta a posunutie sa doprava riešené aj inými spôsobmi (kvadratické pokusy, dvojité hašovanie...).

### *3.2.1 Vlastná implementácia*

Implementácia hašovacej tabuľky s otvoreným adresovaním je rozdelená medzi triedy OpenAddTable (základné funkcionality tabuľky) a TableElement (záznam v tabuľke). Kľúčom vyhľadávania, rovnako ako v prípade hašovacej tabuľky s reťazením, je reťazec pozostávajúci zo siedmych číslic.

#### Insert

Vloženie nového záznamu začína vygenerovaním jeho umiestnenia pomocou metódy hash. Dôležité je tiež overiť, či sa už identický záznam v tabuľke nenachádza. Ak je všetko v poriadku, prvok stačí vložiť na vygenerované miesto. V prípade kolízií musíme postupovať inak, ako pri predchádzajúcej tabuľke. V našej implementácii hašovacej tabuľky s otvoreným adresovaním sa na riešenie kolízií používa „quadratic probing“ (kvadratické pokusy). Jeho činnosť spočíva v tom, že v prípade kolízií namiesto posunutia sa na najbližšie miesto doprava sa uskutoční kvadratický skok definovaný premennou quad. Tým je zabezpečené, že údaje v tabuľke budú rovnomernejšie distribuované.

```
public void insert(String IDCardNum, String name, TableElement[] table) {

    int pos = this.hash(IDCardNum), quad = 1;

    if(this.search(IDCardNum, table, pos) != null) return;

    if(table[pos] == null || table[pos].getID().equals("-1")) table[pos] = new TableElement(name, IDCardNum);
    else {
        while(table[pos] != null && !table[pos].getID().equals("-1")) {
            pos += quad * quad;
            if(pos >= this.tableSize) pos = 0;
            quad++;
        }
        table[pos] = new TableElement(name, IDCardNum);
    }
    if(table != this.hashTable) return;
    this.entries++;
    if((this.entries / this.tableSize) >= 0.5) this.resizeTable();
}
```

*Obr. 3.6 Metóda insert triedy OpenAddTable*

#### Search

Funkcionalita search je v podstate identická, ako pri predchádzajúcom type hašovacej tabuľky, rozdiel spočíva iba v tom, že po zistení indexu poľa nie je potrebné prehľadávať spájaný zoznam s cieľom nájsť prvok (keďže na každom mieste môže byť uložený maximálne jeden záznam), ale postup poľom bude realizovaný kvadratickými skokmi.

```
public TableElement search(String IDCardNum, TableElement[] table, int p) {

    int pos = (p == -1 ? this.hash(IDCardNum) : p), quad = 1;

    while(table[pos] != null && !table[pos].getID().equals(IDCardNum)) {
        pos += quad * quad;
        if(pos >= this.tableSize) pos = 0;
        quad++;
    }
    return table[pos];
}
```

*Obr. 3.7 Metóda search triedy OpenAddTable*

### Delete

Funkcionalita delete je riešená zaujímavým spôsobom. Ak by totiž boli prvky kompletne odstránené, bola by prerušená postupnosť prvkov, čo by malo značný dopad na metódu search, ktorá by týmpádom nedokázala uskutočniť vyhľadávanie. Namiesto úplného mazania je preto úlohou metódy delete iba prepísať obsah daného záznamu (nastaví sa mu kľúč „-1“, ktorý indikuje, že sa jedná o miesto s vymazaným záznamom, a vymažú sa uložené údaje). Týmto spôsobom dokáže metóda search bezproblémovo preskočiť zmazané prvky, ale zároveň ich dokáže identifikovať aj metóda insert, ktorá na ich miesto môže kedykoľvek vložiť nový záznam rovnako, akoby sa jednalo o kompletne prázdne miesto v tabuľke.

```
public void delete(String IDCardNum) {

    int pos = this.hash(IDCardNum), quad = 1;

    while(this.hashTable[pos] != null && !this.hashTable[pos].getID().equals(IDCardNum)) {
        pos += quad * quad;
        if(pos >= this.tableSize) pos = 0;
        quad++;
    }
    if(this.hashTable[pos] == null) return;
    this.hashTable[pos] = new TableElement("null", "-1");
    this.entries--;
}
```

*Obr. 3.8 Metóda delete triedy OpenAddTable*

### Zmena veľkosti tabuľky

Zmena veľkosti je taktiež riešená v podobnom duchu ako pri tabuľke s reťazením, ale absencia spájaných zoznamov ju čiastočne zjednoduší. Pri každom existujúcom zázname (okrem tých označených ako zmazané) stačí vygenerovať jeho nové umiestnenie a vložiť do novej tabuľky.

```
public void resizeTable() {
    this.tableSize *= 2;
    TableElement[] newTable = new TableElement[this.tableSize];

    for(int i = 0; i < this.hashTable.length; i++) {
        if(this.hashTable[i] != null && !this.hashTable[i].getID().equals("-1")) {
            String ID = this.hashTable[i].getID();
            String name = this.hashTable[i].getName();
            this.insert(ID, name, newTable);
        }
    }
    this.hashTable = newTable;
}
```

*Obr. 3.9 Metóda resizeTable triedy OpenAddTable*

# 4 Testovanie efektivity

Implementované dátové štruktúry boli podrobené testovaniu pomocou vlastného testovacieho programu, ktorý umožňuje tri druhy testov: sekvenčné testovanie, náhodné testovanie s údajmi zo súboru a manuálne testovanie. Testovanou veličinou bola rýchlosť vykonávania operácií.

## 4.1 Sekvenčné testovanie

Pri sekvenčnom testovaní sa simuluje situácia, aká sa často vyskytuje v online obchodoch. Unikátny kľúč má zohrávať úlohu čísla objednávky, pričom väčšina obchodov tieto čísla negeneruje náhodne, ale sekvenčne za sebou (príklad: 12542, 12543...). Po zvolení tejto možnosti má používateľ úlohu vybrať si počet záznamov, ktoré sa majú vložiť do dátových štruktúr, vyhľadať a následne odstrániť. Horný limit je 10 000 000 záznamov. Po zvolení hodnoty sa najprv do dátovej štruktúry sekvenčne vložia údaje s kľúčmi od 1 do zvolenej hodnoty, následne sa uskutoční sekvenčné vyhľadanie všetkých záznamov (tiež od 1 do zvolenej hodnoty) a nakoniec nasleduje sekvenčné zmazanie záznamov. Po každej vykonanej operácii sa do konzoly vypíše čas trvania v milisekundách.

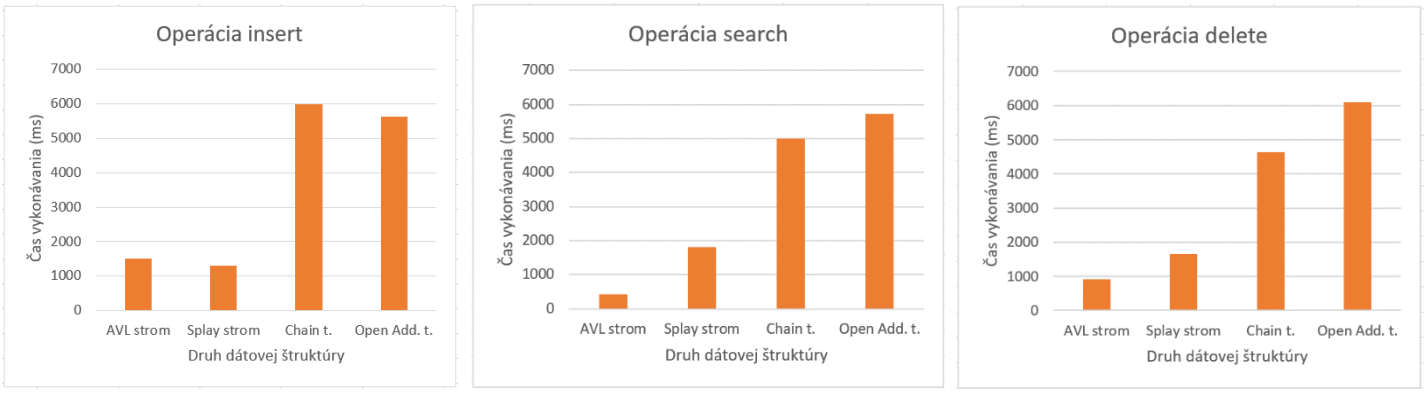
10 000 000 záznamov

| | insert | search | delete |
|---|---|---|---|
| **AVL strom** | *1503ms* | *437ms* | *926ms* |
| **Splay strom** | *1295ms* | *1811ms* | *1655ms* |
| **Chain t.** | *5979ms* | *4991ms* | *4640ms* |
| **Open Add. t.** | *5617ms* | *5720ms* | *6101ms* |

1 000 000 záznamov

| | insert | search | delete |
|---|---|---|---|
| **AVL strom** | *96ms* | *47ms* | *106ms* |
| **Splay strom** | *223ms* | *114ms* | *100ms* |
| **Chain t.** | *433ms* | *453ms* | *356ms* |
| **Open Add. t.** | *458ms* | *533ms* | *455ms* |

*Tab. 4.1 Čas trvania sekvenčného testovania operácií pri 10 000 000 a 1 000 000 záznamoch*




*Obr. 4.1 Grafické porovnanie jednotlivých operácií pri 10 000 000 záznamoch*

Zo zistených údajov zachytených v Tab. 4.1 a na Obr. 4.1 sa dá skonštatovať, že pre danú situáciu je najvyhovujúcejšou dátovou štruktúrou AVL strom. Naopak, hašovacie tabuľky pre túto situáciu nie sú vôbec vhodné. V prípade Splay stromu nastáva zaujímavá situácia, keďže pri 10 000 000 záznamoch je insert rýchlejší, ako pri AVL strome, no v prípade menšieho množstva záznamov sa naopak AVL strom stáva rýchlejším. Za zmienku stojí aj vysoká rýchlosť vyhľadávania AVL stromu.

## 4.2 Testovanie s údajmi zo súboru

Pre účely tohto testovania bol vytvorený súbor s názvom *data.txt*, ktorý obsahuje 1 000 000 dvojíc kľúč – údaj. Na jeho tvorbu bol použitý jednoduchý skript napísaný v programovacom jazyku Python (Obr. 4.2).

```
import random
import string

file = open("data.txt", "w")
size = 1000000

for x in range(0, size):
    id = "";
    name = "";

    for i in range(0, 6):
        id += str(random.randint(0, 9))
    for i in range(0, 10):
        name += str(random.choice(string.ascii_letters))
    file.write(id + " " + name)
    if(x != size - 1):
        file.write("\n")
file.close()
```

*Obr. 4.2 Skript použitý na vytvorenie súboru data.txt*

Tento druh testovania sa líši od sekvenčného tým, že na začiatku testovania sa zo súboru data.txt náhodne načíta používateľom zadefinované množstvo dvojíc kľúč – údaj (maximálne 1 000 000, neodporúča sa však načítať viac ako 100 000, pretože dojde k výraznému spomaleniu programu, ktoré je spôsobené metódou čítania zo súboru). Po úspešnom načítaní budú zvolené záznamy postupne vkladané do jednotlivých dátových štruktúr rovnako, ako pri sekvenčnom testovaní, s cieľom zaznamenať rýchlosť operácií insert, search a delete.

10 000 záznamov zo súboru

| | insert | search | delete |
|---|---|---|---|
| **AVL strom** | 21ms | 4ms | 9ms |
| **Splay Strom** | 10ms | 8ms | 8ms |
| **Chain t.** | 10ms | 2ms | 3ms |
| **Open Add. t.** | 10ms | 2ms | 2ms |

20 000 záznamov zo súboru

| | insert | search | delete |
|---|---|---|---|
| **AVL strom** | 24ms | 9ms | 13ms |
| **Splay Strom** | 20ms | 23ms | 29ms |
| **Chain t.** | 17ms | 3ms | 4ms |
| **Open Add. t.** | 17ms | 4ms | 5ms |

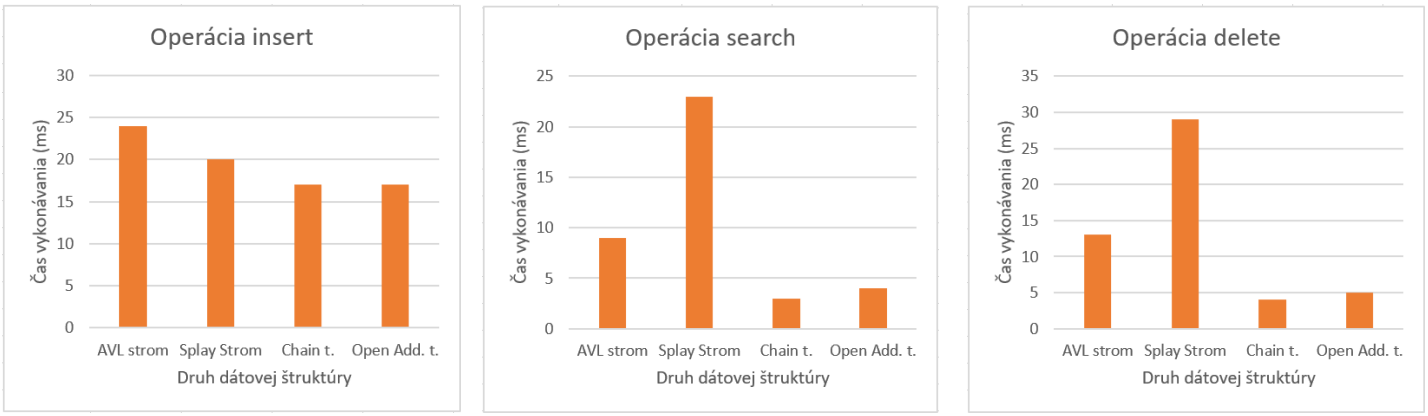
30 000 záznamov zo súboru

| | insert | search | delete |
|---|---|---|---|
| **AVL strom** | 34ms | 12ms | 25ms |
| **Splay Strom** | 31ms | 41ms | 53ms |
| **Chain t.** | 26ms | 10ms | 9ms |
| **Open Add. t.** | 30ms | 9ms | 7ms |

50 000 záznamov zo súboru

| | insert | search | delete |
|---|---|---|---|
| **AVL strom** | 57ms | 23ms | 51ms |
| **Splay Strom** | 70ms | 78ms | 79ms |
| **Chain t.** | 30ms | 10ms | 11ms |
| **Open Add. t.** | 33ms | 10ms | 9ms |

*Tab. 4.2 Čas trvania testovania s rozličným počtom údajov zo súboru*




*Obr. 4.3 Grafické porovnanie jednotlivých operácií pri 20 000 záznamoch*

Namerané údaje v tabuľkách aj grafoch jednoznačne hovoria v prospech hašovacích tabuliek. Keďže boli použité náhodné kľúče, údaje v tabuľke boli rovnomernejšie distribuované, čo podstatne zrýchlilo operácie search a delete. Problémy nenastali ani s operáciou insert. Naopak, najväčšie spomalenie sa dá zaznamenať v prípade Splay stromu. Toto spomalenie sa dá vysvetliť tým, že žiadny z testov neobsahoval podmienky, na ktoré bol Splay strom navrhnutý (časté vyhľadávanie rovnakých informácií). Ak by tieto podmienky boli splnené, Splay strom by bol s veľkou pravdepodobnosťou rýchlejší, ako AVL strom.

## 4.3 Manuálne testovanie

Manuálne testovanie umožňuje vytvorenie hociktorej zo štyroch implementovaných dátových štruktúr a taktiež poskytuje voľnú ruku pri vkladaní, vyhľadávaní a mazaní záznamov. Používateľ po výbere dátovej štruktúry má možnosť vykonať ľubovoľný počet operácií insert, delete a search a v ľubovoľnom poradí. Údaje je potrebné zadávať z konzoly.

```
-------------------------------------
1 - Sequential testing
2 - Random values from file
3 - Manual testing
0 - Exit
Your choice: 3
Choose data structure
1 - AVL tree, 2 - Splay tree, 3 - Chaining table, 4 - Open Add. table: 2
1 - insert, 2 - delete, 3 - search, 0 - end: 1
Enter key (int) and value (string) to be inserted: 50 Jan
1 - insert, 2 - delete, 3 - search, 0 - end: 3
Enter key (int) to be searched: 50
Found value: Jan
1 - insert, 2 - delete, 3 - search, 0 - end: 3
Enter key (int) to be searched: 40
Value not found.
1 - insert, 2 - delete, 3 - search, 0 - end: 2
Enter key (int) to be deleted: 50
1 - insert, 2 - delete, 3 - search, 0 - end: 3
Enter key (int) to be searched: 50
Value not found.
1 - insert, 2 - delete, 3 - search, 0 - end: 0
-------------------------------------
1 - Sequential testing
2 - Random values from file
3 - Manual testing
0 - Exit
Your choice: 0
```

*Obr. 4.4 Ukážka priebehu manuálneho testovania*