Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Zadanie 1

Dátové štruktúry a algoritmy

Adam Jurčišin

Dátum : 27. 3. 2023

Obsah

1. ÚVOD 2
2. DÁTOVÉ ŠTRUKTÚRY 2
3. BINÁRNE VYHĽADÁVACIE STROMY 2
3.1. AVL STROM 3
3.1.1. Pravá rotácia 4
3.1.2. Ľavá rotácia 4
3.1.3. Pravá - ľavá rotácia 5
3.1.4. Ľavá - pravá rotácia 5
3.1.5. *Vkladanie* 5
3.1.6. Vyhľadávanie 6
3.1.7. Vymazanie 6
3.2. SPLAY STROM 7
3.2.1. Zig rotácia 8
3.2.2. Zag rotácia 9
3.2.3. Zig - zag rotácia 9
3.2.4. Zag - zig rotácia 9
3.2.5. Zgg - zig rotácia 9
3.2.6. Zag - zag rotácia 10
3.2.7. Vkladanie 10
3.2.8. Vyhľadávanie 10
3.2.9. Vymazanie 11
4. HEŠOVACIE TABUĽKY 12
4.1. REŤAZENIE 13
4.1.1. Vkladanie 13
4.1.2. Vyhľadávanie 14
4.1.3. Vymazanie 15
4.2. OTVORENÉ ADRESOVANIE 15
4.2.1. Vkladanie 15
4.2.2. Vyhľadávanie 16
4.2.3. Vymazanie 16
5. MERANIE 17
5.1. VKLADANIE 17
5.2. VYHĽADÁVANIE 18
5.3. VYMAZÁVANIE 20
6. ZÁVER 21

Adam Jurčišin
ID: 116203 27.3.2023

1. Úvod

V tomto zadaní máme za úlohu porozumieť čo dátová štruktúra je a následne vytvoriť 2 rozdielne dátové štruktúry v podobe binárnych vyhľadávacích stromov a hešovacích tabuliek. Vytvorené implementácie dátových štruktúr následné podľahnú testovaniu, počas ktorého meraním času počas operácií na vloženie, nájdenie a vymazanie s rôznym počtom údajov preukážu svoju efektivitu.

Zadanie budeme vytvárať v objektovo-orientovanom programovacom jazyku Java, v ktorom okrem jednotlivých tried potrebných na vytvorenie dátových štruktúr vytvoríme taktiež samostatnú triedu pre ich testovanie.

2. Dátové štruktúry

Každá dátová štruktúra pozostáva zo súboru uzlov, ktoré sú v nej uložené. Uzly obsahujú dvojicu dôležitých častí. Kľúča, pomocou ktorého vieme s uzlom v dátovej štruktúre pracovať a následne z dát, ktoré chceme v uzle uchovávať. V našom prípade sme za dáta považovali dátový typ String a za kľúč dátový typ Integer.

Existuje viacero typov dátových štruktúr, v tomto zadaní sme mali za úlohu vytvoriť 2 typy binárneho vyhľadávacieho stromu a 2 typy hešovacej tabuľky.

3. Binárne vyhľadávacie stromy

Binárny vyhľadávací strom je dátová štruktúra stromového typu obsahujúca ľubovoľný počet uzlov.

Binárne vyhľadávacie stromy s nenulovým počtom uzlov musia pozostávať z jedného hlavného uzlu nazývaného koreň, pomocou ktorého sa vieme následne pohybovať v strome ďalej. Každý uzol v strome môžeme taktiež pomenovať názvom rodič v prípade, že obsahuje odkaz na jeho potomkov. Odkaz na potomka sa v rodičovskom uzly musí nachádzať na ľavej strane, pokiaľ je kľúč k prístupu k danému uzlu menší ako kľúč jeho rodiča. Ak je kľúč k prístupu väčší, musí sa nachádzať na strane pravej. Koreň stromu nemá rodičovský uzol.

Medzi dôležité základné pojmy binárnych vyhľadávacích stromov patria taktiež hĺbka a výška. Hĺbka je cesta, počítaná od koreňa ku potomkom, v koreni hĺbka začína na hodnote 0 a následne sa v každom uzle potomka zvyšuje. Výška je cesta, počítaná od najvzdialenejšieho potomka nahor až ku koreňu. Pre zistenie výšky uzla, berieme do úvahy len väčšiu hodnotu z výšok jeho potomkov, ktorú následne zväčšíme o 1. Každý uzol v strome má svoju vypočítateľnú hĺbku a výšku.

Základné operácie, ktoré má binárny vyhľadávací strom spĺňať sú vkladanie, vyhľadávanie a vymazanie uzlov s informáciami na základe ich prístupového kľúča. Pri vkladaní viacerých údajov, ktorých kľúče sú založené na ich postupnosti môžeme docieliť stav, kedy náš strom môže fungovať ako spájaný zoznam, kde uzly majú len jedného svojho

potomka. Takéto stromy nie sú dostatočne efektívne a nazývame ich nevyvážene. Aby sa predišlo týmto neefektívnym binárnym stromom, existujú takzvané samovyvažovacie stromy, ktoré na základe rôznych algoritmov posúvajú svoje uzly aby bol prístup k uzlom čo najefektívnejší.

V tomto zadaní sme si zvolili implementáciu AVL stromu a Splay stromu.

3.1. AVL Strom

AVL strom je typ samovyvažovacieho binárneho vyhľadávacieho stromu, ktorý vykonáva operácie vyvažovania na základne faktoru balansu.

AVL strom sa vyvažuje podľa faktoru balansu, ktorý sa vypočítava v každom uzle stromu. Výpočet faktoru balansu spočíva v rozdiele medzi výškou potomkov uzlu. V prípade, že je strom vyvážený správne, faktor balansu sa pohybuje na hodnotách -1, 0 alebo 1. V inom prípade je strom nevyvážený a podstúpi potrebné operácie rotácií uzlov pre zmenu faktoru balansu.

V našom vypracovanom zadaní obsahuje každý uzol dátový typ Integer s uloženou hodnotou faktoru balansu. Následne sa faktor balansu nastaví po volaní metódy setBalanceFactor (obr.1). Metóda využíva pomocnú metódu returnNumber (obr. 2), ktorá nám vracia výšku potomka zadaného ako parameter metódy, metóda nám napomáha v prípade, že zadaný potomok neexistuje, teda odkaz na neho je nastavený na hodnotu null. V tomto prípade metóda vracia výšku 0.

```
public void setBalanceFactor() {
    balanceFactor = returnNumber(left) - returnNumber(right);
}
```

Obr. 1 Metóda setBalanceFactor

```
private int returnNumber(Node node) {
    if (node == null) return 0;
    else return node.getHeight();
}
```

Obr. 2 Metóda returnNumber

V AVL strome existujú 2 typy rotácií. Pravá rotácia a ľavá rotácia. Tieto rotácie sa taktiež kombinujú a vytvárajú nám pravú-ľavú rotáciu a ľavú-pravú rotáciu, V prípade, že faktor balansu v niektorom uzle nadobúda hodnotu väčšiu ako 1, znamená to, že strom je nevyvážený a, že v danom uzle musí vykonať pravú rotáciu alebo ľavú-právu rotáciu. Naopak, ak je faktor balansu menší ako -1, je potrebné aby strom vykonal v danom uzle ľavú rotáciu alebo pravú-ľavú rotáciu.

3.1.1. Pravá rotácia

Ak strom potrebuje vykonať v niektorom uzle pravú rotáciu, znamená to zámenu daného uzla s jeho ľavým potomkom.

Nami vytvorená metóda pravej rotácie (obr. 3) prijíma ako parameter metódy uzol a následne vracia jeho ľavého potomka, ktorý už má po rotácií ako pravého potomka odkaz na uzol jeho predošlého rodiča.

```
private Node rightRotate(Node node) {
    Node node2 = node.left;
    if (node2 != null) {
        node.left = node2.right;
        node.setHeight(higherHeight(node));
        node2.right = node;
        node2.setHeight(higherHeight(node2));
    }
    return node2;
}
```

Obr. 3 Metóda pravej rotácie

3.1.2. Ľavá rotácia

Ak strom potrebuje vykonať v niektorom uzle ľavú rotáciu, znamená to zámenu daného uzla s jeho pravým potomkom.

Nami vytvorená metóda ľavej rotácie (obr. 4) prijíma ako parameter metódy uzol a následne vracia jeho pravého potomka, ktorý už má po rotácií ako pravého potomka odkaz na uzol jeho predošlého rodiča.

```
private Node leftRotate(Node node) {
    Node node2 = node.right;
    if(node2 != null) {
        node.right = node2.left;
        node.setHeight(higherHeight(node));
        node2.left = node;
        node2.setHeight(higherHeight(node2));
    }
    return node2;
}
```

Obr. 4 Metóda ľavej rotácie

3.1.3. Pravá – ľavá rotácia

Ak strom potrebuje vykonať v niektorom uzle pravú – ľavú rotáciu, znamená to vykonanie pravej rotácie, ktorú nasleduje ľavá rotácia.

3.1.4. Ľavá – pravá rotácia

Ak strom potrebuje vykonať v niektorom uzle ľavú pravú rotáciu, znamená to vykonanie ľavej rotácie, ktorú nasleduje pravá rotácia.

3.1.5. Vkladanie

Vkladanie uzlov patrí medzí základné operácie binárnych vyhľadávacích stromov. V AVL strome, sa vložený uzol stane koreňom stromu pokiaľ je strom prázdny, v inom prípade sa použijú základné pravidlá binárnych vyhľadávacích stromov, kde porovnávame kľúč vkladaného uzla postupne s ostatnými kľúčmi uzlov v ktorých sa posúvame. Začíname od koreňa, ak je kľúč vkladaného uzlu väčší posunieme sa doprava, inak sa posunieme doľava a opäť porovnávame kľúče. Tento proces AVL strom vykonáva automaticky pri vkladaní každého uzlu až kým nedorazí na koniec stromu, kde potomok skúmaného uzla už neexistuje, v tom momente sa vkladaný uzol stane jeho potomkom.

Po vložení uzla sa nastaví jeho výška a následne sa spustí proces opätovného vrátenia rovnakou cestou naspäť až ku koreňu, počas tohto procesu strom aktualizuje v každom prejdenom uzly svoju výšku a faktor balansu. V prípade nevhodného faktoru balansu sa vykonajú potrebné rotácie. Tento proces opätovného vrátenia s aktualizáciou faktoru balansu a vykonaní potrebných rotácií sa nazýva samovyvažovací proces AVL stromu (obr. 5).

```
private Node insertIntoTree(Node node, Node dataNode) {
    if (node == null) {
        node = new Node(dataNode.getKey(), dataNode.getData());
        node.setHeight(1);
        return node;
    }
    if (node.getKey() > dataNode.getKey()) {
        node.left = insertIntoTree(node.left, dataNode);
    }
    else if (node.getKey() < dataNode.getKey()) {
        node.right = insertIntoTree(node.right, dataNode);
    }

    node.setHeight(higherHeight(node));
    node.setBalanceFactor();
    node = updateBalanceFactor(node);
    return node;
}
```

Obr. 5 Metóda vkladania do AVL stromu

3.1.6. Vyhľadávanie

Vyhľadanie uzlov patrí medzi základné operácie binárnych vyhľadávacích stromov. V neprázdnom AVL strome, vyhľadávanie začína pri koreni, kde porovnáme kľúč koreňa s kľúčom hľadaného uzlu, ak sa kľúče rovnajú, našli sme hľadaný uzol, inak sa použijú základné pravidlá binárnych vyhľadávacích stromov, kde porovnáme kľúč vkladaného uzlu postupne s ostatnými kľúčmi uzlov v ktorých sa posúvame. Ak strom počas vykonávania tohto procesu narazí na kľúče sebe rovné, našiel hľadaný uzol, inak sa hľadaný uzol v strome nenachádza a návratová hodnota metódy na vyhľadávanie bude rovná null (obr. 6).

```
public Node search(Node node, Node dataNode) {
    if (node == null) return null;
    else if (node.getKey() == dataNode.getKey()) return node;
    else if (node.getKey() > dataNode.getKey()) return search(node.left, dataNode);
    else if (node.getKey() < dataNode.getKey()) return search(node.right, dataNode);
    return null;
}
```

Obr. 6 Metóda vyhľadávania v AVL strome

3.1.7. Vymazanie

Vymazanie uzlov patrí medzi základné operácie binárnych vyhľadávacích stromov. V neprázdnom AVL strome, pomocou využívania rovnakého princípu ako pri vyhľadávaní nájdeme uzol, ktorý chceme vymazať. Pri mazaní uzla existujú 3 možné prípady, ktoré môžu nastať.

Prvý prípad je, ak uzol nedisponuje žiadnymi potomkami, v tomto prípade uzol nastavíme na hodnotu null, čím zabezpečíme, že rodič mazaného uzlu už naďalej nebude disponovať odkazom na tento uzol.

Druhý prípad je, ak uzol disponuje len jedným potomkom, v tomto prípade zmeníme v rodičovskom uzli mazaného uzla odkaz na mazaný uzol za odkaz na potomka mazaného uzla, rodičovský uzol naďalej nebude disponovať odkazom na mazaný uzol.

Tretí prípad je, ak uzol disponuje dvomi potomkami, v tomto prípade nájdeme pomocou metódy findInorderSuccessor() uzol s najmenším väčším kľúčom (obr. 7). Následne zameníme hodnoty z nájdeného uzla do uzla, ktorý chceme vymazať, po tomto kroku sme sa zbavili hodnoty v uzli, ktorý sme chceli vymazať ale v strome máme duplikát, uzol duplikátu nájdeme a podobne ako v prvom prípade, odkaz na duplicitný uzol odstránime.

```
private Node findInorderSuccessor(Node node) {
    node = node.right;
    while(node.left != null) node = node.left;
    return node;
}
```

Obr. 7 Metóda findInorderSuccessor

Po vymazaní uzla v AVL strome sa rovnako ako pri vkladaní rovnakou cestou vraciame ku koreňu stromu, aktualizujeme výšku uzlov a ich faktor balansu a strom využíva samovyvažovacie procesy, čím zabezpečuje, že je strom neustále vyvážený (obr. 8).

```
private Node deleteFromTree(Node node, Node dataNode) {
    if (node == null) return null;
    else if (node.getKey() > dataNode.getKey()) node.left = deleteFromTree(node.left, dataNode);
    else if (node.getKey() < dataNode.getKey()) node.right = deleteFromTree(node.right, dataNode);
    else {
        //CASE 1
        if (node.left == null && node.right == null) return null;
        //CASE 2
        else if (node.left == null) return node.right;
        else if (node.right == null) return node.left;
        //CASE 3
        else {
            Node inOrderNode = findInorderSuccessor(node);
            node.setKey(inOrderNode.getKey());
            node.setData(inOrderNode.getData());
            node.right = deleteFromTree(node.right, node);
            if (node.right != null) {
                node.right.setHeight(higherHeight(node.right));
                node.right.setBalanceFactor();
                node.right = updateBalanceFactor(node.right);
            }
        }
    }
    node.setHeight(higherHeight(node));
    node.setBalanceFactor();
    node = updateBalanceFactor(node);
    return node;
}
```

Obr. 8 Metóda vymazania z AVL stromu

3.2. Splay Strom

Splay strom je typ samovyvažovacieho binárneho vyhľadávacieho stromu, ktorý vykonáva operácie vyvažovania na základne metódy splay.

Na rozdiel od AVL stromu, obsahuje každý uzol v Splay strome navyše odkaz na jeho rodiča, uchovávanie tohto odkazu je dôležité pre vyvažovanie stromu.

Splay metóda vykonáva 6 druhov rotácií, zig, zag, zig – zag, zag – zig, zig – zig a zag - zag rotáciu pomocou ktorých zabezpečuje, že po zavolaní metódy, uzol, ktorý obdrží ako parameter metódy vyváži a opustí metódu ako nový koreň Splay stromu (obr. 9).

```
nodeToSplay = null;
while (node.getParent() != null) {
    if (node.getParent() == root) {
        if (node == node.getParent().left) {
            return root = rightRotate(node.getParent());
        }
        else if(node == node.getParent().right){
            return root = leftRotate(node.getParent());
        }
    }
    else {
        Node parent = node.getParent();
        Node grandParent = parent.getParent();
        if (node == node.getParent().left && parent == parent.getParent().left) {
            grandParent = rightRotate(grandParent);
            parent = rightRotate(parent);
        }
        else if (node == node.getParent().right && parent == parent.getParent().right) {
            grandParent = leftRotate(grandParent);
            parent = leftRotate(parent);
        }
        else if (node == node.getParent().left && parent == parent.getParent().right) {
            parent = rightRotate(parent);
            grandParent = leftRotate(grandParent);
        }
        else {
            parent = leftRotate(parent);
            grandParent = rightRotate(grandParent);
        }
    }
}
return node;
```

Obr. 9 Metóda Splay

3.2.1. Zig rotácia

Ak strom potrebuje vykonať v niektorom uzle zig rotáciu, znamená to vykonanie pravej rotácie.

Nami vytvorená metóda pravej rotácie (obr. 10) prijíma ako parameter metódy uzol a následne vracia jeho ľavého potomka, ktorý už má po rotácií ako pravého potomka odkaz na uzol jeho predošlého rodiča. Uzol má zmenený odkaz na jeho rodiča podľa odkazu jeho predošlého rodiča, nový pravý potomok má ako odkaz na jeho rodiča odkaz na uzol.

```
private Node rightRotate(Node node) {
    Node node2 = node.left;
    node.left = node2.right;
    if (node.left != null) node.left.setParent(node);
    if(node.getParent() != null && node == node.getParent().left) node.getParent().left = node2;
    else if(node.getParent() != null && node == node.getParent().right) node.getParent().right = node2;
    node2.right = node;
    node2.setParent(node.getParent());
    node.setParent(node2);
    return node2;
}
```

Obr. 10 Metóda pravej rotácie

3.2.2. Zag rotácia

Ak strom potrebuje vykonať v niektorom uzle zag rotáciu, znamená to vykonanie ľavej rotácie.

Nami vytvorená metóda ľavej rotácie (obr. 11) prijíma ako parameter metódy uzol a následne vracia jeho pravého potomka, ktorý už má po rotácií ako ľavého potomka odkaz na uzol jeho predošlého rodiča. Uzol má zmenený odkaz na jeho rodiča podľa odkazu jeho predošlého rodiča, nový pravý potomok má ako odkaz na jeho rodiča odkaz na uzol.

```
private Node leftRotate(Node node) {
    Node node2 = node.right;
    node.right = node2.left;
    if (node.right != null) node.right.setParent(node);
    if(node.getParent() != null && node == node.getParent().left) node.getParent().left = node2;
    else if(node.getParent() != null && node == node.getParent().right) node.getParent().right = node2;
    node2.left = node;
    node2.setParent(node.getParent());
    node.setParent(node2);
    return node2;
}
```

Obr. 11 Metóda ľavej rotácie

3.2.3. Zig – zag rotácia

Ak strom potrebuje vykonať v niektorom uzle zig – zag rotáciu, znamená to vykonanie pravej rotácie, ktorú nasleduje ľavá rotácia.

3.2.4. Zag – zig rotácia

Ak strom potrebuje vykonať v niektorom uzle zag – zig rotáciu, znamená to vykonanie ľavej rotácie, ktorú nasleduje pravá rotácia.

3.2.5. Zig – zig rotácia

Ak strom potrebuje vykonať v niektorom uzle zig – zig rotáciu, znamená to vykonanie pravej rotácie, ktorú nasleduje ďalšia pravá rotácia.

3.2.6. Zag – zag rotácia

Ak strom potrebuje vykonať v niektorom uzle zag – zag rotáciu, znamená to vykonanie ľavej rotácie, ktorú nasleduje ďalšia ľavá rotácia.

3.2.7. Vkladanie

Vkladanie uzlov patrí medzí základné operácie binárnych vyhľadávacích stromov. V Splay strome, sa vložený uzol stane koreňom stromu pokiaľ je strom prázdny, v inom prípade sa použijú základné pravidlá binárnych vyhľadávacích stromov, kde porovnávame kľúč vkladaného uzla postupne s ostatnými kľúčmi uzlov v ktorých sa posúvame. Začíname od koreňa, ak je kľúč vkladaného uzlu väčší posunieme sa doprava, inak sa posunieme doľava a opäť porovnávame kľúče. Tento proces Splay strom vykonáva automaticky pri vkladaní každého uzlu až kým nedorazí na koniec stromu, kde potomok skúmaného uzla už neexistuje, v tom momente sa vkladaný uzol stane jeho potomkom. Po vložení nového uzlu sa v uzli nastaví odkaz na jeho rodiča (obr. 12).

```
private Node insertIntoTree(Node node, Node dataNode) {
    if (node == null) {
        node = new Node(dataNode.getKey());
        nodeToSplay = node;
        return node;
    }
    if (node.getKey() > dataNode.getKey()) {
        node.left = insertIntoTree(node.left, dataNode);
        node.left.setParent(node);
    }
    else if (node.getKey() < dataNode.getKey()) {
        node.right = insertIntoTree(node.right, dataNode);
        node.right.setParent(node);
    }
    return node;
}
```

Obr. 12 Metóda vkladania do Splay stromu

Po vložení uzla do Splay stromu voláme metódu splay, ktorej cez parameter odosielame vložený uzol. Metóda vykoná všetky potrebné rotácie a vráti nám naspäť nový koreň stromu.

3.2.8. Vyhľadávanie

Vyhľadanie uzlov patrí medzi základné operácie binárnych vyhľadávacích stromov. V neprázdnom Splay strome, vyhľadávanie začína pri koreni, kde porovnáme kľúč koreňa s kľúčom hľadaného uzlu, ak sa kľúče rovnajú, našli sme hľadaný uzol, inak sa použijú základné pravidlá binárnych vyhľadávacích stromov, kde porovnáme kľúč vkladaného uzlu postupne s ostatnými kľúčmi uzlov v ktorých sa posúvame. Ak strom počas vykonávania tohto

procesu narazí na kľúče sebe rovné, našiel hľadaný uzol a volá sa metóda splay, ktorá vykoná potrebné rotácie okolo nájdeného uzlu, inak sa hľadaný uzol v strome nenachádza a návratová hodnota metódy na vyhľadávanie bude rovná null (obr. 13).

```
public Node search(Node node, Node dataNode) {
    while (node != null) {
            if(node.getKey() == dataNode.getKey()) {
                return root = splay(node);
            }
            if(node.getKey() > dataNode.getKey()) {
                node = node.left;

            }
            else if(node.getKey() < dataNode.getKey()){
                node = node.right;
            }
        }
    return null;
}
```

Obr. 13 Metóda vyhľadania v Splay strome

3.2.9. Vymazanie

Vymazanie uzlov patrí medzi základné operácie binárnych vyhľadávacích stromov. V neprázdnom Splay strome, pomocou metódy na vyhľadávanie nájdeme uzol, ktorý chceme vymazať. Ak nájdený uzol nemá ľavého potomka, zameníme uzol za pravého potomka uzla. Pokiaľ nájdený uzol má ľavého potomka, vyhľadáme pomocou metódy findPreorderSuccessor (obr . 14) najväčší menší uzol a vykonáme s daným uzlom splay metódu (obr. 15).

```
private Node findPreorderSuccessor(Node node) {
    node = node.left;
    while (node != null && node.right != null) node = node.right;
    return node;
}
```

Obr. 14 Metóda findPreorderSuccessor

```
public void delete(Node dataNode) {
    root = search(root, dataNode);
    if (root != null) {
        if (root.left == null) {
            root = root.right;
            if(root != null) root.setParent(null);
        }
        else {
            Node nodeRight = root.right;
            Node node = findPreorderSuccessor(root);
            if (node != null) root = splay(node);
            root.right = nodeRight;
            if (nodeRight != null) nodeRight.setParent(root);
        }
    }
}
```

Obr. 15 Metóda vymazania v Splay strome

4. Hešovacie tabuľky

Hešovacia tabuľka je dátová štruktúra obsahujúca ľubovoľný počet uzlov. Umožňuje vykonávanie základných operácií vkladania, vyhľadávania a vymazávania uzlov na základe kľúča. Hešovacia tabuľka je zoznam s určitou veľkosťou.

Hešovacie tabuľky využívajú proces hešovania, ktorý vypočíta index, na ktorý v tabuľke uzol s dátami uložíme. Táto hešovacia metóda sa využíva pri každej našej operácií v hešovacej tabuľke, to znamená, že v skutočnosti poznáme len kľúč ale index na ktorom je uzol s dátami uložený nie. V našej implementácií využívame jednoduchý princíp hešovacej metódy (obr. 16).

```
private int hash(int key) {
    return key % sizeOfTable;
}
```

Obr. 16 Metóda hash

Prázdna hešovacia tabuľka je veľkosti 1, následne využíva faktor naplnenia, ktorý po vložení alebo vymazaní uzla kontroluje jej naplnenie, v prípade, že faktor naplnenia presiahne 70% alebo je menší ako 30% vykoná tabuľka proces zmeny veľkosti volaním metódy resizeHashTable (obr. 17). Týmto je tabuľka zabezpečená, aby vykonávané operácie fungovali efektívne.

Rôzne druhy hešovacej tabuľky sa líšia podľa ich procesu zaobchádzania s kolíziami, dvomi alebo viacerými uzlami, ktorým hešovacia metóda pridelí rovnaký index na uloženie. V našej implementácií sme vytvorili typ reťazenia a typ otvoreného adresovania.

```
private void resizeHashTable(int sizeOfTable) {
    ArrayList<ChainingNode> oldTable = table;
    this.sizeOfTable = sizeOfTable;
    numberOfBucketsUsed = 0;
    table = new ArrayList<>(sizeOfTable);
    for(int i = 0; i < sizeOfTable; i++) {
        table.add(null);
    }
    for(ChainingNode bucket : oldTable) {
        while (bucket != null) {
            ChainingNode node = bucket.getNext();
            bucket.setNext(null);
            insert(bucket);
            bucket = node;
        }
    }
}
```

Obr. 17 Metóda resizeHashTable

4.1. Reťazenie

Reťazenie je princíp zaobchádzania s kolíziami v hešovacej tabuľke. Pokiaľ sa na jednu pozíciu má uložiť viacero uzlov, vznikne na danej pozícií štruktúra podobná spájanému zoznamu. V princípe reťazenia obsahuje vkladaný uzol odkaz na ďalší uzol s rovnakým indexom uloženia.

4.1.1. Vkladanie

Vkladanie uzlov patrí medzi základné operácie v hešovacích tabuľkách. V hešovacej tabuľke s reťazením sa najprv pomocou hešovacej metódy vypočíta index na uloženie uzla. Ak má tabuľka miesto s daným indexom prázdne, uzol sa na pozíciu uloží. V prípade, že dochádza ku kolízií a na danom mieste sa už nachádza iný uzol, skontrolujeme jeho odkaz na ďalší uzol, ak nadobúda hodnotu null, odkaz presmerujeme na náš vkladaný uzol, v inom prípade sa posúvame ďalej po odkazoch na ďalší uzol až pokiaľ nenájdeme voľné miesto. Po vložení uzla aktualizujeme faktor naplnenia a v prípade potreby vykonáme zmenu veľkosti (obr. 18).

```java
public void insert(ChainingNode node) {
    int position = hash(node.getKey());
    if(table.get(position) == null) numberOfBucketsUsed++;
    table.set(position, insertIntoBucket(table.get(position), node));

    if (checkLoadFactor() >= 0.8) resizeHashTable( sizeOfTable: sizeOfTable * 2);
}

2 usages
private ChainingNode insertIntoBucket(ChainingNode node, ChainingNode dataNode) {
    if (node == null) {
        node = new ChainingNode(dataNode);
        return node;
    }
    else {
        ChainingNode head = node;
        while (node.getNext() != null) {
            node = node.getNext();
        }
        node.setNext(insertIntoBucket(node.getNext(), dataNode));
        return head;
    }
}
```

Obr. 18 Metódy vkladania v reťazení

4.1.2. Vyhľadávanie

Vyhľadávanie uzlov patrí medzi základné operácie v hešovacích tabuľkách. V hešovacej tabuľke s reťazením sa najprv pomocou hešovacej metódy vypočíta index, kde je hľadaný uzol uložený. Následne na danej pozícii porovnávame hľadaný kľúč s kľúčom uzlov, pokiaľ sa rovnajú, uzol sme našli, inak sa posúvame ďalej po odkazoch na ďalší uzol až pokiaľ nenájdeme zhodu v kľúčoch (obr. 19).

```java
public ChainingNode search(ChainingNode searchingNode) {
    int position = hash(searchingNode.getKey());
    ChainingNode node = table.get(position);
    while (node.getKey() != searchingNode.getKey()) {
        node = node.getNext();
    }
    return node;
}
```

Obr. 19 Metóda vyhľadania v reťazení

4.1.3. Vymazávanie

Vymazávanie uzlov patrí medzi základné operácie v hešovacích tabuľkách. V hešovacej tabuľke s reťazením sa najprv pomocou hešovacej metódy vypočíta index na ktorom sa uzol, ktorý chceme vymazať nachádza. Následne porovnávame kľúč uzla, ktorý chceme vymazať s uzlom v ktorom sa nachádzame, pokiaľ sa nerovnajú posúvame sa pomocou odkazu na ďalší uzol ďalej. Ak uzol nájdeme, ak neobsahuje odkaz na ďalší uzol, tak zmeníme odkaz na tento uzol na null, ak uzol obsahuje odkaz na ďalší uzol, zmeníme odkaz na tento uzol za odkaz na uzol na ktorý mazaný uzol odkazuje. Ak je mazaný uzol na prvom mieste na svojej pozícií v tabuľke, uložíme na danú pozíciu uzol na ktorý mazaný uzol odkazuje. Po vymazaní uzla aktualizujeme faktor naplnenia a v prípade potreby vykonáme zmenu veľkosti (obr. 20).

```
public void delete(ChainingNode deletingNode) {
    int position = hash(deletingNode.getKey());
    ChainingNode node = table.get(position);
    ChainingNode head = node;
        if(node.getNext() == null) table.set(position, null);
        else {
            if(node.getKey() == deletingNode.getKey()) table.set(position, node.getNext());
            else {
                while (node.getNext() != null) {
                    if (node.getNext().getKey() == deletingNode.getKey()) node.nodeNext(node.getNext().getNext());
                    else node = node.getNext();
                }
                table.set(position,head);
            }
        }
        if (table.get(position) == null) numberOfBucketsUsed--;
        loadFactor = checkLoadFactor();
        if (loadFactor <= 0.3) resizeHashTable( sizeOfTable: sizeOfTable / 2);
}
```

Obr. 20 Metóda vymazania v reťazení

4.2. Otvorené adresovanie

Otvorené adresovanie je princíp zaobchádzania s kolíziami v hešovacej tabuľke.

4.2.1. Vkladanie

Vkladanie uzlov patrí medzi základné operácie v hešovacích tabuľkách. V hešovacej tabuľke s otvoreným adresovaním sa najprv pomocou hešovacej metódy vypočíta index na uloženie uzla. Ak je v tabuľke na danej pozícií voľné miesto, uzol sa vloží, inak dochádza ku kolízií, následne sa zvyšuje index na uloženie uzla až kým nenájdeme v tabuľke pozíciu s voľným miestom. V prípade, že narazíme na koniec tabuľky, začíname od prvej pozície v tabuľke. Po vložení uzla aktualizujeme faktor naplnenia a v prípade potreby vykonáme zmenu veľkosti (obr. 21).

```
public void insert(OaNode nodeToInsert) {
    int position = hash(nodeToInsert.getKey());
    while (table.get(position) != null) {
        position = (position + 1) % sizeOfTable;
    }
    numberOfBucketsUsed++;
    table.set(position, nodeToInsert);
    if (checkLoadFactor() >= 0.7) resizeHashTable( sizeOfTable: sizeOfTable * 10);
}
```

Obr. 21 Metóda na vkladanie v otvorenom adresovaní

4.2.2. Vyhľadávanie

Vyhľadávanie uzlov patrí medzi základné operácie v hešovacích tabuľkách. V hešovacej tabuľke s otvoreným adresovaním sa najprv pomocou hešovacej metódy vypočíta index, kde je predpokladané, že je uzol uložený. Následne na danej pozícií porovnávame hľadaný kľúč s kľúčom uzlu, pokiaľ sa rovnajú, uzol sme našli, inak sa index zvyšuje a kľúče sa porovnávajú až pokiaľ nenarazíme na zhodu v kľúčoch (obr. 22).

```
public OaNode search(OaNode nodeToSearch) {
    int position = hash(nodeToSearch.getKey());
    while (table.get(position).getKey() != nodeToSearch.getKey()) {
        position = (position + 1) % sizeOfTable;
        while (table.get(position) == null) position = (position + 1) % sizeOfTable;
    }
    return table.get(position);
}
```

Obr. 22 Metóda vyhľadania v reťazení

4.2.3. Vymazávanie

Vymazávanie uzlov patrí medzi základné operácie v hešovacích tabuľkách. V hešovacej tabuľke s otvoreným adresovaním sa najprv pomocou hešovacej metódy vypočíta index na ktorom sa uzol, ktorý chceme vymazať nachádza. Následne porovnávame kľúč uzla, ktorý chceme vymazať s uzlom na pozícií na ktorej sa nachádzame, pokiaľ sa nerovnajú posúvame sa na ďalšiu pozíciu v tabuľke a porovnávame kľúče až pokiaľ nenarazíme na zhodu. Ak uzol nájdeme, nastavíme, že tabuľka na tejto pozícií obsahuje voľnú pozíciu. V prípade, že narazíme na koniec tabuľky, začíname od prvej pozície v tabuľke. Po vymazaní uzla aktualizujeme faktor naplnenia a v prípade potreby vykonáme zmenu veľkosti (obr. 23).

```
public void delete(OaNode nodeToDelete) {
    int position = hash(nodeToDelete.getKey());
    while (table.get(position) == null) position = (position + 1) % sizeOfTable;
    while (table.get(position).getKey() != nodeToDelete.getKey()) {
        position = (position + 1) % sizeOfTable;
        while (table.get(position) == null) position = (position + 1) % sizeOfTable;
    }
    table.set(position, null);
    if (table.get(position) == null) numberOfBucketsUsed--;
    loadFactor = checkLoadFactor();
    if (loadFactor <= 0.25) resizeHashTable( sizeOfTable: sizeOfTable / 2);
}
```

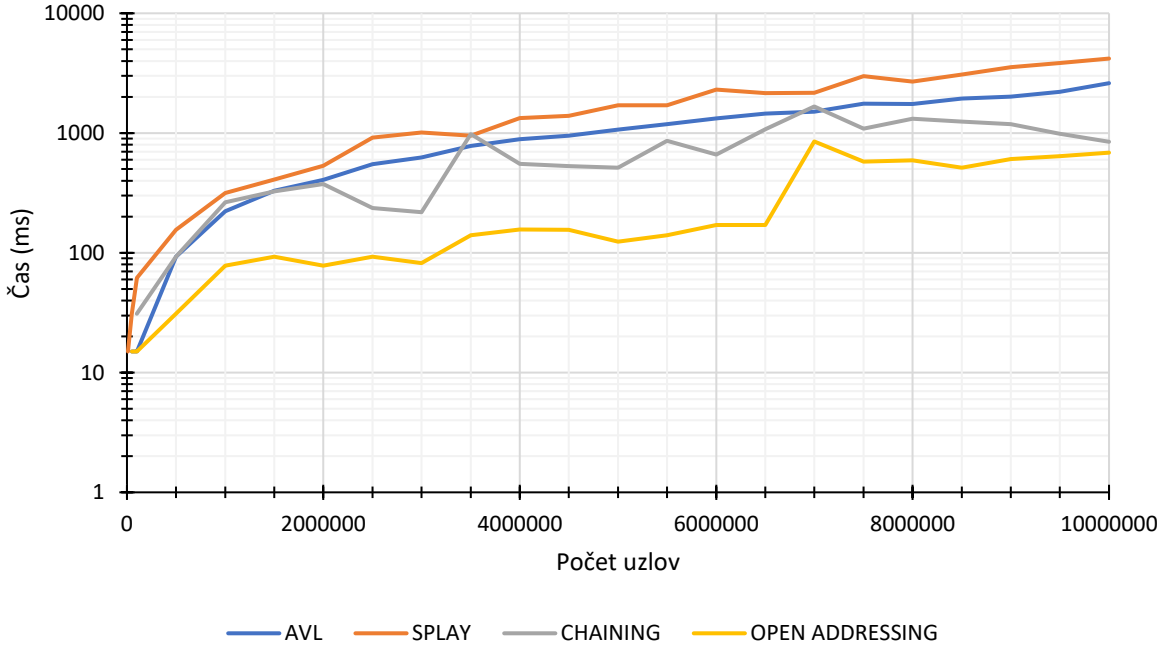
Obr. 23 Metóda vymazania v otvorenom adresovaní

5. Meranie

Podstúpením merania preukážeme časovú efektivitu nami implementovaných dátových štruktúr. Merali čas pre implementované operácie ako vkladanie, vyhľadávanie a vymazávanie. Výsledkom merania je čas v mikrosekundách, počas ktorého dátové štruktúry vložia, vyhľadajú alebo vymažú určitý počet uzlov. Každá operácia sa pri rovnakom počte uzlov vykonávala dvakrát, raz pri práci s uzlami, ktorých kľúče nasledovali zaradom a druhýkrát pri práci s uzlami, ktorých kľúče boli náhodné. Pre meranie každej operácie sme vykonali 25 meraní s rozdielnym počtom uzlov.

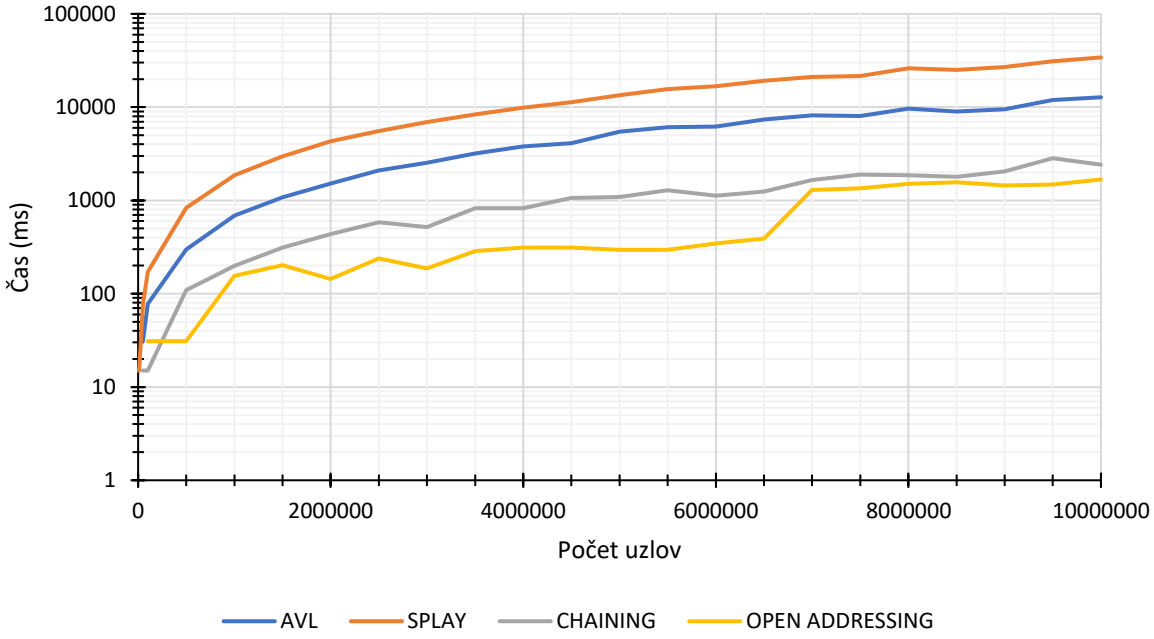
5.1. Vkladanie

Prvé meranie operácie vkladania pozostávalo z vkladania uzlov s kľúčmi zaradom. Výsledkom bolo zistenie, že nami implementovaná dátová štruktúra hešovacej tabuľky s otvoreným adresovaním je v tomto prípade najefektívnejšia. Najmenej efektívna je dátová štruktúra binárneho vyhľadávacieho stromu splay (graf 1).



Graf 1: Výsledok vkladania zaradom

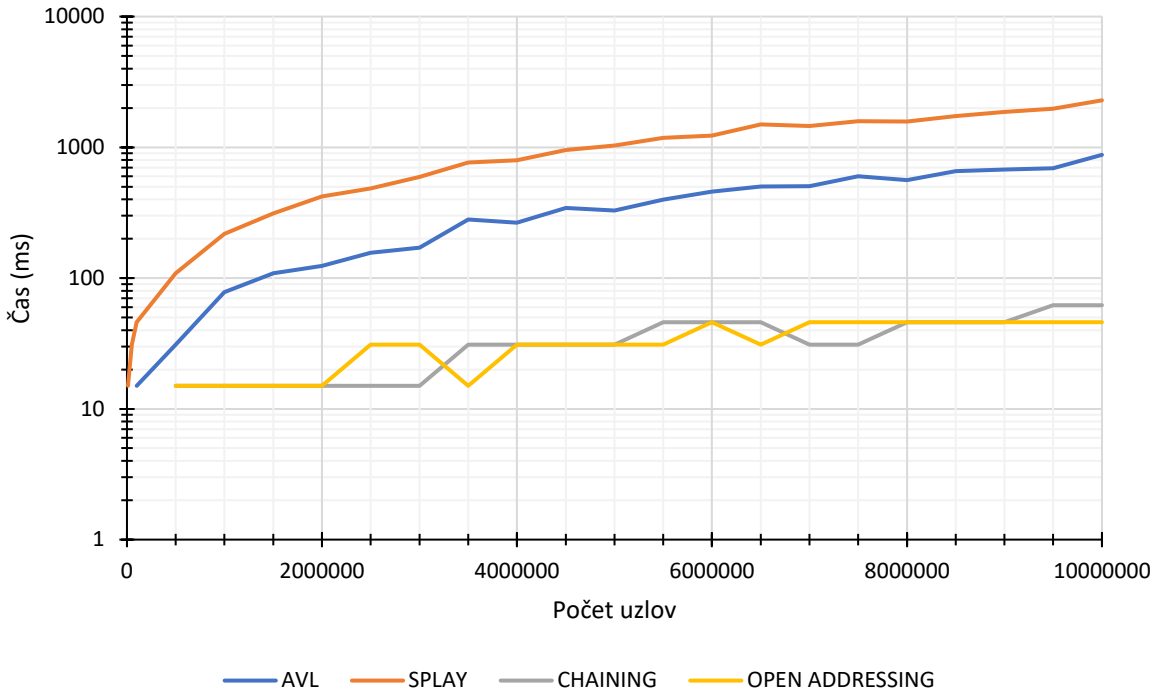
Druhé meranie operácie vkladania pozostávalo z vkladania uzlov s náhodnou postupnosťou kľúčov. Výsledkom bolo zistenie, že nami implementovaná dátová štruktúra hešovacej tabuľky s otvoreným adresovaním je v tomto prípade najefektívnejšia. Najmenej efektívna je dátová štruktúra binárneho vyhľadávacieho stromu splay (graf 2).



Graf 2: Výsledok vkladania s náhodnou postupnosťou

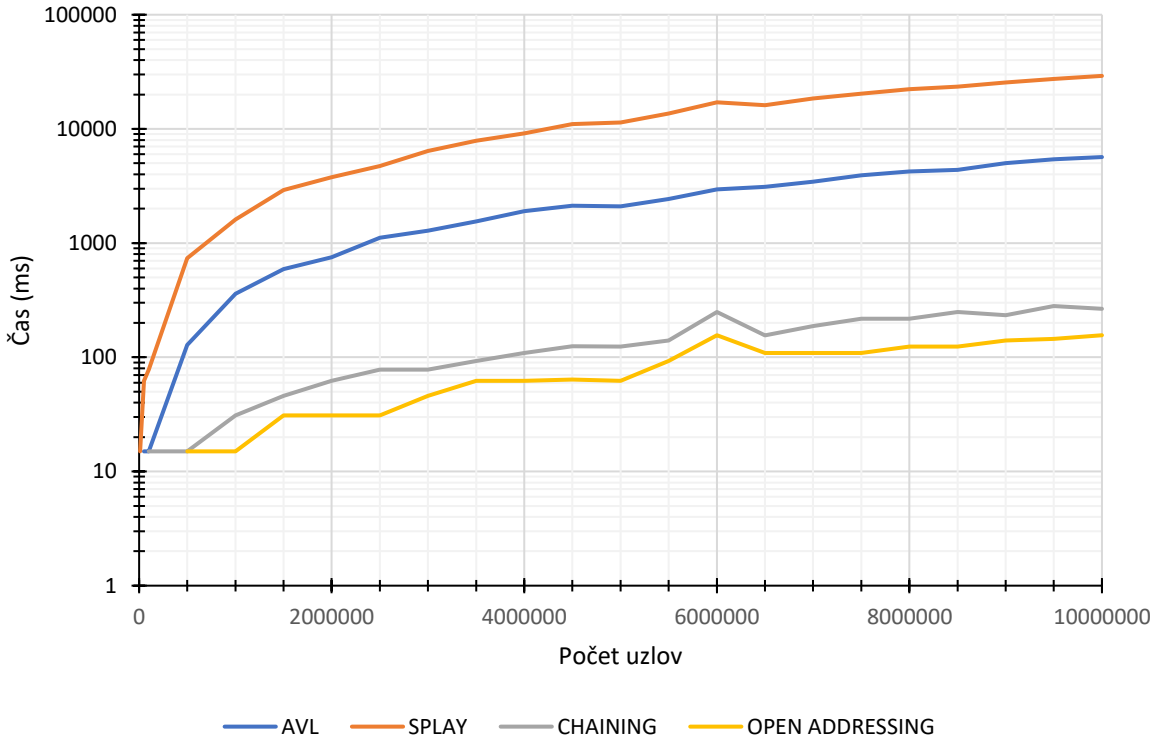
5.2. Vyhľadávanie

Prvé meranie operácie vyhľadávania pozostávalo z vyhľadania uzlov s kľúčmi zaradom. Výsledkom bolo zistenie, že nami implementované dátové štruktúry hešovacích tabuliek s otvoreným adresovaním a reťazením sú v tomto prípade rovnako efektívne a zároveň najefektívnejšie. Najmenej efektívna je dátová štruktúra binárneho vyhľadávacieho stromu splay (graf 3).



Graf 3: Výsledok vyhľadávania zaradom

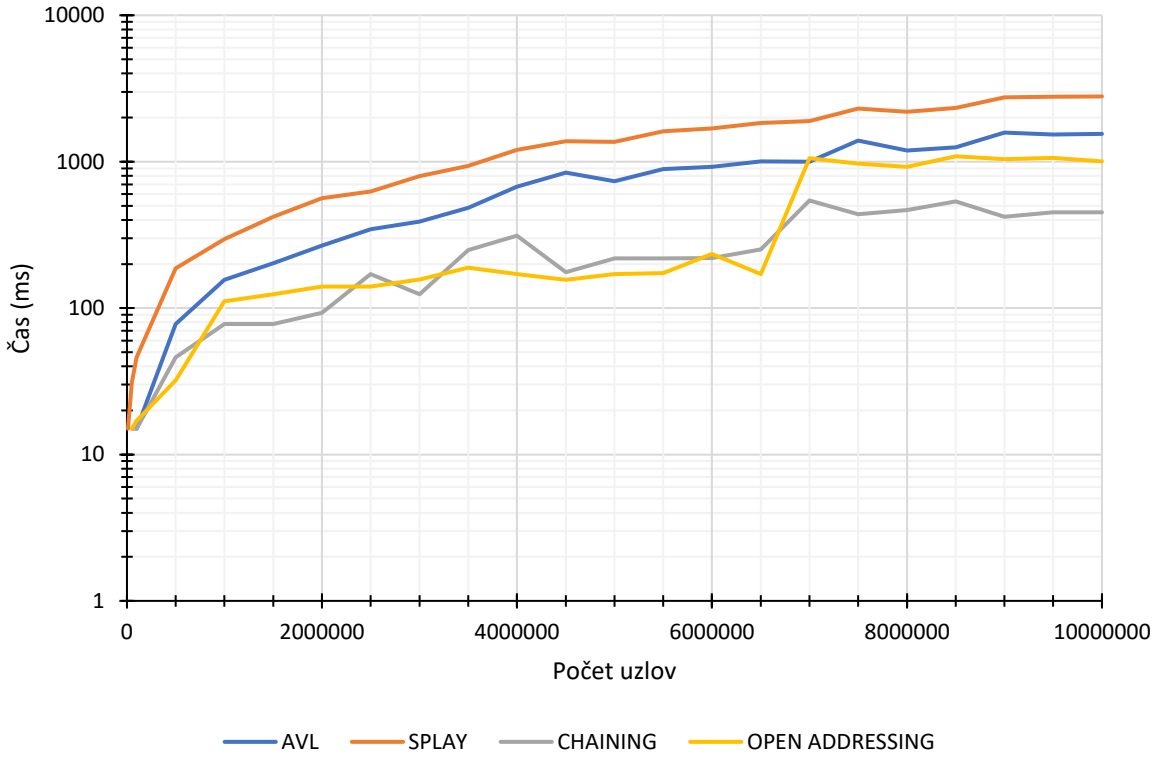
Druhé meranie operácie vyhľadávania pozostávalo z vyhľadania uzlov s náhodnou postupnosťou kľúčov. Výsledkom bolo zistenie, že nami implementované dátové štruktúry hešovacích tabuliek s otvoreným adresovaním a reťazením sú v tomto prípade takmer totožne efektívne no tabuľka s otvoreným adresovaním je najefektívnejšia. Najmenej efektívna je dátová štruktúra binárneho vyhľadávacieho stromu splay (graf 4).



Graf 4: Výsledok vyhľadávania s náhodnou postupnosťou

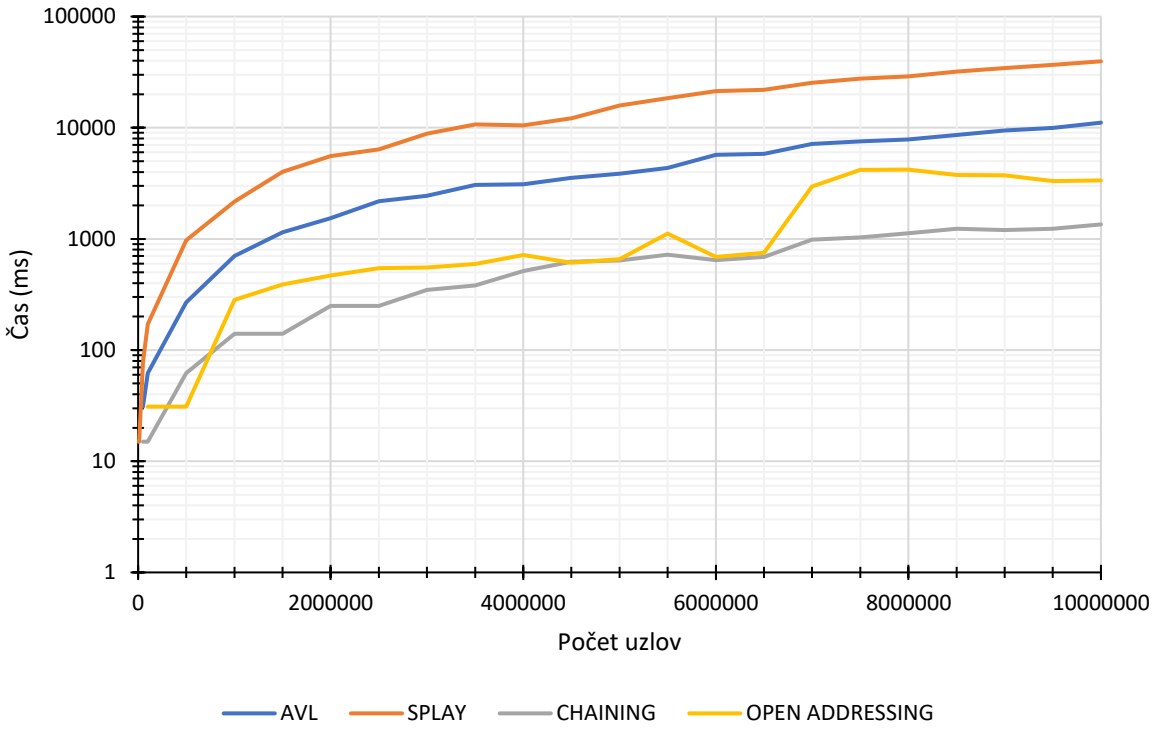
5.3. Vymazávanie

Prvé meranie operácie vymazávania pozostávalo z vymazania uzlov s kľúčmi zaradom. Výsledkom bolo zistenie, že nami implementovaná dátová štruktúra hešovacej tabuľky s reťazením je v tomto prípade najefektívnejšia, nakoľko hešovacia tabuľka s otvoreným adresovaním nadobudla okolo hodnôt 7 miliónov nevhodný časový posun. Najmenej efektívna je dátová štruktúra binárneho vyhľadávacieho stromu splay (graf 5).



Graf 5: Výsledok vymazávania zaradom

Druhé meranie operácie vymazávania pozostávalo z vymazania uzlov s náhodnou postupnosťou kľúčov. Výsledkom bolo zistenie, že nami implementovaná dátová štruktúra hešovacej tabuľky s reťazením je v tomto prípade najefektívnejšia. Hešovacia tabuľka s otvoreným adresovaním opäť nadobudla okolo hodnôt 7 miliónov nevhodný časový posun. Najmenej efektívna je dátová štruktúra binárneho vyhľadávacieho stromu splay (graf 6).



Graf 6: Výsledok vymazávania s náhodnou postupnosťou

6. Záver

V tomto zadaní sa nám úspešne podarilo implementovať 2 typy binárnych vyhľadávacích stromov a 2 typy hešovacích tabuliek s riešením kolízií. Implementované dátové štruktúry podľahli meraniu, počas ktorého sme zistili ich časovú efektivitu. Výsledkom bolo zistenie, že hešovacie tabuľky sú rýchlejšie na prácu s údajmi. Časovo najefektívnejšou dátovou štruktúrou je hešovacia tabuľka s otvoreným adresovaním, najmenej efektívny je binárny vyhľadávací strom splay, ktorý mal najhoršie merané časy vo všetkých meraniach.