Dátové štruktúry a algoritmy

Zadanie 1 – Vyhľadávanie v dynamických množinách

Tomáš Brček

FIIT STU, Bratislava

2023

Informácie k zadaniu

Programovací jazyk: Java 19

IDE: Intellij IDEA Community Edition 2022.3.2

OS: MS Windows 11 Home / Ubuntu 22.10

CPU: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, 2419 MHz, počet jadier: 4, počet logických procesorov: 8

RAM: 16 GB

AVL strom

AVL strom je vyvážený binárny strom, ktorý slúži na efektívne ukladanie a vyhľadávanie prvkov v pamäti. Jeho názov pochádza od jeho tvorcov Adelsona-Velského a Landisa.

```java
public class AVLTree{
    8 usages
    private Node root;
    8 usages
    private int total;

    11 usages
    public AVLTree(){
        this.root = null;
        this.total = 0;
    }
```

```java
class Node {
    int data;
    String str;
    Node left;
    Node right;
    Node parent;
    int height;

    public Node(int data, String str){
        this.data = data;
        this.str = str;
        this.left = null;
        this.right = null;
        this.parent = null;
        this.height = 1;
    }
}
```

V AVL strome platí, že výška každého podstromu sa líši maximálne o jednu úroveň. Tento rozdiel výšok sa nazýva balance faktor a môže byť -1, 0 alebo 1. Ak je balance faktor väčší ako 1 alebo menší ako -1, strom je nevyvážený.

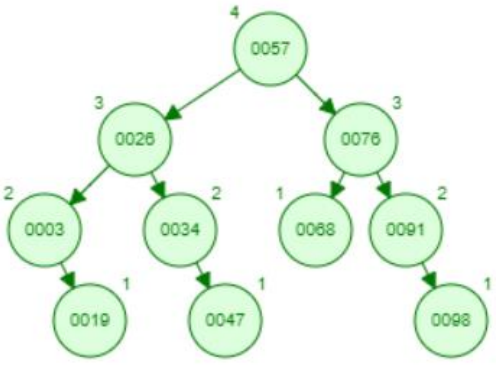
Pri vkladaní nového prvku do AVL stromu sa najprv prejde cestou od koreňa k listom a hľadá sa miesto, kam nový prvok patrí. Potom sa nový prvok vloží do stromu ako v bežnom binárnom strome, ale následne sa prejdú všetky uzly na ceste od vkladaného prvku ku koreňu a zisťuje sa, či balance faktor každého z týchto uzlov je v poriadku. Ak nie je, strom sa reorganizuje pomocou rotácií, aby sa zachovala vyváženosť.

Rotácie sú operácie, pri ktorých sa mení štruktúra stromu tak, aby sa zachovala vyváženosť. Existujú dva typy rotácií - jednoduchá rotácia a dvojitá rotácia. Pri jednoduchej rotácii sa presunú dva uzly tak, aby sa výška príslušných podstromov vyrovnala. Pri dvojitej rotácii sa aplikujú dve jednoduché rotácie na rôzne úrovne stromu.

Testovanie správnosti riešenia

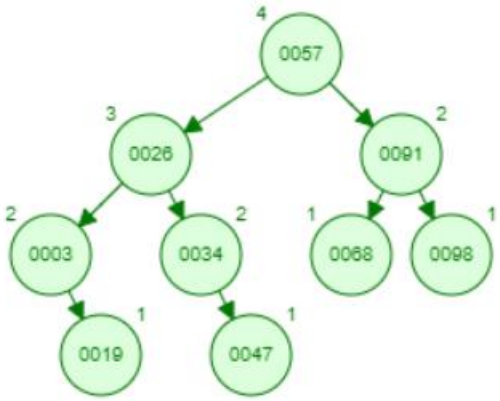
Uvažujme dáta: 57, 3, 91, 68, 34, 26, 47, 76, 98, 19

```
└── 57 (MBF9F)
    ├── 26 (fHvT3)
    │   ├── 3 (WNiL4)
    │   │   └── 19 (0wLuw)
    │   └── 34 (24Rur)
    │       └── 47 (Essnp)
    └── 76 (CeyY9)
        ├── 68 (CDiqw)
        └── 91 (V3xYy)
            └── 98 (OTpC2)
```



Po vymazaní údaju s číslom 76:

```
└── 57 (MBF9F)
    ├── 26 (fHvT3)
    │   ├── 3 (WNiL4)
    │   │   └── 19 (0wLuw)
    │   └── 34 (24Rur)
    │       └── 47 (Essnp)
    └── 91 (V3xYy)
        ├── 68 (CDiqw)
        └── 98 (OTpC2)
```



Splay strom

Splay strom je samovyvažovací binárny strom, ktorý slúži na efektívne ukladanie a vyhľadávanie prvkov v pamäti. Jeho hlavnou vlastnosťou je, že pri každom prístupe k uzlu sa daný uzol presunie na koreň stromu pomocou rotácií.

```java
public class SplayTree {
    private Node1 root;
    private int total;

    public SplayTree(){
        this.root = null;
        this.total = 0;
    }
```

```java
class Node1 {
    int data;
    String str;
    Node1 left;
    Node1 right;
    Node1 parent;

    public Node1(int data, String str){
        this.data = data;
        this.str = str;
        this.left = null;
        this.right = null;
        this.parent = null;
    }
}
```

Pri vkladaní nového prvku do Splay stromu sa najprv prejde cestou od koreňa k listom a hľadá sa miesto, kam nový prvok patrí. Potom sa nový prvok vloží do stromu ako v bežnom binárnom strome. Následne sa pomocou rotácií presunie vkladaný prvok a jeho rodič na koreň stromu, pričom sa snaží minimalizovať celkovú vzdialenosť.

Pri vyhľadávaní uzla v Splay strome sa najprv prejde cestou od koreňa k hľadanému uzlu. Potom sa hľadaný uzol pomocou rotácií presunie na koreň stromu. Ak sa hľadaný uzol nenájde, tak sa presunie jeho najbližší príbuzný.

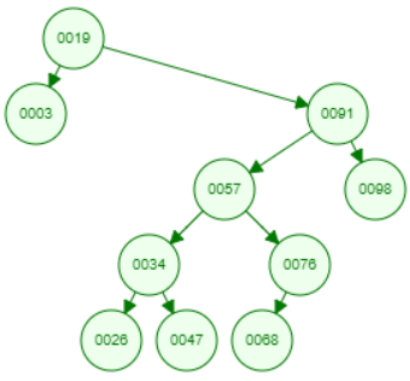
Splay strom má výhodu, že uzly, ktoré boli pristupované často, sú umiestnené blízko koreňa stromu. Tým sa zlepšuje výkonnosť pre opakované prístupy k rovnakým uzlom, pretože menej rotácií je potrebných na presun uzla na koreň stromu.

Nevýhodou Splay stromu je, že nie je zaručené, že strom bude úplne vyvážený, a teda môže sa stať, že niektoré operácie budú časovo náročné. Okrem toho, operácie vkladania a mazania môžu byť zložitejšie ako v iných stromoch.

Testovanie správnosti riešenia

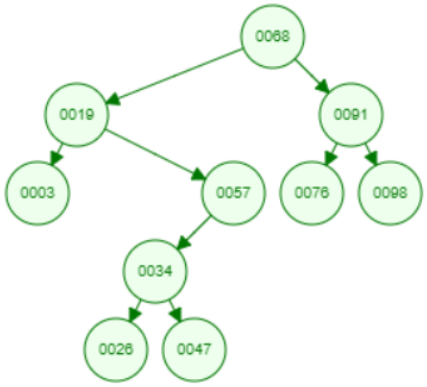
Uvažujme dáta: 57, 3, 91, 68, 34, 26, 47, 76, 98, 19

```
└── 19 (DAIxUvR6I9)
    ├── 3 (h4u7h8DmSb)
    └── 91 (snm5BcFQvU)
        ├── 57 (ENi7XGJv1x)
        │   ├── 34 (d7CCKinBTM)
        │   │   ├── 26 (RCEAp5K3rZ)
        │   │   └── 47 (RcsbVRKGOK)
        │   └── 76 (Vj8B6Z5Qxn)
        │       └── 68 (IFq7Sx6PQ8)
        └── 98 (0iP3i4wmeE)
```



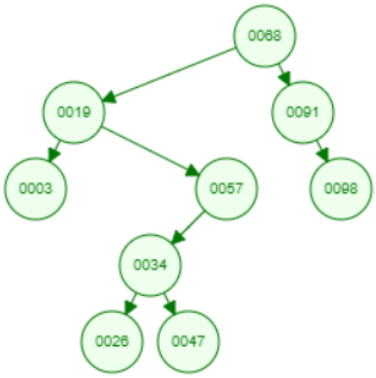
Po vyhľadaní údaju s číslom 68:

```
└── 68 (IFq7Sx6PQ8)
    ├── 19 (DAIxUvR6I9)
    │   ├── 3 (h4u7h8DmSb)
    │   └── 57 (ENi7XGJv1x)
    │       └── 34 (d7CCKinBTM)
    │           ├── 26 (RCEAp5K3rZ)
    │           └── 47 (RcsbVRKGOK)
    └── 91 (snm5BcFQvU)
        ├── 76 (Vj8B6Z5Qxn)
        └── 98 (0iP3i4wmeE)
```



Po vymazaní údaju s číslom 76:

```
└── 68 (IFq7Sx6PQ8)
    ├── 19 (DAIxUvR6I9)
    │   ├── 3 (h4u7h8DmSb)
    │   └── 57 (ENi7XGJv1x)
    │       └── 34 (d7CCKinBTM)
    │           ├── 26 (RCEAp5K3rZ)
    │           └── 47 (RcsbVRKGOK)
    └── 91 (snm5BcFQvU)
        └── 98 (0iP3i4wmeE)
```



Hashovanie s riešením kolízii pomocou reťazenia

```java
public class HashChain{
    int bucketNum;
    int size;
    HashData[] buckets;

    private static class HashData{
        String key;
        int value;
        HashData next;

        public HashData(String key, int value){
            this.key = key;
            this.value = value;
            this.next = null;
        }
    }

    public HashChain(int bucketNum){
        this.bucketNum = bucketNum;
        this.buckets = new HashData[this.bucketNum];
        this.size = 0;
        for(int i=0; i<this.bucketNum; i++){
            buckets[i] = null;
        }
    }
```

Hashovacia tabuľka je dátová štruktúra, ktorá slúži na efektívne ukladanie a vyhľadávanie prvkov na základe ich kľúča. Hashovacia funkcia mapuje kľúče na indexy v tabuľke. Keďže môže existovať viacero kľúčov, ktoré sa mapujú na ten istý index, môže dôjsť k situácii, kedy sa v hashovacej tabuľke nachádza viacero prvkov (kolízia).

```java
4 usages
private int hashFunction(String key){
    return Math.abs(key.hashCode() % this.bucketNum);
}
```

V prípade riešenia kolízií pomocou reťazenia sa každý prvok s rovnakým indexom ukladá do zoznamu, ktorý je uložený na tomto indexe. Tento zoznam sa nazýva reťazec a obsahuje všetky prvky s rovnakým hashom. Pri vyhľadávaní prvku sa najprv vypočíta jeho hash, nájde sa príslušný reťazec a prehľadá sa celý zoznam, kým sa nenájde prvok s hľadaným kľúčom.

Použitie reťazenia na riešenie kolízií má niekoľko výhod. Je to jednoduché a efektívne riešenie, ktoré vyžaduje minimálnu pamäťovú náročnosť. Navyše, ak je hashovacia funkcia dobre navrhnutá, tak je zriedkavé, že by sa reťazce stali príliš dlhými, čo by spomaľovalo vyhľadávanie.

Nevýhodou tohto riešenia je, že pri zvyšujúcej sa plnosti hashovacej tabuľky dochádza k zhoršovaniu jej efektivity. Čím viac prvkov sa nachádza v reťazcoch, tým dlhšie trvá ich prehľadávanie. Preto som v mojom riešení implementoval aj zmenu veľkosti hashovacej tabuľky.

```java
if((double)this.size / this.bucketNum >= 0.75){
    resize( newSize: 2 * this.bucketNum);
}
```

Zväčšenie v prípade, že podiel počtu prvkov a veľkosti tabuľky je viac ako 0,75.

```java
if((double)this.size / this.bucketNum <= 0.25){
    resize( newSize: bucketNum / 2);
}
```

Zmenšenie v prípade, že podiel počtu prvkov a veľkosti tabuľky je menej ako 0,25.

```
0:
1: q58Pgl75tf (712623) ->
2: uMIeN2pnxP (7436509) -> YnQ0tFfPbt (3440271) -> mcn47dTqiE (5910545) ->
3:
4:
5:
6:
7:
8:
9:
10: 34lvXNERs3 (6159679) ->
11: 41YUQZxSXR (3920554) -> zcR6wVprKt (7495361) ->
12:
13:
14:
15: 7JF4j4vJ4h (1096816) ->
16:
17: Kha8SbkKEK (1540485) ->
18: ni2K7awBfs (6731810) ->
19:
```

Ukážka riešenia hashovacej tabuľky pre 10 náhodných údajov.

Hashovacia tabuľka s riešením kolízii pomocou lineárneho open addressingu

```java
public class HashOpen{
    int capacity;
    int size;
    HashData[] slots;

    private static class HashData{
        String key;
        int value;

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

    public HashOpen(int bucketNum){
        this.capacity = bucketNum;
        this.slots = new HashData[this.capacity];
        this.size = 0;
        for(int i=0; i<this.capacity; i++){
            slots[i] = null;
        }
    }
```

Hashovacia tabuľka s riešením kolízii pomocou lineárneho open addressingu je ďalšou dátovou štruktúrou, ktorá slúži na efektívne ukladanie a vyhľadávanie prvkov na základe ich kľúča. Ak sa pri vkladaní nového prvku do hashovacej tabuľky vyskytne kolízia, tak sa nový prvok ukladá na ďalšiu voľnú pozíciu v tabuľke.

Konkrétne, ak sa prvok s rovnakým hashom ako nový prvok nachádza na pozícii, na ktorú sa má nový prvok vložiť, tak sa nový prvok uloží na nasledujúcu voľnú pozíciu v tabuľke. Toto sa opakuje, kým sa nenájde voľná pozícia v tabuľke.

Pri vyhľadávaní prvku sa najprv vypočíta jeho hash, nájde sa príslušný index a prehľadá sa tabuľka na tejto pozícii. Ak sa prvok s hľadaným kľúčom nenachádza na tejto pozícii, tak sa prehľadávajú nasledujúce pozície v tabuľke.

Výhodou riešenia kolízii pomocou lineárneho open addressingu je, že je jednoduché a má malú pamäťovú náročnosť.

Nevýhodou tohto riešenia je, že pri zvyšujúcej sa plnosti hashovacej tabuľky sa znižuje pravdepodobnosť, že nový prvok sa bude nachádzať na svojej pozícii. Tento problém riešim zmenou veľkosti hashovacej tabuľky podobne, ako v predchádzajúcom príklade.

```
0 -> ZYblyN3Uc3 ->940911
1 -> J0er2kdt9R ->4812669
2 ->
3 -> 8Tq60HPmBs ->5549117
4 -> 5n4ECdQ1r3 ->1460028
5 -> IwfGXIaCz8 ->5989905
6 -> GnEokVLZok ->6582258
7 -> dGKvPPOA4I ->5253812
8 ->
9 -> oNhrctgTMQ ->8739544
10 ->
11 ->
12 -> o3X57xuSl8 ->1446396
13 ->
14 ->
15 ->
16 ->
17 ->
18 ->
19 -> qGQmRG4EBo ->7327642
```

Ukážka riešenia hashovacej tabuľky
pre 10 náhodných údajov.

Časová zložitosť

Časová zložitosť algoritmu je miera, ako rýchlo sa zvyšuje čas potrebný na vykonanie algoritmu so zvyšujúcim sa vstupom. Zvyčajne sa meria počtom krokov (operácií) potrebných na vykonanie algoritmu. Časová zložitosť sa vyjadruje pomocou veľkosti vstupu (n). Čím rýchlejšie sa zvyšuje počet krokov potrebných na vykonanie algoritmu s rastúcou veľkosťou vstupu, tým vyššia je jeho časová zložitosť.

AVL tree

AVL tree	najhorší prípad	priemerný prípad
insert	O(log n)	O(log n)
search	O(log n)	O(log n)
delete	O(log n)	O(log n)

Splay tree

Splay tree	najhorší prípad	priemerný prípad
insert	O(n)	O(log n)
search	O(n)	O(log n)
delete	O(n)	O(log n)

Hashovanie s riešením kolízii pomocou reťazenia

Chaining	najhorší prípad	priemerný prípad
insert	O(n)	O(1)
search	O(n)	O(1)
delete	O(n)	O(1)

Hashovacia tabuľka s riešením kolízii pomocou lineárneho open addressingu

Open addressing	najhorší prípad	priemerný prípad
insert	O(n)	O(1)
search	O(n)	O(1)
delete	O(n)	O(1)

Testovanie

V tomto zadaní je našou úlohou otestovať rýchlosti rôznych dátových štruktúr. Z vyvažovacích algoritmov som si ako svoje štruktúry vybral AVL strom a Splay strom. Z riešenia kolízii v hashovacích tabuľkách som si ako prvú vybral metódu chainingu, ktorej rýchlosť budem porovnávať s metódou lineárneho open addressingu.

Rýchlosť týchto algoritmov budem porovnávať na rôznych poradiach metód insert, search a delete. A to presne:

1. insert
2. search
3. delete
4. insert -> delete
5. insert -> search
6. insert -> delete -> search
7. insert ->search -> delete

Do týchto metód budem postupne vkladať rôzny počet náhodných čísel a náhodných reťazcov. Postupne 100, 1000, 100 000, 200 000, 300 000, 400 000, 500 000, 600 000, 700 000, 800 000, 900 000, 1 000 000.

Pre všetky štruktúry potrebujem mať rovnaké vstupné dáta, ktoré po vygenerovaní posielam ako argument do funkcie, kde ho následne spracujem. V mojich štruktúrach netolerujem duplikáty a preto všetky dáta musia byť unikátne.

Na testovanie týchto situácii mám vytvorené funkcie, ktoré odmerajú čas a vypíšu ho do konzoly.

Vo funkcii main tieto testovacie funkcie postupne volám s rôznym počtom vstupných dát a čas meriam v milisekundách.

Výsledky - BVS

INSERT		
Počet prvkov	AVL /s	Splay /s
100	0,001	0
1000	0,001	0,002
100 000	0,063	0,123
200 000	0,078	0,153
300 000	0,141	0,194
400 000	0,179	0,282
500 000	0,283	0,398
600 000	0,361	0,478
700 000	0,472	0,6
800 000	0,555	0,683
900 000	0,634	0,973
1 000 000	0,744	0,969

SEARCH		
Počet prvkov	AVL /s	Splay /s
100	0	0
1000	0,001	0,001
100 000	0,031	0,063
200 000	0,047	0,11
300 000	0,063	0,189
400 000	0,094	0,368
500 000	0,14	0,48
600 000	0,173	0,54
700 000	0,299	0,541
800 000	0,272	0,758
900 000	0,3	0,923
1 000 000	0,381	1,143

DELETE		
Počet prvkov	AVL /s	Splay /s
100	0	0,001
1000	0,001	0,001
100 000	0,063	0,094
200 000	0,079	0,18
300 000	0,125	0,258
400 000	0,141	0,384
500 000	0,204	0,556
600 000	0,273	0,635
700 000	0,472	0,671
800 000	0,413	0,88
900 000	0,461	1,288
1 000 000	0,541	1,143

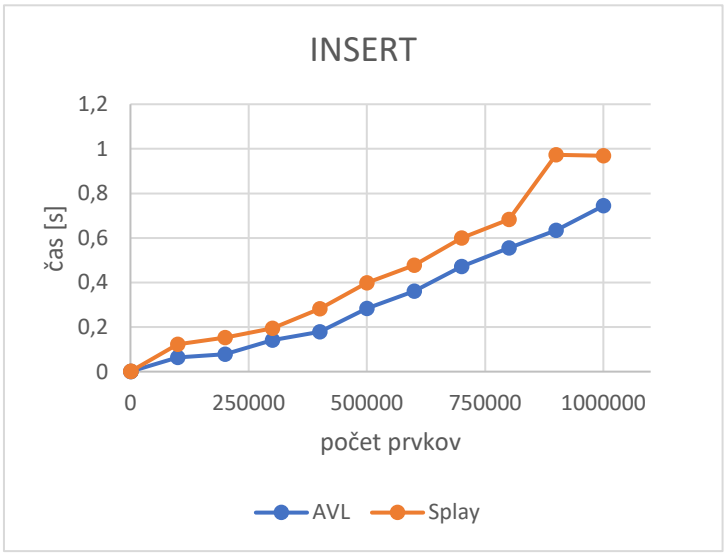
INSERT -> DELETE		
Počet prvkov	AVL /s	Splay /s
100	0,001	0
1000	0,001	0,001
100 000	0,078	0,146
200 000	0,189	0,336
300 000	0,245	0,439
400 000	0,383	0,635
500 000	0,572	0,782
600 000	0,774	1,039
700 000	0,922	1,714
800 000	0,952	1,474
900 000	1,257	1,746
1 000 000	1,477	2,035

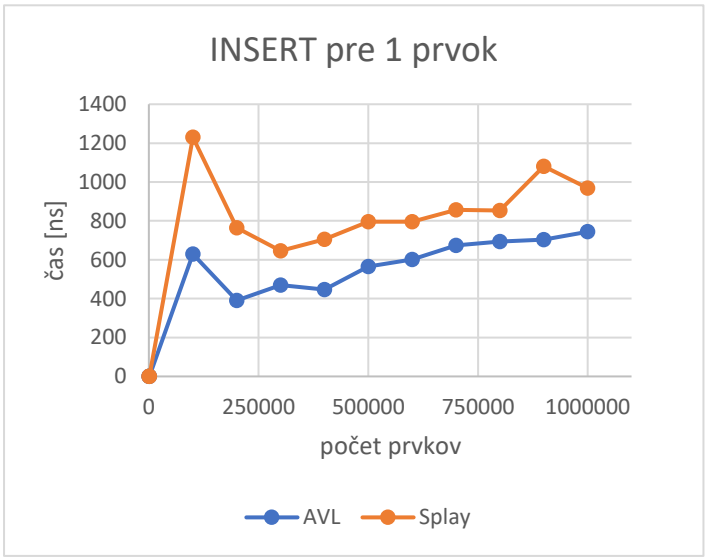
INSERT -> SEARCH		
Počet prvkov	AVL /s	Splay /s
100	0	0
1000	0,001	0,001
100 000	0,078	0,093
200 000	0,156	0,255
300 000	0,189	0,366
400 000	0,334	0,523
500 000	0,43	0,753
600 000	0,567	0,887
700 000	0,711	1,115
800 000	0,908	1,369
900 000	1,036	1,586
1 000 000	1,207	2,086

INSERT -> DELETE -> SEARCH		
Počet prvkov	AVL /s	Splay /s
100	0	0,001
1000	0,001	0,002
100 000	0,063	0,131
200 000	0,178	0,321
300 000	0,247	0,424
400 000	0,4	0,629
500 000	0,549	0,838
600 000	0,701	1,054
700 000	0,872	1,602
800 000	1,066	1,532
900 000	1,304	2,103
1 000 000	1,496	2,058

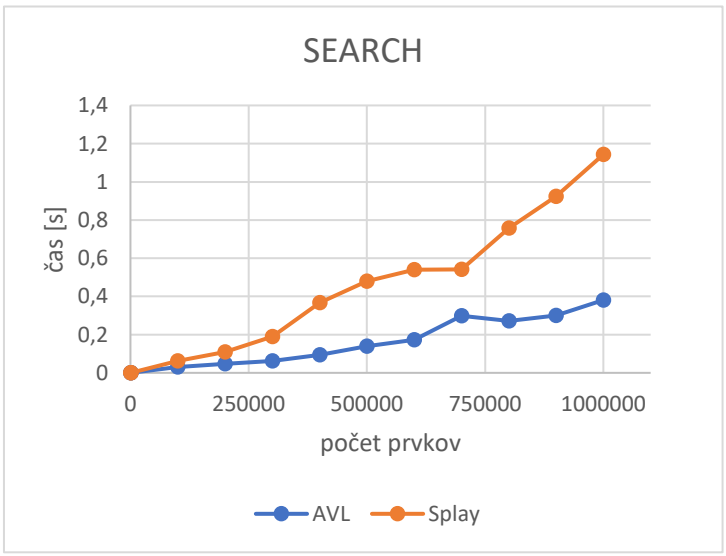
INSERT -> SEARCH -> DELETE		
Počet prvkov	AVL /s	Splay /s
100	0	0,001
1000	0,001	0,003
100 000	0,078	0,173
200 000	0,227	0,409
300 000	0,334	0,6
400 000	0,561	0,84
500 000	0,688	1,452
600 000	0,908	1,45
700 000	1,124	1,78
800 000	1,327	2,139
900 000	1,588	2,524
1 000 000	1,94	2,962

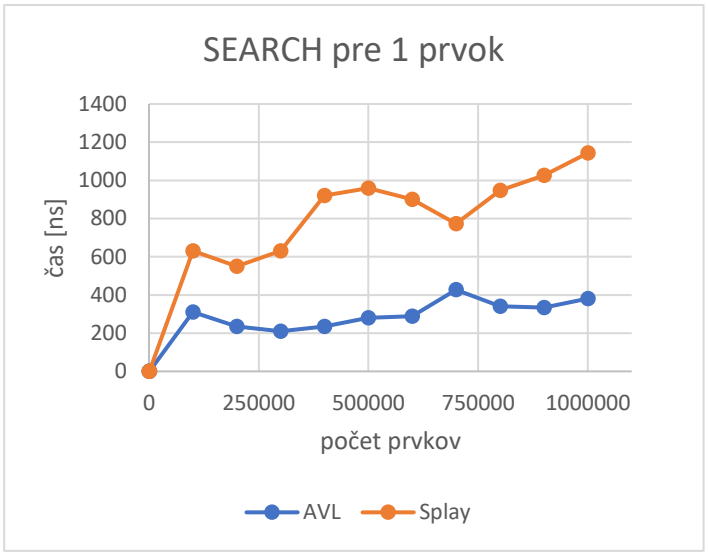
Grafy



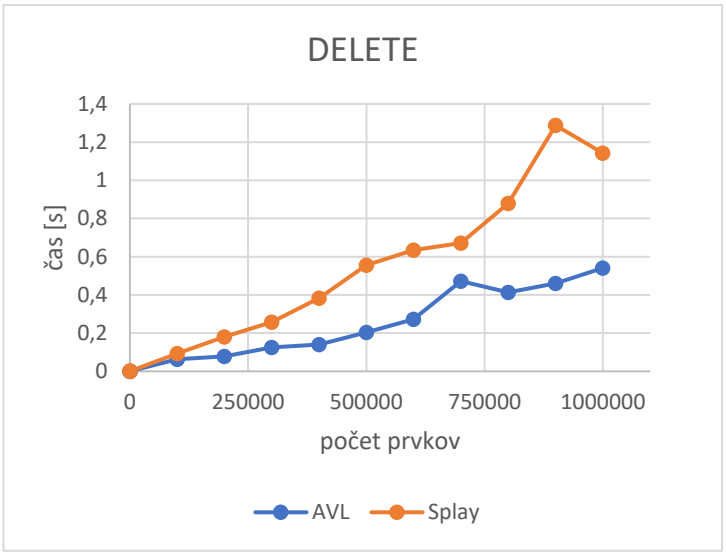


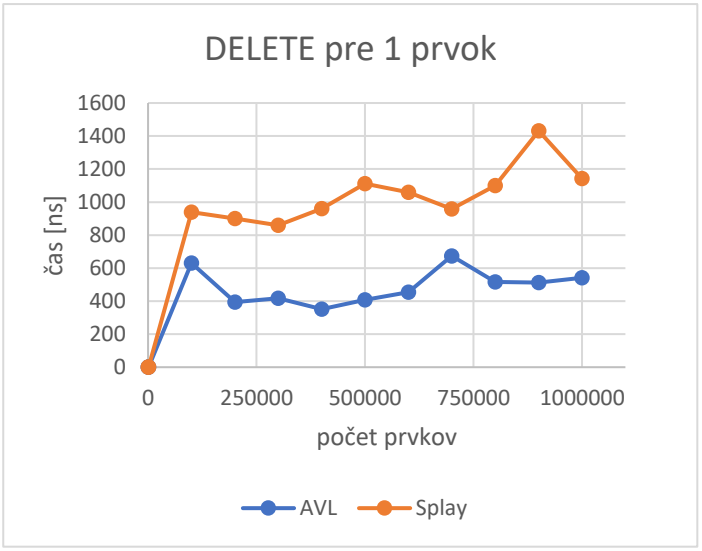
V tomto grafe môžeme vidieť porovnanie rýchlostí vkladania prvkov do AVL stromu a Splay stromu. Môžeme si všimnúť, že ich rýchlosti sú do počtu prvkov 800 000 približne rovnaké, ale pri väčších počtoch prvkov sa začne ukazovať, že AVL strom je pri vkladaní rýchlejší.



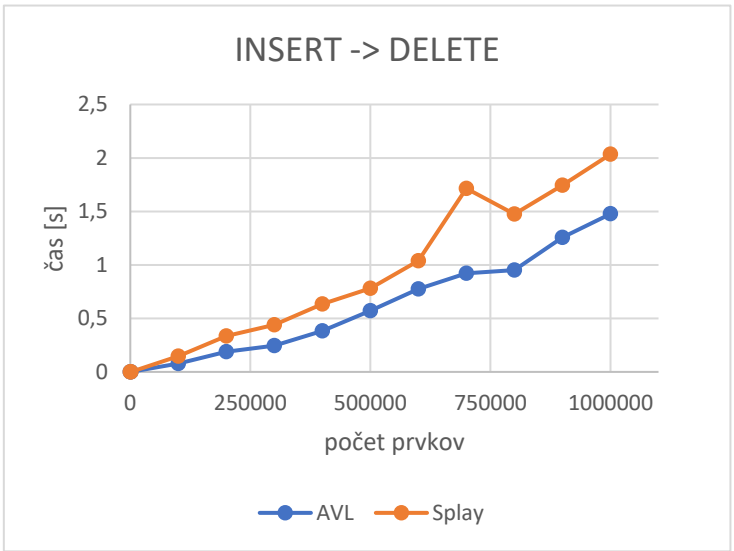


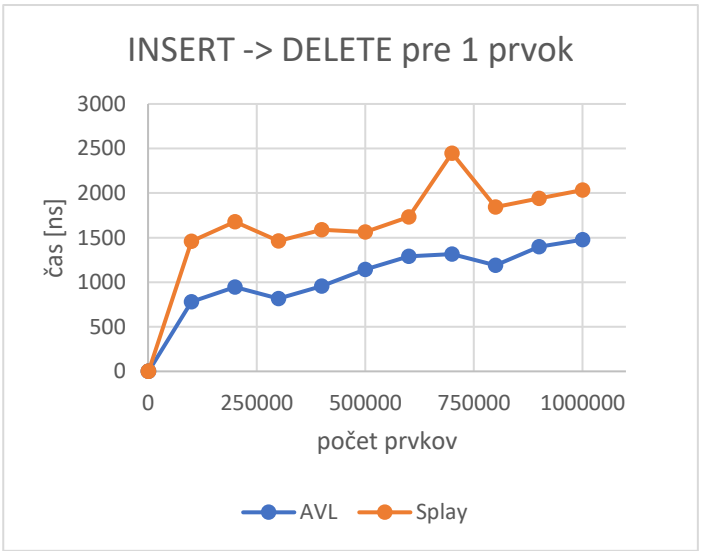
V druhom grafe môžeme vidieť, že metóda search je podstatne rýchlejšia pri AVL strome v porovnaní so Splay stromom.



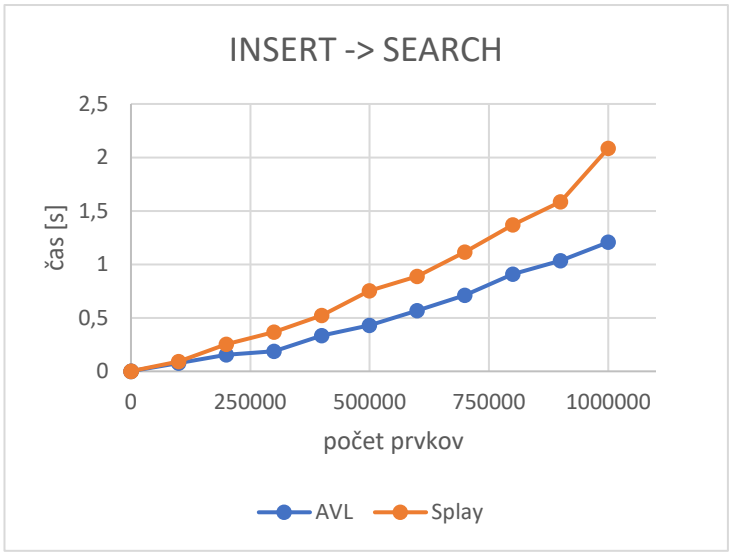


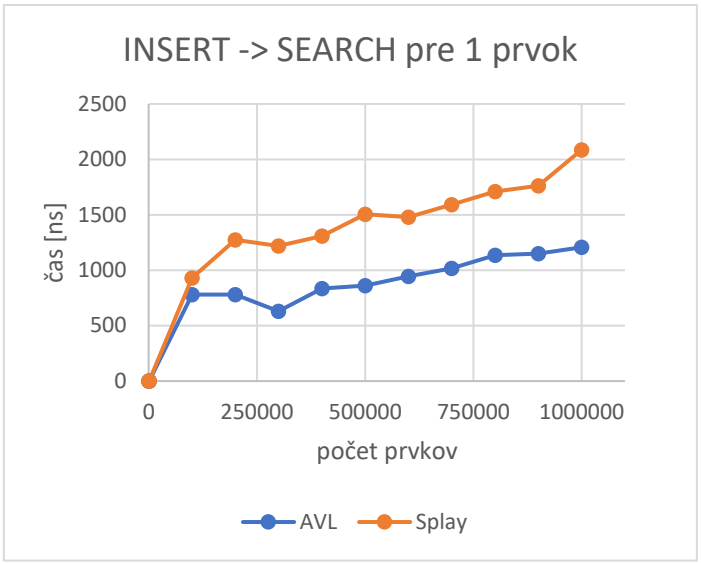
V tomto grafe je možné si všimnúť rozdielv rýchlosti metódy delete v stromoch AVL a Splay. Môžeme vidieť, že podobne ako pri metóde search je AVL strom o veľa rýchlejší.



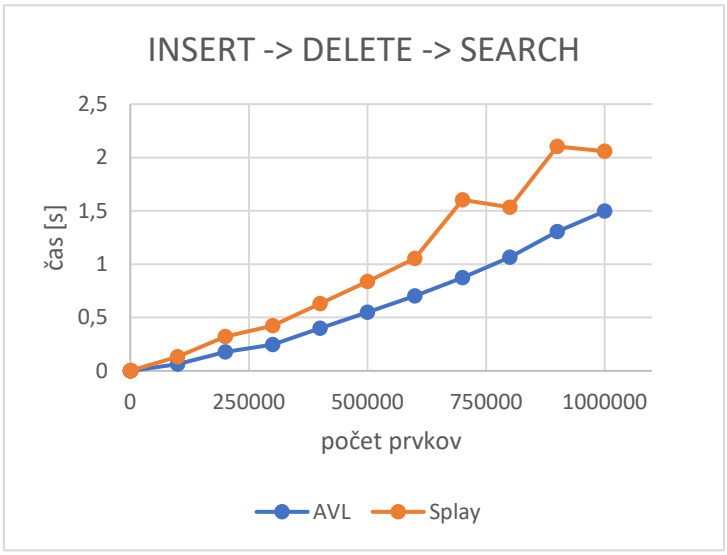


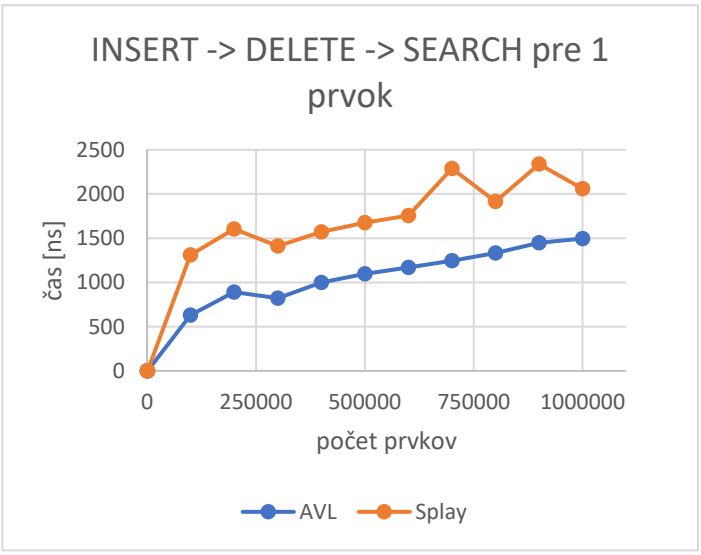
Nasledujúci graf ukazuje porovnanie rýchlostí metód insert a delete spolu. Táto metóda funguje tak, že odmeria čas vkladania určitého počtu prvkov a následného vymazania všetkých týchto prvkov. Výsledný čas bol zaznamenaný a zapísaný do tabuľky a grafu, v ktorom môžeme vidieť, že tieto dve metódy prebehnú rýchlejšie v AVL strome.



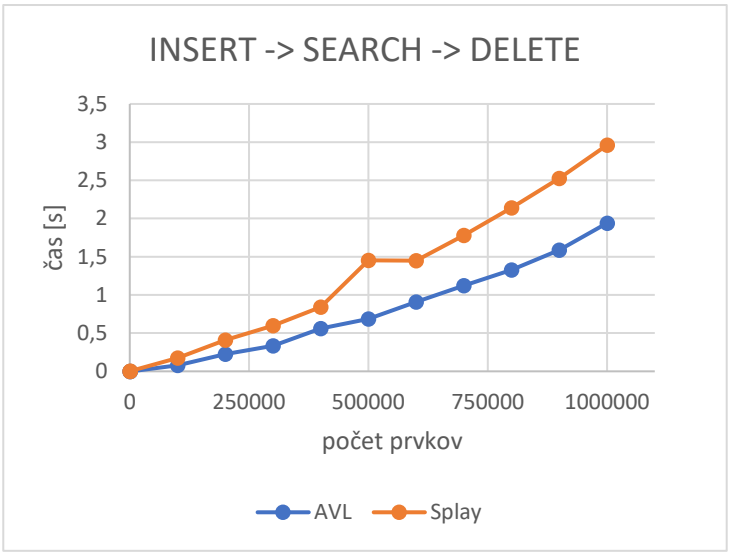


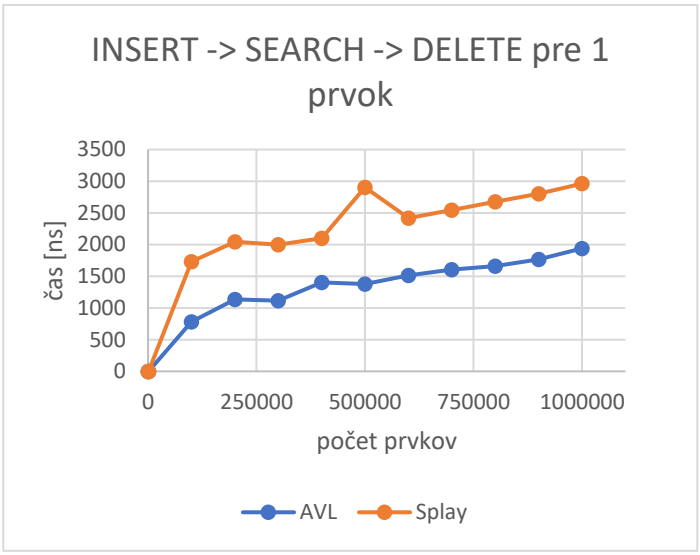
V ďalšom grafe je možné pozorovať porovnanie rýchlostí metód insert a search spolu. Táto metóda funguje tak, že odmeria čas vkladania určitého počtu prvkov a následného vyhľadania všetkých týchto prvkov. Výsledný čas bol zaznamenaný a zapísaný do tabuľky a grafu, v ktorom môžeme vidieť, že tieto dve metódy prebehnú rýchlejšie v AVL strome.





Graf vyššie zobrazuje porovnanie rýchlostí metód insert, delete a search v tomto poradí. Táto metóda funguje tak, že odmeria čas vkladania určitého počtu prvkov, následného vymazania všetkých týchto prvkov a pokus všetky tieto prvky vyhľadať. Výsledný čas bol zaznamenaný a zapísaný do tabuľky a grafu, v ktorom môžeme vidieť, že tieto tri metódy prebehnú rýchlejšie v AVL strome.





Nasledujúci graf zobrazuje porovnanie rýchlostí metód insert, search a delete v tomto poradí. Táto metóda funguje tak, že odmeria čas vkladania určitého počtu prvkov, následného vyhľadania všetkých týchto prvkov a taktiež vymazanie všetkých prvkov. Výsledný čas bol zaznamenaný a zapísaný do tabuľky a grafu, v ktorom môžeme vidieť, že tieto tri metódy prebehnú rýchlejšie v AVL strome.

Záverečné zhodnotenie vyvažovacích algoritmov

Po vykonaní všetkých testovacích scenárov, ktoré som si určil na začiatku, som zistil, že AVL strom dopadol lepšie vo všetkých vykonaných testoch. Môžem teda povedať, že moja implementácia AVL stromu je lepšie optimalizovaná a pre väčší počet vstupných dát beží rýchlejšie ako Splay strom. Čím väčší počet dát, tým je rozdiel medzi týmito dvomi algoritmami na vyvažovanie väčší, a preto sa viac oplatí používať binárny vyhľadávací strom s implementáciou AVL vyvažovacieho algoritmu.

Výsledky – Hashovanie

INSERT		
Počet prvkov	Chain /s	Open /s
100	0,001	0
1000	0,001	0,001
100 000	0,031	0,021
200 000	0,048	0,062
300 000	0,047	0,047
400 000	0,062	0,094
500 000	0,107	0,11
600 000	0,111	0,115
700 000	0,178	0,165
800 000	0,184	0,204
900 000	0,205	0,258
1 000 000	0,21	0,26

SEARCH		
Počet prvkov	Chain /s	Open /s
100	0,001	0
1000	0	0
100 000	0,011	0
200 000	0,015	0,016
300 000	0,015	0,015
400 000	0,015	0,016
500 000	0,018	0,023
600 000	0,02	0,032
700 000	0,032	0,03
800 000	0,032	0,035
900 000	0,036	0,047
1 000 000	0,04	0,047

DELETE		
Počet prvkov	Chain /s	Open /s
100	0	0
1000	0,001	0,001
100 000	0,024	0,024
200 000	0,033	0,024
300 000	0,031	0,027
400 000	0,046	0,047
500 000	0,046	0,062
600 000	0,062	0,062
700 000	0,069	0,094
800 000	0,104	0,11
900 000	0,109	0,118
1 000 000	0,109	0,146

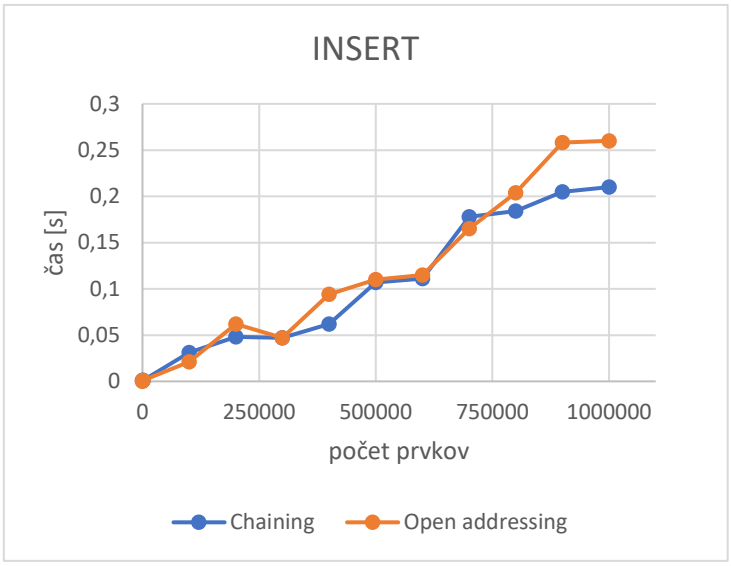
INSERT -> DELETE		
Počet prvkov	Chain /s	Open /s
100	0	0,001
1000	0,001	0,002
100 000	0,014	0,015
200 000	0,063	0,049
300 000	0,063	0,094
400 000	0,094	0,11
500 000	0,141	0,178
600 000	0,155	0,173
700 000	0,173	0,211
800 000	0,267	0,291
900 000	0,26	0,291
1 000 000	0,282	0,308

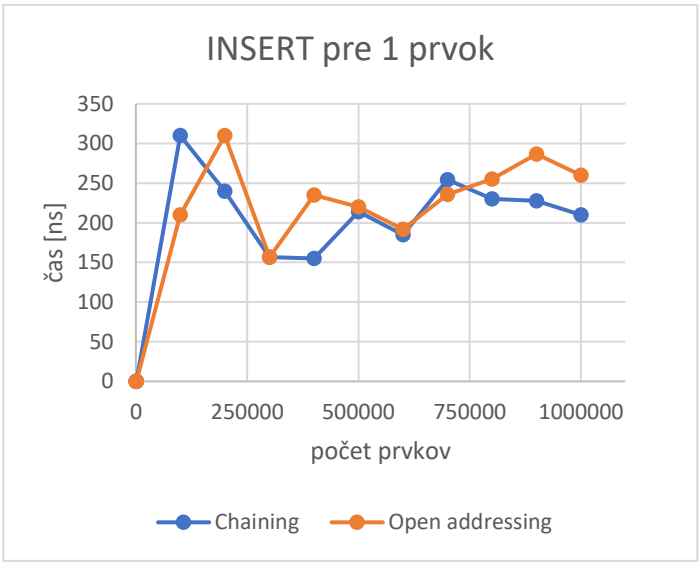
INSERT -> SEARCH		
Počet prvkov	Chain /s	Open /s
100	0	0,001
1000	0,001	0,001
100 000	0,015	0,031
200 000	0,047	0,041
300 000	0,046	0,094
400 000	0,047	0,065
500 000	0,094	0,141
600 000	0,1	0,188
700 000	0,11	0,188
800 000	0,188	0,193
900 000	0,11	0,189
1 000 000	0,247	0,298

INSERT -> DELETE -> SEARCH		
Počet prvkov	Chain /s	Open /s
100	0	0
1000	0,001	0
100 000	0,016	0,031
200 000	0,045	0,054
300 000	0,063	0,094
400 000	0,078	0,093
500 000	0,135	0,191
600 000	0,157	0,22
700 000	0,188	0,224
800 000	0,2	0,252
900 000	0,204	0,273
1 000 000	0,304	0,339

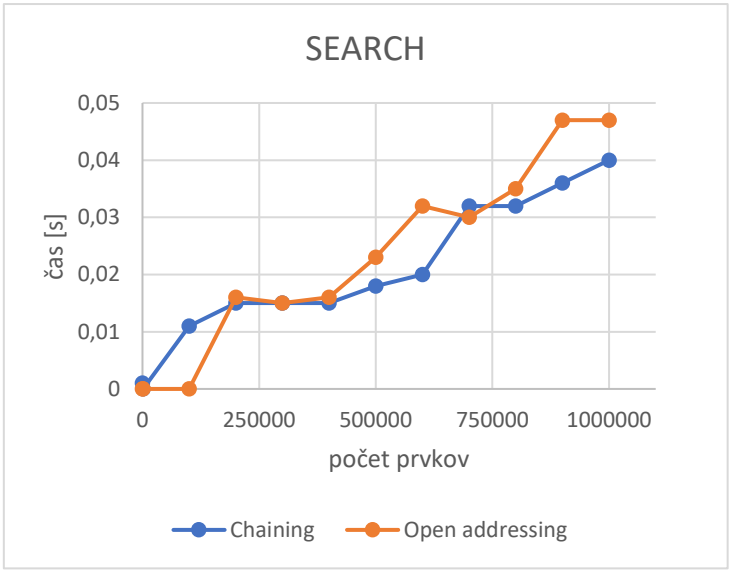
INSERT -> SEARCH -> DELETE		
Počet prvkov	Chain /s	Open /s
100	0	0
1000	0,001	0,001
100 000	0,032	0,016
200 000	0,038	0,063
300 000	0,1	0,072
400 000	0,076	0,101
500 000	0,142	0,194
600 000	0,19	0,224
700 000	0,188	0,251
800 000	0,267	0,303
900 000	0,188	0,35
1 000 000	0,336	0,575

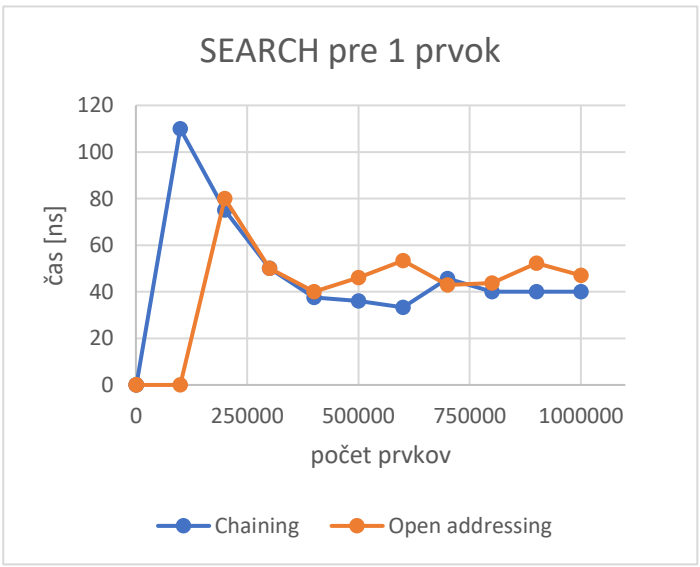
Grafy



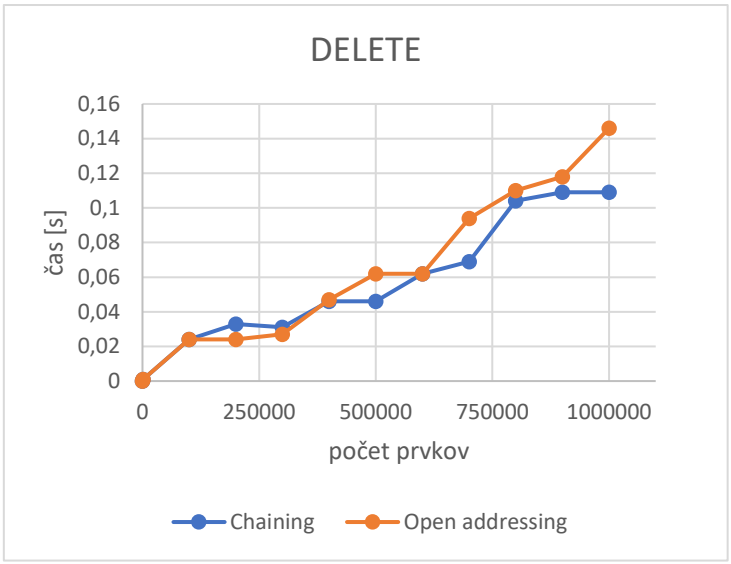


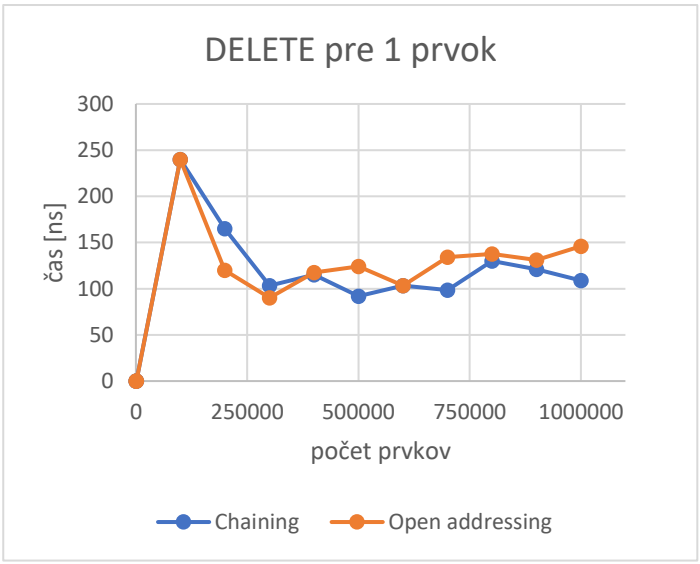
V tomto grafe môžeme vidieť porovnanie rýchlostí vkladania prvkov do hashovacej tabuľky s rôznym riešením kolízii. Môžeme si všimnúť, že ich rýchlosti sú približne rovnaké, rozdiel sa začne ukazovať až pri počte prvkov 900 000 a 1 000 000, kedy sa riešenie kolízii pomocou chainingu javí ako rýchlejšie.



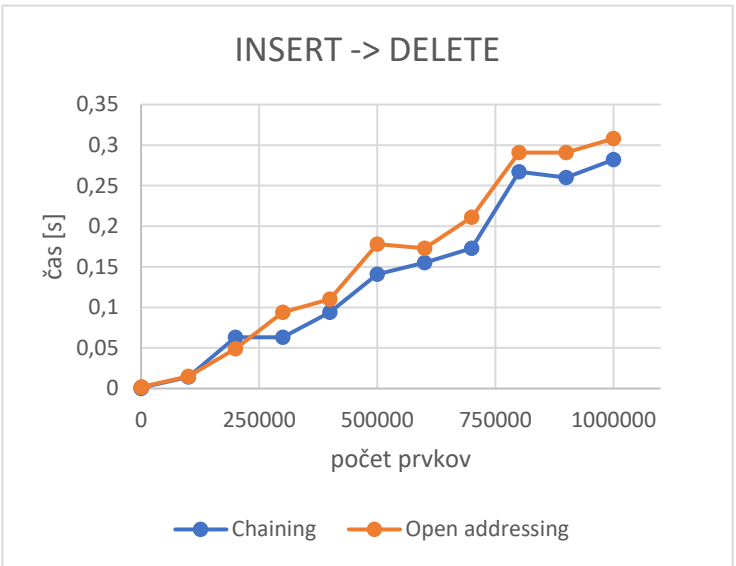


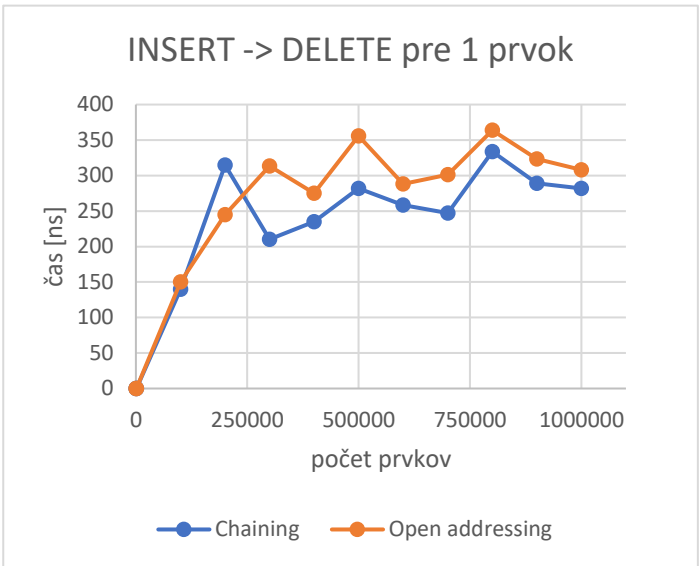
Nasledujúci graf zobrazuje porovnanie rýchlostí vyhľadania prvkov v hashovacej tabuľke s rôznym riešením kolízii. Môžeme vidieť, že v tomto prípade je hashovacia tabuľka s riešením kolízii pomocou chainingu o niečo rýchlejšia.



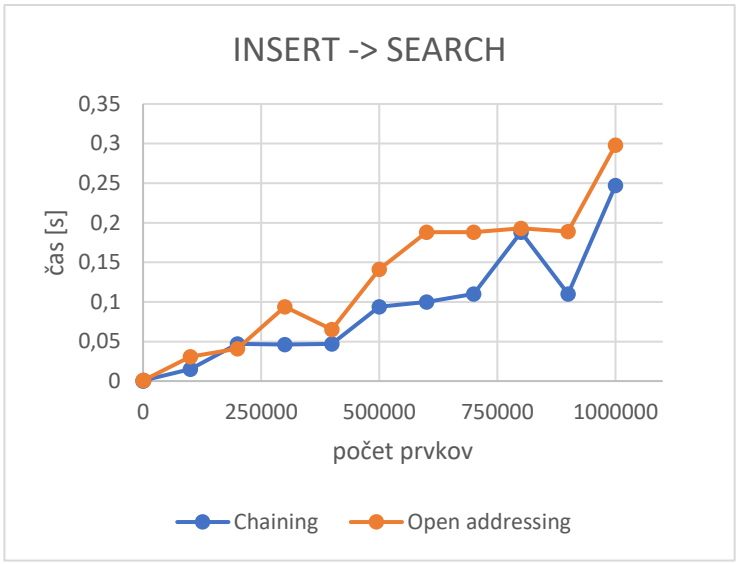


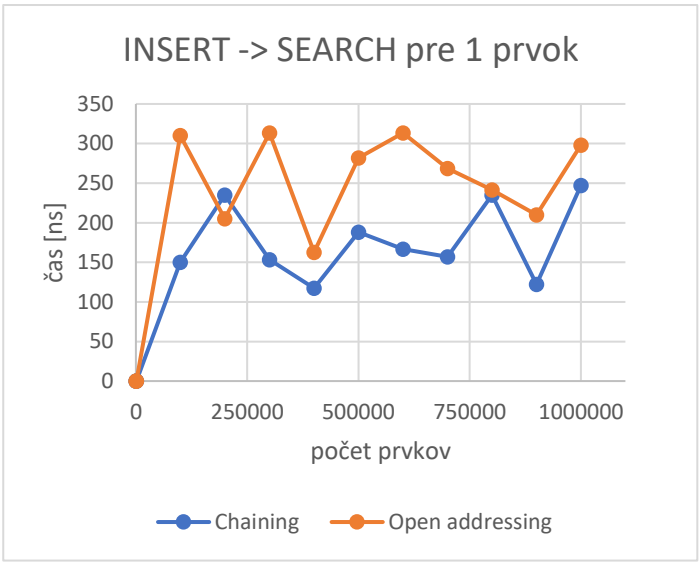
V tomto grafe je možné pozorovať, rozdiel v rýchlosti mazania prvkov z hashovacej tabuľky. Rozdiel medzi týmito dvomi riešeniami nie je veľký, ale predsa len riešenie kolízii pomocou chainingu vyzerá byť rýchlejšie.



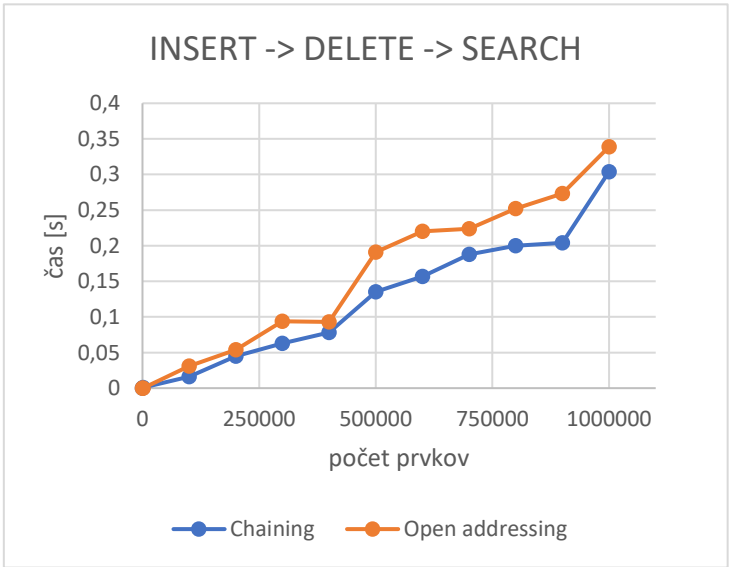


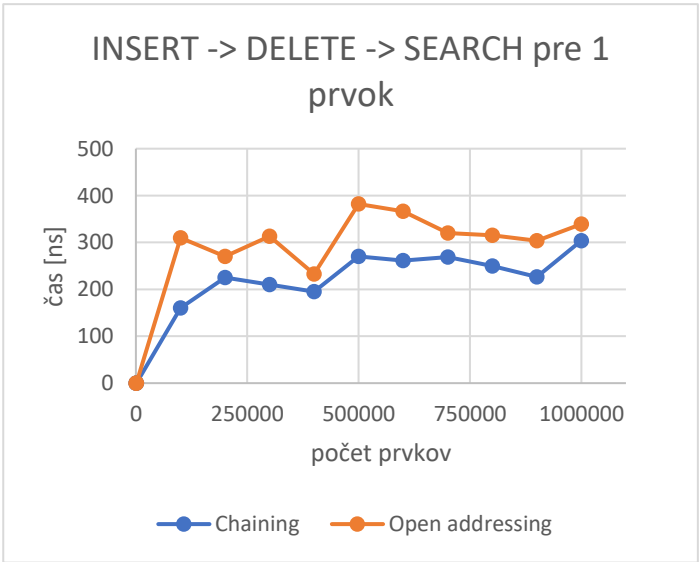
Nasledujúci graf ukazuje porovnanie rýchlostí metód insert a delete spolu. Táto metóda funguje tak, že odmeria čas vkladania určitého počtu prvkov a následného vymazania všetkých týchto prvkov. Výsledný čas bol zaznamenaný a zapísaný do tabuľky a grafu. Môžeme pozorovať, že tieto dve riešenia kolízii sú v tomto prípade takmer identické a rozdiely medzi nimi sú minimálne.



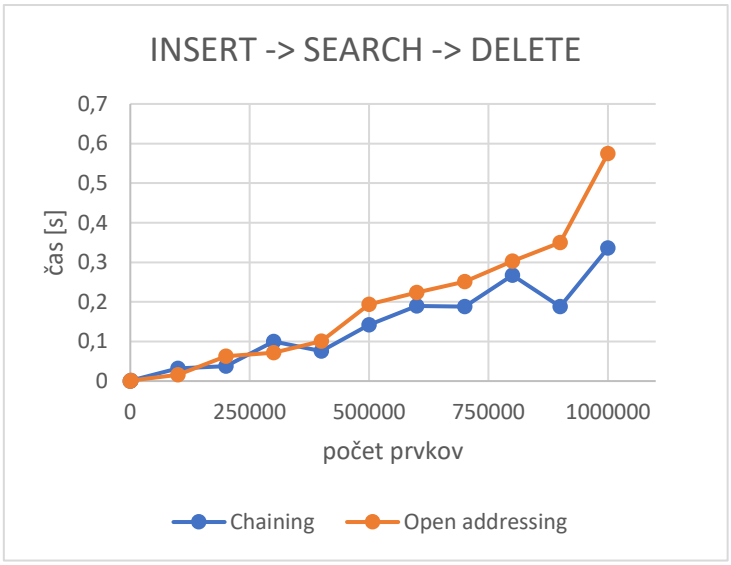


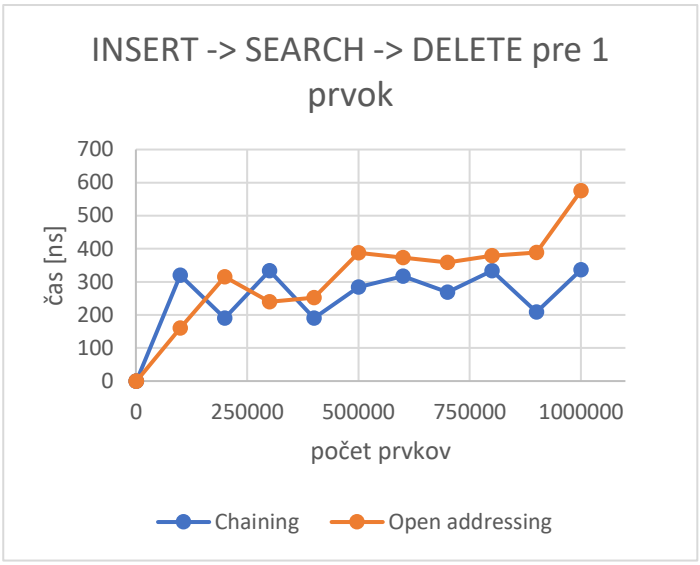
V ďalšom grafe je možné pozorovať porovnanie rýchlostí metód insert a search spolu. Táto metóda funguje tak, že odmeria čas vkladania určitého počtu prvkov a následného vyhľadania všetkých týchto prvkov v tabuľke. Výsledný čas bol zaznamenaný a zapísaný do tabuľky a grafu, v ktorom môžeme vidieť, že tieto dve metódy prebehnú rýchlejšie v hashovacej tabuľke s riešením kolízii formou chainingu, aj napriek tomu, že pri počte prvkov 800 000, bol čas takmer rovnaký.





Graf vyššie zobrazuje porovnanie rýchlostí metód insert, delete a search v tomto poradí. Táto metóda funguje tak, že odmeria čas vkladania určitého počtu prvkov, následného vymazania všetkých týchto prvkov a pokus všetky tieto prvky vyhľadať. Výsledný čas bol zaznamenaný a zapísaný do tabuľky a grafu. Po porovnaní výsledkov sa hashovacia tabuľka s riešením kolízii pomocou chainingu javí ako lepšie riešenie.





Nasledujúci graf zobrazuje porovnanie rýchlostí metód insert, search a delete v tomto poradí. Táto metóda funguje tak, že odmeria čas vkladania určitého počtu prvkov, následného vyhľadania všetkých týchto prvkov a taktiež vymazanie všetkých prvkov. Výsledný čas bol zaznamenaný a zapísaný do tabuľky a grafu, v ktorom môžeme vidieť, že tieto riešenia sú do počtu prvkov 800 000 takmer identické, ale pri väčšom počte prvkov sa riešenie kolízii formou chainingu ukazuje ako lepšie riešenie.

Záverečné zhodnotenie riešenia kolízii v hashovacích tabuľkách

Po vykonaní všetkých testovacích scenárov, ktoré som si určil na začiatku, som zistil, že hashovacia tabuľka s riešením kolízii pomocou chainingu dopadla o čosi lepšie takmer vo všetkých vykonaných testoch, aj napriek tomu, že rozdiely pri niektorých testoch boli minimálne. Môžem povedať, že moja implementácia hashovacej tabuľky s riešením kolízii pomocou chainingu je lepšie optimalizovaná a pre väčší počet vstupných dát beží rýchlejšie ako hashovacia tabuľka s open addressing riešením kolízii.

Vyhodnotenie

V tomto zadaní bolo našou úlohou implementovať a následne porovnať 4 implementácie dátových štruktúr z hľadiska efektivity operácii insert, delete a search v rozličných situáciách. Ako naše dátové štruktúry sme si mali zvoliť dve implementácie BVS s rôznymi algoritmami na vyvažovanie a dve implementácia hashovacej tabuľky s rôznym riešením kolízii.

Pomocou grafu som sa snažil zistiť, ktorý z algoritmov je najefektívnejší. Testoval som to na rôznych testovacích scenároch s rôznym počtom prvkov, tieto údaje som zapísal do tabuľky a zostrojil som graf, v ktorom môžeme pozorovať rozdiely medzi jednotlivými algoritmami.

Ako algoritmy na vyvažovanie som si zvolil AVL a Splay strom, z ktorých v mojom teste dopadol lepšie AVL strom. Na riešenie kolízii som si vybral metódu reťazenia (chainingu) a lineárneho open addressingu. Na základe testov mi vyšlo, že metóda chainingu je na tom o niečo lepšie, aj keď rozdiely medzi nimi nie sú veľké.

Zistil som, že hashovanie je podstatne rýchlejšie ako binárne vyhľadávacie stromy. Čím viac dát, tým je rozdiel medzi nimi väčší.