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

**2022/2023**

Dátové štruktúry a algoritmy

# Zadanie č.1

**Cvičiaca: Ing. Giang Nguyen Thu, PhD.**
**Čas cvičení: Pondelok 09:00 – 10:50**

**Vypracoval: Oleksandr Kovaliuk**
**AIS ID: 116220**

# Obsah

# Binárny vyhľadávací strom

## 1. Úvod

Binárny strom je dátová štruktúra pozostávajúca z uzlov, z ktorých každý môže mať od 0 do 2 potomkov - uzly na nižšej úrovni (rozlišujte medzi pravými a ľavými deťmi).
*Strom rastie zhora nadol.
*Na vrchole je koreň.
*Uzol bez detí sa nazýva list. Listy sú v spodnej časti stromu.
*Obrázok na objasnenie terminológie: koreň, listy, ľavé dieťa, pravé dieťa, úroveň.

## 2. AVL

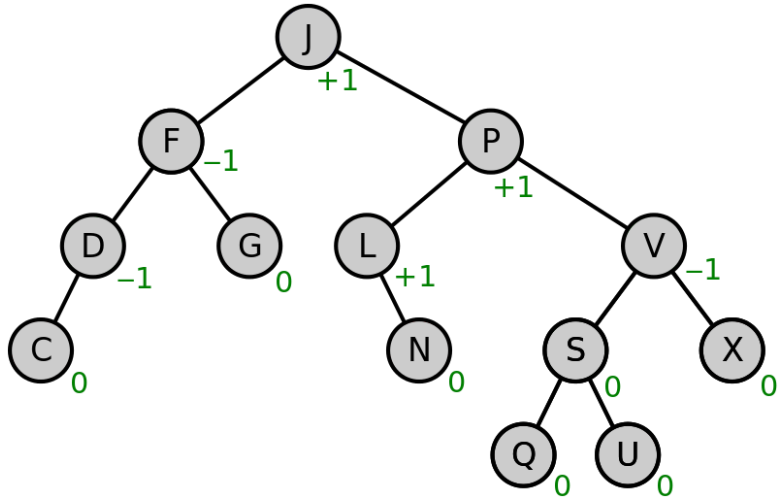
### 2.1. Začiatok

Tu je definovaný uzol a začiatok pre strom AVL

```
private int height;
public AVLTreeNode(int key, T value)
{
    this.key = key;
    this.height = 1;
    this.left = this.right = null;
    this.value = value;
}
```

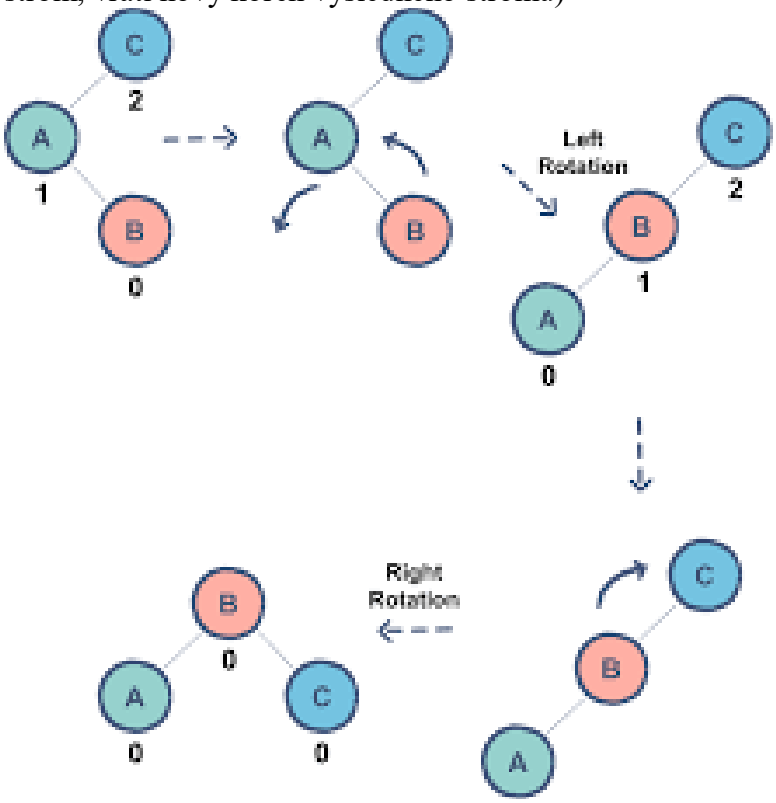
### 2.2. Balansovanie

Na implementáciu nášho stromu AVL musíme sledovať balance factorovi pre každý uzol v strome. Urobíme to tak, že sa pozrieme na výšku ľavého a pravého podstromu pre každý uzol. Formálnejšie definujeme faktor rovnováhy pre uzol ako rozdiel medzi výškou ľavého podstromu a výškou pravého podstromu.



```
int balance = getBalance(root);

if(balance > 1 && key < root.getLeft().getKey())
    return rightRotate(root);

else if(balance < -1 && key > root.getRight().getKey())
    return leftRotate(root);

else if(balance > 1 && key > root.getLeft().getKey())
{
    root.setLeft(leftRotate(root.getLeft()));
    return rightRotate(root);
}

else if(balance < -1 && key < root.getRight().getKey())
{
    root.setRight(rightRotate(root.getRight()));
    return leftRotate(root);
}
```

Kód, ktorý implementuje správnu rotáciu, vyzerá takto (ako obvykle, každá funkcia, ktorá zmení strom, vráti nový koreň výsledného stromu)



### 2.3 Insert/Search/Delete

```
// Insert
for (int i = 0; i < testsCount; i++)
{
    System.gc();
    test = new Test("AVLTree Fill " + i);
    test.startTime = System.nanoTime();
    test.memoryBefore = 0;
    ///
    tree = newTreeAvl();
    fillTreeRandom(tree, elementsCount);
    ///
    test.endTime = System.nanoTime();
    test.memoryAfter = (runtime.totalMemory() - runtime.freeMemory()-
test.memoryBefore)/(1024);
    testSet.testsInsert.add(test);
}
```

Začnime tým, že pred každou z troch funkcií zapnem garbage collector (gc). Toto je začiatok merania využitia pamäte od začiatku po vymazaní všetkého. Takže naše merania budú presnejšie. Začneme tiež merať rýchlosť funkcie. Vkladáme prvky do našej funkcie a fixujeme časové a pamäťové využitie programu

```
// Search
for (int i = 0; i < testsCount; i++)
{
    System.gc();
    test = new Test("AVLTree Search " + i);
    test.startTime = System.nanoTime();
    test.memoryBefore = 0;
    ///
    tree = newTreeAvl();
    fillTreeRandom(tree, elementsCount);
    tree.search(tree.getRoot(), randomInt(0, 10000000));
    ///
    test.endTime = System.nanoTime();
    test.memoryAfter = (runtime.totalMemory() - runtime.freeMemory()-
test.memoryBefore)/(1024);
    testSet.testsSearch.add(test);
}
```

Tu sa deje to isté, len tu už hľadáme nejaký prvok. Môže ich byť veľa, podľa toho, koľko si vyberieme.

```
// Delete
for (int i = 0; i < testsCount; i++)
{
    System.gc();
    test = new Test("AVLTree Delete " + i);
    test.startTime = System.nanoTime();
    test.memoryBefore = 0;
    ///
    tree = newTreeAvl();
    fillTreeRandom(tree, elementsCount);
    tree.delete(tree.getRoot(), randomInt(0, 10000000));
    ///
    test.endTime = System.nanoTime();
    test.memoryAfter = (runtime.totalMemory() - runtime.freeMemory()-
```

```
test.memoryBefore)/(1024);
    testSet.testsDelete.add(test);
}
```

Tu môžeme odstrániť akýkoľvek prvok. Používam tu náhodnú funkciu, ktorá určuje, ktorý prvok chceme odstrániť. Čo sa týka pamäte a rýchlosti, je tu všetko rovnaké ako v predchádzajúcich.

# 3. SPLAY

Binárny vyhľadávací strom neobsahujúci žiadne ďalšie v dátovej štruktúre (žiadne vyváženie, farba atď.)

## 3.1 Uvod Splay tree

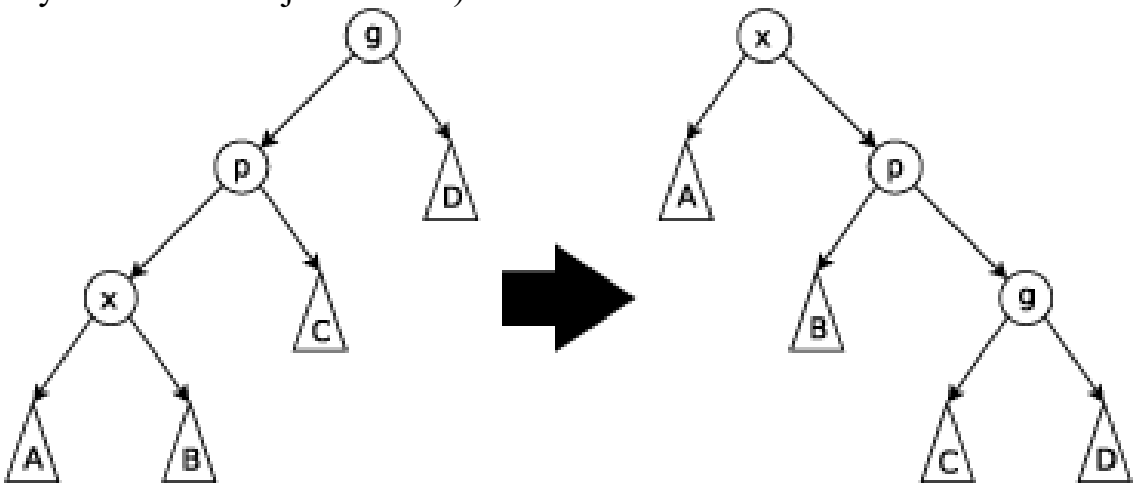
 Zaručená nelogaritmická zložitosť najhoršieho prípadu, a amortizovaná logaritmická zložitosť:

 Ľubovoľná postupnosť “m” operácií so slovníkom (hľadať, vkladať, mazať) cez „n“ prvkov, počnúc z prázdneho stromu, má zložitosť O(m log n)

 Priemerná zložitosť jednej operácie O(log n)

 Niektoré operácie môžu byť zložité Θ(n)

 Nie sú vytvorené žiadne distribučné predpoklady pravdepodobnosti stromových kľúčov a slovníkových operácií (t. j. že niektoré operácie boli vykonávané častejšie ako iné).



### 3.2 Splay rotation

```
public void splay(SplayTreeNode<T> node)
{
    if (node == null)
        return;

    while (node.getParent() != null)
    {
        SplayTreeNode<T> parent = node.getParent();
        SplayTreeNode<T> grandParent = parent.getParent();

        if (grandParent == null)
        {
            if (node == parent.getLeft())
                rightRotate(parent);
            else
                leftRotate(parent);
        }
        else if (node == parent.getLeft() && parent == grandParent.getLeft())
        {
            rightRotate(grandParent);
            rightRotate(parent);
        }
        else if (node == parent.getRight() && parent ==
grandParent.getRight())
        {
            leftRotate(grandParent);
            leftRotate(parent);
        }
        else if (node == parent.getRight() && parent ==
grandParent.getLeft())
        {
            leftRotate(parent);
            rightRotate(grandParent);
        }
        else
        {
            rightRotate(parent);
            leftRotate(grandParent);
        }
    }
}
```

Toto je metóda nazývaná splay, ktorá funguje na uzle binárneho vyhľadávacieho stromu typu SplayTreeNode. Účelom tejto metódy je posunúť daný uzol až ku koreňu stromu pri zachovaní vlastnosti binárneho vyhľadávacieho stromu. Metóda používa algoritmus rozloženia, čo je typ samonastavujúceho sa algoritmu binárneho vyhľadávacieho stromu. Algoritmus rozloženia upravuje štruktúru stromu v reakcii na postupnosť prístupových vzorov tak, že často prístupné uzly sa pohybujú bližšie ku koreňu a menej často prístupné uzly sa vzďaľujú od koreňa. Metóda začína kontrolou, či je daný uzol nulový, v takom prípade sa jednoducho vráti bez toho, aby urobil čokoľvek. Ak daný uzol nie je nulový, metóda vstúpi do cyklu, ktorý pokračuje, pokiaľ má uzol rodiča. V rámci cyklu metóda skúma rodiča a starého rodiča aktuálneho uzla, aby určila, ako vykonať sériu rotácií, ktoré posunú aktuálny uzol až ku koreňu stromu.

P.S. ***Vložiť/vyhľadať/vymazať funguje pre tento binárny strom, ako aj pre predchádzajúci s malými zmenami v dôsledku zriedkavých rotácií***

# Hashovanie

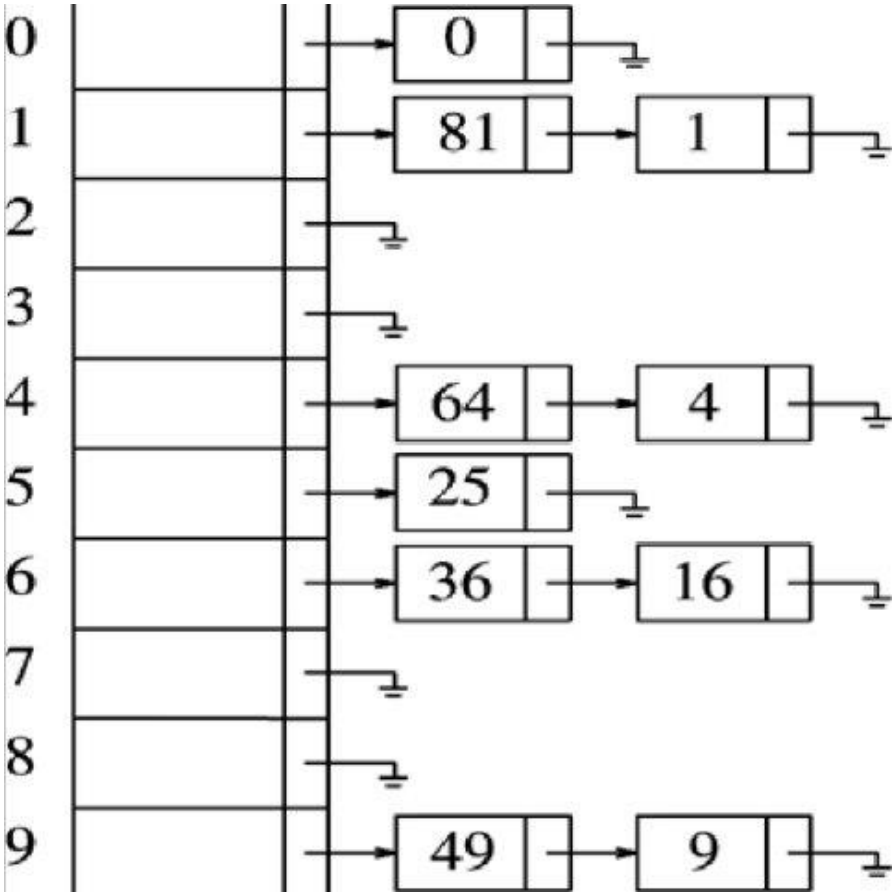
## 4. Úvod

Hašovanie je technika používaná v programovaní na mapovanie údajov ľubovoľnej veľkosti na hodnotu s pevnou veľkosťou, zvyčajne na číselný index alebo hašovací kód. Na údaje sa aplikuje hašovacia funkcia, ktorá vytvára hašovaciu hodnotu, ktorú možno použiť na indexovanie do dátovej štruktúry, ako je hašovacia tabuľka alebo množina hašovacích údajov. Hašovanie sa bežne používa na rýchle získavanie a porovnávanie údajov, ako aj na účely zabezpečenia a integrity údajov, ako sú digitálne podpisy a hašovanie hesiel..

## 5. Chaining HashTable

### 5.1. Uvod pre Chaining HashTable

Chaining HashTable je dátová štruktúra, ktorá používa hašovaciu funkciu na mapovanie kľúčov na indexy v poli. Namiesto ukladania hodnôt priamo do poľa pri týchto indexoch sa na riešenie kolízií používa prepojený zoznam, kde sa viaceré kľúče mapujú na rovnaký index. Každý prvok v poli je hlavičkou prepojeného zoznamu a hodnoty sa vkladajú na koniec zoznamu, ktorý zodpovedá indexu kľúča. To umožňuje efektívne vkladanie a vyhľadávanie hodnôt aj pri veľkom počte kolízií.



## 5.2. Collision Resolution Strategy

```
private void collision(K key, V value, int index)
{
    HashItemChaining<K, V> item = hashTable[index];

    while (item.getNext() != null)
    {
        if (item.getKey().equals(key))
        {
            item.setValue(value);
            return;
        }
        item = item.getNext();
    }
    if (item.getKey().equals(key))
    {
        item.setValue(value);
        return;
    }
    item.setNext(new HashItemChaining<>(key, value));
    count++;
}
```

Metóda má tri argumenty: kľúč a hodnotu nového prvku a index, do ktorého by mal byť prvok vložený na základe jeho hash hodnoty. Tu je podrobný rozpis toho, čo kód robí:

Metóda získa existujúcu položku hash tabuľky na danom indexe a priradí ju k položke lokálnej premennej.

Metóda potom vstúpi do cyklu while, ktorý iteruje cez prepojený zoznam položiek počnúc položkou. Pokračuje v iterácii cez zoznam, pokiaľ item.getNext() vracia nenulovú hodnotu, čo znamená, že v zozname je viac položiek na kontrolu.

Vo vnútri cyklu while metóda skontroluje, či sa kľúč aktuálnej položky zhoduje s novým kľúčom, ktorý sa vkladá, volaním item.getKey().equals(key). Ak sú rovnaké, znamená to, že kľúč už v tabuľke existuje a zodpovedajúca hodnota by sa mala aktualizovať na novú vkladanú hodnotu. Metóda potom nastaví novú hodnotu volaním item.setValue(value) a vráti sa z metódy.

Ak kľúče nie sú rovnaké, metóda sa presunie na ďalšiu položku v zozname nastavením položky na item.getNext() a pokračuje v iterácii cez zoznam.

Po dosiahnutí konca zoznamu metóda skontroluje, či sa kľúč poslednej položky rovná novému kľúču. Ak sú rovnaké, hodnota existujúcej položky by sa mala aktualizovať na novú hodnotu. Metóda potom nastaví novú hodnotu volaním item.setValue(value) a vráti sa z metódy.

Ak sa kľúče nezhodujú, znamená to, že nový pár kľúč – hodnota je potrebné pridať na koniec prepojeného zoznamu. Metóda vytvorí nový objekt HashItemChaining s novým párom kľúč-hodnota a nastaví ho ako ďalšiu položku v zozname volaním item.setNext(new HashItemChaining<>(kľúč, hodnota)). Metóda tiež zvyšuje premennú počet, aby sledovala počet položiek v tabuľke.

Celkovo tento kód implementuje reťazcovú stratégiu riešenia kolízií, kde sa prepojený zoznam položiek používa na riešenie kolízií, ku ktorým dochádza, keď viaceré kľúče hašujú rovnaký index v tabuľke.

### 5.3. Realisation of put/get/remove

```
@Override
public void put(K key, V value)
{
    if(count == (int)(size * 0.5))
    {
        size *= 2;
        HashItemChaining<K, V>[] newHashTable = new HashItemChaining[size];

        for (HashItemChaining<K, V> kvHashItemChaining : hashTable)
        {
            HashItemChaining<K, V> item = kvHashItemChaining;
            while (item != null)
            {
                int index = hash(item.getKey());

                if (newHashTable[index] == null)
                    newHashTable[index] = new
HashItemChaining<>(item.getKey(), item.getValue());

                else
                {
                    HashItemChaining<K, V> newItem = newHashTable[index];

                    while (newItem.getNext() != null)
                        newItem = newItem.getNext();

                    newItem.setNext(new HashItemChaining<>(item.getKey(),
item.getValue()));
                }
                item = item.getNext();
            }
        }
        hashTable = newHashTable;
    }

    int index = hash(key);
    if (hashTable[index] == null)
    {
        hashTable[index] = new HashItemChaining<>(key, value);
        count++;
        return;
    }
    collision(key, value, index);
}
```

Táto metóda vloží do hašovacej tabuľky nový pár kľúč – hodnota. Ak je počet položiek v tabuľke väčší alebo rovný polovici veľkosti tabuľky, veľkosť tabuľky sa zdvojnásobí a všetky existujúce položky sa prepracujú a vložia do novej tabuľky. Metóda potom vypočíta hašovací index nového kľúča a skontroluje, či už položka v tomto indexe existuje. Ak neexistuje žiadna položka, vytvorí sa nový objekt HashItemChaining a vloží sa do indexu. Ak položka už existuje, metóda zavolá metódu kolízie() na spracovanie kolízie pridaním nového páru kľúč – hodnota do prepojeného zoznamu v indexe.

```
@Override
public V get(K key)
{
    int index = hash(key);
    HashItemChaining<K, V> item = hashTable[index];

    while (item != null)
    {
        if (item.getKey().equals(key))
            return item.getValue();

        item = item.getNext();
    }
    return null;
}
```

Táto metóda načíta hodnotu spojenú s daným kľúčom z hašovacej tabuľky. Metóda vypočíta hash index kľúča a iteruje cez prepojený zoznam v tomto indexe, kým nenájde položku so zodpovedajúcim kľúčom. Ak sa nájde zodpovedajúca položka, vráti sa zodpovedajúca hodnota. Ak sa nenájde žiadna zodpovedajúca položka, vráti sa hodnota null.

```
@Override
public void remove(K key)
{
    int index = hash(key);
    HashItemChaining<K, V> prev = null;
    HashItemChaining<K, V> item = hashTable[index];

    while (item != null)
    {
        if (item.getKey().equals(key))
        {
            if (prev == null)
            {
                hashTable[index] = item.getNext();
                count--;
                return;
            }
            prev.setNext(item.getNext());
            count--;
            return;
        }
        prev = item;
        item = item.getNext();
    }
}
```

Táto metóda odstráni pár key –value z hašovacej tabuľky na základe daného kľúča. Metóda vypočíta hash index kľúča a iteruje cez prepojený zoznam v tomto indexe, kým nenájde položku so zodpovedajúcim kľúčom. Ak sa nájde zodpovedajúca položka, odstráni sa zo zoznamu aktualizáciou nasledujúcej referencie predchádzajúcej položky tak, aby ukazovala na nasledujúcu položku. Ak je zodpovedajúca položka prvou položkou v zozname, pole hashTable sa aktualizuje tak, aby ukazovalo na ďalšiu položku v zozname. Metóda tiež znižuje premennú počet, aby odrážala znížený počet položiek v tabuľke.

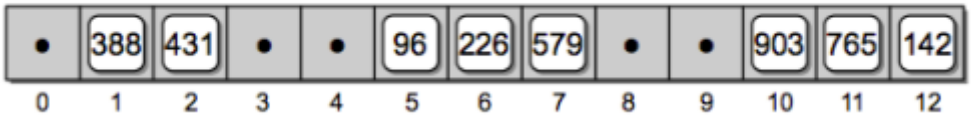
# 6. „Closed“ Hashing

To je to dátová štruktúra, ktorá mapuje kľúče na hodnoty a používa hashovaciu funkciu na výpočet kľúčových indexov.

### 6.1. Uvod pre Closed HashTable

V uzavretom hašovaní používate iba jedno pole na všetko. Kolízie ukladáte do rovnakého poľa. Trik je použiť nejaký šikovný spôsob, ako skákať z kolízie do kolízie, kým nenájdete to, čo chcete. A urobte to v reprodukovateľnom formáte

```
h(765) => 11          h(579) => 7
h(431) => 2           h(226) => 5   => 6
h(96)  => 5           h(903) => 6   => 7  => 10
h(142) => 12          h(388) => 11  => 12 => 2  => 7 => 1
```




Uzavretá hašovacia tabuľka, známa aj ako hašovacia tabuľka „otvorené adresovanie“ alebo „uzavreté adresovanie“, funguje tak, že položky ukladá priamo do poľa hašovacej tabuľky a nie do prepojených zoznamov. Keď je položka vložená do hašovacej tabuľky, hašovacia funkcia vypočíta index v poli, kde by mala byť položka uložená. Ak je tento index už obsadený, hašovacia tabuľka použije „sondovaciu“ sekvenciu na vyhľadanie ďalšieho dostupného indexu v poli.

### 6.2. Realisation of put/get/remove

```java
@Override
public V get(K key)
{
    int index = hash(key);
    HashItemClosedHashing<K, V> item = table[index];

    if(item == null)
        return null;

    if(item.getKey().equals(key))
        return item.getValue();

    for(int i = index; i < table.length; i++)
    {
        if(table[i] == null)
            continue;

        if(table[i].getKey().equals(key))
            return table[i].getValue();
    }

    for(int i = 0; i < index; i++)
    {
        if(table[i] == null)
            continue;

        if(table[i].getKey().equals(key))
            return table[i].getValue();
    }

    return null;
}
```

Táto metóda berie kľúč ako vstup a vracia zodpovedajúcu hodnotu z hašovacej tabuľky. Najprv vypočíta hash index pre kľúč pomocou metódy hash() a potom skontroluje, či položka v tomto indexe má rovnaký kľúč ako vstupný kľúč. Ak áno, vráti hodnotu danej položky. Ak nie, vyhľadá v zostávajúcich pozíciách v hašovacej tabuľke položku so vstupným kľúčom a v prípade potreby sa zabalí na začiatok tabuľky. Ak nájde položku pomocou vstupného kľúča, vráti jej hodnotu. Ak prehľadá celú tabuľku a nenájde položku so vstupným kľúčom, vráti hodnotu null.

```
@Override
public void remove(K key)
{
    int index = hash(key);
    HashItemClosedHashing<K, V> item = table[index];

    if(item == null)
        return;

    if(item.getKey().equals(key))
    {
        table[index] = null;
        count--;
        return;
    }

    for(int i = index; i < table.length; i++)
    {
        if(table[i] == null)
            continue;

        if(table[i].getKey().equals(key))
        {
            table[i] = null;
            count--;
            return;
        }
    }

    for(int i = 0; i < index; i++)
    {
        if(table[i] == null)
            continue;

        if(table[i].getKey().equals(key))
        {
            table[i] = null;
            count--;
            return;
        }
    }
}
```

Táto metóda odstráni položku s daným kľúčom z hašovacej tabuľky. Funguje to podobne ako metóda get(), najprv vypočíta hašovací index pre kľúč a potom skontroluje, či má položka v tomto indexe rovnaký kľúč ako vstupný kľúč. Ak sa tak stane, odstráni túto položku z tabuľky nastavením príslušného slotu na hodnotu null, zníži počet položiek v tabuľke a vráti späť. Ak nie, vyhľadá v zostávajúcich pozíciách v hašovacej tabuľke položku so vstupným kľúčom a v prípade potreby sa zabalí na začiatok tabuľky. Ak nájde položku pomocou vstupného kľúča, odstráni ju z tabuľky, zníži počet položiek a vráti späť. Ak prehľadá celú tabuľku a nenájde položku so vstupným kľúčom, neurobí nič a jednoducho sa vráti.

```
@Override
public void put(K key, V value)
{
    if(count == (size/2))
    {
        size *= 2;
        HashItemClosedHashing<K, V>[] newHashTable = new
HashItemClosedHashing[size];

        for(HashItemClosedHashing<K, V> item : table)
        {
            if(item == null)
                continue;

            int index = hash(item.getKey());
            HashItemClosedHashing<K, V> newItem = item;

            if (newHashTable[index] == null)
                newHashTable[index] = new HashItemClosedHashing<>(item.getKey(),
item.getValue());
            else
            {
                boolean flag = false;
                for(int i = index; i < newHashTable.length; i++)
                {
                    if(newHashTable[i] == null)
                    {
                        flag = true;
                        newHashTable[i] = new
HashItemClosedHashing<>(item.getKey(), item.getValue());
                        break;
                    }
                }
                if(!flag)
                {
                    for(int i = 0; i < index; i++)
                    {
                        if(newHashTable[i] == null)
                        {
                            newHashTable[i] = new
HashItemClosedHashing<>(item.getKey(), item.getValue());
                            break;
                        }
                    }
                }
            }
        }
        table = newHashTable;
    }

    int index = hash(key);

    if(table[index] == null)
        table[index] = new HashItemClosedHashing<>(key, value);
    else
    {
        boolean flag = false;
        for(int i = index; i < table.length; i++)
        {
            if(table[i] == null)
            {
                flag = true;
                table[i] = new HashItemClosedHashing<>(key, value);
                break;
```

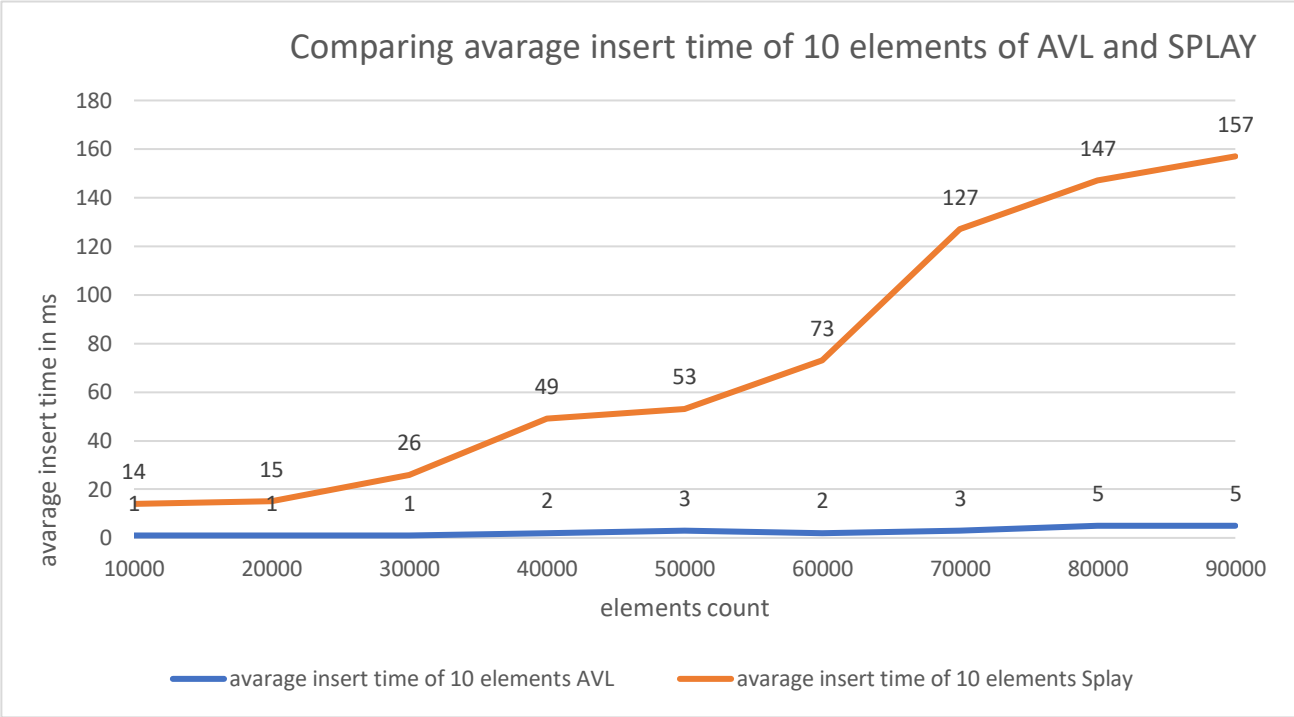
```
                }
            }
            if(!flag)
            {
                for(int i = 0; i < index; i++)
                {
                    if(table[i] == null)
                    {
                        table[i] = new HashItemClosedHashing<>(key, value);
                        break;
                    }
                }
            }
        }
        count++;
}
```

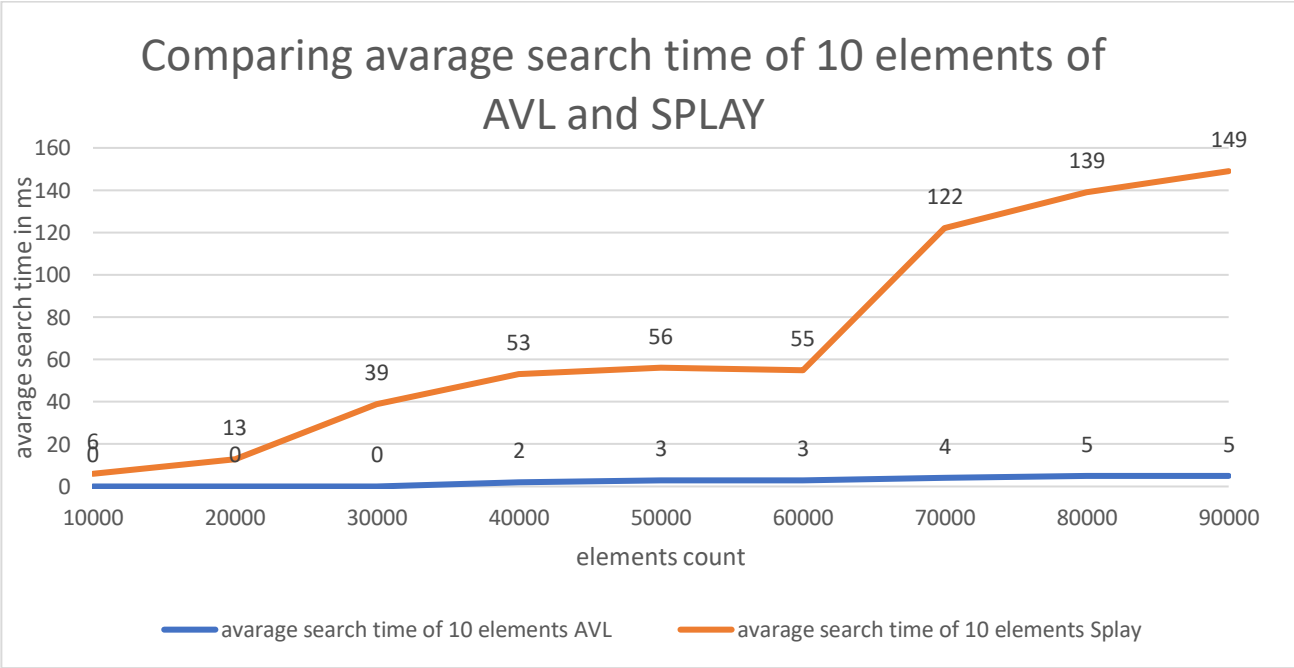
Táto metóda pridá do hašovacej tabuľky pár key –value. Ak je stôl už z polovice plný, zdvojnásobí sa veľkosť stola a všetky položky sa prepracujú. Najprv vypočíta hash index pre kľúč pomocou metódy hash(). Ak je slot v tomto indexe prázdny, jednoducho pridá nový pár kľúč – hodnota do tohto slotu. Ak je slot už obsadený, vyhľadá v zostávajúcich slotoch v tabuľke prázdny slot, aby vložil pár kľúč – hodnota, a v prípade potreby sa zabalí na začiatok tabuľky. Ak nájde prázdny slot, vloží do neho nový pár kľúč – hodnota. Ak prehľadá celú tabuľku a nenájde prázdny slot, zdvojnásobí veľkosť tabuľky a pred vložením nového páru kľúč – hodnota prehodnotí všetky položky.

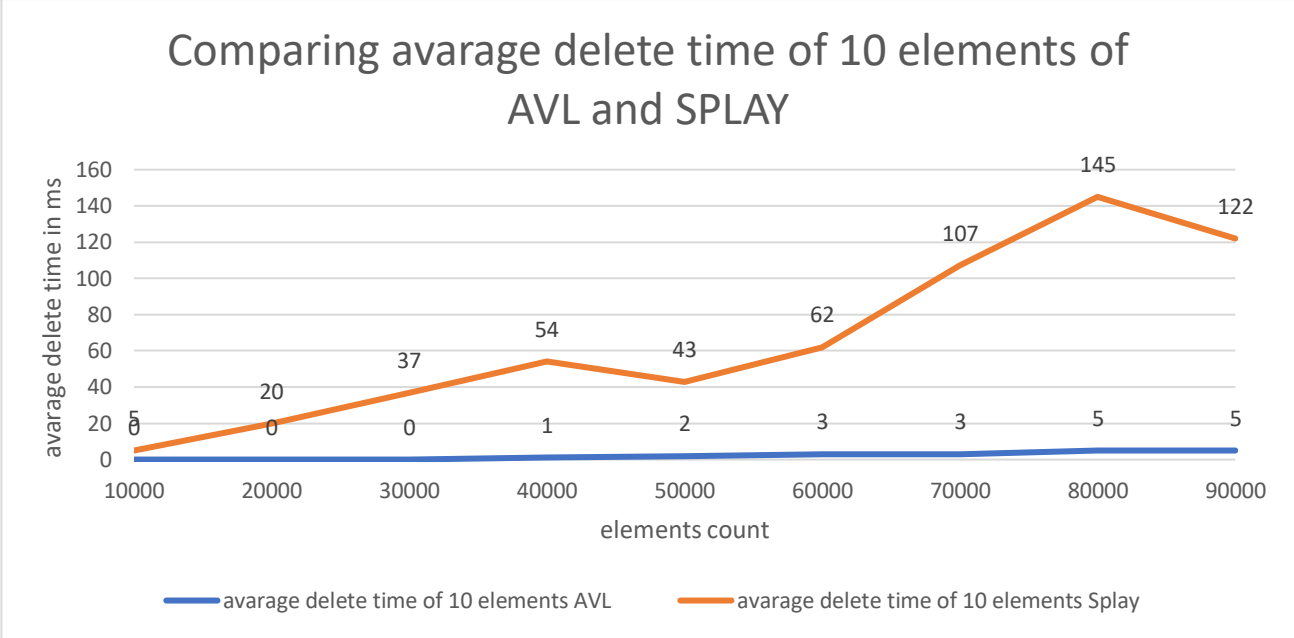
# Testovanie

## 7. Vyhľadávacie stromy

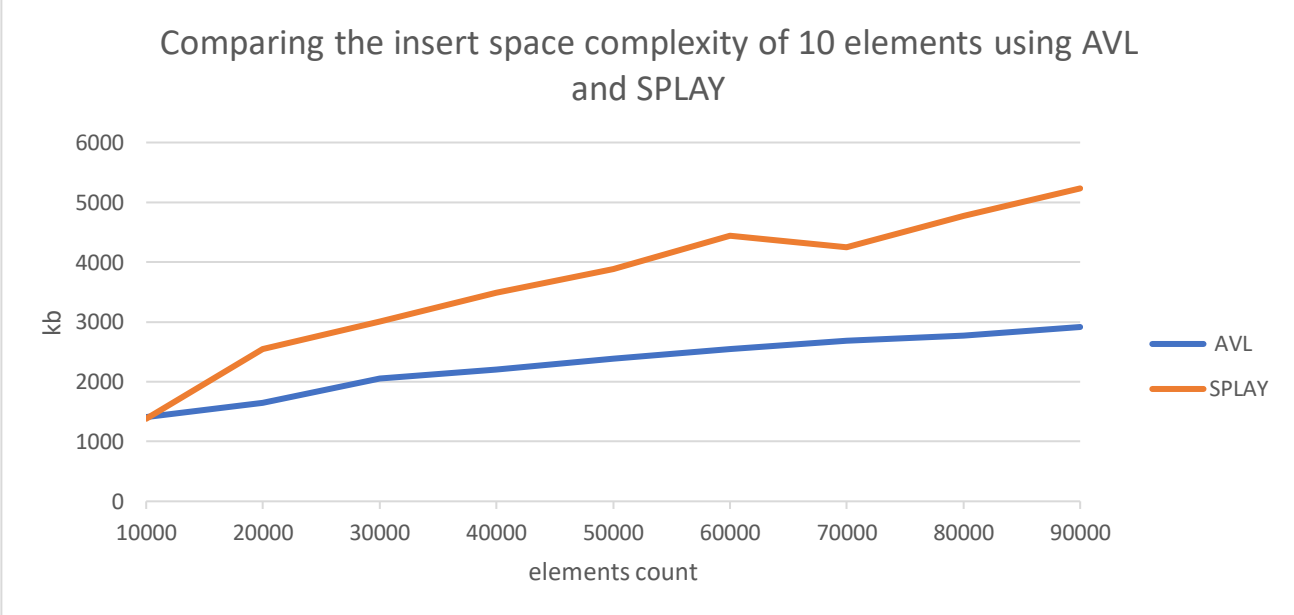
Programy testujem tak, že najskôr dám 10 000 prvkov do binárnych stromov a zakaždým sa táto hodnota zvýši o 10 000, kým nepríde 90 000. Program spočíta, vloží, odstráni 10 prvkov z 10 000 a viac pomocou randomizéra. . Vo svojich testoch kontrolujem rýchlosť mojich troch funkcií a koľko pamäte zaberajú.

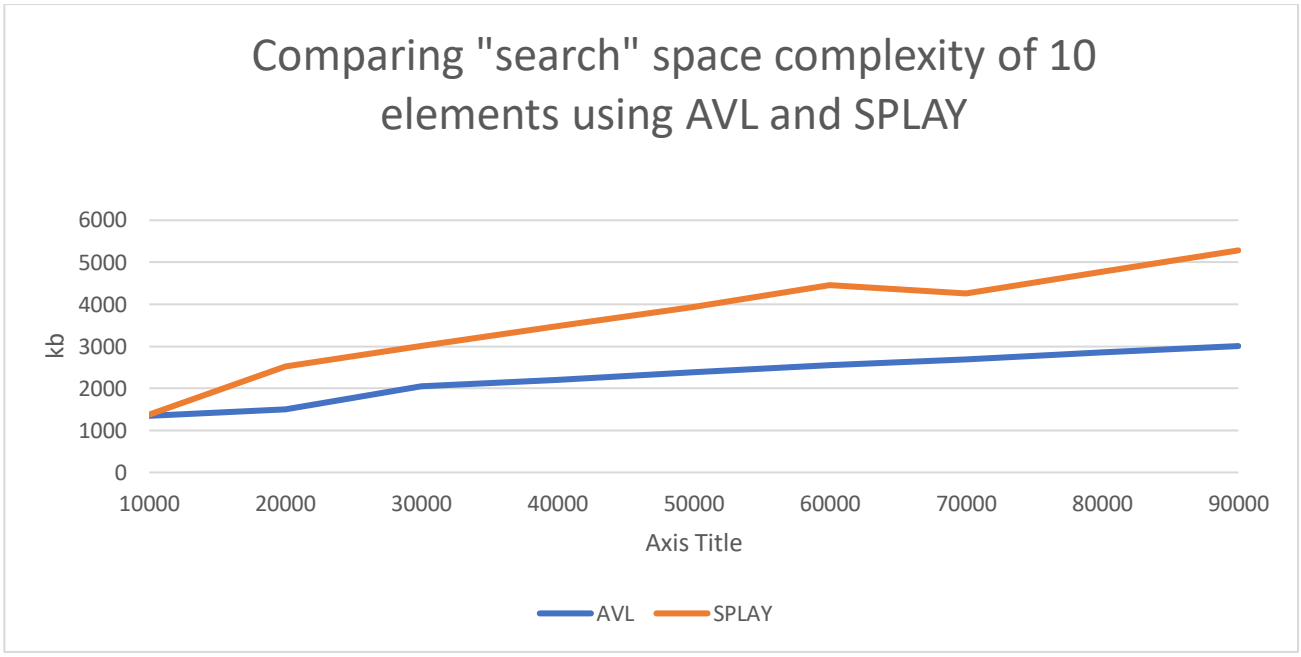


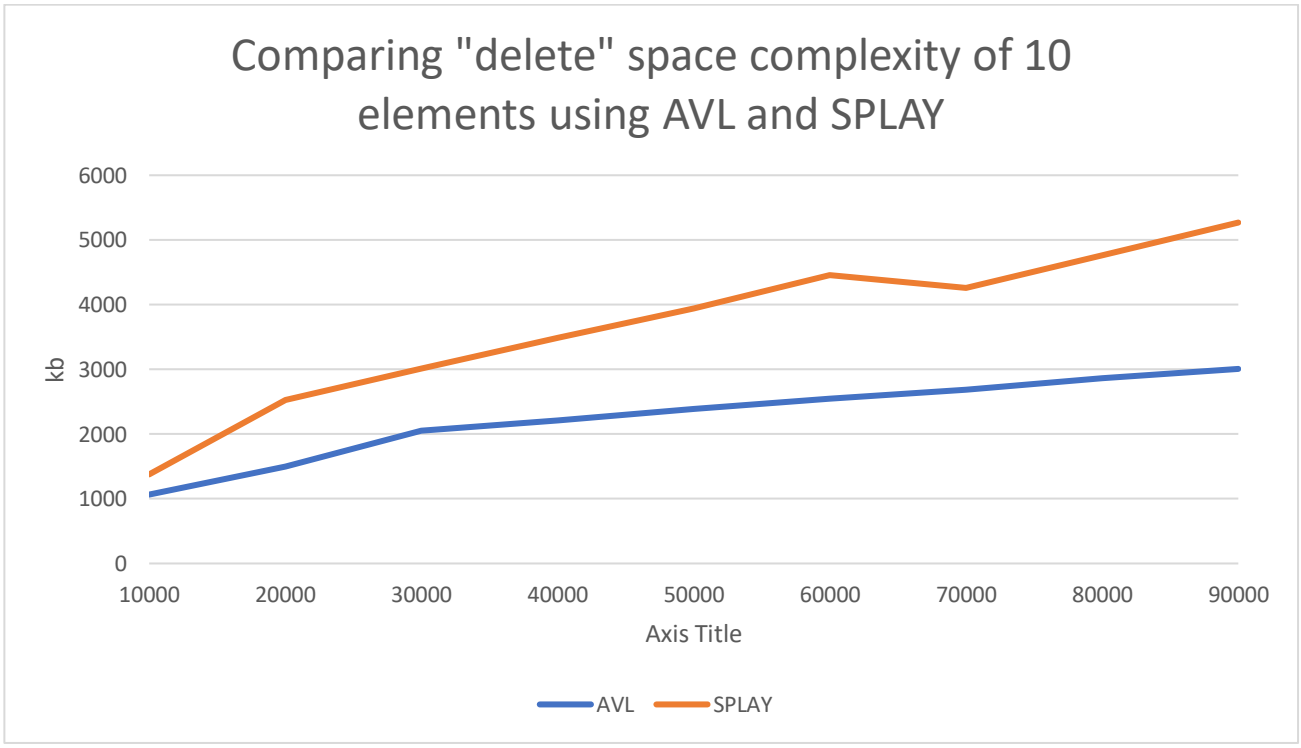






## Space Complexity:



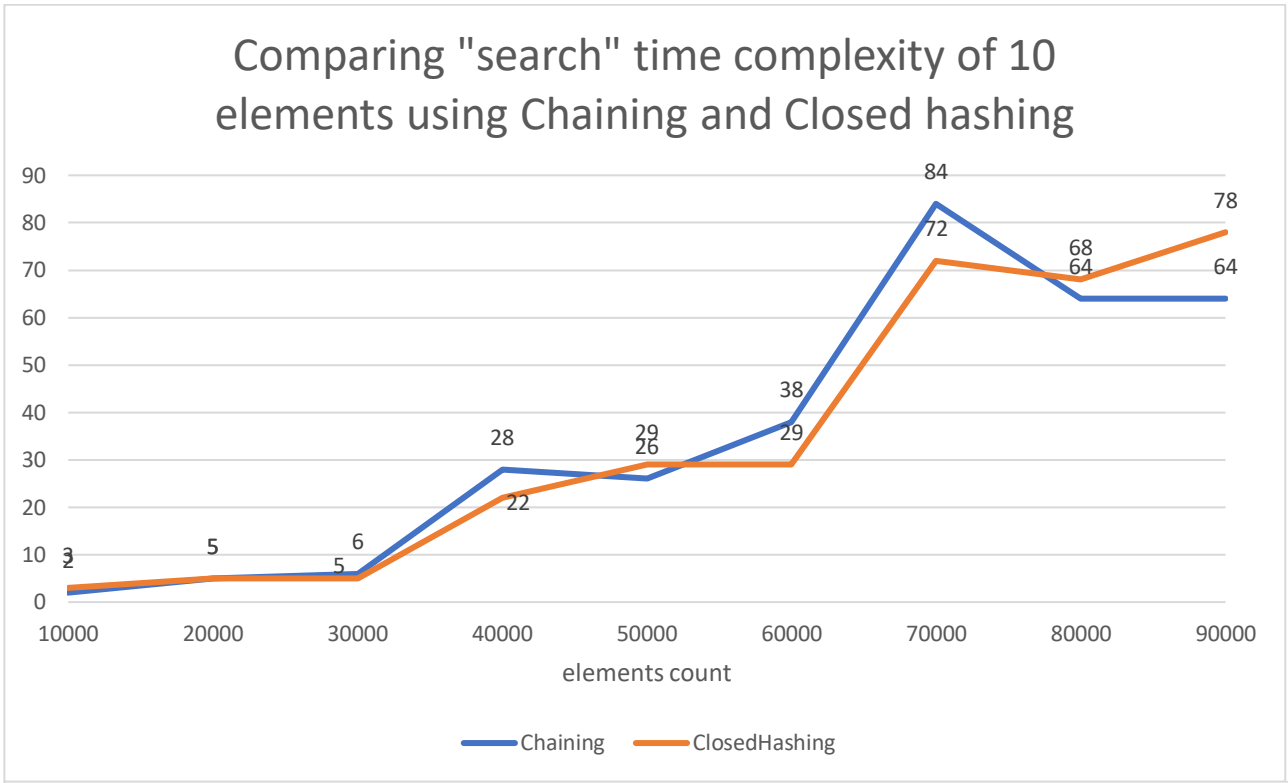


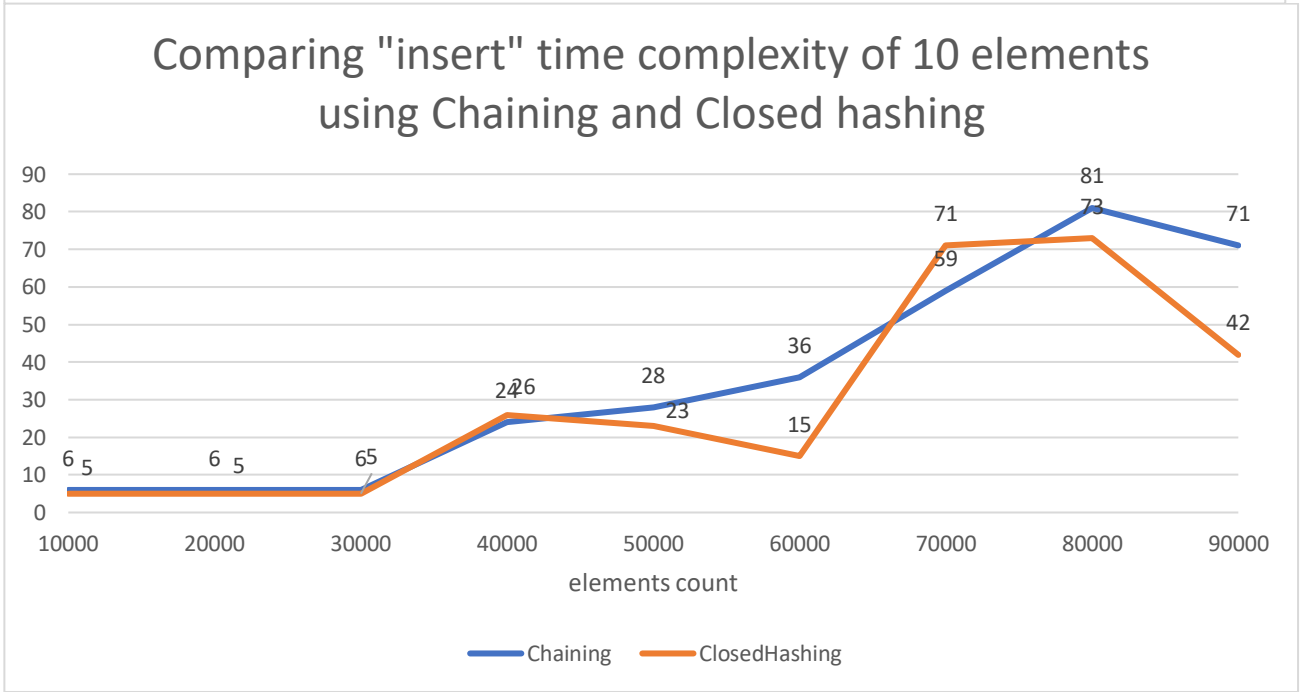



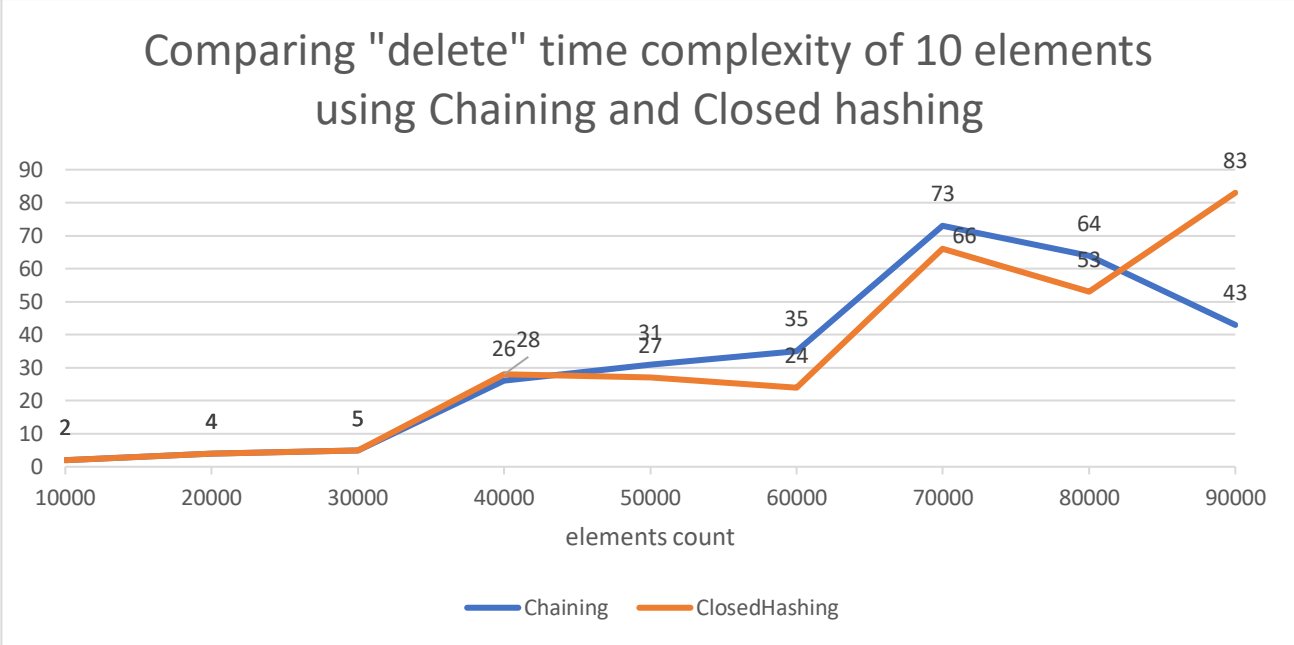

Pri mojej implementácii týchto binárnych stromov sa ukazuje, že binárny strom AVL predbieha SPLAY nielen v rýchlosti, ale aj v načítavaní pamäte

# 8. Hashovanie

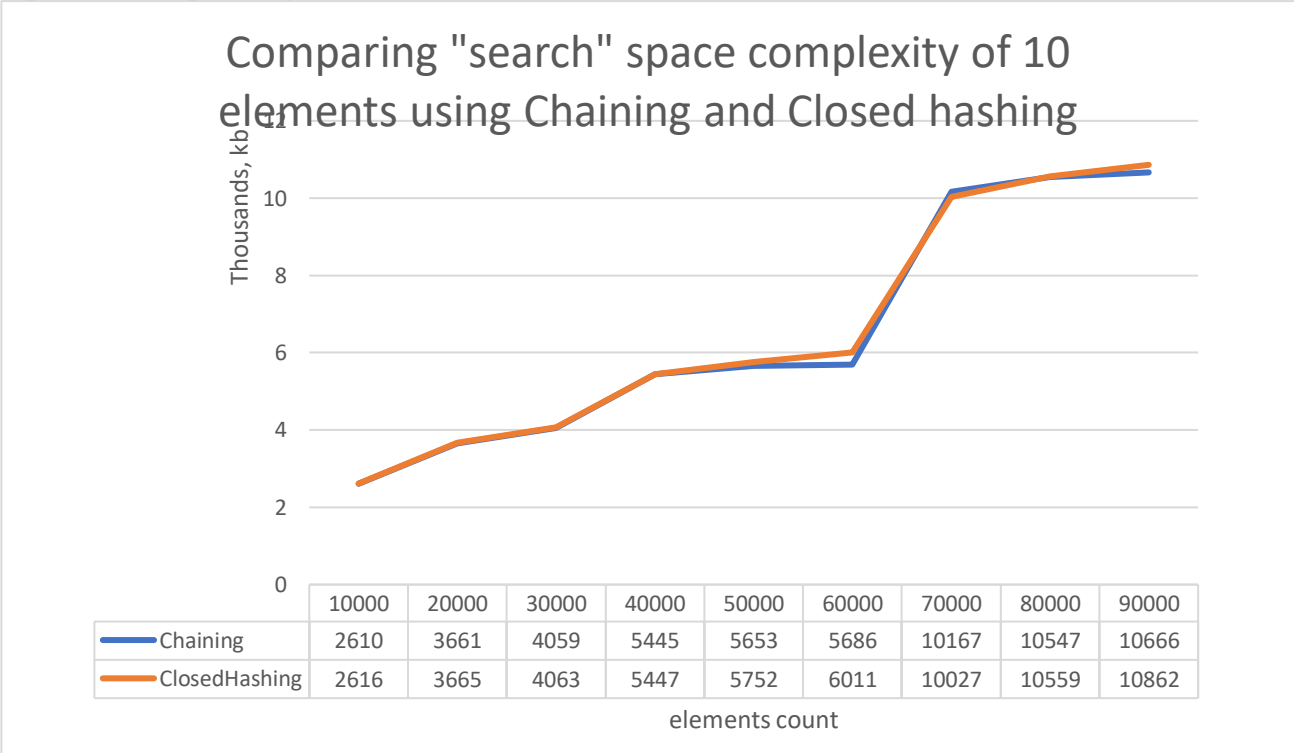
Programy testujem tak, že najskôr dám 10 000 prvkov do hashovych tabuliek a zakaždým sa táto hodnota zvýši o 10 000, kým nepríde 90 000. Program spočíta, vloží, odstráni 10 prvkov z 10 000 a viac pomocou randomizéra. . Vo svojich testoch kontrolujem rýchlosť mojich troch funkcií a koľko pamäte zaberajú.

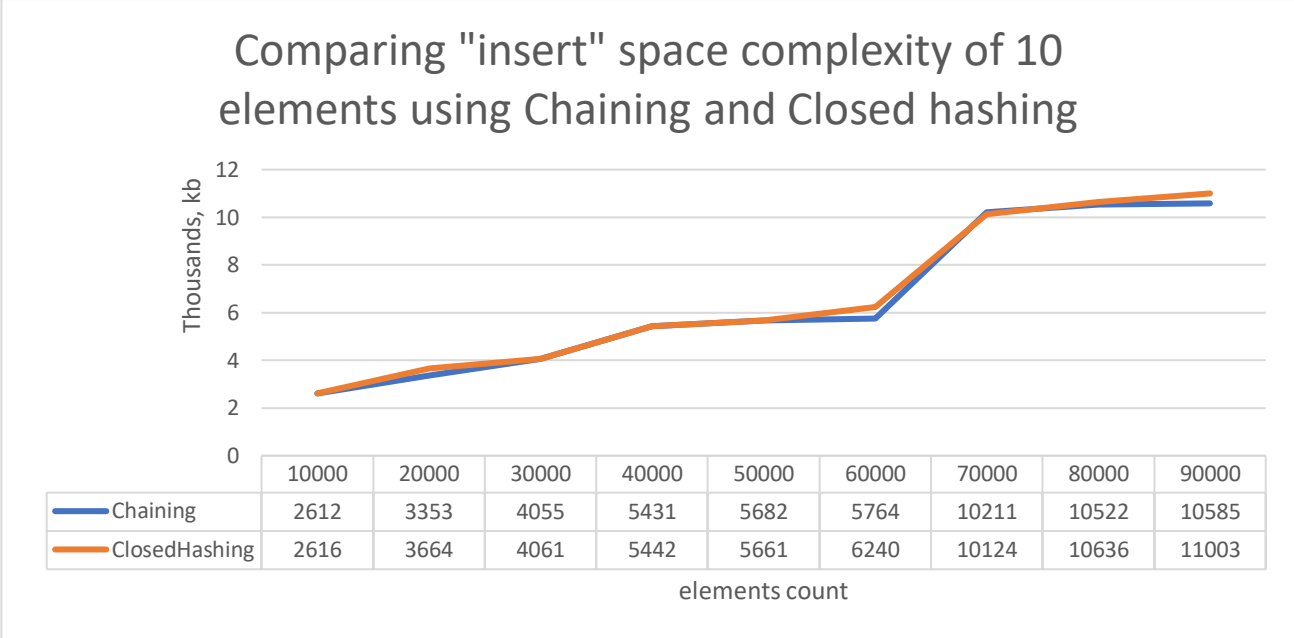


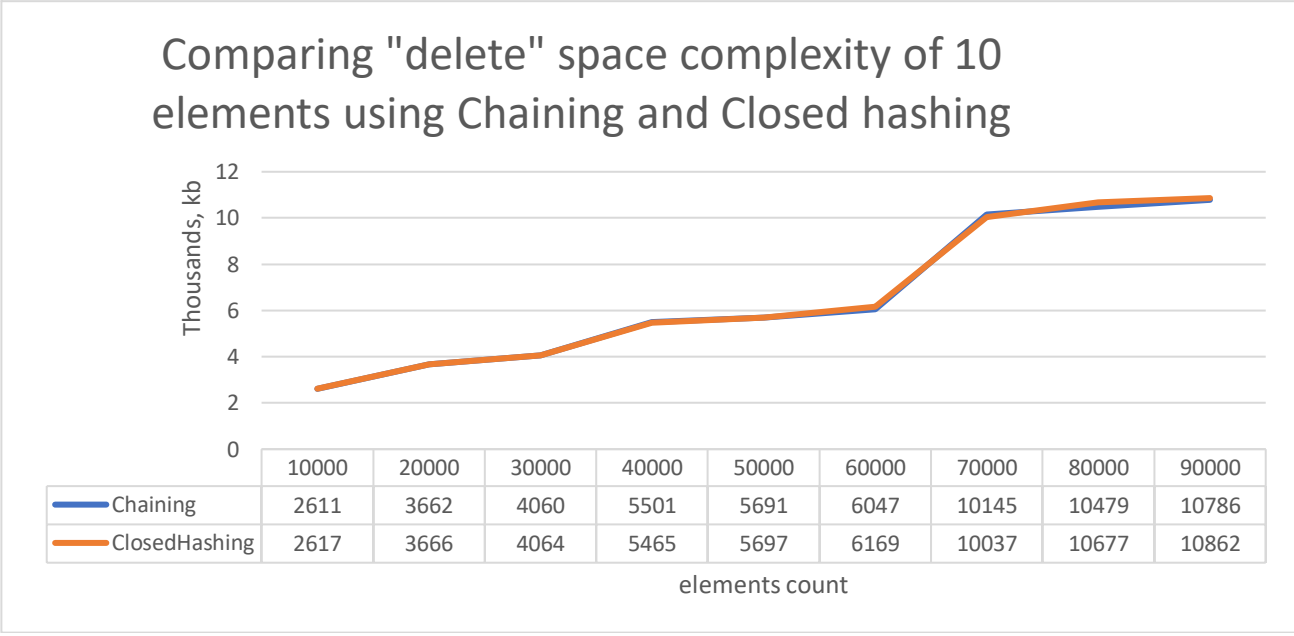



Space complexity:






| | 10000 | 20000 | 30000 | 40000 | 50000 | 60000 | 70000 | 80000 | 90000 |
|---|---|---|---|---|---|---|---|---|---|
| Chaining | 2612 | 3353 | 4055 | 5431 | 5682 | 5764 | 10211 | 10522 | 10585 |
| ClosedHashing | 2616 | 3664 | 4061 | 5442 | 5661 | 6240 | 10124 | 10636 | 11003 |




| | 10000 | 20000 | 30000 | 40000 | 50000 | 60000 | 70000 | 80000 | 90000 |
|---|---|---|---|---|---|---|---|---|---|
| Chaining | 2611 | 3662 | 4060 | 5501 | 5691 | 6047 | 10145 | 10479 | 10786 |
| ClosedHashing | 2617 | 3666 | 4064 | 5465 | 5697 | 6169 | 10037 | 10677 | 10862 |

Voľba medzi Chaining a Closed Hashing závisí od rôznych faktorov, ako je očakávaný počet prvkov v tabuľke, distribúcia kľúčov, náklady na výpočet hašovacej funkcie a náklady na mechanizmus riešenia kolízií. Vo všeobecnosti je Closed Hashing rýchlejšy ako Chaining, keď je faktor zaťaženia (pomer počtu prvkov k veľkosti tabuľky) nízky a hašovacia funkcia je efektívna. Je to preto, že Closed Hashing ukladá prvky priamo do poľa, čím sa vyhýba réžii pri prideľovaní prepojených zoznamov a ukazovateľov. Okrem toho je výkon vyrovnávacej pamäte zvyčajne lepší v Closed Hashing, pretože prvky sú uložené súvisle v pamäti. Keď je však faktor zaťaženia vysoký, Closed Hashingmôže trpieť vysokým počtom kolízií, čo vedie k zvýšeniu nákladov na sondovanie a opätovné hašovanie. V takýchto prípadoch môže byť Chaining efektívnejšie, pretože umožňuje uloženie viacerých prvkov do rovnakého slotu, čím sa znižujú náklady na riešenie kolízií.