Dokumentácia – Zadanie 1

Vypracoval: Filip Paučo

AIS ID: 116270

Obsah

1. AVL strom

Je samovyvažovací vyhľadávací binárny strom. Každá položka uchováva svoju hodnotu (číslo) a ukazovateľ na ľavú a pravú položku pod ňou. Taktiež, položka je špecifikovaná aj číselnou hodnotou výšky, teda cesta k poslednému prvku podstromu +1.

Podstatná funkcia pre správne fungovanie tejto štruktúry je vyváženosť (balance faktor), jeho hodnotu vypočítame ako rozdiel výšky ľavého a pravého podstromu danej položky. Prípustné sú tri hodnoty (-1;0;1). Pri zápornom čísle menšom ako -1 je strom zaťažený na pravej strane, naopak pri kladom čísle vyššom ako 1 je strom zaťažený na ľavej strane. V oboch prípadoch sú potrebné príslušné rotácie.

1.1 Insert

a) Určenie miesta

```
public Node insert(Node node, int data){
    int value;
    if(node != null){
        if(node.data > data){
            node.left = insert(node.left, data);
        }
        else if(node.data < data){
            node.right = insert(node.right, data);
        }
        else if(node.data == data){
            return node;
        }
```

Najskôr je potrebné nájsť miesto, kam sa má daný údaj vložiť. V mojom prípade som využil rekurziu. V prípade, už existujúcej položky, vrátime s funkcie node (čo je v tomto prípade root (prvý prvok) a teda nič sa nevykoná. V prípade novej položky sa vytvorí nová nasledovným príkazom:

b) Výška stromu a balancovanie

```
node.height = 1 + max(height(node.left), height(node.right));
value = balance(node);

if(value > 1 && data > node.left.data){
    node.left = lR(node.left);
    return rR(node);
}
if(value > 1 && data < node.left.data){
    return rR(node);
}
if(value < -1 && data < node.right.data){
    node.right = rR(node.right);
    return lR(node);
}
if(value < -1 && data > node.right.data){
    return lR(node);
}
return node;
```

Výšku stromu zistíme funkciou max, ktorá nám vráti väčšie číslo spomedzi výšky ľavého a pravého podstromu. Následne funkcia balance odpočíta tieto podstromy (ľavý - pravý). Na základe tohto výsledku sa vykonávajú rotácie.
V prípade, že je strom vyvážený podľa požiadaviek, ktoré som už vyššie uviedol, nie je potrebné vykonať rotácie, a vraciame root hodnotu, pre prípadné ďalšie vkladanie. V opačnej situácií, vždy porovnávame 3 prvky. Môžu nastať 4 situácie (2 pre každý podstrom).

a)

b)

a) Vykonávame pravú rotáciu, tak, že číslo 4 bude prvý prvok. Prvok 6 bude napravo a 2 naľavo.
b) Zložitejší prípad, vykonávame 2 rotácie, najskôr ľavú pri prvku 4. Potom pravú pri prvku 6. Vo výsledku bude na vrchu prvok číslo 5. Prvok 6 bude napravo a 4 naľavo.

Tieto rotácie fungujú analogicky aj pre pravý podstrom.

1.2 Search

a) Porovnávanie prvkov

```
void search(Node node,int a){
    if(node == null){
        System.out.println("Cislo "+a+" sa v strome nenachadza");
        return;
    }
    if(node.data == a){
        System.out.println("Toto cislo sa nachadza v strome");
        }
    if(node.data > a){
        search(node.left, a);
    }
    if(node.data < a){
        search(node.right, a);
    }
}
```

Vyhľadávame je veľmi jednoduché. Stojí na porovnávaní prvkov a následným posúvaním sa na príslušnom podstrome. Pri lineárnom vyhľadávaní, je zložitosť algoritmu O(n). Zložitosť vyhľadávania pri binárnom vyváženom strome je O(log n). A teda je to omnoho efektívnejšie, špeciálne pri vysokých číslach.

1.3 Delete

a) Vyhľadanie prvku a kontrola potomkov

```
public Node deleteNode(Node node, int data){
    Node temp = null;

    if(node != null){
        if(node.data > data){
            node.left = deleteNode(node.left,data);
        }
        else if(node.data < data){
            node.right = deleteNode(node.right,data);
        }
        else{
            if((node.left == null) || (node.right == null)){
                if(node.right != null){
                    temp = node.right;
                }
                else if(node.left != null){
                    temp = node.left;
```

Vymazávanie prvkov, vykonávané taktiež pomocou rekurzie. V tejto operácií je podstatné miesto, kde sa prvok pre vymazanie nachádza. V závislosti od tohto miesta, vykonávame špecifické operácie. Najjednoduchší je prípad spodného prvku (listu), naopak zložitejšie je to pri vyšších prvkoch s pravým aj ľavým listom. Vždy je potrebné skontrolovať vyváženie stromu. Na základe toho vykonať potrebné rotácie.

b) Pokračovanie kontroly potomkov

```
        }
        if(temp == null){
            temp = node;
            node = null;
        }
        else{
            node = temp;
        }
        temp = null;
    }
    else{
        temp = min(node.right);
        node.data = temp.data;
        node.right = deleteNode(node.right, temp.data);
    }
}
if(node == null){
    return node;
```

c) Kontrola vyrovnania

```
    node.height = 1+max(height(node.left),height(node.right));
    int value = balance(node);

    if(value > 1 && balance(node.left) >= 0){
        return rR(node);
    }
    if(value > 1 && balance(node.left) < 0){
        node.left = lR(node.left);
        return rR(node);
    }

    if(value < -1 && balance(node.right) >= 0){
        return lR(node);
    }

    if(value < -1 && balance(node.left) < 0){
        node.right = rR(node.right);
        return lR(node);
    }

    return node;
}
else{
    return node;
}
```

2. Červeno-čierny strom

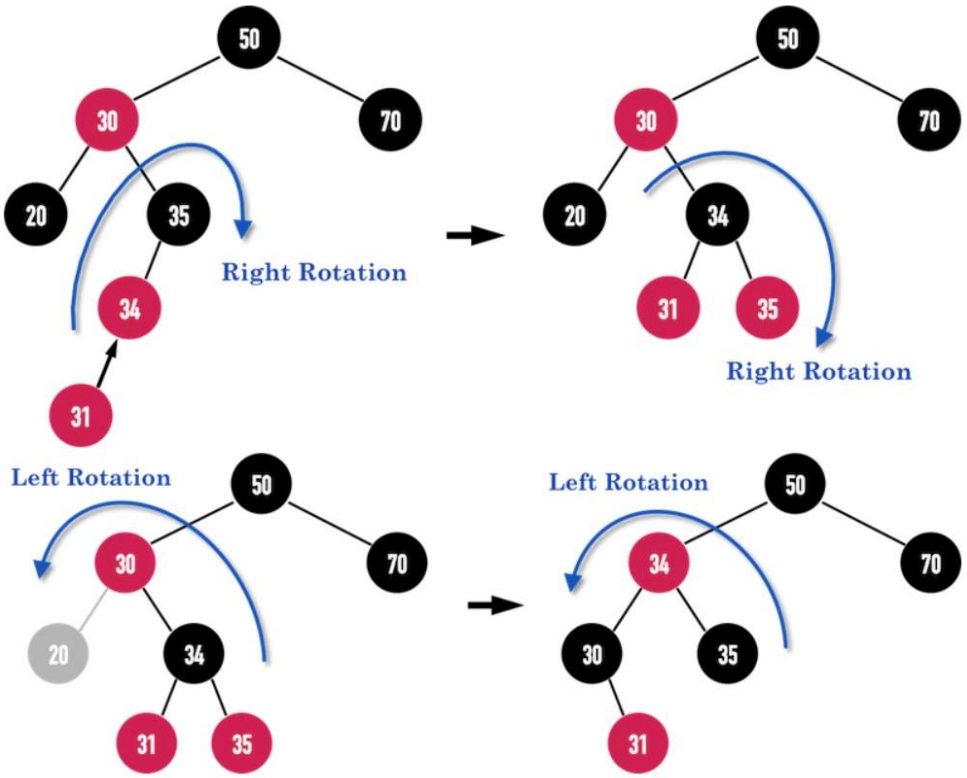
Ďalší typ samovyvažovacieho stromu, určité vlastnosti zdieľa s typom AVL. Avšak, je zložitejší na implementáciu a pri vyvažovaní sa využíva premenná naviac – farba. Červeno-čierny strom je výhodnejší pri častých vkladoch a vymazávaniach prvkov, v porovnaní so stromom AVL.

Základné pravidlá: **1. Každý prvok je čierny alebo červený**
2. Prvý prvok je vždy čierny
3. Žiadne dva červené prvky nie sú spojené
4. Každá cesta do nižšieho prvku má rovnako čiernych prvkov

2.1 Insert

Usporiadanie hodnôt prvkov a rotácie fungujú analogicky so stromom AVL. V tomto prípade je potrebné riešiť prefarbovanie pri vstupoch podľa pravidiel uvedených vyššie. Pre reprezentáciu prvkov sa používajú označenia súrodenec, rodič, strýko, starý rodič.

a) Rotácie so zmenou farby



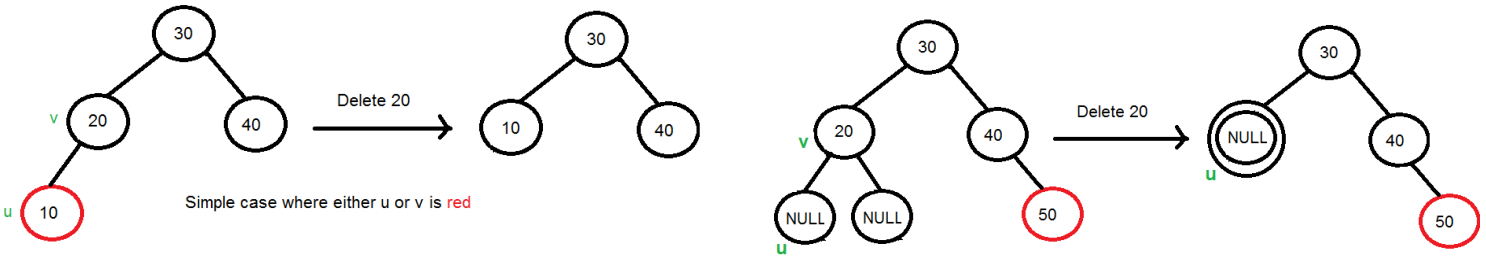
2.2 Search

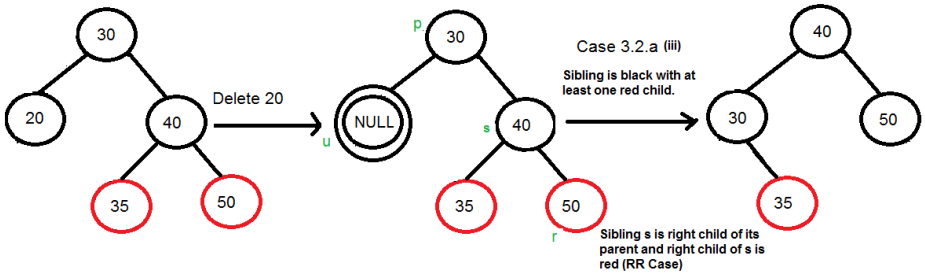
Vyhľadávanie prvkov funguje rovnako ako pri strome AVL, opísal som *ho* ***Search*** *(*pre zobrazenie časti, ctrl + klik na zvýraznený text*).*

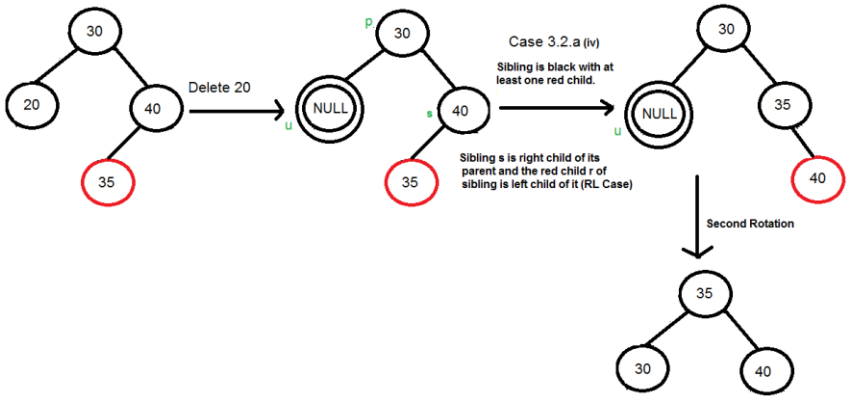
2.3 Delete

Pravdepodobne najťažšou implementáciou je operácia delete. Ako pri vkladaní, je potrebné dodržiavať pravidlá stromu. V tomto prípade je potrebné sledovať **prvok súrodenca**, a podľa toho zistiť, o ktorý prípad sa jedná. Pri odstraňovaní, sa najčastejšie narušuje **pravidlo č. 4,** výšky podľa čiernych prvkov. Tento problém vzniká pri odstránení čierneho prvku z vyrovnaného stromu.

Niekoľko príkladov na ukážku (4):







3. Hashovacie tabuľky

3.1 Základné informácie

Hashtabuľky slúžia na uchovávanie dát vo forme kľúča a hodnoty. Každý kľúč je unikátny, ale hodnoty sa môžu opakovať. Miesto indexov (ako v klasickom poli) sa pre určenie pozície prvku využíva kľúč. Kľúč je generovaný vzorcom: hashovacia funkcia % (modulo) veľkosť tabuľky. Príklad hashovacej metódy:

```
public int hashCode() {
    int hash= 1;
    int number = 11;
    char[] array = text.toCharArray();
    for(char c : array){
        hash = hash * number + c;
    }
    return Math.abs(hash);
}
```

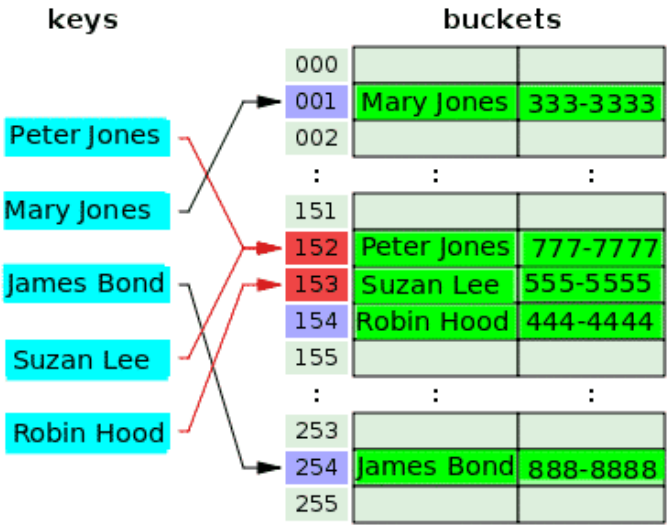
Text bolo potrebné prekonvertovať na pole písmen a počítať s ich číselnou hodnotou. Ďalej, som si zvolil prvočíslo 11. Po poslednom iterovaní v cykle, dostávame hodnotu, ktorá môže byť záporná, preto ju zmeníme pomocou absolútnej hodnoty na kladnú. Následne použijeme tento vzorec: hashovacia funkcia % (modulo) veľkosť tabuľky. Avšak, je zrejmé, že musí prichádzať aj ku kolíziám, napríklad pri zhode vstupného textu, kedy by sa na jednom mieste prepisovali dáta. Tak by sme stratili údaje z pôvodného vstupu. Dané situácie sa riešia dvoma spôsobmi:

a) **reťazenie prvkov na danej pozícií.**
b) **vyhľadanie najbližšieho voľného miesta v tabuľke s využitím mocniny.**

4. Hashovacia tabuľka s reťazením

4.1 Insert

Najskôr je potrebné vypočítať hodnotu indexu pomocou už niekoľkokrát spomínaného vzorca. Pokiaľ sa v danom priestore nič nenachádza, môžeme tam daný prvok vložiť. V opačnom prípade, na danom mieste vytvoríme spájaný zoznam, počnúc prvým prvkom, ktorý tam bol vložený.



4.2 Search

Vyhľadávanie prebieha v dvoch krokoch. A to vo vyhľadaní hlavičky podľa hashovacieho kódu, a následným vyhľadaním kľúča prechádzaním prvkov až po prvok NULL. V prípade úspešného vyhľadania, návratovou hodnotou je hodnota prvku.

4.3 Delete

Opäť začneme rovnakým postupom ako pri vyhľadávaní. Teda nájdeme hlavičku (prvý index tabuľky), kde by sa mal prvok nachádzať. Pokiaľ ho nenájdeme, z metódy sa vrátime prostredníctvom returnu. Inak prechádzame jednotlivé spojené prvky. Pokiaľ nájdeme požadovaný prvok, spojenie z predošlého prvku posunieme na ďalší prvok (na ktorý ukazuje prvok na vymazanie).

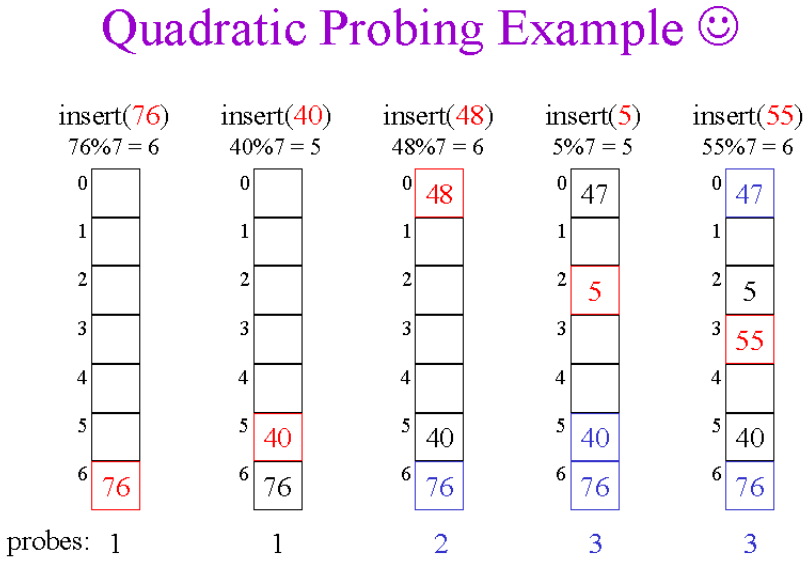
5. Hashovacia tabuľka riešená kvadraticky

5.1 Insert

Tento typ je podobný tabuľke riešenej lineárne. Výpočet hash hodnoty funguje zhodne ako v prípade hashovacej tabuľky riešenej zreťazením. Rozdiel je vo vzorci na výpočet indexu. Index sa vypočíta pomocou vzorca nasledujúceho vzorca:

```
int j = data.hashCode() % table.length;
int k = 1;
while(table[j] != null){
    j = (data.hashCode() + (k*k)) % table.length;
    k++;
```

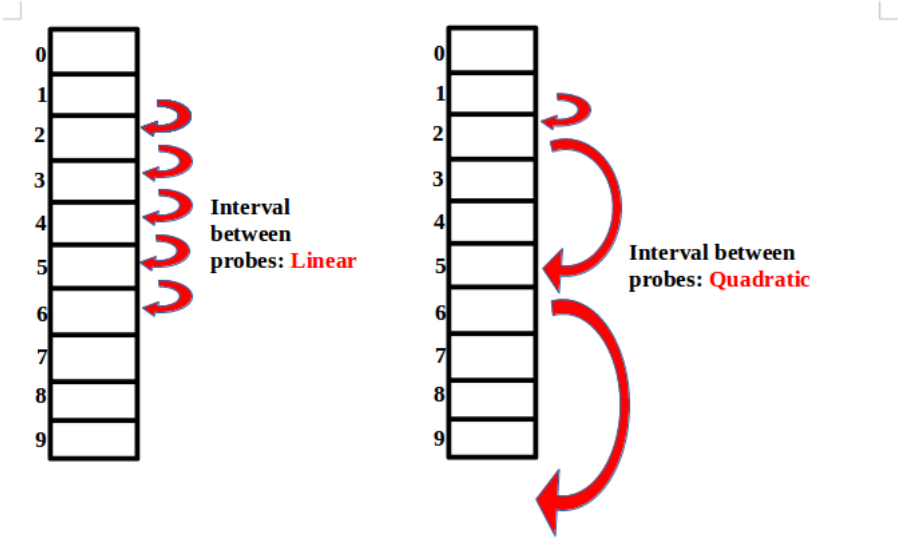
Ako je na obrázku vyššie znázornené hľadáme voľný index kvadraticky. Je to výhodné pri situáciách (kolíziách), kedy je veľa konzekutívnych indexov zaplnených. Pričítanie mocniny zabezpečí väčšie skoky pre indexy v tabuľke. Týmto spôsobom vložíme dané vstupy na príslušné pozície. Je vhodné, aby tabuľka mala veľkosť prvočísla, nakoľko môže nastať situácia množstva rovnakých vstupov, čo pri module vyčerpá priestor tabuľky.



5.2 Search

Po vypočítaní hodnoty indexu, v spomínaným vzorcom pri vkladaní, môžu nastať 3 prípady:

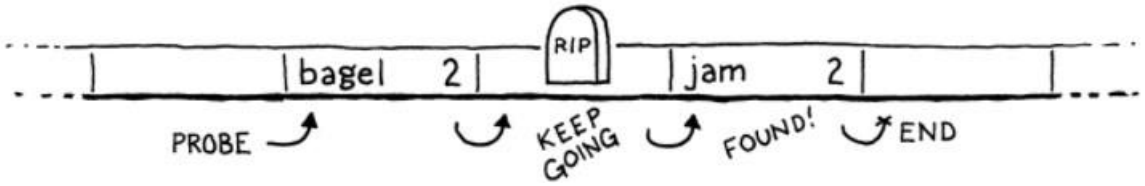
1. Pokiaľ sa na danom mieste tabuľky nič nenachádza, teda je nastavená na null – prvok sme nenašli.
2. Prípad, kedy je práve hľadaná hodnota na danom indexe.
3. V prípade kolízie, kedy na hľadanom mieste je iná položka, než je hľadaná. Vtedy prechádzame tabuľku pomocou pričítaní mocniny, až pokým sa späť nevrátime na daný index.



5.3 Delete

V operácií vymazania je opäť potrebné nájsť hodnotu prvku, a to tak, že najskôr sa pozrieme na prvotné miesto indexu, kde by sa mal prvok nachádzať. Ak na tomto mieste prvok nenájdeme, prehľadávame postupne pričítaním mocniny celú tabuľku.

Po nájdení prvku, prvok zmažeme a nastavíme špecifickú hodnotu kľúča tohto políčka. Toto označenie nám zabezpečí, že pri ďalších operáciách v tabuľke, vieme preskočiť toto označené pole a tak ušetriť čas. Toto miesto sa označuje ako tzv. Tombstone (náhrobný kameň).


RIP
bagel 2
jam 2
PROBE
KEEP GOING
FOUND!
END

6. Testovanie

6.1 Porovnanie stromov

Testovanie stromov prebehlo na princípe generovania náhodných čísel v odlišných intervaloch. Tieto operácie sú merané časom, ktorý je potrebný na vykonanie danej funkcie. Všetky operácie medzi stromami sú následne porovnané v grafoch nižšie.

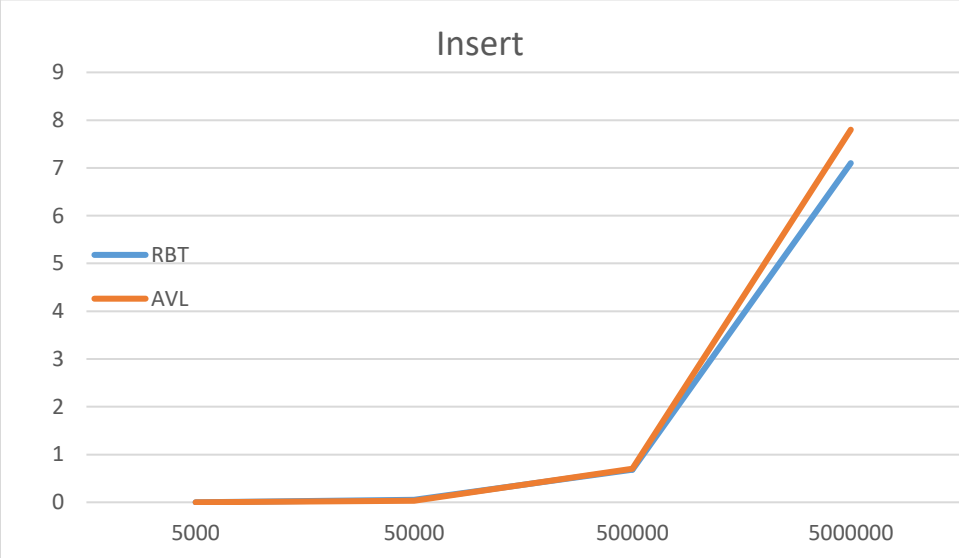
Čas sa meral pri štyroch rôznych veľkostiach:

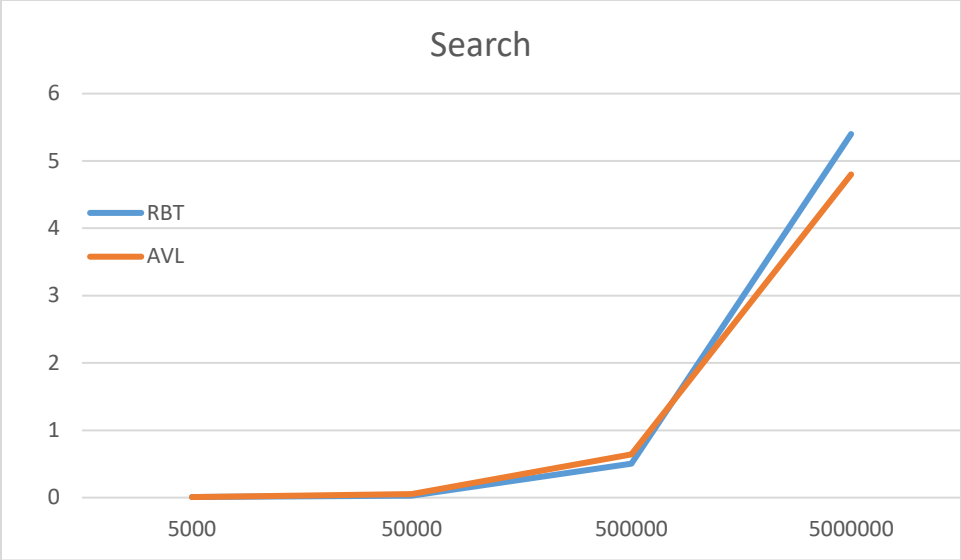
a) Pridávanie a vyhľadávanie pri 5000 prvkoch

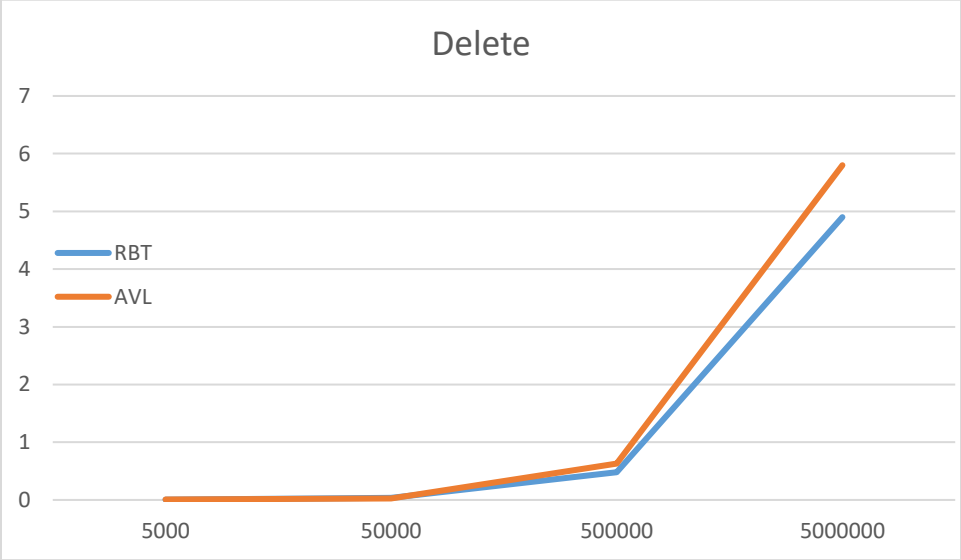
b) Pridávanie a vyhľadávanie pri 50 000 prvkoch

c) Pridávanie a vyhľadávanie pri 500 000 prvkoch

d) Pridávanie a vyhľadávanie pri 5 000 000 prvkoch







Z porovnania vyplýva, že stromy AVL sú vhodnejšie pre vyhľadávania. Avšak v prípade, kedy nastáva množstvo rotácií (vkladanie a vymazávanie) je efektívnejší červeno-čierny strom. V oboch prípadoch platí zložitosť O(log n).

6.2 Porovnanie tabuliek

Testovanie tabuliek prebehlo na princípe generovania náhodných reťazcov dĺžky pätnástich znakov. Každá operácia je meraná časom, ktorý je potrebný na vykonanie danej funkcie. Všetky operácie medzi tabuľkami sú následne porovnané v grafoch nižšie.

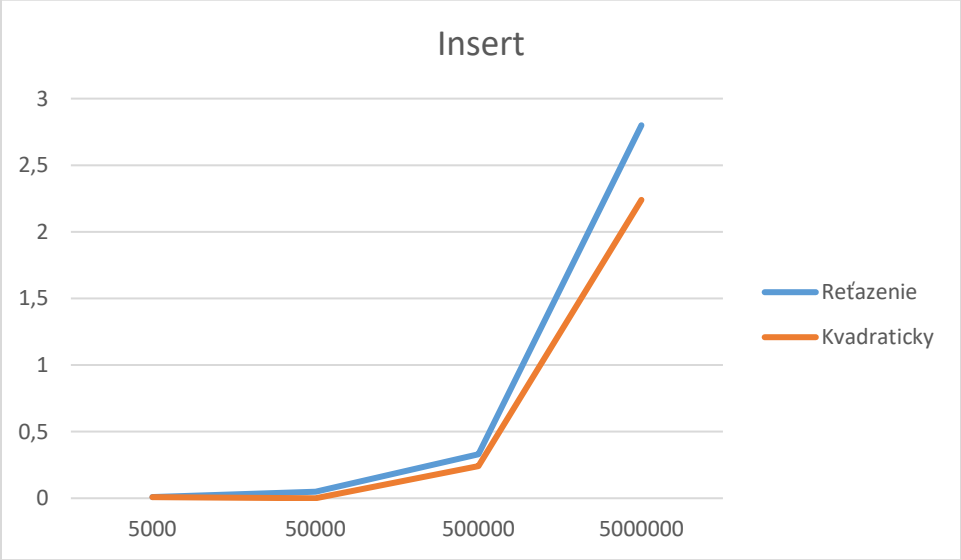
Čas sa meral pri štyroch rôznych veľkostiach pri dĺžke pätnástich znakov:

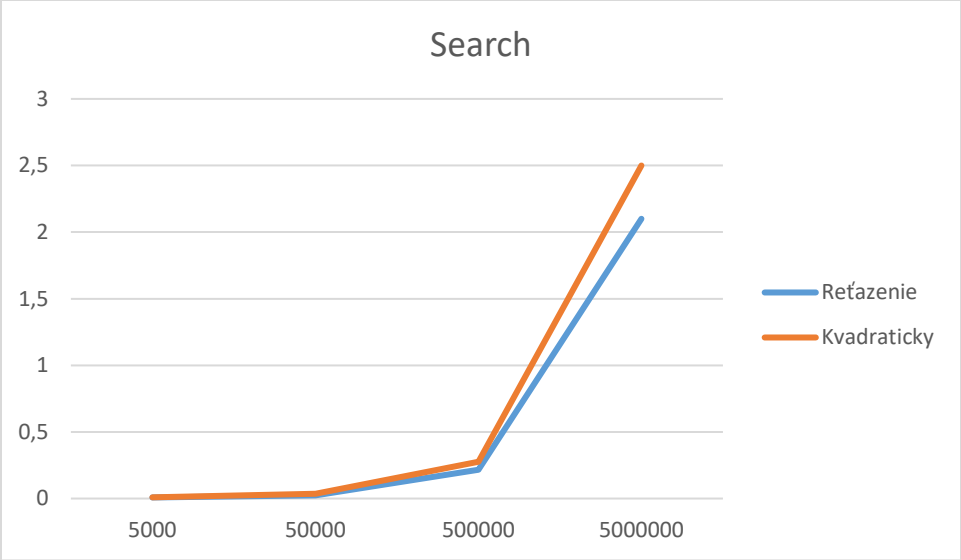
a) Pridávanie a vyhľadávanie pri 5000 prvkoch

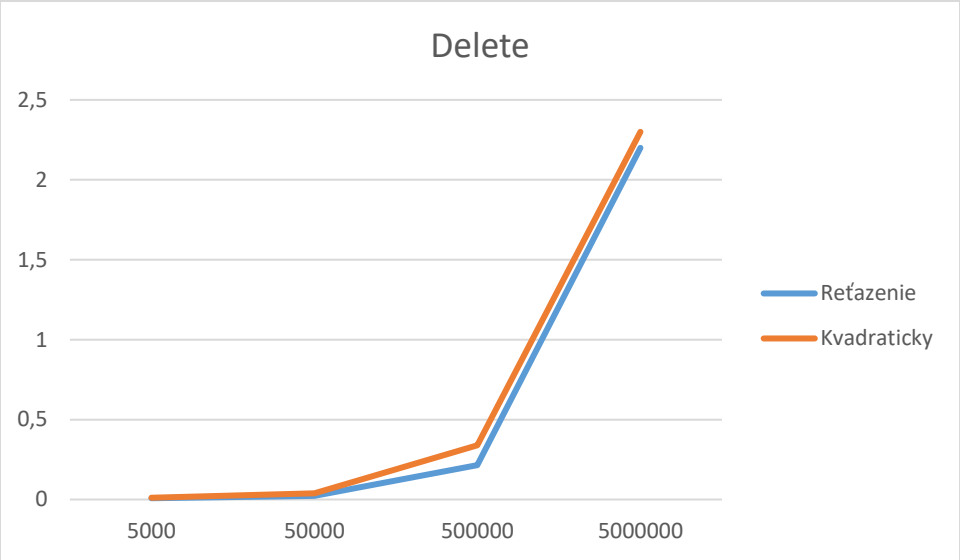
b) Pridávanie a vyhľadávanie pri 50 000 prvkoch

c) Pridávanie a vyhľadávanie pri 500 000 prvkoch

d) Pridávanie a vyhľadávanie pri 5 000 000 prvkoch







Z porovnania vyplýva, že hashovacia tabuľka riešená reťazením je efektívnejšia v prípade vyhľadávania a o nepatrný rozdiel aj v prípade vymazávania. Na druhej strane kvadraticky riešená hashovacia tabuľka je vhodnejšia pri vkladaní prvkov.