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

2020/2021

Dátové štruktúry a algoritmy

Zadanie č.2

Cvičiaci: Mgr. Martin Sabo, PhD.
Čas cvičení: Utorok 18:00 – 19:50

Vypracovala: Monika Zjavková
AIS ID: 105345

Obsah

Binárny vyhľadávací strom

1. Úvod

V binárnom strome sú údaje ukladané v uzloch. Moja implementácia je riešená s použitím štruktúr, ktorá obsahuje údaje o kľúči, výške a smerníky na ďalší ľavý uzol, ktorého kľúč je menší ako údaj uložený v aktuálnom a na ďalší pravý, v ktorom je uložený údaj väčší ako v aktuálnom uzle. V prípade, že takéto uzly neexistujú alebo ešte neboli pridané, smerník je nastavený na NULL.

Prehľadávanie je následne v tom jednoduchšie, že nie je potrebné prehľadávať všetky uzly

2. Vlastná implementácia

2.1. Vkladanie

```
struct node* insert(struct node *node, int data){
   if(node == NULL)                    return new_node(data);
   else if(data < node->data)          node->left = insert(node->left, data);
   else if (data > node->data)         node->right = insert(node->right, data);
   else                                return node;
```

Vkladanie prvkov do binárneho stromu je spravené pomocou rekurzie. Na začiatku skontroluje, či uzol, kde sa má vložiť nový údaj je prázdny. Ak nie, zavolá sa funkcia s ľavým alebo pravým ulom, podľa toho, či je údaj na vstupe menší alebo väčší. Prebieha pokým sa nedostane do prázdneho uzla, kde sa vytvorí nový s požadovanou hodnotou. V prípade, že taký údaj v strome už existuje, nedôjde k zmene. Pri každom vnáraní hlbšie sa zároveň aj nastavuje nová výška.

2.2. Vyvažovanie

Vyvažovanie je v programe riešené implementáciu AVL stromu. V nevyváženom strome vyhľadávanie je časovo závislé od výšky stromu až O(n), ak sú uzly pridávané na jednu stranu. V AVL strome je to O(logn) v priemernom a najhoršom prípade.

Vyvažovanie prebieha pomocou koeficientu vyvažovania. Prehľadáva sa postupne strom od aktuálne pridaného smerom ku koreňu a hľadá sa prvý nevyvážený uzol. Keďže sa do stromu nové uzly pridávajú rekurzívne, program má priamo prístup k predchádzajúcim uzlom a nie je sú potrebné smerníky späť.

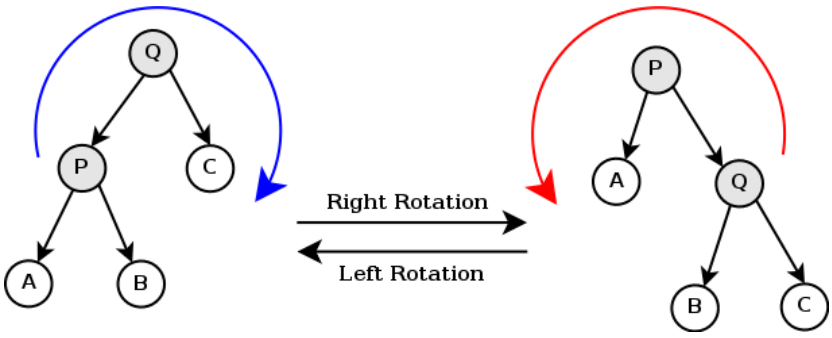
```
long int balance=0;

//vypočítanie koeficientu vyváženia
//výška ľavého - výška pravého podstromu
if(node->left != NULL)
    balance += node->left->height;
if(node->right != NULL)
    balance -= node->right->height;
```

Koeficient sa vypočíta ako odčítanie výšky pravého uzla podstromu od ľavého podstromu, pričom výšky sú uložené v každom podstrome. Ak je koeficient v intervale od 1 do -1 je strom vyvážený. V opačnom prípade je nutné strom vyvážiť rotáciami podľa štyroch scenárov, ak máme tri uzly *a,b,c* tak vyvažovanie je potrebné, ak:

1. *b* je ľavým uzlom *a* a *c* je ľavým *b*, vtedy sa strom vyváži dvakrát rotáciou doľava, ktoré sa dá vykonať raz rotáciou doprava.
2. *c* je pravým uzlom *b* a b je ľavým uzlom c, vtedy sa strom vyváži rotáciou doľava a doprava.

3. *b* je pravým uzlom *a* a *c* je pravým uzlom *b*, vtedy sa strom vyváži dvakrát rotáciou doprava, ktoré sa dá vykonať raz rotáciou doľava.
4. *b* je pravým uzlom *a* a *c* je ľavým uzlom *b,* vtedy sa strom vyváži rotáciou doprava a potom doľava.



Všetky štyri rotácie sa dajú spraviť kombináciou doprava alebo doľava, preto je postačujúce implementovať iba tieto dve funkcie a volať ich podľa potreby. Tiež je potrebné aktualizovať výšku podľa toho, ktoré uzly a ako sa menia. Ak sa rotácie aplikujú aj na koreň stromu, je potrebné zmeniť koreň, aby bolo možné pristupovať ku zmenenému stromu.

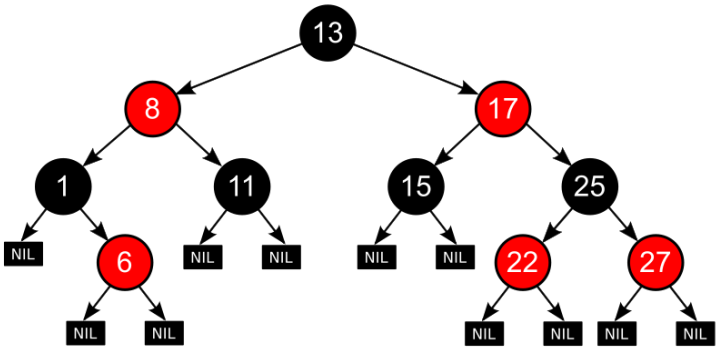
2.3. Vyhľadávanie prvku

Prvky sa prehľadávajú ako v nevyváženom strome. Efektívnejší spôsob je iteratívne, keďže nie je potrebné mať prístup k predchádzajúcim uzlom. Strom program prehľadáva od koreňa a posúva sa doprava alebo doľava podľa toho, či je kľúč väčší alebo menší ako kľúč práve prehľadávaného uzla.

3. Prevzatá implementácia binárneho stromu

Prevzatá implementácia pochádza zo stránky github, poskytnutý autorom Arthura Woimbée, riešená ako červeno-čierny strom. Vyvažovanie prebieha pomocou zafarbenia každého uzlu na červeno alebo čierno, pri čom koreň je vždy čierny a listy označované ako nil sú tiež čierne. Každý červený vrchol má dvoch čiernych potomkov a každá cesta z jedného vrcholu do listov, ktoré sú jeho potomkovia, obsahuje rovnaký počet čiernych vrcholov.

Vkladanie aj vyhľadávanie prvku prebieha v čase O(logn) ako v AVL strome, líšia sa iba v počte rotácii, ktoré je potrebné spraviť.



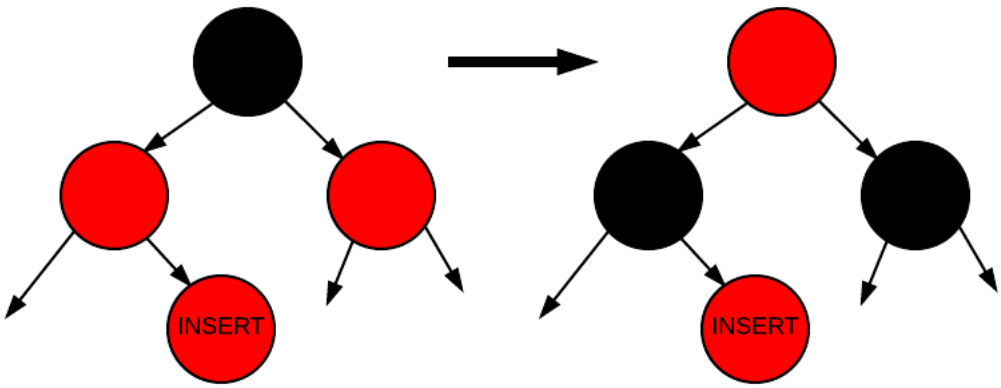
3.1. Vkladanie

Do červeno-čierneho binárneho stromu sa vkladajú uzly ako do nevyváženého, pri čom novovzniknutý uzol je zafarbený na červeno, pokiaľ nie je koreň, vtedy je zafarbený na čierno. Ak sa touto operáciou poruší vlastnosť červeno-čiernych stromov, napríklad nový uzol má tiež červeného rodiča, je potrebné napraviť strom prostredníctvom vyvažovania, ktoré prebieha po vložení nového uzla.

3.2. Vyvažovanie - teória

Po vložení uzla do stromu, je strom nutné vyvážiť. Vyvažovanie v červeno-čiernom strome prebieha v dvoma spôsobmi – najskôr prefarbením uzlov z červenej na čiernu alebo naopak, vždy však musí byť dodržané, že listy sú zafarbené na čierno. Keď to nie je postačujúce, strom sa vyvažuje rotáciami. V implementácii červeného-čierneho stromu sa používajú rovnaké rotácie ako pri použití AVL stromu – pravá rotácia a ľavá rotácia, pri čom obe rotácie pracujú v čase O(1).

Pri vyvažovaní, keď zafarbíme novo pridaný uzol na červeno, skontrolujeme, či jeho rodič nie je tiež, červený. Ak je, následne skontrolujeme uzol vedľa rodiča, v prípade, že je červený oba uzly nastavíme na čiernu farbu a skontrolujeme ich rodiča. Ak je červený nastavíme ho na červený a opakujeme rovnaký proces pre neho. Takto program pokračuje ďalej, kým sa nestavia všetky farby správne alebo kým sa nedostane do koreňa stromu, vtedy skončí.



Ak by bol susedný uzol rodiča čierny, vyvažovanie stromu prebieha pomocou rotácii, podobne ako v AVL strome:

1. dvakrát rotácia doľava
2. rotácia doľava a následne doprava
3. dvakrát rotácia doprava
4. rotácia doprava a následne doľava

Hashovanie

4. Úvod

Na ukladanie a vyhľadávanie údajov pomocou hashovania slúži hashovacia tabuľka. Dáta sú v nej uložené pomocou poľa. Umiestnenie v poli určíme v hashovacej funkcii, ktorá pre každý reťazec vypočíta algoritmom jedinečnú hodnotu a následným upravením získame index.

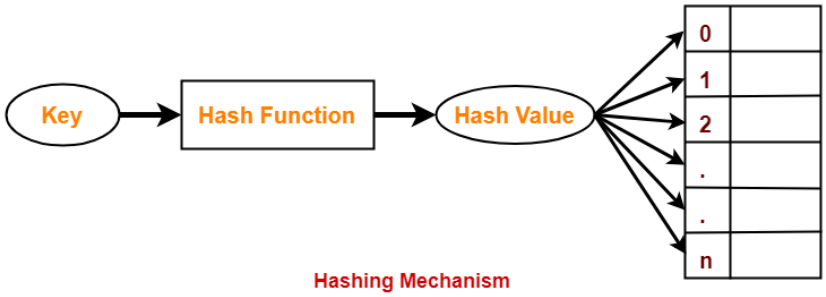
Časová zložitosť vyhľadávania pomocou hashovacej tabuľky v ideálnom prípade, kde nie sú kolízie, by bola O(1), keďže ku pozícii sa dostaneme pomocou hashovacej funkcie. V prípade, že nám však funkcia vráti rovnaký index a je potrebné riešiť kolízie, časová zložitosť je O(n), kde n je veľkosť poľa. To by však nastalo iba vtedy, ak by pre všetky prvky vznikol rovnaký index, inak to bude zvyčajne menej.

Hashovacia tabuľka je v mojej implementácii vytvorená pomocou dvojrozmerného dynamického poľa na ukladanie reťazcov.

5. Vlastná implementácia

5.1. Hashovacia funkcia

Slúži na vypočítanie hash hodnoty pre reťazec, ktoré je uvedený ako parameter.



V mojej implementácii hashovacia funkcia pracuje s ASCII kódom jednotlivých znakov. V cykle prechádza funkcia slovo po znakoch, následne pre každý znak vypočíta ASCII hodnotu. Pred tým, než sa pričíta už k vypočítanému slovu, prenásobí sa mocninou 33 podľa toho na akej pozícii sa nachádza. Týmto sa zabráni rovnakej hash hodnote pre slová z tých istých písmen na rôznych pozíciách napr. abc a cba, by násobenia mocniny 33 mali po sčítaní ASCII kódov tú istú hash hodnotu.

```c
int hash(char string[]){
    int hash=0;
    int letter;
    int len = strlen(string);                          //dĺžka slova

    for(int i = 0; i<len; i++){
        letter = (int)string[i] * pow(33,i);           //výpočet hashu pre jedno písmeno
        hash += letter;                                //pripočítanie písmena už k hotovému slovu
    }

    return hash;
}
```

Napríklad pre prvý znak ostane bez zmeny, lebo sa prenásobí iba 1, druhý znak však už 33, tretí 1089 atď., pri čom pretečenie sa neberie do úvahy. Funkcia nakoniec po sčítaní všetkých písmen vráti hash hodnotu celého reťazca.

5.2. Vkladanie do hashovacej tabuľky

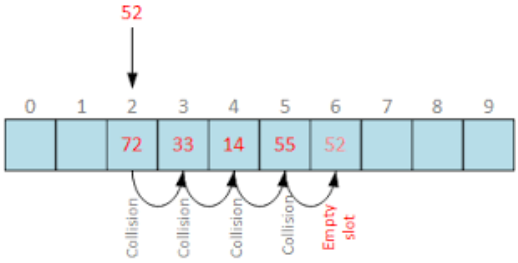
Po tom, ako vráti hashovacia funkcia hash hodnotu, vypočíta sa z nej index pomocou modula, aby sme dostali číslo, ktoré sa nachádza vo vnútri poľa. Pre prípad, že by hash hodnota bola záporná, funkcia počíta s absolútnou hodnotou daného čísla.

```c
h_string = hash(string);                           //vypočítanie hash hodnoty pre dané slovo
index = abs(h_string % (*size));                   //vypočítanie indexu z hash hodnoty
```

Po zistení indexu je následne možné vložiť reťazec do poľa. V prípade, že sa tam už nachádza nejaké slovo – pre dve rôzne slová bol vypočítaný ten istý index, je potrebné riešiť kolízie.

Moja implementácia rieši kolízie pomocou otvoreného adresovania, kde na každej pozícii môže byť najviac jeden prvok. Funkcia používa lineárne skúšanie, čiže ak nastane kolízia, slovo uložíme na najbližšie voľné miesto v poli, čo umožní jednoduchšie vyhľadávanie prvku.

V cykle sa teda prechádza celé pole, v prípade, že sa dostane na koniec, pokračuje od začiatku. Ak sa nájde voľné miesto, vloží sa na tú pozíciu reťazec, ak nie pokračuje, kým sa k nemu nedostane. Ak sa vrátime naspäť do pôvodného indexu, znamená to, že tabuľka je plná alebo nastala chyba.



V prípade, že nie je nutné implementovať vymazávanie zo zoznamu, je to postačujúci a jednoduchý algoritmus, kde nie je nutné použiť pamäť navyše pre smerníky.

Ak sa hashovacia tabuľka naplní do polovičky, je zväčšená na dvojnásobok, aby sa predišlo kolíziám. K novej veľkosti sa pripočíta 33, aby sme predišli tomu, že sa iba index zväčší na dvojnásobok. Vytvorí sa nové pole už s dostatočnou veľkosťou. Do neho sa potom vkladajú reťazce z pôvodnej tabuľky, pri čom sú znova zashasované a majú vypočítaný nový index.

```
int new_size = (*size*2)+33;                                    //vypočítanie novej veľkosti
char **new_array;
for(int i=0; i<*size; i++){                                     //Prehashovanie starej tabuľky a vloženie do novej
    if(array[i] != 0){
        insert(new_array, array[i], &new_size,full);
    }
}

*size = new_size;                                              //nastavenie novej veľkosti
return new_array;
```

Ak by sme neprehashovali starú tabuľku, mohlo by sa to isté slovo nachádzať na dvoch rôznych miestach alebo by vznikli problémy pri riešení kolízii. Po vytvorení, funkcia vráti novú tabuľku, do ktorej sa ďalej môžu ukladať nové reťazce.

5.3. Hľadanie prvku

Hľadanie prvkov prebieha na rovnakom princípe ako vkladanie do hashovacej tabuľky. Na začiatku sa vypočíta hash hodnota a z nej index. Potom môžu nastať tri prípady – ak je na vypočítanom mieste null, znamená to, že sa tam prvok nenachádza, ak nájde zhodu, znamená to, že slovo sa v tabuľke nachádza.

Tretí prípad je taký, že mohla nastať kolízia a slovo a sa nachádza na inom mieste v tabuľke. Vtedy sa znova prechádza tabuľka, kým nenájde zhodu alebo kým sa nedostane funkcia naspäť k pôvodnému indexu, v tom prípade sa slovo v tabuľke nenachádza.

6. Prevzatá implementácia hashovania

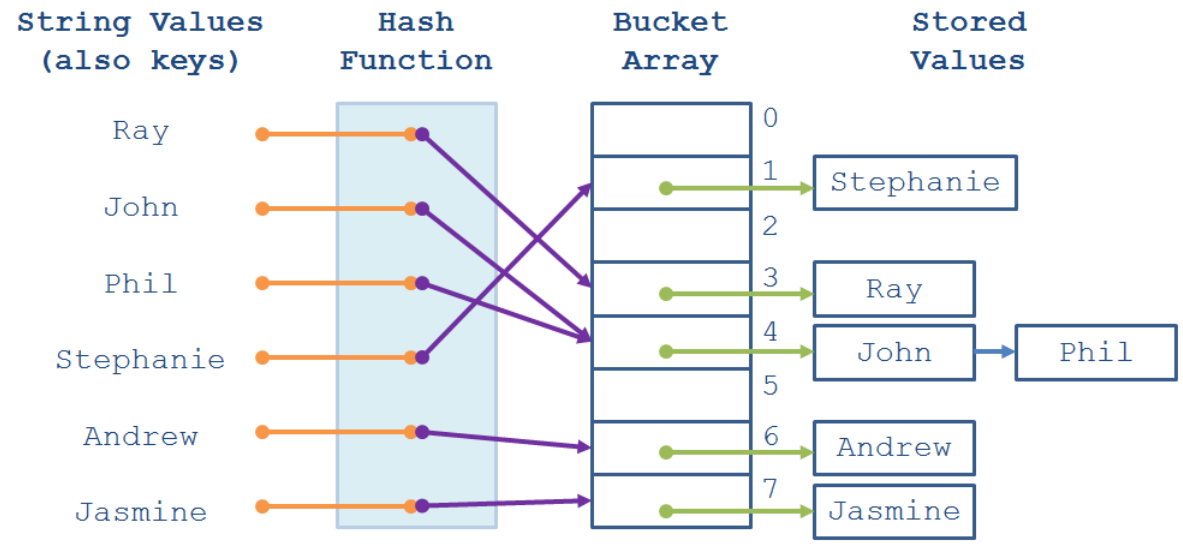
Druhá implementácia hashovania bola prevzatá zo stránky github, kde ju zverejnil používateľ Jiuli Gao. V tejto implementácii je zobrazovaná tabuľka pomocou štruktúr, pri čom na zobrazenie hashovacej tabuľky slúži štruktúra, ktorá obsahuje smerník na jej začiatok a štruktúra na zobrazenie jednotlivých dát obsahuje ukazovateľ na začiatok reťazca a na ďalší blok, s ktorým došlo ku kolízii.

6.1. Vkladanie do tabuľky

Funkcia vkladania do tabuľky funguje podobne ako pri mojej implementácii. Najskôr sa vytvorí nový uzol so zadanými dátami, pri čom ukazovateľ na ďalší uzol je nastavený na NULL a vypočíta sa hash hodnota, ktorá sa následne upraví na index, aby bolo možné vložiť prvok do poľa.

V prípade, že sa na danom mieste v tabuľke nič nenachádza, nový uzol je tam vložený a vkladanie sa ukončí. Ak sa tam už niečo nachádza je potrebné riešiť kolízie.

V prevzatej implementácii sú kolízie riešené metódou chaining, kedy v prípade, že sa vypočíta rovnaký index pre dva rôzne reťazce, vytvorí sa spájaný zoznam na danom indexe a aktuálne pridávané slovo sa pripojí na koniec zoznamu. V prípade teda, že na danom indexe už existuje slovo, prechádza sa celý spájaný zoznam, kým sa nedostane na koniec, kde sa pripojí alebo, keď sa už dané slovo v zozname nachádza, nenastane žiadna zmena.



6.2. Vyhľadávanie

Pri vyhľadávaní sa hľadaný reťazec zahashuje a vypočíta sa index, na ktorom mieste by sa mal v zozname nachádzať. Ak sa tam nachádza vráti hodnotu, ak nie znamená to, že mohla nastať kolízia, alebo sa v zozname reťazec nenachádza. V tom prípade je potrebné prezrieť celý spájaný zoznam, ktorý sa na danom indexe nachádza. Ak sa prvok nenachádza ani tam, znamená to, že došlo k chybe a prvok do tabuľky nikdy nebol vložený.

```
HashNode *node = hashtable->slots[slot];
while (node != NULL && strcmp(node->value, value)){
    node = node->next;
}

if (node == NULL){
    printf("KeyError\n");
}

return node;
```

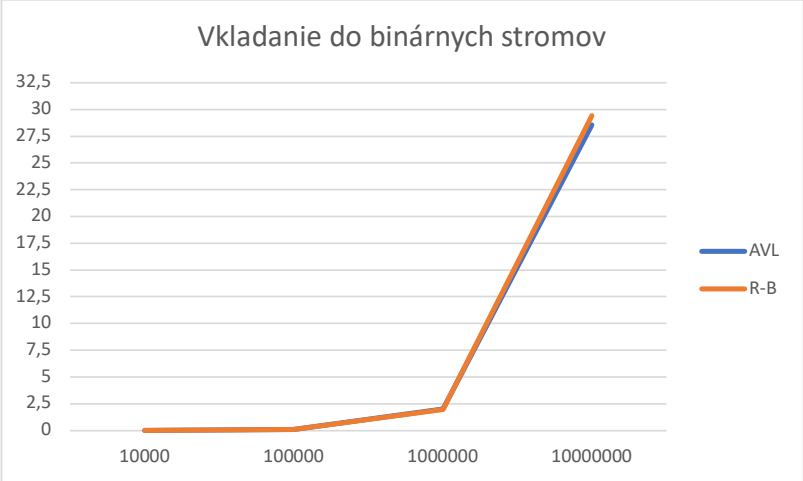
Testovanie

7. Vyhľadávacie stromy

Testovanie binárneho vyhľadávacieho stromu prebieha na základe generovania a pridávania náhodných generovaných čísel v rôznych intervaloch. Keď prebehne pridávanie bez chyby, prebieha testovanie vyhľadávania prvkov.

Pri oboch scenároch je meraný čas, ktorý je potrebný na vykonanie danej funkcie a porovnávaný medzi oboma implementáciami. Čas sa meral pri štyroch rôznych veľkostiach:

a) Pridávanie a vyhľadávanie pri 10 000 000 prvkoch
b) Pridávanie a vyhľadávanie pri 1 000 000 prvkoch
c) Pridávanie a vyhľadávanie pri 100 000 prvkoch
d) Pridávanie a vyhľadávanie pri 10 000 prvkoch



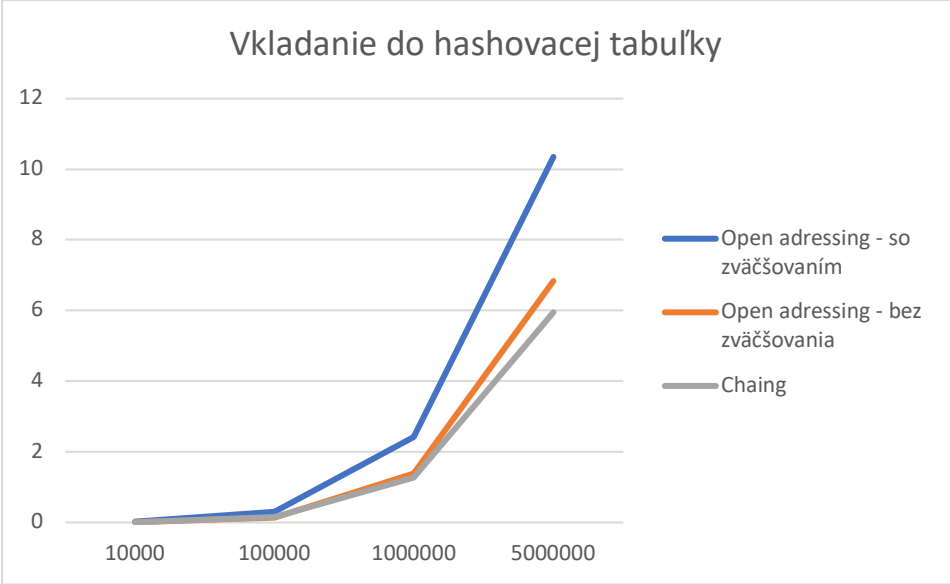
AVL stromy sú viacej vyvážené, ale pri pridávaní kvôli tomu môže nastať viacej rotácii v porovnaní s červeno-čiernymi stromami. Preto v prípade, že je potrebné časté pridávanie a vymazávanie do stromu, sú črevno-čierne stromy efektívnejšie. Ak je vyhľadávanie častejšia operácia, AVL stromy sú výhodnejšie.

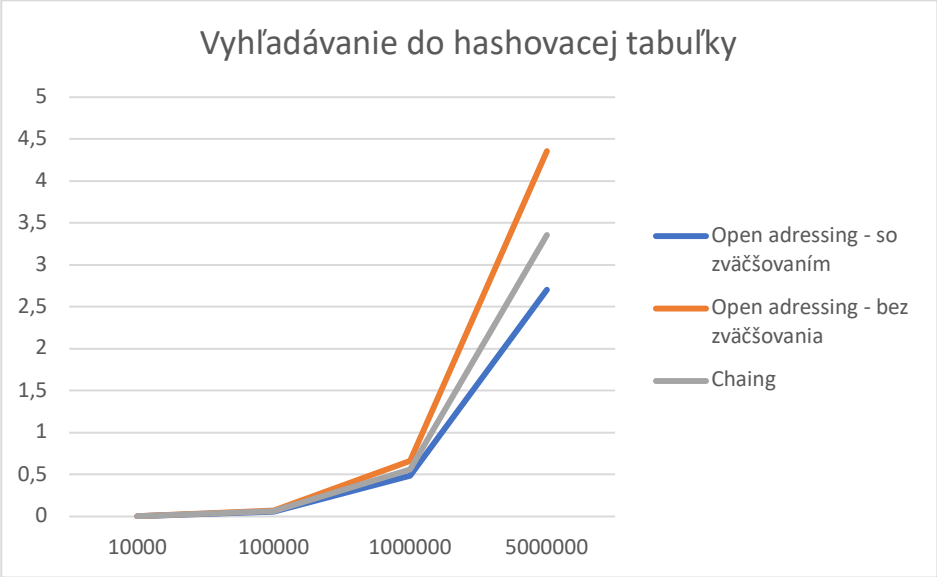
8. Hashovanie

Vyhľadávanie a pridávanie do hashovacej tabuľky sa testovalo na základe pridávania generovania náhodných reťazcov. Tiež podobne ako pri stromoch, keď bolo vloženie úspešné, zmeria sa čas na vyhľadávanie v zozname. Keďže prevzatá implementácia bola bez možnosti zväčšovania hashovacej tabuľky, testovanie prebehlo aj pri scenároch s dostatočne veľkou tabuľkou už na začiatku a v prípade, keď ju bolo potrebné zväčšovať.

Pri oboch scenároch je meraný čas, ktorý je potrebný na vykonanie danej funkcie a porovnávaný medzi oboma implementáciami. Čas sa meral pri štyroch rôznych veľkostiach:

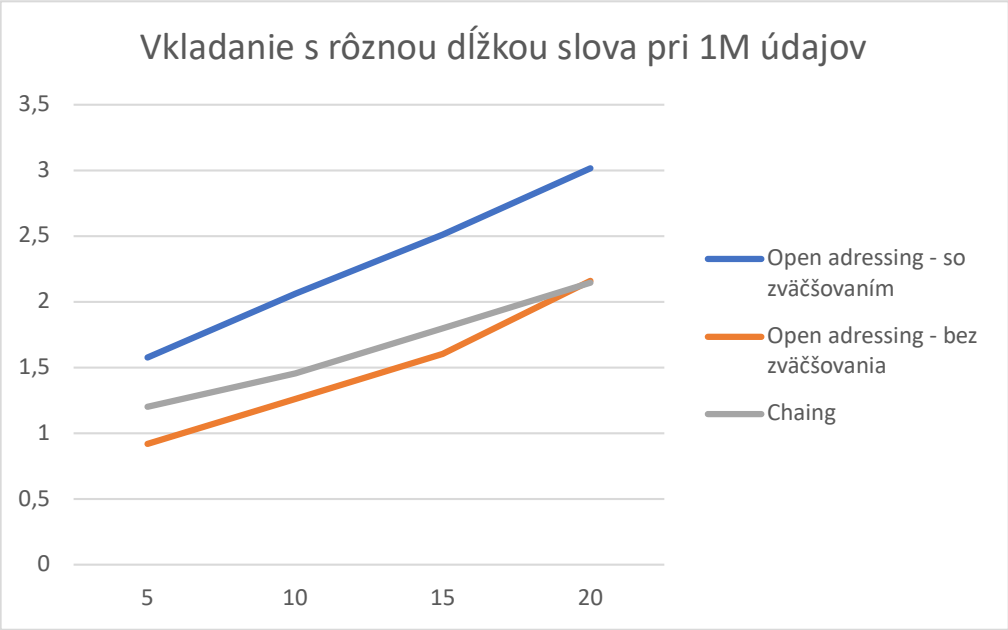
a) Pridávanie a vyhľadávanie pri 5 000 000 prvkoch
b) Pridávanie a vyhľadávanie pri 1 000 000 prvkoch
c) Pridávanie a vyhľadávanie pri 100 000 prvkoch
d) Pridávanie a vyhľadávanie pri 10 000 prvkoch





Okrem počtu údajov záleží aj na dĺžke slova, ktoré sa vkladá do tabuľky. Ďalšie scenáre predstavujú ukladanie do tabuľky s počtom dát 1 milión v prípadoch, kedy:

a) Slovo je dlhé 20 znakov
b) Slovo je dlhé 15 znakov
c) Slovo je dlhé 10 znakov
d) Slovo je dlhé 5 znakov



Otvorené adresovanie je efektívnejšie, pre pamäť keďže nie sú potrebné navyše smerníky, pre reťazce, kde vznikli kolízie. Tiež je k dátam jednoduchší prístup, keďže sú všetky uložené v jednom poli. Naopak v prípade vymazávania by bolo otvorené adresovanie oveľa náročnejšie, keďže by bolo potrebné nájsť pozíciu prvku, ktorý chceme vymazať a následne by museli byť tiež upravené prvky, s ktorými vznikla kolízia.