[DSA] Zadanie 2 – Vyhľadávanie v dynamických množinách

Autor: Ema Richnáková

Vkladané dáta do **stromov** sú čísla.
Riešenie kolízii: ak sa do stromu vloží hodnota, ktorá sa v strome už nachádza, vloží sa do ľavej vetvy (ako menšie čísla), ale nijako sa neodlišuje od druhej rovnakej hodnoty v strome, takže pri vyhľadávaní nejakej hodnoty, ktorá je tam viac ako 1x, vždy nájde prvú, na ktorú pri hľadaní narazí.

Použité algoritmy pre stromy:

- Moja implementácia
  - nevyvážený binárny vyhľadávací strom (BVS)
  - vyvážený binárny vyhľadávací strom s algoritmom vyvažovania AVL
- Cudzia implementácia
  - vyvážený binárny vyhľadávací strom s algoritmom vyvažovania červeno - čierneho stromu

Vkladané dáta do **hashovacích tabuliek** sú stringy alebo teda slová náhodne zložené z malých a veľkých písmen a čísel.

Použité algoritmy pre hashovanie:

- Moja implementácia
  - hashovanie stringov, riešenie kolízii: pomocou dvojitého hashovania
- Cudzia implementácia
  - hashovanie stringov, riešenie kolízii: ak nastane kolízia, prepíše prvok

Binárny vyhľadávací strom

- ## Nevyvážený binárny vyhľadávací strom

Implementácia nevyváženého binárneho vyhľadávacieho stromu (skr. NBVS) sa skladá zo 4 funkcii:

- void BVS_search(BVSnode *tree, int wanted_data);

- void BVS_print(BVSnode *tree);

- BVSnode *BVS_new_node(BVSnode *parent, int data);

- BVSnode *BVS_insert(BVSnode *tree, int new_data);

```
typedef struct BVSnode {
    int data;
    struct BVSnode *parent, *left, *right;
} BVSnode;
```

Štruktúra NBVS sa skladá z dát (číslo) a zo smerníkov na rodiča a na pravé a ľavé dieťa.

- **void BVS_search(BVSnode *tree, int wanted_data);**

Daná funkcia vyhľadáva určité dáta v strome. Hľadané dáta sú uložené v premennej *wanted_data. Vo funkcii sa postupuje tak, že ak sú vložené dáta menšie ako aktuálny prvok, posúva sa do ľavej vetvy, alebo tzv. podstromu a vyhľadáva medzi prvkami z tejto novej podmnožiny. Naopak, ak vložené dáta sú* väčšie ako aktuálny prvok, posúva sa do pravej vetvy. Ak sa dostane na prvok, ktorého dáta sú rovnaké s vyhľadávaním prvkom, vypíše na akej úrovni v strome sa nachádza a ukončí funkciu.
Zisťovanie úrovne je umožnené globálnou premennou *BVS_count_lvl* , ktorá je rovná číslu 1 a ktorá sa inkrementuje pri každom posunutí do ľavej alebo pravej vetvy stromu. Ak sa nájde hľadaný prvok, daná premenná sa znova bude rovnať 1.

- void BVS_print(BVSnode *tree);

Funkcia vypisuje jednotlive prvky stromu. Pre prvok vypíše, aké dáta ukladá v sebe; dáta, ktoré ukladá jeho rodič; dáta, ktoré ukladá jeho ľavé a pravé dieťa. Ak rodič alebo deti nie sú definované, vypíše NONE.

- BVSnode *BVS_new_node(BVSnode *parent, int data);

Funkcia vytvorí nový prvok, ktorému vyhradí miesto v pamäti funkciou malloc a priradí mu príslušné dáta, rodiča a keďže je to nový prvok, nemá zatiaľ žiadne deti priradené.

- BVSnode *BVS_insert(BVSnode *tree, int new_data);

```
BVSnode *BVS_insert(BVSnode *tree, int new_data) {
    BVSnode **root = &tree; //aktualny prvok
    BVSnode *parent = NULL; //rodic aktualneho prvku

    while (1) {
```

```
        if (*root == NULL) { //ak nie je na danom mieste prvok, vyrvori novy p
rvok a vlozi ho tam
            *root = BVS_new_node(parent, new_data);
            return tree;

        } else if (new_data <= (*root)-
>data) { //ak vkladane data su mensie alebo rovne ako aktualny prvok, posuva s
a dolava
            parent = *root;
            root = &(*root)->left;

        } else { //ak vkladane data su vacsie ako aktualny prvok, posuva sa do
prava
            parent = *root;
            root = &(*root)->right;
        }
    }
}
```

Funkcia vkladá nový prvok do stromu. Vykonáva to takým spôsobom, že vždy porovná dáta aktuálneho prvku v strome, ak sú vkladané dáta menšie alebo rovné , posúva sa v strome doľava a ak sú väčšie, posúva sa doprava. Toto sa vykonáva dovtedy, kým aktuálny prvok nebude NULL, teda na danom mieste neexistuje prvok, tak na to miesto sa vloží nový prvok, ktorý je vytvorený funkciou *BVS_new_node.*

- ## Vyvážený binárny vyhľadávací strom s algoritmom vyvažovania AVL

Implementácia AVL stromu sa skladá z 8 funkcii:

- void AVL_search(AVLnode *tree, int wanted_data);

- void AVL_print(AVLnode *tree);

- int max(int left, int right);

- int AVL_height(AVLnode *tree, char ch);

- AVLnode *AVL_right_rotation(AVLnode *tree);

- AVLnode *AVL_left_rotation(AVLnode *tree);

- AVLnode *AVL_new_node(AVLnode *tree, AVLnode *parent, int data);

- AVLnode *AVL_insert(AVLnode *tree, AVLnode *parent, int new_data);

```c
typedef struct AVLnode {
    int data;
    struct AVLnode *parent, *left, *right;
    int lh, rh;
} AVLnode;
```

Štruktúra AVL stromu sa skladá z dát (číslo), smerníkov na rodiča, na pravé a ľavé dieťa a výšky* ľavej a pravej vetvy.
(* všeobecne výška je počet prvkov medzi aktuálnym prvkom a najvzdialenejším listom jeho podstromu)

- void AVL_search(AVLnode *tree, int wanted_data);

Funguje tak isto ako *void BVS_search(BVSnode *tree, int wanted_data).*

- void AVL_print(AVLnode *tree);
Funguje podobne ako *void BVS_print(BVSnode *tree).* Len navyše vypisuje výšky pre daný prvok.

- int max(int left, int right);
Pomocná funkcia na zistenie maximálnej hodnoty z dvoch čísel.

- int AVL_height(AVLnode *tree, char ch);

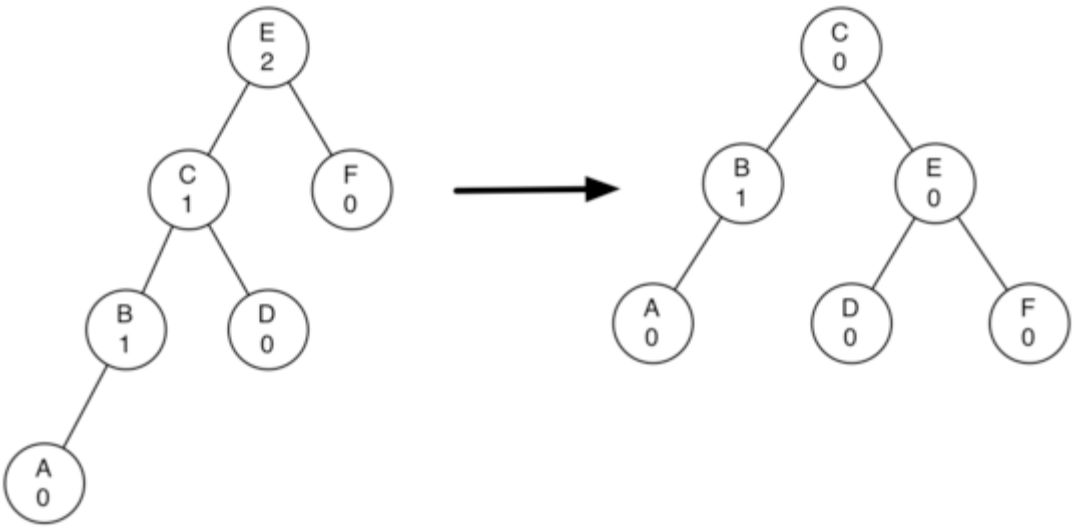
```c
int AVL_height(AVLnode *tree, char ch) {
    if (ch == 'l') { //zistenie vysky pre lavu vetvu
        if (tree->left == NULL) {
            return 0;
        } else {
            return max(tree->left->lh, tree->left->rh) + 1;
        }
    } else if (ch == 'r') { //zistenie vysky pre pravu vetvu
        if (tree->right == NULL) {
            return 0;
        } else {
            return max(tree->right->lh, tree->right->rh) + 1;
        }
    } else { //ak bola zadana ina hodnota premennej ch
        printf("return -1");
        return -1;
    }
}
```

Funkcia zisťuje výšku buď pre ľavú alebo pravú vetvu podstromu. Do premennej *ch* sa vkladá znak. Ak je daný znak 'l', vypočítava výšku pre ľavý podstrom tak, že zoberie ľavú a pravú výšku ľavého dieťaťa, vyberie maximum z týchto hodnôt a pripočíta 1. Ak nemá ľavé dieťa, to znamená, že ľavé dieťa neexistuje a teda neexistuje ani ľavý podstrom a daný prvok je teda list, tak jeho ľavá výška je 0.
Rovnaký proces sa deje, ak daný znak 'r', len teda berie hodnoty pravého dieťaťa a zisťuje teda hodnotu pravej vetvy.

- AVLnode *AVL_right_rotation(AVLnode *tree);

```
AVLnode *AVL_right_rotation(AVLnode *tree) {
    AVLnode *left_child = tree->left;

    //otocenie prvku tree doprava, prvok left_child sa dostane nad prvok tree
    tree->left = left_child->right;
    left_child->right = tree;

    //prepisanie rodicov oboch prvkov
    left_child->parent = tree->parent;
    tree->parent = left_child;
    if (tree->left != NULL) {
        tree->left->parent = tree;
    }

    //prepocitanie vysok pre oba prvky, kedze sa vyrovnavali vetvy (a teda men
ila sa ich vyska)
    tree->lh = AVL_height(tree, 'l');
    tree->rh = AVL_height(tree, 'r');
    left_child->lh = AVL_height(left_child, 'l');
    left_child->rh = AVL_height(left_child, 'r');

    return left_child;
}
```

Funkcia otáča prvok *tree* doprava. Ukážka na obrázku 1. nižšie.

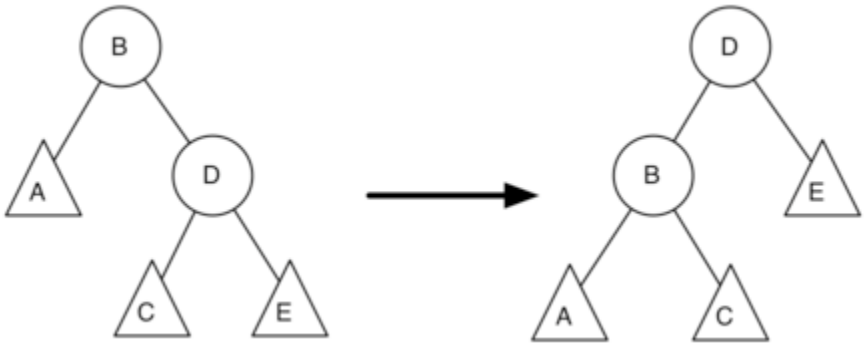


Obrázok 1. Pravá rotácia prvku C s vyobrazenými výškami (zdroj: https://runestone.academy/runestone/books/published/pythonds/Trees/AVLTreeImplementation.html)

- AVLnode *AVL_left_rotation(AVLnode *tree);

```
AVLnode *AVL_left_rotation(AVLnode *tree) {
    AVLnode *right_child = tree->right;

    //otocenie prvku tree dolava, prvok right_child sa dostane nad prvok tree
    tree->right = right_child->left;
    right_child->left = tree;

    //prepisanie rodicov oboch prvkov
    right_child->parent = tree->parent;
    tree->parent = right_child;
    if (tree->right != NULL) {
        tree->right->parent = tree;
    }

    //prepocitanie vysok pre oba prvky, kedze sa vyrovnavali vetvy (a teda men
ila sa ich vyska)
    tree->lh = AVL_height(tree, 'l');
    tree->rh = AVL_height(tree, 'r');
    right_child->lh = AVL_height(right_child, 'l');
    right_child->rh = AVL_height(right_child, 'r');

    return right_child;
}
```

Funkcia otáča prvok *tree* doľava. Ukážka na obrázku 2. nižšie.



Obrázok 2. Ľavá rotácia prvku B (zdroj: https://runestone.academy/runestone/books/published/pythonds/Trees/AVLTreeImplementation.html)

- AVLnode *AVL_new_node(AVLnode *tree, AVLnode *parent, int data);

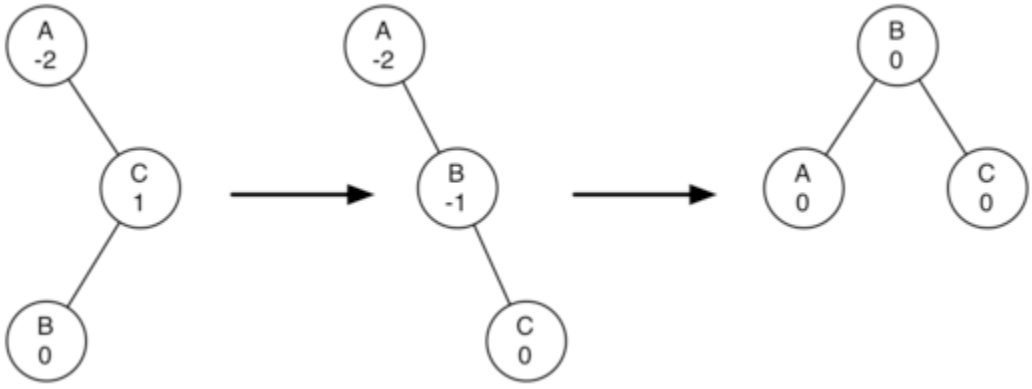
Funguje podobne ako BVSnode *BVS_new_node(BVSnode *parent, int data). Len navyše pridáva údaj o výškach pravej a ľavej vetvy pre daný prvok.

- AVLnode *AVL_insert(AVLnode *tree, AVLnode *parent, int new_data);

```
AVLnode *AVL_insert(AVLnode *tree, AVLnode *parent, int new_data) {
    if (tree == NULL) { //ak nie je na danom mieste prvok, vyrvori novy prvok
a vlozi ho tam
        tree = AVL_new_node(tree, parent, new_data);

    } else if (new_data <= tree-
>data) { //ak vkladane data su mensie/rovne ako aktualny prvok, posuva sa dola
va...
        tree->left = AVL_insert(tree-
>left, tree, new_data); //...novy prvok bude v lavej vetve aktualneho prvku
        tree-
>lh = AVL_height(tree, 'l'); //prepocitanie vysky pre lavy podstrom aktualneho
 prvku

        //ak faktor vyvazenia (lava vyska - prava vyska) dosiahne hodnotu 2, j
e strom nevyvazeny a treba robit rotaciu
        if((tree->lh - tree->rh) == 2) {
            if(new_data <= tree->left->data){
                //ak vlozena hodnota je mensia/rovna, ako hodnota laveho dieta
ta aktualneho prvku, robi sa prava rotacia
```

```
                //aktualneho prvku, aby sa mensie lave dieta dostalo vyssie v
strome a vyrovnal sa rozdiel medzi
                //lavym novym prvkov a kratsou pravou vetvou stromu
                tree = AVL_right_rotation(tree);

            } else {
                //inak sa robi lava rotacia laveho dietata a prava rotacia akt
ualneho prvku
                //touto postupnostou sa zachovava myslienka ,,mensie cislo nal
avo, vacsie/rovne napravo"
                tree->left = AVL_left_rotation(tree->left);
                tree = AVL_right_rotation(tree);
            }
        }

    } else { //ak vkladane data su vacsie ako aktualny prvok, posuva sa doprav
a...
        tree->right = AVL_insert(tree-
>right, tree, new_data); //...novy prvok bude v pravej vetve aktualneho prvku
        tree-
>rh = AVL_height(tree, 'r'); //prepocitanie vysky pre pravy podstrom aktualneh
o prvku

        //ak faktor vyvazenia (lava vyska - prava vyska) dosiahne hodnotu -
2, je strom nevyvazeny a treba robit rotaciu
        if((tree->lh - tree->rh) == -2) {
            if(new_data > tree->right->data){
                //ak vlozena hodnota je vacsia ako hodnota praveho dietata akt
ualneho prvku, robi sa lava rotacia
                //aktualneho prvku, aby sa vacsie prave dieta dostalo vyssie v
 strome a vyrovnal sa rozdiel medzi
                //pravym novym prvkom a kratsou lavou vetvou stromu
                tree = AVL_left_rotation(tree);

            } else {
                //inak sa robi prava rotacia praveho dietata a lava rotacia ak
tualneho prvku
                //touto postupnostou sa zachovava myslienka ,,mensie/rovne cis
lo nalavo, vacsie napravo"
                tree->right = AVL_right_rotation(tree->right);
                tree = AVL_left_rotation(tree);
            }
        }
    }

    return(tree);
}
```

Funkcia prechádza daným stromom rekurzívne. Ak sú vkladané dáta menšie alebo rovné ako dáta aktuálneho prvku, posúva sa v strome doľava, ak sú väčšie, posúva sa doprava. Ak aktuálny prvok neexistuje, tak na dané miesto v strome sa uloží nový prvok, ktorý nám vráti funkcia *AVL_new_node(tree, parent, new_data).*
Po tom, ako sa nový prvok vloží na vhodné miesto, sa vracia spätne po prvkoch, ktoré sa porovnávali s novým prvkov (vďaka rekurzii). Každému prvku sa upraví ľavá alebo pravá výška. To závisí od toho, či sa vkladalo do ľavého alebo pravého podstromu. Po úprave jednej z výšok sa skontroluje, či sú vetvy daného prvku vyvážené, teda či rozdiel medzi ľavou a pravou vetvou nie je väčší ako 1 (teda nie je jedna z vetiev dlhšia o viac ako 1 prvok).
Ak sú vetvy vyvážené, nič sa nemení a pokračuje sa na ďalší (teda predošlí prvok).
A ak nie sú vyvážené, tak sa bude aktuálny prvok rotovať. Podľa rôznych podmienok sa aktuálny prvok môže rotovať len doprava (vizuálna ukážka vyššie na obrázku 1.) alebo len doľava (vizuálna ukážka vyššie na obrázku 2.) alebo môže nastať aj taká situácia, že najprv bude rotovať ľavé dieťa aktuálneho prvku doľava a potom aktuálny prvok doprava alebo bude rotovať pravé dieťa aktuálneho prvku doprava a potom aktuálny prvok doľava (viď obrázok 3. nižšie). Všetky tieto situácie sa dejú z dôvodu zachovania myšlienky uloženia prvkov v binárnom strome, t.j. menšie alebo rovnaké prvky sú uložené v ľavej vetve a väčšie prvky v pravej vetve.



Obrázok 3. Pravá rotácia prvku C (pravé dieťa aktuálneho prvku A) a ľavá rotácia prvku aktuálneho prvku A (zdroj: https://runestone.academy/runestone/books/published/pythonds/Trees/AVLTreeImplementation.html)

- **Vyvážený binárny vyhľadávací strom s algoritmom vyvažovania červeno - čierneho stromu**

Kód je prevzatý z
https://gist.github.com/aagontuk/38b4070911391dd2806f.

Pravidlá červeno - čierneho stromu:
(zdroj: https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/)

1) Každý prvok má buď červenú alebo čiernu farbu.
2) Koreň stromu musí byť vždy čierny.
3) Nikdy nie sú pri sebe červené prvky (teda nemôžu byť rodič aj dieťa naraz červený).

4) Každá cesta od nejakého prvku (vrátane koreňa) po prvok v jeho podstrome, ktorý je rovný NULL má rovnaký počet čiernych prvkov.

Implementácia červeno - čierneho stromu sa skladá z 13 funkcii (prvé 3 sú mnou pridané):

- void RB_search(struct node *tree, int wanted_data);
- void RB_set_root();
- struct node *RB_get_root();
- void left_rotate(struct node *x);
- void right_rotate(struct node *x);
- void tree_print(struct node *x);
- void red_black_insert(int key);
- void red_black_insert_fixup(struct node *z);
- struct node *tree_search(int key);
- struct node *tree_minimum(struct node *x);
- void red_black_transplant(struct node *u, struct node *v);
- void red_black_delete(struct node *z);
- void red_black_delete_fixup(struct node *x);

```
struct node{
    int key;
    int color;
    struct node *parent;
```

```
    struct node *left;
    struct node *right;
};
```
Štruktúra Červeno - čierneho stromu sa skladá z dát (číslo) (premenná key), smerníkov na rodiča, na pravé a ľavé dieťa a farby prvku.

-void RB_search(struct node *tree, int wanted_data);
Zdôvodnenie implementácie sa nachádza aj v kóde pri danej funkcii ako komentár, ale napíšem to aj sem.

Rovnaká funkcia search, ako je implementovaná v mojom nevyváženom a vyváženom strome, aj keď daná implementácia má funkciu search, chcela som implementovať moju funkciu search, kvôli, zjednoteniu výpisov pri testoch, aby sa mi lepšie spracovávali údaje pri analýze testov.

- void RB_set_root();

Znova zdôvodnenie implementácie sa nachádza aj v kóde pri danej funkcii ako komentár, ale napíšem to aj sem.

Pôvodne sa daný kus kódu nachádzal v main-e (ktorý viete nájsť hore odkomentovaný), ale aby som vedela pracovať s danou implementáciou, musela som daný kód osamostatniť do funkcie, aby som ho vedela aplikovať v mojich testoch

- struct node *RB_get_root();
Do tretice, zdôvodnenie implementácie sa nachádza aj v kóde pri danej funkcii ako komentár, ale napíšem to aj sem.

Táto funkcia slúži len na to, aby som vedela získať adresu, kde sa daná stromová štruktúra nachádza a mala ju uloženú v mojom main file a vedela ju vložiť do mojej search funkcie, ktorú som nechcela upravovať.

- void left_rotate(struct node *x);
Funkcia vykonáva ľavú rotáciu aktuálneho prvku x.

- void right_rotate(struct node *x);
Funkcia vykonáva pravú rotáciu aktuálneho prvku x.

- void tree_print(struct node *x);

Funkcia vypíše prvky stromu vzostupne (teda od najmenšieho ľavého prvku po najväčší pravý prvok).

- void red_black_insert(int key);
Funkcia vkladá nový prvok ako pri BVS alebo AVL strome, čiže porovnáva vkladaný prvok z prvkami stromu a podľa toho sa posúva buď doľava alebo doprava na základe toho, či je vkladaný prvok menší/rovný alebo väčší ako aktuálny prvok stromu. Ale pri vkladaní sa musia dodržiavať určité pravidlá pre Červeno - čierny strom (pravidlá sú vypísané úplne hore v tejto sekcii). Vkladaný prvok je vždy červený. A aby sa neporušili niektoré pravidlá, tak funkcia ďalej pokračuje funkciou ***void red_black_insert_fixup(struct node *z),*** kde sa prefarbujú (ak je treba) predošlé prvky a rotujú sa jednotlive podstromy.

- struct node *tree_search(int key);
Funkcia vyhľadáva prvok v strome. Ak ho nájde, funkcia ho vráti.

- struct node *tree_minimum(struct node *x);
Funkcia nájde najmenšiu hodnotu v strome.

- void red_black_transplant(struct node *u, struct node *v);

Funkcia nahradí prvok *u* prvkom *v*.

- void red_black_delete(struct node *z);
Funkcia vymaže prvok *z* zo stromu.

- void red_black_delete_fixup(struct node *x);
Funkcia opraví strom po zmazaní prvku, tak, aby spĺňal pravidlá.

Hashovanie

- ## Moja implementácia hashovania

Moja implementácia hashovania sa skladá z 10 funkcii:

- int get_next_prime(int size);

- int get_prev_prime(int size);

- HASHMAP *MY_init(int size);

- int hash_fun1(int key, int size);

- int hash_fun2(int index1, int count, int key, int prev_prime, int size);

- HASHMAP *MY_resize(HASHMAP *hash_table);

- HASHMAP *MY_insert(HASHMAP *hash_table, char *data, int key);

- void MY_search(HASHMAP *hash_table, char *data, int wanted_key);

- void MY_print(HASHMAP *hash);

- void MY_delete_hashmap(HASHMAP *hash);

```
typedef struct {
    int key;
    char *data;
} ELEMENT;

typedef struct{
    ELEMENT *table;
    int size, number_of_elements;
    double fullness;
} HASHMAP;
```

Štruktúra hashovacej tabuľky sa skladá z veľkosti tabuľky, jej naplnenia a zo smerníku na tabuľku, ktorá v sebe ukladá prvky typu ELEMENT, ktoré v sebe ukladajú kľúč a dáta (slovo).

- int get_next_prime(int size);

Funkcia, ktorá zistí najbližšie ďalšie prvočíslo k vloženému číslu (využívaná pre veľkosti tabuľky, keďže jej veľkosť chceme len ako prvočíslo).

- int get_prev_prime(int size);

Funkcia, ktorá zistí najbližšie predchádzajúce prvočíslo k vloženému číslu (využívaná v druhej hashovacej funkcii).

- HASHMAP *MY_init(int size);

Inicializácia hashovacej tabuľky.

- int hash_fun1(int key, int size);

```
int hash_fun1(int key, int size) {
    return (key % size);
}
```

Prvá hashovacia funkcia -> (kľúč prvku) % (veľkosť tabuľky).

- int hash_fun2(int index1, int count, int key, int prev_prime, int size);

```
int hash_fun2(int index1, int count, int key, int prev_prime, int size){
    return ((index1 + ((count % 2003) * ((prev_prime - (key % prev_prime)))))
% size);
}
```

Druhá hashovacia funkcia. Premenná *prev_prime* -> prvočíslo, ktoré je najbližšie menšie k veľkosti tabuľky.

- HASHMAP *MY_resize(HASHMAP *hash_table);

Zmena veľkosti tabuľky a rehashovanie sa udeje, ak alfa-naplnenosti dosiahne hodnotu 0,9. Je to tak zvolené, lebo z automatického testovania som zistila, že pri takejto hodnote je moje hashovanie najrýchlejšie. Daná funkcia funguje tak isto, ako funckia *HASHMAP *MY_insert(HASHMAP *hash_table, char *data, int key)*, len teda myšlienka danej funkcie je taká, že zoberie všetky prvky z pôvodnej tabuľky a generuje indexy pre novú tabuľku (s 10x väčšou veľkosťou ako je pôvodná). Opis procesu generovania indexov je podrobnejšie popísané o odstavec nižšie.

- HASHMAP *MY_insert(HASHMAP *hash_table, char *data, int key);

```
HASHMAP *MY_insert(HASHMAP *hash_table, char *data, int key) {
    if ((double)((double)hash_table->number_of_elements / (double)hash_table-
>size) > hash_table->fullness) { //ak je tabulka plna, treba ju zvacsit
        hash_table = MY_resize(hash_table);
    }

    int count = collision_count, prev_prime = get_prev_prime(hash_table-
>size);

    int index1 = hash_fun1(key, hash_table-
>size); //vypocet prveho indexu z prvej hashovacej funkcie
```

```
    int index2 = index1;
    ELEMENT *table = hash_table->table;

    while (1) {
        if (table[index2].key == 0) { //ak sa na danom indexe nic nenachadza,
ulozi data tam
            table[index2].data = data;
            table[index2].key = key;
            hash_table->number_of_elements += 1;
            collision_count = count;
            return hash_table;

        } else {
            //ale ak je dany index uz obsadeny, novy index dostaneme scitanim
vysledku z prvej hashovacej
            //funkcie a vysledku z druhej hashovacej funkcie vynasobenim pocit
adlom kolizii
            //(pocitadlo kolizii je preto modulovane %2000, lebo pri vkladani
100 000 prvkov uz hodnota
            // pocitadla bola moc vysoka na typ int)
            index2 = hash_fun2(index1, count, key, prev_prime, hash_table-
>size);
            count++;
        }
    }
}
```

Funkcia, ktorá vkladá prvok do hashovacej tabuľky. Index prvku, na ktorom v tabuľke bude uložený, je vygenerovaný najprv prvou hashovacou funckiou. Ak na vygenerovanom indexe sa nenachádza prvok, vloží ho na daný index, ale ak tam už nejaký prvok je, vygeneruje sa druhý index, podľa druhej hashovacej funkcie. A tento proces sa opakuje, pokiaľ *index2* nebude taký index, na ktorom v tabuľke bude miesto pre nový prvok.

- void MY_search(HASHMAP *hash_table, char *data, int wanted_key);

Funkcia vyhľadá chcené dáta z tabuľky pomocou kľúča. Vyhľadávací proces prebieha nasledovne: najprv sa vygeneruje index pomocou kľúča a prvej hashovacej funkcie, ak na danom indexe sa nenachádza rovnaký kľúč, aký sa do funkcie vložil, generuje sa nový index, ktorý je generovaný pomocou druhej hashovacej funkcie. A ak teda hľadáme podľa iného indexu, ako je vygenerovaný prvou hashovacou funkciou, musíme prehľadať možnosti s celým počtom kolízii, keďže nevieme, pri koľkej kolízii sa daný prvok uložil.

```
for (int i = 1; i <= collision_count; i++) {
```

```
            index2 = hash_fun2(index1, i, wanted_key, prev_prime, hash_table->size);
            if (hash_table->table[index2].key == wanted_key) {
                //a kedze nevieme, pri akom pocte kolizii sa prvok ulozil do tabulky, musime hladat index pre
                //cely pocet kolizii
                printf("MY_HASH Data '%s' is on index %d\n", data, index2);
                printf("O(%d)\n", i + 1);
                return;
            }
        }
```

- void MY_print(HASHMAP *hash);
Funkcia na vypísanie celej tabuľky (aj prázdnych políčok tabuľky).

- void MY_delete_hashmap(HASHMAP *hash);
Funkcia na uvoľnenie (vymazanie), celej hashovacej tabuľky.

- ## Cudzia implementácia hashovania

Alfa naplnenia je 1.

Cudzia implementácia hashovania sa skladá zo 6 funkcii:

- hashmap* hashmapCreate(int startsize);

- void hashmapInsert(hashmap*, const void* data, unsigned long key);

- void* hashmapRemove(hashmap*, unsigned long key);

- void* hashmapGet(hashmap*, unsigned long key);

- long hashmapCount(hashmap*);

- void hashmapDelete(hashmap*);

```
typedef struct {
    void* data;
    int flags;
    long key;
} hEntry;

struct s_hashmap{
```

```
    hEntry* table;
    long size, count;
};
```
Štruktúra hashovacej tabuľky sa skladá z veľkosti tabuľky, jej počtu prvkov a zo smerníku na tabuľku, ktorá v sebe ukladá prvky typu hEntry, ktoré v sebe ukladajú kľúč, označenie (plné/prázdne) a dáta (v mojom testovaní konkrétne slovo).

- hashmap* hashmapCreate(int startsize);

Inicializácia hashovacej tabuľky.

- void hashmapInsert(hashmap*, const void* data, unsigned long key);

Funkcia vkladá dáta do hashovacej tabuľky. Ak vkladaný prvok má rovnaký kľúč ako prvok na indexe, vygenerovaného hashovacou funkciou,

```
index = key % hash->size;
```
tak daný prvok prepíše.

- void* hashmapRemove(hashmap*, unsigned long key);

Funkcia vyhľadá prvok podľa kľúča a vymaže daný prvok.

- void* hashmapGet(hashmap*, unsigned long key);
Funkcia vyhľadá prvok podľa kľúča.

- long hashmapCount(hashmap*);
Funkcia vráti počet prvkov v hashovacej tabuľke.

- void hashmapDelete(hashmap*);
Uvoľni hashovaciu tabuľku.

Testovanie

2 fázy testovania:

- Manuálne
- Automatické

Manuálne testovanie

Manuálne testovanie som robila pri samotnom vytváraní jednotlivých algoritmov. Manuálne testovanie prebiehalo nasledovne:
Manuálne som vkladala rôzne prvky podľa predom predpripravených scenárov a vypisovala som si výsledné stromy/hashovacie tabuľky.

- Scenáre pre stromy (scenáre som si najprv načrtla na papieri)
  - vkladanie takých prvkov, aby bolo otestované samotne vkladanie, potom rotácia doľava/doprava a ľavá-pravá rotácia/pravá-ľavá rotácia, pričom som sledovala priraďovanie rodičov a detí, výpočty výšok a samotné správanie pri vkladní menších/väčších/rovnakých prvkov.
- Scenáre pre hashovanie
  - vkladanie takých prvkov, aby bolo otestované samotné vkladanie prvkov a správanie pri riešení kolízii pri rovnakých kľúčoch

Automatické testovanie

Testovanie sa zautomatizovalo po manuálnom otestovaní plnej funkčnosti algoritmov. Testovanie pozostáva z vkladania 10, 100, 1 000, 10 000 a 100 000 prvkov.

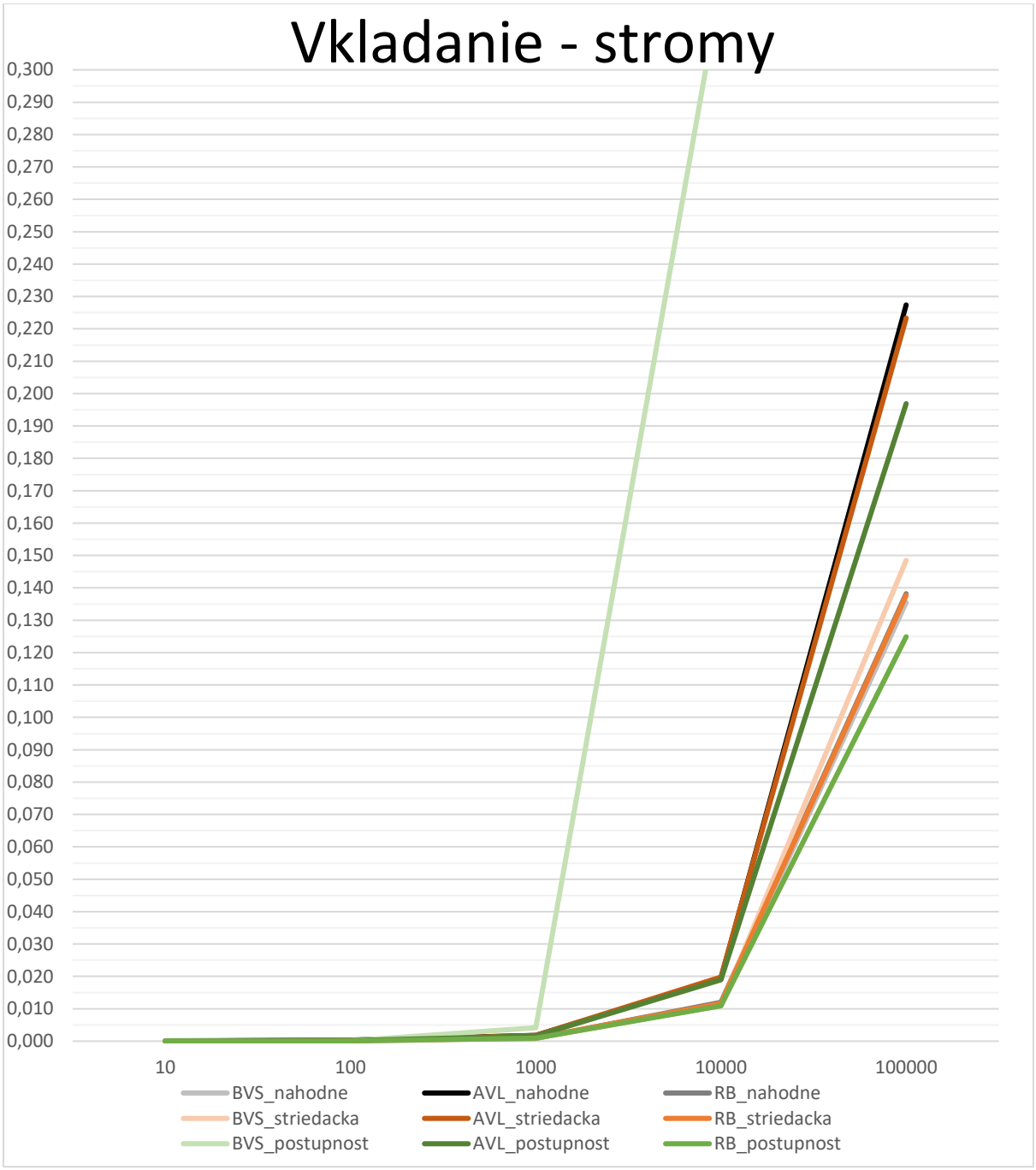
- Testovacie scenáre pre stromy
  - vkladanie náhodných prvkov
    - priemerný prípad zložitosti, keďže vstup by mal byť rovnomerne náhodný pre danú veľkosť a mal by sa vykonať priemerný počet krokov pre vloženie náhodného prvku
  - vkladanie prvkov "na striedačku"
    - najprv sa určil náhodný koreň a potom sa vkladali na striedačku menšie čísla od koreňa a väčšie čísla od koreňa
    - priemerný prípad pre vyvažovacie stromy, keďže stále dochádza k rotáciám – ale je to ideálny prípad, pre nevyvážený, keďže vždy na každej strane sa prechádza skoro rovnaký počet prvkov
  - vkladanie postupnosti čísel, čísla sú zoradené vzostupne
    - najhorší prípad pri nevyváženom, ak chce vložiť k-ty prvok, musí tak prejsť k-1 prvkov, čo je pri veľkých číslach zdĺhavé
- Testovacie scenáre pre hashovanie
  - vkladanie náhodných slov o maximálnej dĺžky 10

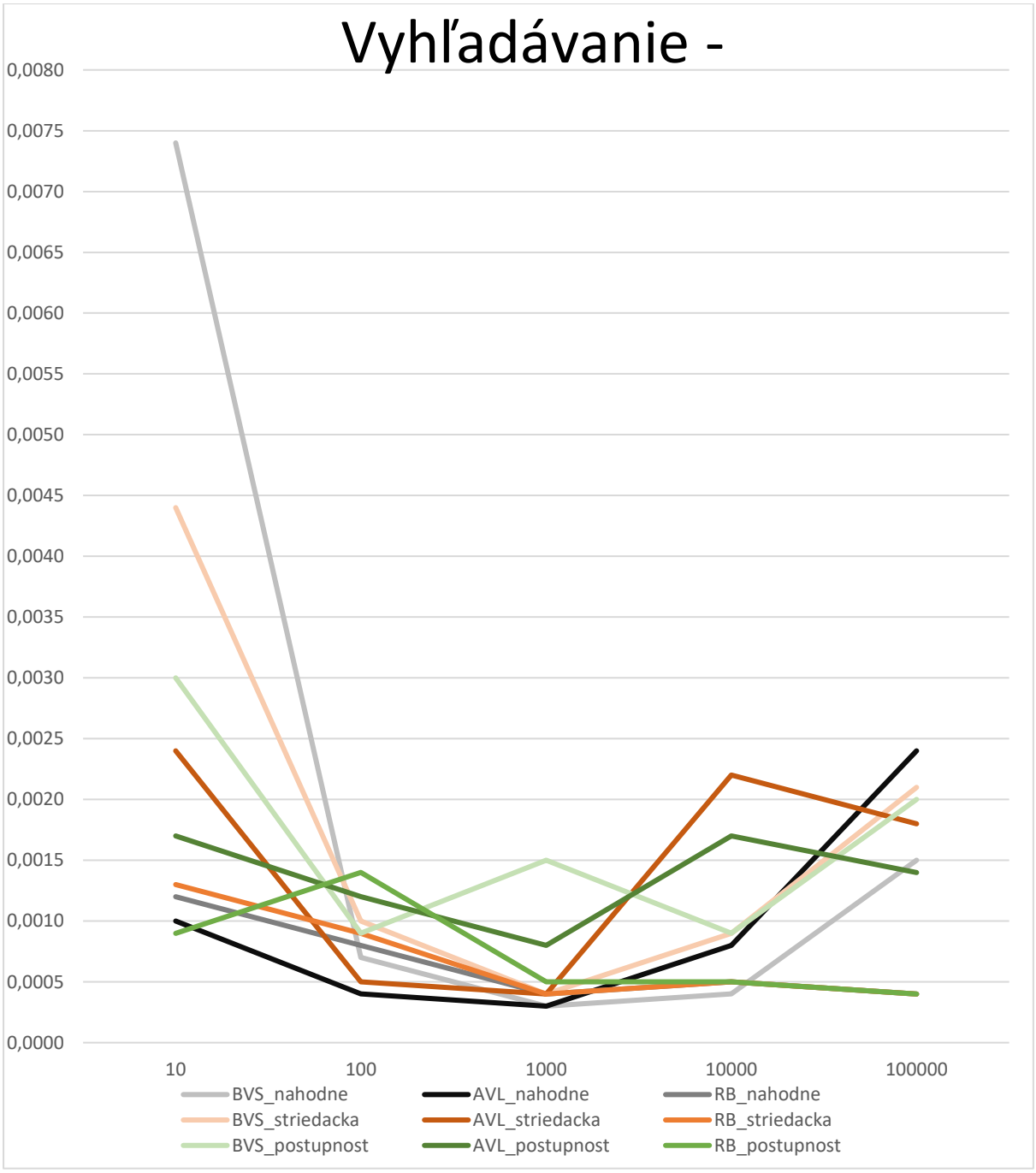
Výsledky testov stromy: (udaje v tabulkach su v sekundach)

Stromy - Nahodne cisla								
TEST	BVS			AVL			RB	
	Insert 10		Search	Insert 10		Search	Insert 10	Search
Priemer	0		0,0074	0		0,001	0,0001	0,0012
	Insert 100		Search	Insert 100		Search	Insert 100	Search
Priemer	0		0,0007	0,0003		0,0004	0,0003	0,0008
	Insert 1 000		Search	Insert 1 000		Search	Insert 1 000	Search
Priemer	0,001		0,0003	0,0018		0,0003	0,0008	0,0004
	Insert 10 000		Search	Insert 10 000		Search	Insert 10 000	Search
Priemer	0,011		0,0004	0,0193		0,0008	0,012	0,0005
	Insert 100 000		Search	Insert 100 000		Search	Insert 100 000	Search
Priemer	0,1355		0,0015	0,2274		0,0024	0,1382	0,0004

Stromy - Vkladanie na "striedacku"								
TEST	BVS			AVL			RB	
	Insert 10		Search	Insert 10		Search	Insert 10	Search
Priemer	0		0,0044	0		0,0024	0	0,0013
	Insert 100		Search	Insert 100		Search	Insert 100	Search
Priemer	0		0,001	0,0002		0,0005	0,0002	0,0009
	Insert 1 000		Search	Insert 1 000		Search	Insert 1 000	Search
Priemer	0,0009		0,0004	0,0018		0,0004	0,001	0,0004
	Insert 10 000		Search	Insert 10 000		Search	Insert 10 000	Search
Priemer	0,011		0,0009	0,0198		0,0022	0,0117	0,0005
	Insert 100 000		Search	Insert 100 000		Search	Insert 100 000	Search
Priemer	0,1485		0,0021	0,2233		0,0018	0,1377	0,0004

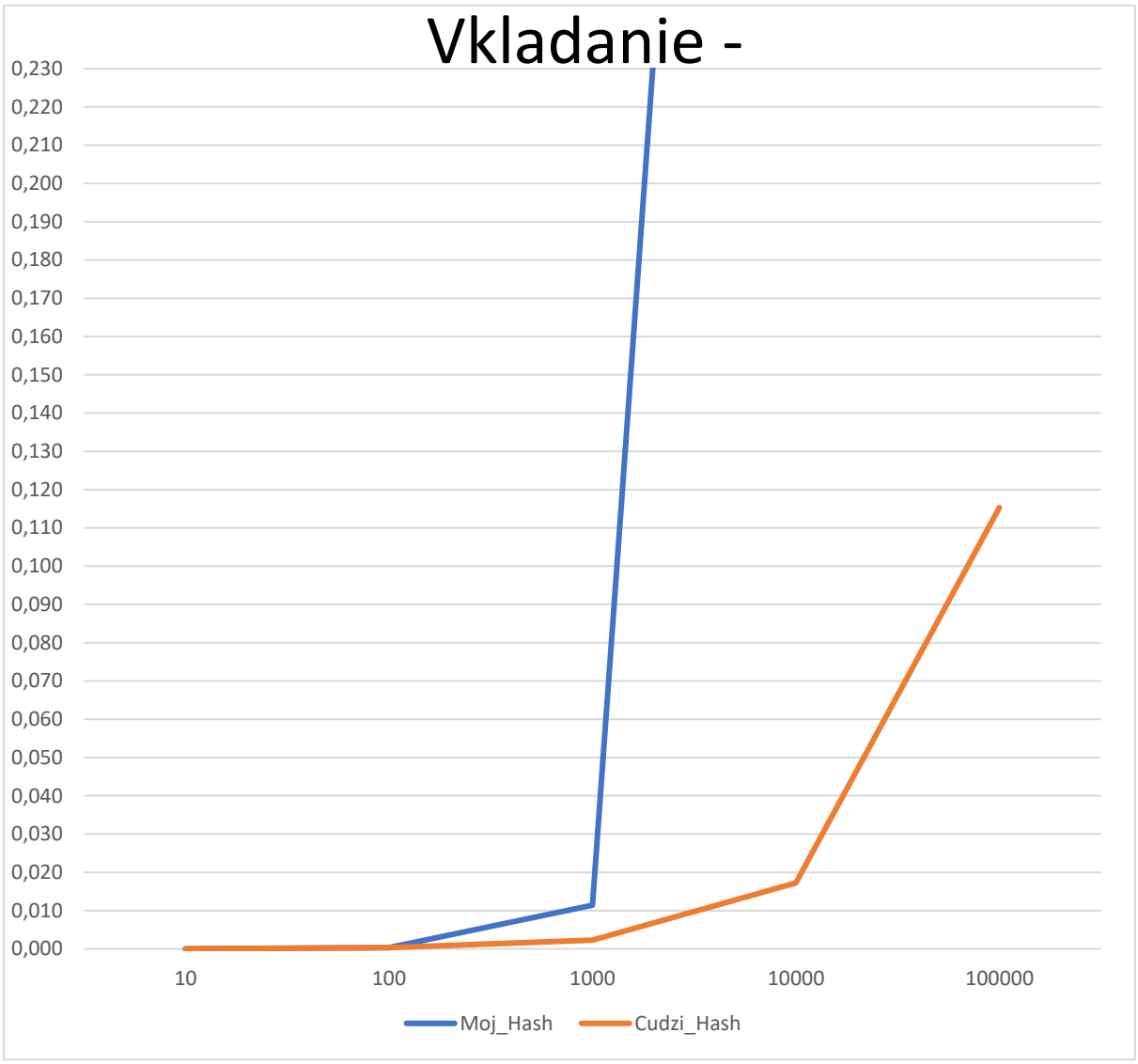
Stromy - Postupnosti								
TEST	BVS			AVL			RB	
	Insert 10		Search	Insert 10		Search	Insert 10	Search
Priemer	0		0,003	0		0,0017	0	0,0009
	Insert 100		Search	Insert 100		Search	Insert 100	Search
Priemer	0,0001		0,0009	0,0002		0,0012	0,0001	0,0014
	Insert 1 000		Search	Insert 1 000		Search	Insert 1 000	Search
Priemer	0,0041		0,0015	0,0016		0,0008	0,0009	0,0005
	Insert 10 000		Search	Insert 10 000		Search	Insert 10 000	Search
Priemer	0,3266		0,0009	0,0189		0,0017	0,0109	0,0005
	Insert 100 000		Search	Insert 100 000		Search	Insert 100 000	Search
Priemer	32,6083		0,002	0,1969		0,0014	0,1249	0,0004

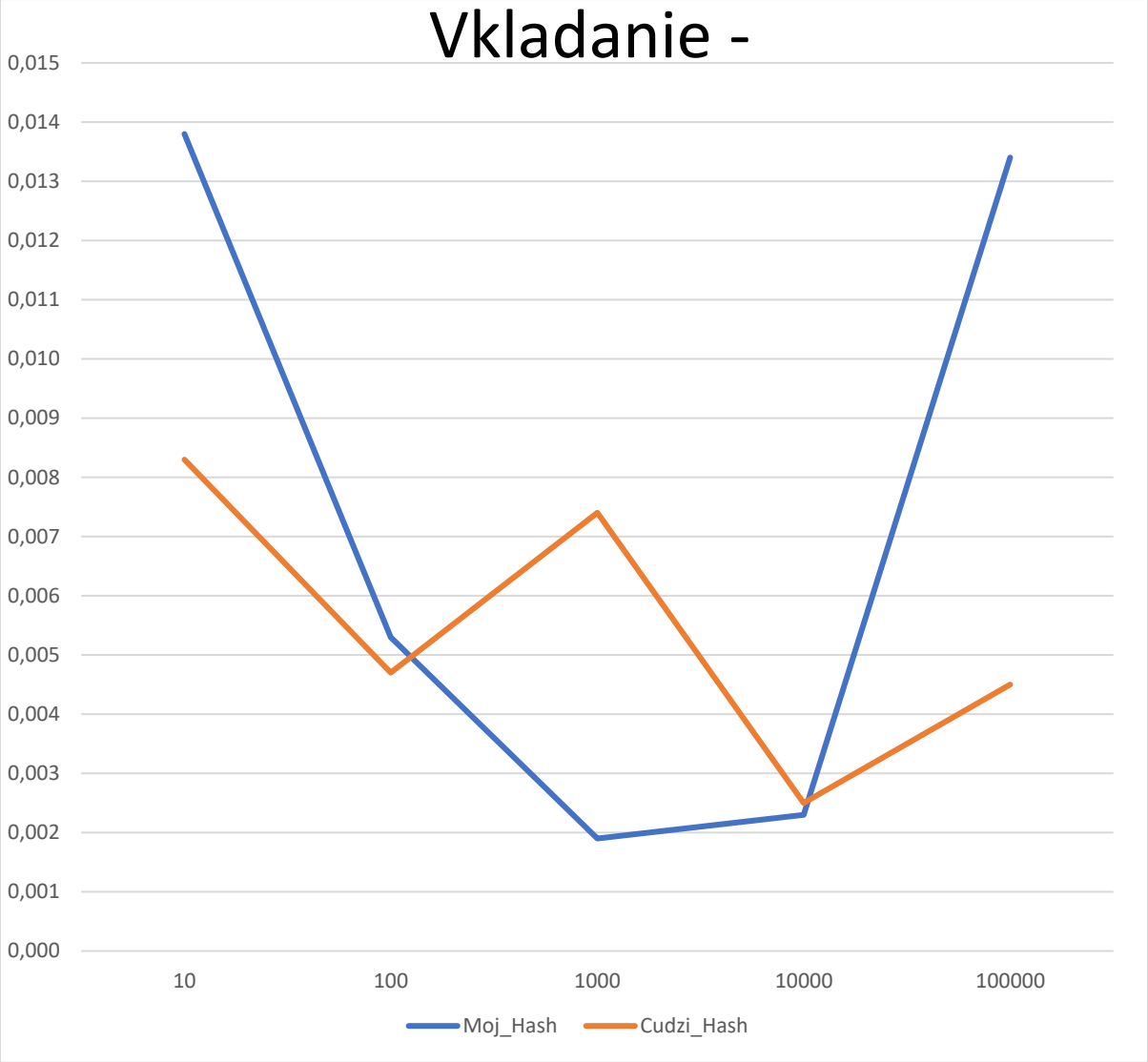

Vkladanie - stromy
0,300
0,290
0,280
0,270
0,260
0,250
0,240
0,230
0,220
0,210
0,200
0,190
0,180
0,170
0,160
0,150
0,140
0,130
0,120
0,110
0,100
0,090
0,080
0,070
0,060
0,050
0,040
0,030
0,020
0,010
0,000
10
100
1000
10000
100000
BVS_nahodne
AVL_nahodne
RB_nahodne
BVS_striedacka
AVL_striedacka
RB_striedacka
BVS_postupnost
AVL_postupnost
RB_postupnost


Vyhľadávanie -
0,0080
0,0075
0,0070
0,0065
0,0060
0,0055
0,0050
0,0045
0,0040
0,0035
0,0030
0,0025
0,0020
0,0015
0,0010
0,0005
0,0000
10
100
1000
10000
100000
BVS_nahodne
AVL_nahodne
RB_nahodne
BVS_striedacka
AVL_striedacka
RB_striedacka
BVS_postupnost
AVL_postupnost
RB_postupnost

Výsledky testov hashovacie funkcie: (udaje v tabulkach su v sekundach)

Hashovanie - nahodne slova							
TEST	MOJ HASH				CUDZI HASH		
	Insert 10		Search		Insert 10		Search
Priemer	0		0,0138		0		0,0083
	Insert 100		Search		Insert 100		Search
Priemer	0,0003		0,0053		0,0003		0,0047
	Insert 1 000		Search		Insert 1 000		Search
Priemer	0,0114		0,0019		0,0023		0,0074
	Insert 10 000		Search		Insert 10 000		Search
Priemer	0,7491		0,0023		0,0172		0,0025
	Insert 100 000		Search		Insert 100 000		Search
Priemer	68,8085		0,0134		0,1153		0,0045


Vkladanie -
0,230
0,220
0,210
0,200
0,190
0,180
0,170
0,160
0,150
0,140
0,130
0,120
0,110
0,100
0,090
0,080
0,070
0,060
0,050
0,040
0,030
0,020
0,010
0,000
10
100
1000
10000
100000
Moj_Hash
Cudzi_Hash



Zhodnotenie

Zložitosti:

- Priestorová zložitosť samotných stromov/hashovacích tabuliek (64-bitový počítač)
  - BVS - O(počet prvkov * veľkosť štruktúry (1 * int + 3 * smerník = 4 + 3 * 8 = 28)) -> O(počet prvkov * 28)
  - AVL - O(počet prvkov * veľkosť štruktúry (3 * int + 3 * smerník = 3 * 4 + 3 * 8 = 36)) -> O(počet prvkov * 36)
  - ČČ - O(počet prvkov * veľkosť štruktúry (2 * int + 3 * smerník = 2 * 4 + 3 * 8 = 32)) -> O(počet prvkov * 32)
  - Môj hash - O(2 * int + smerník na tabuľku(8) + veľkosť tabuľky(veľkosť štruktúry ELEMENT * počet prvkov = (int + x) * počet prvkov = (4 + x) * počet prvkov)) -> O(12 + (4 + x) * počet prvkov)
    - x je veľkosť uloženého slova v prvku (v danej zložitosti počítam s tým, že každé slovo je rovnako dlhé)

- Cudzí hash - O(2 * long + smerník na tabuľku(8) + veľkosť tabuľky(veľkosť štruktúry ELEMENT * počet prvkov = (int + long + x) * počet prvkov = (4 + 8 + x) * počet prvkov)) -> O(24 + (12 + x) * počet prvkov)
  - x je veľkosť uloženého slova v prvku (v danej zložitosti počítam s tým, že každé slovo je rovnako dlhé)

- Časová zložitosť vkladania (64-bitový počítač) (n je počet prvkov v strome)
  - BVS - záleží, cez koľko prvkov sa bude presúvať funkcia, môže to byť od O(1) (vloženie koreňa) po O(n) (vkladanie postupnosti čísel)
  - AVL - záleží, cez koľko prvkov sa bude presúvať funkcia, môže to byť od O(1) (vloženie koreňa) po O(log n) (časová zložitosť rotácii sa neberie do úvahy, tak sa označuje len ako O(1))
  - ČČ – O (log n)
  - Môj hash - môže to byť od O(1) (index z prvej hashovacej funkcie bude v tabuľke voľný) po O(n)
  - Cudzí hash - môže to byť od O(1) (index z hashovacej funkcie bude v tabuľke voľný) po O(n) (bude hľadať po krokoch a prejsť celú tabuľku maximálne nx)

- Časová zložitosť vyhľadávania (64-bitový počítač) (je počet prvkov v strome) (rovnaké ako pri vkladaní)
  - BVS - záleží, cez koľko prvkov sa bude presúvať funkcia, môže to byť od O(1) (vloženie koreňa) po O(n) (vkladanie postupnosti čísel)
  - AVL - záleží, cez koľko prvkov sa bude presúvať funkcia, môže to byť od O(1) (vloženie koreňa) po O(n) (vkladanie postupnosti čísel) + treba pripočítať časovú zložitosť rotácie... priraďme jednej rotácii O(1) a pri vkladaní môže dochádzať k niekoľko rotáciám
  - ČČ – O (log n)
  - Môj hash - môže to byť od O(1) (index z prvej hashovacej funkcie bude v tabuľke voľný) po O(n)
  - Cudzí hash - môže to byť od O(1) (index z hashovacej funkcie bude v tabuľke voľný) po O(n) (bude hľadať po krokoch a prejsť celú tabuľku maximálne n-krát)

Najlepšia stromová štruktúra na vkladanie je Červeno-čierny strom. Najhoršie je na tom nevyvážený BVS. AVL strom je taký zlatý stred pre vkladanie a vyhľadávanie.

Moja hashovacia implementácia je omnoho časovo náročnejšia ako cudzia implementácia a to kvôli reálnemu riešeniu kolízii kľúčov, pričom cudzia

prepisuje dáta, ak sú kľúče zhodné.