

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
Fakulta informatiky a informačných technológií  
Ilkovičová 2, 842 16 Bratislava 4

# Zadanie 2 – Vyhľadávanie v dynamických množinách

ADRIÁN VANČO  
ID: 103171

Cvičenie: Utorok 18:00  
5.4.2021

## Úvod

Existuje veľké množstvo algoritmov, určených na efektívne vyhľadávanie prvkov v dynamických množinách: binárne vyhľadávacie stromy, viaceré prístupy k ich vyvažovaniu, hašovanie a viaceré prístupy k riešeniu kolízií. Rôzne algoritmy sú vhodné pre rôzne situácie podľa charakteru spracovaných údajov, distribúcií hodnôt, vykonávaným operáciám, a pod.

V tomto zadaní som porovnal AVL vyvažovací strom, červeno-čierny vyvažovací strom, dvojité hašovanie s otvorenou adresáciou a hašovanie, ktoré rieši kolízie pomocou zreťazenia.

## Vlastná implementácia AVL stromu

AVL strom je samovyvažovací binárny vyhľadávací strom. V AVL strome sa pre každý uzol rozdiel výšky dvoch podstromov detských uzlov líšia najviac o jednotku, preto je známy aj ako výškovo vyvážený. Hľadanie a vkladanie majú zložitosť  $O(\log n)$  v priemernom aj najhoršom prípade. Pridávanie môže vyžadovať vyváženie stromu jednou alebo viacerými rotáciami stromu. Koeficient vyváženia sa počíta podľa vzorca:

```
//ziskanie faktoru na balancovanie
int getBalanceFactor(NODEAVL* node){
    return height(node->right) - height(node->left);
}
```

Oproti obyčajnému stromu si vo vrchole drží aj maximálnu hĺbku podstromu. Táto hĺbka sa využíva pri rotovaní stromu. Ak je vrchol, na ktorom sa nachádzam, nevyvážený (to znamená, že jeden podstrom je o dva hlbší ako druhý), začínam rotovať. Celkovo rozoznávam štyri prípady. Ak je nevyvážený doprava a zároveň aj jeho pravý potomok je nevyvážený doprava, tak rotujem doľava. Ale ak je potomok nevyvážený doľava, tak najskôr ten musím zarotovať doprava a až potom jeho rodiča doľava. Toto isté platí symetricky aj pre druhú stranu. AVL strom je vždy, čo najlepšie vyvážený, čo spôsobuje, že funkcia insert je pomalšia ako pri iných implementáciách. Avšak funkcia search by mala byť rýchlejšia, pretože strom je vždy čo najlepšie vyvážený na rozdiel od červeno-čierneho stromu.

Používam 1 globálnu premennú a štruktúru pre reprezentáciu AVL prvku.

```
NODEAVL* root = NULL;

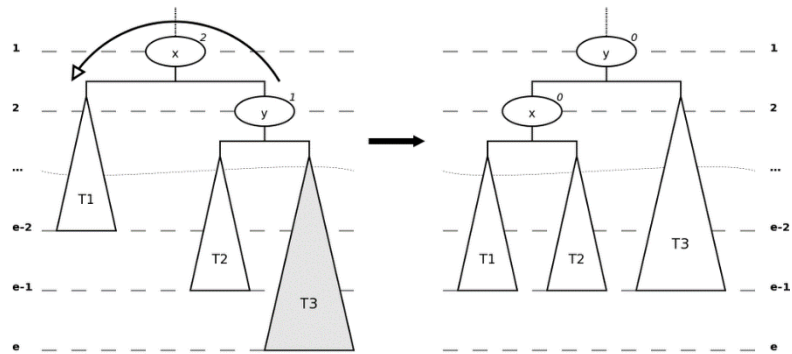
typedef struct NodeAVL {
    int data; //hodnota, ktoru chcem uložiť
    int height; //vyska prvku
    struct NodeAVL* left;
    struct NodeAVL* right;
}NODEAVL;
```

## Zoznam funkcií

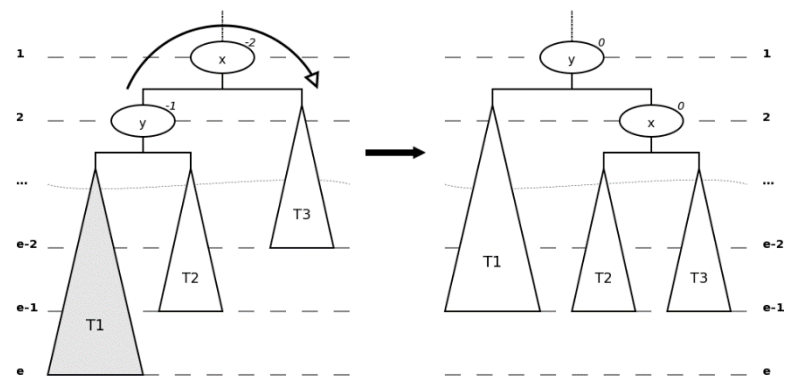
```
NODEAVL* createNODEAVL(int data); //vrati vytvoreny prvok
int height(NODEAVL* node); //vrati vysku prvku
int maximum(int a, int b); //vrati vacsie cislo
int getBalanceFactor(NODEAVL* node); //vrati hodnotu faktoru balancovania
NODEAVL* leftRotate(NODEAVL* node);
NODEAVL* rightRotate(NODEAVL* node);
NODEAVL* insertAVLTree(NODEAVL* node, int data); //vlozi do avl stromu
NODEAVL* searchAVLTree(NODEAVL* node, int data); //vyhlada v avl strome
void inOrder(NODEAVL* node); //vypis stromu
void freeAVL(NODEAVL** node); //uvolnenie stromu
```

## Vizualizácia rotácií

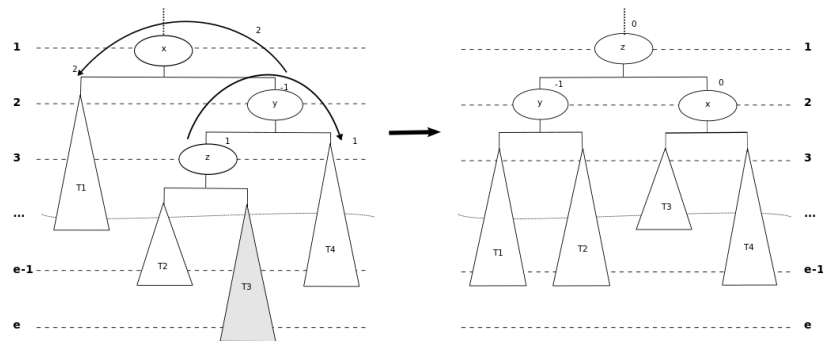
Ľavá rotácia



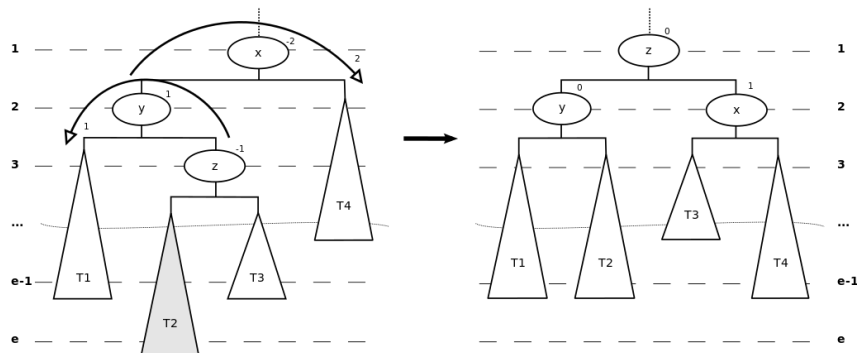
Pravá rotácia



Pravá a ľavá rotácia



Ľavá a pravá rotácia



## Prevzatá implementácia Červeno-čierneho stromu

Je to ďalší algoritmus na vyvažovanie binárneho stromu, taký že:

1. Koreň je **čierny**.
2. Listy neobsahujú dáta a sú **čierne** (v tejto implementácii sú listy **NULL**).
3. Cesty z koreňa do listov majú rovnaký počet čiernych vrcholov a tento počet označujeme **čierna výška** stromu.
4. Ak je vrchol **červený**, tak jeho deti sú **čierne**.

Vlastnosti:

- Na žiadnej ceste nie sú dva **červené** vrcholy za sebou.
- Dĺžka cesty z koreňa do najvzdialenejšieho listu nie je viac ako dvakrát dlhšia ako cesta do najbližšieho listu.
- Každý vnútorný vrchol má dvoch potomkov

Algoritmus poskytuje oproti AVL stromu rýchlejšie operácie vkladania a vyberania hoci má rovnakú zložitosť  $O(\log n)$  ako AVL strom, pretože sa robí menej rotácií kvôli relatívne voľnejšiemu vyvažovaniu. Avšak poskytuje pomalšie vyhľadávanie, keďže môže byť horšie vyvážený oproti AVL stromu. Túto implementáciu som prevzal od Amit Bansal [1].

### Zoznam funkcií

```
void levelorder(struct Node* root);    //vypis stromu
struct Node* RB_insert(struct Node* T, int data); //vkladanie do stromu
void preorder(struct Node* root);    //vypis stromu
struct Node* RB_delete(struct Node* T, struct Node* z); //vymazanie zo stromu
struct Node* BST_search(struct Node* root, int x); //vyhladavanie v strome
```

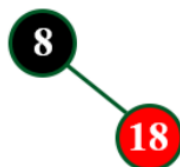
### Príklad

Vytvoríme červeno-čierny strom s postupným vkladáním čísel 8, 18, 5, 15, 17, 25, 40, 80.

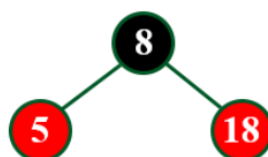
1. Vloženie 8 – strom je prázdny, takže 8 bude koreň a bude čierny



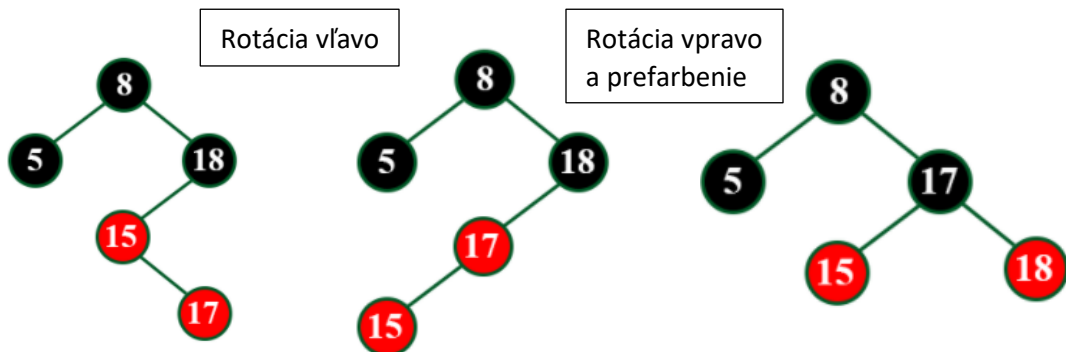
2. Vloženie 18 – strom nie je prázdny, takže 18 bude červený.



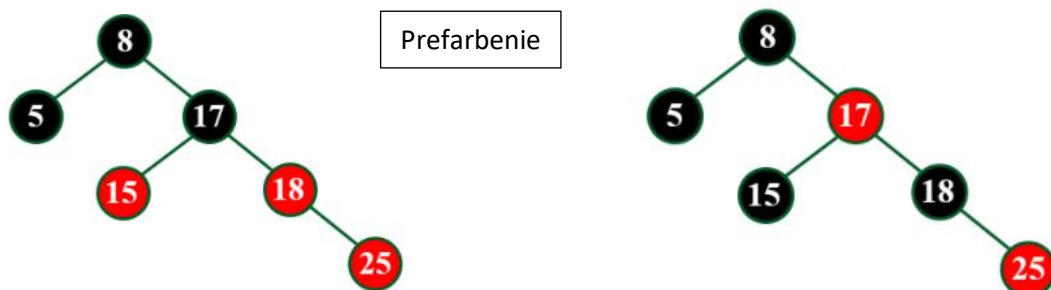
3. Vloženie 5 - strom nie je prázdny, takže 5 bude červený.



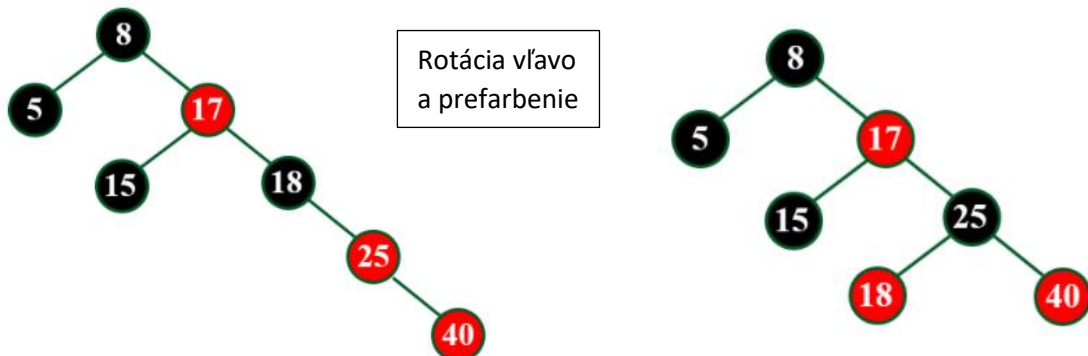
4. Vloženie 17 - strom nie je prázdny, takže 17 bude červený. Tu nastáva prípad kedy sú dva červené prvky za sebou, takže potrebujem rotovať a pri rotácii aj prefarbovať. Tu sa konkrétne uplatní rotácia doľava a doprava.



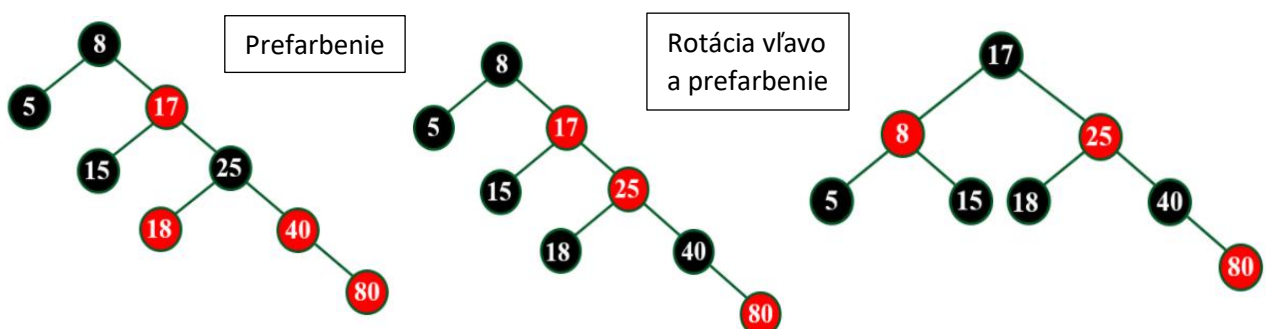
5. Vloženie 25 – strom nie je prázdny, takže 25 bude červený. Tu nastáva prípad kedy sú dva červené prvky za sebou. Rodič (18) je červený a rodič rodiča(17) nie je koreň, takže ho môžeme prefarbiť.



6. Vloženie 40 – strom nie je prázdny, takže 40 bude červený. Rotácia vľavo a prefarbenie.



7. Vloženie 80 – strom nie je prázdny, takže 40 bude červený.



## Vlastná implementácia Hash tabuľky

Implementoval som si tabuľku, ktorá rieši kolízie zreťazením. Veľkosť tabuľky je prvočíslo. Je to preto, lebo prvočísla majú menej súdeliteľných čísel a teda nastáva menej kolízií. Táto implementácia má jednu hašovaciu funkciu, ktorá vracia index v poli pre vkladané číslo. Ak na indexe v tabuľke sa nič nenachádza, vloží prvok, ale ak sa tam nachádza už list prvkov s rovnakým indexom program skontroluje či sa vkladané číslo už nachádza v liste. Ak sa nachádza tak už nevkladáme, ak nie tak vkladany prvok vloží na začiatok zoznamu. Čo sa týka pamäťovej zložitosti, tak tá je nevýhodou oproti metóde otvorenej adresácie.

Zložitosť algoritmu pre vkladanie je  $O(n)$ , kde  $n$  je počet prvkov v liste pokiaľ nenastane potreba zväčšenia tabuľky, potom by bola  $O(k * n + p + r)$ , kde  $k$  je počet prvkov v tabuľke,  $n$  je počet prvkov v liste,  $p$  je počet iterácií na nájdenie prvočísla a  $r$  je počet prvkov, v starej tabuľke, ktoré sa uvoľnia.

Tabuľka sa zväčšuje pri faktore naplnenia  $\alpha \geq 0.1$ , ktorý sa vypočíta ako podiel počtu prvkov na indexe a veľkosti tabuľky.

Zložitosť algoritmu pre hľadanie je  $O(n)$ , kde  $n$  je počet prvkov v liste.

Používam 2 globálne premenné a 2 štruktúry pre reprezentáciu tabuľky.

```
ARRAYNODE* array;    //ukazovatel na tabulku
int arraySize;        //velkost tabulky

//bunka tabulky
typedef struct ArrayNode {
    int count;
    NODE* list;
}ARRAYNODE;

//bunka listu
typedef struct Node {
    int data;
    struct Node* next;
}NODE;
```

## Zoznam funkcií

```
void initLinkedHash(int size);        //vytvori tabulku
int sizeofArrayLinkedHash();          //vrati velkosti tabulky
void insertLinkedHash(int data);      //vlozi do tabulky
void freeOldLinkedHash(ARRAYNODE** array, int size); //uvolni staru tabulku
void freeLinkedHash();                //uvolni tabulku
int* searchLinkedHash(int data);      //vyhlada data v tabulke
bool isPrimeLinkedHash(int number);   //zisti ci cislo je prvocislo
int getNextPrimeLinkedHash(int number); //vrati najblizsie prvocislo na pravo
void rehashLinkedHash();              //zvacsi tabulku
int hashLinkedHash(int x);            //zahašuje hodnotu a vrati poziciu v tabulke
void printLinkedHash();               //vypise celu tabulku
```

## Hash funkcia

Funkcia vracia vypočítaný index podľa  $(a * x + b) \% \text{arraySize}$ . „a“ a „b“ sú ľubovoľne zvolené čísla a arraySize je veľkosť tabuľky a je prvočíslo pre lepší rozptyl.

```
int hashLinkedHash(int x) {
    int a = 7;
    int b = 3;
    return(abs(a*x + b) % arraySize);
}
```

## Zväčšovanie tabuľky

Zväčšovanie som riešil tak, že si odložím adresu začiatku starej tabuľky do ukazovateľa temp, ktorú idem zväčšovať, uloží si aj jej veľkosť, pretože po prenesení dát do novej tabuľky, starú tabuľku uvoľním. Novú veľkosť sa vypočíta cez implementovanú funkciu, ktorá vráti prvočíslo, a tu už nastáva zväčšovanie časovej náročnosti, ktorá by sa dala ešte vylepšiť. Napríklad vylepšenie algoritmu na hľadanie prvočísla, alebo viac zväčšiť tabuľku aby sa nemusela častejšie zväčšovať.

```
void rehashLinkedHash() {
    ARRAYNODE* temp = array;
    NODEDATA* list = NULL;
    int oldSize = sizeofArrayLinkedHash(); //ziskanie velkosti aktualnej tabulky
    int newSize = getNextPrimeLinkedHash(2 * oldSize); //ziskanie novej velkosti (velkost bude vzdy prvocislo)
    initLinkedHash(newSize); //inicializovanie novej tabulky

    //presuvanie hodnot zo starej tabulky do novej vacsej tabulky
    for (int i = 0; i < oldSize; i++) {
        list = (temp + i)->list;

        if (list == NULL) {
            continue;
        }
        else {
            while (list != NULL) {
                insertLinkedHash(list->data);
                list = list->next;
            }
        }
    }

    freeOldLinkedHash(&temp, oldSize); //uvolnenie starej tabulky
}
```

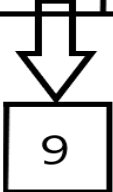
## Vizuálna reprezentácia

Majme tabuľku o veľkosti 11, a chceme vložiť číslo 9 do tabuľky, zavoláme hash funkciu, ktorá vráti hodnotu indexu kam sa má hodnota uložiť, index je 0 a na indexe je prázdny list, tak sa vloží na začiatok.

0	1	2	3	4	5	6	7	8	9	10
9										

Následne vkladáme čísla 11 a 20. Číslo 11 sa vloží podobne ako číslo 9. A pri čísle 20 sme dostali od hash funkcie index 0 a na indexe 0 sa už nachádza list prvkov tak tento list napojím na vkladany prvok a prvok vložím na index.

0	1	2	3	4	5	6	7	8	9	10
20			11							



## Prevzatá implementácia otvorenej adresácie

V otvorenej adresácii v políčku tabuľky môže byť najviac jeden prvok. V tejto implementácii je dvojité hašovanie a teda ak nastane kolízia prichádza na rad druhá hašovacia funkcia ktorá, ktorá index posunie o danú hodnotu. Zložitosť vkladania v prípade kolízie je  $O(n)$ , v prípade, že bude musieť program prejsť celý zoznam. Preto sa tu tiež využíva faktor naplnenia, ten je v implementácii nastavený na polovičné naplnenie, teda keď počet prvkov dosiahne polovicu veľkosti tabuľky, vytvorí sa väčšia tabuľka do ktorej sa prenesú všetky prvky z predchádzajúcej a predchádzajúca sa uvoľní to zaberie približne  $O(n)$ .

Vyhľadávanie v tejto implementácii má zložitosť  $O(n)$ , kde  $n$  je počet prvkov s rovnakým hašom.

Túto implementáciu som prevzal od Matej Delincak [2].

```
typedef struct hashTable {  
    int countOfNumb;    //pocet prvkov v tabulke  
    int maxSize;  
    int primeSmall; //najblizsie prvocislo mensie ako velkost tabulky  
    int* newArr;  
}HASHTABLE;
```

### Zoznam funkcií

```
void insertDoubleHash(HASHTABLE** paTable, int paVal);  
void printDoubleHash(HASHTABLE* paTable);  
int* searchDoubleHash(HASHTABLE** paTable, int paVal);  
void deleteDoubleHash(HASHTABLE** paTable);
```

### Príklad

Majme tabuľku o veľkosti 11, a chceme vložiť číslo 9 do tabuľky, zavoláme hash funkciu, ktorá vráti hodnotu indexu kam sa má hodnota uložiť, index je 0 a na indexe nie je nič, tak sa vloží na index.

0	1	2	3	4	5	6	7	8	9	10
9										

Ako ďalšie chceme vložiť číslo 20. zavoláme hash funkciu, ktorá vráti hodnotu indexu, kam sa má hodnota uložiť, index je 0 a na indexe 0 už je číslo 9, tak sa zavolá druhá hash funkcia, ktorá vráti hodnotu o ktorú sa má prvok posunúť v tomto príklade nám druhá hash funkcia vrátila hodnotu 3 a teda pozrieme sa na index  $0 + 3$  a vidíme že index 3 je voľný tak vložíme, keby nebol znova sa volá druhá hash funkcia až kým nedostaneme voľný index kam sa prvok môže uložiť.

0	1	2	3	4	5	6	7	8	9	10
9			20							



## Testovanie

Pre testovanie som vytvoril program, ktorý ma v sebe zahrnuté implementácie a obsahuje funkciu na vygenerovanie poľa čísel so zvolenou postupnosťou a funkciami pre testovanie každej implementácie na vkladanie a vyhľadávanie.

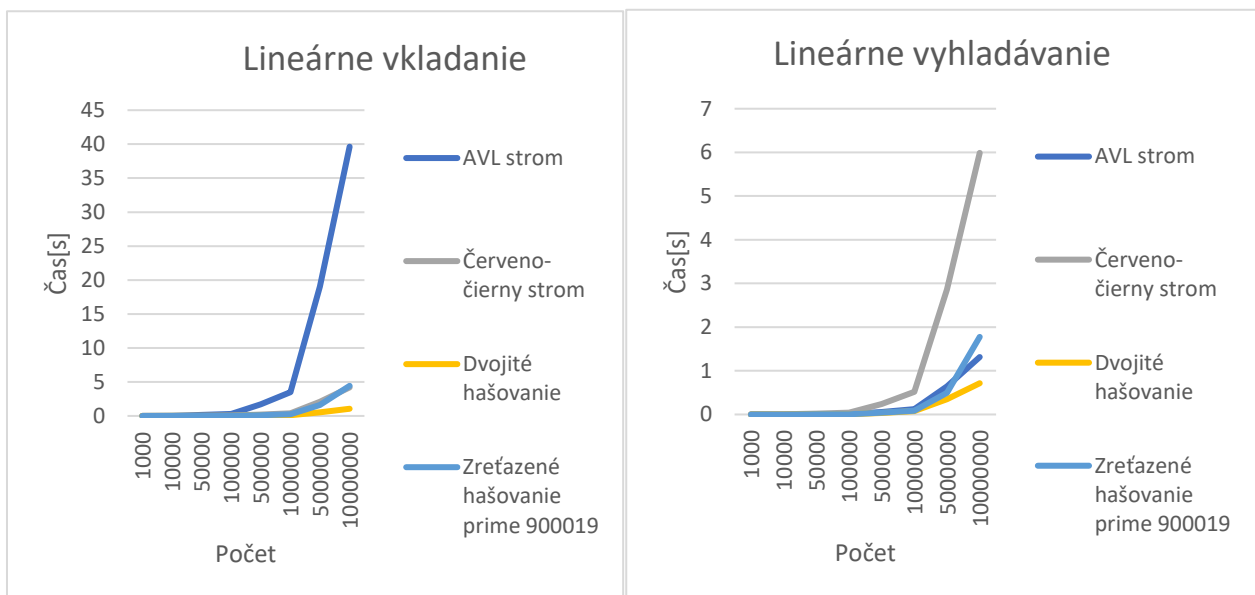
```
void testAVL(int* numbers, int count, int type);
void testRB(int* numbers, int count, int type);
void testLinkedHash(int tableSize, int* numbers, int count, int type);
void testDoubleHash(int* numbers, int count, int type);
int* generateNumbers(int count, int type);
```

V testovacích scenároch na efektivitu som sa zamerlal na tri kategórie:

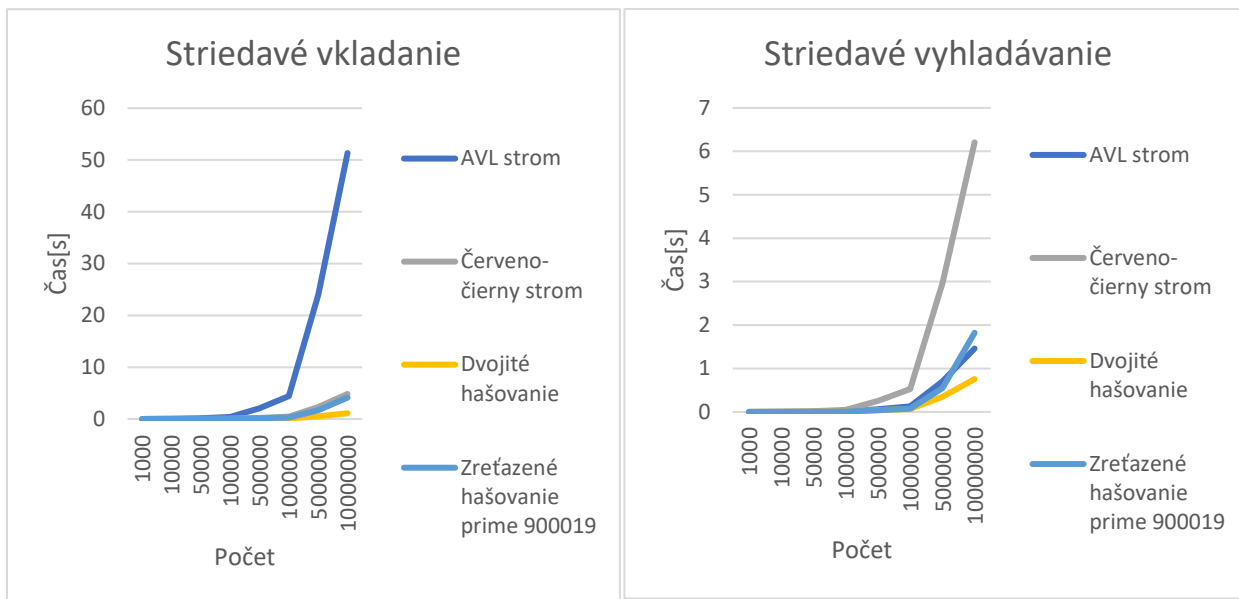
- Lineárna postupnosť čísel od 0 po N-1
- Striedavá postupnosť čísel: 0, N-1, 1, N-2,...
- Náhodná postupnosť čísel

Tieto vstupy som testoval na mojich a prevzatých implementáciách na vkladanie a na vyhľadávanie.

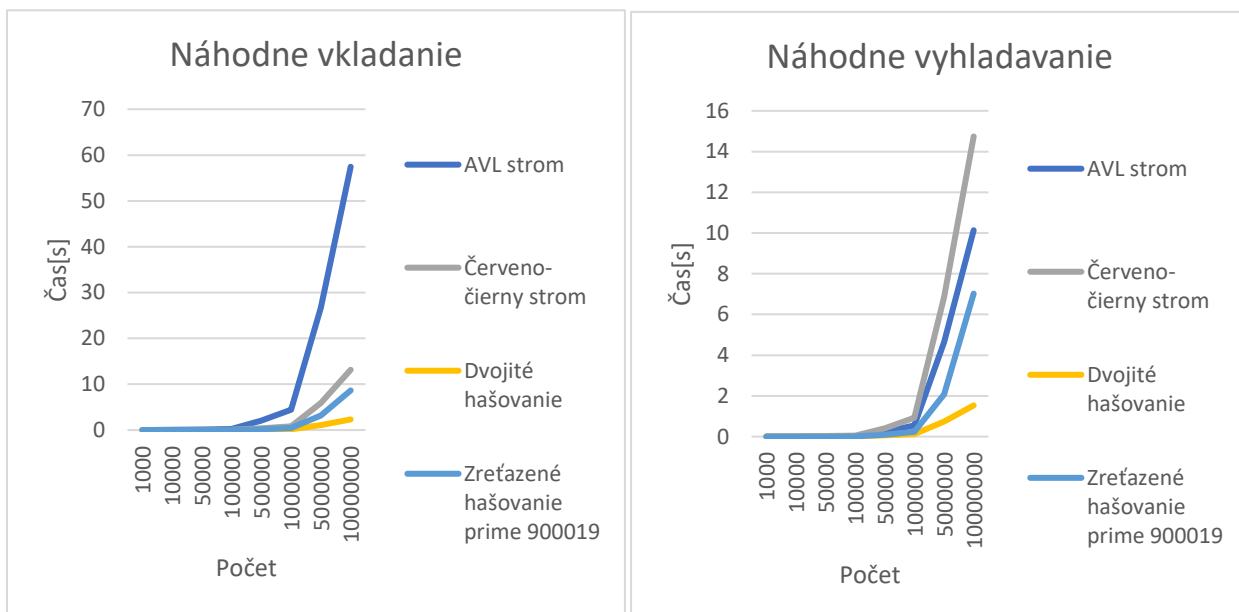
## Lineárna postupnosť



## Striedavá postupnosť



## Náhodná postupnosť



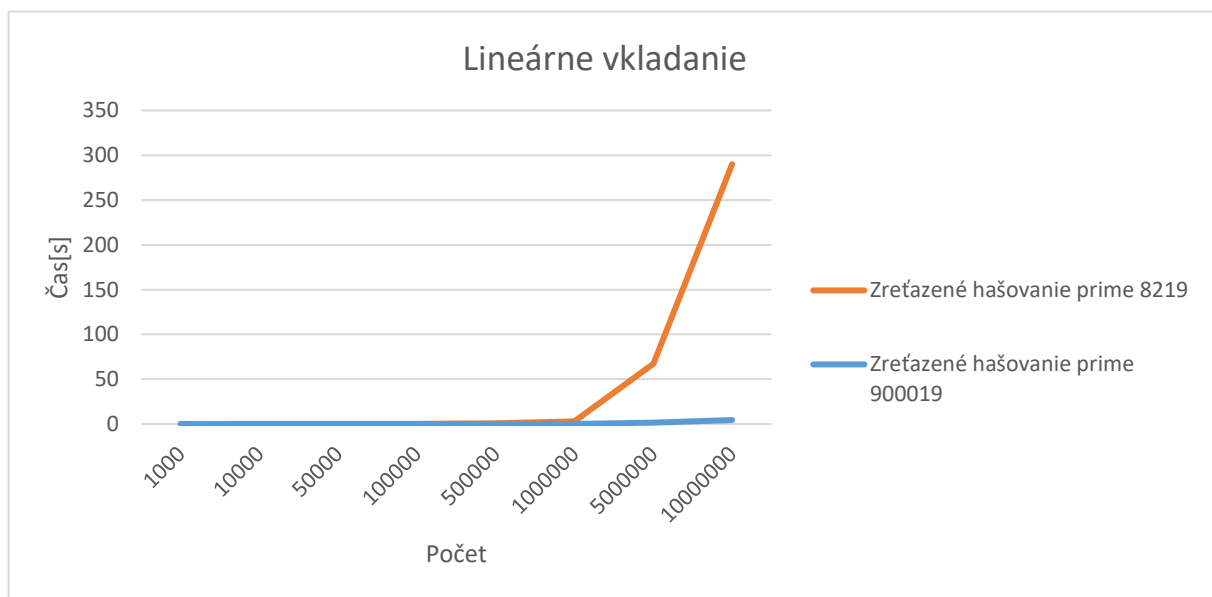
## Zhodnotenie

Pri lineárnej a striedavej postupnosti vkladanie trvalo najdlhšie do AVL stromu, najrýchlejšie do tabuľky s otvorenou adresáciou a dvojitém hašovaním, červeno-čierny strom a hašovacia tabuľka s reťazením boli o niečo pomalšie ako tabuľka s otvorenou adresáciou a dvojitém hašovaním. Pri vkladaní 10M prvkov bol rozdiel medzi AVL a tabuľkou s otvorenou adresáciou a dvojitém hašovaním cez 50 sekúnd.

Pri lineárnej a striedavej postupnosti vyhľadávanie trvalo najdlhšie tento krát červeno-čiernemu stromu a najrýchlejšie tabuľke s otvorenou adresáciou a dvojitém hašovaním. AVL strom a hašovacia tabuľka s reťazením boli o niečo pomalšie ako tabuľka s otvorenou adresáciou a dvojitém hašovaním. Pri vyhľadávaní 10M prvkov bol rozdiel medzi červeno-čiernym stromom a tabuľkou s otvorenou adresáciou a dvojitém hašovaním okolo 5 sekúnd.

Pri náhodnej postupnosti to je zaujímavejšie pretože lebo sa približuje skutočnému využitiu. Pri vkladaní poradia implementácií je rovnaké ako v lineárnej a striedavej postupnosti ale pri vkladaní 10M prvkov bol rozdiel medzi AVL a tabuľkou s otvorenou adresáciou a dvojitém hašovaním cez 55 sekúnd. Pri náhodné vyhľadávaní dopadla najlepšie tabuľkou s otvorenou adresáciou a dvojitém hašovaním za ňou nasledovala tabuľka s reťazením, potom AVL strom a posledný skončil červeno čierny strom.

Ešte pri tabuľke s reťazením je dôležité efektívne a menej často zväčšovať tabuľku. Nakoľko časté zväčšovanie zvyšuje časovú zložitosť. Je to vidieť na nasledujúcom porovnaní.



## Zdroje prevzatej implementácie

- [1] <https://github.com/amitbansal7/Data-Structures-and-Algorithms/blob/master/9.Red-Black-tree/RedBlackTrees.c>
- [2] [https://github.com/mateju25/DSA\\_Zadanie2/blob/master/DoubleHash.c](https://github.com/mateju25/DSA_Zadanie2/blob/master/DoubleHash.c)

Obrázky a príklad pre červeno-čierny strom prevzaté zo:

[http://www.btechsmartclass.com/data\\_structures/red-black-trees.html](http://www.btechsmartclass.com/data_structures/red-black-trees.html)

Obrázky pre AVL rotácie prevzaté zo:

<https://cs.wikipedia.org/wiki/AVL-strom>