Meno : Richard Szarka
ID : 111958

Dátové štruktúry a algoritmy

Dokumentácia k zadaniu

Richard Szarka

Slovenské technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

xszarkar@stuba.sk

ID: 111958

Zadanie: 2

Cvičiaci: Mgr. Martin Sabo, PhD.

Meno : Richard Szarka
ID : 111958

Obsah

1 Zadanie 3
2 Binárne stromy 4
2.1 AVL samovyvažovací binárny strom 5
2.2 Moja implementácia AVL stromu 6
2.21 Knižnice makrá a štruktúra vrcholu 6
2.22 Inicializácia stromu 7
2.23 Vkladanie nových údajov do stromu 7
2.24 Vyvažovanie stromu 8
2.25 Vyhľadávanie prvkov 10
2.3 SPLAY strom 11
3 Hašovacie tabuľky 12
3.1 Lineárne sondovanie 12
3.2 Moja implementácia lineárneho sondovania 13
3.21 Knižnice, makrá, štruktúra tabuľky a štruktúra prvku v tabuľke 13
3.22 Inicializácia tabuľky 13
3.23 Hašovacia funkcia 14
3.24 Vkladanie prvkov do tabuľky 14
3.25 Zväčšovanie tabuľky 14
3.26 Vyhľadávanie prvkov v hašovacej tabuľke 15
3.3 Riešenie kolízii zreťazovaním 16
4 Testovanie 17
4.1 Testovanie rýchlosti 18
4.2 Testovanie počtu rotácii stromov (pri vkladaní všetkých prvkov aj vyhľadaní všetkých prvkov) 21
4.3 Testovanie počtu kolízií 22
4.31 Testovanie správnosti počtu kolízií 23
5 Záver 24

Meno : Richard Szarka
ID : 111958

1 Zadanie

V tomto zadaní máme za úlohu vytvoriť dva algoritmy na vyhľadávanie v dynamických množinách. Prvý z algoritmov je binárny strom. Druhý z algoritmov je hašovacia tabuľka. Vytvorené kódy/algoritmy následne porovnáme medzi sebou, a aj s inou prevzatou implementáciou daných algoritmov.

Binárny strom:

- Moja implementácia binárneho vyhľadávacieho stromu používa AVL (výškovo vyvážený strom) algoritmus na vyvažovanie .

- Prevzatý kód, ktorý porovnám s mojou implementáciou používa SPLAY strom algoritmus na vyvažovanie. (zdroj: https://www.codesdope.com/course/data-structures-splay-trees/)

Hašovacie tabuľky:

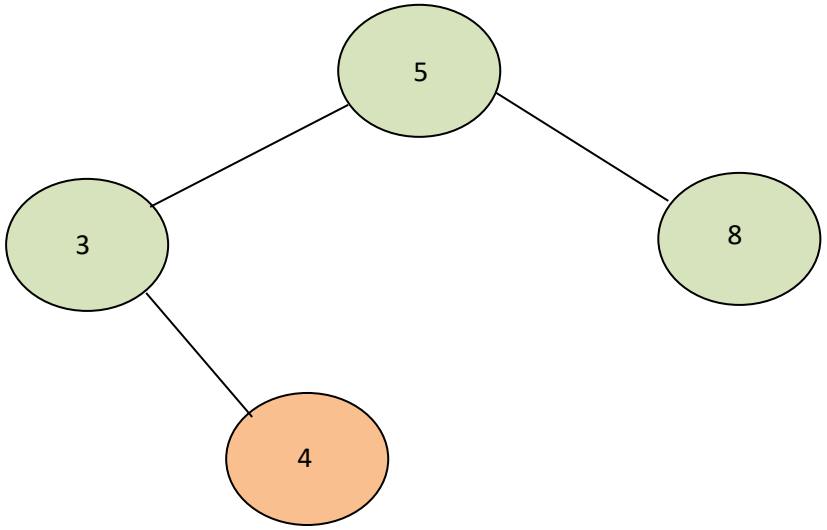
- Moja implementácia hašovacej tabuľky používa algoritmus linear probing na riešenie kolízií.

- Prevzatú implementáciu hašovacej tabuľky s ktorou porovnávam moju implementáciu používa algoritmus chaining. (zdroj: https://github.com/kuwairo/Hashtable)

Meno : Richard Szarka
ID : 111958

2 Binárne stromy

Binárne vyhľadávacie stromy sú jedným z najznámejších vyhľadávacích algoritmov na svete. Princíp je založený v tom, že každý vrchol v danom strome môže mať najviac troch „susedov“ (2 deti a 1 rodič). Vkladanie a usporiadanie vrcholov funguje tak, že porovná sa kľúč na danom navštívenom vrchole, s vkladaným kľúčom. Ak je hodnota vkladaného vrcholu menšia ako hodnota prehľadávaného, vkladaný vrchol ide navštíviť ľavé dieťa prehľadávaného vrcholu (ak ľavé dieťa neexistuje, vytvorí sa tam vrchol s vkladaným kľúčom). Ak je kľúč vkladaného vrcholu väčší ako kľúč prehľadávaného vrcholu, vkladaný vrchol (kľúč) ide navštíviť pravé dieťa prehľadávaného vrcholu (ak pravé dieťa neexistuje, vytvorí sa tam vrchol s vkladaným kľúčom). Smer navštívenia pri rovnakom kľúči vrcholov záleží od implementácii (ak sa kľúč rovná ide vždy doľava / doprava).

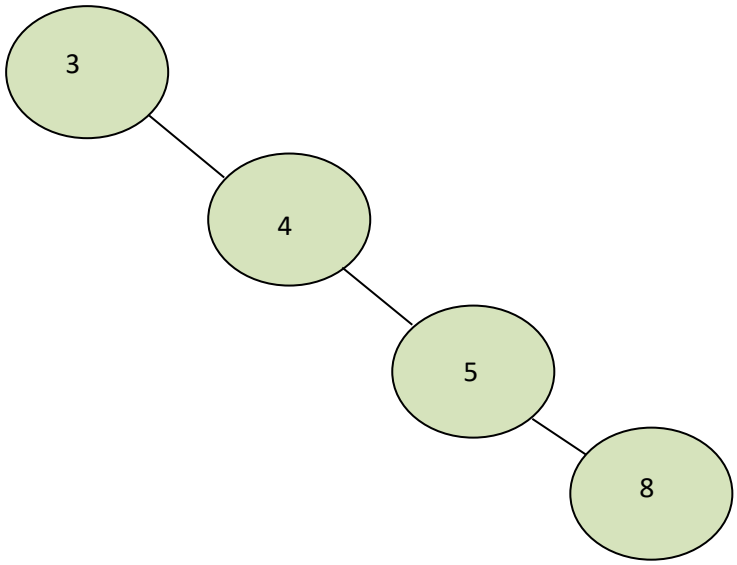


Vkladanie kľúča 4

Na obrázku vidíme príklad kedy sme do binárneho stromu vložili kľúč 4. Navštívil sa koreň stromu (prvý kľúč – najvyššia – v stromu), ktorý má kľúč 5, takže ďalej sa ide navštíviť jeho ľavé dieťa. Ľavé dieťa má kľúč 3 a to je menej ako vkladaný kľúč, takže sa ide navštíviť jeho pravé dieťa. Pravé dieťa vrcholu s kľúčom 3 neexistuje, takže to je to miesto kam sa vloží vrchol s kľúčom 4.

Meno : Richard Szarka
ID : 111958

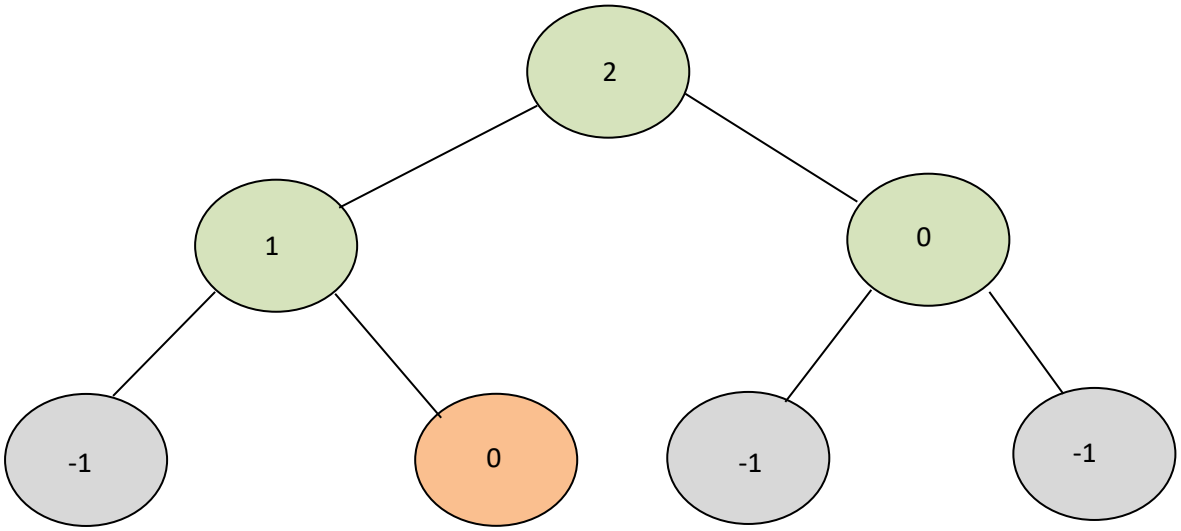
Čas vyhľadávania v takomto strome je O(logN) čo je veľmi výhodné. Avšak môže nastať prípad kedy nám prídu kľúče na vloženie do stromu v „nešťastnom“ poradí. Napríklad ak by kľúče z vyššieho stromu prišli v poradí 3, 4, 5, 8, tak by binárny strom vyzeral takto:



Ako vidíme, v tomto strome časová zložitosť na prehľadávanie je O(n), čo nie je veľmi výhodné. Preto existujú rôzne algoritmy ktoré stromy vyvažujú aby časová zložitosť ostala O(logN).

2.1 AVL samovyvažovací binárny strom

Algoritmus na vyvažovanie ktorý som si vybral pre moju implementáciu je AVL samovyvažovací strom. Strom funguje na princípe že každý vrchol si udržiava informácie nie len o ľavom a pravom dieťati, o kľúče a o hodnote na danom vrchole ale aj o výške od najvzdialenejšieho listu. Ukážka hodnôt výšok jednotlivých vrcholov z príkladu vyššie:



Meno : Richard Szarka
ID : 111958

Ak dieťa nejakého vrcholu neexistuje (NULL ukazovateľ), hodnota jeho výšky je -1 (zvykne sa tomu hovoriť aj že vrchol je ešte „pod vodou“).

Samotná potreba vyvažovania na danom vrchole sa bude robiť jednoduchým výpočtom:

| (Výška ľavého dieťaťa) – (výška pravého dieťaťa) | <= 1

Ak výpočet vyjde väčší ako 1 tak na danom vrchole je potrebné urobiť vyvážanie. Ak zistíme, že je potrebné urobiť vyváženie nastane otázka: „Treba presunúť váhu z ľavého dieťaťa do pravého alebo z pravého dieťaťa do ľavého?“ Na túto otázku nájdeme odpoveď zas v rovnici vyššie. Ak nám výpočet bez absolútnej hodnoty vyjde väčší ako 1, tak prvý člen v rozdiele (ľavé dieťa) muselo mať väčšiu výšku. Takže väčšina váhy je v podstrome ľavého dieťaťa. Ak nám rozdiel vyjde menší ako -1 tak druhý člen rozdielu musel byť väčší. Takže v tom prípade vačšina váhy je v podstrome pravého dieťaťa. Vkladanie vrcholov je rekurzívne, keďže každý podstrom sa dá brať ako osobitný strom, takže vyvažovanie sa kontroluje pri spätnom behu rekurzie. Samotný proces vyvažovania sa robí pomocou rotácií, ktoré si vysvetlíme neskôr.

2.2 Moja implementácia AVL stromu

V mojej implementácii budem používať ako kľúč celé číslo a ako hodnotu reťazec.

2.21 Knižnice makrá a štruktúra vrcholu

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define left_child_height(x) (((x)->left==NULL) ? -1 : ((x)->left->height))
// vráť hodnotu lavého dieťata
#define right_child_height(x) (((x)->right==NULL) ? -1 : ((x)->right-
>height)) // vráť hodnotu praveho dieťaťa
#define left_child(x) ((x)->left) //lave dieťa
#define right_child(x) ((x)->right) //prave dieťa
#define VERTEX_balance(x) ((left_child_height(x))-(right_child_height(x)))
//zisti rovnovahu na danom vrchole

#define HIGHER_child_height(x)
((left_child_height(x))>(right_child_height(x)) ? (left_child_height(x)+1)
: (right_child_height(x)+1) ) //vrať väčšiu vyšku detí

#define INORDER // druh vypisu
#define LEFT_HEAVY 1 //hodnota pri ktorej je viac váhy nalavo
#define RIGHT_HEAVY (-1) //hodnota pri ktorej je viac váhy napravo

//struktura vrchola
typedef struct vertex{

    int vertex_key; //kluc
    int height; //vyska
    char * value; //hodnota
    struct vertex * left; //lave dieta
    struct vertex * right; //prave dieta

}VERTEX;
```

Meno : Richard Szarka
ID : 111958

2.22 Inicializácia stromu

Inicializáciu stromu, vytvorenie prvého vrcholu (koreň), je realizovaná pomocou funkcie:

```
VERTEX * tree_init(VERTEX * root) //inicializuj prazdny strom
{
    root=(struct vertex*)malloc(sizeof(struct vertex));
    root->vertex_key=0;
    root->left=NULL;
    root->right=NULL;
    root->height=-1;
    return root;
}
```

Funkcia alokuje miesto v pamäti o veľkosti jedného vrcholu a inicializuje hodnoty v danej štruktúre (vrcholu). Nastaví sa hodnota výšky vrcholu na -1, aby pri prvom vkladaní hodnoty sa vedelo, že strom je čerstvo po inicializácii a údaje v ňom sú neplatné. Funkcia vráti inicializovaný strom (1. vrchol).

2.23 Vkladanie nových údajov do stromu

Vkladanie nových údajov do stromu (vytváranie nových vrcholov) sa realizuje pomocou funkcie:

```
VERTEX * tree_insert(int key,char * value,VERTEX * act_root) //vkladanie
{
    if (act_root->height==-1)
    {
        act_root->vertex_key = key;
        act_root->value = value;
        act_root->height = 0;
        return act_root;
    } //ak je strom hned po incializacii

    if(key<=act_root->vertex_key) //ak je vkladana vertex_value mensia
alebo rovna ako vertex_value na aktualnom vrchole -> chod do laveho dietata
    {
        if(act_root->left!=NULL){ //ak lave dieta existuje chod dalej
            act_root->left=tree_insert(key,value, act_root->left);
        }
        else{
            act_root->left=create_vertex(key, value); //ak lave dieta
neexistuje vytvor tam vrchol s hodnotou
        }
    }
    else //ak je vkladana vertex_value vacsia ako vertex_value na aktualnom
vrchole -> chod do praveho dietata
    {
        if(act_root->right!=NULL){ //ak prave dieta existuje chod dalej
            act_root->right=tree_insert(key,value, act_root->right);
        }
        else{ //ak prave dieta neexistuje vytvor novy vrchol s hodnotou
            act_root->right=create_vertex(key,value);
        }
    }
    update_height(act_root); // aktualizuj vysky vrcholov
    act_root=balancing(act_root);
    return act_root;}
```

Meno : Richard Szarka
ID : 111958

Funkcia má argumenty kľúč (celé číslo), reťazec (hodnotu) a koreň stromu (prípadne podstromu). Najprv sa skontroluje, či je strom čerstvo po inicializácii (výška koreňa bude -1). Ak áno, vložia sa tam žiadané hodnoty. Ak nie, skontroluje sa či vkladaný kľúč je väčší alebo menší ako kľúč na aktuálnom koreni podstromu. Ak je menší alebo rovný a ľavé dieťa neexistuje, vytvorí sa na jeho mieste nový vrchol funkciou create_vertex. Ak existuje ľavé dieťa, zavolá sa funkcia tree_insert s rovnakým kľúčom a hodnotou, ale na podstrome ľavého dieťaťa. Celý tento proces sa rekurzívne opakuje. Ak je vkladaný kľúč väčší ako kľúč na aktuálnom vrchole (koreň podstromu), tak sa proces opísaný vyššie prebieha analogicky aj v tomto prípade len na pravom dieťati.

Na konci funkcie tree_insert sa zavolá funkcia update_height ktorá aktualizuje výšky potrebných vrcholov a funkcia balancing ktorá má na starosti vyvažovanie stromu.

2.24 Vyvažovanie stromu

Na vyvažovanie stromu budeme používať spomenutý vzorec, v ktorom odčítame výšku ľavého dieťaťa od pravého dieťaťa.

```
VERTEX * balancing(VERTEX * act_root)
{
    if (abs(VERTEX_balance(act_root))<=1)
        return act_root;
    else if(VERTEX_balance(act_root)>LEFT_HEAVY)
    {
        act_root=right_rotation(act_root);
    }
    else if(VERTEX_balance(act_root)<RIGHT_HEAVY)
    {
        act_root=left_rotation(act_root);
    }
    return act_root;
}
```

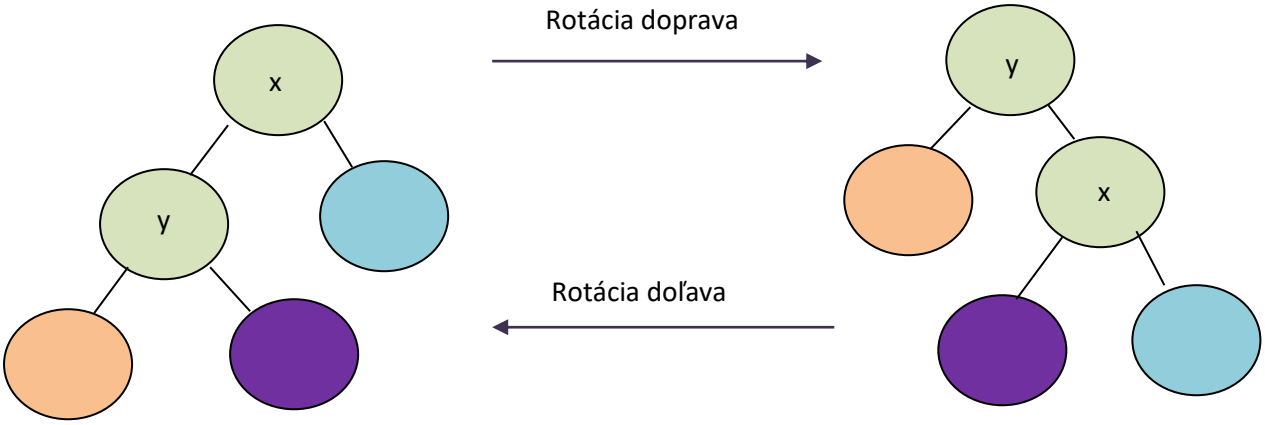
Ak sa zistí, že absolutná hodnota rozdielu výšok detí je menšia ako 1, vyvažovanie nie je potrebné. Ak je ale hodnota rozdielu väčšia ako 1 (väčšia váha stromu je na strane ľavého dieťaťa) alebo je menšia ako -1 (väčšia váha stromu je na strane pravého dieťaťa), treba uskutočniť rotácie stromu.

Rotácie poznáme štyri druhy:

- Rotácia doľava
- Rotácia doprava
- Rotácia doľava a potom doprava
- Rotácia doľava a potom doprava

Meno : Richard Szarka
ID : 111958

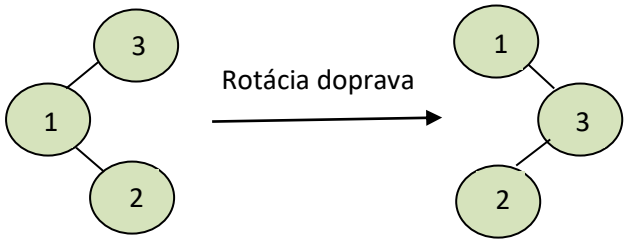
Rotácia doľava a doprava



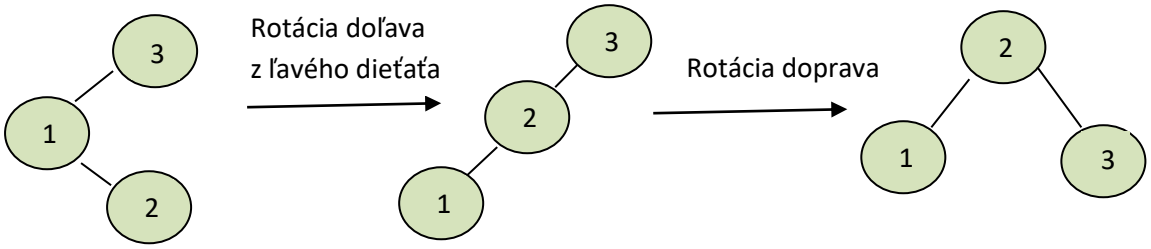
Rotácie prenesú váhu stromu z jednej strany na druhú. Hlavnú úlohu tu hrajú vrcholy X a Y ktoré sa vymenia ako je znázornené na obrázku. Ich deti sa vymenia podľa toho aby sa zachovali nerovnosti. Napríklad: Fialový vrchol vľavo je menší ako X ale väčší ako Y, tým pádom po rotácii doprava musí byť tiež menší ako Y ale väčší ako X. Na konci vyvažovania sa zavolá funkcia **update_height** na aktualizovanie výšok po **rotáciach**. Rotácie realizujú funkcie **left_rotation** a **right_rotation**.

Rotácia doľava potom doprava a doprava a potom doľava

Tento druh rotácii sa využíva vtedy ak by nám jedna rotácia spravila ten istý prípad len na druhej strane stromu.



Ako vidíme, vzniklo nám rovnako zlé riešenie, je to z toho dôvodu že ľavé dieťa (1) malo váhu na pravej strane. Preto keď to zrotujeme, tak váha pravého dieťaťa bude na ľavej strane a máme rovnako zlé riešenie. Ak sa vyskytne prípad, že potrebujeme spraviť rotáciu doprava, a váha ľavého dieťaťa je napravo, musíme najprv spraviť rotáciu doľava z ľavého dieťaťa (rotácia doľava a potom doprava). Princíp je rovnaký aj na rotácie doľava, len zrkadlovo.



Meno : Richard Szarka
ID : 111958

2.25 Vyhľadávanie prvkov

Vyhľadávanie prvkov je sprostredkované funkciou search_value:

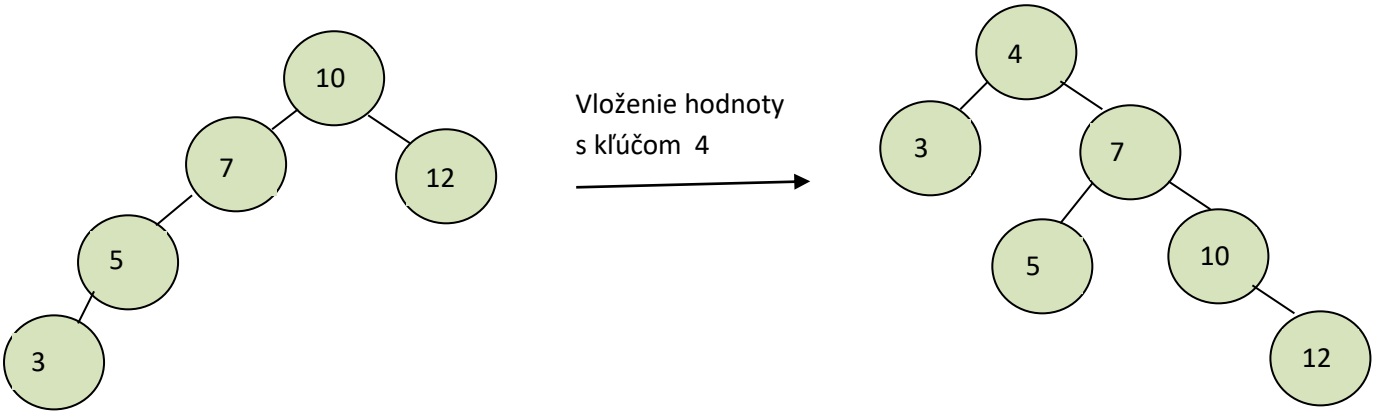
```
char* search_value(int key,VERTEX * act_root) //najdenie hodnoty v grafe
{
    if (act_root==NULL) //ak aktualny vrchol neexistuje vrat hodnotu 0
        return "Not found\n";
    else if(act_root->vertex_key == key) //ak// aktualny vrchol ma hodnotu
hladanej hodnoty vrat 1
        return act_root->value;
    else if(key<act_root->vertex_key) //ak hladana vertex_value je mensia
ako vertex_value vrcholu, chod do laveho dietata
        search_value(key, act_root->left);
    else //inac chod do praveho dietata
        search_value(key, act_root->right);
}
```

Argumenty funkcie sú celé číslo **key** (kľúč) a strom (prípadne **podstrom**). Vyhľadávanie funguje podobne rekurzívne ako samotné vkladanie. Podľa toho či vyhľadávaný kľúč je väčší, menší, alebo rovný sa rozhodne či „preskúma“ pravého alebo ľavého potomka aktuálneho vrcholu. Ak nájde vrchol s rovnakým kľúčom, prvok sa našiel. Ak pri preskúmavaní narazí na dieťa ktoré neexistuje, tak tam mala daná hodnota existovať. Vypíše sa „Not found“ (daná hodnota nenájdená).

Meno : Richard Szarka
ID : 111958

2.3 SPLAY strom

Hlavná úloha **splay** stromu je aktuálnu použitú hodnotu nastaviť ako koreň celkového stromu. Tým pádom ak vkladáme novú hodnotu s kľúčom, tak sa nastavia ako koreň a všetky ostatné hodnoty sa im prispôsobia tak, aby boli splnené nerovnosti, ktoré pri binárnych stromoch sú nám už známe. Výhoda **splay** stromu je, že často používané dáta sú vždy „po ruke“. **Splay** strom nie je dokonale vyvážený ako AVL strom, keďže vyvažovanie je realizované algoritmom, ktorý nastavuje používané hodnoty na koreň stromu. Hovorí sa mu aj „polo-vyvažovanie“ (semi-balancing). Keďže **splay** strom rotuje používané hodnoty na koreň stromu, znamená to že aj pri každom vyhľadávaní zrotuje vyhľadávanú hodnotu (vrchol) na koreň.



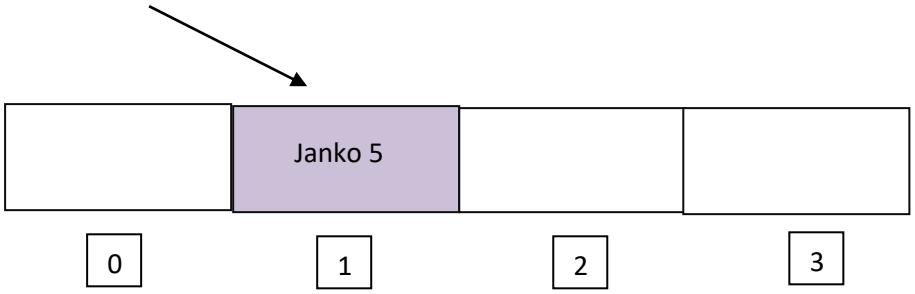
Nie vlastnú implementáciu tohto algoritmu binárneho stromu neskôr porovnám s binárnym stromom a hašovacími tabuľkami.

Meno : Richard Szarka
ID : 111958

3 Hašovacie tabuľky

Ďalší známe vyhľadávacie algoritmi patria do rodiny hašovacích tabuliek. Srdce samotného vyhľadávacieho algoritmu je hašovacia funkcia, ktorá pomocou rôznych operácií (vždy rovnakých), zo zadaného kľúča vytvorí haš. Po prevedených operaciach je haš ešte zmodulovaný veľkosťou tabuľky . Haš je ďalej použitý ako index miesta kam daná hodnota s daným kľúčom budú uložené v hašovacej tabuľke. Príklad hašovacej funkcie „klúč * 3+2“ o tabuľky veľkosti 4 , kľúča 5 a hodnoty „Janko“:

Hash=(5*3+2)%4 -> Hash=1 (výsledok hašu určuje index v pozícii tabuľky)



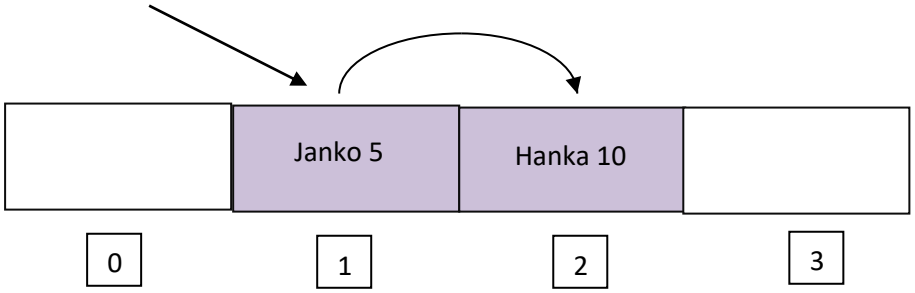
Avšak pri vytváraní takejto hašovacej tabuľky vznikajú rôzne komplikácie. Prvá z nich je zaplnenie samotnej tabuľky. Ak sa nám tabuľka zaplní nemôžeme ďalej vkladať do nej prvky. Druhá sú tzv. kolízie. Čo ak nám 2 rôzne kľúče vygenerujú ten istý haš? Táto problematika je riešená rôznymi spôsobmi. Jedna z nich je metóda Lineárnych pokusov (metóda lineárneho sondovania) a ďalšia je pomocou zreťazovania.

Vyrieši open a closed adresing

3.1 Lineárne sondovanie

Ak pri lineárnom sondovaní nastane kolízia, algoritmus skúša nájsť miesto na väčšom indexe. Ak by už vyšiel von z tabuľky, vráti sa na jej začiatok. Príklad kolízie pomocou hašovacej tabuľky vyššie (pridáme hodnotu „Hanka“ s kľúčom 7):

Hash=(7*3+2)%4 -> Hash=1 (výsledok hašu určuje index v pozícii tabuľky)



Ak zistí, že miesto na danom indexe je zabrané, skúsi prvok vložiť na miesto vpravo od neho.

Meno : Richard Szarka
ID : 111958

Zväčšovanie tabuľky je riešené tak, že ak tabuľka je zaplnená na X% (x-lubovolná hodnota v %, záleží od implementácie), tak sa vytvorí nová alebo zväčší pôvodná a všetky prvky sa premiestnia do novej (ale musia sa všetky nanovo prehašovať).

3.2 Moja implementácia lineárneho sondovania

Moja implementácia hašovacej tabuľky bude používať na kľúče celé čísla (integery) a na hodnotu reťazce. Tým pádom, že je rovnaká implementácia kľúčov a hodnôt aj v binárnych stromoch, testovanie a porovnávanie algoritmov bude dosahovať relevantnejšie výsledky

3.21 Knižnice, makrá, štruktúra tabuľky a štruktúra prvku v tabuľke

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define table_item_value(x) (hash_table->items[x]->value) //vrat mi hodnotu itemu v tabulke
#define table_item_key(x) (hash_table->items[x]->key) //vrat mi kluc itemu v tabulke


typedef struct item{ //struktura predmetu - prvku
    char * value;
    int key;
}ITEM;

typedef struct Hash_table{ //struktura tabulky
    int size;
    int occupied;
    ITEM ** items;
}TABLE;
```

3.22 Inicializácia tabuľky

```c
TABLE * hash_table_init(int size) { // inicializacia zakladu tabulky
    TABLE * Hash_table = (TABLE*)calloc(1,sizeof(TABLE)); // 1 tabulka
    Hash_table->size = size; //velkost tabulky
    Hash_table->occupied = 0; //okopuvaných elementov
    Hash_table->items = (ITEM**)calloc(size,sizeof(ITEM*)); //prvky v tabulke

    return Hash_table;
}
```

Inicializácia hašovacej tabuľky je vykonávaná funkciou **hash_table_init**. Argumentom funkcie je celé číslo, ktoré určuje inicializačnú počiatočnú veľkosť tabuľky. Vyhradí sa miesto v pamäti na tabuľku, vyhradí sa miesto aj pre prvky tabuľky, zapíše sa aktuálna veľkosť tabuľky a počet obsadených miest v tabuľke za nastaví na 0.

Meno : Richard Szarka
ID : 111958

3.23 Hašovacia funkcia

```
int hashing(int key,int size){ //hashovacia funkcie
    int long long hashc=(((key+457)*797)-257);
    int hash=hashc%size;
    return hash;
}
```

Moja implementácia hašovacej funkcie ku kľúču pripočíta 457, následne vynásobí číslom 797 a nakoniec odčíta 257. Nakoniec operácia modula o hodnote aktuálnej veľkosti tabuľky. Čísla sú zvolené nie náhodne. Každé jedno z daných čísel je prvočíslo, aby počet kolízii bol čo najmenší a tým sa aj rýchlosť vyhľadávania zvýši.

3.24 Vkladanie prvkov do tabuľky

Vkladanie prvkov do tabuľky vykonáva funkcia **table_insert**. Vo funkcii sa vypočíta haš daného kľúča. Ďalej sa vojde do while-cyklu, ktorý **iteruje**, ak je splnená podmienka, že na indexe v tabuľke daného existuje nejaký prvok. Ak prvok neexistuje, nenastala kolízia a môže tam vytvoriť prvok s danou hodnotou a kľúčom. Ak ale prvok na danom mieste existuje, skontroluje sa či vkladaný kľúč a kľúč na danom mieste nie sú zhodné. Ak sú zhodné, hodnota prvku sa aktualizuje a vráti sa z funkcie. Ak už nejaký prvok na danom mieste existuje, vo while-cykle sa **inkrementuje haš** a zmoduluje sa veľkosťou tabuľky (ošetrenie aby nevyšiel mimo tabuľky a vrátil sa na začiatok). Ak nájde voľné miesto, môže analogicky vytvárať prvok na danom mieste s danou hodnotou a daným kľúčom.

Na konci funkcie sa inkrementuje hodnota obsadených miest v tabuľke a skontroluje sa či tabuľka nie je už veľmi plná (70% pôvodných miest je už obsadených) . Ak je tak sa tabuľka zväčší.

3.25 Zväčšovanie tabuľky

```
TABLE * resize(TABLE * hash_table) {
    int new_size=(hash_table->size)*2+1; //zvacsenie tabulky
    TABLE * new_table = (TABLE*)calloc(1,sizeof(TABLE)); // 1 tabulka
    new_table->size = new_size; //velkost tabulky
    new_table->occupied = 0; //okopuvaných elementov
    new_table->items = (ITEM**)calloc(new_size,sizeof(ITEM*)); //spravenie
nových elementov

    for(int i=0;i<hash_table->size;i++) //premiestnovanie starej tabulky do
novej (podla hashu samozrejme)
    {
        if(hash_table->items[i]) //ak existuje prvok v tabulke premiestni
ho
        {
            table_insert(new_table,hash_table->items[i]->value,hash_table-
>items[i]->key,0);
            free(hash_table->items[i]);
        }
    }
    free(hash_table);
    return new_table;
}
```

Zväčšenie novej tabuľky je spravované funkciou **resize**. Ako prvé sa zistí nová veľkosť hašovacej tabuľky. Hodnota novej je vždy dvojnásobok veľkosti starej tabuľky zväčšený o 1, aby veľkosť bola nepárna (nastáva menej kolízii).

Meno : Richard Szarka
ID : 111958

Analogicky ako v **hash_table_init** sa inicializujú základné charakteristiky novej tabuľky. Ďalšie nastane premiestnenie prvkov zo starej tabuľky do novej pomocou funkcii **table_insert**. Vždy keď sa vloží prvok do novej, zo starej sa prvok uvoľní. Keď sa už premiestnili všetky prvky uvoľní sa stará tabuľka a funkcia vráti novú tabuľku.

3.26 Vyhľadávanie prvkov v hašovacej tabuľke

```
char* search_table(TABLE * hash_table,int key) { //prehladaj tabulku
    int hash = hashing(key, hash_table->size); //zisti hash podla klucu
    while(hash_table->items[hash] && table_item_key(hash)!=key) //kym
existuje prvok na danom mieste a zaroven kluc nie je zhodny
    {
        hash=(hash+1)%hash_table->size;
    }
    if(hash_table->items[hash]) { //ak existuje prvok
        return hash_table->items[hash]->value;
    }//ak neexistuje prvok
    else{
        return "Not Found";
    }
}
```

Vyhľadávanie prvku v tabuľke je vykonávané funkciou **search_table**. Aby sme vyhľadali prvok v tabuľke potrebujeme zistiť na akom indexe v našej tabuľke sa prvok nachádza (poprípade by sa mal nachádzať). Tým pádom najprv zistíme aký haš prislúcha ku kľúču ktorý chceme vyhľadať. Ak existuje prvok na danom mieste a kľúč nie je zhodný, tak to nie je prvok ktorý hľadáme a na danom mieste nastala kolízia. Tým pádom by sme ho mali hľadať napravo od momentálne skúmaného prvku. Z **while cyklu** sa vyjde ak nebude splnená podmienka, že prvok existuje alebo že kľúče sú rozdielne. Ak vyšiel z dôvodu že kľúče sú rozdielne, tak našiel prvok ktorý existuje a zároveň má rovnaký kľúč ako hľadaný kľúč. Takže sa našiel hľadaný prvok a funkcia vráti jeho hodnotu. Ak vyšiel z dôvodu, že na skúmanom mieste neexistuje prvok, tak daný prvok sa nenachádza v našej tabuľke. Ak by sa nachádzal tak by vedel byť nájdený hašom z kľúča zadaným v argumente funkcie **search_table**.

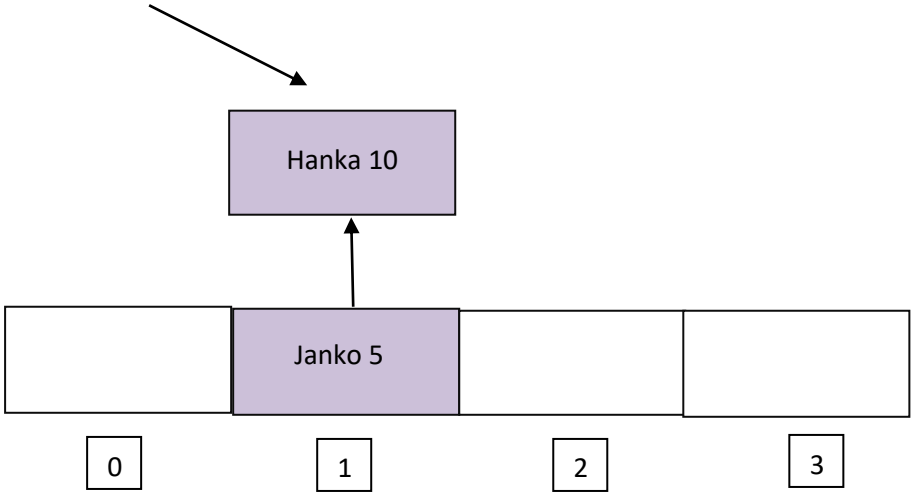
Meno : Richard Szarka
ID : 111958

3.3 Riešenie kolízii zreťazovaním

Pri metóde zreťazenia má štruktúra prvku aj ukazovateľ na ďalší prvok. Ak v metóde zreťazovania nastane kolízia, prvok sa nepošle na ďaľšie miesto v tabuľke ako v metóde lineárneho sondovania, ale ukazovateľ prvku s ktorým vkladaný prvok mal kolíziu sa nastaví na vkladaný prvok.

Kolízia z príkladu vyššie (linearne sondovanie)

Hash=(7*3+2)%4 -> Hash=1 (výsledok hašu určuje index v pozícii tabuľky)



Otázka zväčšovania tabuľky je princípovo rovnaká ako pri metóde lineárnych pokusov (koľko % tabuľky je zaplnenej), ale môže sa tu nastaviť aj hranica na to koľko prvkov sa za sebou môže maximálne zreťaziť. Tieto parametre však závisia od konkrétnej implementácie.

Nie vlastnú implementáciu tohto algoritmu budem porovnávať s mojimi algoritmami vyššie rozobratými (aj so SPLAY stromom).

Prebratý algoritmus som upravil tak, aby kľúče boli celé čísla a hodnoty boli reťazce. Následne som musel upraviť aj hašovaciu funkciu keďže pôvodne kľúč bol reťazec.

Meno : Richard Szarka
ID : 111958

4 Testovanie

Testovanie prebehne pomocou funkcií **test_tree** a **test_table**. **Test_tree** otestuje rýchlosť jednotlivých úkonov daných stromov. **Test_table** otestuje rýchlosť jednotlivých úkonov daných tabuliek.

Do môjho aj prevzatého stromu som pridal funkciu **count_rotation** a **count_splay_rotation** aby som zistil koľko rotácii spravili stromy. Tak isto moju funkciu na vkladanie do tabuľky a prevzatú funkciu na vkladanie do tabuľky som prerobil tak aby vedeli rátať kolízie (nerátajú sa kolízie pri **rehašovaní** tabuliek – ak je posledný argument vkladacích funkcii 1 bude rátať kolízie pri vkladaní). Kolízie rátajú funkcie **count_collision** a **count_collision_chain.**

Na začiatku v main funkcii si vytvorím pole kľúčov o veľkosti požadovaného množstva vkladania. Následne for cyklom vygenerujem náhodne kľúče a vložím ich do poľa. Zistil som že na windowse rand() pri hocijakej veľkej zadanej hodnote generuje max do 30 tisíc. Preto som spravil bitový posun o 5 doľava aby hodnoty boli vždy okolo 1 milióna.

```
for(int i=0;i<ELEMENTS;i++)
{
    int temp = rand() % (MAX_KEY)+MIN_KEY;
    temp=temp<<5;
    random[i]=temp+rand() % (MAX_KEY+1)-1+MIN_KEY;
}
```

V testoch budem využívaj aj grafy s logaritmickou hlavnou osou, čo znamená že bude znázorňovať nie vzdialenosť medzi jednotlivými úsekmi ale nárast ich pomeru. Ak chýba uzol v grafe je to z toho dôvodu, že čas alebo iná hodnota bola tak malá, že v samotnom testovači bola nenamerateľná (vypísal 0.0000 s).

Testovanie binárnych stromov

V testovaní stromov vždy vložím všetky vygenerované kľúče pod hodnotou „test“, vyhľadám každý kľúč a nakoniec vyhľadám 10% náhodne vybraných kľúčov, pričom počítam počet rotácií.

Testovanie hašovacích tabuliek

V testovaní tabuliek vložím všetky vygenerované kľúče pod hodnotou „test“, vyhľadám každý kľúč a nakoniec vyhľadám 10% náhodne vybraných kľúčov, pričom sa počíta koľko kolízií nastalo.

Knižnice a makrá použité v testovači

```
#include "tabulka.h"
#include "tree.h"
#include "splay_tree.h"
#include "chaining.h"
#include <time.h>

#define ELEMENTS 100000
#define MAX_KEY 1000000
#define MIN_KEY 1
#define TABLE_SIZE 150000
```

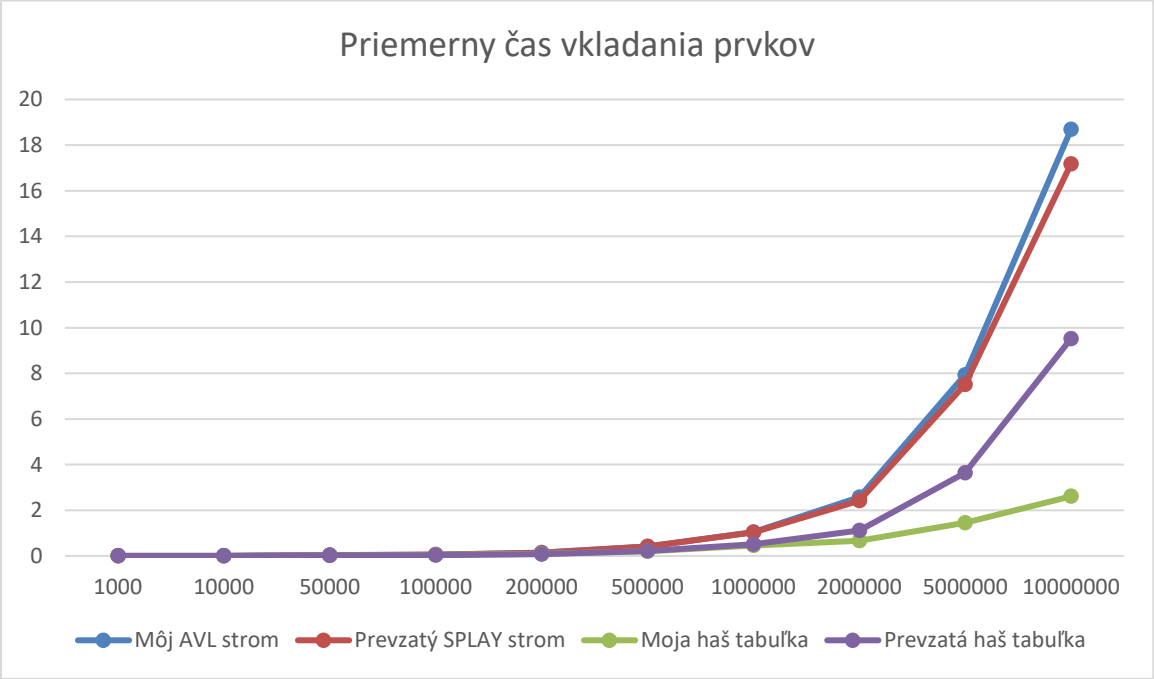
Meno : Richard Szarka
ID : 111958

4.1 Testovanie rýchlosti

Pri testovaní rýchlosti jednotlivých prehľadávacích algoritmov budem testovať zvlášť vkladanie prvkov, hľadanie prvkov, hľadanie náhodných prvkov a celkový čas. Zadané množstvo prvkov na test bude 1000, 10000, 50000, 100000, 200000, 500000, 1000000, 2000000, 5000000, 10000000

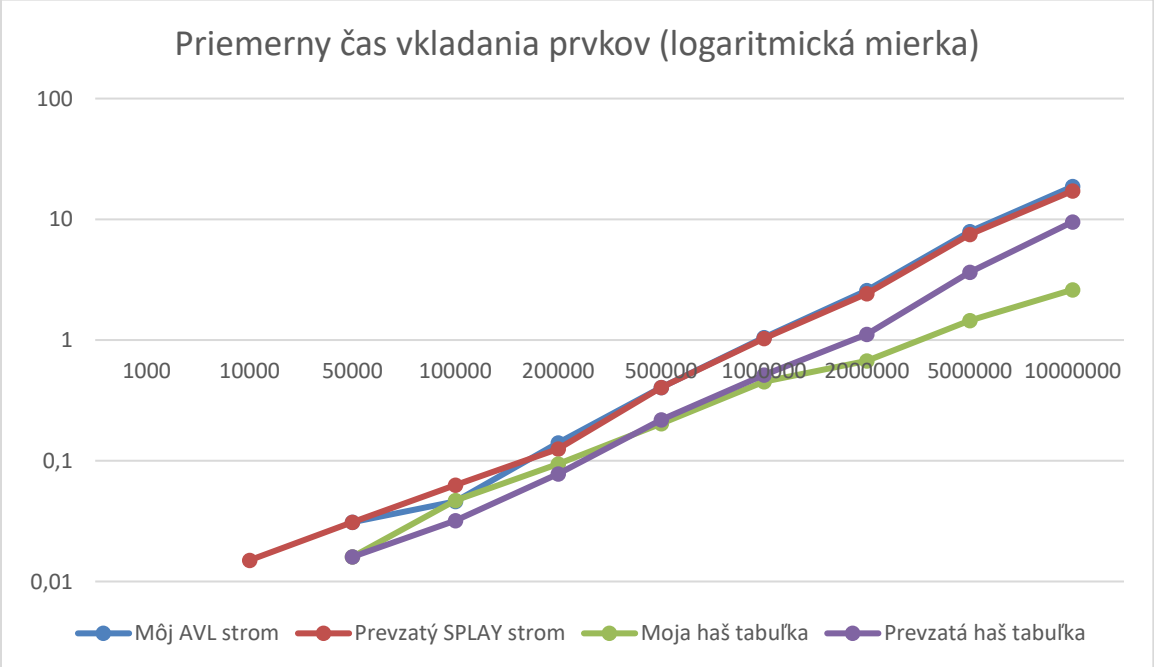
Inicializačná veľkosť tabuliek bude vždy 10% z požadovaného množstvo vkladaných prvkov.

Testovanie vkladania



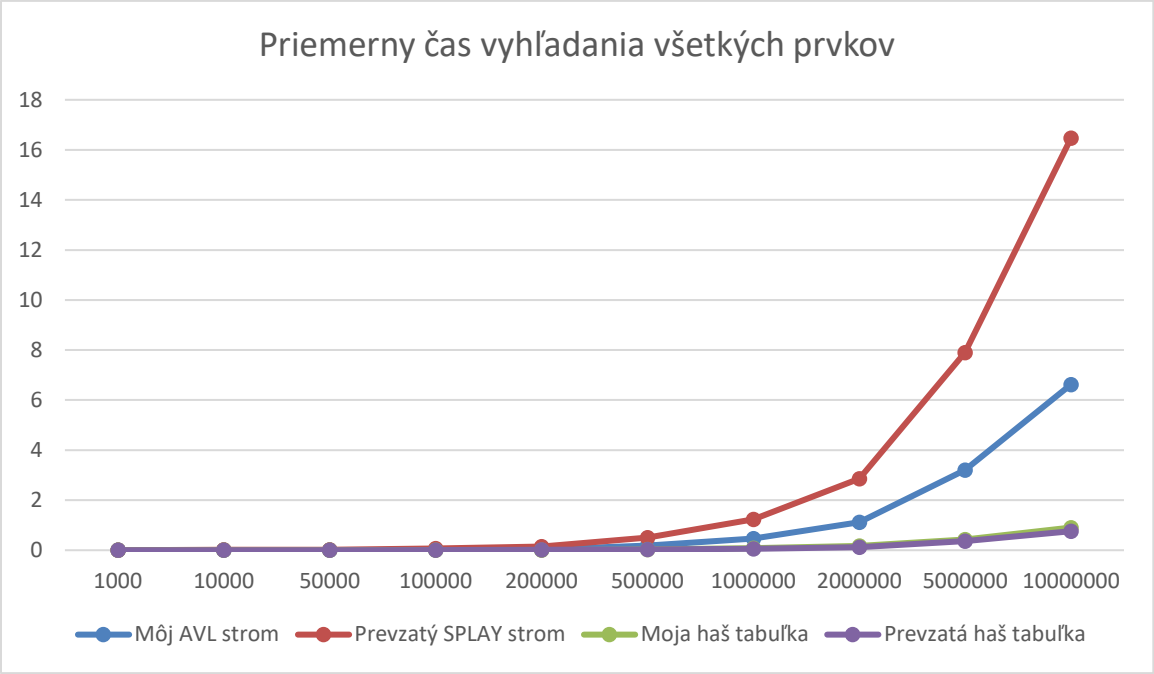
X os – počet prvkov

Y os – čas v [s]



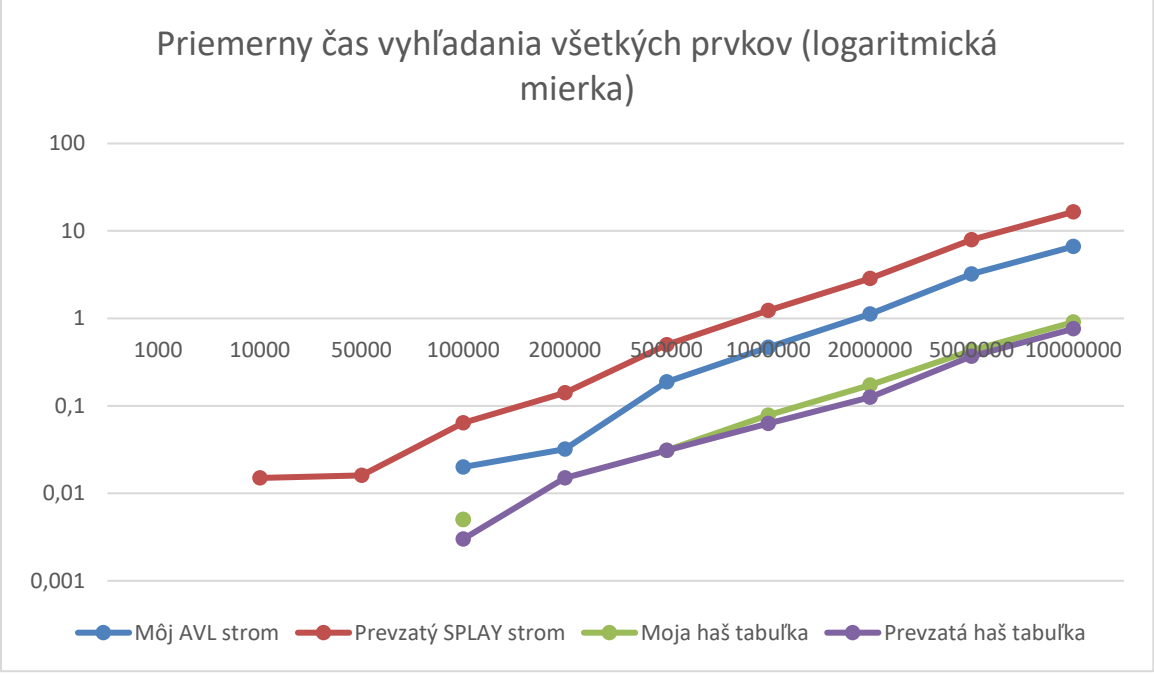
Meno : Richard Szarka
ID : 111958

Testovanie vyhľadávania všetkých vložených prvkov



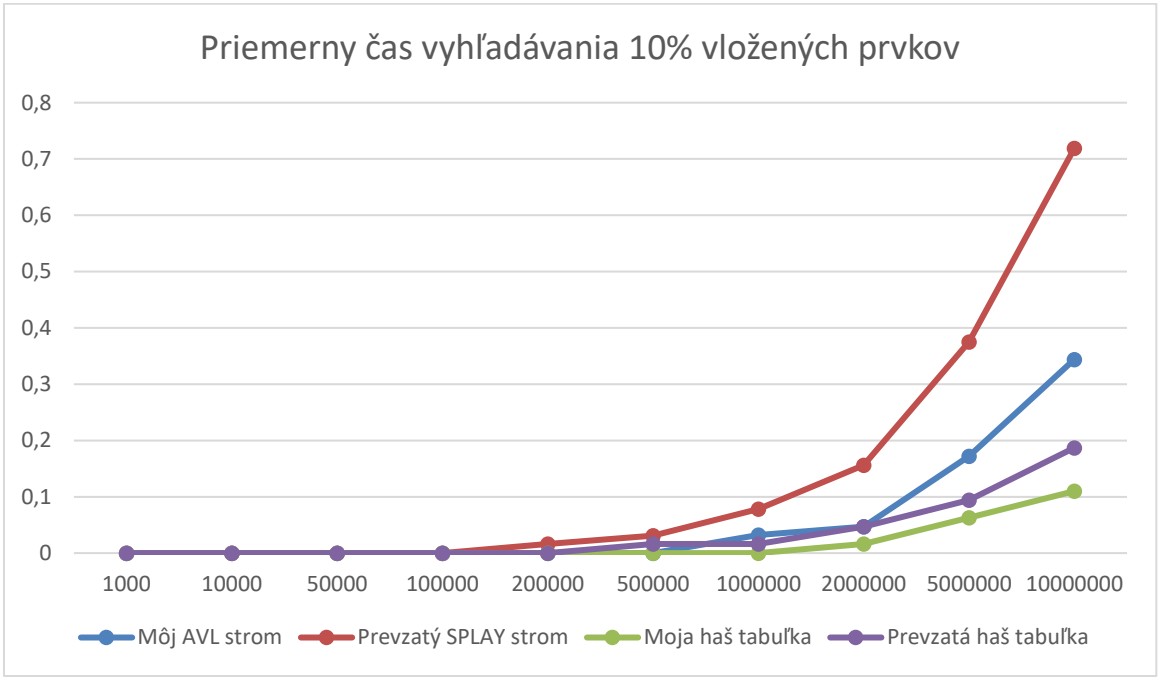
X os – počet prvkov

Y os – čas v [s]



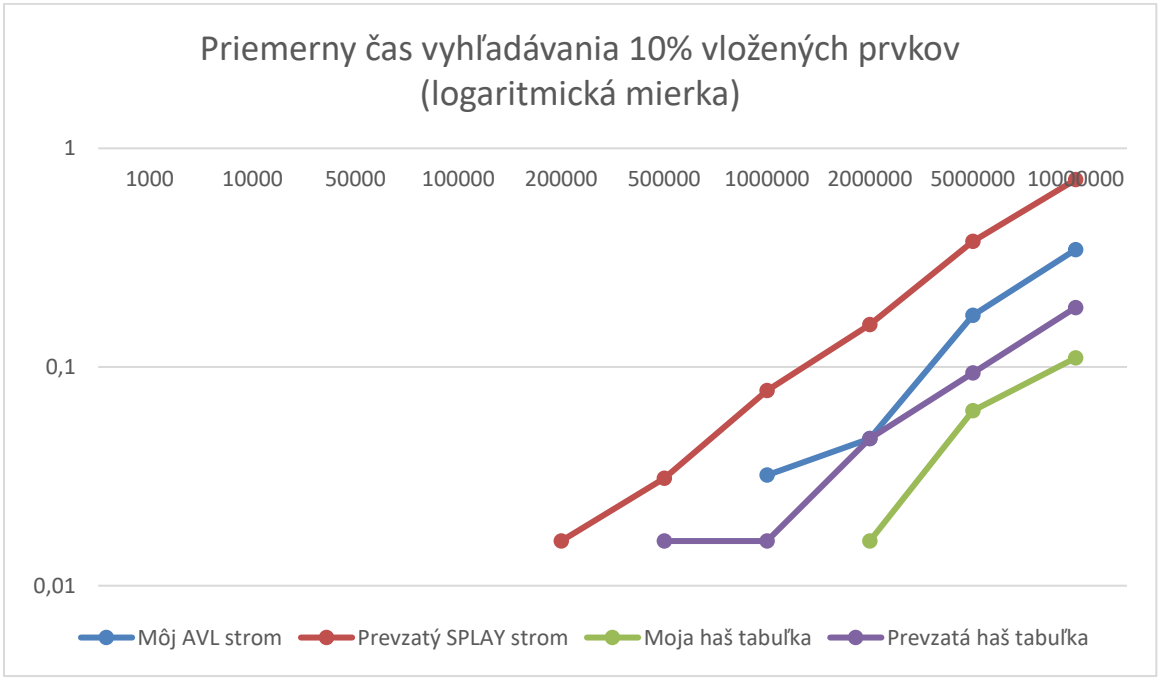
Meno : Richard Szarka
ID : 111958

Testovanie vyhľadávania 10% náhodných vložených prvkov



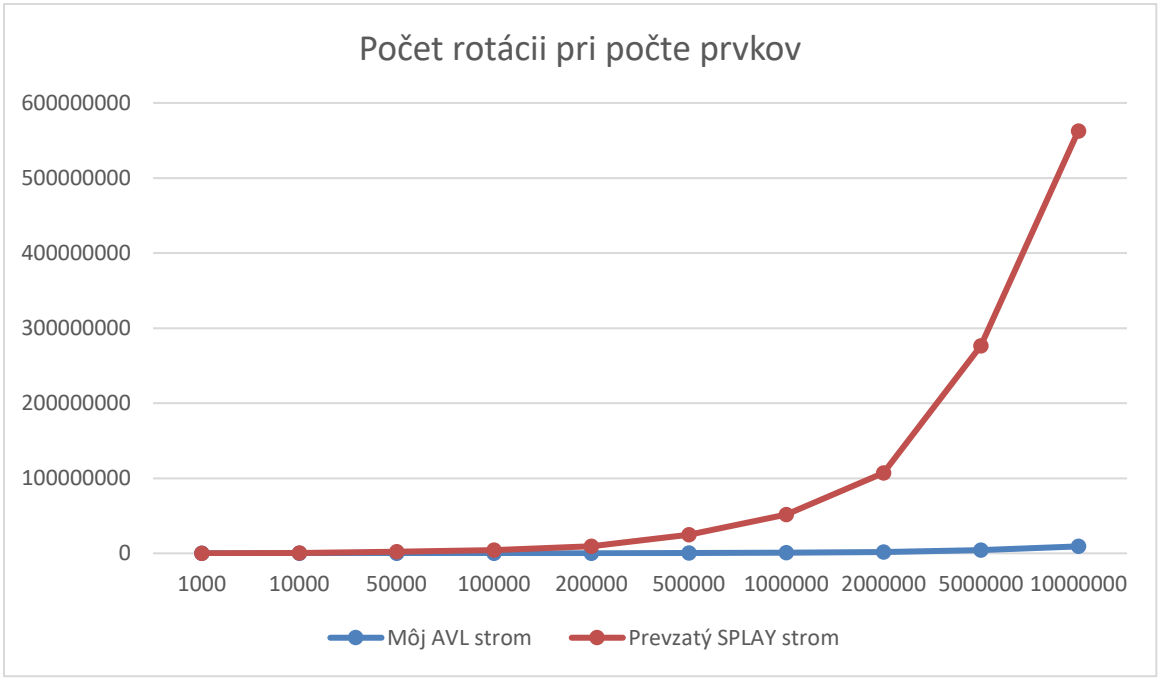
X os – počet prvkov

Y os – čas v [s]



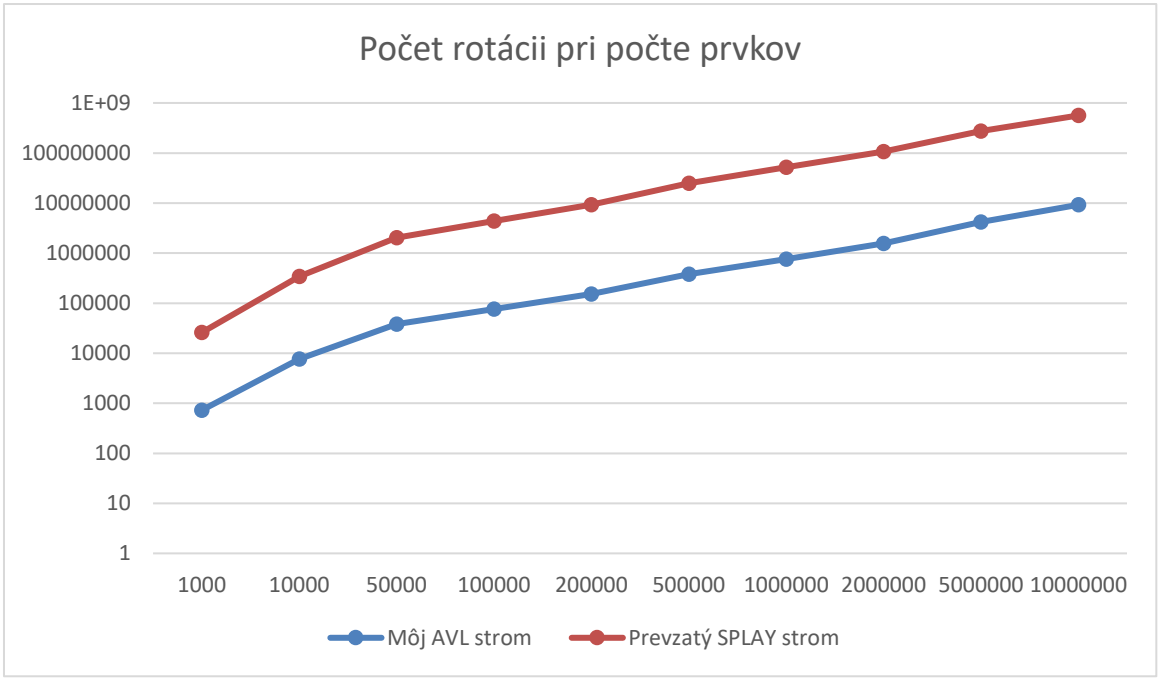
Meno : Richard Szarka
ID : 111958

4.2 Testovanie počtu rotácii stromov (pri vkladaní všetkých prvkov aj vyhľadaní všetkých prvkov)



X os – počet prvkov

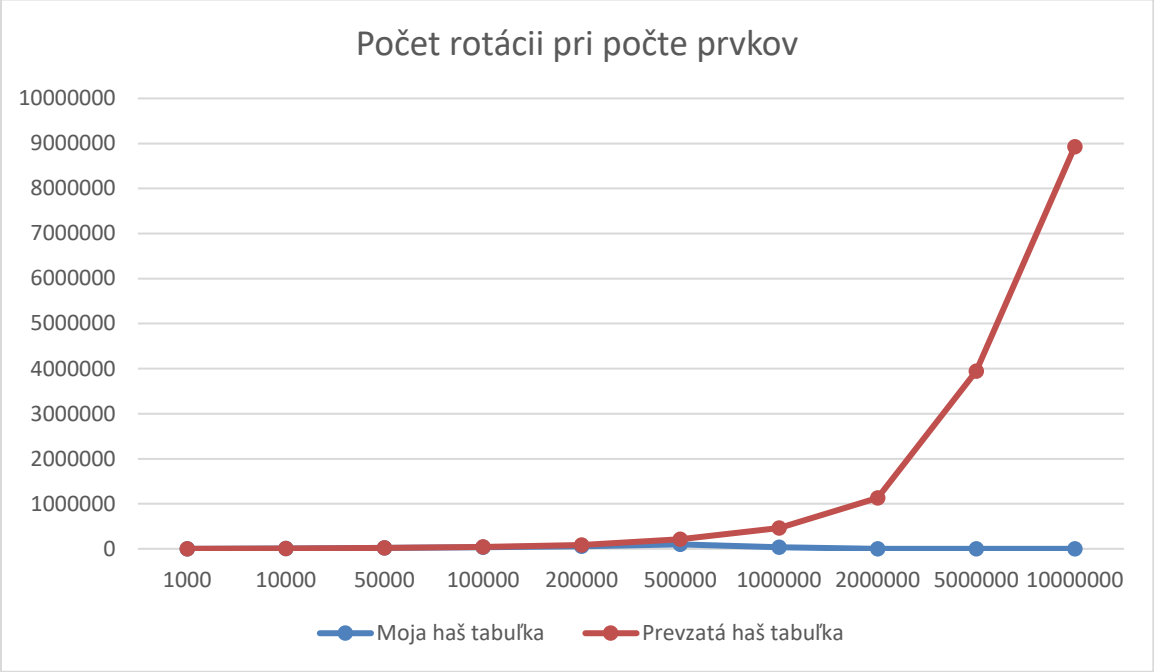
Y os – počet rotácií



Meno : Richard Szarka
ID : 111958

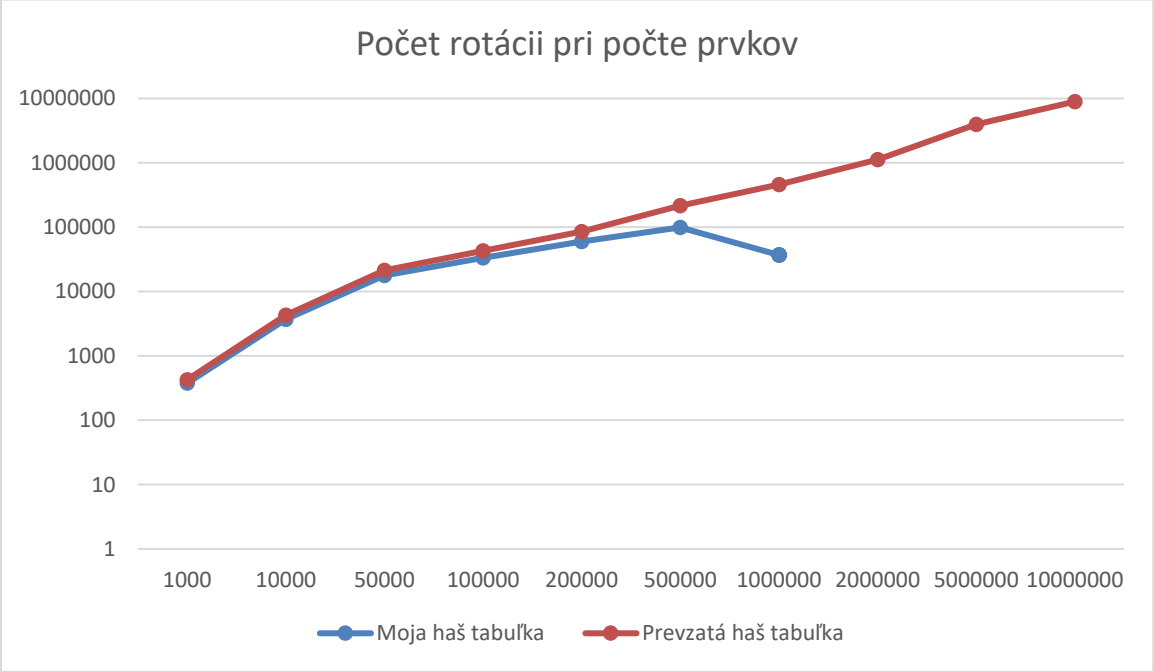
4.3 Testovanie počtu kolízií

Keďže som musel upraviť hašovaciu funkciu druhého kódu, tak toto porovnanie je len na prehľad koľko kolízií približne vzniká. (tabuľku nechám vždy rovnakej veľkosti ako počet vkladaných prvkov)



X os – počet vkladaných prvkov

Y os – počet kolízií



Meno : Richard Szarka
ID : 111958

4.31 Testovanie správnosti počtu kolízií

Keďže výsledky počtu kolízií v mojej tabuľke vyšli podozrivo dobre, rozhodol som sa otestovať dvojitým for cyklom či počet kolízií približne sedí (stačí približne kvôli možnosti prepisovania). Pri tomto teste však nesmie nastať zväčšovanie tabuľky.

Časť kódu na kontrolu danej vlastnosti:

```
int * my_hashes=(int*)calloc(ELEMENTS,sizeof(int));
    int cnt=0;

    for(int i=0;i<ELEMENTS;i++) { //testovanie spravnosti kolízii
        my_hashes[i] = hashing(random[i], TABLE_SIZE); // vytvorenie poľa
hasov
    }
    for(int i=0;i<ELEMENTS;i++) //kontrola kazdeho prvku s kazdym (ak sa
zhoduju nastala kolizia)
{
        for(int j=0;j<ELEMENTS;j++)
        {
          if(my_hashes[i]==my_hashes[j] && random[i]!=random[j])
         {
             cnt++;
         }
       }

}
```

Kontrolu budem robiť do tabuľky dvojnásobne väčšej ako je počet prvkov aby nenastalo zväčšovanie. Taktiež kontrolu budem robiť do vkladania 200000 prvkov, lebo dva for cykly v sebe majú časovú zložitosť $O(n^2)$.

	1000	*10000*	*50000*	*100000*	*200000*
Výsledok z tabuľky	255	2461	11359	20325	32308
Výsledok z dvoch for cyklov	257	2496	11424	20387	32581

Z tabuľky môžeme vidieť, že hodnoty sedia. Hodnoty sa líšia kvôli tomu že dva for cykly nevedia zistiť kedy sa jedná o skutočnú kolíziu (rátajú aj prepis hodnoty).

Meno : Richard Szarka
ID : 111958

5 Záver

Vyhľadávacie algoritmy sú veľmi užitočné. Môžeme vidieť na grafoch, že dané množstvo dát sa vyhľadalo za veľmi krátky čas. Tiež môžeme usúdiť, že haš tabuľky sú rýchlejšie ako binárne stromy.

Porovnanie binárnych stromov

Z grafov je jasne vidieť, že SPLAY strom má lepší čas vkladania prvkov ale skoro trikrát horšie vyhľadanie všetkých prvkov. Keďže SPLAY strom je výhodný použiť pri súboru dát, v ktorých sa používa často nejaká podmnožina z celkového množstva dát v súbore. SPLAY strom splay algoritmom udržiava často použité prvky čo najvyššie, preto nie je jeho vyváženie také dobré ako stromu AVL. SPLAY strom robí omnoho viac rotácií, keďže rotuje aj pri vyhľadávaní.

Porovnanie haš tabuliek

Na prvý pohľad vidíme, že nie je veľký rozdiel medzi danými tabuľkami. Najväčší rozdiel nastal v čase vkladania do tabuľky. Tam moja implementácia hašovacej tabuľky mala výrazne lepší čas ako tabuľka prevzatá. Pri teste kolízií sme zistili že moja tabuľka (lineárne sondovanie) mala výrazne menej kolízií pri väčších číslach, čo mohlo zapríčiniť tento rozdiel. Čo sa týka vyhľadávania prvkov, moja tabuľka vyhľadala náhodne prvky rýchlejšie ako prevzatá, ale prevzatá vyhľadala rýchlejšie všetky prvky. Nepatrné rozdiely v časoch mohli nastať z daných subjektívnych implementácií tabuľky. Zlepšenie tabuliek by určite pomohla lepšia hašovacia funkcia.

Časová zložitosť

AVL binárny strom – O(logN)

SPLAY binárne strom – O(logN)

Hašovacia tabuľka (lineárne sondovanie) – Priemerne O(1) , najhorší prípad O(logN) (bez zväčšenia tabuľky O(N)

Hašovacia tabuľka (reťazenie) – Priemerne O(1), najhorší prípad O(logN) (bez zväčšenia tabuľky O(N)

Pamäťová zložitosť

AVL binárny strom – O(N)

SPLAY binárne strom – O(N)

Hašovacia tabuľka (lineárne sondovanie) – O(N)

Hašovacia tabuľka (reťazenie) – O(N)