

Zadanie 2 – Rafaj dokumentácia

Program 1 (2B) – Binárny vyhľadávací strom s vyvažovaním AVL

AVL strom je typ samovyvažovacieho stromu, kde každý vložený uzol má svoj balance faktor. Balance faktor je číslo, ktoré poukazuje na vyváženosť daného uzlu / prvku v strome. Akceptovateľné čísla vo vyváženom strome, sú 0 (strom je dokonalo vyvážený) alebo 1 / -1 (strom je naklonený do niektorej strany), kde je buď viac toho napravo alebo viac toho naľavo, avšak pre danú skladbu prvkov nie je možné už viac vyvažovať. Avšak po pridaní napr. už predtým prevažujúcu pravú, či ľavú stranu ďalšieho uzlu sa faktor zmení na -2 alebo 2 a to spôsobí to, že jedna strana bude prevážaná a preto sa dá strom vyvážiť.

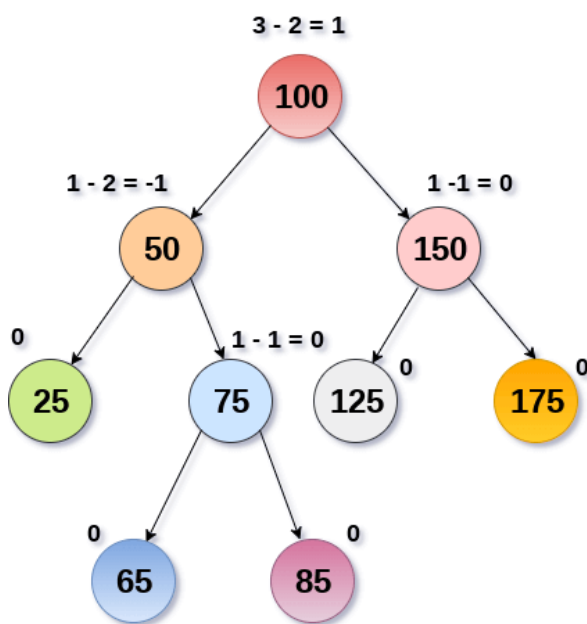
Balance faktor(ozn. BF) sa počíta na základe jednoduchého vzorca

$BF = \text{Hĺbka ľavého podstromu} - \text{Hĺbka pravého podstromu}$ (ja som implementoval tento prípad)

alebo

$BF = \text{Hĺbka pravého podstromu} - \text{Hĺbka ľavého podstromu}$

Ukážka výpočtu BF:



AVL Tree

Implementoval som nasledujúcu a vyžadujúcu funkcionality cez uvedené funkcie :

1. `struct HEAD* placeKey(struct HEAD* head, int key)`
2. `int keySearch(struct HEAD *head, int key)`
3. `struct HEAD *rightRotate(struct HEAD *goDown)`
4. `struct HEAD *leftRotate(struct HEAD *goDown)`

Každú funkciu sa posnažím popísať, čo najjednoduchšie.

Prvá funkcia placeKey

Táto funkcia zabezpečuje takzvané insertovanie prvku do stromu.

V prípade, že pointer ešte predtým nebol inicializovaný tak ho vytvorí a prvý vkladany key / prvok (ďalej už len key) sa stáva koreňom / uzlom stromu a funkcia returnuje pointer na tento koreň / uzol. Koreňom sa vytvára iba prvom, pri ostatných sa v podstate tvoria uzly.

```
if (head == NULL) {
    struct HEAD* head = (struct HEAD*) malloc(sizeof(struct HEAD));
    head->key = key;
    head->left = head->right = NULL;
    head->height = 1;
    return head;
}
```

V prípade, že už strom bol predtým inicializovaný začína sa teda vyhľadávať vhodné miesto pre daný key v strome. V prípade, že narazíme ale na duplikát sa nič nedeje a returnujeme miesto obsahu duplikátu. Následne sa vykoná porovnanie key-u a hodnoty nášeho aktuálneho uzlu. V prípade, že key je menší ako aktuálny uzol tak sa vnárame doľava a začíname rekurzívne volať túto funkciu, kde ako argument bude stále rovnaký key ale začínajúci pointer už nebude ako pri prvom volaní koreň ale jeho potomok. V prípade, že key je väčší ako hodnota aktuálneho uzlu tak sa vnárame doprava.

```
if(key == head->key) return head;
if(key < head->key) head->left = placeKey(head->left, key);
if(key > head->key) head->right = placeKey(head->right, key);
```

Key sa teda vloží tak, že sa vnárame až dovtedy pokiaľ nenájdeme duplikát alebo vnorenie sa doprava, či doľava nebude null a vtedy vytvorí daný uzol s žiadaným keyom.

Po vykonaní tejto rekurzie je potrebné update-ovať hĺbku uzlu, kde sa nachádzame a tú vyrátame ako väčšiu hodnotu hĺbky z child uzlov, na ktoré aktuálny uzol ukazuje + 1 z dôvodu, že hĺbku začíname od 1 a nie od 0.

```
int max = 0;
if(head->left!=NULL) max= head->left->height;
if(head->right!=NULL)
{
    if(max<head->right->height) max = head->right->height;
}
head->height = 1 + max;
```

Následne je potrebné update-ovať hodnotu balance factor-u. Ktorý v prípade, že sú oba child uzly prázdne bude 0 a prípade, že sú tam vypočíta sa spomínaným vzorcom vyššie.

```
if(head->left!=NULL && head->right!=NULL) balanceFactor = head->left->height - head->right->height;
else if(head->left!=NULL) balanceFactor = head->left->height;
else if(head->right!=NULL) balanceFactor = - head->right->height;
else balanceFactor = 0;
```

Posledná časť, ktorá zostáva je následná kontrola, či sa neporušilo vyváženie stromu (BF je rôzny od 1 | 0 | -1). V prípade, že je balance menší alebo rovný -2 znamená to, že daný uzol je prevážený smerom doprava. V opačnom prípade, že BF je rovný alebo väčší ako 2 je prevážený doľava.

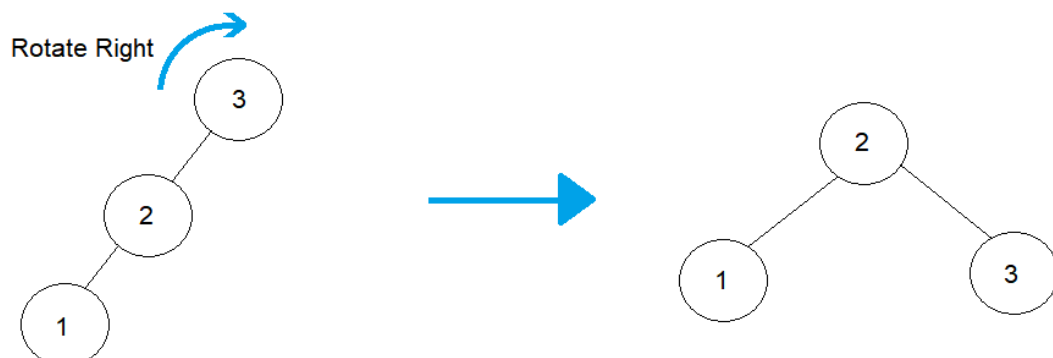
```
if (balanceFactor <= -2)
{
    if (key > head->right->key)
    {
        return leftRotate(head);
    }
    else if (key < head->right->key)
    {
        head->right = rightRotate(head->right);
        return leftRotate(head);
    }
}

if (balanceFactor >= 2)
{
    if (key < head->left->key)
    {
        return rightRotate(head);
    }
    else if (key > head->left->key)
    {
        head->left = leftRotate(head->left);
        return rightRotate(head);
    }
}
```

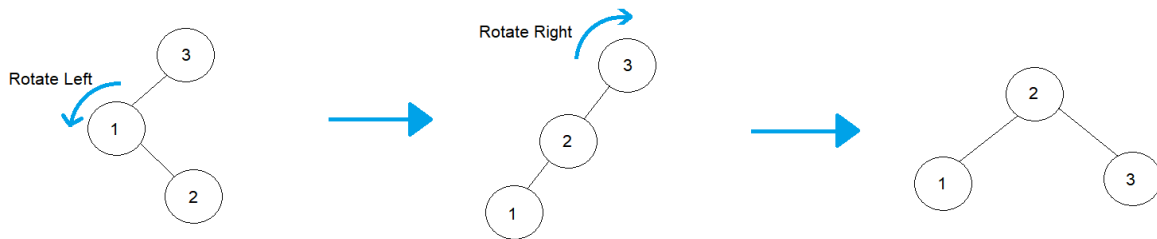
Rozlišujeme tu ale 2 prípady a v každom prípade 2 menšie podprípady.

V 1. prípade, ide o preťaženie ľavú stranu, takže $BF \geq 2$ a je teda potrebné to vyvážiť. Avšak je dôležité kam umiestnime key. Preto ešte rozlišujeme ešte dva menšie prípady.

Prvý menší prípad je taký, že na základe porovnania urobíme len bežnú rotáciu doprava. Vyváženie sa rozhoduje na základe porovnania key-u a podstromov. Dodávam grafický opis situácie ku kódu zobrazenému vyššie.

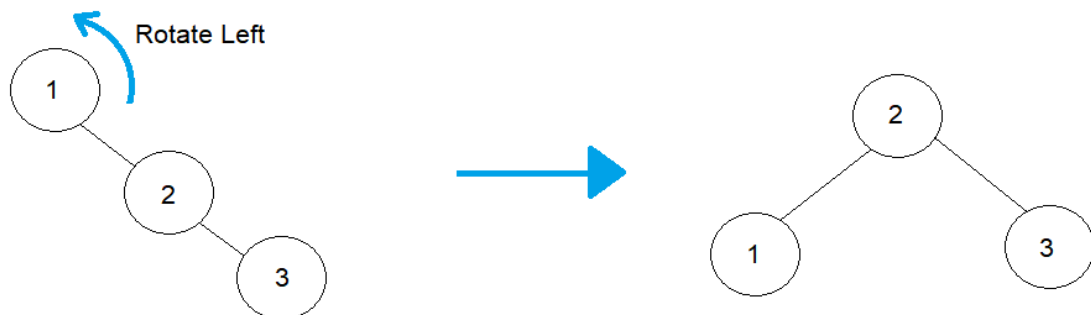


Avšak v druhom prípade, je potrebné urobiť najprv ľavú rotáciu a potom pravú rotáciu aby došlo ku korektnému vyváženiu. Vyváženie sa rozhoduje na základe porovnania key-u a podstromov.

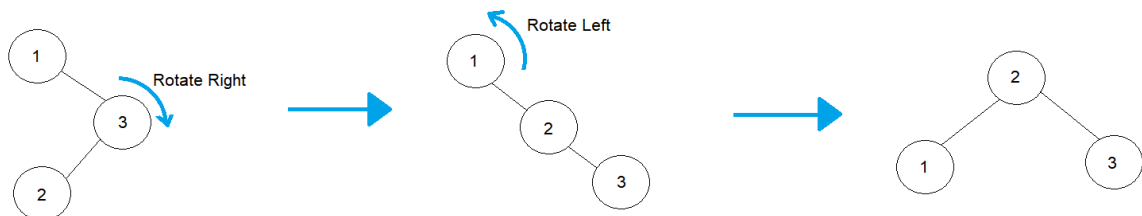


V 2. prípade, ide o preťaženie pravú stranu, takže $BF \leq -2$ a je teda potrebné to vyvážiť. Avšak je dôležité kam umiestnime key. Preto ešte rozlišujeme ešte dva menšie prípady.

Prvý menší prípad je taký, že urobíme len bežnú rotáciu doľava.



Druhý menší prípad je obdobne taký, že treba urobiť 2 rotácie. Najprv rotáciu doprava a potom rotáciu doľava. V



Všetky rotácie som znázornil graficky pre lepšie pochopenie, čo sa deje. Po týchto krokoch ostnú všetky BF z $\{-1, 0, 1\}$. Pretože sa to vykoná pre každý uzol v rekurzii.

Druhá funkcia keySearch

Cieľom funkcie je vrátiť 1 ak sa podarí nájsť hľadaný key v strome.

Funkcia je napísaná rekurzívne. V prípade, že je poslaný do funkcie prázdny strom / NULL pointer returnuje -1 (v prípade ak nič nenájde).

V prípade, že narazí na key, ktorý sa nachádza v strome vráti jedna. Inak sa vnára na základe porovnávania veľkosti hodnoty uzla. Aj key väčší vnára sa doprava, ak menší vnára sa doľava.

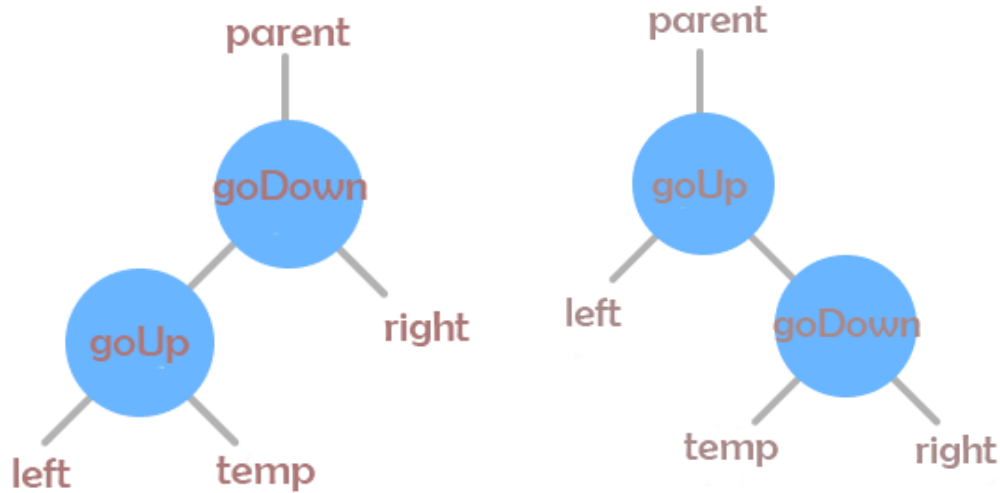
Tretia funkcia rightRotate

Najprv si inicializujeme pointery goUp bude ľavý podstrom goDown uzlu a temp bude pravý potomok stromu goUp.

```
struct HEAD *goUp = goDown->left;  
struct HEAD *temp = goUp->right;
```

Následne urobíme prvú zmenu, kde sa stane goDown uzol pravý podstromom goUp uzlu a pointer na temp, ktorý sme si uložili bude naň teda ukazovať uzol goDown naľavo a to je celá rotácia, ešte je potrebné updateovať hĺbky rotovaných uzlov t.j. goUp a goDown.

Naľavo je pred rotáciou, napravo je po rotácii.

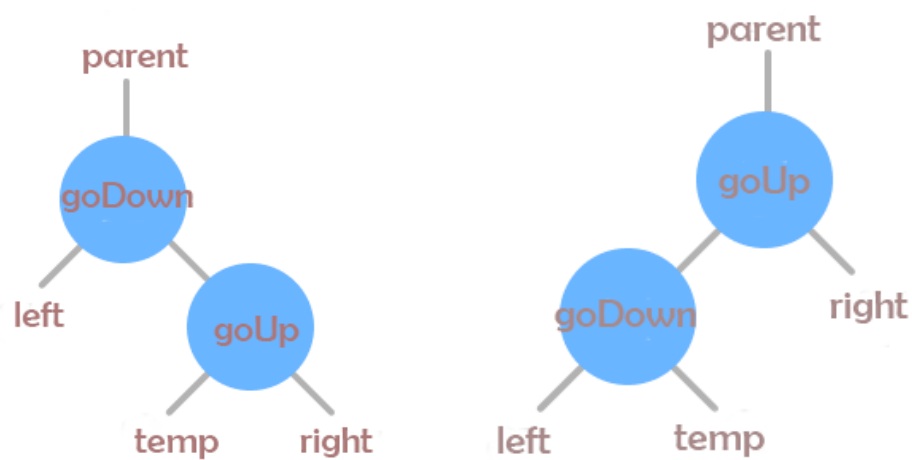


Štvrtá funkcia leftRotate

Najprv si inicializujeme pointery goUp, ktorý ukazuje na pravého potomka stromu goDown a uložíme si pointer goUp ľavého podstromu do temp.

```
struct HEAD *goUp = goDown->right;  
struct HEAD *temp = goUp->left;
```

Následne priradíme ukazatelu na ľavého potomka pointeru goUp pointer na goDown a uzlu goDown priradíme jeho pravému potomkovi uložený pointer na temp. Ukážka(naľavo začiatkový strom a napravo je už zrotovaný strom).



Program 2(1B) – Binárny vyhľadávací strom s vyvažovaním SPLAY (prevzatý)

Prevzatý program je od :

An implementation of top-down splaying; D. Sleator sleator@cs.cmu.edu ; March 1992

Táto implementácia obsahuje aj funkciu delete ale tú popisovať nebudem. Vykonával som menšie úpravy a zmenil som obsah mainu pre účely testovania. Moje zásahy nijako nezlepšili ani neovplyvnili funkčnosť programu. Ide o zásahy pre možnosť vykonania testov.

Je oproti predošlej implementácii AVL stromu rýchlejší, čo sa týka vkladania, pretože pri vkladaní nevyvažuje celý strom ale iba premiestňuje daný key na vrchol stromu, čiže koreň. Ale z dôvodu nie natoľko úspešného a precízneho vyvažovania by mal byť o niečo pomalší pri vyhľadávaní. Pokúsim sa túto skutočnosť overiť pri jednotlivých testoch.

Funkcia insert

Argumenty vkladania sú teda key and strom.

Na začiatku sa vytvoril nový prázdny strom. V prípade, že bol NULL tak sa vypíše, že došla pamäť a vyskočí sa s programu.

V prípade, že ako argument prišiel prázdny strom / uzol, vytvorí sa teda koreň / nový uzol, ktorý bude mať ľavých aj pravých potomkov null. A vráti sa pointer na tento strom.

V prípade, že tak nebolo pokračuje sa a spustí funkcia, ktorá vyhľadá miesto v strome, kam sa má uložiť zadaný key.

V prípade, že key je menší ako hodnota nájdeného ukazovateľa na strom kam sa má vložiť tak sa do nového stromu naľavo vloží ľavý potomok nájdeného, doprava sa uloží nájdený strom a naľavo pôvodného stromu sa uloží NULL aby stále neukazoval na svojho pôvodného potomka naľavo.

V prípade, že key je väčší ako hodnota nájdeného ukazovateľa na strom kam sa má vložiť tak sa do nového stromu napravo vloží pravý potomok nájdeného, doľava sa uloží nájdený pointer na strom a doprava pôvodného stromu sa dá NULL aby stále neukazoval na svojho pravého potomka.

Další prípad je ten, že key je rovnaký a vtedy sa nedeje nič. Len sa vyskočí z funkcie a uvoľní sa novovytvorený strom, ktorý sa nakoniec nepoužil. Tuto máme ukážku vyššie popisovaného kódu.

```

if (i < t->item) {
    new->left = t->left;
    new->right = t;
    t->left = NULL;
    size ++;
    return new;
} else if (i > t->item) {
    new->right = t->right;
    new->left = t;
    t->right = NULL;
    size++;
    return new;
} else { /* We get here if it's already in the tree */
    /* Don't add it again */
    free(new);
    return t;
}

```

Funkcia splay

Slúži na vyhľadávanie miesta pre daný key v zadanom strome. V prípade zhody vráti jeho pointer. V prípade, že do funkcie pošleme prázdny ukazovateľ tak ho aj vráti.

```

if (t == NULL) return t;

```

Deklarujeme si pomocný strom N, ktorý má pravého a ľavého potomka NULL.

Následne vbehneme do cyklu, ak je key(resp. i) menší ako hodnota stromu, ktorý prišiel ako argument a hodnota ľavého potomka nie je null tak sa vykoná rotácia alebo linknutie.

Rotácia doprava sa vykoná ak je ľavý potomok väčší. Do pomocnej y sa zapíše potomok pôvodného stromu naľavo, prepíšeme ukazovateľa potomka naľavo pôvodného stromu na pomocnú napravo. A do pomocnej napravo uložíme ukazovateľ na strom. Do ukazovateľa stromu dáme pointer na y.

Potom obdobne ak je key väčší ako strom, ktorý prišiel ako argument a jeho podstrom nie je null tak vykonáme opačnú rotáciu ako predtým, vykonáme rotáciu doľava, pričom predtým sme vykonávali rotáciu doprava. A takto sa rotujeme doľava / doprava až kým nebude ľavý / pravý potom NULL.

Po skončení rotácii linkujeme pomocou

```

r->left = t;
r = t;
t = t->left;

```

linkujeme pomocou tohoto kódu doprava

```

l->right = t;
l = t;
t = t->right;

```

a pomocou tohto kódu linkujeme strom doľava

Program 3(2B)

Vlastná implementácia hašovania s dynamickou tabuľkou pomocou open-addressing a linear probing

Implementoval som tabuľku ako štruktúru, ktorá obsahuje svoju veľkosť a obsahuje pointer na tzv. „array“ typu DATA, čo je struct s obsahom key-u.

```
typedef struct data {  
    int key;  
} DATA;  
  
typedef struct table {  
    int size;  
    DATA **arr;  
} TABLE;
```

Hashovacia funkcia je implementovaná pomocou linear probing. Do funkcie insert alebo search posielam ako argument tabuľku a key, ktorý treba vložiť / nájsť. Tu je ukážka časti kódu, ktorá má na starosti linear probing. Najprv to zahashujem bežne a v prípade potreby hashujem opätovne ale hashkey-u pripočítavam i.

```
unsigned long hash_f(unsigned int key, TABLE *ht)  
{  
    unsigned int hash;  
    hash = key % ht->size;  
    if(hash > 0) return hash;  
    else return -hash;  
}  
  
int insert(TABLE *ht, unsigned int key)  
{  
    int hashkey = hash_f(key);  
    for(i=0; i<ht->size; i++)  
    {  
        index = (hashkey+i)%ht->size;  
        DATA* curr = ht->arr[index];  
        if(curr == NULL)  
            return 1;  
        if(curr->key == key)  
            return 0;  
    }  
    return -1;  
}
```

Funkcia insertToTable

Vstupnými parametrami je tabuľka a key, ktorý treba vložiť do tabuľky.

Urobí sa prvotný hash a následne sa robí linear probing metóda, kde sa snažím nájsť vhodné miesto pre uloženie do tabuľky. V prípade, že počas celého cyklu nenájdem žiadne miesto / nepodariť sa mi vložiť, zväčšujem tabuľku a pokúsim sa vložiť opätovne.

Zväčšovanie tabuľky vrámci funkcie insertToTable

Zväčšovanie tabuľky sa vykoná vtedy ak sa nenájde vhodné miesto pomocou linear probing metódy (tabuľka je teda plná) a dochádza k zavolaniu funkcie, ktorá callocne nový väčší array a zmení hodnotu veľkosti tabuľky na dvojnásobnú. Následne prehashuje starý array do nového, uvoľní starý array, a tabuľke priradí ukazovateľ na nový array. Nová tabuľka je 2*väčšia a obsahuje aj zväčšené prehashované pole. Následne pridá predtým neúspešne vložený prvok.

Duplikáty

Ako aj v predošlých implementáciách, tak aj v tejto sa duplikát nevkladá dvakrát ale ignoruje sa.

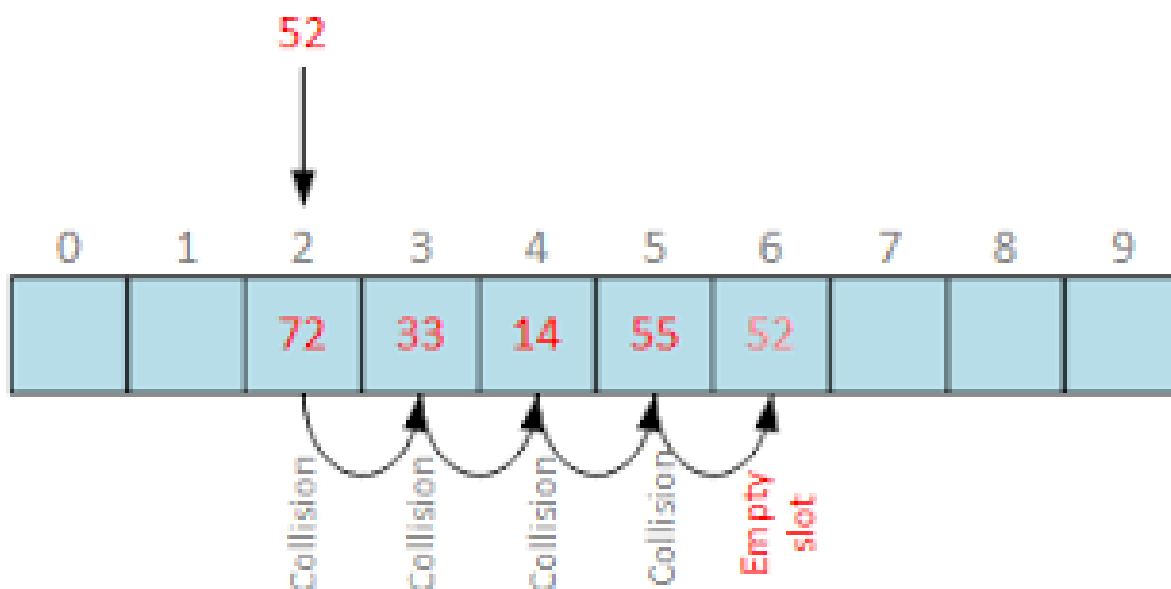
Funkcia searchD

Je veľmi podobná tej insertToTable funkcii, hashujem pomocou linear probing a snažím sa nájsť prvok, v prípade, že nenájdem v celej tabuľke na vhodných miestach returnujem NULL.

V prípade, že nájdem vraciam pointer na nájdený key, ktorý prišiel ako hľadaný argument do funkcie.

Ukážka linear probing-u

V prípade, že key 52 dostal rovnaký hash index ako 72, inkrementuje sa i vo fore a posúva sa doprava. V prípade, že aj tam je kolízia, posúva sa viac doprava a takto až kým nenájde voľné miesto alebo neskončí tabuľka, kedy prichádza k zväčšovaniu tabuľky.



Program 4(1B)

Prevzatá implementácia hašovania pomocou chaining metódy

Prebraté z

<http://entusiaststudent.blogspot.com/2017/03/hashing-with-separate-chaining-c-program.html>

Implementácia má napevno danú veľkosť tabuľky. Implementácia netvorí vyvážené binárne stromy pri tzv. chainingu dát, ktorým bola vyrátaná rovnaká pozícia v hash tabuľke ale tvorí bežný spájaný zoznam s ukazovateľmi na next. Funkciu display v mojich testoch nikde nevyužívam, a preto ju nebudem ani vysvetľovať ale nechcem robiť nepotrebné zásahy do implementácie autora jej mazaním.

Funkcia insertKey

Parameter je key, ktorý je potrebné vložiť do tabuľky. Zahashovaním sa vypočíta index, kam treba key položiť, vytvorí sa nový uzol, do dát uzla sa zapíše hodnota key-u a pointer na ďalší bude NULL. Následne sa pozrie, či je hľadaný index NULL, v prípade, že áno, tak sa do neho uloží predtým vytvorený uzol a v prípade, že nie je NULL vnára sa až kým uzol->next nie je NULL a ak je NULL vloží do neho vytvorený uzol s daným kľúčom.

Funkcia search

Dostáva ako parameter key, ktorý rovnako ako v inserte zahashuje, dostane sa na pozíciu kam sa vkladal a ak tam nie je nič returnuje 0. V prípade, že nájde key returnuje 1. V prípade, že sa na prvej pozícii nenachádza NULL ale nenachádza sa tam ani hľadaný key tak sa vnára až kým nepríde ku koncu zoznamu a ak to nie je ani tam returnuje 0.

Testovanie (4B)

Urobil 7(nastaviteľných 6) plošných testov, kde každý test urobí pre 4 štruktúry 2 menšie podtesty (približne 56 testov), testuje rýchlosť vkladania a rýchlosť vyhľadávania. Snažil som sa nájsť vhodné a nevhodné / vhodné scenáre pre jednotlivé štruktúry, aby sa napr vytvárali dlhé vetvy, aby sa hľadali prvky nedávno pridané a podobne. Cieľom testovania bolo porovnať jednotlivé štruktúry a ich efektívnosť pre dané testy.

Test 1

V tomto teste som vkladal a hľadal 5 miliónov sortnutých prvkov od najmenšieho po najväčší.

Vkladal som teda 0,1,2,3...

```
for(i=0;i<5000000;i++)  
{  
    insertKey(i);  
}  
  
for(i=0;i<5000000;i++)  
{  
    search(i);  
}
```

Z prvých dvoch výpisov vidíme, že AVL strom vkladá o niečo pomalšie ako SPLAY strom a to hlavne z dôvodu, že SPLAY strom má menej striktné vyvažovanie, čo sa ale neskôr ukázalo pri hľadaní prvkov toho istého druhu, kde bol značne lepší AVL strom vyvážený, a to zabezpečuje zložitosť $O(\log(N))$. Stromy boli oproti hash štruktúram jasne rýchlejšie. Všetky testy som začínal v prebratej hash implementácii s tabuľkou o veľkosti 100k a vlastná implementácia hash s tabuľkou veľkosti 100, ktorá sa dynamicky zväčšovala. V chaining metóde môžeme vidieť dosť dlhý čas pri hľadaní prvkov a to najmä z dôvodu, že pri veľkosti 5 miliónov prvkov a tabuľke veľkosti 100 tisíc vznikali dlhé chainy a to trvá značne dlhšie oproti ostatným metódam.

```
C:\Users\Rafaj-PC\Desktop\projekt\main.exe  
Tree - AVL method (vlastna implementacia)  
Inserting sorted array of 5000000 values ...  
It took me 389 clicks (0.389000 seconds).  
Searching array of 5000000 values ...  
It took me 37 clicks (0.037000 seconds).  
-----  
Tree - SPLAY method (prebrata implementacia)  
Inserting sorted array of 5000000 values ...  
It took me 303 clicks (0.303000 seconds).  
Searching array of 5000000 values ...  
It took me 96 clicks (0.096000 seconds).  
-----  
Hash - Open-adressing linear probing method (vlastna implementacia)  
Inserting sorted array of 5000000 values ...  
It took me 883 clicks (0.883000 seconds).  
Searching array of 5000000 values ...  
It took me 69 clicks (0.069000 seconds).  
-----  
Hash - chaining method (prebrata implementacia)  
Inserting sorted array of 5000000 values ...  
It took me 1937 clicks (1.937000 seconds).  
Searching array of 5000000 values ...  
It took me 1453 clicks (1.453000 seconds).
```

Test 2

Vkladal a hľadal som N prvkov (v tomto prípade 2 milióny). Výsledky testovania boli veľmi podobné s tým prvým testom ale ukázali sa väčšie rozdiely medzi SPLAY a AVL stromom. Kde už jasné, vidíme, že vkladanie do AVL stromu je pomalšie z dôvodu precízneho vyvažovania a na základe toho máme oveľa efektívnejšie vyhľadávanie všetkých prvkov voči SPLAY stromu. Ale najväčšiu zmenu môžeme sledovať pri implementácii hash chaining metódy, kde sa čas ukladania aj hľadania značne znížil a to bude zrejme z dôvodu vytvárania lepšieho a rovnomernejšieho rozloženia „chainov“ v tabuľke. Ostatné parametre a výsledky zostali približne rovnaké ako v teste 1.

```
C:\Users\Rafaj-PC\Desktop\projekt\main.exe
Tree - AVL method (vlastna implementacia)

Inserting values ...
It took me 156 clicks (0.156000 seconds).
Searching values ...
It took me 19 clicks (0.019000 seconds).

-----
Tree - SPLAY method (prebrata implementacia)

Inserting values ...
It took me 120 clicks (0.120000 seconds).
Searching values ...
It took me 40 clicks (0.040000 seconds).

-----
Hash - Open-adressing linear probing method (vlastna implementacia)

Inserting values ...
It took me 440 clicks (0.440000 seconds).
Searching values ...
It took me 30 clicks (0.030000 seconds).

-----
Hash - chaining method (prebrata implementacia)

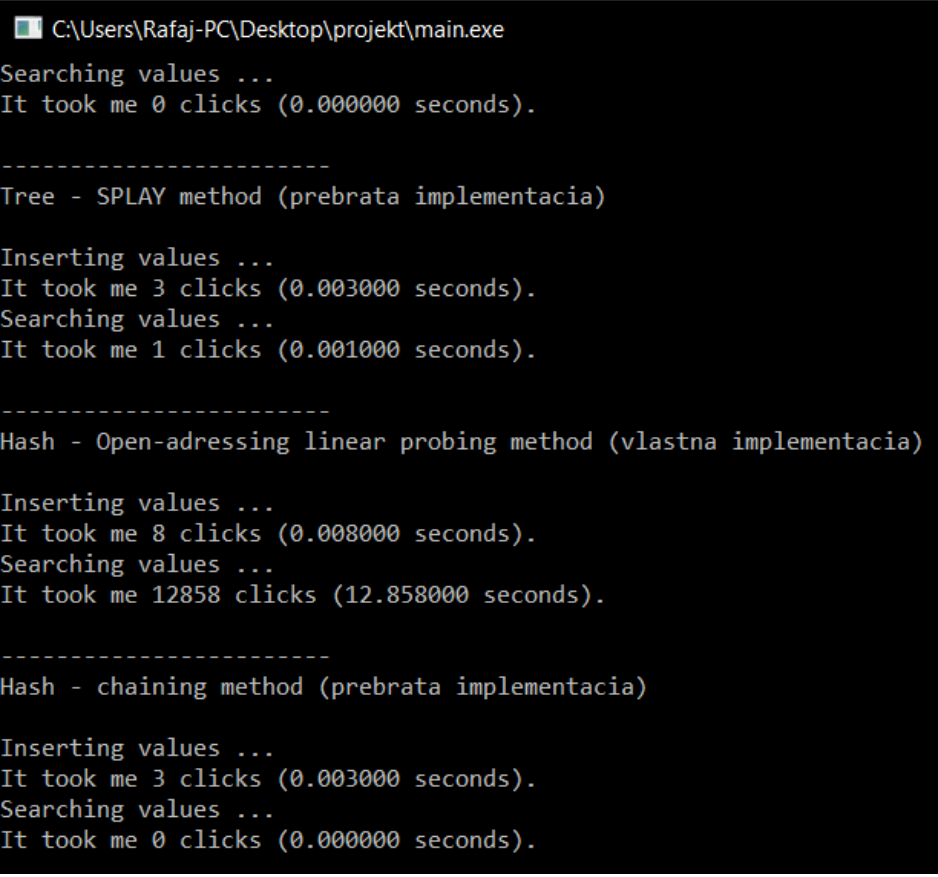
Inserting values ...
It took me 298 clicks (0.298000 seconds).
Searching values ...
It took me 163 clicks (0.163000 seconds).
```

Test 3

V tomto teste vkladali a hľadali rozptýlené prvky. Pomocou vzorca :

```
METODA TESTOVANIA
for( i = N/2 ; i >= 1 ; i--)
{
    if( N%2 == 0)insertKey(i);
    if( N%2 == 1)insertKey(N-i);
}
```

Pri tomto teste som vkladal aj hľadal 100 tisíc prvkov. Všetky implementácie to zbehli relatívne rýchlo až na vlastnú implementáciu open-adressing hashovania, kde hľadanie trvalo voči ostatným veľmi dlho a bude to zrejmé z titulu toho, že linear probing metóda tam vytvárala veľké cluster. To v skratke znamená, čím viac kľúčov príde tým väčšia je šanca, že dostanú hash, kde už niečo je a je potom potrebné sa posúvať až kým sa nenájde voľné miesto, čo trvá o dosť viac času. A pri hľadaní je už väčšina políček obsadená, a preto to trvá relatívne dlho.



```
C:\Users\Rafaj-PC\Desktop\projekt\main.exe
Searching values ...
It took me 0 clicks (0.000000 seconds).

-----
Tree - SPLAY method (prebrata implementacia)

Inserting values ...
It took me 3 clicks (0.003000 seconds).
Searching values ...
It took me 1 clicks (0.001000 seconds).

-----
Hash - Open-adressing linear probing method (vlastna implementacia)

Inserting values ...
It took me 8 clicks (0.008000 seconds).
Searching values ...
It took me 12858 clicks (12.858000 seconds).

-----
Hash - chaining method (prebrata implementacia)

Inserting values ...
It took me 3 clicks (0.003000 seconds).
Searching values ...
It took me 0 clicks (0.000000 seconds).
```

Test 4

V tomto teste som testoval metódu vlož & nájdi pre sortnuté pole od najmenšieho po najväčšie pre N prvkov (v tomto prípade 2 milióny).

```
for( i=0 ; i<N ; i++)  
{  
    insertKey(i);  
    search(i);  
}
```

Môžeme vidieť, že najlepšie dopadol SPLAY strom, ktorý vložil všetky rozhodne najrýchlejšie pre jeho nie tak precízne vyvažovania a v SPLAY strome sa prvky vložené nedávno nájdu rýchlo, čo môžeme vidieť aj v tomto teste, pretože tie prvky sú ešte buď teda koreňom alebo v blízkosti koreňa.

```
C:\Users\Rafaj-PC\Desktop\projekt\main.exe  
Tree - AVL method (vlastna implementacia)  
Inserting & searching values ...  
It took me 168 clicks (0.168000 seconds).  
-----  
Tree - SPLAY method (prebrata implementacia)  
Inserting & searching values ...  
It took me 153 clicks (0.153000 seconds).  
-----  
Hash - Open-adressing linear probing method (vlastna implementacia)  
Inserting & searching values ...  
It took me 461 clicks (0.461000 seconds).  
-----  
Hash - chaining method (prebrata implementacia)  
Inserting & searching values ...  
It took me 401 clicks (0.401000 seconds).
```


Test 4.1

Pri 20M prvkoch už vidno, že splay je jasne najrýchlejší pri vkladaní a hľadani nedávno vložených prvkov. Najhoršie je na tom chaining metóda, ktorá vytvára dlhé vety a je časovo náročné ich prejsť.

```
C:\Users\Rafaj-PC\Desktop\projekt\main.exe
Tree - AVL method (vlastna implementacia)

Inserting & searching values ...
It took me 1669 clicks (1.669000 seconds).

-----
Tree - SPLAY method (prebrata implementacia)

Inserting & searching values ...
It took me 1422 clicks (1.422000 seconds).

-----
Hash - Open-adressing linear probing method (vlastna implementacia)

Inserting & searching values ...
It took me 3845 clicks (3.845000 seconds).

-----
Hash - chaining method (prebrata implementacia)

Inserting & searching values ...
It took me 68771 clicks (68.771004 seconds).
```

Test 5

Vkladal som N prvkov (v tomto teste 600 tisíc) kde i je zvyšované až do N od 0 a je roztrúsené spôsobom $i*3$ a $i*2$, čiže vkladane prvky budú vyzerat :

0, 0, 3, 2, 6, 4, 9, 6, atď..

Vo výpise môžeme vidieť, že AVL sa podarilo prekonať rýchlosť vkladania aj oproti SPLAY stromu a to z toho titulu, že prvky vkladane týmto spôsobom nespôsobujú potrebné až toľké vyvažovanie pri ich vkladani. Pri vyhľadávaní už je to obdobne rýchlejšie ako SPLAY strom. Najhorší výkon pri vkladani môžeme vidieť pri open-adressing metóde a to zrejme z toho titulu, že vznikalo viacero kolízií a následne sa tvorili dlhé blocky s veľkými medzerami a hľadanie voľného miesta so zväčšovaním zabralo veľmi veľa času. Pri hľadaní už vidím jasné zrýchlenie z titulu, že všetky prvky už boli uložené a môžeme predpokladať, že tabuľka bola aspoň z polovica plná. V prípade chaining metódy s veľkosťou vkladanie nemalo až také problémy aj pri určitých kolíziách sa vytvorili len menšie chainy, ktoré sa dali potom aj rýchlo dohľadať.

```
C:\Users\Rafaj-PC\Desktop\projekt\main.exe
Tree - AVL method (vlastna implementacia)

Inserting values ...
It took me 50 clicks (0.050000 seconds).
Searching values ...
It took me 4 clicks (0.004000 seconds).

-----
Tree - SPLAY method (prebrata implementacia)

Inserting values ...
It took me 54 clicks (0.054000 seconds).
Searching values ...
It took me 11 clicks (0.011000 seconds).

-----
Hash - Open-adressing linear probing method (vlastna implementacia)

Inserting values ...
It took me 70238 clicks (70.237999 seconds).
Searching values ...
It took me 10 clicks (0.010000 seconds).

-----
Hash - chaining method (prebrata implementacia)

Inserting values ...
It took me 76 clicks (0.076000 seconds).
Searching values ...
It took me 29 clicks (0.029000 seconds).
```

Test 6

Vkladanie rozptýleného poľa väčších intov. Pomocou vzorca :

```
for(i=N;i>0;i--)
{
    if(i%2==0)insertKey(i);
    else insertKey(2*i);
}
t = clock() - t;
```

```
for(i=N;i>0;i--)
{
    if(i%2==0)search(i);
    else search(2*i);
}
```

Suverénne najhorší výkon mala implementácia s linear-probing metódou, kde jej dosť dlho trvalo nájsť vhodné miesta pre prvky a to zrejme z titulu, že pri hashovaní vznikol obrovský počet kolízií, pretože prvky boli väčšie ako rozsah tabuľky a to pri hashovaní tvorilo veľa rovnakých indexov, čo následne tvorilo tzv. „clustery“ a potom sa to odrazilo v čase. Pri hľadaní sa to stalo opäť a cez toľko kolízií sa to hľadalo dlho, avšak kratšie ako vkladanie, ktoré robilo ešte pár operácií navyše. Jemný výkyv malo aj hľadanie pri AVL strome, čo zrejme spôsobilo to, že sa takmer stále išlo až skoro do najväčšej hĺbky a koreňom bol prvok približne v polovici.

```
C:\Users\Rafaj-PC\Desktop\projekt\main.exe
Tree - AVL method (vlastna implementacia)
Inserting values ...
It took me 7 clicks (0.007000 seconds).
Searching values ...
It took me 705 clicks (0.705000 seconds).

-----
Tree - SPLAY method (prebrata implementacia)
Inserting values ...
It took me 4 clicks (0.004000 seconds).
Searching values ...
It took me 0 clicks (0.000000 seconds).

-----
Hash - Open-adressing linear probing method (vlastna implementacia)
Inserting values ...
It took me 1974 clicks (1.974000 seconds).
Searching values ...
It took me 1015 clicks (1.015000 seconds).

-----
Hash - chaining method (prebrata implementacia)
Inserting values ...
It took me 3 clicks (0.003000 seconds).
Searching values ...
It took me 0 clicks (0.000000 seconds).
```

Test 7

Vkladal som N prvkov(v tomto prípade 5M) od najmenšieho po najväčší a následne som dal prehľadávať štruktúry tými istými prvkami ale zostupne(čiže od najväčšieho po najmenšie).

Opäťovne môžeme vidieť o trochu rýchlejší SPLAY strom v tejto situácii, kde ukladal o dosť rýchlejšie a následne vyhľadával podobne rýchlo ako štruktúra AVL stromu. Ale v tejto situácii môžeme vidieť dosť zlý výkon poslednej implementácie hashu pomocou chainingu, kde vznikalo veľké množstvo dlhých vetiev a to zaberalo čas aj pri vkladaní aj pri vyhľadávaní. Z toho vyplýva, že pri veľkom počte voči veľkosti tabuľky je tento hash veľmi neefektívny. Pri open-adressing implementácii môžeme vidieť relatívne rýchle vyhľadávanie v prípade už vložených čísel ale môžeme vidieť relatívne pomalšie vkladanie pre kolízie.

```
Select C:\Users\Rafaj-PC\Desktop\projekt\main.exe
Tree - AVL method (vlastna implementacia)

Inserting values ...
It took me 401 clicks (0.401000 seconds).
Searching values ...
It took me 39 clicks (0.039000 seconds).

-----
Tree - SPLAY method (prebrata implementacia)

Inserting values ...
It took me 314 clicks (0.314000 seconds).
Searching values ...
It took me 37 clicks (0.037000 seconds).

-----
Hash - Open-adressing linear probing method (vlastna implementacia)

Inserting values ...
It took me 957 clicks (0.957000 seconds).
Searching values ...
It took me 116 clicks (0.116000 seconds).

-----
Hash - chaining method (prebrata implementacia)

Inserting values ...
It took me 2344 clicks (2.344000 seconds).
Searching values ...
It took me 1833 clicks (1.833000 seconds).
```

Zhrnutie testov

Po vykonaní desiatky testov je jasne vidieť, že najefektívnejšie sa pri veľkom počte usporiadaných prvkov preukázala implementácia AVL stromu, ktorá síce nevkladá najrýchlejšie ale dá sa povedať, že vyhľadáva najrýchlejšie, pretože najviac krokov k nájdenie bude $O(\log(n))$, čo je veľmi dobré. Pri SPLAY stromoch môžeme vidieť, že majú podobný a porovnateľný výkon s AVL stromami, vkladajú rýchlejšie avšak zväčša hľadajú pomalšie až na prípady, kde im dávame hľadať nedávno vložené prvky. Hash-chaining implementácia je relatívne efektívna pokiaľ vzniká málo kolízií a tabuľka je dostatočne veľká. No v prípade, že vkladáme značne viac prvkov do tejto štruktúry ako je veľkosť jej tabuľky vznikajú dlhé chainy, čo veľmi ovplyvňuje výkon vkladania, či vyhľadávania. Pri open-adressing metóde môžeme vidieť väčšinou problém pri vkladaní, kde často ostávajú veľké medzery a čím je tabuľka plnšia, tým viac kolízií vzniká. Snažil som sa o čo najpresnejšie testy, kde som sa snažil poukázať slabiny, či naopak výhody jednotlivých štruktúr. Všetky testy sú obsiahnuté v jednotlivých implementáciách a následne stačí zavolať pomocou main.c testovacieho súboru jednotlivé testy a tie zbehnú.