Marcel Valent
Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Ilkovičova 2, 842 16 Bratislava 4
xvalentm@stuba.sk
5. Apríl 2021

ZADANIE 2 – Vyhľadávanie v dynamických množinách

Prednášajúci: Ing. Lukáš Kohútka, PhD.
Cvičiaci: Mgr. Martin Sabo, PhD.
Cvičenie: Utorok 16:00

Obsah

1. Zadanie

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, distribúcii hodnôt, vykonávaným operáciám, a pod. V tomto zadaní máte za úlohu implementovať a porovnať tieto prístupy.

Vašou úlohou v rámci tohto zadania je porovnať viacero implementácií dátových štruktúr z hľadiska efektivity operácií **insert** a **search** v rozličných situáciách (operáciu delete nemusíte implementovať):

• (2 body) **Vlastnú implementáciu binárneho vyhľadávacieho stromu** (BVS) s ľubovoľným algoritmom na vyvažovanie, napr. AVL, Červeno-Čierne stromy, (2,3) stromy, (2,3,4) stromy, Splay stromy, ...

• (1 bod) **Prevzatú (nie vlastnú!) implementáciu BVS** s iným algoritmom na vyvažovanie ako v predchádzajúcom bode. Zdroj musí byť uvedený.

• (2 bod) **Vlastnú implementáciu hašovania** s riešením kolízií podľa vlastného výberu. Treba implementovať aj operáciu zväčšenia hašovacej tabuľky.

• (1 bod) **Prevzatú (nie vlastnú!) implementáciu hašovania** s riešením kolízií iným spôsobom ako v predchádzajúcom bode. Zdroj musí byť uvedený.

2. Implementácia

2.1. Binárny strom

Na vlastnú implementáciu binárneho stromusom som si zvolil AVL strom, ktorý je schopný sa sám vyvažovať. Vyvažovanie stromu riešime pomocou rotácii.

V štruktúre je uložená pomocná hodnota, ktorá ukladá počet rovnakých dát, potom výška stromu a nakoniec sú uložené pointre na ľavé a pravé dieťa.

Štruktúra AVL stromu vyzerá takto:

```
typedef struct AVLTree {
    int num;
    short height;
    struct AVLTree *left;
    struct AVLTree *right;
}AVLtree;
```

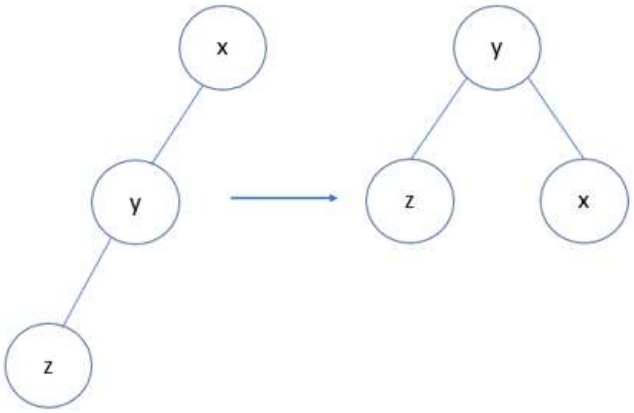
2.1.1. Funkcie AVL stromu

Všetky funkcie, ktoré používam pri riešení AVL stromu

```
int SearchMyAVL(AVLtree *treeInc, int num);
int InsertMyAVL(int numInc, AVLtree **treeInc);
// uvolnenie AVL stromu
void FreeAVLTree(){ ... }
void FreeMyAVLTree(AVLtree *tree){ ... }
//hladanie v AVL strome
int SearchAVL(int num){ ... }
int SearchMyAVL(AVLtree *treeInc, int num){ ... }
int GetMax(int a, int b){ ... }
//Ziskanie vysky stromu
int GetHeight(AVLtree *treeInc){ ... }
//Nastavenie vysky stromu
void SetHeight(AVLtree *treeInc){ ... }
//Nasledovne kody riesia rotáciu AVL stromu
//Lava rotacia
AVLtree* RotateLeft(AVLtree *treeInc){ ... }
//Prava rotacia
AVLtree* RotateRight(AVLtree *treeInc){ ... }
//Lavo-prava rotacia
AVLtree* RotateLeftRight(AVLtree *treeInc){ ... }
//Pravo-lava rotacia
AVLtree* RotateRightLeft(AVLtree *treeInc){ ... }
//Ziskanie vysky potrebnej na balancovanie stromu
int GetSpecialHeight(AVLtree *treeInc){ ... }
//Vypocet vybalancovania stromu
int GetBalance(AVLtree *treeInc){ ... }
//Vlozenie do AVL stromu
int InsertAVL(int num){ ... }
int InsertMyAVL(int numInc, AVLtree **treeInc){ ... }
```

2.1.2. RotateRight(AVLtree *treeInc)

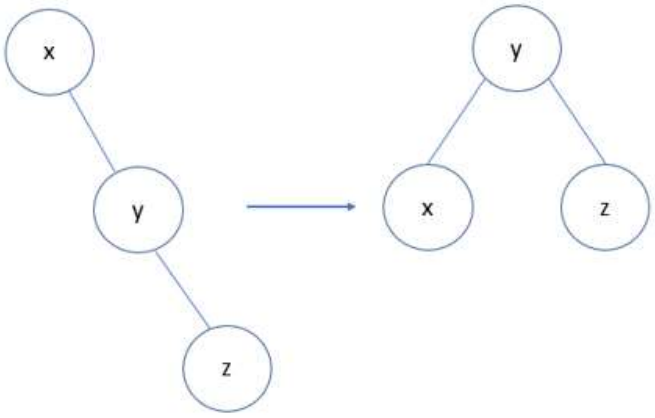
Táto funkcia rieši rotáciu AVL stromu vpravo. Vo funkcii využívame pomocné premenné, kde nastavíme ľavý uzol a pravý uzol, ktorý vznikne z novovytvoreného uzlu. Potom nastavíme uzol na vstupe a ľavý uzol. Nakoniec nastavíme novú výšku stromu a funkcia nám vráti nový uzol.



Obr. 1 Pravá rotácia

2.1.3. RotateLeft (AVLtree *treeInc)

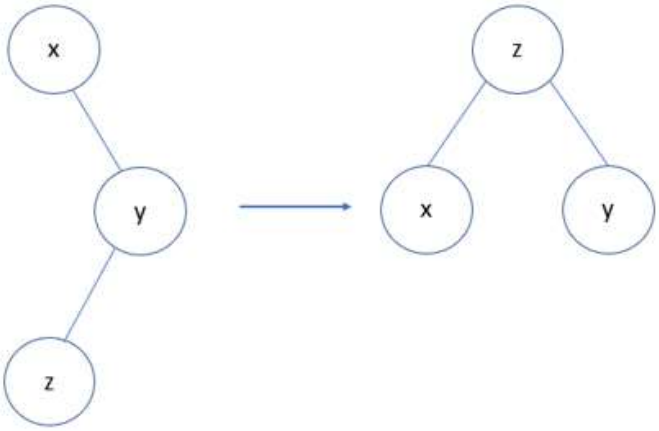
Táto funkcia rieši rotáciu AVL stromu vpravo. Priebeh funkcie je rovnaký ako pri rotácii vpravo. Premenné sa nastavujú rovnako ako pri rotácii vpravo,len v tomto prípade do funkcie berieme ľavý poduzol.



Obr. 2 Ľavá rotácia

2.1.4. RotateRightLeft (AVLtree *treeInc)

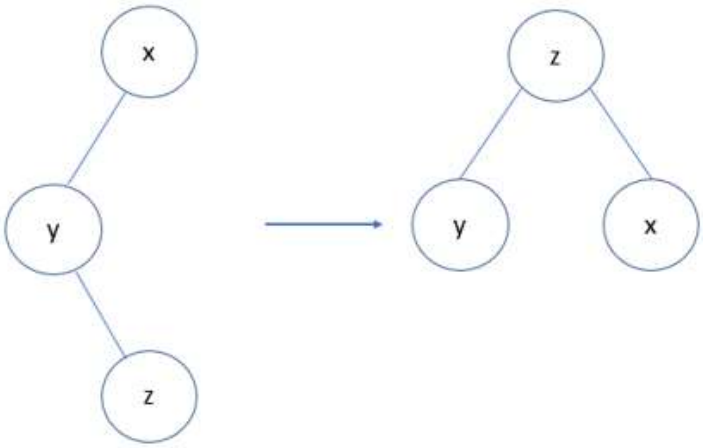
Túto funkciu využívame vtedy, ak do stromu vložíme hodnotu, ktorá je väčšia ako hodnota v jeho koreni, a následne sa vloží hodnota ktorá je menšia ako hodnota vložená predtým. Vykoná sa rotácia vpravo a následne vľavo.



Obr. 3 Rotácia doprava a následne doľava

2.1.5. RotateLeftRight(AVLtree *treeInc)

Túto funkciu využívame vtedy, ak do stromu vložíme hodnotu, ktorá je menšia ako hodnota v jeho koreni, a následne sa vloží hodnota ktorá je väčšia ako hodnota vložená predtým. Vykoná sa rotácia vľavo a následne vpravo



Obr 4. Rotácia vľavo a následne vpravo

2.1.6. Zložitosť vlastnej implementácie

Časová zložitosť: O (log n) kde n je počet prvkov v strome.

Pamäťová zložitosť: O(n) kde n je počet prvkov v strome.

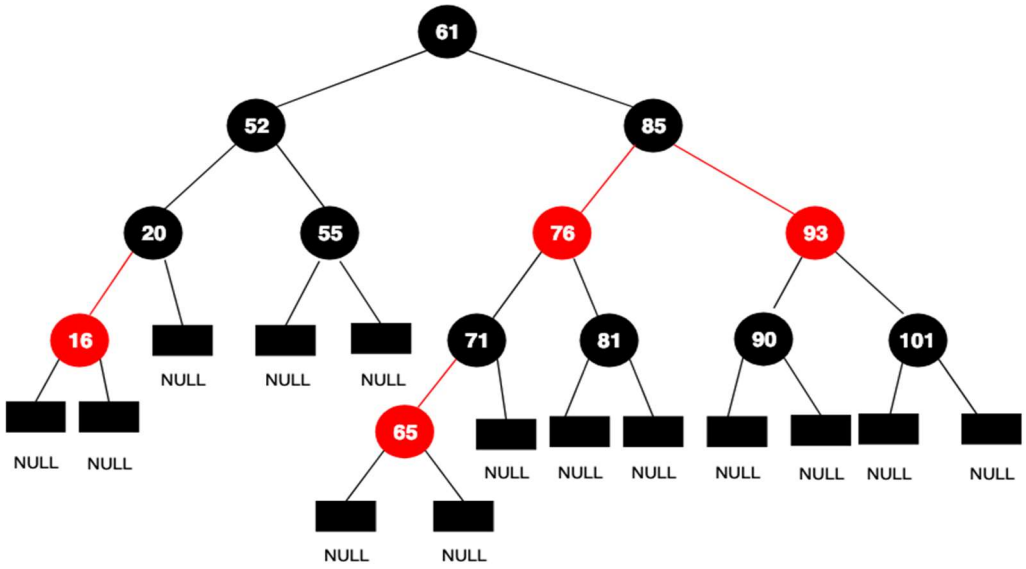
2.2.Prevzatá implementácia Red Black stromu

Ako prevzatú implementáciu binárneho stromu som si zvolil **Red Black strom.**

Red Black strom ma vyriešené balansovanie stromu pomocou farbenia uzlov na čiernu a červenú farbu, s tým, že keďnastane zmena v strome, tak sa aj prefarbia uzly, preto je potrebné mať vyhradený 1 bit na zapamätanie farby stromu.

Červeno – čierny strom musí splniť niekoľko pravidiel:

- Každý uzol je buď čierny alebo červený
- Koreň musí byť vždy čierny
- Listy sú považované za čierne vrcholy
- Každý červený vrchol má 2 čiernych synov
- Každá cesta z vrcholu do ich listov pod nimi obsahuje rovnaký počet čiernych vrcholov



Obr. 5 Jednoduchý príklad RB stromu

Medzi najväčší rozdiel patrí to, že vkladanie (insert) do RB stromu prebehne omnoho rýchlejšie ako pri AVL strome, pretože pri AVL strome prebehne mnoho rotácií.

2.2.1. Zložitosť prevzatej implementácie

Časová zložitosť: O (log n) kde n je počet prvkov v strome.
Pamäťová zložitosť: O(n) kde n je počet prvkov v strome.

2.3. Hashovacia tabuľka

Ide o pole údajov, ktoré sú hashovacou funkciou zakódované do kľúčov. Na základe týchto kľúčov je im potom priradené miesto v tabuľke.

Moja hashovacia funkcia vyzerá nasledovne:

- (num + attempt) % size

2.3.1. Zoznam použitých funkcií

```
//vytvorenie hashovacej tabulky
int CreateTable(int sizeInc){ ... }
//hashovacia funkcia
int HashFunction(int num, unsigned long long attempt, int size){ ... }
//vlozenie hashovacej funkcie
myTable* InsertMyHTFunc(myTable *tmpTable, int number){ ... }
//zvacsenie hashovacej tabulky
void ResizeTable(){ ... }
//hladanie v hashovacej tabulke
int SearchMyHT(int number){ ... }
//Vkladanie do hashovacej tabulky
int InsertMyHT(int number){ ... }
//Uvolnenie hashovacej tabulky
void FreeMyHT(){ ... }
```

2.4.Prevzatá implementácia hashovacej tabuľky

Ako prevzatú implementáciu hashovacej tabuľky som si zvolil hashovaciu tabuľku zo stránky https://github.com/qzchenwl/hashtable

3. Testovanie

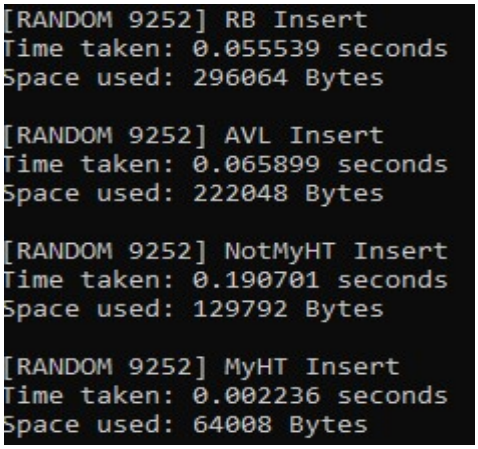
Testovanie prebiehalo na:

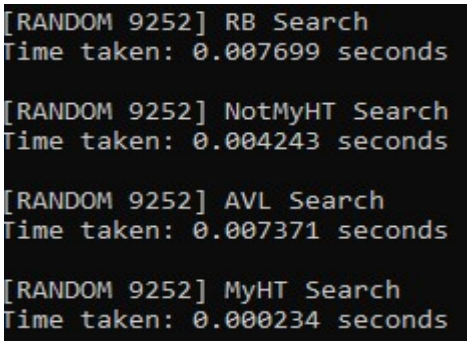
- OS: **Windows 10**
- Procesor: **AMD Ryzen 5 1600 3.5 GHz**

Testoval som nasledujúce scenáre:

- **Vkladanie a hľadanie 10 000 prvkov**, ktoré sú náhodne vygenerované
- **Vkladanie a hľadanie 200 000 prvkov**, ktoré sú náhodne vygenerované

Test 1. Pri tomto teste sme vkladali a hľadali 10 000 prvkov.Následne som si na stránke random.org vygeneroval náhodné číslo 1-10 000, pri ktorom som urobil porovnanie časov insertu a searchu.





Vkladanie

Ako sme si mohli všimnúť, pri malom počte prvkov je RB strom rýchlejší, avšak môj AVL strom viac pamäťovo efektívny. Pri porovnaní hashovacích tabuliek si však môžeme všimnúť,že moja implementácia hashovacej tabuľky je pri malom počte prvkom omnoho rýchlejšia aj pamäťovo efektívnejšia ako prevzatá implementácia.

Hľadanie

Pri hľadaní sú oba stromy porovnateľne rýchle. Pri hľadaní v hashovacích tabuľkách si môžeme všimnúť, že moja implementácia hashovacej tabuľky je niekoľkonásobne rýchlejšia ako prevzatá implementácia.

Test2. Pri tomto teste sme vkladali a hľadali 200 000 prvkov. Následne som si na stránke random.org vygeneroval náhodné číslo 1-200 000, pri ktorom som urobil porovnanie časov insertu a searchu, aby sme lepšie pochopili správanie sa algoritmov pri veľkom počte insertov a searchov.

```
[RANDOM 191000] RB Insert
Time taken: 0.833585 seconds
Space used: 6400000 Bytes

[RANDOM 191000] AVL Insert
Time taken: 1.719523 seconds
Space used: 4800000 Bytes

[RANDOM 191000] NotMyHT Insert
Time taken: 0.503238 seconds
Space used: 0 Bytes

[RANDOM 191000] MyHT Insert
Time taken: 130.147564 seconds
Space used: 2048008 Bytes
```

```
[RANDOM 191000] RB Search
Time taken: 0.206346 seconds

[RANDOM 191000] NotMyHT Search
Time taken: 0.099948 seconds

[RANDOM 191000] AVL Search
Time taken: 0.195487 seconds

[RANDOM 191000] MyHT Search
Time taken: 0.004506 seconds
```

Ako si môžeme všimnúť pri väčšom počte vložených prvkov sú prevzaté implementácie RB stromu a hashovacej tabuľky omnoho rýchlejšie ako moje vlastné implementácie. Pri hľadaní je moja aj prevzatá implementácia binárneho stromu približne rovnako rýchla. Avšak pri hashovacej tabuľke môžeme pozorovať,že vyhľadávanie v mojej hashovacej tabuľke je omnoho rýchlejšie ako pri prevzatej implementácii.

Záver

Cieľom tohto zadania bola vlastná implementácia hashovacej tabuľky a binárneho stromu a ich následné porovnanie s prevzatými implementáciami. Pri tomto porovnávaní a implementovaní sme mohli lepšie pochopiť binárne stromy a hashovacie tabuľky a ich viaceré spôsoby implementovania, ich výhody a nevýhody.