Zadanie 2 – Vyhľadávanie v dynamických množinách

Filip Vida, Fakulta informatiky a informačných technológií STU v Bratislave 12/04/2020

Binary Search Tree

Binárny vyhľadávací strom je dátová štruktúra odvodená od binárneho stromu . Prvky v nej sú usporiadané tak, aby sa urýchlilo vyhľadávanie. Prvky na ľavo od aktuálneho prvku sú vždy menšie ako aktuálny prvok, prvky na pravo sú zas väčšie. Takýto strom sa však môže stať nevyváženým, to znamená že niektorý podstrom určitého prvku je väčší alebo menší než druhý podstrom toho istého prvku. Takéto nevyváženie zvyšuje čas potrebný na vykonanie vyhľadávania, čo nie je žiadúce. Na vyriešienie takéhoto nevyváženia existuje niekoľko riešení, v tomto vypracovaní zadania sa pozriemie na dve.

AVL tree

AVL strom ponúka jedno z riešení na problém nevyváženia. Pri každom prvku stromu sledujeme tkz. balance faktor, ktorý sleduje rozdiel medzi pravým a ľavým podstromom.Insertion prebieha ako pri Binárnom vyhľadávacom strome, no na konci tejto operácie je nutné zmeniť hodnotu balance. Balance hodnoty väčšie ako 1 alebo menšie ako -1 znamenajú, že strom je nevyvážený. Toto napravíme pomocou použitia 1 alebo 2 rotácií.

LEFT Rotation

Túto rotáciu vykonáme vtedy, keď je pravý podstrom väčší ako ľavý.

```
struct node *leftRotate(struct node *root1)
{
    struct node *son = root1->right;
    struct node *grandSon = son -> left;

    root1 -> right = grandSon;
    son ->  left = root1;
    //changeHeight(grandSon);
    changeHeight(root1);
    changeHeight(son);
    return son;
}
```

RIGHT Rotation

Túto rotáciu vykonáme vtedy, keď je ľavý podstrom väčší ako pravý.

```
struct node *rightRotate(struct node *root2)
{
    struct node *son = root2->left;
    struct node *grandSon = son -> right;

    root2 -> left = grandSon;
    son ->  right = root2;
    //changeHeight(grandSon);
    changeHeight(root2);
    changeHeight(son);
    return son;
}
```

Double RIGHT Rotation

Takúto rotáciu treba vykonať na prvok pri ktorom by sme použili RIGHT rotáciu, no jeho ľavý potomok má pravého potomka. Pri tejto situácii použijeme LEFT rotáciu na ľavého potomka, nie na prvkok samotný. Potom použijeme rotáciu RIGHT.

Double LEFT Rotation

Takúto rotáciu treba vykonať na prvok pri ktorom by sme použili LEFT rotáciu, no jeho pravý potomok má ľavého potomka. Pri tejto situácii použijeme RIGHT rotáciu na pravého potomka, nie na prvkok samotný. Potom použijeme rotáciu LEFT.

Red–black tree

Red black tree je vyvážený Binárny vyhľadávací strom ktorého každý prvok obsahuje bit ktorý obsahuje farbu daného prvku. Vyváženie stromu sa dosahuje s využitím týchto farieb. Každý Red-black tree musí spĺňať tieto požiadavky:

1.Každý prvok je red alebo black

2.Root je black

3.Listy sú black

4.Ak je prvok red, jeho potomkovia sú black

5.Každá cesta od prvku ku listu prechádza cez rovnaký poćet black prvkov

Insertion operácia prebieha podobne ako pri Binárnom vyhľadávacom strome, no tu sa nový prkov nepridá ako list, nový prvok nahradí existujúci list a k novému prvku sa pridajú dalšie listy. Po inserte môže nastať niekoľko situácií.

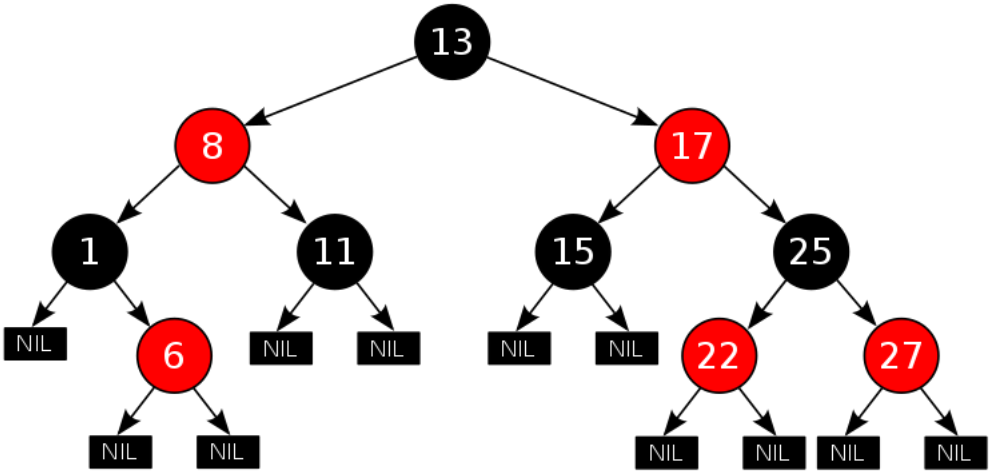
1.N je root

2.Rodič N (P) je black

3.P je red and N's strýko (U) je red

4.P je red a U je black

Každá situácia si vyžaduje vlastnú implementáciu.

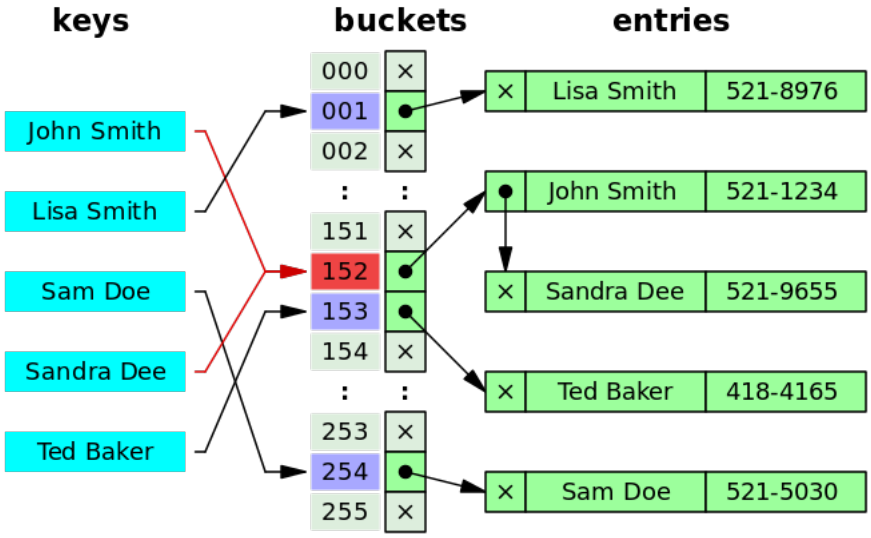


Hash table

Hash table je dátová štruktúra na ukladanie párov dát.Tieto dáta sú uložené v poli.Vkladané dáta sa spravidla volajú key a data.Môžu byť rôzneho dátového typu. Key musí byť unikátny, aby bolo možné dáta nájsť. Key využijeme na tkz. zahashovanie pomocou ktorého určíme miesto, na ktoré vstupné dáta uložíme. Na toto využijeme hashovaciu funkciu, ktorá nám z key spravý číslo, ktoré vieme použiť ako index do tabuľky. 2 rôzne keye môžu avšak mať rovnaký hash a vtedy nastane tkz. kolízia. Kolízie sa dajú riešiť niekoľkými metódami. Uvedieme si 2.

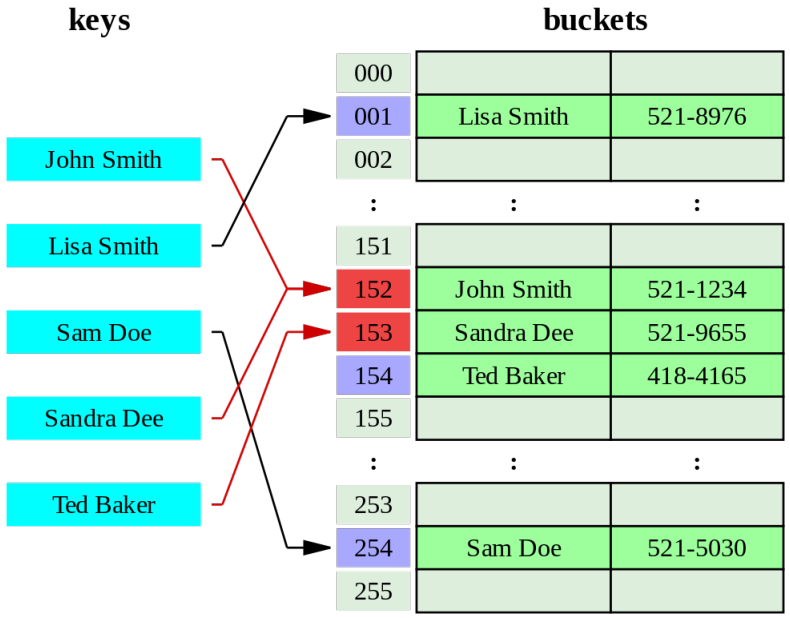
Separate chaining with linked lists

V tomto riešení kolízií je v každom indexe(bucket) uložený linked list. Ak sa stane kolízia, dáta sa pridajú do tohto linked listu.



Open addressing

Pri metóde open addresing spadá na každý index(bucket) len 1 pár dát. Pri kolízii sa v poli posúvame a hladáme najbližšie voľné miesto.



Testy

Testy boli vykonávané automaticky pomocou funkcií typu testHash(int pocetInsertov, int pocetSearchov). Testoval som v množstvách od 1000 do 20000000. Hodnoty boli generované funkciou random, okrem hodnôt pre key pre Hash Tabuľky, kvôli veľmi častým duplikátom. Tu som použil postupnosť. Každý z testov bol vykonaný 50 krát a do grafov bol zapísaný priemer.

Použité implementácie:

Vlastné:

1.AVL Tree

2.Chaining Hash Table using Linked Lists

Prevzaté:

1.Red-black tree

2.Linear Open Addresing Hash Table

Výsledky

RBT vs AVL

RBT a AVL majú všeobecne veĺmi porovnatelné hodnoty. V mojej a prevzatej implementácii sú rozdiely veľmi malé, no RBT pri väčších hodnotách vykonáva insert rýchlejśie, zatial čo AVL je rýchlejší pri funckcii search.

Linear Hash vs Chaining Hash

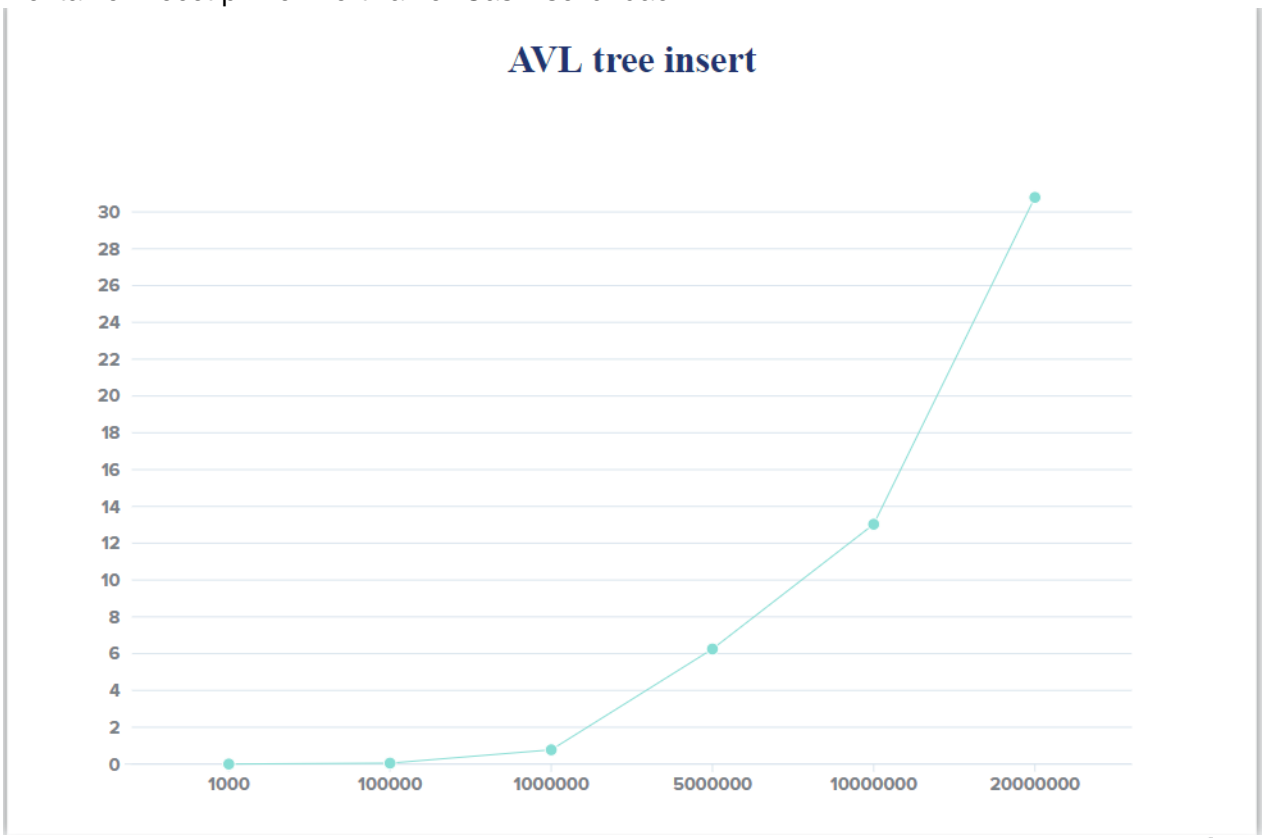
Pri tomto porovnávaní som porovnával tieto implementácie aj z hľadiska toho, že moje implementácia Chaining Hash obsahovala funckiu na zväčšenie, zatial čo prevzatá Linear Hash nie. Toto znamenalo že Chaining Hash bola pamäťovo oveľa úspornejšia. Na druhú stranu si však funkcia resize vyžadovala značný čas a znížila rýchlosť operácie insert. Zároveň veľká prednastavená veľkosť Linear Hash tabuľky znížila počet kolízií a tým pádom aj zvýšila týchlosť funckie search. Využíva však rovnkakú veľkosť pamäte na uchovanie hocijakého počtu dát, čo je neefektívne a môže znamenať že sa tabuľka raz úplne zaplní.

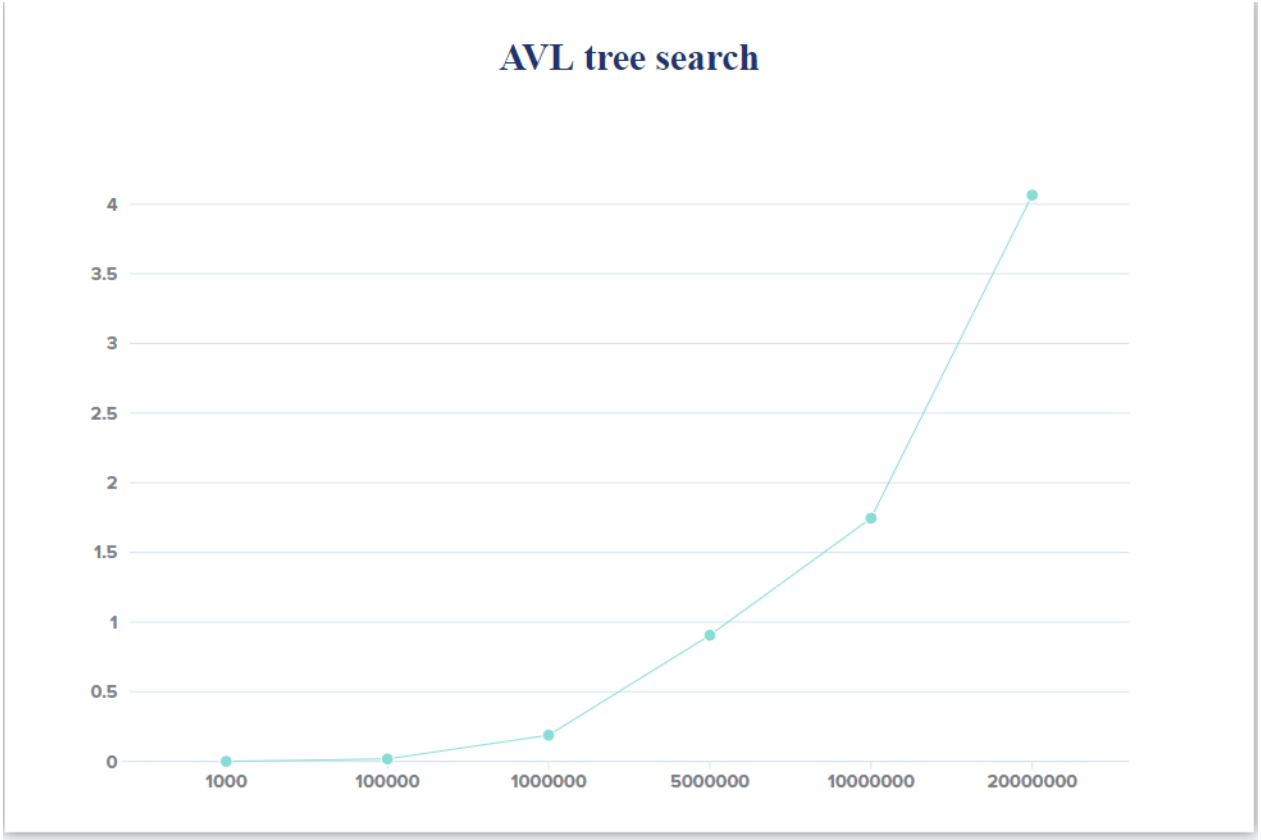
Hash vs Binary Tree

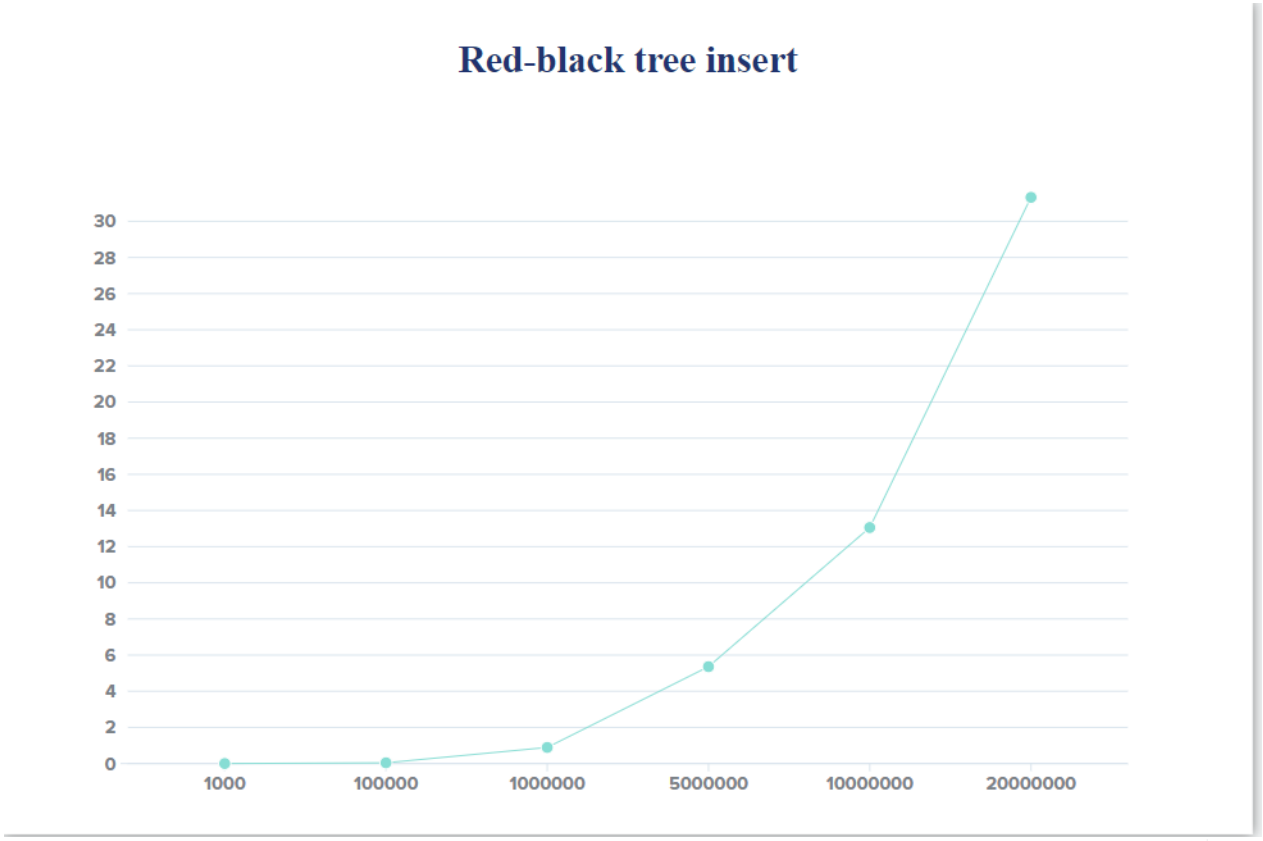
Pri porovnávaní týchto dvoch dátových štruktúr je zjavné že v rýchlosti jednoznačne vyhrávajú Hash Tabuľky s nemalým nadskokom, či už pre operácie insert alebo search .

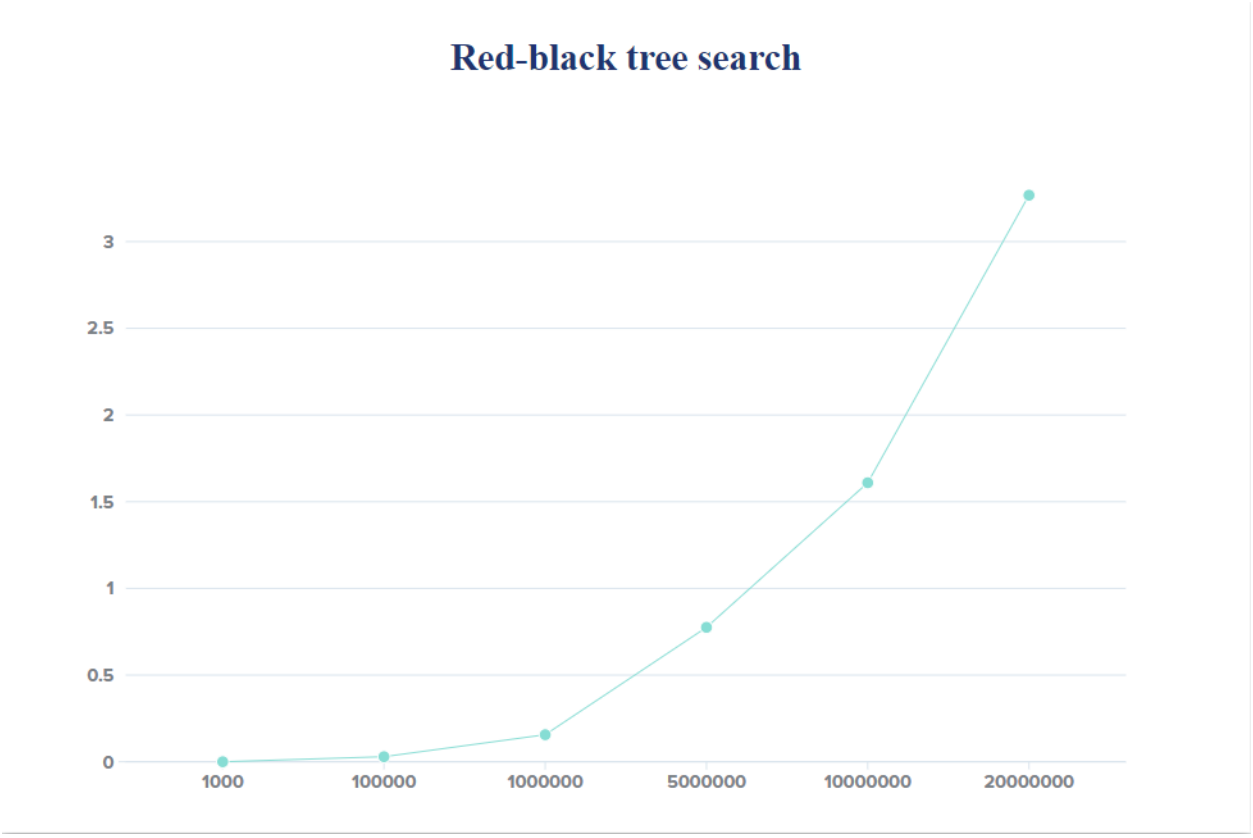
Grafy

Horizontálne: Počet prvkov Vertikálne: Čas v sekundách

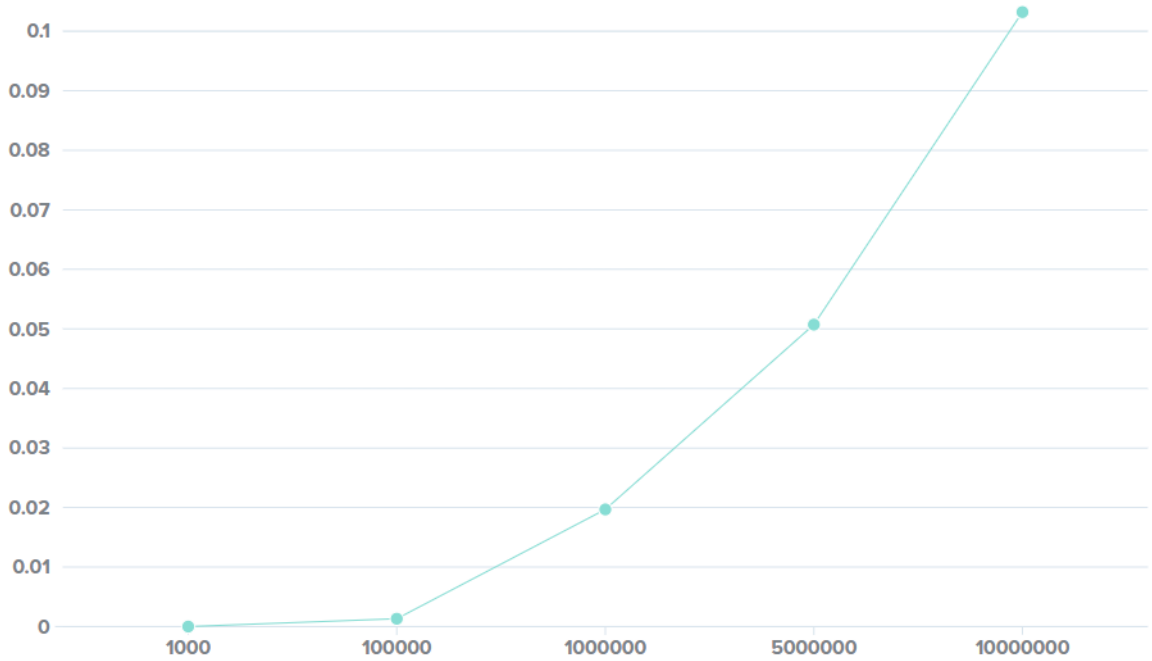




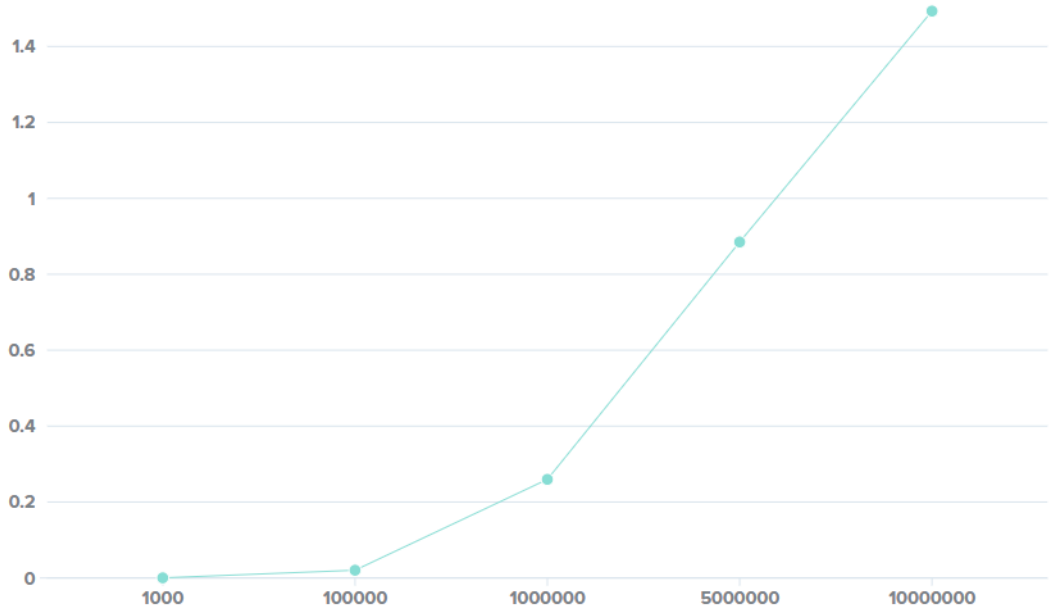

Red-black tree insert
30
28
26
24
22
20
18
16
14
12
10
8
6
4
2
0
1000
100000
1000000
5000000
10000000
20000000


Red-black tree search
3
2.5
2
1.5
1
0.5
0
1000
100000
1000000
5000000
10000000
20000000

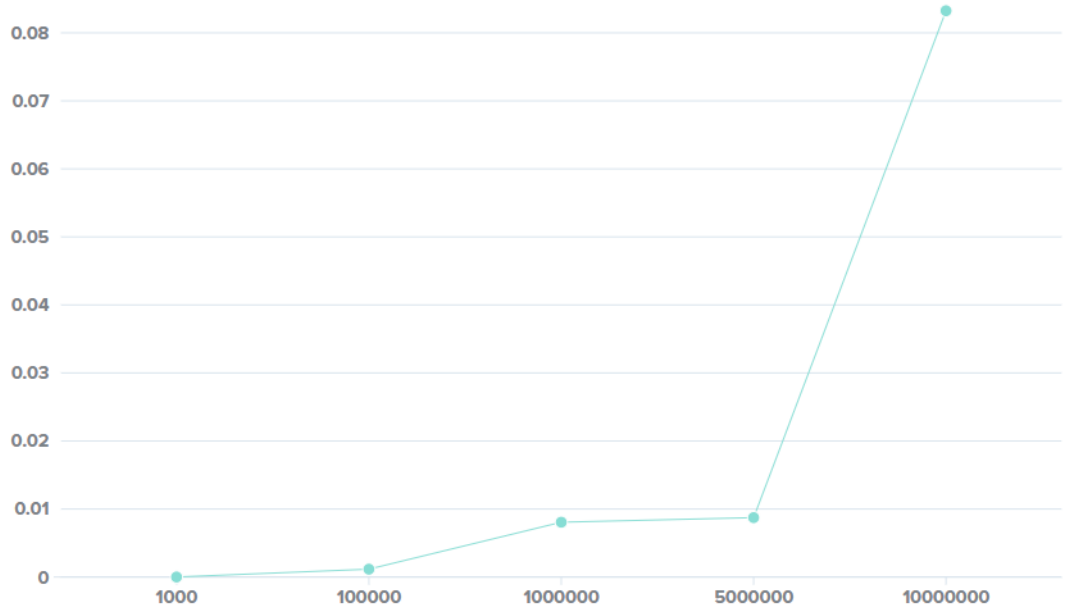
Chaining Hash Search



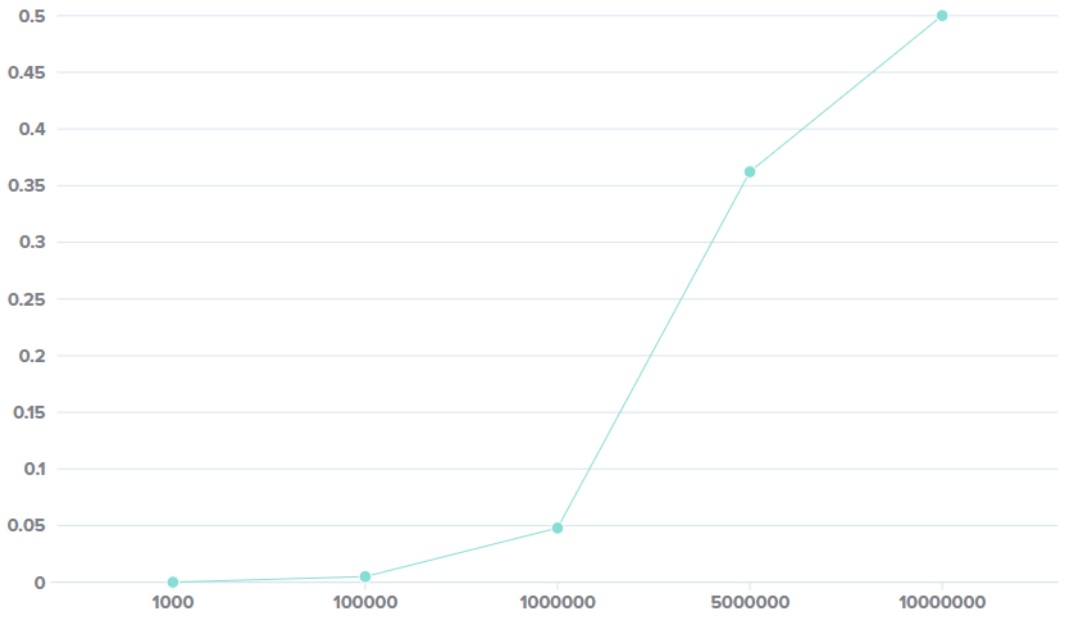
Chaining Hash Insert



Linear Hash Search



Linear Hash Insert



Zdroje

AVL teória a implementácia:http://pages.cs.wisc.edu/~paton/readings/liblitVersion/AVL-Tree-Rotations.pdf
Red-black tree teória: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
Red-black tree prevzatá implementácia: https://gist.github.com/aagontuk/38b4070911391dd280
Linear Hash prevzatá implementácia: https://www.tutorialspoint.com/data_structures_algorithms/hash_table_program_in_c.htm?fbclid=IwAR1kkHSnjl0iRlqCtZAcvmRvmSGDDNfx
Chaining hash teória: https://en.wikipedia.org/wiki/Hash_table
Chaining hash implementácia: prednášky, cvičenia