FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

SLOVENSKÁ TECHNICKÁ UNIVERZITA

Ilkovičova 2, 842 16 Bratislava 4

2021/2022

Dátové štruktúry a algoritmy

Zadanie č.2

Cvičiaci: MSc. Mirwais Ahmadzai

Čas cvičení: Štvrtok 11:00 - 12:50

Vypracoval: Maroš Bednár

AIS ID: 116822

Obsah

Hashovanie

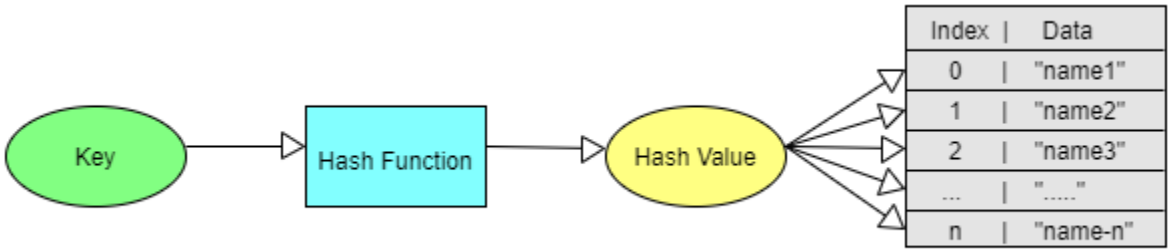
1. Úvod

Hashovacia tabuľka je dátová štruktúra, ktorá má podobné vlastnosti ako pole. Skladá sa z n-pozícií, na ktoré je možné ukladať, mazať či vyhľadávať dáta. Funguje to na báze rýchleho prístupu. V ideálnych prípadoch, kde nenastávajú kolízie je časová zložitosť konštantná O(1). Najhoršie prípady nastávajú pri kolízií, kedy rýchlosť klesá až na O(n). Vtedy prehľadávame každý prvok, ale zvyčajne to trvá kratšiu dobu. Nevýhoda je však pri spotrebe pamäte, kedy sa jej minie viac ako je potreba.

V mojej implementácii využívam pole štruktúr obsahujúcich dáta ako meno a vek.

2. Prvá vlastná implementácia

Spočíva v iteratívnom vyhľadávaní voľného miesta v pamäti. Ak sa takéto miesto dlho vyhľadáva, tabuľka sa automaticky zväčšuje.



Obrázok 1. – Prvá implementácia hashovacej tabuľky

2.1. Hashovacia funkcia

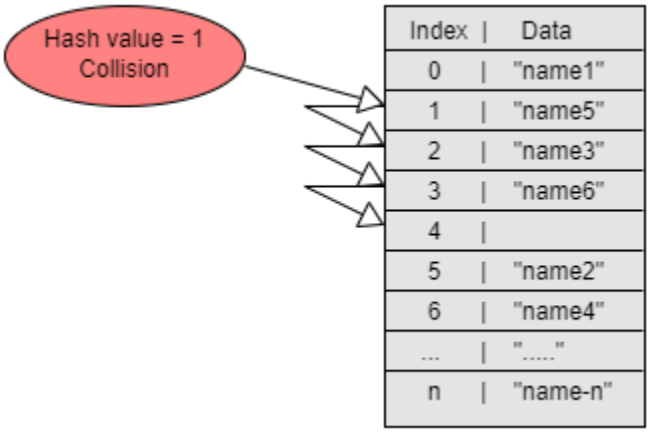
Je to funkcia, ktorá kóduje aktuálne hodnoty a ako produkt vráti celé číslo reprezentujúce pozíciu v tabuľke. Konkrétny index dosiahneme matematickými operáciami násobenia konkrétneho písmena slova zväčšeného o aktuálnu pozíciu v slove ďalším písmenom zväčšeným o aktuálny index – celková dĺžka slova. Ďalej ho násobím prvočíslom, ktoré mi zabezpečí rovnomerné rozloženie a taktiež aktuálnou pozíciou v znaku slove. Následne použitím operácie modulo o veľkosti tabuľky zabezpečíme index z intervalu <0, n>.

2.2. Vkladanie do hashovacej tabuľky

Vkladanie do tabuľky pracuje s vrátenou hodnotou hashovacej funkcie. Hodnota nám určuje index, kam sa bude prvok vkladať.

Pri tomto druhu vkladania môžu nastať dva prípady. Prvý nastane, keď index tabuľky je prázdny. Vtedy sa do tabuľky prvok zapíše bez problému. Rýchlosť je konštantná O(1). Druhý prípad nastáva, keď index tabuľky prázdny nie je. Vzniká tu kolízia hashov.

Následná implementácia znázorňuje prvé riešenie kolízie (Obrázok 2.). Princíp spočíva v posúvaní sa po indexoch na ďalšie miesto až kým neobjavíme prázdnu pozíciu. V tomto momente program stráca svoju rýchlosť a môže dôjsť až na O(n) – extrémny prípad keď je tabuľka plná.



Obrázok 2. – Kolízia pri vkladaní prvkov

2.3. Hľadanie prvku

Hľadanie prvkov funguje rovnako ako pridávanie prvkov do tabuľky. Najskôr sa vygeneruje hash, ktorý určí pozíciu v tabuľke. Potom môžu nastať tieto prípady:

-Ak je index tabuľky prázdny, znamená to, že prvok sa v tabuľke nenachádza...

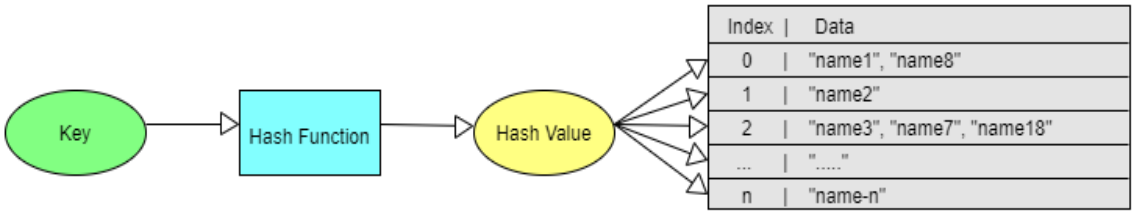
-Ak sa prvok na indexe rovná hľadanému prvku, tak tabuľka prvok obsahuje...

-Ak sa prvok na indexe nerovná hľadanému prvku, tak overujeme nasledujúci prvok tabuľky. Takto overujeme až kým prvok objavíme alebo kým sa nevrátime na prvý index. Vtedy sa prvok v tabuľke nenachádza.

Rýchlosť je tak konštantná O(1).

3. Druhá vlastná implementácia

Spočíva reťazovom vyhľadávaní voľného miesta v pamäti. Ak sa takéto miesto dlho vyhľadáva, tabuľka sa automaticky zväčšuje.



Obrázok 3. – Druhá implementácia hashovacej tabuľky so spájaným zoznamom

3.1. Hashovacia funkcia

Funguje rovnako ako pri prvej implementácii.

Je to funkcia, ktorá kóduje aktuálne hodnoty a ako produkt vráti celé číslo reprezentujúce pozíciu v tabuľke. Konkrétny index dosiahneme matematickými operáciami násobenia konkrétneho písmena slova zväčšeného o aktuálnu pozíciu v slove ďalším písmenom zväčšeným o aktuálny index – celková dĺžka slova. Ďalej ho násobím prvočíslom, ktoré mi zabezpečí rovnomerné rozloženie a taktiež aktuálnou pozíciou v znaku slove. Následne použitím operácie modulo o veľkosti tabuľky zabezpečíme index z intervalu <0, n-1>.

3.2. Vkladanie do hashovacej tabuľky

Na začiatku sa vygeneruje hash index kam sa bude prvok ukladať. Následne sa overia nasledujúce prípady:

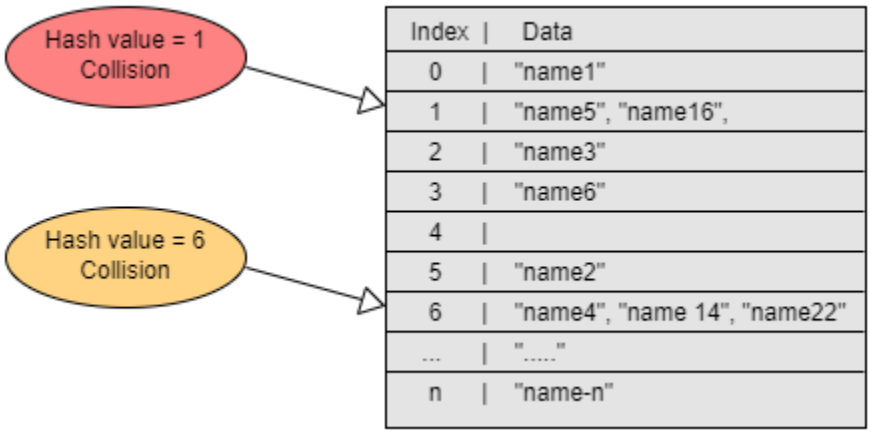
-Ak je index prázdny, prvok sa pridá do tabuľky

-Ak index nie je prázdny, nastáva kolízia a je potrebné pripojiť prvok na koniec spájaného zoznamu (Obrázok 4.)

-Ak je príliš veľa prvkov zreťazených, tabuľka sa automaticky zväčšuje a nanovo hashuje existujúce prvky

Prvý prípad je najrýchlejší. Rýchlosť je konštantná O(1).

V druhom a treťom prípade treba prvok pripojiť na koniec spájaného zoznamu rovnakých hashov. Je potrebné spustiť funkciu, ktorá sa posúva po všetkých prvkoch až kým nenarazí na prázdny prvok spájaného zoznamu. Vtedy sa jednoducho napojí na posledný prvok postupnosti. V tomto prípade program spomaľuje, pretože musí prebehnúť každý prvok v zozname na aktuálnom indexe. Najpomalší prípad nastane len vtedy, keď budú všetky prvky uložené na rovnakom indexe. Najpomalšia rýchlosť by tak bola O(n).



Obrázok 4. – Kolízia a reťazenie druhej implementácie

3.3. Hľadanie prvku

Funguje rovnako ako jeho pripájanie.

Na začiatku sa vygeneruje hash index, z ktorého sa zistí či je prvok v tabuľke. Následne sa overia nasledujúce prípady:

-Ak je index prázdny, prvok nie je v tabuľke

-Ak index nie je prázdny, overujeme, či sa hľadaný prvok rovná aktuálnemu prvku. Takto postupujeme, až kým neobjavíme hľadaný prvok alebo neprebehneme celý spájaný zoznam na aktuálnom indexe tabuľky. To znamená že sa prvok v tabuľke nenachádza.

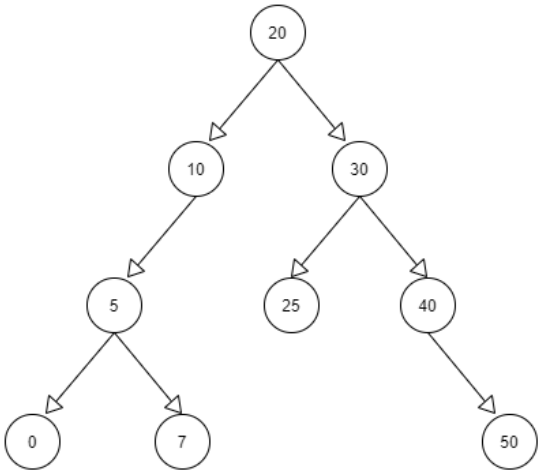
Priemerná vyhľadávacia rýchlosť je konštantná O(1). Najhorší možný scénar môže byť O(n) kedy by sa tabuľka nezväčšovala a prvky neprehashovali. Vtedy by boli všetký zreťazené v jednom riadku.

Binárny vyhľadávací strom

4. Úvod

Binárny vyhľadávací strom (Obrázok 5.) je dátová štruktúra, ktorá umožňuje rýchlejšie prehľadávanie údajov. Funguje na základe delenia ľavej a pravej strany, kde ľavá strana predstavuje hodnoty menšie ako tie čo sú nad ňou. Pravá strana logicky väčšie hodnoty.

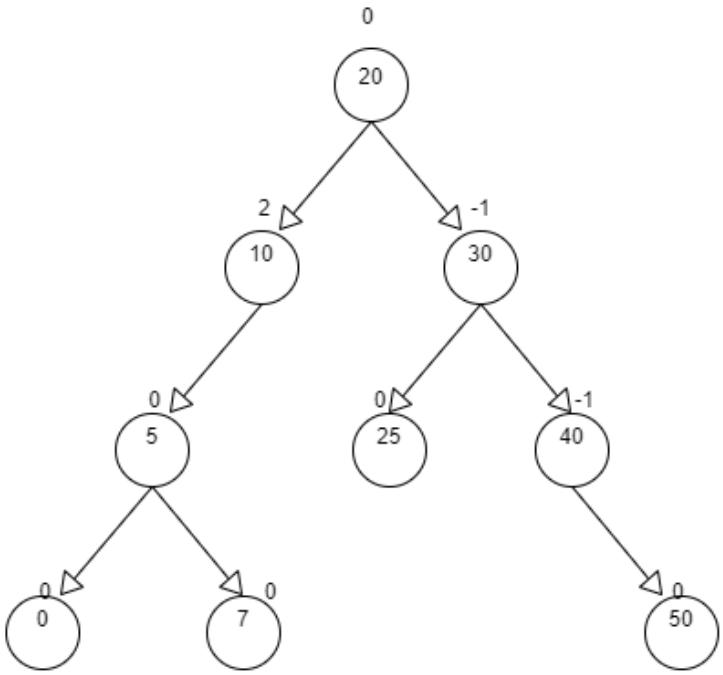
Moje implementácie predstavujú AVL strom, ktorý obsahuje informáciu o hĺbke a dvoch dcérskych údajov a Červeno-čierny strom, ktorý funguje podobne ako AVL strom, ale miesto informácie o hĺbke je obohatený o informáciu o farbe a o ukazovateľ na rodiča.



Obrázok 5. – Binárny strom

5. Prvá vlastná implementácia - AVL Strom

Prvá implementácia predstavuje AVL strom, ktorý je vyrovnávaný na približne rovnakú hĺbku posledných členov. Ak rozdiel posledných členov patrí intervalu <-1, 1> tak strom je vyrovnaný. Ak interval nespĺňa (Obrázok 6.), je potrebné vykonať rotácie, ktoré zabezpečia rovnosť. Takto vyrovnaný strom zabezpečuje optimálnu vyhľadávaciu rýchlosť O(log n), pretože sa pristupuje vždy k menšine prvkov, ktoré sú rovnomerne balansované.



Obrázok 6. – Nevyrovnaný AVL strom

V mojej implementácii využívam dynamické štruktúry, obsahujúce celočíselnú hodnotu, dva ukazovatele na typ štruktúry a informáciu o hĺbke. Takto definovaná štruktúra vytvára dynamický AVL strom.

5.1. Vkladanie

Vkladanie do AVL stromu funguje rýchlo. Funkcia vkladanie najskôr objaví bod, kam sa má nová štruktúra pridať rekurzívnym spôsobom. Môžu nastať tri fázy objavovania:

- Prvky sa rovnajú a preto ho nepridáme
- Prvok je menší ako aktuálny prvok, tak prehľadávame ľavú stranu
- Prvok je väčší ako aktuálny prvok, preto prehľadávame pravú stranu
- Prvok je NULL, tak ho pridáme na túto pozíciu

Ak prvok pridávame, tak vytvárame nový prvok v pamäti, s informáciami, ktoré do stromu vkladáme a rodičom, ktorý je o jednu hĺbku nad prvkom naň ukážeme.

Treba však skontrolovať hĺbky stromu a následne aj balancovanie stromu. Postupujeme rekurzívne a akonáhle objavíme prvok, ktorého rozdiel ľavej a pravej strany nepatrí intervalu <-1, 1> , tak spustíme vyvažovanie 6.2.

Takto je jasne definovaný AVL strom.

5.2. Vyvažovanie

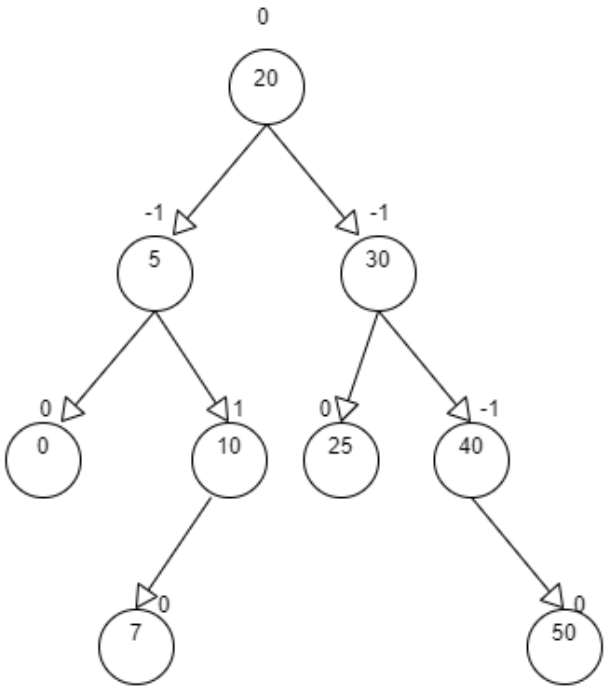
Ak sa v strome nachádza nevyrovnaná časť, ktorá spôsobuje spomalenie vyhľadávania v strome, Potrebujeme tuto chybu hneď odstrániť a docielime to rotáciou stromu.

Z obrázku 6. je vidieť, že strom na prvku 10 je nevyrovnaný. Ľavá strana má dĺžku o 2 dlhšiu ako pravá strana. Preto je potrebné uplatniť pravú rotáciu:

- Dcérsky prvok nevyrovnanej bunky presunieme nad nevyrovnanú bunku a jej pravý ukazovateľ nastavíme na nevyrovnaný prvok
- Pravý prvok dcérskej bunky sa musí odkazovať na pôvodnú nevyrovnanú bunku

Rotovaný strom z Obrázku 6. následne vyzerá ako Obrázok 7.

Následne overujeme rovnováhu ostatných prvkov a ak objavíme následný nevyrovnaný prvok, tak vyvažovanie opakujeme s aktuálnou bunkou.



Obrázok 7. – Vyrovnaný AVL strom

5.3. Hľadanie prvku

Hľadanie prvku v strome prebieha až kým prvok neobjavíme alebo nenarazíme na NULL prvok. Pri aktuálnom prvku uplatňujeme nasledujúce pravidlá:

- Prvok a hľadaná hodnota sa rovnajú a preto sa prvok nachádza v strome
- Hľadaná hodnota je menšia ako aktuálny prvok, tak prehľadávame ľavú stranu
- Hľadaná hodnota je väčšia ako aktuálny prvok, preto prehľadávame pravú stranu
- Prvok je NULL a preto sa prvok v strome nenachádza

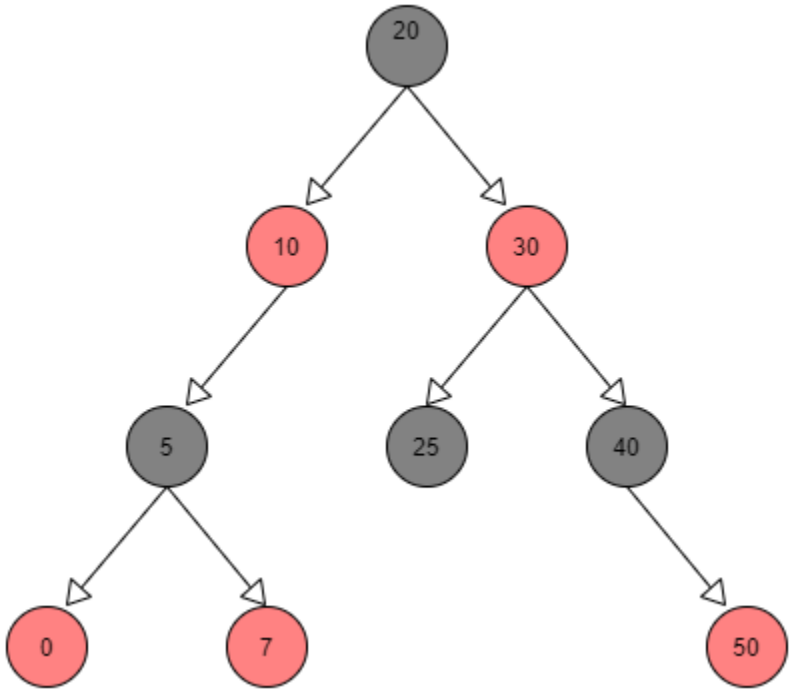
6. Druhá implementácia - RB Strom (nedokončený)

V mojej implementácii strom obsahuje celočíselnú hodnotu, informáciu o farbe a 3 ukazovatele na typ štruktúry. Sú to rodič, ľavá a pravá dcérska strana. Rozdiel s AVL stromom je taký, že AVL používa na vyrovnávanie hĺbku prvkov, zatiaľ čo Červeno-čierny strom využíva farbu. Nie je to dokonalý vyrovnávací mechanizmus, ale dokáže zabezpečiť vyhľadávaciu rýchlosť O(log n).

Pri strome uplatňujeme pravidlá:

- Prvky sú buď čierne alebo červené
- Začiatok stromu je vždy čierny
- Nemôžu na seba ukazovať dva červené prvky
- Každá cesta od ľubovoľného bodu po posledný NULL má rovnaký počet čiernych prvkov
- Všetky listy sú čierne (list je NULL)

Takto určené pravidlá jasne definujú RB Strom.



Obrázok 8. – Červeno-čierny strom

6.1. Vkladanie

Je veľmi podobné AVL stromu. Časová náročnosť vkladania je logaritmická O(log n). Je to vďaka jeho vyvažovaniu, kedy prvky rovnomerne rozdeľujeme na ľavú a pravú stranu pomocou pravidiel o čiernej a červenej farbe.

Nároky na pamäť sú lineárne O(n), pretože miesto sa rezervuje iba pre jeden prvok a to ten, ktorý práve vkladáme.

6.2. Vyvažovanie

Vyvažujeme ho na základe vyššie zmienených pravidiel:

- Prvky sú buď čierne alebo červené
- Začiatok stromu je vždy čierny
- Nemôžu na seba ukazovať dva červené prvky
- Každá cesta od ľubovoľného bodu po posledný NULL má rovnaký počet čiernych prvkov
- Všetky listy sú čierne (list je NULL)

6.3. Hľadanie prvku

Hľadanie prvku funguje rovnako ako pri AVL strome.

Hľadanie prvku v strome prebieha až kým prvok neobjavíme alebo nenarazíme na NULL prvok. Pri aktuálnom prvku uplatňujeme nasledujúce pravidlá:

- Prvok a hľadaná hodnota sa rovnajú a preto sa prvok nachádza v strome
- Hľadaná hodnota je menšia ako aktuálny prvok, tak prehľadávame ľavú stranu
- Hľadaná hodnota je väčšia ako aktuálny prvok, preto prehľadávame pravú stranu
- Prvok je NULL a preto sa prvok v strome nenachádza

Testovanie

Testovanie prebiehalo na rovnakých, náhodne vygenerovaných prvkoch, ktoré boli parametrom podávané do jednotlivých dátových štruktúr. Funkčná implementácia sa spúšťa v súbore main.c, odkiaľ sa vyvolávaju jednotlivé dátove štruktúry.

Z mojich implementácií som sa dozvedel nasledujúce informácie obsiahnuté v kapitolách 7. 8. a 9...

Porovnanie s implementáciami z internetu vysvetlím v kapitole 9.1.

7. Hashovanie

Hashovanie je pomerne jednoduchý nástroj na ukladanie, hľadanie či mazanie údajov. V priemerných prípadoch je rýchlosť hashovania konštantná O(1) a v kritických prípadoch lineárna O(n). V týchto prípadoch nastáva prehľadávanie každého jedného prvku.

Nevýhoda je v zaberaní miesta, kedy môže nastať situácia, že tabuľka sa zväčší a my už nepridáme ani jeden prvok do nej. Obsah by tak môže dosahovať O(2n).

8. Vyhľadávacie stromy

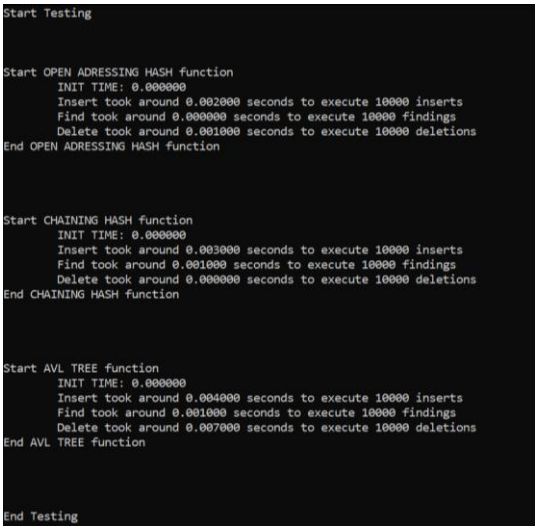
Vyhľadávací strom AVL je efektívna štruktúra, ktorá umožňuje logaritmické pracovanie s údajmi O(log n). V najhoršom scenári je rýchlosť taktiež O(log n), pretože AVL obsahuje balansovanie stromu. Vtedy každý podstrom obsahuje približne rovnaký počet prvkov.

Veľká výhoda týchto stromov je, že nezaberajú až tak veľa pamäte počítača a preto sú vhodné, keď si nemôžeme dovoliť plýtvať priestorom. Obsah tak dosahuje O(n).

9. Štatistiky

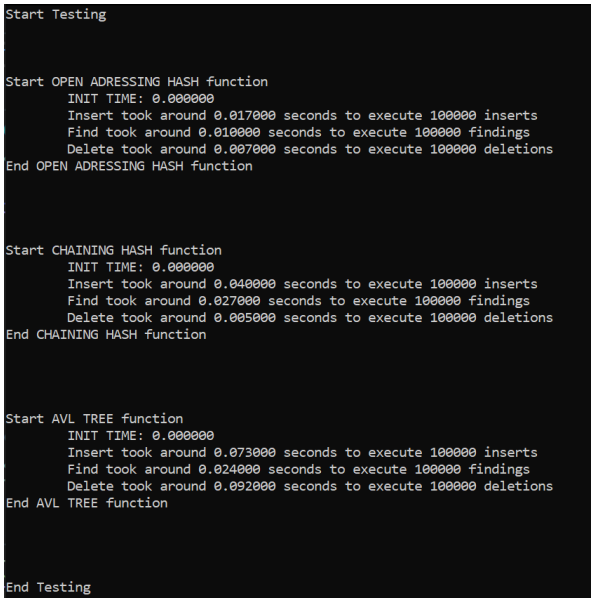
Nasledujúce štatistiky obsahujú informácie potrebné na porovnanie štruktúr.

Obrázok 9. – vstup 10 000 prvkov. Rozdiely sú zanedbateľné. Avšak víťaz vkladania je hash. Tab. S otvoreným adresovaním. Hľadanie vyhráva taktiež hash. tab. s otvoreným adresovaním. A napokon mazanie bezkonkurenčne hash. Tab. S otvoreným adresovaním. V tomto prípade jasne volíme hash. Tab. S otvoreným adresovaním pre jej skvelú rýchlosť.



Obrázok 9. – Testovanie 10 000 prvkov

Obrázok 10. – vstup 100 000 prvkov. Podobne ako pri vstupe 10 000 prvkov tak dominuje hash. tab. S otvoreným adresovaním pri vkladaní a vyhľadávaní. Avšak pri mazaní je jasný víťaz hash. tab. s reťazením. V túto chvíľu si volíme či budeme častejšie mazať alebo vkladať či vyhľadávať.



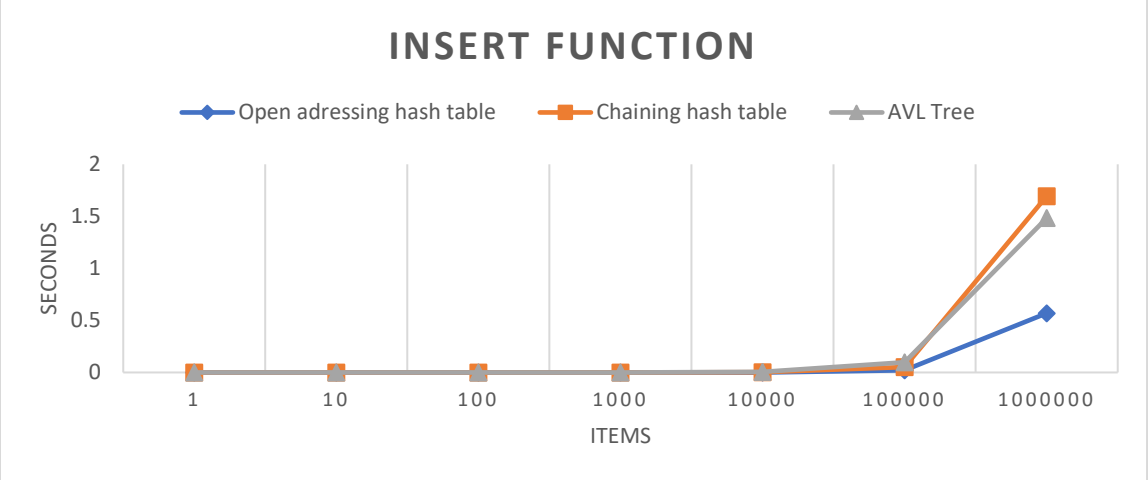
Obrázok 10. – Testovanie 100 000 prvkov

Obrázok 11. – vstup 1 000 000 prvkov. Najrýchlejšie vkladanie prvkov bolo v hashovacej tabuľke s otvoreným adresovaním. Najrýchlejšie čítanie dát bolo v hashovacej tabuľke s reťazením. Napokon najrýchlejší delete bol v hashovacej tabuľke s reťazením. Pri tomto množstve dát je potrebné dbať, či sa budú dáta často vkladať (volíme hash. tab. s otvoreným adresovaním), alebo čítať (hash. tab. s reťazením alebo AVL strom). Napokon ak sa potrebuje často mazať (volíme hash. tab. s reťazením).

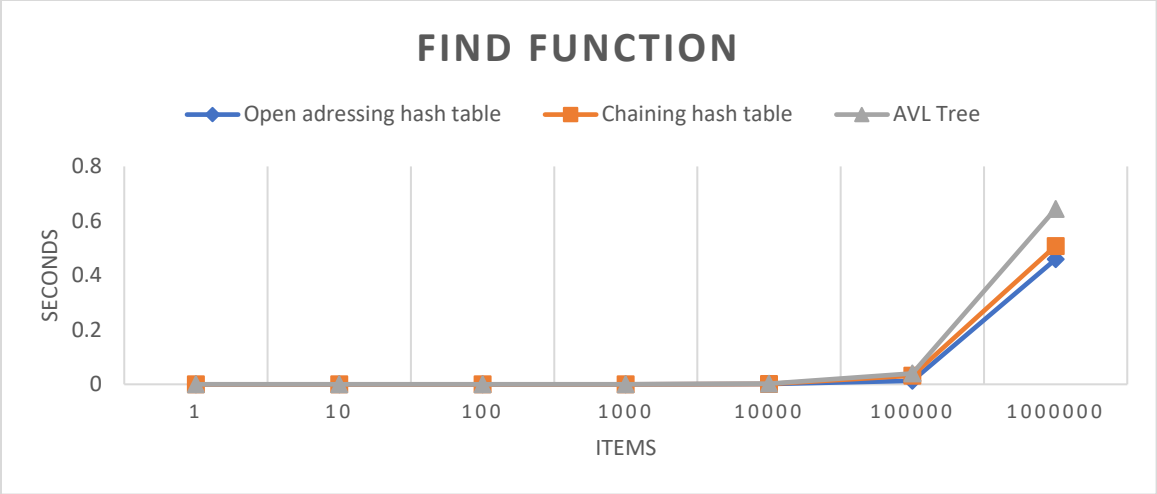
```
Start Testing


Start OPEN ADRESSING HASH function
        INIT TIME: 0.000000
        Insert took around 0.608000 seconds to execute 1000000 inserts
        Find took around 0.578000 seconds to execute 1000000 findings
        Delete took around 0.180000 seconds to execute 1000000 deletions
End OPEN ADRESSING HASH function



Start CHAINING HASH function
        INIT TIME: 0.000000
        Insert took around 1.363000 seconds to execute 1000000 inserts
        Find took around 0.506000 seconds to execute 1000000 findings
        Delete took around 0.062000 seconds to execute 1000000 deletions
End CHAINING HASH function



Start AVL TREE function
        INIT TIME: 0.000000
        Insert took around 1.363000 seconds to execute 1000000 inserts
        Find took around 0.579000 seconds to execute 1000000 findings
        Delete took around 1.831000 seconds to execute 1000000 deletions
End AVL TREE function



End Testing
```

Obrázok 11. – Testovanie 1 000 000 prvkov

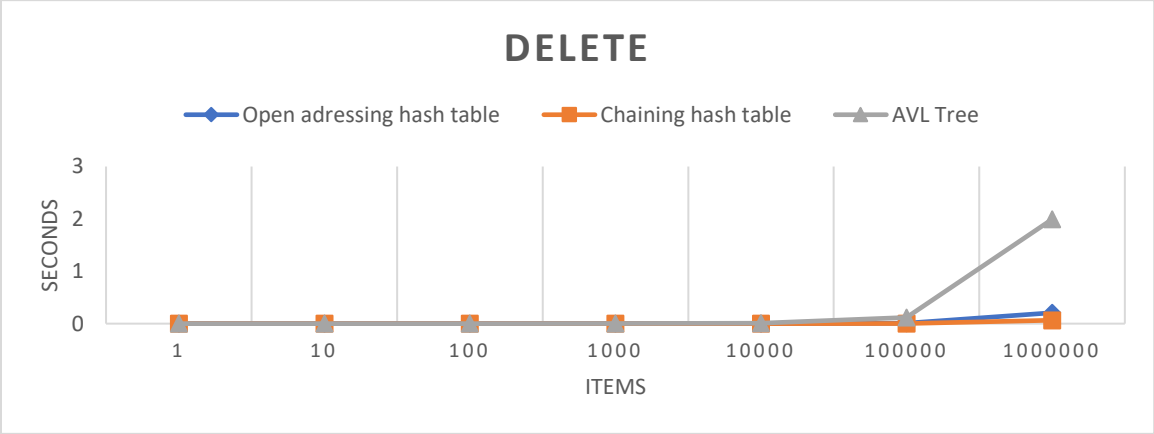
Napokon grafické zobrazenie celého zadania z ktorého zreteľne vidieť výsledky (tabuľky 1-3).



Tabuľka 1. – Funkcia vkladanie



Tabuľka 2. – Funkcia hľadania



Tabuľka 3. – Funkcia mazanie

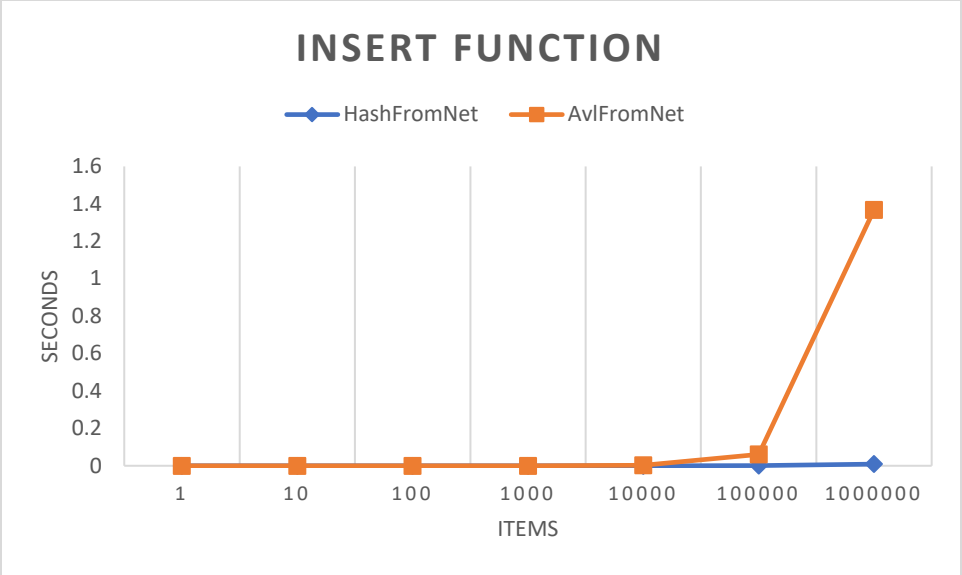
9.1. Porovnanie s implementáciami z internetu

Pri porovnávaní tabuliek 1, 3. a tabuliek 4, 5. vidíme rozdiely medzi mojími implementáciami a implementáciami z internetu.

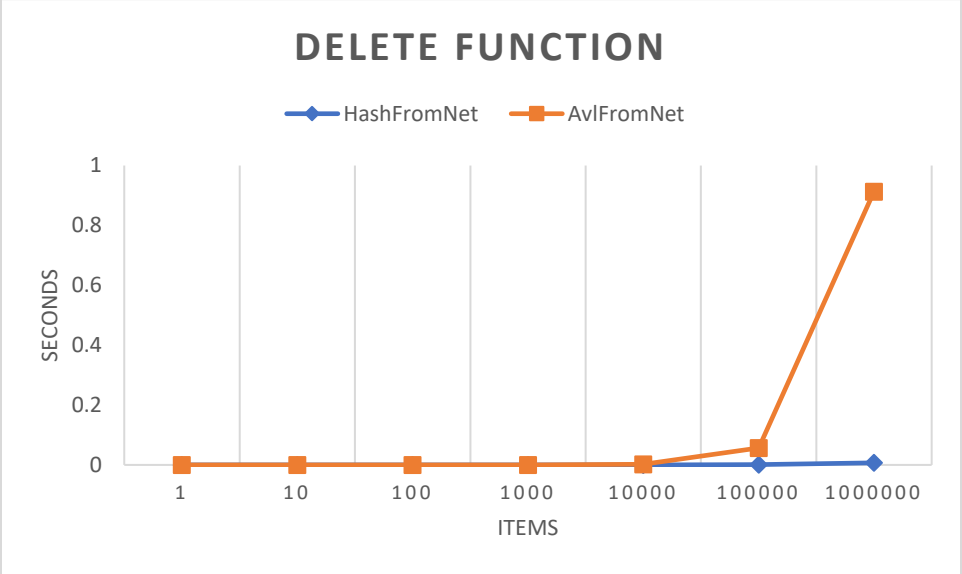
Funkcia vkladanie je približne rovnaká v mojej implementácii AVL ako aj v prevzatej implementácii. Pri milióne prvkov je čas +- 1.4s. Väčší rozdiel vidieť pri implementáciach hashovacích tabuliek s otvoreným adresovaním. Pri milióne prvkov je prevzatá implementácia o 0.56s rýchlejšia ako moja. Napriek tomu sú stále najrýchlejšie.

Funkcia mazanie je v prevzatej implementácii taktiež rýchlejšia. Rozdiel medzi mojou a prevzatou hashovacou tabuľkou je 0.201s pri milióne prvkov. Prevzatý AVL strom maže o takmer celú sekundu rýchlejšie ako môj AVL strom.

Napriek tomu ostáva výsledok rovnaký a najvýkonnejšie sú hashovacie tabuľky. Ponúkajú rýchle spracovanie údajov ako aj jednoduchú implementáciu kódu.



Tabuľka 4. – Vkladacia funkcia implementácií z internetu



Tabuľka 5. – Mazacia funkcia implementácií z internetu