Matej Delinčák, ID: 102895

Zadanie 2 - Vyhľadávanie v dynamických množinách

Problematika

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.

V tomto zadaní som porovnal obyčajný vyhľadávací strom, AVL vyvažovací strom, červeno-čierny vyvažovací strom, dvojité hašovanie(otvorená adresácia) a hašovanie, ktoré rieši kolízie pomocou zreťazenia prvkov do zoznamu.

Binárny vyhľadávací strom

Ide o štruktúru, ktorá drží koreň stromu. V tomto strome sa nachádzajú prvky, z ktorých každý má nejakú hodnotu a dvoch nasledovníkov. Ľavého a pravého. Hodnota ľavého potomka je menšia ako hodnota rodiča a hodnota pravého potomka je zase väčšia. Týmto zabezpečíme zoradenú štruktúru, v ktorej sa dá rýchlo vyhľadávať a to so zložitosťou O(log n). Avšak problém nastáva, keď strom začne byť nevyvážený. Vtedy sa zo stromu stáva lineárny zoznam zo zložitosťou vyhľadania O(n).

Moja implementácia

V implementácií pre obyčajný binárny vyhľadávací strom som vo funkcii *insert* nepoužil rekurziu, ale iteratívny cyklus. Je to preto, aby som ukázal, že viem aj spraviť aj taký. A zároveň preto, lebo keď je strom nevyvážený, ľahko mu môže narásť výška. A už pri volaní rekurzie so stromom o hĺbke okolo 70000, padol program. Takýto problém je aj s funkciou uvoľnenia stromu, tzv. *deleteTree*. V tomto momente mám pridanú podmienku, že ak je strom príliš vysoký, tak sa nič neuvoľní. Uvoľňovanie mam totiž zrobenú cez rekurziu. Rekurzia by sa dala obísť tým, že by som si vytvoril vlastný zásobník, kde by som držal potrebné ukazovatele a tie následne uvoľňoval – vlastný zásobník. Ale toto nebolo súčasť zadania. Tak som sa takýmto niečím nezapodieval.

Zoznam funkcií:

- *createBVS()* – vytvorí vrchol
- *insertBVS(koren, hodnota)* – vloží do stromu
- *printBVS(koren, medzera*) – vykreslí strom
- *searchBVS*(koren, hodnota) – vyhľadá v strome
- *deleteBVS(koren*) – vymaže strom

AVL strom

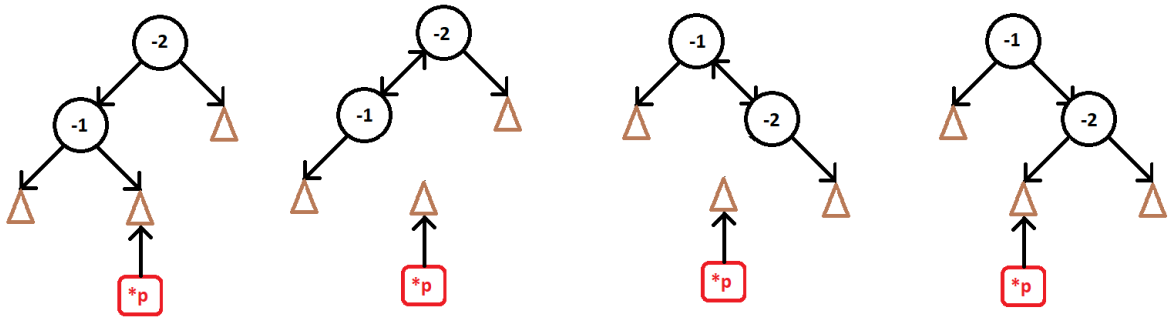
Problém nevyváženosti rieši napríklad AVL vyvažovací strom. Túto metódu som aj sám implementoval. Tento strom taktiež pracuje pri funkciách *insert* a *search* so zložitosťou O(log n).

Oproti obyčajnému stromu si vo vrchole drží aj maximálnu hĺbku podstromu. Táto hĺbka sa využíva pri rotovaní stromu. Ak je vrchol, na ktorom sa nachádzam, nevyvážený (to znamená, že jeden podstrom je o dva hlbší ako druhý), začínam rotovať. Celkovo rozoznávam štyri prípady. Ak je nevyvážený doprava a zároveň aj jeho pravý potomok je nevyvážený doprava, tak rotujem doľava. Ale ak je potom nevyvážený doľava, tak najskôr ten

musím zrotovať doprava a až potom jeho rodiča doľava. Toto isté platí symetricky aj pre druhú stranu. AVL strom je vždy, čo najlepšie vyvážený, čo spôsobuje, že funkcia *insert* je pomalšia ako pri iných implementáciách. Avšak funkcia *search* by mala byť rýchlejšia, pretože strom je vždy čo najlepšie vyvážený na rozdiel od červeno-čierneho stromu.

Moja implementácia

Moju funkciu *insert* som riešil rekurzívne. Porovnávam, či je hodnota aktuálneho vrcholu väčšia alebo menšia a na základe toho sa rozhodujem, do ktorého podstromu sa vydám. Po vložení prvku sa začínam vracať späť do koreňa. Počas tohto vracania, aktualizujem hĺbku podstromu. Keď je aktualizovaná, pozriem sa, či je vrchol vyvážený. Ak nie, nastupuje funkcia balance. Táto v sebe zahŕňa štyri kľúčové podmienky, ktoré som už spomínal vyššie. Takto prejdem celý strom. Makro DIFF_RANK berie ako parameter smerník na vrchol. A vráti rozdiel hĺbiek jeho podstromov. Toto číslo mi hovorí, či vrchol je vyvážený, alebo nie. Na obrázku nižšie ukážem ako mi funguje obyčajná rotácia vpravo.



Najskôr presmerujem smerníky a ako posledný krok prerátam ich výšky.

Funkcia *search* je založená na podobnom princípe. Ale nie rekurzívne. Používam cyklus while, ktorý je rýchlejší, ako rekurzívna funkcia *search*. Vydávam sa do príslušného podstromu na základe hľadanej hodnoty. Ak sa hodnota nájde, vráti ukazovateľ na ňu. Ak nie, vráti NULL.

Zoznam funkcií:

- *createAVL*() – vytvorí vrchol
- *insertAVL*(*koren*, *hodnota*) – vloží do stromu
- *printAVL*(*koren*, *medzera*) – vykreslí strom
- *searchAVL*(*koren*, *hodnota*) – vyhľadá v strome
- *deleteAVL*(*koren*) – vymaže strom
- *balanceAVL*(*koren*) – vybalancuje strom
- *rotateLeftRight*(*koren*) – rotácia najskôr vpravo a potom vľavo
- *rotateRight*(*koren*) – rotácia vpravo
- *rotateRightLeft*(*koren*) – rotácia najskôr vľavo a potom vpravo
- *rotateLeft*(*koren*) – rotácia vľavo

Červeno-čierny strom

Ďalší algoritmus na vyvažovanie stromu je tzv. červeno-čierny strom. Tento som prevzal od [1]. Do tejto implementácie som dorobil uvoľnenie stromu. Oproti obyčajnému stromu, si vo vrchole drží aj farbu vrcholu. Vrcholy majú buď čiernu alebo červenú farbu. Koreň je vždy čierny. Listy tohto stromu sú prázdne vrcholy s čiernou farbou. Zároveň podmienka pre to, aby takýto strom bol vyvážený, je, aby žiadna dva za sebou nasledujúce vrcholy neboli červené. A oba podstromy vrchola, musia mať rovnaký počet čiernych vrcholov. Ak tieto podmienky nie sú splnené, prichádzajú rotácie, alebo prefarbenie.

Červeno-čierny strom taktiež pracuje pri funkciách *insert* a *search* so zložitosťou O(log n). Avšak oproti AVL stromu je vo funkcií *insert* rýchlejší. Je to preto, lebo tento typ algoritmu, nevyvažuje strom úplne. Avšak keďže nie je úplne vyvážený, funkcia *search* je pomalšia oproti AVL.

Prevzatá implementácia

Insert funguje iteratívnym spôsobom ako môj BVS strom bez vyvažovania. Najskôr sa vloží nový vrchol, ktorého potomkovia sú oba NILL (NILL je v tejto implementácií NULL vrchol) a jeho farba je červená. Následne sa zavolá funkcia na opravu stromu. Buď sa vrcholy prefarbia tak, aby to vyhovovalo kritériám alebo sa pristupuje k rotovaniu.

- Jeden príklad kedy sa bude prefarbovať je, keď je vrchol červený a jeho pravý ujo alebo pravý potomok deda je tiež červený. Vtedy sa rodičovi a ujovi nastaví farba čierna a dedovi červená. V tomto momente sa dedo stáva nový vrchol, ktorý porušuje pravidlá.
- Ak ale pravý ujo nie je červený, zafarbí otca na čierno, deda na červeno a prichádza pravá rotácia deda.

Toto isté platí symetricky aj pre opačnú stranu stromu. Rotácie fungujú rovnako ako v mojej implementácií.

Hash tabuľka

Ide o pole údajov, ktoré sú nejakým spôsobom zakódované do kľúčov. Na základe týchto kľúčov im je priradené miesto v danej tabuľke. Problém však nastáva, ak hašovacia funkcia vytvorí pre dva rozlične údaje rovnaký kľuč. Vtedy nastáva kolízia a tento problém treba vyriešiť. Je veľa spôsobov ako tento problém vyriešiť, ale ja som implementoval otvorenú adresáciu s dvojitým hašovaním a porovnával som ju s prevzatou implementáciou, ktorá kolízie rieši reťazením.

Moja implementácia otvorenej adresácie

Vo svojej hašovacej tabuľke si držím počet prvkov v poli a najbližšie prvočíslo menšie ako veľkosť poľa. Veľkosť poľa taktiež udržiavam na prvočísle. Je to preto, lebo prvočísla majú menej súdeliteľných čísel a teda nastáva menej kolízií. Na obrázku nižšie ukazujem moje dve hašovacie funkcie. Keď vo funkcii *insert* vkladám hodnotu, najskôr sa pozriem na index, ktorý mi vráti prvá hašovacia funkcia. Keď už tam prvok existuje, na radu prichádza druhá hašovacia funkcia, ktorá tento index posúva o danú hodnotu. Tento proces má v najlepšom prípade zložitosť O(1) ak sa nepoužije druhá funkcia. Ale v najhoršom prípade môže dosiahnuť až O(n), v prípade, že bude musieť prejsť celý zoznam. Preto je efektívnejšie udržiavať tabuľku „poloprázdnu“. Už pri polovičnom naplnení tabuľky, volám funkciu *expandTable*. Táto mi vytvorí nové pole s približne dvojnásobnou veľkosťou a všetky hodnoty zo starej, nanovo vložím do novej tabuľky (Toto samozrejme zaberie približne O(n)). Funkcia

search bude mať rovnakú zložitosť ako *insert*, len rozdiel je v tom, že pri tejto funkcií, sa tabuľka nezväčšuje. Ak by som dovolil plniť tabuľku po polovici, nárast kolízií je enormný a spomaľuje sa algoritmus.

Výhoda mojej implementácie sa týka pamäti. V každom momente viete, akú veľkú tabuľku máte. Všetko sa uchováva v jednom poli na rozdiel od reťazenia.

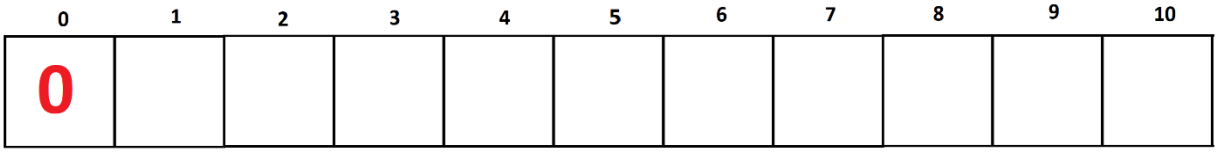
```
int hashFunctionFirst(HASHTABLE* paTable, int paKey) {
    return paKey % paTable->maxSize;
}

int hashFunctionSecond(HASHTABLE* paTable, int paKey) {
    return paTable->primeSmall - (paKey % paTable->primeSmall);
}
```

Moje dve hašovacie funkcie

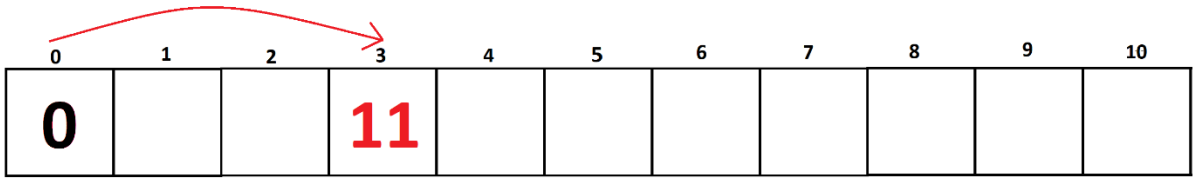
Zoznam funkcií:

- *createHash(tabulka)* – vytvorí tabuľku
- *hashFunctionFirst(tabulka, kluc)* – hašovacia funkcia, vracia kľúč modulo veľkosť poľa
- *hashFunctionSecond(tabulka, kluc)* – vracia prvočíslo – (kľúč modulo prvočíslo)
- *isPrime(hodnota)* – zisti, či je číslo prvočíslo
- *newSizeAndPrime(tabulka)* – nastaví nové vlastnosti tabuľky ako veľkosť a prvočíslo pre druhu hašovaciu funkciu
- *expandTable(tabulka)* – zväčší tabuľku
- *insertDoubleHash(tabulka, hodnota)* – vloží do tabuľky
- *searchDoubleHash(tabulka, hodnota)* – vyhľadá v tabuľke
- *deleteDoubleHash(tabulka)* – vymaže tabuľku
- *printDoubleHash(tabulka)* – vypíše tabuľku

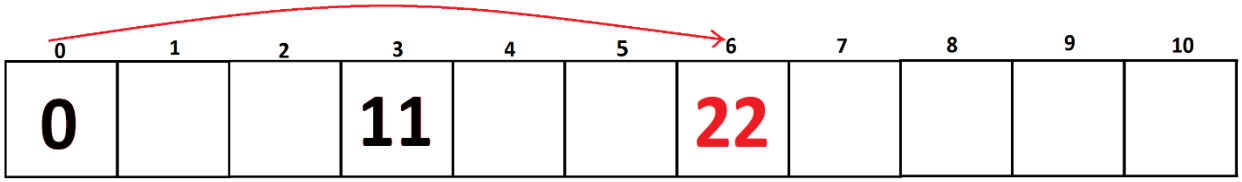
Ďalej uvediem príklad môjho algoritmu otvorenej adresácie. Majme pole o veľkosti 11 a vložím hodnotu 0. Keďže je tabuľka prázdna, vloží sa na miesto, ktoré určila prvá funkcia (0 mod 11 = 0).



Ako ďalšie číslo vložím 11, a tu prichádza na rad druhá funkcia. Na hodnote (11 mod 11 = 0) je už číslo, tak musím vyrátať index, o ktorý sa budem posúvať. Zoberiem prvočíslo menšie ako veľkosť poľa (7 – (11 mod 7)). A teda vložím číslo na index 0 + 3.



Následne idem vložiť číslo 22. Použijem rovnaký postup a aj napriek tomu, že pre čísla 11 aj 22 nám dá prvá funkcia nulu, druhá funkcia nám už nedá rovnaký index a vyhli sme sa druhej kolízií.

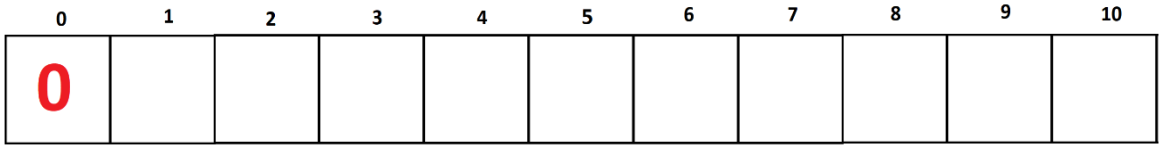


Prevzatá implementácia hašovacej tabuľky so zreťazením

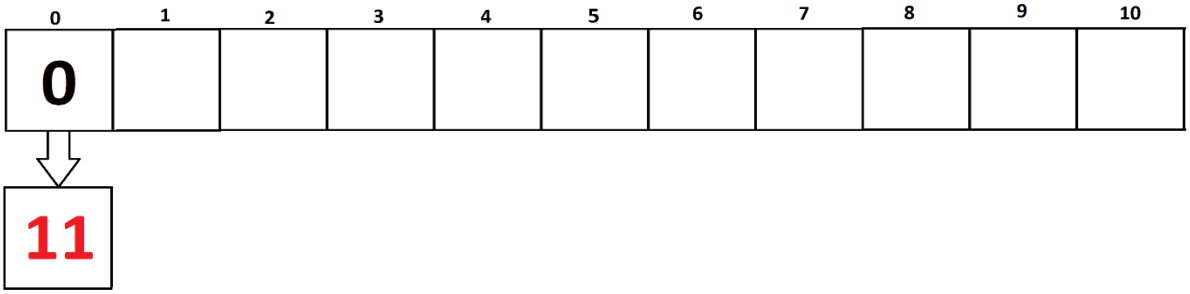
Na porovnanie som si vybral tabuľku, ktorá rieši kolízie zreťazením. Táto implementácia má jednu hašovaciu funkciu (takú istú ako ja). Táto určí index do poľa. Ak sa tam nič nenachádza, vloží prvok. Ale ak hej, tak túto hodnotu nebude posúvať niekde inde do poľa ako pri otvorenej adresácií, ale zaradí ju na koniec zoznamu, ktorý sa nachádza na danom indexe. V podstate táto implementácia pracuje s poľom smerníkov na zreťazené zoznamy. Zložitosť algoritmu na vkladanie je O(1), lebo vkladám priamo do zoznamu na posledné miesto. Na rozdiel od mojej implementácie sa tabuľka zväčšuje pri faktore naplnenia 0,75. Táto metóda si to „môže“ dovoliť, lebo hašovacia funkcia vždy vyberie správny zoznam a hodnota sa len zaradí na koniec.

Problém môže však nastať pri metóde *search*, kedy v najhošom prípade musí program prehľadať celý konkretný zoznam. Čiže ide o zložitosť O(n). Čo sa týka pamäťovej zložitosti, tak toto je nevýhoda tohto prístupu. Nikdy neviete, aké veľké zoznamy sa vytvorili a ako veľa zaberajú. Oproti otvorenej adresácií, kedy vždy viete, akú veľkú pamäť potrebujete. Tento algoritmus som prevzal od [2]. Zároveň som kód jemne upravil, lebo autor nevyriešil uvoľňovanie pamäte.

Nižšie si teraz ukážeme príklad takéhoto prístupu. Majme rovnakú veľkosť poľa ako pred chvíľou a vložíme prvok 0. Hašovacia funkcia nám vráti 0 mod 11 = 0.



Avšak teraz keď vložíme číslo 11, nastane kolízia, a toto číslo len pridáme na koniec zoznamu.



Testovanie

Pre overenie správností rotácií a správnosti fungovania hašovacích tabuliek som vytvoril funkcie na vykreslenie stromov, vypísanie hĺbky a výpis tabuliek. Takýmto štýlom som skontroloval správnosť použitých implementácií, či už mojich alebo prevzatých.

V testovacích scenároch na efektivitu som sa zameral na tri hlavné kategórie vstupov:

- Lineárna postupnosť čísiel od 0 po N
- Striedavá postupnosť čísiel: 0, n, 1, n-1, ...
- Náhodná postupnosť čísiel so seedom 1

Najzaujímavejší vstup je ten náhodný, lebo sa približuje skutočnému využitiu v reálnom živote.

Tieto vstupy som testoval celkovo na piatich implementáciách v troch rôznych prípadoch (*insert*, *search*, *insert*/*search*):

- Binárny vyhľadávací strom bez vyvažovania
- Samovyvažovací vyhľadávací strom AVL
- Samovyvažovací vyhľadávací červeno-čierny strom
- Hašovacia tabuľka s otvorenou adresáciou
- Hašovacia tabuľka s reťazením

Výsledky

Na ďalších stranách rozoberiem podrobne jednotlivé testovania. Celkovo pri porovnávaní stromov je v funkcií *search* najrýchlejší AVL. Ale v funckií *insert* je to červeno-čierny. Obyčajný nevyvážený strom zlyháva na usporiadaných postupnostiach, avšak na náhodnej je porovnateľný.

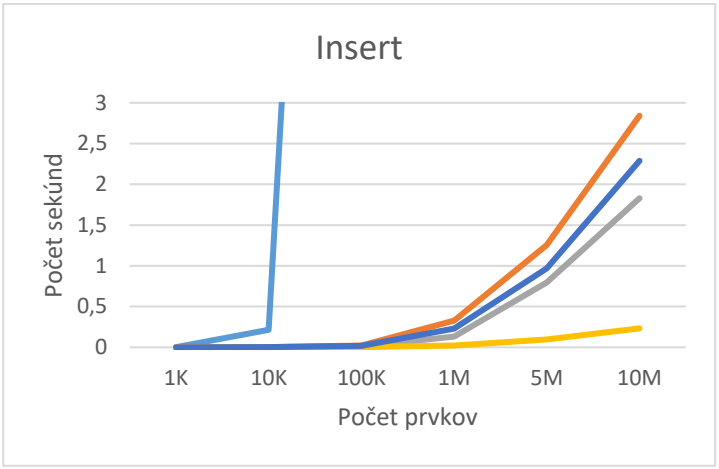
Čo sa týka hašovacej tabuľky, je vyhľadávanie takmer totožné v oboch implementáciách. Ale *insert* je pomalší u zreťazenia. Je to dôsledkom, že implementácia zreťazenia je pomalšia pri zväčšovaní tabuľky.

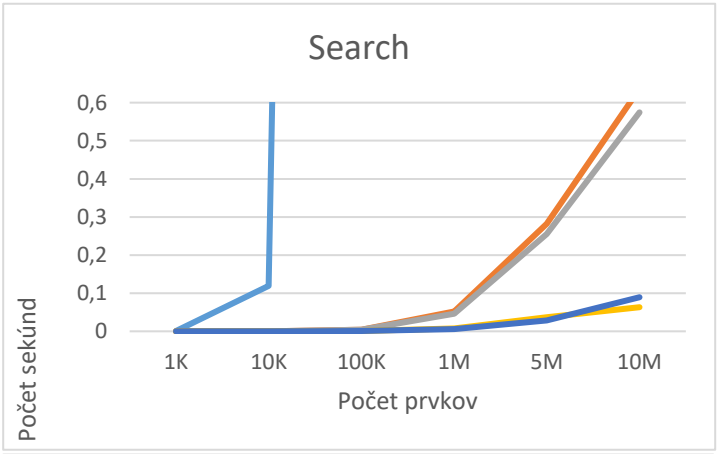
Najlepší algoritmus spomedzi testovaných vyhráva otvorene hašovanie ako aj v operácií vloženia a aj v operácií vyhľadávania.

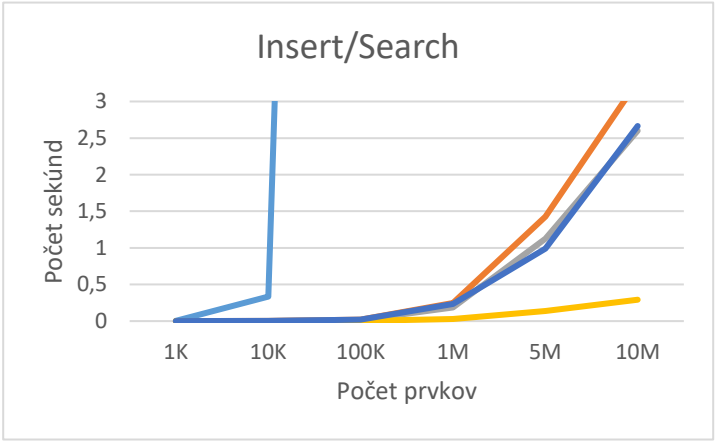
Výsledky testovania

Lineárna postupnosť

V tomto testovaní vyšiel najhoršie nevyvážený strom. Je to preto, lebo sa v podstate vytváral len zreťazený zoznam. Najlepšie dopadla hašovacia tabuľka s otvorenou adresáciou. Je dobre si všimnúť, že implementácia AVL stromu je pomalšia ako červeno-čierneho, čo sa v inom teste zmení. Vyhľadavanie v hašovacích tabuľkách je zhruba rovnaké.

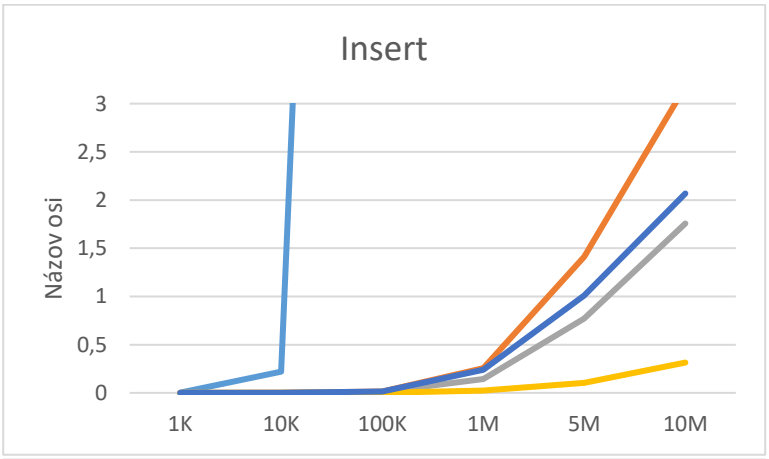


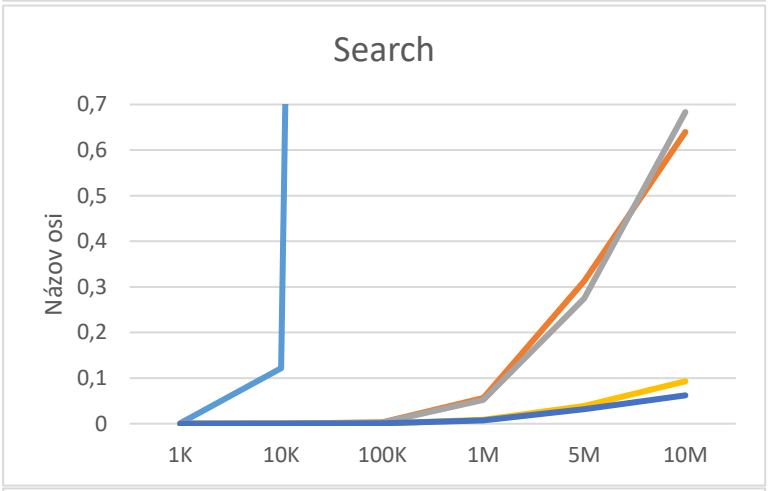


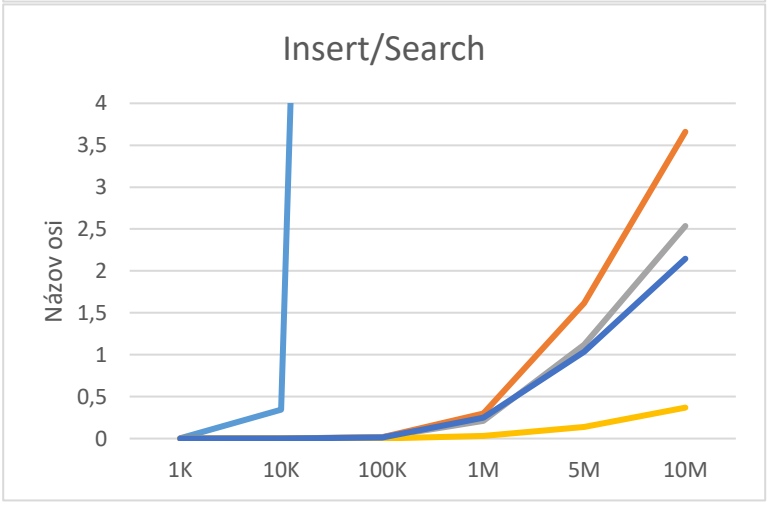


Striedavá postupnosť

Ako v predchádzajúcom teste, aj tu dopadol najhoršie nevyvážený strom. Zmena nastáva pri funkcií *search*, kde AVL strom začína získavať prevahu nad červeno-čiernym. Vyhľadavanie v hašovacích tabuľkách je aj tu zhruba rovnaké, ale pri väčších číslach je zretazenie rýchlejšie o trocha.

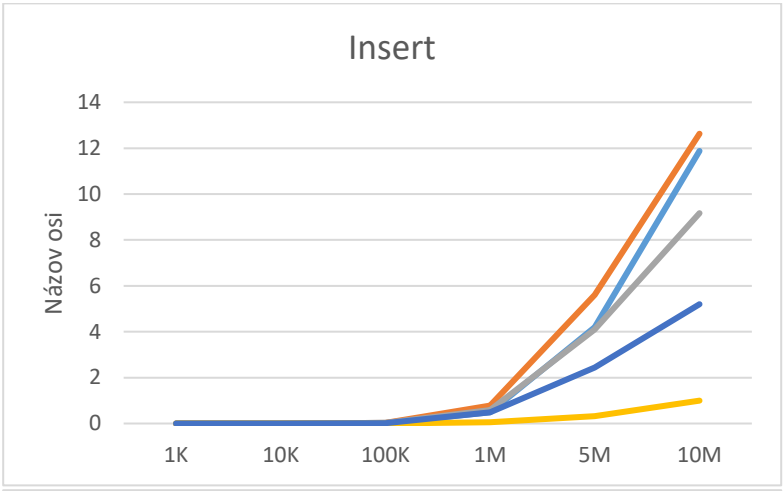


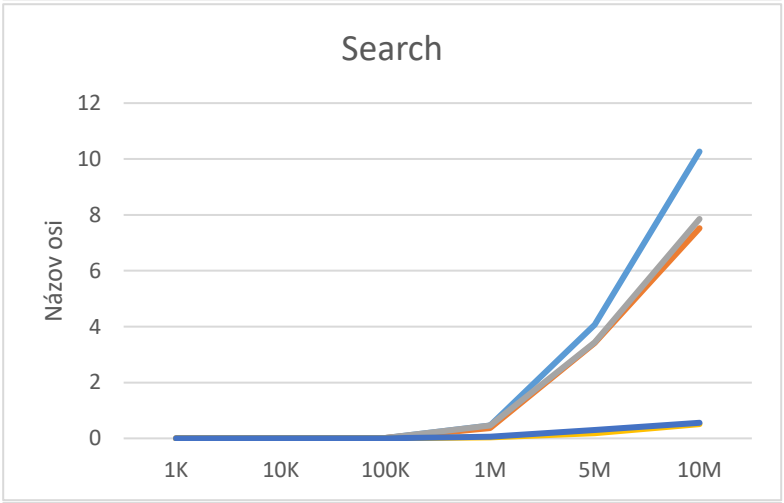


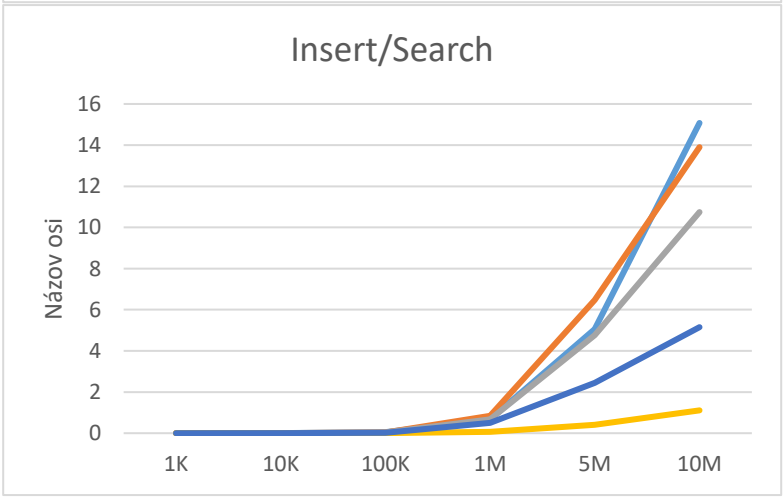


Náhodná postupnosť

V tejto postupnosti sa dá porovnávať aj nevyvážený strom. V predchádzajúcich testoch naberal obrovské veľkosti a časy, ale v náhodnej postupnosti je porovnateľný oproti vyvažovaním stromom. AVL strom už je v popredí pri vyhľadávaní, čo sa týka stromov. Avšak obe hašovacie tabuľky sú stále rýchlejšie. Ak porovnáme samotné hašovacie tabuľky, v funkcií *search* sú takmer totožné, ale v *insert* nie.







Priložil som aj excelovsky súbor s dátami zozbieranými pri testovaní.

Zdroje prevzatej implementácie

[1] Ashfaqur Rahman, 2015, https://gist.github.com/aagontuk/38b4070911391dd2806f/revisions,

[2] Manish Bhojasia, 2019, https://www.sanfoundry.com/c-program-implement-hash-tables-chaining-with-singly-linked-lists