**Zadanie 2 – Vyhľadávanie v dynamických množinách**

**Pavol Krajkovič 103004**

**FIIT STU 2020**

1. **Znenie zadania**

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.

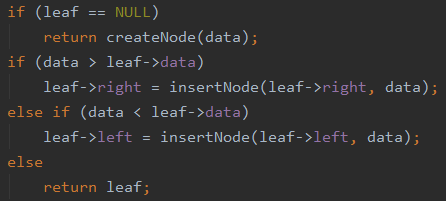
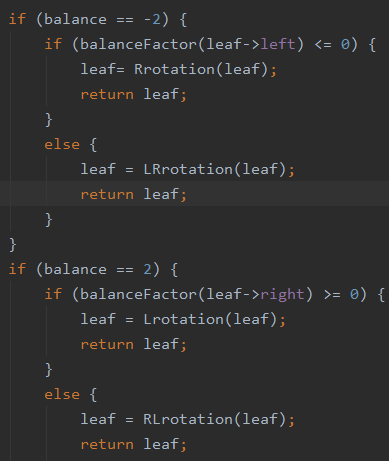
**1.1 Vlastné implementácie**

Zadanie som sa rozhodol spraviť celé a nie iba jedno porovnanie. Na vlastnú implementáciu vyváženého binárneho vyhľadávaieho stromu som sa rozhodol pouziť AVL a na vlastnú implementáciu hashovacej tabulky som si vybral typ Linear chaining.

**1.1.1 Vlastný AVL**

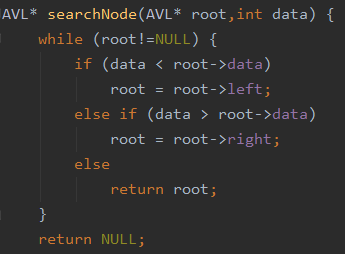
AVL metóda je jeden z najľahšie implementovatelnejších stromov. V každom strome si okrem hodnoty (data) a potomkov (left, right) pamätám aj výšku. Výška je najdlhšia cesta z daného vrchola k jeho najvzdalenejšiemu potomku. V AVL strome sa pre každý uzol rozdiel výšky dvoch podstromov detských uzlov líšia najviac o jednotku, preto je známy aj ako výškovo vyvážený. Koeficient vyváženia uzla je výška jeho pravého podstromu mínus výška jeho ľavého podstromu. Uzol s koeficientom vyváženia 1, 0 alebo -1 sa považuje za vyvážený. Uzol s koeficientom vyváženia -2 alebo 2 sa považuje za nevyvážený a vyžaduje vyváženie stromu. Koeficient vyváženia sa buď ukladá priamo v každom uzle alebo sa počíta z výšiek podstromov, ktoré sú prípadne uložené v uzloch. Uzly AVL stromu mám riešené cez štruktúru. Pre rýchlejšie vyznanie v kóde mám pomocné funkcia ako je napriklad „updateHeight, xyRotation, balanceFactor“. Keďže som spravil avl rekurzívne, nemusel som ukladať smernik na rodica v struct. Na zistinie spravnosti a správneho vyváženia AVL stroma som si spravil funkciu avlCorrect().

Kedže čo nám môže narušiť vyváženosť AVL stromu sú rotácie, tak som ich potreboval skontrolovať. V avlCorrect() si vytvorím štyri stromy. Ku každému priradím tri hodnoty tak, aby vznikli štyri scenáre na rôzne rotácie. Ak po inserte tychto hodnôt nebude mať daný strom 2 listy (tym, že rotácia nam vyvažuje strom a zabranuje tomu aby sme mali dvoch potomkov na jedenj strane a žiadneho na druhej strane, tak po rotácii bude mať daný podstrom dva listy namiesto jedneho), tak mi nevráti nič, ak je vyvážený tak mi vráti správu, že je vyvážený.

 **INSERT –** funkciu som riešil rekurzívne a je rozdelená do troch častí. V prvej časti postupujem v strome ako keby som ten uzol hladal, kde by sa mohol nachádzať. Keďže tam ešte nie je tak predposledný uzol->left/right (miesto kde by sa nachadzal uzol) bude NULL a v ďalšom cykle budem ukazovať na NULL, vtedy splním podmienku, že leaf je NULL a na tom mieste vytvorím novy list/uzol. Ak sa tento uzol už nachádza, nič sa nemení.

**INSERT 2a –** rekurziu som si vybral kvôli tejto a tretej časti. V druhej časti nastavím výšku uzola a vypočítam mu faktor vyváženosti. Logicky bude faktor 0, keďže uzol sme teraz pridali a nema žiadnych potomkov. Nebude treba rotáciu pre tento uzol, takže preskočím tretiu časť pre tento uzol. Tým, že som to spravil rekurzívne a vraciam uzol, dostávam sa po tom o level vyššie (rodica) a tak ako sme v prvej časti hladali, kde by sa mohol nachadzať nový uzol, tak sme chcene prechadzali cez budúcich predkov nového uzlu. Do sledovania som dostal rodiča nového uzlu, tak isto mu upravim velkosť, vypočítam faktor vyváženosti (na rotáciu doplatí až prarodič nového uzla) a posuniem sa na rodiča tohto rodiča (prarodica noveho uzla). Vypočítam mu novú veľkosť, faktor vyváženosti a spravím potrebné rotácie. Takto sa posuvam až na root.

**INSERT 3-** podla toho aky faktor vyváženosti sme dostali robime nasledovné rotácie. Faktor vyváženosti počítam podľa prezentacie ako výška pravého dietata – výška laveho dietata + 1.



**SEARCH –** teoreticky by som v search funkcii mohol pouziť aj prvú časť INSERT funkcie ale prišla mi to ako obrovské plytvanie pamäte pre tak ľahkú funkciu. V search nemeníme strom, iba vraciame uzol. Vďaka vlastnostiam BVS vieme presne zistiť kde by sa uzol mohol nachádzať a nemože sa stať že by sme náhodou prešli na zleho potomka. A preto nepotrebujeme predkov toho uzla, ale iba uzol. Iteratívne riešenie nie je až tak zložíté na pamäť ako rekurzia.

**1.1.2 Vlastná Hashovacia tabulka – Linear Chaining**

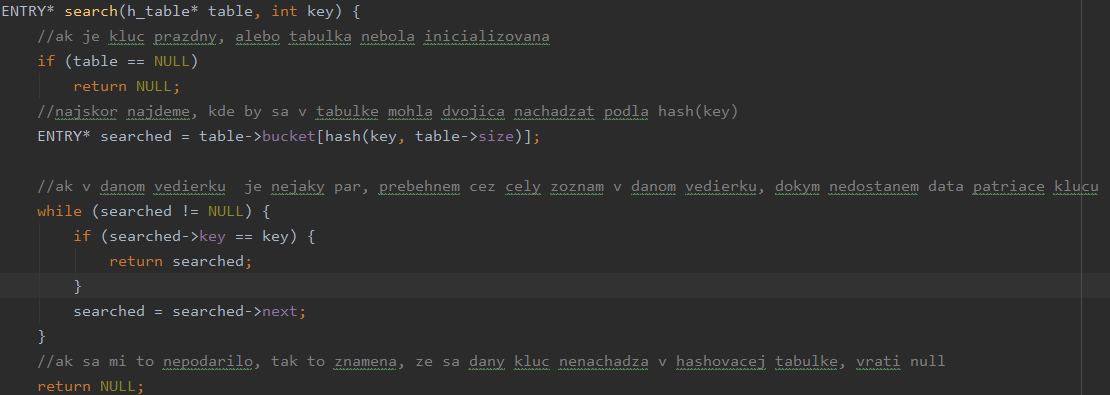
Na vlastnú implementáciu hashovacej tabulky som si vybral takú, ktorá rieši kolízie cez chaining. Tabulka mi mení veľkosť ak je koeficient naplnenia väčší ako 75 percent (počet vložených prvkov/velkosť tabulky). Tabulka berie ako kluč integer, a hashovacia funkcia mi vracia hodnotu kluča modulo velkosť tabulky. Velkosť tabulky je vždy prvočíslo aby som zabránil čo najviac kolíziam. Tabulka zahrnuje aj resize funkciu, ktorá je volaná výhradne vo funkcií insert. Trackovanie prvočísel je v liste integerov, kde mám uložené velke prvočísla.

**INSERT -** funkcia je rozdelená na tri časti. V prvej vypočítam miesto, kde sa tento kluč bude nachadzať ako poziciu v tabulke. Ak je toto miesto v tabulke prázdne, a teda nie je v tom vedierku uložený žiaden údaj. Tak vytvorím novú štruktúru a uložím tam nám zadaný kĺuč. Ak to miesto nie je prázne a teda už v tom vedierku je uložený nejaký údaj, tak sa posúvam v linked liste v tom vedierku doprava, dokým nenájdem a) rovnaký kluč – nič nevkladám

b) koniec zoznamu – vložím klúč.

Na záver, vždy po vložení kluča checknem, či netreba tabuľku resizeovať.

**SEARCH –** tak ako vo funkcií INSERT, tak aj tu si najskôr pomocou hash funkcie vypočítam miesto v tabulke. Ak to je môj klúč tak vrátim štruktúru, ak sa posúvam v poli pokiaľ nenájdem môj požadovaný klúč, ak nenájdem vrátim NULL.



**RESIZE –** funkcia resize mi zväčšuje tabulku a je výhradne volaná vo funkcii INSERT. Resize funkcia je rozdelená do nasledujúcich časti. Najskôr si vytvorím novú tabulku, ktorá bude mať velkosť dvakrát väčšie prvočíslo ako stará tabulka. Potom iterujem vedierkami starej tabulky, ak je vedierko prázdne pokračujem, ak je plné tak celý linked list z tabulky hashnem a insertnem do novej tabulky. Na záver pridadím.

**1.2 Prebraté implementácie**

Prebraté imlementácie musia fungovať na inom princípe ako moje vlastné implementácie. Preto som si vybral pre BVS typ červeno-čierneho stromu a pre hashovaciu tabulku typ otvorenej adresacie – linear probing. Keďže úlohou zadania je zdokumentovať testy a nie štruktúry tak pre prebraté implementácie nebudem popisovať ako fungujú funkcie iba všeobecne zhrniem dané dátové štruktúry. Zdroje budú spomenuté na konci.

**1.2.1 Červeno čierny strom**

Červeno čierny strom funguje na princípe:

* každý vrchol je buď červený alebo čierny
* koreň je vždy čierny
* listy sú vždy čierne
* každý čierny vrchol má 2 čiernych potomkov
* Každá cesta z jedného vrcholu k listu obsahuje rovnaký počet čiernych vrcholov
* Vložený prvok zafarbím na červeno
* Strom nie je vyrovnaný ak sú pri sebe(rodič a potomok) červené

**1.2.2 Hash table s linear probing**

Tabuľka využíva otvorene adresovanie, insert zahashuje kľúč a uloží ho na danú pozíciu, ak je daná pozícia obsadená tak sa pozrie na miesto +1 dokým nenájde voľné miesto. Implementovaná tabuľka prichádza aj s resize funkciou, ktorá zväčší tabulku keď bude load factor väčší ako konštanta (dá sa zvoliť v programe), a automaticky zväčší tabulku na 1.66 násobok starej veľkosti.

1. **Testovanie**

Aby som správne otestoval dátové štruktúry, tak som implementoval dva typy testov. Pri prvom sa prídavajú kľúče od 0 do TEST\_SIZE. V druhom type testov, mám svoj vlastný dynamický list klucov, ktoré sú pomocou prevzatej funkcie shuffle() rozhádzané. Druhý typ testu som považoval za potrebný kvôli stromom, lebo keby som pridával stále väčšie číslo napr do AVL stromu, tak sa bude hodnota vždy ukladať do pravého dieťaťa a neotestujeme všetky možné rotácie stromu.

**2.1 Pridávanie zväčšujúcich sa čísel**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **INSERT - In A Row** | 10K | 100K | 1M | 5M | 10M | 20M |
| AVL | 0.002967 | 0.053149 | 0.3601 | 1.975372 | 4.103119 | 8.529289 |
| RB\_prevzaty | 0.001415 | 0.027343 | 0.190016 | 1.04988 | 2.209189 | 4.722148 |
| HT | 0.000646 | 0.005909 | 0.056094 | 0.705359 | 1.423097 | 2.755319 |
| HT\_prevzata | 0.000215 | 0.002146 | 0.022355 | 0.332302 | 0.661484 | 1.318679 |

Pomocou grafu môžeme vidieť, že najpomalšie pridavalo prvky môj AVL strom a najrýchlejšie prevzata Hashtable s linear probing.

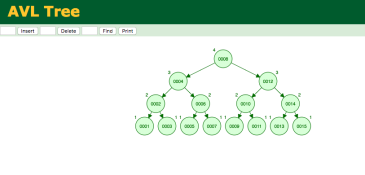
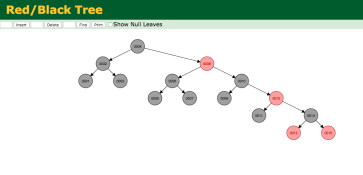
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SEARCH - In A Row** | 10K | 100K | 1M | 5M | 10M | 20M |
| AVL | 0.000645 | 0.011654 | 0.074534 | 0.397666 | 0.826297 | 1.717745 |
| RB\_prevzaty | 0.000709 | 0.012035 | 0.07926 | 0.41911 | 0.86889 | 1.865229 |
| HT | 0.000116 | 0.000951 | 0.007524 | 0.062342 | 0.125173 | 0.147822 |
| HT\_prevzata | 0.000072 | 0.000725 | 0.006968 | 0.057203 | 0.10869 | 0.127241 |

Pomocou grafu vidime že najrychlejšie hladalo prvky prevzata hashtabulka a najpomalšie červevno čierny strom. Dalej sa posuniem na porovnanie dvoch štruktúr.

**2.1.1 Porovnanie AVL s červeno-čiernym stromom**

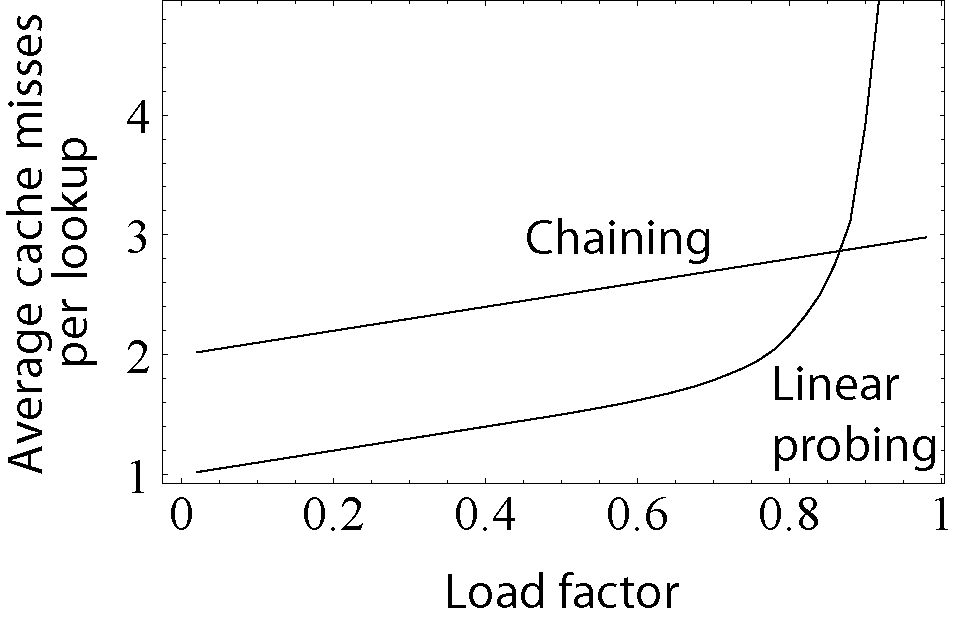
AVL stromy pri inserte (delete sme nemali implementovat) dodržujú striktnú rovnováhu pri inserte, cielom tejto štruktúry je pristupovať k údajom za čo najrýchlejší čas a tak pri priemernom inserte máme viacej rotácií, ktorým cielom je zmenšovať výšku stromu ako v červeno-čiernom strome, ktorý má volnejšiu rovnováhu. Inak povedane červeno čierne stromy potrebujú menej štrukturných zmien aby boli podli ich pravidiel vyvážene a preto je insert u nich rýchlejší. Avšak obidva stromy ponúkajú search,insert v O notácií O(logn)

AVL stromy vďaka drahému insertu sú ako som povedal viacej vyvážené a tak majú menšiu výšku ako červeno čierne stromy. Vďaka tomuto vieme pristupovať k údajom AVL stromu rýchlejšie. V O notacii majú search obidve O(logn).

 Na obrázkoch môžeme vidieť ako AVL strom má prísnejšiu štruktúru po vložení 15 prvkov ako červeno čierny strom. Zatiaľ čo AVL strom má výšku 3, červeno čierny strom má výšku 5!

Čo sa týka pamäte, pri dokonalom červeno čiernom strome by sme potrebovali iba jeden bit na uloženie farby (netýka sa to mojho prevzateho kodu) a pri AVL potrebujeme jeden integer na uloženie výšky.

**2.1.2 Porovnanie hashtable s chaining a linear probing**

Obidve tabulky majú svoju vlastnú resize() funkciu s takými istými prvočíslami a nastavené sú na taký istý loadfactor. Nevýhoda tabulky s linearnym skúšanim je, že neefektivita rýchlejšie stúpa ako v tabulke so zretazenim, ked sa približuje loadfactor k 1.

Tabulky s metodou zretazenia vlastne ani nepotrebujú funkciu resize(), lebo sa alokujú vzdy nove miesta a nikdy neprestanú fungovat, čo sa avšak týka času, tak obyčajná tabulka, ktorá má dvakrát viacej prvkov ako je jej velkosť bude potrebovať 2 krát viacej času. To vďaka tomu, že čas je lineárny so zaplnením tabulky.

V otázke pamäte je v popredí metóda s lineárnym skúšanim, kvoli tomu, že nepotrebuje alokovať zbytočné miesto v pamäti, ak sa nejedná o zväčšovanie pamate. Väčšinou avšak tabulky s linearnym skúšanim sú obrovské a k resize nedochádza tak často. Lineárne skúšanie je najlepšie implementované v prostrediach, ktoré majú málo pamäťe, s tým, že hospordári s pamätou lepšie ako metoda s chaining. Vďaka tomuto sa nám nemôže stať, že by nám nahodou malloc neuspel.

Na rozdiel od tabuliek s metodou zreťazenia, ak dostaneme pozíciu v tabulke, na ktorej sa už nachádza nejaký prvok, tak alokujem v pamäti nový blok, a uložím môj údaj do tohto bloku. Takto vytváram v každom vedierku jeden single linked list.

Čo sa ešte týka pamäte, údaje v tabulke s linearnym skúšaním sú blízko seba v pamäti zatial čo narozdiel od tabulky so zretazenim sú poprehadzovane po pamati.

Najvacši rozdiel tabulky s metodou s linearnym skúsanim potrebuje lepšie premyslenú hashovaciu funkciu, aby sa zabránilo klastrovaniu, teda veľa klúčov na malom mieste. Ak sa zaplní tabulka s linear probing, je nefunkčná na insert.

Pri prvom teste nedôjde teoreticky k žiadnym kolíziam lebo pridávam postupne čisla od jeden inkrementované +1 až po TEST\_SIZE. Obidve tabulky začínajú na tej istej veľkosti a pokial nám búdú pribúdať čísla menšie ako veľkosť tabulky (prvy test), tak sa budú udaje ukladať vedľa seba, a tak nezažiari ani jedna z týchto tabuliek a budú mať insert velmi podobný.

Najhorší prípad pre linear probing by bol ak by sme napr začali hladať/vkladať na prvom mieste a všetko až na posledne miesto by bolo zabrané a tak by sme museli prejsť celu tabulku, kde vo vyjadrení v o notacii to je O(n). Najlepší prípad je O(1), bez kolízí.

Najhorší prípad pre separate chaining by bol, ak by všetky klúče boli uložené v jednom vedierku,

tak by sme dostali tú najhoršiu vlastnosť linked list a to je, že search/insert by bol v O notacií O(n), kde n predstavuje počet uložených záznamov. Najlepší prípad zase O(1), bez kolizií.

**2.2 Test s poprehadzovanými kľúčami**

Druhý test som spravil kvôli nedostatku informácií z prvého testu. Po prvé, v AVL strome zatial dochádzalo iba k jednému typu rotácií. Po druhé, v hashovacích tabulkách sa zatiaľ nedostalo k skoro žiadnej kolízií, lebo som pridával čísla mänšie ako je tabulka.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **INSERT - Random** | 10K | 100K | 1M | 5M | 10M | 20M |
| AVL | 0.004916 | 0.062011 | 0.730349 | 3.8982 | 8.135251 | 16.195488 |
| RB\_prevzaty | 0.002028 | 0.025443 | 0.2949 | 1.59527 | 3.312537 | 6.7709964 |
| HT | 0.000709 | 0.006362 | 0.067053 | 0.823269 | 1.694205 | 3.262858 |
| HT\_prevzata | 0.000231 | 0.002555 | 0.026593 | 0.348658 | 0.700478 | 1.471885 |

Môžeme vidieť, že pri vkladaní náhodných prvkov dochádza k viacerým rotáciam a preto je insert v AVL oveľa pomalší. Očakávane je, že insert v prevzatej tabulke je oveľa rýchlejší. Inak si opäť udržali poradie aké bolo aj v prvom teste. Najviac sa teda zmenili AVL a hashtable so separate chaining.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SEARCH - In A Row** | 10K | 100K | 1M | 5M | 10M | 20M |
| AVL | 0.000956 | 0.017457 | 0.221926 | 1.198644 | 2.365947 | 4.9385162 |
| RB\_prevzaty | 0.001016 | 0.01906 | 0.2368 | 1.263487 | 2.457517 | 5.3603032 |
| HT | 0.000083 | 0.001084 | 0.025941 | 0.087817 | 0.162755 | 0.318259 |
| HT\_prevzata | 0.000082 | 0.001028 | 0.014972 | 0.065891 | 0.136191 | 0.27797 |

Stále môžeme vidieť, že search je v AVL porovnatelne rýchlejší ako v červeno čiernom strome a search v hashovacích tabulkách je približne rovnaký. V O notacií O(1).V search si tiež udržali svoje poradie.

**Použíte zdroje**

1. Implementácia červeno čierneho stromu - <https://gist.github.com/aagontuk/38b4070911391dd2806f>
2. Implementacia tabulky - <https://github.com/Apress/joys-of-hashing/blob/master/JoyChapter4-master/open_addressing_prime.c>
3. Prevzatá funkcia shuffle - <https://www.techiedelight.com/shuffle-given-array-elements-fisher-yates-shuffle/>
4. <https://sk.wikipedia.org/wiki/AVL_strom>
5. <https://sk.wikipedia.org/wiki/Ha%C5%A1ovacia_tabu%C4%BEka>