Fakulta informatiky a informačných technológií STU v Bratislave Ilkovičova 2, 842 16 Bratislava 4

Tibor Dulovec

Zadanie 2 – Vyhľadávanie v dynamických množinách

DSA LS 2020/21
Cvičiaci P. Lehoczký
Pondelok 11:00 – 12:50

Obsah

Opis zadania

Zadanie bolo zamerané na vyhľadávanie v dynamických množinách. Obsiahol som štyri implementácie. Moje vlastné implementácie sú AVL stromy a hash tabuľka s kolíziami riešenými cez chaining.

Zadanie je naprogramované v jazyku Java. Všetky implementácie sú ako samostatné triedy a majú vlastné uzly v ďalšej triede.

V hlavnej triede „main“ sú funkcie pre všetky testy. Testy sú na vzorke 100 tisíc záznamov, ale pre možnosti testovania je vzoriek viac a pre vyskúšanie iných záznamov stačí zmeniť názov súboru hneď na prvom riadku v súbore main.

Obsiahnuté implementácie

- AVL stromy
- Splay Tree
- Hashtable: Chaining
- Hashtable: Open addressing

Zdroje prevzatých implementácií

Obe prevzaté implementácie boli upravené, pre správne fungovanie s mojím kódom.

Splay Tree

https://algorithmtutor.com/Data-Structures/Tree/Splay-Trees/

Hashtable: Open addressing

https://www.algolist.net/Data_structures/Hash_table/Open_addressing

AVL stromy

AVL strom je prvý vynájdený samovyvažovací binárny vyhľadávací strom. Princíp jeho vyvažovanie je v balancií každého uzla. Ta spočíva v rozdiely výšok pravého a ľavého uzla, ktorá sa prepočítava zakaždým po pridaní nového uzla. Vďaka rovnomernému vyváženiu je následné vyhľadávanie prvkov efektívnejšie.

Časová zložitosť vyhľadávania a vkladania je **O(log(n))**.

Funkčnosť AVL stromov je obsiahnutá v triede „AVL“. Pre správne fungovanie tejto triedy je potrebná aj trieda „AVLNode“, podľa ktorej sú vytvorené dane uzly v AVL strome.

Funkcie AVL stromov

Pridávanie hodnoty: addItem()

Za pomocou tejto funkcie pridávame nové uzly to stromu. Táto funkcia je **rekurzívna**. Ak sa dostaneme na miesto v uzly, ktoré ešte nie je obsadené, tak vložený objekt uložíme na jej miesto. Ale ak sa na tom mieste už nejaký objekt nachádza, tak na ten objekt(uzol) znova zavoláme funkciu **addItem**. Potom, ako sa pridá nový uzol, všetky uzly, ktoré boli týmto ovplyvnené vykonajú funkciu triedy AVLNode **rebalanced().** Týmto spôsobom vždy keď pridáme novú hodnotu, tak sa vyvážia všetky uzly, cez ktoré táto hodnota prešla.

Vyváženie uzla rebalanced()

Funkcia je pre triedu AVLNode. Jej úlohou je vyvážiť strom, aby bol viac efektívny, následne cez návratovú hodnotu vrátiť vyvážený uzol.

Trieda AVLNode má okrem iného aj vlastnosti „**depth**“ a „**balance**“, ktoré predstavujú hĺbku a balanciu uzla. Hĺbka značí, na akej je výške.
Napríklad 1 znamená, že pod ňou už nič nie je a 3 znamená, že na jednej strane je uzol s výškou 2.

Na začiatku funkcie rebalanced() sa zavolá funkcia **getBiggerItem()**, ktorá vráti prvok s väčšou hĺbkou z daného uzla. Buď ľavý alebo pravý a podľa neho sa zvýši hĺbka uzla.

Ďalej sa nanovo vypočíta balancia. Tu vypočítame rozdielom hĺbok vnútorných uzlov. Nasledujú štyri if podmienky, ktoré podľa balancie rozhodujú či a akú rotáciu uzol potrebuje. V prípade že je balancia väčšia ako jedna. Vykoná sa **pravá rotácia**. Ak je menšia, vykoná sa **ľavá rotácia**.

Tieto rotácie prebiehajú tak, že sa zmení hodnota hĺbky a daný uzol sa vymení s pravým alebo ľavým. Záleží od toho, či je potrebná pravá alebo ľavá rotácia.
Ľavá a pravá rotácia sa vykonáva v podstate rovnako. Rozdeľujú sa iba tým, ktoré strany sa kde premiestnia.

Existujú ale určite scenáre, kedy sa môže stať, žeby tieto rotácie nefungovali správne.
V týchto prípadoch je potrebné vykonať **right-left** alebo **left-right rotáciu.**

V prípade left-right rotácie overujeme, či balancia ľavej strany uzla je menšia ako 0.
V prípade right-left rotácie overujeme pravú stranu. A to či je balancia väčšia ako 0. Ak áno,

vykonáme prehodenie uzlov a následne spravíme left alebo right rotáciu. Záleží od toho, ktorá rotácia bola vykonaná pred tým.

Po vykonaní rotácií, sa vráti vyvážený uzol a vďaka rekurzívnej funkcie sa celý strom aktualizuje.

Vyhľadávanie hodnoty: findItem()

Funkcia na vyhľadávanie prvku je veľmi podobná ako pri vkladaní. Tiež je rekurzívna a rovnakým spôsobom prechádza prvkami. Rozdiel ale je, že funkcia sa ukončí, keď sa nájde zhoda s hľadaným menom a menom v uzli. Potrebné atribúty pre túto funkciu sú hľadaný výraz a strom v ktorom treba hľadať. Užívateľ zväčša funkcia volá v celom strome, ktorý je uložený v „root“.

Testovacie funkcia pre výpis stromu: printTree()

Táto funkcia je určená iba pre a vývojárske účely. Slúži na vypísanie kódu. Výpis obsahuje každý uzol. Meno (klúč) záznamu a v zátvorke je balancia a hĺbka uzla. Za šípkov sú uzly, ktoré sú na ľavej a na pravej strane. Celý výpis je ukončený riadkom, koľko uzlov strom obsahuje.

```
Zoe Warner(0/1)=> -/-
Zoe Watson(0/1)=> -/-
Zoe Willson(1/5)=> Zoe Wigley/Zoe Wood
Zoe Wigley(-1/4)=> Zoe Wheeler/Zoe Wilkinson
Zoe Wheeler(0/3)=> Zoe West/Zoe Widdows
Zoe West(-1/2)=> -/Zoe Weston
Zoe Weston(0/1)=> -/-
Zoe Widdows(1/2)=> Zoe Whitmore/-
Zoe Whitmore(0/1)=> -/-
Zoe Wilkinson(-1/3)=> Zoe Wild/Zoe Wills
Zoe Wild(0/1)=> -/-
Zoe Wills(1/2)=> Zoe Williams/-
Zoe Williams(0/1)=> -/-
Zoe Wood(0/4)=> Zoe Wilson/Zoe Wooldridge
Zoe Wilson(0/2)=> -/Zoe Windsor
Zoe Windsor(0/1)=> -/-
Zoe Wooldridge(0/3)=> Zoe Woodley/Zoe Yoman
Zoe Woodley(0/2)=> Zoe Woodcock/Zoe Woods
Zoe Woodcock(0/1)=> -/-
Zoe Woods(0/1)=> -/-
Zoe Yoman(0/2)=> Zoe Yarwood/Zoe Zaoui
Zoe Yarwood(0/1)=> -/-
Zoe Zaoui(0/1)=> -/-
Count of items is: 163270
```

Splay Tree

Implementáciu som prevzal zo stránky algorithmtutor.com. Má podobné výhody ako AVL strom. Je samo vyvažujúca a da sa podobne vizuálne znázorniť. Ale uzly sa otáčajú aj po vyhľadávaní. To znamená, že keď pristúpime k nejakému prvku, tak prvok sa posunie vyššie v strome a do budúcna bude rýchlejšie dostupný.

Vzhľadom k tomu, jeho časová zložitosť nemusí byť vždy rovnaká. Môže byť od **O(log(n))** až po **O(n)**

Vkladanie a hľadanie je štandardné ako pri ostatných vyvážených stromov. Rozdiel nastane po vyhľadaní prvku. Vtedy sa uzol otáča ku koreňu stromu. Vďaka tomu, ak budeme v blízkej dobe opakovane vyhľadávať tento prvok, tak bude rýchlejšie prístupný a prvky, ktoré sa vôbec nevyhľadávajú budú hlboko v strome a nebudú negatívne ovplyvňovať čas vyhľadávania. Tieto rotácie sa nazývajú **Zig Zag rotácie.** V určitých scenároch, podobne ako pri AVL stromoch, je potrebná **Zig-zig rotácia**, ktorá je podobná right-left alebo left-right rotáciám.

Výhoda použitia tejto implementácia je, ak sa často pristupuje k rovnakým prvkom. Napríklad pre cash pamäť. Výrazným časovým problémom sa ale stáva to, keď sa k prvkom pristupuje veľmi náhodne.

Hashtable

Hashtable je dynamická množina, ktorá je veľmi efektívna vďaka nižšej časovej náročnosti. Tento princíp spája **kľuč** s hodnotami a preto ak sa chceme dostať k hodnote, stačí poznať kľuč, ktorý je v našom prípade meno, a okamžite dostaneme prvok, ktorý sme vyhľadávali. Pridávanie funguje podobne efektívne.
Časová efektívnosť pridávania a vyhľadávania je **O(1)**

Časová efektívnosť sa môže zvyšovať v scenároch, ktoré sa nazývajú **kolízie**. Stanú sa vtedy, keď po zhashovaní kľúču vyjde hodnota, ktorá sa vygenerovala už pre tým. Práve preto sa snažíme dosiahnuť, aby bol hash čo najviac unikátny. Nie vždy to je možné a preto na riešenie kolízií existuje niekoľko spôsobov. Jeden z najrozšírenejších je **Chaining** (reťazenie).

Chaining

Chaining je riešenie kolízií pre hashovacie tabuľky. V prípade, že kľúč po zhashovaní bude mať hodnotu, ktorá sa už opakovala. Prvok sa napojí na k opakovanému prvku.

Pridanie hodnoty: insert()

Parametre pre túto funkciu sú kľuč a objekt, ktorý chceme pridať. Na začiatku funkcie sa vygeneruje index podľa hash funkcie, do ktorej sa vloží kľúč.

Ak na tomto indexe nič nie je, vložíme tam daný objekt. Ak sa tam ale už niečo nachádza, je potrebné nový objekt zreťaziť.

Uzol pre Hashovanie s kolíziami riešenými cez chaining je definovaný podľa triedy „**ChainingNode**“. V prípade, že je potrebné prvky reťaziť, tak sa ukladajú do prvého prvku na danom indexe. Prvok má vlastnosť ArrayList, v ktorej sú uložené všetky zreťazené prvky. Následne sa do neho pridáva alebo ním prechádza, ak je potrebné niektorý prvok nájsť.

Vyhľadávanie prvku: get()

Parameter pre túto funkciu je kľúč. Tento kľúč sa rovnakou funkciou, ako pri pridávaní, použije na vygenerovanie indexu. A v časovej náročnosti O(1) sa v poli prvkov vyberie. Ak na tomto mieste nič nie je, znamená to že bol zadaný nesprávny kľúč a vráti sa hodnota null. Ak sa nachádza, overí sa či sa meno prvku zhoduje s kľúčom. Ak áno, funkcia tento prvok vráti.

Ak sa ale na danom mieste prvok nachádza a meno sa stále nezhoduje s kľúčom, môže to znamenať, že prvok je v reťazení a preto sa celý ArrayList prehľadá a skúsi nájsť zhoda.

V ideálnom prípade bude zreťazenie minimálne. Ale čím je viac dát na čím menšom poli prvkov, tak tým menej bude vznikať jedinečných indexov a zreťazenie bude narastať. Funkcia „**resizeTable**“ slúži nato, aby v prípade, že sa počet voľných indexov bude výrazne zmenšovať, zväčší celú hash tabuľku.

Zväčšenie tabuľky: resizeTable()

Pri každom pridávaní nového prvku na miesto indexu, sa do hodnoty „usedIndex“ zaznačí využitie nového indexu. Ak voľných indexov je menej ako polovica, tak sa zavolá funkcia resizeTable. Táto funkcia vytvorí nový zoznam prvkov do ktorého podľa nového hashovania

priradí každý jeden prvok. Na konci funkcie táto nová tabuľka, s novou veľkosťou, nahradí tu starú.

Výhoda tejto funkcie je tá, že v prípade, keby sa do kódu implementovala funkcia na vymazávanie prvkov, tak jej upravenie na zmenšenie tabuľky by bolo veľmi jednoduché.

Hashovacia funkcia: hash()

Pôvodne bola táto funkcia obsiahlejšia aby generovala dostatočne jedinečný index. Nakoniec som ju nahradil jednoduchou funkciou „hashCode“, ktorá je v jazyku Java dostupná pre String. Efektívnosť na veľkosti 300 tisíc bola identická, tak som ju touto Java funkciou nahradil.

Open addressing

Táto implementácia bola prezvaná z webstránky: algolist.net

Open addressing je ďalší spôsob riešenia kolízií. Nevzniká tú žiadne reťazenie a všetky prvky sú uložené v poli prvkov. Vďaka tomu je táto implementácia jednoduchšia na zhotovenie. V prípade, žeby všetky vygenerované indexy boli unikátne tak by bola aj časovo efektívnejšia. To ale nevieme zaručiť a tak nárastom prvkov pri tejto implementácií sa časová náročnosť zhoršuje.

Open addressing funguje tak, že ak sa index po hashovani opakuje, vloží sa na najbližší voľný prvok. Pri vyhľadávaní sa potom v prípade prvku s kolíziou, musí prehľadávať celá pamäť, pretože prvok môže byť posunutý veľmi ďaleko.

Testovanie

Testované boli primárne časové zložitosti. Každá implementácia mala odmeranú dobu pridávania a dobu vyhľadávania každého jedného prvku. Vyhľadával som každý jeden prvok, a nie len zopár vybraných, to z dôvodu, aby som zistil či sú všetky prvky úspešne pridané. V súbore „main“ sú všetky testy volané.

Záznamy na testovanie sú importované zo súborov vo formáte „csv“. Tie som vygeneroval pomocou online stránky. Projekt obsahuje 7 týchto súborov, ktoré sa rozlišujú počtom záznamov. Od 50 do 300 000.

Pomocou funkcie „**getItemFromCsvFile**“ sa všetky tieto záznamy nahrajú do ArrayListu „**importedItems**“. Tieto záznamy potom používajú všetky testovacie funkcie rôznych implementácií. Používajú sa rovnaké záznamy pre presné porovnávanie časovej efektívnosti.

```
ArrayList<Person> importedItems = getItemFromCsvFile( fileName: "test300k");

System.out.println("====================================");
System.out.println("Trees: AVL");
AVL avl = new AVL();
AVLInsertItems(avl, importedItems);
AVLSearchItems(avl, importedItems);

System.out.println("====================================");
System.out.println("Trees: Splay Tree");
SplayTree splayTree = new SplayTree();
splayTreeInsertItems(splayTree, importedItems);
splayTreeSearchItems(splayTree, importedItems);

System.out.println("====================================");
System.out.println("Hashtable: Chaining");
Chaining chaining = new Chaining(importedItems.size());
chainingInsertItems(chaining, importedItems);
chainingSearchItems(chaining, importedItems);

System.out.println("====================================");
System.out.println("Hashtable: Open addressing");
Addressing<String, Person> addressing = new Addressing<>(importedItems.size());
addressingInsertItems(addressing, importedItems);
addressingSearchItems(addressing, importedItems);
```

V hlavnej main funkcií sa následne spustia všetky testovacie funkcie. Z dôvodu prehľadnosti sú veľmi podobné názvom aj telom funkcie.

Testovacia funkcia InsertItems()

Každá implementácia ma vlastnú funkciu pre vloženie všetkých prvkov do pamäte. Na začiatku funkcie sa uloží stopa času, ktorá sa porovná s časom po vykonaní funkcie. Tento časový rozdiel sa vypíše aj s údajom koľko celkovo údajov bolo do pamäte vložených.

Testovacia funkcia SearchItems()

Každá implementácia ma vlastnú funkciu ktorá vyhľadá všetky prvky, ktoré mali byť uložené do pamäte. Na začiatku a na konci funkcie sa tak isto uložia stopy času, ktorých rozdiel sa potom vypíše po vyhľadaní všetkých prvkov. K času sa vypíše aj koľko prvkov z koľkých bolo nájdených. V ideálnom prípade to budú vždy dva rovnaké čísla.

Rozdiel unikátnych a opakovaných kľúčov pre hashtable

Pre zlepšenie efektívnosti bolo potrebné vidieť pomer unikátnych kľúčov na konci pridávania všetkých objektov. Z dôvodu, že počet unikátnych kľúčov je potrebný ukladať z pre potreby zmenenia veľkosti tabuľky, tak to ani neovplyvňuje chod programu.

```
Hashtable: Chaining
100000 items was added in: 69 ms
Unique keys: 60820
Duplicated keys: 17155
(100000/100000) items was found in: 35 ms
```

Výpis AVL stromu

Pre testovanie bolo potrebné vidieť stromy. Trieda AVL obsahuje funkciu „**printTree**“, ktorá celý tento strom vypíše.
Formát záznamu je: Kľuč(balancia/hĺbka)=>ľavý uzol/pravý uzol
Tento výpis je ukončený počtom prvkov vo strome

```
Zoe Wilson(0/2)=> -/Zoe Windsor
Zoe Windsor(0/1)=> -/-
Zoe Wooldridge(0/3)=> Zoe Woodley/Zoe Yoman
Zoe Woodley(0/2)=> Zoe Woodcock/Zoe Woods
Zoe Woodcock(0/1)=> -/-
Zoe Woods(0/1)=> -/-
Zoe Yoman(0/2)=> Zoe Yarwood/Zoe Zaoui
Zoe Yarwood(0/1)=> -/-
Zoe Zaoui(0/1)=> -/-
Count of items is: 163270
```

Výpis rotácií pri vyvažovaní v AVL stromoch

Pre prehľad vykonaných rotácií je možné v triede „AVLNode“ zmeniť boolean vlastnosť „**debug**“ na true a vypíše sa každá vykonaná rotácia.

```
Cameron Ellery had left-right rotation
Cameron Ellery had right rotation
Eduardo Lewis had left rotation
Javier Irving had right rotation
Brooklyn Appleton had right rotation
Bree Simpson had right rotation
Livia Herbert had right rotation
Rocco Bentley had left rotation
Leroy Paterson had left-right rotation
Leroy Paterson had right rotation
Clint Yarwood had right-left rotation
Clint Yarwood had left rotation
Melinda Quinton had right-left rotation
Melinda Quinton had left rotation
Holly Todd had right-left rotation
Holly Todd had left rotation
```

Vyhodnotenie dosiahnutých výsledkov

Po zbehnutí všetkých testov máme jasný prehľad na vzorke 100k záznamov, ktoré dynamické pamäte sú najviac efektívne.

```
Trees: AVL
100000 items was added in 115 ms
100000(/100000) items was found in: 98 ms
====================================
Trees: Splay Tree
100000 items was added in 146 ms
100000(/100000) items was found in: 188 ms
====================================
Hashtable: Chaining
100000 items was added in: 68 ms
(100000/100000) items was found in: 36 ms
====================================
Hashtable: Open addressing
100000 items was added in: 75 ms
(100000/100000) items was found in: 50 ms
```

Hashtables sú výrazne efektívnejšie v porovnaní so samovyvažovacími stromami. A pomedzi hashtables je efektívnejšie riešenie kolízií cez metódu reťazenia. Tiež môžeme vidieť, že Splay Tree bol v tomto prípade pri vyhľadávaní najmenej efektívny. To je z dôvodu, že vyhľadávacie prvky sa opakovali veľmi zriedkavo.