DSA Zadanie 2:

Vyhľadávanie v dynamických množinách

Peter Kúdela

STU
FIIT
Slovensko
11. 04. 2020

Table of Contents

AVL strom (vlastný) dokumentácia:

Strom obsahuje uzly ktoré sa skladajú z:

- Ukazovateľa na ľavý podstrom
- Ukazovateľa na pravý podstrom
- Hodnoty uzla
- Výšky uzla

Insert

Prvá časť funkcie sa rekurzívne vnára do stromu vkladajúc zadanú hodnotu. Ak strom ešte neexistuje vráti nový uzol ako koreň stromu. Nový prvok začína ako list (ukazovatele na potomkov sú NULL a výška je 1). V druhej časti funkcie sa vynára z rekurzie pričom opravuje výšky rodičovských uzlov a zároveň sleduje či niektoré z nich sa nestali nevybalancovanými (výška pravého podstromu – výška ľavého podstromu nie je z rozsahu <-1, 1>). Po vynorením z rekurzie funkcia vracia nový koreň stromu. Balancovanie prebieha nasledovne.

Balance

Na balancovanie sme vytvorili 4 funkcie pre 4 prípady ktoré môžu nastať. Nový uzol sme od súčastného uzla pridávali smerom:

- Doľava a znovu doľava (Left left)

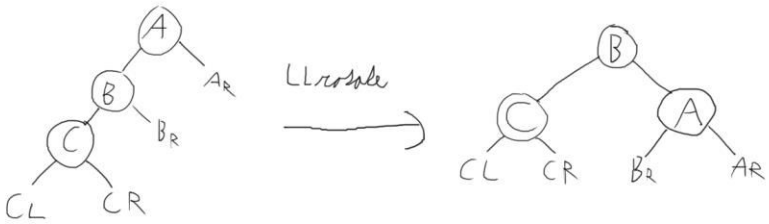

Figure1: Left Left rotation

- Doprava a znovu doprava (Right right)

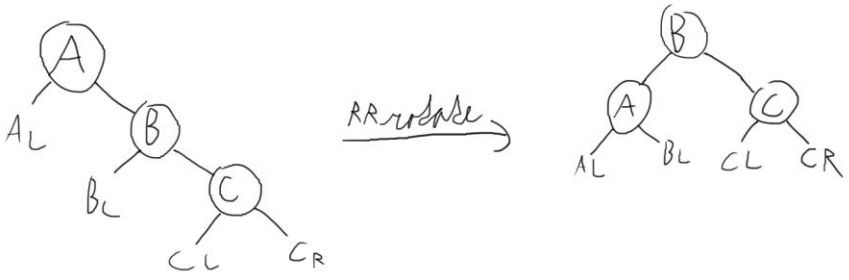

Figure 2: Right Right rotation

- Doľava nasledovne doprava (Left right)

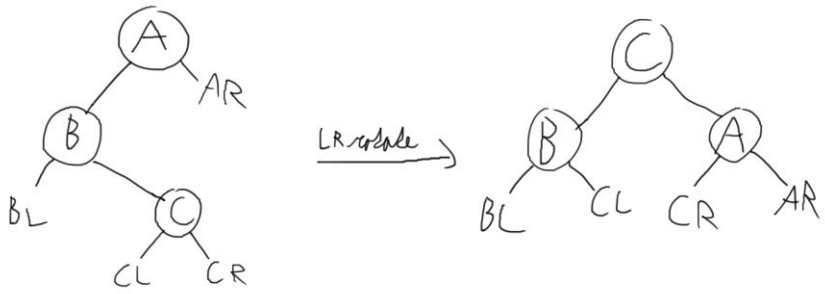

Figure 3: Left Right rotation

- Doprava nasledovne doľava (Right left)

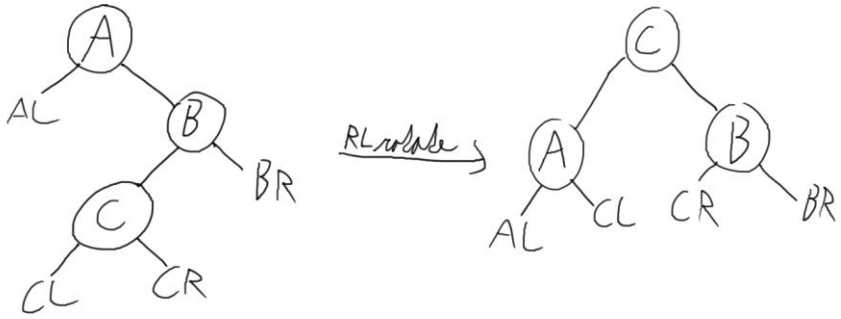

Figure 4: Right Left rotataion

Search

Táto funkcia vráti daný uzol ak sa v strome nachádza, v opačnom prípade vráti NULL. Funkcia sa iteratívne vnára do stromu, pri každom uzle sledujúc, či je hodnota hľadaného uzla väčšia alebo menšia ako súčastného uzla.

Kontroal správnosti

Strom bol testovaný na malom aj veľkom počte pridaných prvkov, správnosť balancovania a celkového prídávania kontrolovaná debugovacími nástrojmi vo VS a pomocnými výpismi do konzole. Velkosť hodnôt v uzloch je limitovaná dátovým typom int.

Hašovacia tabuľka (vlastná: reťazenie) dokumentácia:

Tabuľka je reprezentovaná štruktúrou, ktorá obsahuje :

- velkosť tabuľky
- počet prvkov v tabuľke
- zoznam vedierok pre prvky

Zoznam vedierok je tvorený ako pole začiatkov spájaných zoznamov s prvkami. Každý prvok obsahuje ukazovateľ na ďalší v poradí vo vedierku a hodnotu. Vkladané hodnoty sú v mojej implementácií zároveň jej kľúčom. Vďaka pozmenení hašovacej funkcie je možné vkladať aj záporné hodnoty, takéto riešenie však značne spomalilo pridávanie prvkov.

Hash

Keďže implementácia používa menenie veľkosti tabuľky a preto musí často pre-hašovať všetky jej hodnoty, rozhodol som sa pre čo najjednoduchšiu funkciu, a to lineárne sondovanie.

Insert

Pri pridaní nového uzlu najprv kontroluje či nejaká tabuľka už existuje, ak nie, vytvorí novú o veľkosti 2. Nasledovne zahašuje pridávanú hodnotu, pozre sa na daný index v tabuľke, ak je toto miesto voľné pridá do neho uzol s novou hodnotou, ktorý ukazuje na NULL. Ak nastane koliízia a na tomto mieste sa už uzol nachádza, pridá uzol na koniec spájaného zoznamu. Pri každom pridaní taktiež kontroluje pomer počtu prvkov v tabuľke a jej veľkosti podľa ktorého sa rozhoduje, či je treba zmeniť veľkosť tabuľky.

Resize

Funkcia prechádza starú tabuľku a všetky prvky vkladá do tabuľky o veľkosti prvého prvočísla väčšieho ako číslo 2-krát väčšie veľkosti pôvodnej tabuľky. Pritom taktiež uvoľnuje pôvodnú tabuľku.

Search

Funkcia ako prvé kontroluje či daná tabuľka existuje, ak áno vypočíta haš hodnoty a skontroluje daný index, ak sa na tomto indexe nachádza nejaký uzol kontroluje či sa jeho hodnota rovná hľadanej, ak nie opakuje proces až po koniec spájaného zoznamu. Ak sa prvok našiel, vráti jeho uzol, ak nie vráti NULL.

Kontrola správnosti

Tabuľka bola testovaná na malom aj veľkom počte pridaných prvkov, správnosť pridávania kontrolovaná pomocnými výpismi do konzole funkciou showTable(HT* table) ktorá na každý riadok vypíše obsah jedného vedierka. Veľkosť tabuľky je obmedzená dátovým typom int.

Červeno čierny strom (prevzatý) dokumentácia:

Strom obsahuje uzly ktoré sa skladajú z:

- Ukazovateľa na ľavý podstrom
- Ukazovateľa na pravý podstrom
- Ukazovateľa na rodičovský uzol
- Farbu uzlu

Insert

Funkcia si na začiatku vytvorí nový uzol, ktorý nasledovne vkladá iteratívnym spôsobom do binárneho stromu. Po jeho vložení mu nastaví rodičovský uzol. Po tomto vložení funkcia posiela koreň stromu a nový uzol do funkcie void RB_insert_fixup(struct Node** T, struct Node** z).

Insert fixup

Táto funkcia funguje na balancovanie stromu po vložení uzlu. Iteratívne prehliada strom, začína v novo vloženom uzle. Pozrie sa akej farby je strýko tohto uzla, ak je červený, tak zmení farbu rodiča a strýka na čiernu, a starého rodiča na červenú. Ak je strýko čierny aplikuje rotácie podľa danej situácie.

Search

Vyhľadávacia funkcia implementovaného algoritmu funguje rekurzívne. V strome sa vnára do podstromu podľa toho, či je hľadaná hodnota väčšia alebo menšia ako terajšieho uzla.

---

amitbansal7, Github,18. 04. 2017, https://github.com/amitbansal7/Data-Structures-and-Algorithms/blob/master/9.Red-Black-tree/RedBlackTrees.c, Accessed 19. 04. 2020

Hašovacia tabuľka (prevzatá: otvorené indexovanie) dokumentácia:

Tabuľka je reprezentovaná globálnou premennou ktorá je pole odkazov na uzly. Uzly sú reprezentované ako DataItem a držia v sebe:

- Data
- Key

Dáta sú hodnota ktorú uzol drží, kľúč slúži na vkladanie a hľadanie daných hodnôt. Pre jednoduchosť testovania som do implementácie pridal funkciu ktorá resetuje globálnu premennú s tabuľkou.

Insert

Vložený kľúč funkcia zahešuje, pozrie sa na daný index. Ak sa na indexe už nachádza iný uzol posúva sa doprava (ak sa dostanie nakoniec skočí späť na začiatok tabuľky) až kým nenájde voľné miesto pre nový uzol.

Search

Podobne ako insert, daný kľúč zahešuje, pozrie sa na index, ak je index prázdny, daný kľúč sa v tabuľke nenachádza, ak sa kľúč nezhoduje s nájdeným kľúčom uzlu na danom indexe, posúvame sa doprava. Ak sa prvok nenájde vracia NULL, ak sa nájde vracia daný uzol.

---

tutorialspoint,
https://www.tutorialspoint.com/data_structures_algorithms/hash_table_program_in_c.htm, Accessed 19. 04. 2020

Testovanie:

```
===MY AVL TREE===
Inserting 1000 nodes in ascending order took 2 miliseconds.
Searching 1000 nodes 1000000 times took 75 miliseconds.
Inserting 1000000 nodes in ascending order took 4005 miliseconds.
Searching 1000000 nodes in ascending order took 123 miliseconds.
Inserting 1000000 nodes in descending order took 3781 miliseconds.
Searching 1000000 nodes in descending order took 125 miliseconds.
Inserting 1000000 random nodes took 4158 miliseconds.

===CITED RB TREE===
Inserting 1000 nodes in ascending order took 1 miliseconds.
Searching 1000 nodes 1000000 times took 274 miliseconds.
Inserting 1000000 nodes in ascending order took 443 miliseconds.
Searching 1000000 nodes in ascending order took 555 miliseconds.
Inserting 1000000 nodes in descending order took 448 miliseconds.
Searching 1000000 nodes in descending order took 555 miliseconds.
Inserting 1000000 random nodes took 1333 miliseconds.

===MY HASH TABLE===
Inserting 1000 nodes in ascending order took 0 miliseconds.
Searching 1000 nodes 1000000 times took 58 miliseconds.
Inserting 1000000 nodes in ascending order took 951 miliseconds.
Searching 1000000 nodes in ascending order took 84 miliseconds.
Inserting 1000000 nodes in descending order took 930 miliseconds.
Searching 1000000 nodes in descending order took 88 miliseconds.
Inserting 1000000 random nodes took 1203 miliseconds.

===CITED OPEN HASH TABLE===
Inserting 1000 nodes in ascending order took 0 miliseconds.
Searching 1000 nodes 1000000 times took 53 miliseconds.
Inserting 1000000 nodes in ascending order took 207 miliseconds.
Searching 1000000 nodes in ascending order took 68 miliseconds.
Inserting 1000000 nodes in descending order took 208 miliseconds.
Searching 1000000 nodes in descending order took 67 miliseconds.
Inserting 1000000 random nodes took 133 miliseconds.
```

Figure 5: Výsledky testovania

Pridávanie prvkov

Porovnávanie pridávania malého množstva prvkov je takmer nemožné, hodnoty sú v desiatkach milisekúnd, preto sme sa rozhodli algoritmy porovnávať na základe pridávania miliónu prvkov. Medzi pridávaním prvkov zoradených vzostupne a zostupne nebol takmer žiadny rozdiel ani v jednom z algoritmov.

Zo všetkých štyroch algoritmov je moja implementácia AVL stromu najpomalšia. Oproti ostatným implementáciam je pridávanie pomalé hlavne kvôli použitiu rekurzie na vnáranie sa do stromu. Červeno čierny strom ktorého implementáciu sme zobrali z internetu je v pridávaní cca 8-krát rýchlejšia vďaka vnáraniu sa iteratývnym spôsobom.

Rozdiely v mojej a prevzatej hašovacej tabuľke sú asi 5:1. Moja implementácia je pomalšia kôli potrebe meniť veľkosť tabuľky, hľadania prvočísiel a pomalšej hash funkcie. Prevzatá tabuľka je rýchlejšia pre jej statickú veľkosť .

Hľadanie prvkov:

Rovnako ako pri pridávaní, v hľadaní prvkov vzostupne vs zostupne sa nevyskytol žiaden rozdiel.

Najrýchlejšie boli obe implementácie hašovacých tabuliek, prevzatá tabuľka bola asi o 25% rýchlejšia ako naša, tento rozdiel je pravdepodobne spôsobený rozdielom v použitých hašovacích funkciach.

Medzi naším AVL stromom a prevzatým červeno čiernym stromom bolo v hľadaní prvkov značný rozdiel. Keďže naša implementácia prvky hľadá iteratívne zatiaľ čo prevzatý kód rekurzívne AVL strom je schopný prvky vyhľadať omnoho rýchlejšie.