Raport Final
Proiect Structuri de date

Bejan-Topse Cosmin

Drobnitchi Daniel

Suto Robert

Facultatea De Matematica si Informatica, Universitatea Bucuresti

Contents

1 Introducere

Echipa noastra a ales sa implementeze un arbore AVL.
Arborii AVL sunt un model de arbori binari de cautare echilibrat dupa inaltime, care se reechilibreaza dupa fiecare inserare sau stergere.
La inserare se adauga nodul exact ca intr-un arbore binar de cautare, iar dupa se verifica factorul de balansare si se incepe sau nu balansarea lui (aceasta balansare se face cu rotatii duble).
Proprietatea de baza a unui arbore AVL este aceea ca, diferenta in modul dintre inaltimea nodului stang si cea a nodului drept este mai mica sau egala cu 1.

Structura elementelor unui arbore echilibrat poate fi reprezentata astfel:

```
class AVL
{
public:
    int key;
    AVL *left;
    AVL *right;
    int height;
}
```

- key = valorea nodului
- left/right = pointeri catre copii (stanga, respectiv dreapta)
- height = inaltimea

2 Analiza Timp a programului

2.1 Inaltime

Putem arata ca un arbore AVL cu n noduri are inaltimea O(log n):
Fie N_h minimul de noduri care pot forma un arbore AVL de inaltime h.
Daca stim N_{h-1} si N_{h-2}, putem determina N_h. Stiind ca acest arbore cu N_h noduri are inaltimea h, radacina are un fiu cu inaltimea h-1, adica un subarbore cu N_{h-1} noduri. Cunoscand proprietatile unui arbore AVL, daca un

fiu are inaltimea h-1, inaltimea minima a celuilalt fiu este h-2. Avand un sub-arbore drept cu N_{h-1} noduri si un sub-arbore stang cu N_{h-2} noduri, am construit un arbore AVL cu inaltimea h si cu numarul minim de noduri necesare pentru a alcatui acest arbore.
Acest arbore are in total $N_{h-1} + N_{h-2} + 1$ (radacina) noduri. Folosind aceasta formula putem deduce:

$$N_h = N_{h-1} + N_{h-2} + 1$$

$$N_{h-1} = N_{h-2} + N_{h-3} + 1$$

$$N_h = (N_{h-2} + N_{h-3} + 1) + N_{h-2} + 1$$

$$N_h > 2N_{h-2}$$

$$N_h > 2^{h/2}$$

$$logN_h > log2^{h/2}$$

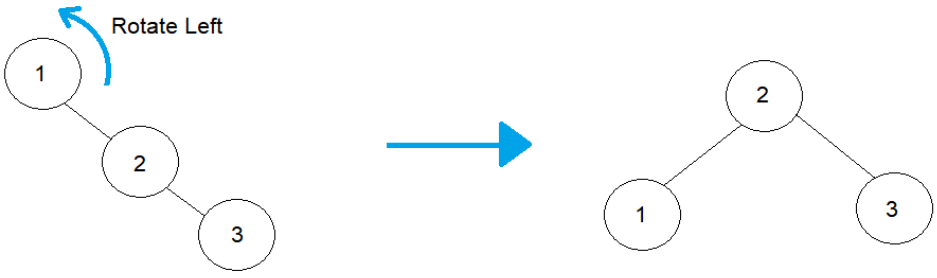
$$2logN_h > h$$

$$h = O(logN_h)$$

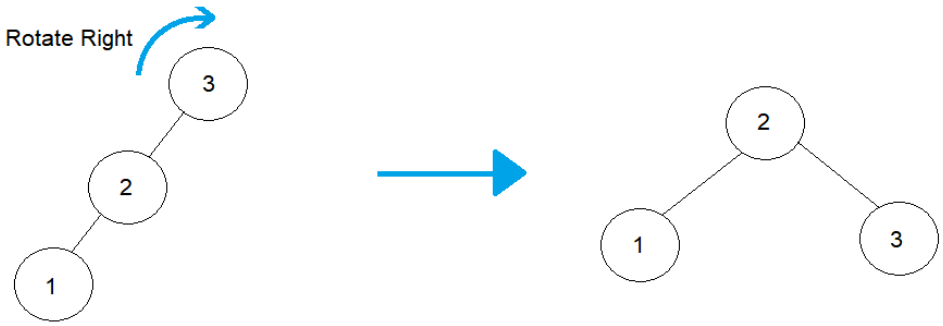
2.2 Rotatii

Pentru a mentine arborele balansat, avem nevoie de doua operatii: o rotatie la stanga si una la dreapta. Rotatiile sunt simple rearanjari a nodurilor pentru a interschimba inaltimile in timp ce pastram ordinea elementelor sale.
O rotatie necesita o reatribuire a fiului drept, stang si al parintelui la cateva noduri, dar nimic mai mult de atat. Rotatiile sunt operatii de timp O(1). Avem 4 tipuri de rotatii:

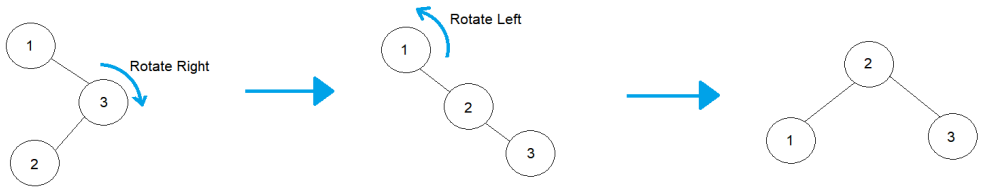
- Dreapta - Dreapta (daca factorul de balans este negativ si factorul de balans al nodului drept este tot negativ)



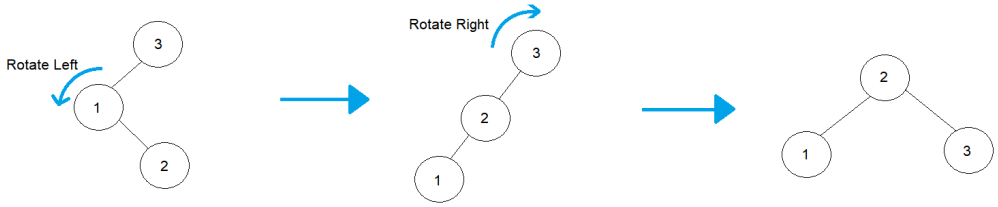
- Stanga - Stanga (daca factorul de balans este pozitiv si factorul de balans al nodului stang este tot pozitiv



- Dreapta - Stanga (daca factorul de balans este negativ si factorul de balans al nodului drept este pozitiv)



- Stanga - Dreapta (daca factorul de balans este pozitiv si factorul de balans al nodului stang este negativ)



2.3 Inserare

Inserarea unui nod intr-un AVL se realizeaza in doua etape:

1. Se adauga nodul intr-un arbore binar de cautare (se cauta pozitia de inserare pe baza comparatiei cheii nodului si cheia nodului stang si drept).

2. Se va actualiza inaltimea fiecarui nod de la pozitia inserarii pana la radacina, iar daca este necesar se aplica operatii de rotire (pentru a mentine proprietatea AVL - ului)

Inserarile intr-un arbore AVL sunt inserarile din arborele binar de cautare, la care se adauga cel mult doua rotatii. Stiind ca inserarile intr-un arbore binar de cautare au complexitatea O(h), rotatiile au O(1) si ca inaltimea arborelui AVL este h=O(log n), concluzionam ca inserarile in AVL necesita timpul O(log n).

2.4 Stergere

Operatia de stergere pentru o cheie dintr-un arbore AVL se aseamana cu stergerea unei chei dintr-un arbore binar de cautare. Aceasta stergere se poate rezuma in 3 etape

1. Cautarea cheii pe care dorim sa o stergem.

2. Stergerea nodului cu cheia respectiva. Daca nodul are 2 succesori, il vom inlocui cu cel mai mare nod din subarborele stang(sau cel mai mic nod din subarborele drept). Altfel il vom inlocui cu unul dintre succesorii sai nenuli, iar in cazul in care acesta este frunza, se va inlocui cu NULL.

3. Reactualizarea inaltimilor si realizarea rotatiilor necesare.

Stiind ca stergerile intr-un arbore binar de cautare au complexitatea O(h), rotatiile au O(1) si ca inaltimea arborelui AVL este h=O(log n), ajungem la concluzia ca stergerile in AVL necesita timpul O(log n)

2.5 Valoarea minima/maxima

Pentru aceste operatii, algoritmul se aseamana.
Astfel, pentru cea mai mica valoare, parcurgem recursiv de la radacina la stanga, pana cand valoarea left este NULL, iar pentru cea mai mare parcurgem recursiv de la radacina la dreapta, pana cand valoarea right este NULL
Asadar, timpul necesar este timpul parcurgerii inaltimii AVL -ului, mai exact O(log n)

2.6 Succesor/predecesor cheie

Aceasta presupun aflare celei mai mari chei y, mai mica decat cheia x , respectiv cele mai mici chei y, mai mare decat cheia x. In cazul unui arbore binar de cautare (evident si in arborele AVL) succesorul cheii unui nod este nodul cu cea mai mica valoare din sub-arborele dreapt al cheii. Daca sub-arborele drept nu exista, atunci elementul pe care il cautam se afla printre stramosii acestuia. Pentru a eficientiza cautarea succesorului, cand incepem parcurgerea de la radacina, chiar daca nu stim daca cheia are sau nu sub-arbore drept, vom salva si compara nodul cu cea mai mica valoare mai mare decat cheia data, asta pentru a nu parcurge inaltimea arborelui de doua ori. Daca nu exista un astfel de nod, cheia data nu are succesor.

Pentru predecesor operatiile se afla oarecum in oglinda, adica se va cauta nodul cu cheia cea mai mare din sub-arborele stang, iar daca nu exista se va lua nodul cu cea mai mica cheie din stramosii acestuia.

In cel mai rau caz, parcurgem intreaga inaltime a arborelui, adica avem o complexitate de timp O(log n).

2.7 Al k-lea element

Pentru a afla al k-lea element, folosim traversarea in ordine (inorder traversal). Ideea este ca in timp ce traversam arborele, numaram nodurile, iar cand ajungem la elementul al k-lea se va returna cheia acestuia.

In cel mai rau caz, parcurgem intreaga lungime a arborelui, adica O(n), unde n este numarul de noduri.

2.8 Cardinal

Pentru afla cardinalul, am introdus o variabila care este incrementata in fiecare inserare si decrementata in fiecare stergere, iar functia aceasta pur si simplu returneaza valoarea variabilei. Complexitatea functiei este asadar, O(1).

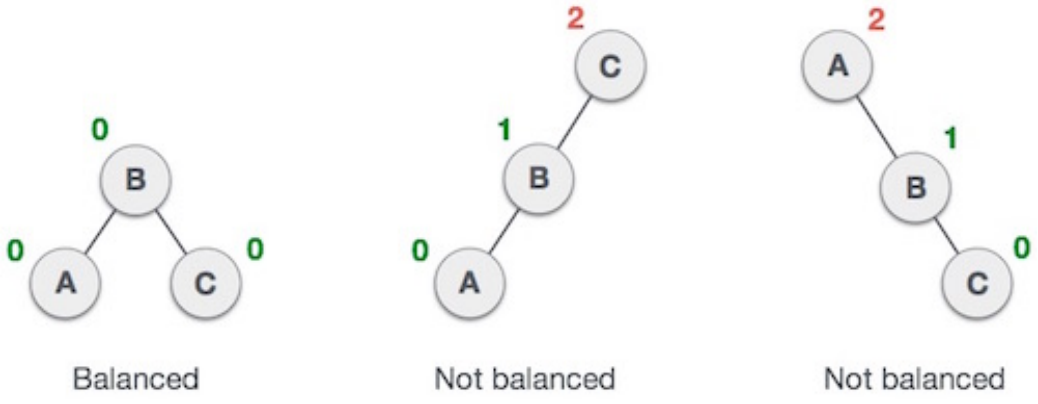
2.9 Este In

Cautarea este la fel ca intr-un arbore binar(asemanatoare cu cautarea binara intr-un array):
Pornim de la radacina si verificam daca cheia data este mai mare, mai mica sau egala decat aceasta. Daca este egala, inseamna ca am gasit elementul, daca este mai mica, apelam recursiv pentru nodul stang, iar daca este mai mare apelam recursiv pentru nodul drept. Daca in urma acestei parcurgeri nu gasim nodul, inseamna ca acesta nu exista.

In cel mai rau caz, parcurgem intreaga inaltime a arborelui AVL, adica O(log n).

3 Motivatia Structurii

Principalul motiv pentru care am ales arborele AVL este diferenta de complexitate fata de un arbore binar de cautare simplu (adusa de factorul de balansare) cand vine vorba de un input ordonat crescator/descrescator.



4 Avantaje si dezavantaje

Arborii AVL au atat avantaje cat si dezavantaje. Unul dintre principalele avantaje ar fi eficienta inalta atunci cand vine vorba de un numar mare de date de intrare care implica o multime de insertii, lucru care poate fi facut in complexitatea O(log n).
Fata de un arbore de cautare normal, AVL-ul rezolva cazul marginal al datelor de intrare ordonate crescator/descrescator.
Operatiile de cautare pot si ele sa fie facute cu aceeasi complexitate.
Cu toate acestea, avem si cateva dezavantaje printre care

- Codul mult mai complex fata de codul unui arbore binar de cautare obisnuit, deoarece avem de tratat multe cazuri marginale.
- Deoarece inaltimea trebuie sa fie mentinuta intr-un arbore AVL, se produc rotatii frecvente care, vor necesita mai multe resurse pentru efectuarea lor in cazul unui numar mare de date de intrare.
- Operatia de stergere de asemenea necesita multe resurse, deoarece implica multe schimbari de pointeri, in cel mai rau caz rotatiile fiind efectuate pana la radacina.

5 Testare

Am generat numerele cu ajutorul unui site (https://numbergenerator.org/) dupa ce ne-am umplut memoria cu un program in python.
Am ales sa testam urmatoarele multimi, pe 3 calculatoare diferite. Mai jos, vom enumera o medie a timpurilor de rulare.

1. Noua numere random.
2. 100 de mii de numere in ordine aleatorie.
3. 100 de mii de numere in ordine aleatorie intre -100 de mii si +100 de mii
4. Numere de la -1.5 milioane la 1 milion ordonate crescator.
5. Numerele de la 1 la 5 milioane ordonate crescator.

Testele au fost realizate folosind functia "clock", din libraria bits/stdc++.h

.

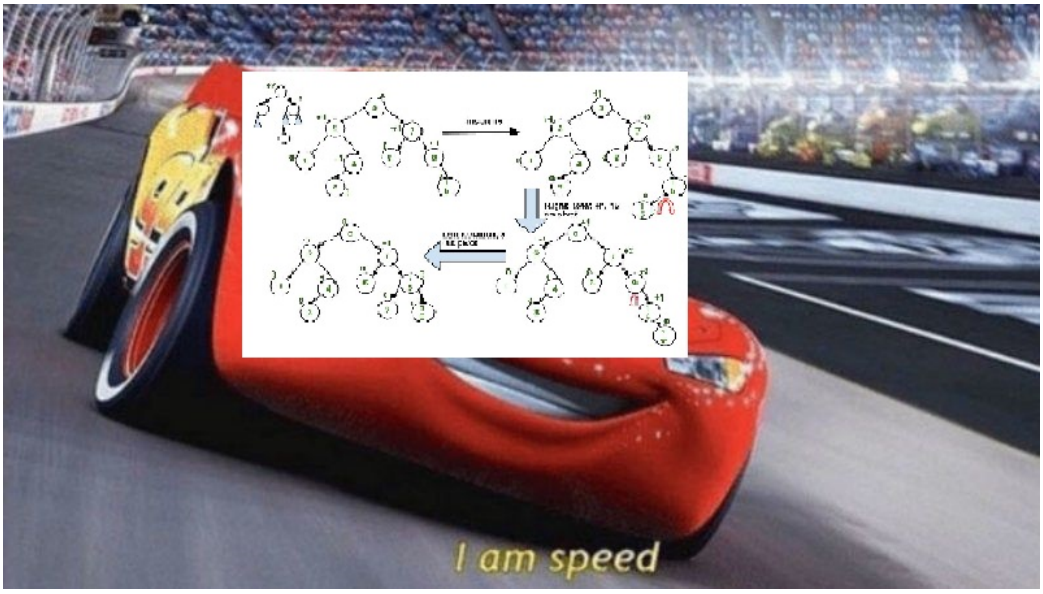
```
clock_t start, end;

start = clock();
//functia pentru care calculam timpul de rulare
end = clock();

double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
cout << "\nTime taken by program is : " << fixed
     << time_taken << setprecision( n: 5);
cout << " sec " << endl;
```

In unele cazuri, functia analizata rula doar cateva zeci de instructiuni, si deoarece procesorul executa un numar foarte mare de instructiuni pe secunda, cronometrul returna 0 (zero) .

Timpul de rulare

Operatia	Caz 1	Caz 2	Caz 3	Caz 4	Caz 5
Inserare	0 sec	≈0.073 sec	≈0.095 sec	≈2.2 sec	≈5.4 sec
Stergere	0 sec	≈0.073 sec	≈0.095 sec	≈2.2 sec	≈5.4 sec
Min/Max	0 sec	0 sec	0 sec	0 sec	0 sec
Suc/Pre	0 sec	≈0.0015 sec	≈0.0015 sec	≈0.0015 sec	≈0.0015 sec
k element	0 sec	≈0.002 sec	≈0.0045 sec	≈0.008 sec	≈0.028 sec
Cardinal	0 sec	0 sec	0 sec	0 sec	0 sec
Este In	0 sec	≈0.0015 sec	≈0.0015 sec	≈0.0015 sec	≈0.0015 sec

5.1 Concluzie



6 Sales Pitch

Perfectly balanced, as all things should be!!