Cozi de prioritate
AVL tree și Heap

Grecu Andrei-George

Grupa 325CA, Facultatea de Automatica și Calculatoare
Universitatea Politehnica București
andrei.g.grecu@gmail.com

Abstract. Aceasta lucrare își propune analizarea a doua cozi de prioritate prin evidențierea avantajelor, cat și dezavantajelor, folosirii fiecăreia dintre acestea.

Keywords. Cozi de prioritate – AVL tree - Heap

1. Introducere

1.1. Descrierea problemei rezolvate

Structurile sunt un mod logic de organizare și stocare al informațiilor, a valorilor de diferite tipuri, în așa fel încât sa fie mai eficiente de accesat și procesat.

Pe zi ce trece, necesitățile oamenilor de a folosi calculatorul devin mult mai complexe, și pentru acestea trebuie sa oferim cele mai rapide soluții și cele mai eficiente, din punct de vedere al timpului și spațiului utilizat.

Voi implementa și analiza, cu ajutorul celor doua structuri, inserarea, căutarea și ștergerea unui element în AVL tree și Heap, pentru a demonstra eficienta utilizării lor.

Pentru un AVL ne-am putea gândi la o aplicație practica pentru trenuri, întrucât foarte rar sunt adăugate rute noi, dar căutarea este frecventa și se lucrează cu multe date (rute), care e un avantaj al acestei structuri. AVL-ul necesita

În schimb, gândindu-ne la un heap, acesta e utilizat la implementarea unui heap, care sta la baza a mai multor aplicații. Un exemplu ar fi algoritmul de triangulare ROAM care calculează o triangulație dintr-o schimbare dinamica a unui teren. Algoritmul folosește două cozi prioritare, una pentru triunghiuri care pot fi împărțite și alta pentru triunghiuri care pot fi îmbinate. În fiecare pas, triunghiul din coada împărțită cu cea mai mare prioritate este împărțit sau triunghiul din coada de îmbinare cu prioritatea cea mai mică este combinat cu vecinii săi.

1.2. Implementarea și specificarea soluției

Am ales cele doua structuri de date pentru a determina diferența de complexitate în cazul cel mai defavorabil pe utilizarea lor în operațiile specifice: inserare, eliminare/extragere, căutare.

De exemplu, **Heap**-urile sunt proiectate sa optimizeze găsirea elementului minim sau maxim. Ele mai sunt folosite la algoritmul de sortare specific, Heap-Sort, care nu folosește memorie suplimentara la sortarea unui vector.

Arborele AVL este un arbore cu auto-echilibrare la inserarea și la ștergerea unui nod, care util atunci când sunt puține inserții și ștergeri. El este folosit și când este nevoie de căutare rapida, limitat însă de costul rebalansării.

Voi implementa ambele structuri în C++ și voi utiliza librarii externe pentru a monitoriza timpul de rulare pe ambele cazuri, ținând cont de timpii de inserție, căutare și extragere a elementelor din structuri, pe exemplele construite.

1.3. Criterii de evaluare

Pentru evaluare, as considera doua etape:

- Testarea corectitudinii soluției
- Testarea performantei soluției

Pentru testarea corectitudinii, voi construi teste mici (pana la 10^3 elemente) ce pot fi verificate manual. Acestea ar trebui sa acopere (în mod ideal) toate cazurile ce pot apărea în problema, verificând astfel implementarea corecta a structurilor.

Pentru testarea performantei, voi construi teste mari (pana la 10^7 elemente), ce vor intra pe cele mai defavorabile cazuri ale implementării, utilizând operații de insert consecutive. Vor exista atât teste cu arbori mari și puține modificări ale structurii din timpul inserării elementelor, cat și teste care vor utiliza proprietățile structurilor permanent (De exemplu multe rotiri ale structurii arborelui binar).

2. Prezentarea soluțiilor

2.1. Descrierea modului în care funcționează algoritmii aleși

2.1.1. AVL tree

Arborele AVL este un arbore binar de căutare, echilibrat după înălțime, ce se reechilibrează la fiecare operație de adăugare sau ștergere a unui element. Diferența de înălțime dintre doi subarbori ai oricărui nod este de maxim 1, menținând o complexitate de ***O*(log*n*)** (Exemplu arbore AVL - Fig. 1).

La inserarea și eliminarea unui nou nod, trebuie menținută proprietatea de arbore binar de căutare. După fiecare modificare, se verifica factorul de balansare și eventual balansarea lui, cu ajutorul rotațiilor sau rotațiilor duble (**R, RL, L, LR**).

Se definesc doua variabile, balance și height, ambele pentru un nod. La aplicarea rebalansării, se face calcularea lui balance, diferența de înălțime (height) dintre nodul drept și stâng (formula în Fig. 2).

Dacă balance este 2

- Dacă copilul stâng al nodului din stânga este mai mare, ca valoare, decât copilul drept al nodului stâng, se face o rotație la dreapta R.
- Altfel, se face o rotație la stânga și după la dreapta RL.

Dacă balance este -2

- Dacă copilul drept al nodului din dreapta este mai mare, ca valoare, decât copilul drept al nodului stâng, se face o rotație la stânga L.
- Altfel, se face o rotație la dreapta și după la stânga LR.

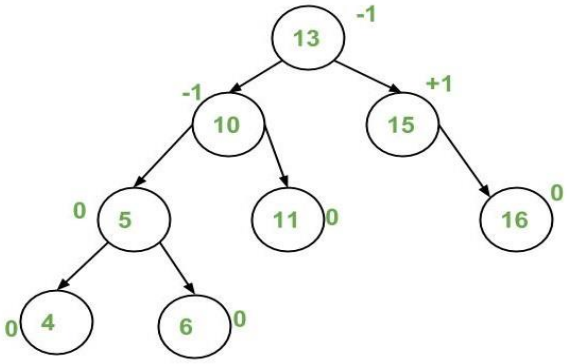


Fig. 1 - Exemplu arbore AVL

```
n->balance = height(n->right) - height(n->left);
```

Fig. 2 - Formula

Pentru a-l compara cu un Max-Heap, a trebuit implementarea unei funcții de parcurgere pentru extragerea elementului maxim din AVL, aflat pe poziția cea mai din dreapta în arbore.

2.1.2. Heap – Max

Heap-ul este un arbore binar complet, care se conforma unei anumite strategii, maxim sau minim, în funcție de nevoia programatorului.

El este o implementare maximala eficientă a unui tip abstract de date, coada de prioritate, care nu este sortată, ci doar parțial ordonată. Un heap este o structură de date utilă când este nevoie să interogăm sau eliminăm, în mod repetat, nodul cu cea mai mare/mică (în funcție de implementare) prioritate.

Max Heap-ul este heap-ul care are fiecare valoare a unui nod intern mai mare sau egala ca valorile copiilor acelui nod.

Am considerat maparea arborelui într-un vector deoarece accesarea nodurilor, părinte, copil stânga și copil dreapta al unui nod, sunt mult mai ușoare (formule pentru aflarea lor - Fig. 3). Astfel, avem pentru inserare și extragerea maximului din heap avem ***O*(log*n*)**.

```
/// returns the parent of the index
    int parent(int i) {return (i-1)/2;}

/// returns the left child of the index
    int left(int i) {return (2*i + 1);}

/// returns the right child of the index
    int right(int i) {return (2*i + 2);}
```

Fig. 3

2.2. Analiza complexității soluțiilor

În cazul implementării unui arbore binar de căutare, în general, complexitatea în cazul cel mai defavorabil este O(n).

În particular, când vorbim de arborii AVL, complexitatea, în cazul cel mai rău, este de O(log*n*). Cazul cel mai defavorabil este atunci când avem de făcut o inserție sau o ștergere și trebuie rebalansat arborele pentru a-și respecta prioritatea. Pentru a compara cu proprietatea heap-ului de a extrage maximul (în cazul meu), am implementat și o funcție de extras maximul din arbore cu complexitatea O(*n*).(Fig. 4)

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

Fig. 4

Când aducem în discuție Heap-ul, complexitățile sunt similare, însă pe cazul average al inserției, acesta excelează împotriva arborelui AVL, complexitatea fiind O(1). Heap-ul mai are un avantaj,

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(n)	O(n)
Insert	O(1)	O(log n)
Delete	O(log n)	O(log n)

el fiind implementat pentru a menține maximul în vârful arborelui, complexitatea fiind **O(1)**, comparativ cu **O(n)** la AVL.

Fig. 5

2.3. Prezentarea principalelor avantaje și dezavantaje

Pro:

Principalul avantaj al folosirii unui arbore binar este faptul ca ne permit sa ținem într-o anumita ordine (putând realiza de asemenea parcurgeri într-o ordine prin care sa obținem date sortate crescător sau descrescător). Daca este nevoie, acestea servesc și la obținerea celui mai mic sau celui mai mare element introdus în structura. Un alt avantaj al arborilor de căutare este complexitatea (în raport cu memoria totala utilizata de structura), care este un lucru foarte interesant și foarte bun: acestea nu folosesc mai mult spațiu decât au nevoie.

Contra:

Ca și dezavantaje, la AVL, cum înălțimea trebuie menținuta și recalculata pentru rotațiile frecvente ale arborelui, acestea vor creste pointer overhead-ul care tinde la un spațiu suplimentar, cert uneori neglijabil, alteori nu, pentru inputuri foarte mari. În plus arborii AVL ștergerea este o operație costisitoare întrucât necesita rotiri multiple și schimbări la nivelul pointerilor.

Heap-urile nu oferă capacitatea de căutare și, în general, nu oferă alte operații ca floor/ceiling sau traversarea în ordine. De asemenea, nu poți elimina elementele arbitrare dintr-un heap, doar maximul sau minimul.

3. Evaluare

3.1. Descrierea modalității de construire a setului de teste

Am realizat un generator de teste care creează teste cu valori random (pseudo-random) pentru a le insera în structuri. Pentru a confirma corectitudinea algoritmilor și a operațiilor, efectuez o serie de operații.

Pentru fiecare test, se iau valorile de input, testele fiind de diferite mărimi, pentru a le da insert în structura corespunzătoare. În fiecare test, la început exista numărul de operații de căutare a maximului, respectiv ștergerea unui element (al doilea fiind pentru numărul de elemente). Conform acestui număr, se aplica asupra structurii corespunzătoare aceste operații, se calculează timpii și se compara.

Primul tester (algo_test.cpp și algo2_test.cpp, în care am adăugat funcții din chrono pentru a verifica timpii), l-am folosit pentru a demonstra performanta algoritmilor. Aplicând testele pe fiecare structura, am urmărit timpii și i-am comparat.

Am modificat puțin un testerele deja existente(doar sa printeze arborele după inserări și după fiecare ștergere și am comparat cu algoritmi implementați de la [1] și [2]), verificând corectitudinea algoritmilor implementați, aplicând pe testele mici, de pana la 10^3, adăugare, căutare, ștergere. Cu ajutorul acelor algoritmi de AVL, respectiv Max Heap (preluați de la [1] și [2]), am făcut o comparație dintre output-urile algoritmilor implementați de mine și output-urile lor.

3.2. Specificațiile sistemului de calcul

Testele au fost rulate pe un sistem având următoarele specificații:

Operating System: Windows 10 Pro
Processor: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 2.81GHz, 4 Core(s)
Memory: 16384MB RAM
Available OS Memory: 8130 MB
Page File: 10969MB used, 7727MB available
DirectX Version: DirectX12
DxDiag Version: 10.00.18362.0387 64bit Unicode

3.3. Ilustrarea, folosind grafice/tabele, a rezultatelor evaluării soluțiilor pe setul de teste

Număr operații	Timp AVL (sec)	Timp Heap (sec)
10	0.00000	0.00000
100	0.00003	0.00001
1000	0.00037	0.00005
10^4	0.00530	0.00050
10^5	0.08973	0.00470
10^6	0.17074	0.04320
10^6	0.17363	0.04338
15*10^5	0.28590	0.08290

Fig. 8 – AVL vs Heap inserare

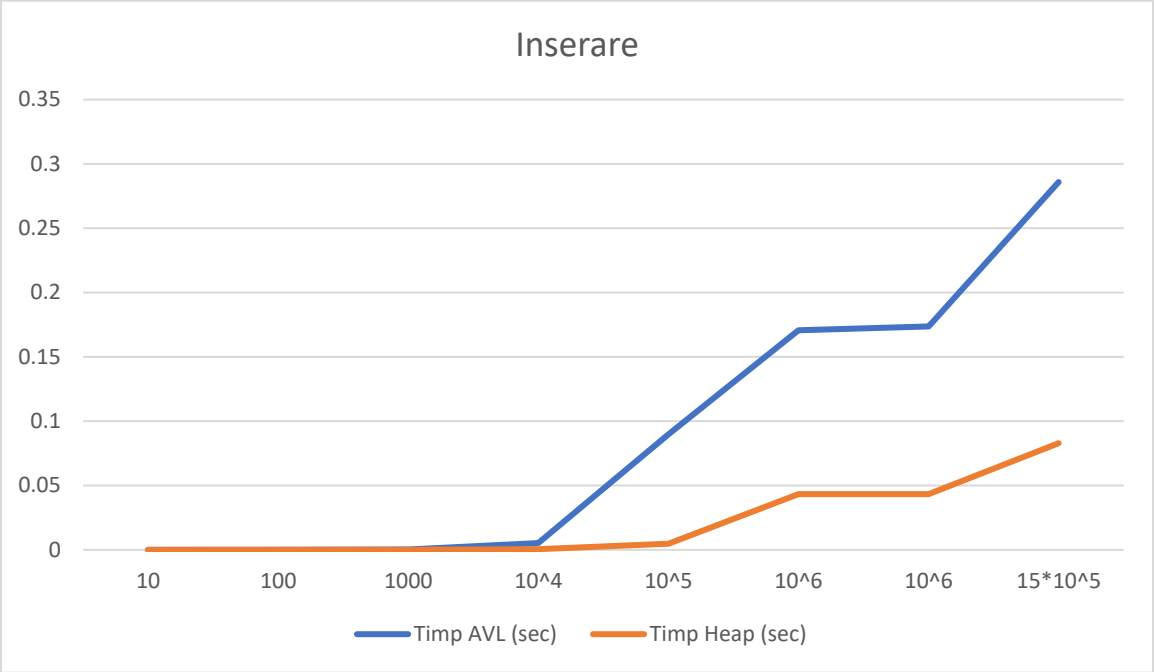
Număr operații	Timp AVL (sec)	Timp Heap (sec)
3 – test1	0.00000	0.00000
25 – test2	0.00000	0.00000
250 – test3	0.00004	0.00000
500 – test4	0.00007	0.00000
5000 – test8	0.00081	0.00000
50000 – test7	0.00107	0.00001

Fig. 9 – AVL vs Heap aflare maxim

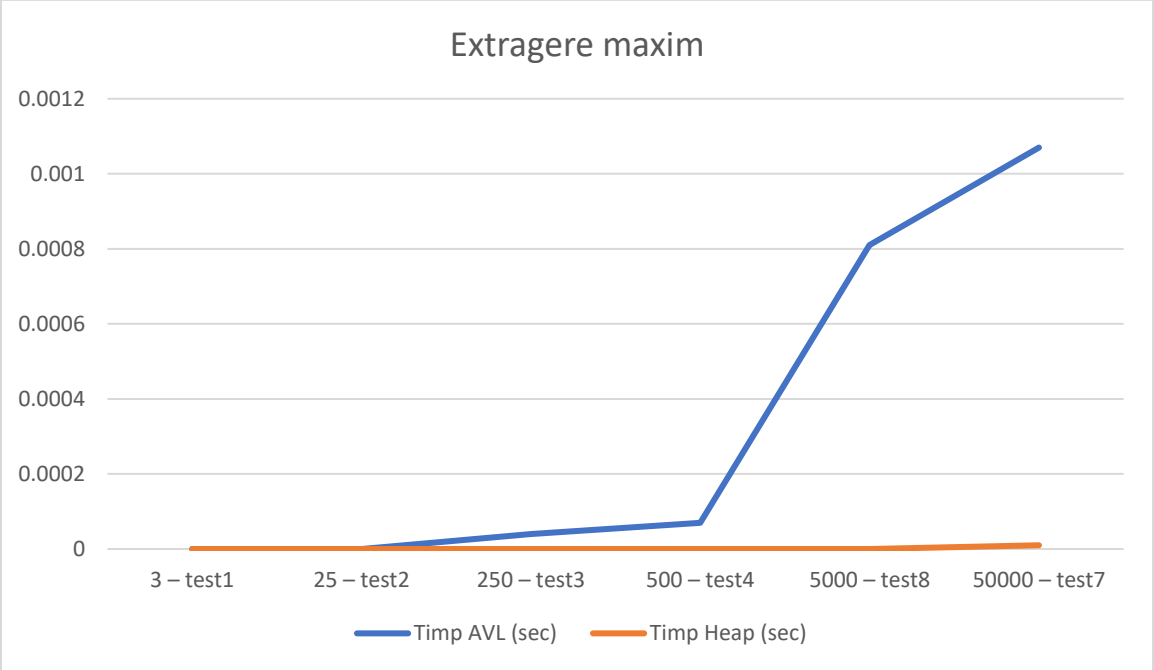
Număr operații	Timp AVL (sec)	Timp Heap (sec)
3 – test1	0.00000	0.00000
25 – test2	0.00000	0.00000
250 – test3	0.00006	0.00098
500 – test4	0.00099	0.00175
5000 – test8	0.01718	0.03566
50000 – test7	0.17317	0.41861

Fig. 10 – AVL vs Heap ștergere maxim

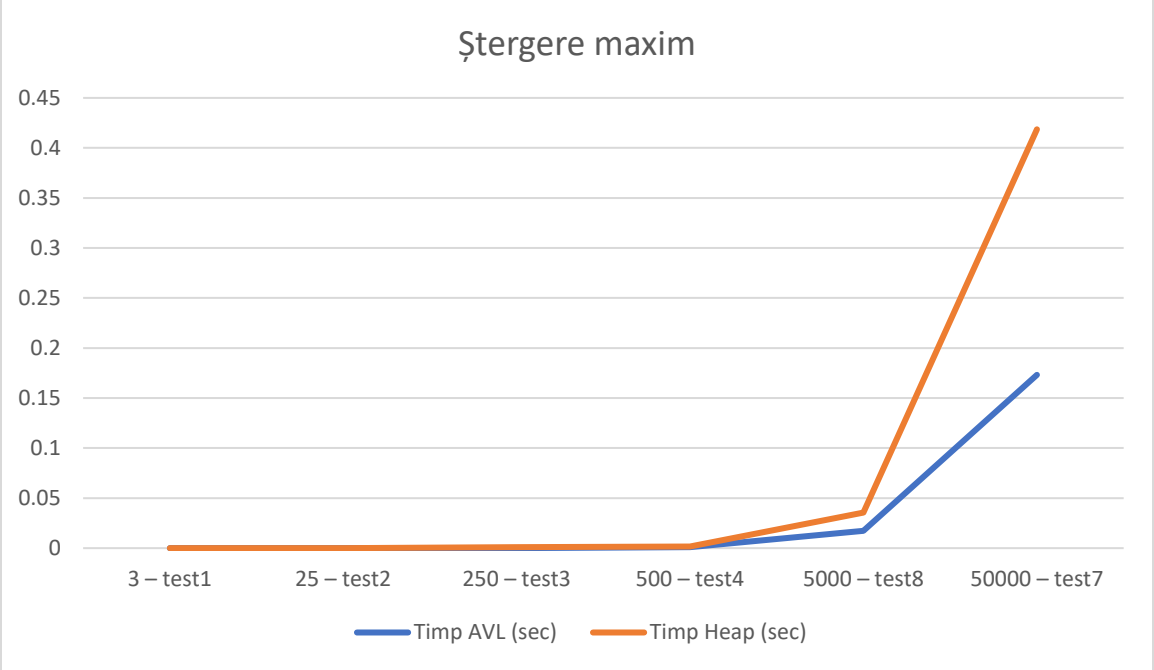
În calcularea timpilor de mai sus am eliminat testul7.in deoarece acesta avea date de intrare foarte mari și ieșeau din graficele următoare. La fel am procedat pentru testele ce aveau numărul de operații de extragere a maximului și ștergere identic, deoarece timpii nu variau foarte mult.



Grafic 1. Inserare în AVL vs. Heap



Grafic 2. Extragere max AVL vs. Heap



Grafic 3. Ștergere maxim AVL vs. Heap

3.4. Prezentarea valorilor obținuta pe teste

După cum se poate observa, în cazul arborilor AVL, inserările (și în general ștergerile) în arbore pot fi foarte costisitoare, motiv pentru care este de preferat folosirea lui doar în cazul căutărilor intensive de elemente.

Având în vedere ca facem o comparație mai amănunțita în legătura cu aflarea maximului din structura, cum heap-ul are O(1) la extragerea acestuia, e clar ca, în ciuda modificărilor aduse arborelui, e mult mai rapid ca rotațiile costisitoare ale AVL-ului.

Însă cel din urmă este mai rapid pe ștergerea maximului din structura, datorită rotațiilor în număr redus, pe subarbori mici, față de modificările heap-ului la ștergerea elementului din vârf (când se aplica heapify, se poate modifica toată structura). Deși acesta nu acesta este motivul pentru care se folosește un heap, am luat și acest caz de comparație in considerare.

4. Concluzii

Având în vedere ca AVL-ul bate la capitolul căutare, acesta este mult mai utilizat în aplicații corelate cu acest avantaj, cum ar fi orice site ce dorește sa colecteze date rapid sau amintim ideea aplicației „Mersul trenurilor” discutata în introducere, în care datele nu se schimbau, dar interogările erau multiple.

Rămânând în aceasta sfera, ducând legătura către rețelistică, pe de alta parte, heap-ul poate fi folosit pentru o aplicație de tip bandwidth management. O implementare a unor cozi de prioritate, cu ajutorul heap-ului, pot fi utilizate pentru a gestiona resurse limitate ca lățimea de banda pe o linie de transmisie de la un router la rețea. În cazul unei cozi de trafic de ieșire, din cauza limitărilor, toate celelalte cozi pot fi oprite pentru a trimite date numai de pe coada cu prioritate cea mai mare.

5. Referințe

1.Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliord Stein: INTRODUCTION TO ALGORITHMS, Second Edition

2.MITOpenCourseware https://ocw.mit.edu/courses/electrical-engineering-and-computer-science

3.OCW: DATA STRUCTURES https://ocw.cs.pub.ro/courses/sd-ca/laboratoare/lab-09
& https://ocw.cs.pub.ro/courses/sd-ca/laboratoare/lab-10

4.GeeksForGeeks https://www.geeksforgeeks.org/time-complexity-of-building-a-heap/
&https://www.geeksforgeeks.org/complexity-different-operations-binary-tree-binary-search-tree-avl-tree/

5. http://www.bradapp.com/ftp/src/libs/C++/AvlTrees.html

[1]: https://wkdtjsgur100.github.io/avl-tree/
[2]: https://codepumpkin.com/heap/