Tema 1
Analiza Algoritmilor
Structuri de Date
Mulțimi

Teodora Stroe

321CA
Facultatea de Automatică și Calculatoare
Universitatea Politehnica din București
teodora.stroe2210@stud.acs.upb.ro

1 Introducere

1.1 Descrierea problemei rezolvate

În informatică și inginerie, un set (o mulțime) este un tip abstract de date care poate stoca valori unice, fiind o implementare computerizată a conceptului matematic al unei mulțimi finite.

Unele structuri de date de tip set sunt *statice*, neputând fi modificate după ce au fost construite. Acestea permit numai operații de interogare asupra elementelor lor - cum ar fi verificarea dacă o anumită valoare este în set sau enumerarea valorilor într-o ordine arbitrară. Alte variante, numite mulțimi *dinamice*, permit și inserarea și ștergerea elementelor din mulțime.

De obicei, structura internă a elementelor stocate în aceste structuri de date se află sub formă de perechi *cheie-valoare*. În multiple implementări ale seturilor, care utilizează diferite structuri de date, valoarea unui element este în același timp și cheia acestuia, care îl identifică în mod unic. Acest lucru se poate realiza fie prin încărcarea acesteia cu același conținut ca al cheii, fie prin încărcarea câmpului destinat valorii cu un conținut fictiv, care este ignorat.

În acest articol se analizează două tipuri de structuri de date ce sunt utilizate pentru implementarea mulțimilor în diferite limbaje de programare.

1.2 Exemple de aplicații practice pentru problema aleasă

O generalizare a noțiunii de set este aceea de *multiset*, care este similară cu un set, dar permite valori repetate, "egale". Acest termen este folosit în două sensuri distincte: fie valorile egale sunt considerate identice și sunt pur și simplu numărate, fie valorile egale sunt considerate echivalente și sunt stocate ca elemente distincte.

Un exemplu în acest sens îl poate reprezenta următoarea situație: considerând o listă de mașini, caracterizate de marcă (ex: BMW, Volkswagen, Ford) și anul de fabricație, ar fi posibilă construirea unui multiset de ani, care să calculeze

numărul mașinilor fabricate într-un anumit an. Alternativ, se poate construi un multiset de mașini, în care două vehicule sunt considerate echivalente dacă anii lor de fabricație sunt aceiași. În acest caz, pot exista mașini diferite, cu mărci diferite, caz în care fiecare pereche (marcă, an de fabricație) trebuie să fie stocată separat.

În bazele de date relaționale, înregistrările unui tabel pot reprezenta un set sau un multiset, în funcție de eventualele constrângeri de unicitate.

1.3 Specificarea soluțiilor alese

Două dintre metodele principale de implementare ale mulțimilor sunt prin [1]:

1. tabele de dispersie (C++11: unordered_set; STL[1]: hash_set; Java: HashSet)
2. arbori binari de căutare echilibrați (C++: set; Java: TreeSet)

În cadrul acestui articol se realizează o analiză comparativă a acestor structuri de date, și anume tabele de dispersie (hashtable-uri) versus arbori binari de căutare echilibrați, în cazul acestora din urmă efectuându-se o discuție pe două cazuri particulare: AVL și Treap-uri.

Din punct de vedere al implementării, pentru fiecare dintre cele două structuri, se vor pune în discuție operațiile de inserare, ștergere, respectiv modificare a unui element. În ceea ce privește operația de modificare, aceasta va reprezenta o compunere a operațiilor de ștergere și inserare. Această abordare este aleasă pentru a imita cât mai bine comportamentul set-urilor actuale din Java.

De asemenea, datele reținute în câmpul de valoare vor avea același conținut cu cheia, rezultând atât o uniformizare a formatului datelor de intrare, cât și o simulare a actualelor implementări.

Nu în ultimul rând, în implementare hashtable-ului, pentru rezolvarea eventualelor coliziuni, se va utiliza tehnica de **înlănțuire**.

1.4 Criterii de evaluare

În realizarea fișierelor utilizate pentru evaluarea algoritmilor menționați anterior, se vor lua în considerare următoarele aspecte:

- tipul de date conținute în test;
- dimensiunea datelor testului;
- rolul testului.

După criteriul dimensiunii și al tipului de date, se vor realiza două categorii de fișiere de intrare, cu date de tip întreg (int) și șiruri de caractere (char), de mici dimensiuni pentru a testa corectitudinea, și de dimensiuni mari, pentru a testa complexitatea și pentru a genera o analiză comparativă între cele două tipuri de structuri.

De asemenea, vor fi create teste pentru verificarea cazurilor limită și a situațiilor speciale, în funcție de structura de date analizată (ex: hashtable - coliziuni, modificarea/ștergerea unei valori inexistente).

[1] The SGI Standard Template Library

2 Prezentarea soluțiilor

2.1 Descrierea modului de funcționare al algoritmilor

2.1.1 Dicționar

O implementare frecvent întânită a unui dicționar este prin utilizarea unui hashtable. Acesta este o structură de date optimizată pentru funcția de căutare, performanță atinsă prin transformarea cheii într-un hash prin utilizarea unei **funcții de hash**.

O funcție hash poate fi utilizată pentru a mapa un set de date de o dimensiune arbitrară la un set de date de o dimensiune fixă, care se încadrează în hashtable. Pentru a obține un mecanism de hashing bun, este important ca funcția de hash să respecte următoarele cerințe de bază [2]:

1. *să fie ușor de calculat*;
2. *să asigure distribuție uniformă*;
3. *să minimizeze numărul coliziunilor.*

Din ultima proprietate rezultă faptul că, pentru a menține performanța unui hashtable, este importantă **gestiunea corectă a coliziunilor** prin diferite tehnici de rezoluție a acestora. În general, există două metode principale de abordare a acestei probleme:

- prin înlănțuire (direct chaining);
- prin adresare deschisă (linear probing/open addressing).

Abordarea prin **înlănțuire** este una dintre cele mai utilizate tehnici de rezolvare a coliziunilor, fiind, de obicei, implementată folosind liste înlănțuite. Astfel, fiecare element al hashtable-ului este o lista înlănțuită. Dacă există vreo coliziune, atunci ambele elemente sunt stocate în aceeași listă. Dacă distribuția cheilor este suficient de uniformă, atunci costul mediu al unei căutări depinde doar de numărul mediu de chei per listă. Din acest motiv, aceasta metodă rămâne eficientă chiar și atunci când numărul de intrări din tabel este mult mai mare decât numărul de bucket-uri (liste) [2].

În **adresarea deschisă**, toate elementele sunt stocate direct în matrice. Când trebuie inserată o nouă intrare, se calculează indexul hash al valorii. Dacă slotul de la indexul calculat este neocupat, atunci elementul este inserat în acesta, fiind introdus în primul slot neocupat. Această metodă o face potrivită pentru factori de încărcare mici, până la moderați. Atunci când numărul de intrări este mare, performanța acesteia scade din cauza grupării primare, o coliziunea având tendința să producă mai multe coliziuni în apropiere [3].

De asemenea, este necesară tratarea cazului când nu se cunosc de dinainte toate operațiile (sau numărul lor) care vor fi aplicate asupra structurii. Pentru a menține o performanță amortizată, un hashtable este **redimensionat dinamic**, iar elementele sunt reinserate. O abordare optimă a acestei operații este de a efectua redimensionarea treptat, dublând dimensiunea actuală, pentru a evita o eventuală fragmentare a memoriei, ce ar putea fi cauzată de dealocarea bruscă a blocurilor mari de memorie din vechiul hashtable [4]. Această problemă nu este întalnită în implementarile cu arbori deoarece pentru memorarea acestora nu sunt utilizate zone contigue de memorie ce necesită realocare.

Inserarea este rezumată în pseudocodul de mai jos:

Algorithm 1 Inserare element

```
1: procedure HT-INSEREAZA(CHEIE, VALOARE, HASHTABLE)
2:     index ← hash(cheie)
3:     for element in hashtable[index] do
4:         if element.cheie = cheie then
5:             element.valoare ← valoare
6:             return
7:     creare-element-nou(cheie, valoare)
8:     adaugare-element-in-hashtable
```

Eliminarea unui element din hashtable se face în modul următor:

Algorithm 2 Stergere element

```
1: procedure HT-STERGE(CHEIE, HASHTABLE)
2:     index ← hash(cheie)
3:     for element in hashtable[index] do
4:         if element.cheie = cheie then
5:             break
6:         if element nu indica finalul listei then
7:             sterge(element)
8:             return
```

Modificarea unui element este o compunere a operațiilor de inserare și ștergere:

Algorithm 3 Modificare element

```
1: procedure HT-MODIFICA(HT, CHEIE, CHEIE2, VALOARE, VALOARE2)
2:     if ht-contine-cheie(cheie) then
3:         ht-sterge(ht, cheie)
4:         ht-insereaza(ht, cheie2, valoare2)
```

2.1.2 Treap

Treap-ul este un tip de arbore binar de căutare echilibrat, fiind utilizat mai ales datorită implementarii relativ ușoare și a modului de operare intuitiv [5].

Fiecare nod conține două câmpuri:

- o cheie;
- o prioritate.

Cheia reține informația în arbore și în funcție de aceasta se fac operațiile de bază (inserare, ștergere, etc). **Prioritatea** este reprezentată printr-un număr generat în mod aleatoriu pe baza căruia se face echilibrarea. Arborele este aranjat incidental, respectand atât proprietatea de arbore binar de căutare, cât și cea de heap [5].

Inserarea unui nou nod poate fi sumarizată prin următorul pseudocod:

Algorithm 4 Inserare nod

```
1: procedure TREAP-INSEREAZA(NOD, CHEIE)
2:     if nod = NULL then
3:         nod ← creare-nod-nou(cheie)
4:         return
5:     if cheie < nod.cheie then
6:         treap-insereaza(nod.stanga, cheie)
7:         if nod.stanga.prioritate > nod.prioritate then
8:             rotatie-dreapta(nod)
9:     else
10:        treap-insereaza(nod.dreapta, cheie)
11:        if nod.dreapta.prioritate > nod.prioritate then
12:            rotatie-stanga(nod)
```

Ștergerea unui nod este ilustrată în pseudocodul de mai jos:

Algorithm 5 Stergere nod

```
1: procedure TREAP-STERGE(NOD, CHEIE)
2:     if nod = NULL then
3:         return
4:     if cheie < nod.cheie then
5:         treap-sterge(nod.stanga, cheie)
6:     else if cheie > nod.cheie then
7:         treap-sterge(nod.dreapta, cheie)
8:     else if nod.stanga = NULL si nod.dreapta = NULL then
9:         free(nod)
10:        nod ← NULL
11:    else
12:        if nod.stanga.prioritate > nod.dreapta.prioritate then
13:            rotatie-dreapta(nod)
14:        else
15:            rotatie-stanga(nod)
16:        treap-sterge(nod, cheie)
```

Modificarea unui nod constituie o compunere a operațiilor de inserare și ștergere:

Algorithm 6 Modificare nod

```
1: procedure TREAP-MODIFICA(NOD, CHEIE, NOD-NOU, CHEIE-NOUA)
2:     if treap-contine-cheie(cheie) then
3:         treap-sterge(nod, cheie)
4:         treap-insereaza(nod-nou, cheie-noua)
```

2.1.3 AVL

AVL-ul este un tip de arbore binar de căutare echilibrat care se reechilibrează după fiecare operație de inserare sau ștergere.

Fiecare nod conține următoarele câmpuri:

- **data**: conținutul propriu zis al nodului;
- **înălțime**: înălțimea subarborelui care are ca rădăcină nodul curent.

Invariantul acestei structuri de date este faptul că diferența între 2 subarbori ai oricărui nod este maxim 1 [6].

Inserarea unui nou nod este reprezentată mai jos sub formă de pseudocod:

Algorithm 7 Inserare nod

```
1:  procedure AVL-INSEREAZA(NOD, CHEIE)
2:      if nod = NULL then
3:          nod ← creare-nod-nou(cheie)
4:          return
5:      if nod.cheie > cheie then
6:          avl-insereaza(nod.stanga, cheie)
7:      else
8:          avl-insereaza(nod.dreapta, cheie)
9:      nod.inaltime = 1 + max(nod.stanga.inaltime, nod.dreapta.inaltime)
10:     aplicare-rotiri(nod)
```

Ștergerea unui nod existent se realizează astfel:

Algorithm 8 Stergere nod

```
1:  procedure AVL-STERGE(NOD, CHEIE)
2:      if nod = NULL then
3:          return
4:      if nod.cheie > cheie then
5:          avl-sterge(nod.stanga, cheie)
6:      else if nod.cheie < cheie then
7:          avl-sterge(nod.dreapta, cheie)
8:      else
9:          if nod are 0 succesori then
10:             free(nod)
11:             nod ← NULL
12:             return
13:         else if nod are 1 succesor then
14:             nod ← succesor(nod)
15:         else
16:             tmp ← max-element(nod.stanga)
17:             copiaza(nod, tmp)
18:             avl-sterge(nod.stanga, tmp.cheie)
19:     if nod ≠ NULL then
20:         nod.inaltime = 1 + max(nod.stanga.inaltime, nod.dreapta.inaltime)
```

21: *aplicare-rotiri(nod)*

Modificarea unui nod este o compunere a operațiilor de inserare și ștergere:

Algorithm 9 Modificare nod

```
1: procedure AVL-MODIFICA(NOD-RADACINA, CHEIE, CHEIE-NOUA)
2:     if avl-contine-cheie(cheie) then
3:         avl-sterge(nod-radacina, cheie)
4:         avl-insereaza(nod-radacina, cheie-noua)
```

2.2 Analiza complexității soluțiilor

Conform pseudocodurilor prezentate anterior, pentru fiecare algoritm în parte vom realiza o discuție în funcție de complexitatea fiecărei operații de bază: inserare, ștergere, modificare.

2.2.1 Hashtable

În cazul hashtable-ului se vor aborda cazul mediu și cel defavorabil.

Inserare și ștergere

În cazul mediu, complexitatea celor două operații este *O(1)* deoarece utilizarea unui număr corespunzator de bucket-uri va determina un număr minim de coliziuni, operațiile presupunând doar găsirea nodului prin accesarea câmpului corespunzător din matrice și execuția propriu-zisă a operației.

În cazul defavorabil, complexitatea operațiilor este *O(n)* pentru că, din cauza coliziunilor, toate elementele se pot afla în același bucket, fiind necesară parcurgerea în întregime a listei [7].

Modificare

Fiind o compunere a operațiilor menționate anterior, complexitățile operației de modificare sunt:
- cazul mediu: *O(1) + O(1) = O(1)*
- cazul defavorabil: *O(log n) + O(log n) = O(log n)*

2.2.2 Treap și AVL

În cazul arborilor se va aborda numai cazul defavorabil.

Inserare

Operația de inserare este compusă din două părți: inserarea propriu-zisă (asemanatoare procedeului dintr-un arbore binar de căutare), la care se adaugă și rotațiile. Inserarea se face inițial într-o frunză, timpul fiind deci proporțional cu înălțimea unui arbore binar de căutare – *O(log n)*. Apoi se realizează rotațiile (la intervale aleatorii pentru Treap și după fiecare operație la AVL), nodul fiind adus

spre radacină, limită superioară a numarului de rotații constituindu-l înălțimea arborelui – *O(log n)* [8].

Deci, complexitatea operației de inserție este:
O(log n) + O(log n) = O(log n)

Ștergere

Ștergerea unui nod se realizează asemănător inserării, fiind necesară parcurgerea arborelui până la găsirea nodului, eliminarea propriu-zisă și reechilibrarea arborelui.

Complexitatea operației de ștergere este *O(log n)*.

Modificare

Fiind o compunerea a celor două operații anterioare, complexitatea acesteia este *O(log n) + O(log n) = O(log n)*.

2.3 Prezentarea principalelor avantaje și dezavantaje

2.3.1 Hashtable

Hashtable-urile sunt o modalitate rapidă și eficientă de căutare, creare și ștergere a datelor conținute în acestea [9].

Sunt foarte utile în cazul în care sunt folosite pentru a memora cantități mari de date, ale căror dimensiune (mărime a volumului de date) poate fi anticipată.

Un dezavantaj constă în faptul că, în urma utilizării unui număr mic de bucket-uri poate rezulta un număr prea mare de coliziuni. Totodată, o implementare cu prea multe bucket-uri utilizată pentru un volum redus de date, este redundantă, consumul de memorie fiind excesiv. În implementarea prezentată a fost realizat un compromis între memorie și eficiență prin utilizarea unui număr de 10 liste înlănțuite.

2.3.2 Treap

Treap-ul este un arbore binar de căutare echilibrat aleatoriu. Echilibrarea după un număr arbitrar de pași reprezintă un avantaj, rezultând astfel un comportament relativ previzibil și o performanță mai buna decât a unui arbore binar de căutare.

În același timp, echilibrarea, având loc doar ocazional, în funcție de prioritățile nodurilor, nu oferă o performanță la fel de bună ca cea a altor arbori binari de căutare echilibrați.

2.3.3 AVL

AVL este un tip de arbore binar de căutare echilibrat, al cărui avantaj principal este faptul că prin utilizarea acestuia se obține un timp mai bun de căutare al cheilor, acesta fiind garantat de complexitate *O(log n)* [10].

Un arbore AVL are dezavantajul de a efectua operațiile de rearanjare după fiecare operație de inserare și ștergere, fapt ce are totuși un impact relativ mic

asupra complexității temporale, raportat la nivelul ridicat de eficiență al algoritmului.

3 Evaluare

3.1 Descrierea modalității de construire a setului de teste

În procesul de testare, au fost utilizate atât teste cu un număr mai mic de 30 de date de intrare pentru a verifica corectitudinea implementării, cât și fișiere de intrare cu un grad de complexitate ridicat pentru a testa eficiența algoritmilor. Din punct de vedere al tipului de date, există două categorii de fișiere de intrare: cu date de tip intreg și șiruri de caractere.

În realizarea fișierelor utilizate pentru validarea implementărilor algoritmilor, a fost folosit un generator de teste în Bash. Pentru o utilizare corectă a acestuia, se introduc din linia de comandă argumentele **no_commands** și **input_type**, acestea desemnând un număr natural pozitiv ce semnifică numărul de comenzi conținute în test, respectiv tipul de date al acestuia.

Pe prima linie a testului generat se află numărul n de comenzi conținute, urmat de tipul de input. Pe următoarele n linii se află câte o comandă aleasă aleator din cele 3 tipuri de operații disponibile: inserare, ștergere, modificare.

Dacă tipul de input este un număr întreg, atunci elementul fiecărei inserări va fi un număr generat aleatoriu, cu o valoare *short int*. Altfel, elementul introdus este un cuvânt în engleză, ales dintr-un dicționar de 84.000 de cuvinte [11].

Pentru operațiile de ștergere și modificare a unui element existent cu o entitate nouă, acesta este ales aleatoriu din argumentele ultimelor 10 comenzi utilizate. Dacă comanda din care este ales elementul are două argumente, este selectat cel din urmă. Astfel, se minimizează posibilitatea ca elementul selectat să nu mai existe (dintr-o operație anterioară).

Această abordare generează un tip de input ce facilitează verificarea cazurilor limită și a situațiilor speciale, precum modificarea sau ștergerea unei valori inexistente.

3.2 Specificațiile sistemului de calcul

În procesul de evaluare, testele au fost rulate pe o mașină virtuală personală cu următoarele specificații:

- Sistem de operare: Ubuntu 20.04.2 LTS
- Procesor: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
- RAM: 8GB
- Memorie totala: 69GB
- Memorie disponibila: 16GB

3.3 Rezultatele evaluării

În urma rulării testelor și, ulterior, a consultării valorilor temporale obținute de fiecare algoritm, rezultatele pot fi grupate în trei categorii principale, în funcție de numărul de operații de executat: teste relativ mici, medii și mari.

3.3.1 Număr mic de date de intrare

Valorile rezultate în urma rulării algoritmilor pe testele cu valori mai mici sau egale cu 1000 de instrucțiuni (testele 1-22) sunt ilustrate în Tabelul 1.

Tabelul 1. Timpii de execuție ai algoritmilor pe un număr mic de date de intrare

Nr. test	Nr. instrucțiuni	**Timp AVL**	Timp Treap	Timp Hashtable
2	5	**0.001**	0.020	0.020
3	5	**0.001**	0.017	0.014
4	5	**0.001**	0.027	0.020
1	5	**0.001**	0.025	0.012
5	7	**0.001**	0.021	0.019
6	10	**0.001**	0.023	0.015
7	10	**0.001**	0.026	0.015
8	15	**0.001**	0.031	0.026
9	15	**0.001**	0.016	0.029
10	15	**0.001**	0.028	0.025
11	30	**0.001**	0.027	0.026
12	30	**0.003**	0.023	0.014
13	30	**0.003**	0.016	0.021
14	30	**0.007**	0.020	0.026
15	50	**0.001**	0.026	0.017
16	50	**0.004**	0.021	0.016
17	100	**0.001**	0.029	0.020
18	100	**0.002**	0.015	0.019
19	500	**0.001**	0.017	0.024
20	500	**0.002**	0.019	0.018
21	1000	**0.001**	0.027	0.034
22	1000	**0.001**	0.025	0.024

Graficul rezultat în urma acestui tabel este reprezentat în Fig.1

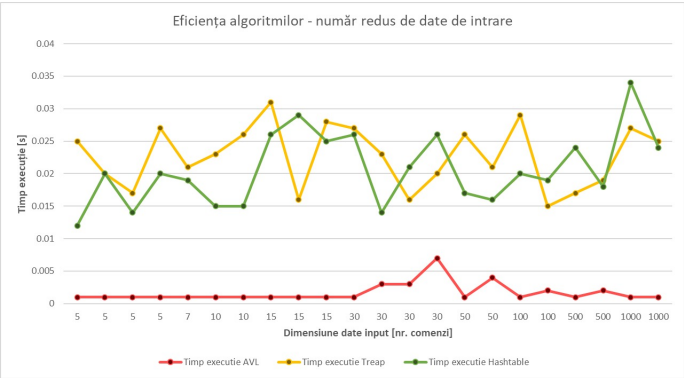

Fig. 1. Reprezentare graficaă - număr mic date de intrare

3.3.2 Număr mediu de date de intrare

Valorile rezultate în urma rulării algoritmilor pe testele cu valori cuprinse între 5000 și 40000 de instrucțiuni (testele 23-28) sunt reprezentate în Tabelul 2.

Tabelul 2. Timpii de execuție ai algoritmilor pe un număr mediu de date de intrare

Nr. test	Nr. instrucțiuni	**Timp AVL**	Timp Treap	Timp Hashtable
23	5000	**0.005**	0.051	0.142
24	5000	**0.009**	0.066	0.167
25	10000	**0.009**	0.095	0.484
26	10000	**0.008**	0.076	0.643
27	40000	**0.037**	0.265	4.286
28	40000	**0.035**	0.303	4.466

Graficul rezultat în urma acestui tabel este reprezentat în Fig.2

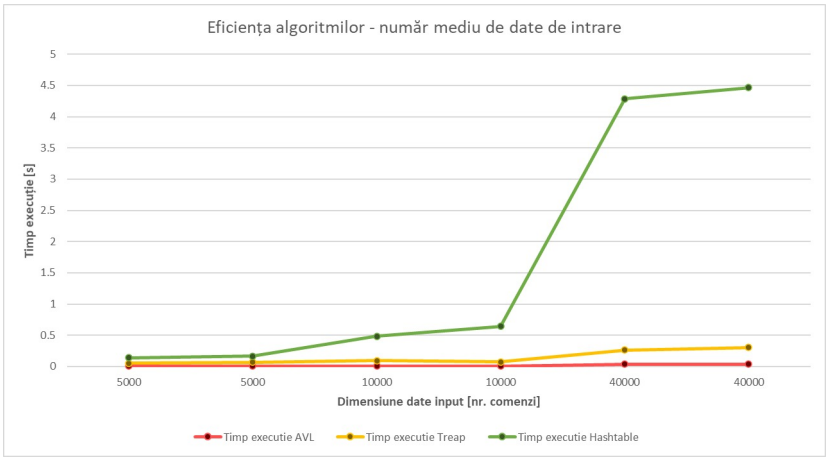

Fig. 2. Reprezentare grafică - număr mediu date de intrare

3.3.3 Număr mare de date de intrare

Valorile rezultate în urma rulării algoritmilor pe testele cu valori cuprinse între 40000 și 100000 de instrucțiuni (testele 27-30) sunt reprezentate în Tabelul 3.

Tabelul 3. Timpii de execuție ai algoritmilor pe un număr mare de date de intrare

Nr. test	Nr. instrucțiuni	**Timp AVL**	Timp Treap	Timp Hashtable
27	40000	**0.037**	0.265	4.286
28	40000	**0.035**	0.303	4.466
29	100000	**0.160**	0.909	19.999
30	100000	**0.116**	0.721	18.290

Graficul rezultat în urma acestui tabel este reprezentat în Fig.3

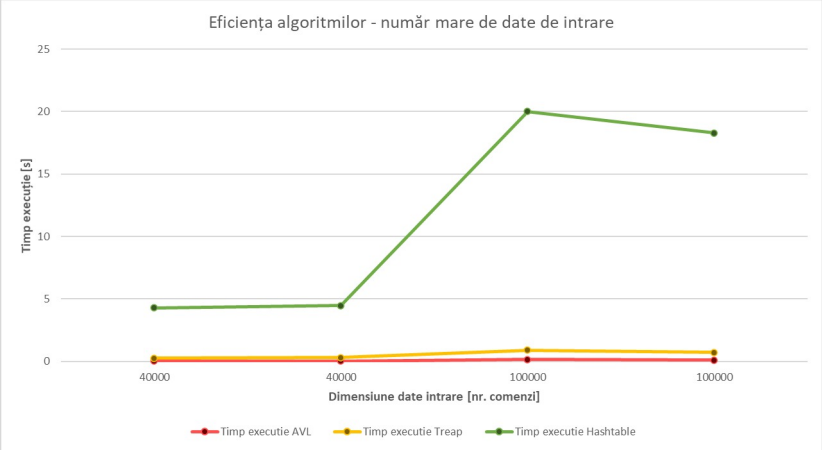

Fig. 3. Reprezentare grafică - număr mare date de intrare

3.4 Prezentarea valorilor obținute pe teste

În urma analizei reprezentărilor anterioare, concluzionăm că algoritmii se comportă conform așteptărilor.

În cazul arborelui AVL apare o valoare neașteptată, testul 14 (cu 30 de instrucțiuni) având un timp de rulare de 7 ori mai mare decât testul 22 (cu 1000 de instrucțiuni). Efectând o analiză comparativă a conținutului din cele două teste, se ajunge la valorile din Tabelul 4.

Tabelul 4. Analiză testul 14 vs. testul 22

Nr. test	14	22
Nr. total operații	30	1000
Insert	56.666%	32%
Delete	23.333%	34.4%
Modify	20%	33.6%
Timp [s]	0.007	0.001

Se observă că în primul test analizat, procentul de insert-uri este mult mai mare, raportat la numărul total de operații efectuate. Acest lucru cauzează diferența temporală deoarece, în cazul algoritmului AVL, în cadrul fiecarei operații de inserare, arborele se reechilibrează, operație costisitoare din punct de vedere temporal.

În cazul celorlalți doi algoritmi se poate observa următorul comportament: pentru testele de input cu un numar relativ mic de comenzi, hashtable-ul are o performanță mai bună decât treap-ul. Acest lucru se poate datora faptului că cel din urmă execută în plus și operații de echilibrare.

4 Concluzii

În urma testării aplicate asupra implementărilor celor trei algoritmi și a analizei complexității temporale a acestora, precum și a comparației avantajelor și dezavantajelor, se poate concluziona faptul că, în practică, este recomandată utilizarea acestor structuri de date astfel:

- hashtable-urile – pentru seturi de date al căror volum poate fi anticipat, pentru o dimensiune a datelor mică spre medie;

- arborii de tip treap și AVL – în cazul în care datele de intrare sunt medii și mari, fiind încurajată utilizarea arborilor AVL în situația în care se aplică un număr mare de interogări asupra setului de date conținut.

Bibliografie

1. Documentație disponibilă la `www.cplusplus.com` Ultima accesare: 02.11.2020
2. Basics of Hash Tables [Online] Ultima accesare: 16.12.2021
3. D. Liu and S. Xu, "Comparison of Hash Table Performance with Open Addressing and Closed Addressing: An Empirical Study", International Journal of Networked and Distributed Computing, vol. 3, no. 1, p. 60, 2015
4. S. Friedman, A. Krishnan, N. Leidefrost, "Hash Tables for Embedded and Real-time systems", 2003 [Articol] Ultima accesare: 17.12.2021
5. G. E. Blelloch, M. Reid-Miller, "Fast Set Operations Using Treaps" [Articol] Ultima accesare: 17.12.2021
6. Laborator 11 - AVL & Red-Black Trees, SD [Online] Ultima accesare: 17.12.2021
7. Laborator 4 - Dictionar, SD [Online] Ultima accesare: 17.12.2021
8. E. D. Demaine, C. E. Leiserson, "Problem Set 4 Solutions", Introduction to Algorithms, 2005 [Articol] Ultima accesare: 17.12.2021
9. K. Chresfield, Pros & Cons of Hash Tables, 2019 [Online] Ultima accesare: 17.12.2021
10. AVL Search trees: Height-balanced binary trees [Online] Ultima accesare: 17.12.2021
11. DICTIONARIES [Online] Ultima accesare: 17.12.2021