# Mulțimi (1)

Valentin - Ionuț Bobaru

Universitatea POLITEHNICA,
Facultatea de Automatică și Calulatoare

## 1 Introducere

Conceptul de mulțime (set) este des întâlnit și esențial pentru lumea ingineriei software. Pentru implementarea unei mulțimi se pot folosi o gamă largă de structuri de date (liste, vectori, arbori, stiva, heap), fiecare având avantajele și dezavantajele sale (lucrând mai eficient în funcție de modul în care sunt folosite). În această lucrare se vor compara două structuri de date, tabela de dispersie (*hash-table*) și un arbore binar de căutare echilibrat (*AVL tree*), urmărindu-se eficența temporală în funcție de numărul de valori de intrare și operațiile aplicabile unei mulțimi (inserare, ștergere, înlocuire, afișare).

**Aplicație practică:** O bază de date ce conține diferite informații despre populația dintr-un stat (numărul de pensionari, șomeri, copii etc.) și se actualizează la o anumită perioadă de timp.

## 2 Descrierea structurilor de date

### 2.1 Hash-table

Un hash-table sau tabela de dispersie este o structură de date ce mapează chei pentru valori, folosindu-se de o funcție de hash. Această funcție primește la intrare cheia și returnează un index reprezentând poziția într-un vector de sloturi, unde este ținută valoarea căutată. Pentru a putea reprezenta API-ul specific unei mulțimi, se va aplica funcția de hash pe elementele ce urmează să fie conținute de tabelă, ele fiind plasate, în vector, la poziția returnată de funcția de hash. O metrică importantă a unei tabele de dispersie este gradul de încărcare, definit ca raportul dintre sloturile ocupate în vector și numărul total de sloturi.

#### Descrierea complexităților

- ***Adăugarea, ștergerea și căutarea unui element*** se realizează, pe cazul mediu în O(1), dar poate ajuge la O(n), depinzând de funcția de hash dar și de gradul de încărcare al tabelei
- ***Găsirea elementului minim/maxim*** se realizează parcurgând întreaga tabelă, rezultând o complexitate temporală liniară
- ***Sortarea elementelor*** se realizează folosindu-se algoritmi de sortare cunoscuți, având deci o complexitate temporală de $O(n \cdot \log n)$ sau $O(n^2)$

**Importanța funcției de hash**

Ne dorim ca o funcție de hash să returneze o valoare unică pentru fiecare valoare de intrare. În realitate acest lucru este imposibil de realizat (exceptând cazul în care deja se cunosc elementele ce vor fi mapate), rezultând *coliziunile* (atunci când funcția de hash întoarce aceeași valoare pentru elemente diferite). Pentru tratarea coliziunilor se disting două mari soluții:

- ***Înlănțuire***, unde valorile ce provoacă coliziuni sunt înlănțuite folosindu-se o listă simplu înlănțuită. Astfel, vectorul de sloturi devine un vector de liste. Această abordare crește complexitatea temporală a operațiilor de căutare, inserare și ștergere până la O(n) (în cazul cel mai defavorabil, când funcția de hash returnează aceeași valoare pentru toate valorile de intrare).
- ***Adresare directă***, unde se caută un slot neocupat în vector pentru noul element. Această căutare se poate face în mai multe moduri: fie se caută primul slot disponibil, fie se verifică sloturile generate în urma aplicării unui polinom asupra hash-ului și numărului de probe (verificări) deja efectuate, fie prin aplicarea succesivă a mai multor funcții de hash. Similar înlănțuirii, complexitatea temporală crește pentru operațiile de bază.

**Înlănțuire vs Adresare directă**

Pentru a spune ce metodă de tratare a coliziunilor este mai eficientă, trebuie ținut cont de gradul de încărcare al tabelei. Astfel, pentru o tabelă de dispersie cu un grad de încărcare mic, adresarea directă se va descurca mai bine în detrimentul înlănțuirii, lucru ce se schimbă pentru un grad de încărcare apropiat de 1 (un grad de încărcare egal cu 1 va produce la inserarea într-o tabelă cu adresare directă un ciclu infinit). Această diferență se bazează pe modul în care este folosită memoria cache și este evidențiată aici[1]. Totodată, adresarea directă necesită operația de redimensionare a tabelei pe când înlănțuirea nu are limitări în ceea ce privește gradul de încărcare. Totuși, costul efectuării unei redimensionări poate fi justificat de menținerea unei complexități constante pentru operația de căutare în ambele cazuri.

### 2.2 AVL tree

Un AVL tree este un arbore binar de căutare echilibrat, în care înălțimile subarborelui stâng și celui drept diferă cu maxim 1 pentru orice nod al arborelui. Această caracteristică este menținută prin efectuarea unor rotații specifice la fiecare inserare/ștergere în arbore. Costul acestor rotații este nesemnificativ, având o complexitate temporală constantă.

**Descrierea complexităților**

- ***Adăugarea, ștergerea și căutarea unui element*** se realizează în O(h), unde h este înălțimea arborelui, iar pentru că acesta este echilibrat, $h = \log n$, unde n este numărul total de noduri din arbore, complexitatea temporală fiind astfel logaritmică

- ***Găsirea elementului minim/maxim*** se realizează, de asemenea, în timp logaritmic (deoarece căutăm "cel mai din stânga" respectiv "cel mai din dreapta" element și pentru a ajunge la acestea, trebuie să trecem prin fiecare nivel al arborelui, rezultând o complexitate O(h) sau O($\log n$))
- ***Sortarea elementelor***, fiind un arbore binar de căutare, se obține prin parcurgerea inordine a acestuia, deci o complexitate temporală liniară

### 2.3 Hash-table vs AVL tree

Pentru a decide care dintre cele două structuri de date se pretează mai bine pentru implementarea unei mulțimi, trebuie să știm ce tip de operații se vor aplica pe aceasta în majoritatea situațiilor. Astfel, un hash-table va fi preferat în cazul unei mulțimi pe care se aplică predominant operații de adăugare, ștergere, căutare datorită complexității temporale constante (pe cazul mediu), pe când un arbore AVL va fi mai eficient pentru o mulțime unde se dorește păstrarea valorilor într-un mod ordonat (acesta fiind cel mai mare avantaj pe care îl oferă arborele binar de căutare).

Un alt avantaj al arborilor AVL vine din memoria pe care o ocupă. Un arbore va reține exact câte valori vor fi adăugate, pe când un hash-table va avea un vector de dimensiune fixă în care nu se garantează că toate sloturile vor fi ocupate (din contră, se recomandă ca tabela de dispersie să aibă un grad maxim de încărcare între 0.6 și 0.75).

## 3 Evaluarea soluțiilor

Pentru evaluare se va măsura timpul de execuție al unui program scris în C ce testează funcționalitățile structurilor de date pentru mai multe seturi de intrare. Seturile vor conține un număr diferit de valori ce urmează să fie stocate (vor avea 100, 250, 500, 1000, 2500, și 5000 valori de intrare, de tip int). Se va testa eficența pentru inserare, ștergere și pentru un set de operații aleatoare (inserare, ștergere, înlocuire element, afișare elemente).

## 4 Prezentarea soluțiilor alese

Pentru hashtable s-a ales implementarea folosind înlănțuirea directă, precum și funcționalitatea de redimensionare. Tabela de dispersie se va redimensiona dacă factorul de încărcare scade sub 0.15 sau crește peste 0.75. Totodată, am decis ca fiecare listă din "vectorul de liste" al tabelei să se inițializeze treptat, doar când este nevoie (dat fiind faptul că cel puțin 25% din liste nu vor fi niciodată folosite; alternativa ar fi inițializarea tuturor listelor la crearea unui nou hashtable, aducând un plus de complexitate prin crearea listelor ce nu vor fi niciodată folosite). Ca principal dezavantaj al acestei abordări ar fi chiar redimensionarea tabelei care vine cu o complexitate temporală liniară, totuși această

funcționalitate menținând un nivel de apariție al coliziunilor mai scăzut (acesta fiind totodată un mare avantaj).

Pentru arborele AVL s-a folosit implementarea de aici[2], adăugându-se funcția de replace() și fiind modificate câteva detalii ale codului pentru a se preta pe modul meu de lucru. Avantajele și dezavantajele sunt cele specifice unui arbore AVL și anume: complexitate constantă pentru inserare, căutare și ștergere ($\log n$), memoria folosită eficient.

Pentru ambele implementări și pentru rularea testelor se asigură eliberarea memoriei alocate dinamic.

### 4.1 Descrierea complexităților

În timp ce complexitățile arborelui binar echilibrat nu sunt influențate de diverși factori, ele rămânând aceleași ca cele prezentate mai sus, pentru tabela de dispersie trebuie făcute câteva mențiuni. Tabela de dispersie va fi mereu inițializată cu 6 poziții, redimensionarea însemnând dublarea / înjumătățirea dimensiunii. Astfel, se poate observa că pentru inserarea a n numere în tabela de dispersie, funcția de redimensionare va fi apelată de $\log n - 1$ ori. Folosindune de analiza amortizată, costul mediu al unei operații de inserție ajunge la $\log n$, astfel, ne așteptăm la un comportament similar operației de inserare de la un arbore AVL. Totuși, aici intervine încă un factor foarte important, acela că operația de redimensionare se va face de mai multe ori la început (când dimensiunea tabelei este mai mică), și de mai puține ori când dimensiunea tabelei devine mai mare. Deci, teoretic, ne așteptăm ca diferența dintre timpul consumat de operațiile de inserare a celor două structuri de date să fie foarte mică sau să fie în favoarea arborelui pentru mai puține intrări și să crească în favoarea tabelei de dispersie, pentru un număr de intrări cât mai mare. Un comportament similar îl are și operația de ștergere, singura diferență fiind aceea că operația de redimensionare se apelează de $\log n-2$ ori într-o tabelă cu n intrări, ajungându-se la un cost mediu de $\log n-1$ per operație, deci ne așteptăm la un comportament similar / puțin mai bun pentru ștergerea într-o tabelă de dispersie.

## 5 Evaluare

Fișierele de intrare sunt menite să testeze diferite funcționalități, acestea având dimensiuni diferite. Astfel se testează:

- operația de inserare, măsurându-se timpul necesar creării structurii de date și inserării propriu-zise a elementelor
- operația de ștergere, măsurându-se timpul necesar ștergerii elementelor si dezalocării structurii de date; la acest pas se folosește structura de date populată anterior

– un set de operații aleatoare, incluzând inserarea, ștergerea, înlocuirea, afișarea elementelor din structura de date, măsurându-se timpul creării structurii de date, realizării operațiilor și dezalocării structurii

Fiecare dintre categoriile de mai sus se va folosi de 6 teste, acestea conținând din ce în ce mai multe intrări (pentru inserare și ștergere fiecare test are 100, 250, 500, 1000, 2500 sau 5000 de linii, iar pentru testarea operațiilor aleatoare, testele vor avea același număr de linii reprezentând operații de inserție, la care se mai adaugă, în procent de $70 - 80\%$ alte operații). Aceste teste sunt generate de scriptul de python "tests_generator.py".

### 5.1 Specificațiile sistemului

– Procesor: AMD Ryzen 7 5800U, 1.9GHz până la 4.4GHz, 8 nuclee, 16 thread-uri

– Memorie RAM: 16GB DDR4

## 6 Rezultate

Pentru determinarea rezultatelor, fiecare test a fost rulat de câte 10 ori, pentru analiză folosindu-se media aritmetică a valorilor de timp obținute (valori returnate de funcția clock[3], reprezentând o metrică a timpului petrecut de un program pe procesor). Este de menționat că acești timpi reprezintă în mod exclusiv realizarea operațiilor specifice, dar și scrierea / citirea din fișiere. Rezultatele obținute sunt prezentate în următorul tabel:

| | Insertion | | Deletion | | Random ops | |
|---|---|---|---|---|---|---|
| | Hashtable | AVL tree | Hashtable | AVL tree | Hashtable | AVL tree |
| 100 | 54 | 45 | 32 | 40 | 188 | 194 |
| 250 | 87 | 104 | 50 | 86 | 1016 | 702 |
| 500 | 232 | 187 | 144 | 171 | 2269 | 1952 |
| 1000 | 341 | 415 | 209 | 383 | 7512 | 7233 |
| 2500 | 948 | 1113 | 492 | 1023 | 49200 | 45171 |
| 5000 | 1936 | 2329 | 994 | 2122 | 184396 | 170377 |

### 6.1 Inserare

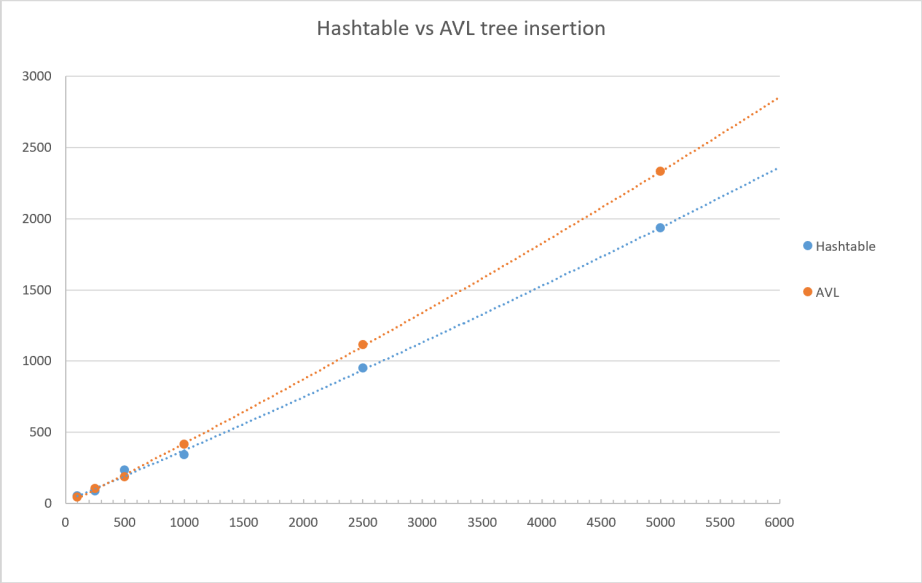


**Fig. 1.** Comparație între timpii de inserare la hashtable și arbore AVL

Așa cum era de așteptat, inserarea într-un hashtable se descurcă puțin mai bine decât inserarea în arborele AVL. Comparând valorile din tabel, observăm că arborele pornește cu un avantaj la timpul de inserție a 100 de valori (lucru datorat, foarte probabil, redimensionărilor foarte dese pentru acest număr de valori), fapt ce se schimbă odată cu creșterea numărului de valori de intrare (unde influențează, probabil, și necesitatea realizărilor operațiilor de rotație ale arborelui).

### 6.2 Ștergere

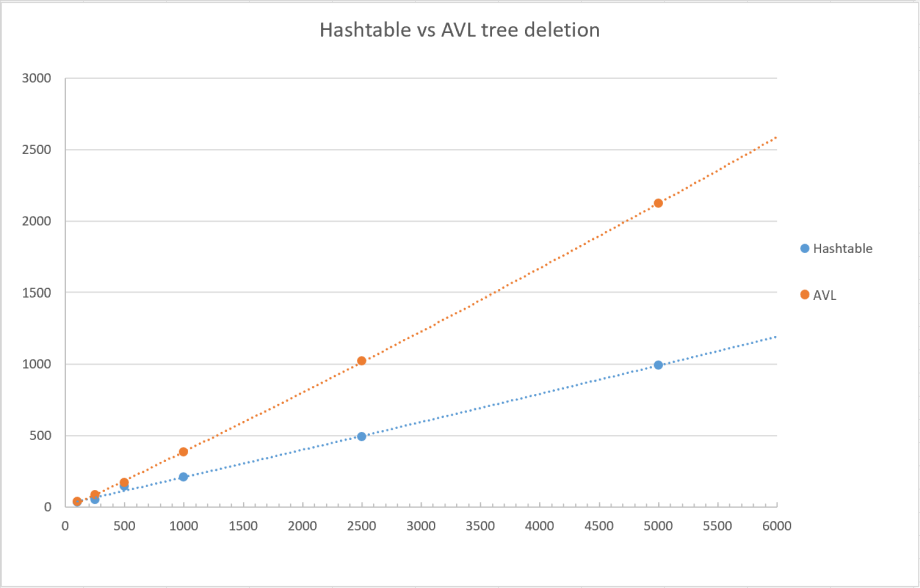


**Fig. 2.** Comparație între timpii de ștergere la hashtable și arbore AVL

În ceea ce privește ștergerea, tabela de disperise tinde să se descurce de două ori mai rapid decât arborele AVL. Această diferență pare destul de drastică, considerând costurile medii per operație ca fiind $\log n - 1$ pentru tabelă și $\log n$ pentru arbore. Probabil diferența este accentuată și de operațiile de rotație pe care arborele le face pentru a-și menție caracteristica de echilibrat, dar și de caracterul recursiv al funcției apelate.

### 6.3 Operații aleatoare

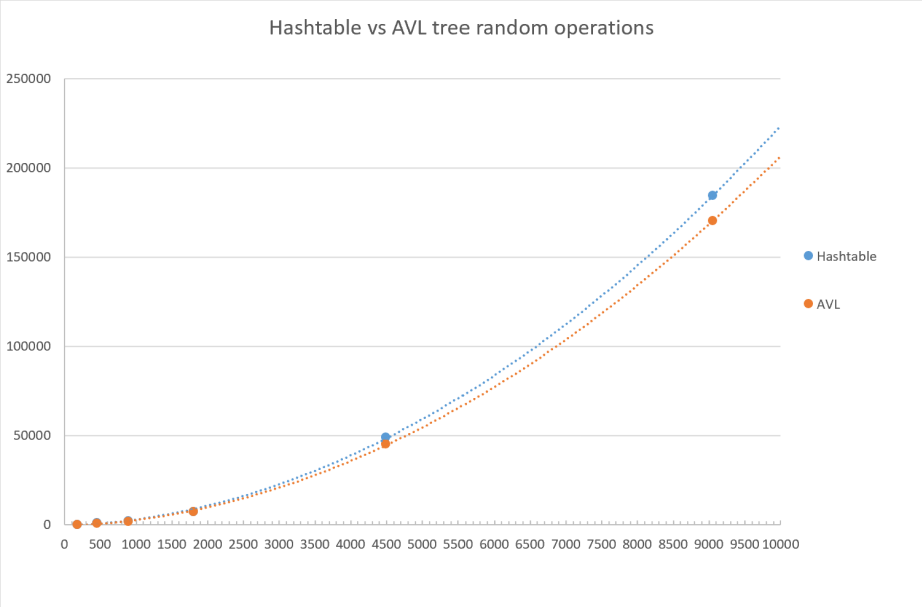


**Fig. 3.** Comparație între timpii de realizare a diferitelor operații la hashtable și arbore AVL

Având în vedere faptul că setul de teste pentru această etapă a fost creat în mod aleator, neexistând o proporționalitate între numărul de operații de fiecare tip, este mai relevantă diferența dintre timpii obținuți pe fiecare test, ci nu tendința generală. Astfel, comparând valorile din tabelul cu rezultate, dar și observând graficul, concluzionăm că cele două structuri de date se comportă similar, cu un mic avantaj pentru arborele AVL. Având în vedere că afișarea elementelor se realizează în timp liniar pentru ambele structuri (ba chiar cu un plus de complexitate pentru arbore, întrucât operația se realizează prin apeluri recursive), tind să cred că diferența de timp vine din discontinuitatea operațiilor de inserare, ștergere și înlocuire. Deși am arătat mai sus că tabela de dispersie se descurcă mai bine atât pentru operații de inserare cât și de ștergere, trebuie luat în considerare faptul că am testat inserarea / ștergerea succesivă a mai multor valori. În testele cu operații aleatoare, ștergerea și inserarea nu se mai realizează în mod succesiv, ci aleator. De aceea numărul de redimensionări se poate schimba. Acest lucru poate fi și un avantaj, dacă ne asigurăm că tabela de dispersie va avea mereu gradul de încărcare între 0.15 si 0.75, vom ști că tabela nu se va mai redimensiona, rezultând un timp de execuție mai scăzut.

## 7 Concluzii

Conform datelor și analizei prezentate mai sus, putem distinge cazurile în care fiecare structură de date se va descurca cel mai bine. Astfel, pentru aplicații care presupun multe inserări și ștergeri, în blocuri mari, alegerea mai bună este o tabelă de dispersie. Totuși, dacă aplicația practică necesită o varietate mai mare de operații ce presupun mai degrabă prelucrarea datelor, ci nu modificarea lor (spre exemplu, afișarea elementelor, determinarea minimului / maximului, parcurgeri, sume etc.), ar fi mai înțelept să alegem implementarea unui arbore binar de căutare echilibrat. Totodată, contează foarte mult volumul de date cu care se lucrează, deoarece, pentru un volum nu foarte mare de date (precum cele considerate în această lucrare), diferența dintre cele două structuri de date este nesemnificativă.

## References

1. Înlănțuire vs Adresare Directă, https://en.wikipedia.org/wiki/File:Hash_table_average_insertion_time.png, ultima accesare 22.11.2022
2. AVL tree implementation, https://www.programiz.com/dsa/avl-tree, ultima accesare 19.12.2022
3. Clock manual page, https://man7.org/linux/man-pages/man3/clock.3.html, ultima accesare 19.12.2022