Proiect Analiza Algoritmilor
Mulțimi (1)

Văideanu Renata-Georgia

Grupa 322CD
Facultatea de Automatică și Calculatoare
Universitatea Politehnica din București

Rezumat

În cadrul acestui proiect, vor fi analizate două structuri de date (tabele de dispersie și arbore binar de căutare echilibrat) care pot fi folosite pentru a reprezenta API-ul specific unei mulțimi: adăugarea unui element, ștergerea elementului minim (/maxim), afișarea elementului minim (/maxim), modificarea unui element din mulțime, afișarea elementelor mulțimii. Scopul documentului este de a scoate în evidență diferențele dintre cele două moduri de implementare, printr-un set complet de variante de mulțimi, care să testeze toate aspectele de eficiență.

Cuvinte cheie:
Tabel de dispersie, arbore binar de căutare echilibrat, API, structuri de date, eficiență, timp de execuție, memorie folosită.

1 Introducere

În informatică, o structură de date este o metodă sistematică de stocare a informațiilor și datelor într-un calculator, în așa fel încât ele să poată fi folosite în mod eficient. Deseori o alegere bine făcută a structurii de date va permite și implementarea unui algoritm eficient. Structura de date aleasă este derivată de multe ori dintr-un tip de dată abstract. O structură de date bine concepută permite efectuarea unei varietăți de operații de bază, utilizând puține resurse (ca de exemplu memoria necesară și timpul de execuție). Structurile de date se implementează utilizând tipuri de date, referințe și operații asupra acestora, toate facilitate de către un limbaj de programare.

În general, structura de arbore se referă la o relație de ramificare la nivelul nodurilor, asemănătoare aceleia din cazul arborilor din natură. Un arbore binar echilibrat este definit de proprietatea ca, pentru fiecare nod, diferența dintre adâncimea subarborelui stâng și cea a subarborelui drept este cel mult 1.

Un arbore binar de căutare este un arbore binar care are în plus următoarele proprietăți: cheile stocate în noduri (informația utilă) aparțin unei mulțimi peste care există o relație de ordine, cheia din fiecare nod este mai mare decât cheile tuturor nodurilor din subarborele stâng, cheia din fiecare nod este mai mică decât cheile nodurilor ce compun subarborele drept. Arborii binari de căutare permit menţinerea datelor în ordine şi o căutare rapidă a unei chei, ceea ce îi recomandă pentru implementarea de mulţimi şi dicţionare ordonate.
O importantă caracteristică a arborilor de căutare, este aceea că parcurgerea în inordine produce o secvență ordonată crescător a cheilor din nodurile arborelui.

Tabelele de dispersie reprezintă o modalitate foarte eficientă de implementare a dicţionarelor. Asemănător dicţionarului, pentru acest tip de dată fiecărei intrări îi este asociată o anumită valoare (sau chiar mai multe). Valoarea poate fi una numerică sau nu. Operația de găsire a unei valori asociate unei chei poartă numele de indexare, aceasta fiind și cea mai importantă operație (din acest motiv dicționarele se mai numesc și array-uri asociative - fac asocierea între o cheie și o valoare).

În practică, tabelele de dispersie pot fi folosite pentru sistemele de gestiune a bazelor de date (SGBD), mai ales cele care necesită acces aleator eficient (spre deosebire de cele cu acces secvenţial), în acest caz inputul va fi un număr.

2 Analizarea celor două tipuri de structuri de date

2.1 Arbore binar de căutare echilibrat

Un arbore binar de căutare echilibrat are proprietatea de echilibru. Aceasta trebuie respectata pentru fiecare nod - înălțimea subarborelui stâng al nodului diferă de inaltimea subarborelui drept al nodului prin cel mult o unitate.

Operația de inserarea a unui element într-un arbore binar de căutare echilibrat adaugă elementul ca frunza a arborelui, iar mai apoi nodul cu cheia data va fi inserat ca fiu stâng sau drept în așa fel încât să se respecte proprietatea arborelui. Indiferent de situație, dezechilibrul poate fi tratat, printr-o singura rotație simpla sau dubla aplicată unui nod. Pentru a pune în evidenta nodul care trebuie prelucrat printr-o rotație trebuie examinata ramura parcursa pentru determinarea poziției în care se face inserarea.

La partea de ștergere, trebuie avut în vedere:

-dacă nodul este frunză - se eliberează spațiul ocupat și se seteaza legatura părinților către acel nod cu NULL;

-dacă nodul are doar un subarbore - nodul e scos din arbore și singurul său copil (subarbore) e legat direct la părintele său;

-dacă nodul are doi subarbori - se bazează pe ideea că același set de valori poate fi reprezentat folosind doi arbori diferiți în așa fel încât să ne reducem la unul dintre cele două cazuri de mai sus.

În cazul în care se dorește ștergerea elementului maxim/minim, algoritmul va trata cazul de ștergere a unei frunze. Asemanator se procedează în cazul afișării elementului maxim/minim.

În cazul acestui proiect s-a decis să folosesc structura unui AVL. Arborii AVL elimină neajunsul major al arborilor binari: faptul că înălţimea şi, deci, viteza de căutare depinde de ordinea în care sunt introduse cheile în arbore. Arborii AVL permit obţinerea unei viteze de căutare constante prin garantarea faptului că arborele este echilibrat la orice moment. Structura unui nod este cea a unui nod de arbore binar la care se mai adaugă un câmp numit BF (Balance Factor) care reprezintă diferenţa dintre înălţimea subarborelui drept (RH) şi înălţimea subarborelui stâng (LH).

2.2 Tabele de dispersie

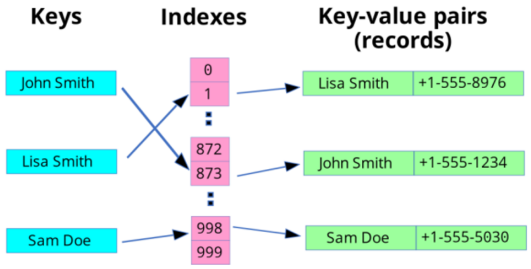


Figura 1: Tabelă de dispersie

Un tabel de dispersie este o structură de date optimizată pentru funcția de lookup (în medie, timpul de căutare este constant; O(1)). Astfel, aceasta structura de date reprezintă de fapt un vector în care accesul la date se face prin intermediul unui index special, denumit cheie . În cadrul acestei colecții de date are loc o transformare a unei chei într-un coș de dispersie, care oferă o anumită informație specifică. Este de dorit ca o funcția necesara pentru rezolvarea programului să fie injectivă, adică la două chei diferite k1 != k2 să corespundă valori de dispersie diferite hash(k1) != hash(k2), în caz contrar apare o coliziune.

În mod normal, numărul de intrări dintr-un tabel de dispersie este mic prin comparaţie cu mulţimea cheilor posibile. Prin urmare, cele mai multe funcţii de dispersie asociază mai multe chei aceleiaşi intrări în tabelă. Atunci când două chei se asociază aceleiaşi intrări se spune că ele se lovesc, fenomen denumit coliziune. Adresare directă - atunci când o funcţie de dispersie garantează că nu există două coduri identice pentru chei diferite, atunci tabelul de dispersie este denumit cu adresare directă. O implementare a unui tabel de dispersie care tratează coliziunile se numește înlănțuire directă (direct chaining). Cea mai simplă formă folosește câte o listă înlănțuită pentru fiecare bucket. Fiecare listă este asociată unei anumite chei. Inserarea în hash presupune găsirea indexului corect și adăugarea elementului la lista corespunzătoare. Ștergerea presupune căutarea și scoaterea elementului din lista corespunzătoare.

Metode de rezolvare a coliziunilor:

-*Înlănțuire* → În fiecare poziție din tabel ținem o listă înlănțuită: insert, delete și search parcurg toată lista. Pe un caz pur teoretic, toate cele N elemente ar putea fi repartizate în aceeași locație, însă pe cazuri practice lungimea medie a celui mai lung lanț este de lg(N). Dispersia cu înlănţuire separată are dezavantajul că necesită pointeri, alocare de memorie, deci este lentă.

-*Adresare deschisă* → toate elementele sunt memorate în interiorul tabelei de dispersie (fiecare intrare în tabelă conţine fie un element al mulţimii dinamice, fie NIL). Nu există liste sau elemente memorate în afara tabelei, aşa cum se întâmplă în cazul înlănţuirii. Avantaj: adresarea deschisă evită total folosirea pointerilor. Spaţiul de memorie suplimentar, eliberat prin faptul că nu memorăm pointerii, oferă tabelei de dispersie un număr mai mare de locaţii pentru acelaşi spaţiu de memorie, putând rezulta coliziuni mai puţine şi acces mai rapid.

3 Evaluarea soluțiilor

3.1 Implementarea algoritmilor

	Worst case		Average case	
	AVL	Hash	AVL	Hash
Search	$O(\log_2 n)$	O(1)	$O(\log_2 n)$	O(n)
Insert	$O(\log_2 n)$	O(1)	$O(\log_2 n)$	O(n)
delete	$O(\log_2 n)$	O(1)	$O(\log_2 n)$	O(n)

Tabel 1: Complexitate timp pentru AVL și Tabela de dispersie

Tabelele de dispersie acestea asigură complexitate constantă O(1) în medie, pentru operaţiile de inserare, ştergere şi căutare. Dispersia nu presupune ordonarea informaţiei păstrate. Operaţiile cu arbori, care presupun ordonarea informaţiei între elemente sunt mai puţin eficiente (O(log n)). Dacă:

-structura de date (tabela de dispersie) e accesată prin valoarea unei chei în O(1);

-funcţia care transformă cheia într-o poziţie într-un tabel are complexitate O(1) atunci inserarea / ştergerea / căutarea se vor face cu complexitate O(1).

Complexitatea timp de parcurgere a unui arbore binar de căutare echilibrat este de maxim O(logN) - cost amortizat pe mai multe operații. Valoarea logN este dată de numărul de elemente ale arborelui. Astfel, operațiile de căutare/adăugare/ștergere/găsire min-max de element vor avea aceeași complexitate timp. Cazul cel mai defavorabil în cadrul operației de inserare, va fi acela cand se va dori inserarea unui element care se află la mijlocul intervalului [a,b] (acesta fiind intervalul în care se găsesc valorile nodurilor), întrucât se va face un număr mai mare de rotiri.

Astfel, în cadrul testelor, principala diferență dintre cele două structuri de date va fi dată de numărul de elemente adăugate. Dacă setul de date este unul foarte mare nu se garantează că nu vor exista informații duplicate, ceea ce va determină apariția coliziunilor în cadrul tabelelor de dispersie, și va costa mult mai multă memorie alocată.

În urma seturilor de teste ce vor fi aplicate asupra celor doua tipuri de structuri de date se va realiza o expertiza mai concreta asupra timpului de executie, tinandu-se cont totodata de cazurile de excepție.

3.2 Formularea testelor

Fisierele de input sunt de forma:

Pe prima linie se află numărul N de query-uri

Pe următoarele N linii se află câte un query de forma:

0 a - Se adaugă elementul ‘a’ în mulțime

1 a - Se elimină elementul ‘a’ din mulțime

2 a - Verifică dacă elementul ‘a’ aparține mulțimii

1 cm 3 a b - Elementul ‘a’ din mulțime (dacă există) este înlocuit cu elementul ‘b’

4 - Afișarea tuturor elementelor mulțimii

N	a, b	test nr
10 -> 10^2	1 -> 10^2	00 - 04
10^2 -> 10^3	1 -> 10^2	05 - 09.
10^3 -> 10^4	1 -> 10^2	10 - 14.
10^4 -> 10^5	1 -> 10^2	15 - 19.
10 -> 10^2	1 -> 10^3	20 - 24.
10^2 -> 10^3	1 -> 10^3	25 - 29.
10^3 -> 10^4	1 -> 10^3	30 - 34.
10^4 -> 10^5	1 -> 10^3	35 - 39.
10 -> 10^2	1 -> 10^4	40 - 44.
10^2 -> 10^3	1 -> 10^4	45 - 49.
10^3 -> 10^4	1 -> 10^4	50 - 54.
10^4 -> 10^5	1 -> 10^4	55 - 59
10 -> 10^2	1 -> 10^5	60 - 64.
10^2 -> 10^3	1 -> 10^5	65 - 69.
10^3 -> 10^4	1 -> 10^5	70 - 74.
10^4 -> 10^5	1 -> 10^5	75 - 79.
10 -> 10^2	1 -> 10^6	80 - 84.
10^2 -> 10^3	1 -> 10^6	85 - 89.
10^3 -> 10^4	1 -> 10^6	90 - 94.
10^4 -> 10^5	1 -> 10^6	95 - 99.

Tabel 2: Organizarea testelor

Asupra celor doi algoritmi, s-au aplicat 100 de teste, acestea fiind organizate în seturi de câte 5 în funcție de numărul de intrări (N) și de intervalul în care se încadrează elementele (a și b), așa cum se poate observa în tabelul 1.

N = numărul de query-uri;

a,b = intervalul elementelor care trebuie șters/ adăugate/ schimbate/ căutate;

test nr = numerele testelor care îndeplinesc condițiile date;

Fiecare test din fișierul destinat testelor este de forma testXX.in. De fiecare dată XX va fi înlocuit cu un număr intre 0 și 99.

Generarea testelor se face în mod aleator, fiind setate doar intervalele pentru N, respectiv a și b. S-a optat pentru o gama mai larga de teste, cate 5 pentru fiecare interval pentru a se putea observa mai ușor variația de timp pentru fiecare dintre cei doi algoritmi.

3.3 Analizarea rezultatelor

Din cele doua tabele de mai jos se poate observa că cei doi algoritmi au performante asemănătoare atunci când elementele care se regăsesc în AVL/tabela de dispersie sunt relativ mici. Diferențele mai mari de timp se pot vedea atunci când pragul superior a intervalui în care se găsesc a și b devine din ce în ce mai mare. În cazul în care nu se cunoaște intervalul elementelor care trebuie adăugate/căutate/șterse/înlocuite se va aloca memoria maxima pentru tabela de dispersie, 106. În cazul contrar, se poate folosi pragul superior a intervalului în care se găsesc elementele care urmează a fi introduse în mulțime și se poate aloca memorie pentru : $10^{(int)(\log_{10}(max)+1)}$ elemente, unde:

$(int)(\log_{10}(max)+1)$ este numărul de cifre al celui mai mare element care urmează a fi adăugat în mulțime (ex: daca elementul maxim este 999, $(int)(\log_{10}(999)+1)$ va fi 3, alocându-se memorie pentru 1000 elemente.

a, b / N	1 - 10^2	1 - 10^3	1 - 10^4	1 - 10^5	1 - 10^6	Med / a,b
10 - 10^2	**0**	**0**	**0**	3.125	34.375	7.5
10^2 - 10^3	**0**	**0**	23.4375	21.875	256.25	60.3125
10^3 - 10^4	**3.125**	37.5	87.5	453.125	2928.125	701.875
10^4 - 10^5	15.625	**140.625**	709.375	2062.5	6259.375	1837.5
Med / N	4.6875	44.53125	205.07813	635.15625	2369.5313	651.796875

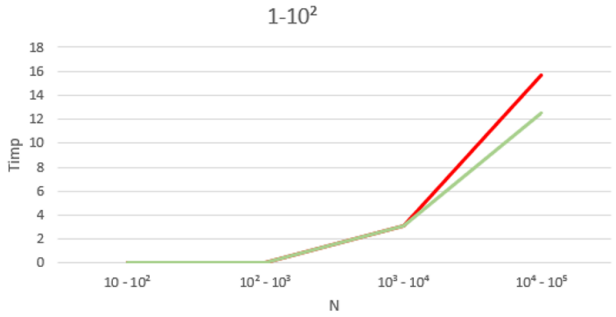
Tabel 3: Variația de timp (ms) a tabelei de dispersie în raport cu intervalele datelor

a, b / N	1 - 10^2	1 - 10^3	1 - 10^4	1 - 10^5	1 - 10^6	Med / a,b
10 - 10^2	**0**	**0**	**0**	**0**	**0**	0
10^2 - 10^3	**0**	**0**	**13.020833**	**0**	**0**	2.604166667
10^3 - 10^4	**3.125**	**25**	**53.125**	**93.75**	**56.25**	46.25
10^4 - 10^5	**12.5**	146.875	**618.75**	**684.375**	**268.75**	346.25
Med / N	3.90625	42.96875	171.22396	259.375	81.25	98.77604167

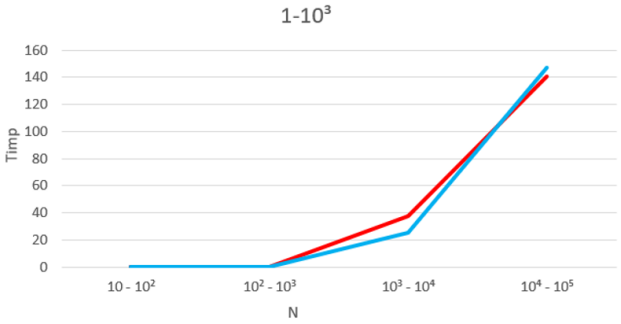
Tabel 4: Variația de timp (ms) a AVL-ului în raport cu intervalele datelor

3.4 Interpretare grafice

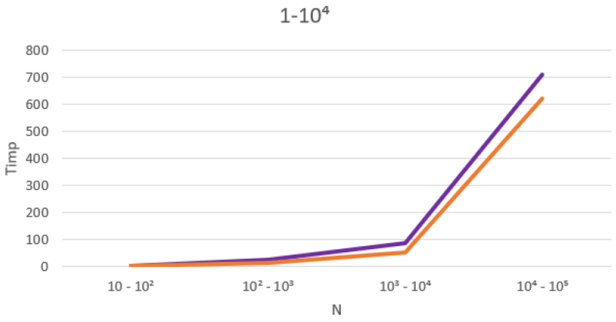
Cu ajutorul tabelelor 3 și 4 se pot forma grafice pentru a se putea observa mai ușor modul în care fluctuează timpul în cazul celor structuri de date. Axa Oy este destinată timpului mediu și axa Ox este N, numărul de acțiuni care vor fi executate de cele două structuri de date.



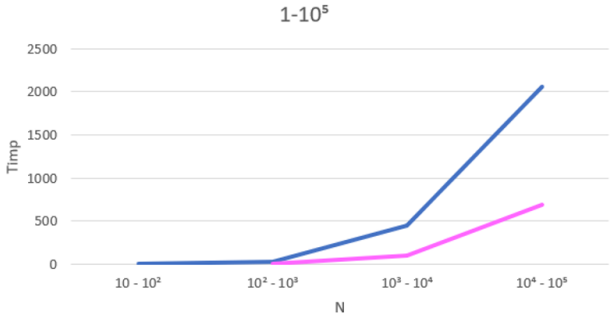
Astfel, în cazul testelor cu indicele de la 0 la 19, elementele din AVL(verde)/ tabela de dispersie(roșu) sunt mai mici decât 100. Ținând cont de acest fapt, se va aloca în prealabil memorie pentru 100 de elemente în cazul hash-ului. Creșterea timpului este una constanta în ambele cazuri, AVL-ul fiind mai rapid, în medie, cu 3ms.



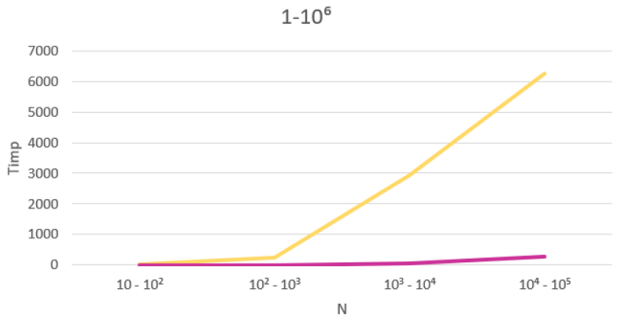
În continuare, testele de la 20 la 39 vor trata elementele din AVL(albastru)/ tabela de dispersie(roșu) mai mici decât 1000. După cât se poate vedea, timpul de execuție este unul asemănător, diferența dintre cele doua metode de rezolvare fiind de aproximativ 2ms.



Testele de la 40 la 59 conțin elemente mai mici decât 10000. Și în acest caz se poate observa, ca și în cazul anterior, o creștere treptata. Algoritmul care folosește tabela de dispersie(mov) este mai lent decât cel care folosește AVL(portocaliu). Diferența dintre cele doua este mai mare de 30ms.



Testele de la 60 la 79 vor trata elementele din AVL(mov)/ tabela de dispersie(albastru) mai mici decât 100000. Aici se pot observa diferențe mai mari intre cele doua structuri de date. Acest fapt poate fi datorat parcurgerii tabelei de dispersie din ce în ce mai mare, în cazul afișării tuturor elementelor.



Testele de la 80 la 99 vor trata elementele din AVL(mov)/ tabela de dispersie(galben) mai mici decât 1000000. Ținând cont că memoria alocata pentru hash este de 106, parcurgerea întregii mulțimi va fi mult mai lenta (trebuind parcurse și elemente care sunt goale) decât cea a AVL-ului (toate elementele au valori, nefiind elemente goale.

3.5 Specificațiile sistemului

Sistemul de calcul pe care s-au rulat testele este:
Hardware:
- Procesor: Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz 1.19 GHz;
- RAM: 8.00 GB (7.75 GB usable);
- Memorie disponibilă: 590 GB;

Software:
- OS: Ubuntu 7.5.0-3ubuntu1 18.04;
- GCC: gcc 7.5.0.

4 Concluzie

După cate se poate observa, atât din tabele cât și din grafice, cele doua structuri de date au un timp de execuție asemănător. Principalul factor de diferențiere este întâlnit atunci când se cere afișarea întregii mulțimi, fiind necesara parcurgerea tuturor elementelor.

În cazul AVL-ului se va realiza o parcurgere a tuturor valorilor din arbore, rezultând o complexitate timp de O(n), n fiind numărul total de noduri. În cazul tabelelor de dispersie daca se dorește afișarea tuturor elementelor, va trebui să se parcurgă întreaga dimensiune a structurii de date, chiar daca nu se găsesc valori la unele poziții. Daca cheia exista atunci va trebui să fie parcursa și lista simplu înlănțuita, rezultând un algoritm mai ineficient.

Cu toate acestea, pentru average case, tabelele de dispersie se comporta mai bine, aceasta având o complexitate timp constant de O(1), în timp ce AVL-ul are o complexitate timp de O(log2n), n fiind numărul de elemente din arbore.

Astfel, structura de date de tip tabele de dispersie se comporta mai bine în cazurile în care exista mai multe căutări/ adăugări/ ștergeri decât în cazurile în care sunt mai multe afișări. Pe de alta parte, structura de tip AVL conferă o complexitate timp, in medie, mai mare pentru inserări/ ștergeri/ căutări, și mai mica pentru afișări.

În concluzie, este recomandata folosirea AVL-urilor întrucât nu necesita alocarea de memorie pentru arborele întreg și nici nu necesita folosirea listelor simplu înlănțuite pentru inserarea dublurilor. Astfel, se concluzionează că AVL-ul este o structura mult mai eficienta si mai sigura decât tabelele de dispersie in majoritatea situațiilor, fiind o opțiune mai buna de folosit in practica.

5 Bibliografie

[1] Structuri de date - Wikipedia(ro)

[2] Structuri de date - Wikipedia(eng)

[3] Arbore binar de căutare

[4] Arbore binar de căutare echilibrat - OCW

[5] Arbore binar de căutare echilibrat - Wikipedia(eng)

[6] Tabele de dispersie - OCW

[7] Tabele de dispersie - Wikipedia(eng)

[8] AVL - OCW

[9] Tabele de dispersie - Digital Ocean