Raport Tehnic

Florea Maria-Alexandra
Sasu Alexandru-Cristian
Toader Vlad-Marian
(Grupa 142)

1 Descrierea structurii alese - AVL tree

Arborii binari de căutare echilibraţi AVL (Adelson-Velski, Landis) sunt arborii binari de căutare care au următoarele proprietăţi:

- pentru fiecare nod din arbore, înălţimea subarborelui stâng diferă de înălţimea subarborelui drept prin maxim un nod;
- fiecare subarbore este un arbore binar de căutare AVL.

2 Motivația structurii de date alese

Am ales să folosim arborii AVL, deoarece sunt cei mai frecvent folosiți arbori binari de căutare echilibrați și susțin inserția, ștergerea și căutarea în garanție de timp O(log n). Arborii AVL sunt mai rigid echilibrați și, prin urmare, oferă căutări mult mai rapide decât ceilalți arbori. Astfel, pentru o activitate intensă de căutare, este indicat să folosim arborii AVL. Înălțimea maximă a unui arbore AVL este de aprox. 1.44*log(n), deci în cazul cel mai rău căutarea într-un arbore AVL nu necesită mai mult de 44% comparații

față de cele necesare într-un arbore perfect echilibrat sau într-un arbore binar de căutare simplu (50%). În medie, este necesară o rotație (simplă sau dublă) cam la 46,5% din adăugări și este suficientă o singură rotație pentru refacere (consecințe directe ale teoremei demonstrate de Adelson, Velskii și Landis).

3 Avantajele și dezavantajele structurii de date folosite

3.1 Avantaje

- un arbore AVL produce cel mai echilibrat arbore în cazul cel mai defavorabil;
- fiind perfect balansat, arborii AVL scot cel mai mic timp de căutare dintre toţi ceilalţi arbori binari de căutare. Constanta pentru căutare (informație obținută din cartea lui Knuth) este 1.5 (deci 1.5*log(n)), pe când arborii R-B au o constantă de 2 (deci 2*log(n) pentru căutare);
- sunt ușor de înțeles algoritmii;
- avem spațiu extra, 2 biți pentru fiecare nod (trebuie reținut +1,0,-1), dar se poate chiar folosi un singur bit pentru fiecare nod prin utilizarea singurului bit al fiului.

3.2 Dezavantaje

- un arbore AVL efectuează un număr mai mare de rotaţii; codul pentru un arbore AVL este mult mai complex decât codul pentru un arbore binar de căutare simplu, pentru că trebuie rezolvate toate cazurile extreme (corner cases);
- operațiile de ștergere au un cost mare în arborii AVL, chiar dacă complexitatea este de O(log n), pentru că aceștia implică schimb de pointeri și rotații, spre exemplu arborii R-B efectuează o singură rotație.

4 Analiza timp a programului

În cadrul descrierii complexităților următoare, excludem analiza funcțiilor "GenerareTest" și "VerificaFisiere", întrucât nu sunt corelate cu eficiența structurii de date proiectate, ci cu modalitățile de a genera teste pentru aceasta, respectiv de a valida corectitudinea operațiilor sale.

În cele ce urmează, vom nota cu litera "n" numărul de noduri ale arborelui asociat structurii de date.

- **"insereaza" – Complexitate timp O(log n)**

  Datorită operațiilor efectuate de arborii de tip AVL, care pot surveni în urma inserării nodurilor: rotație simplă stânga, rotație dublă stânga, rotație simplă dreapta, rotație dublă dreapta - numele metodelor prin care se realizează echilibrarea arborelui, a căror complexitate timp este O(1); înălțimea maximă a arborelui va fi, întotdeauna, aproximativ 1,44*logn. Prin urmare, complexitatea timp a operației prin care se verifică daca nodul care se dorește a fi introdus se află deja in arbore, (operație realizată la fel ca și pentru arborii binari de căutare simpli) este O(logn). Datorită celor două operații de mai sus, (operația de rotație, alături de cea prin care se caută nodul în arbore) inserarea unui nod are o complexitate timp O(logn)+O(logn) = O(logn) (primul O(logn) este datorat numărului maxim de rotații care pot avea loc în urma inserării unui nod).

- **"sterge" – Complexitate timp O(log n)**

  Asemenea inserării, operația de ștergere este alcătuită din două operații, fiecare având complexitatea timp O(log n): găsirea nodului care se dorește a fi șters, alături de echilibrarea arborelui în urma ștergerii nodului (realizată prin cele patru tipuri de rotații utilizate la inserare); ceea ce rezultă într-o complexitate timp totală O(log n).

- **"min", "max", "succesor", "predecesor", "este_in", "AflareNodMinim" – Complexitate timp O(log n)**

  Având înălțimea maximă de 1,44*log n, orice operație de căutare a unui nod (având la bază operația de căutare a unui nod într-un arbore binar de căutare simplu) în cadrul arborelui, se realizează într-o complexitate timp O(1,44*logn), adică O(logn).

- **“k_element” – Complexitate timp O(n)**

  Aflarea celui de-al k-lea element din arbore în ordine crescătoare se realizează prin intermediul unei parcurgeri în inordine a arborelui, a cărei complexitate timp este O(n).

- **“cardinal” – Complexitate timp O(1)**

  Pe măsură ce se inserează sau se șterg elemente din arbore, se incrementează, respectiv decrementează, o variabilă care reține numărul de elemente ale structurii. Așadar, operația de aflare a numărului de elemente din arbore se reduce la accesarea unei variabile, ceea ce implică o complexitate timp O(1).

- **“RotatieStanga”, “RotatieDreapta” – Complexitate timp O(1)**

  Funcții auxiliare, care realizează operațiile de rotație precizate în descrierea funcției “insereaza”.

- **“InaltimeArbore” – Complexitate timp O(1)**

  Funcție auxiliară, care află înălțimea unui arbore. Aceasta presupune accesarea unui singur câmp al unui nod, anume câmpul care reține înălțimea arborelui care are ca rădăcină nodul accesat.

- **“Maxim” – Complexitate timp O(1)**

  Funcție auxiliară, care află maximul dintre două numere.

- **“AfisareArboreInord” – Complexitate timp O(n)**

  Funcție auxiliară, care afișază nodurile arborelui în inordine.

- **“AflareFactorEchilibrare” – Complexitate timp O(1)**

  Funcție auxiliară, care calculează factorul de echilibrare al unui nod. Acest calcul se realizează prin apelarea funcției “InaltimeArbore” de două ori, pentru nodul dat. O dată pentru a afla înălțimea subarborelui stâng. Iar a doua oară, pentru a afla înălțimea subarborelui drept. Aceste două înălțimi se scad, obținându-se astfel factorul de echilibrare al nodului.

Datorită funcțiilor “k_element” și “AfisareArboreInord”, complexitatea timp totală a structurii de date, pentru apelarea fiecărei funcții o singură data, este O(n).

5 Descrierea modului de testare

Pentru a dovedi corectitudinea programului și eficiența structurii de date alese, am decis ca testele de input să fie generate aleator. Astfel, garantăm că programul rulează într-un timp scurt și pe teste cu un număr mare de elemente, (cel mai mare test generat are peste 30000 de numere) dar și pe teste cu un număr mic de elemente.

Cele 5 seturi de fișiere de input au fost realizate folosind funcția "GenerareTest", cu ajutorul căreia se generează un număr aleator n, un șir de n numere distincte alese aleator (care sunt introduse în fișierul "random_order.txt" - ce va servi ca input pentru inserarea în arbore) și un număr aleator de numere pentru a testa funcția de ștergere (care sunt introduse în fișierul "f_delete.txt"). De asemenea, se crează un fișier de output, "sorted_order.txt", în care sunt afișate operațiile corespunzătoare arborelui, dar realizate pe liste și set-uri, pentru a demonstra astfel că ambele rezultate coincid (cele din fișierul "sorted_order.txt" - output-ul listelor și set-urilor, și cele din fișierul "output_avl.txt" - output-ul arborelui).

De asemenea, dacă, în cadrul funcției "VerificaFisiere", se decomentează secțiunea care apelează funcția "GenerareTest", se pot genera noi teste, care se introduc automat în fișierele .txt (timpul de rulare al programului va crește datorită generării numerelor aleatorii).

Se poate observa că în urma rulării programului cu cele 5 seturi de fișiere de input, s-au obținut rezultate semnificativ mai bune decât dacă s-ar fi folosit structuri de date clasice (precum arbori binari de căutare simpli sau heap-uri). Astfel, rezultatele obținute au fost:

- pentru testul cu 5692 numere - 0.104s;
- pentru testul cu 8298 numere - 0.044s;
- pentru testul cu 17 numere - 0.026s;
- pentru testul cu 31880 numere - 0.119s;
- pentru testul cu 28737 numere - 0.111s.

Rezultatele obținute sunt mai bune decât dacă s-ar fi folosit heap-uri datorită complexității mai mici a operațiilor, și datorită implementării în cod, care este tot una eficientă.

Astfel, tragem concluzia că structura de date aleasă este eficientă, reușind să facă toate cele 9 operații într-un timp scurt, având avantaje considerabile în comparație cu structurile de date clasice (arbori binari de căutare simpli, heap-uri).

6 Sales pitch

Considerăm că implementarea mulțimii cu ajutorul unui arbore binar echilibrat de tip AVL este una benefică, deoarece are o complexitate de timp mai scăzută decât alte structuri de date similare (arbori binari de căutare simpli, heap-uri). În plus, arborii binari de tip AVL sunt cei mai frecvent folosiți arbori binari de căutare echilibrați, având complexitățile de timp pentru operațiile de inserare, ștergere și căutare O(log n). În plus, un arbore AVL produce cel mai echilibrat arbore în cazul cel mai defavorabil.