Proiect de laborator
STRUCTURI DE DATE

Cordun Diana-Alexandra

Postolache Andreea-Miruna

Sandu Raluca-Ioana

Universitatea București

Facultatea de Matematică și Informatică

Anul 1, seria 14, grupa 142

1 Motivația alegerii structurii de date

TREAP-ul este un arbore binar de căutare echilibrat, ce combină arborele binar și heap-ul. Prin intermediul TREAP-ului, se pot realiza o multitudine de funcții în mod eficient: se poate accesa poziția oricărui element în O(log N), se pot adăuga sau șterge elemente, se pot afla succesorul și predecesorul unui nod, se poate afla al k-lea element. Toate acestea se pot face mai ușor prin SPLIT și JOIN, operații specifice acestei structuri de date. De aceea, am considerat că TREAP-ul este cea mai bună alegere.

2 Avantaje

TREAP-ul are printre caracteristici prioritățile, ceea ce reprezintă un real avantaj. Fără ele, această structură de date ar fi un arbore binar de căutare obișnuit, ceea ce ar însemna că ar fi mult mai lent, având complexitatea O(N) pentru mai multe funcții.

3 Dezavantaje

Implementarea este puțin mai dificilă decât pentru un arbore binar obișnuit, neechilibrat, iar bug-urile pot fi mai greu de găsit. De aceea, am testat codul pe măsură ce l-am scris.

4 Analiza de timp a programului

Prioritatea pe care o dăm nodurilor, împreună cu valoarea acestora, fixează în mod unic modul în care arată TREAP-ul , existând o singură variantă de a-l construi. Dat fiind faptul că alegem prioritățile aleatoriu, TREAP-ul va avea adâncimea unui arbore binar, care este log N. Așadar, SPLIT și JOIN, fiind ambele funcții recursive ce coboară în TREAP, au fiecare complexitatea O(adâncimea TREAP-ului), adică O(log N), unde N este numărul de elemente existente în total în arbore. Toate celelalte operații se reduc la un număr constant de SPLIT-uri și JOIN-uri, deci complexitatea structurii de date implementate de noi rămâne aceeași, O(log N).

1. Pentru funcția Este_in se apelează funcția Find, care parcurge TREAP-ul recursiv până când se găsește valoarea căutată. Dacă se găsește, întoarce 1, altfel 0. Complexitatea acestei funcții este O(log N) pentru că nu se parcurge întreg TREAP-ul.

2. Pentru funcția de inserare se apelează funcția Insert, care verifică dacă există deja valoarea în treap(O(1)). Dacă nu există acea valoare se creează un nod nou (O(1)). Pentru a insera, mai întâi se face split după valoarea care se vrea a fi inserată (O(log N)). Se face join între valorile mai mici decât valoare inserată și valoarea inserată, iar rezultat join-ului face join cu valorile mai mari decât valoarea inserată.(O(log N)). Astfel, această funcție are complexitatea O(log N).

3. Pentru funcția de ștergere se apelează funcția Delete, care face split pentru valoare care se dorește a fi ștearsă (O(log N)). Se face un split pentru (valoare -1) în mulțimea valorilor mai mici sau egale cu valoarea respectivă. Se șterge nodul(O(1)) și se face join (O(log N)) între arborele care are valori mai mici decât valoarea ștearsă și cel care are valori mai mari decât valoarea ștearsă. Astfel, complexitatea acestei funcții este O(log N) .

4. Pentru funcția care calculează care este elementul de pe o poziție dată (k_element) se apelează funcția GetKthElement care face două split-uri (2O(log N)), salvează valoarea elementului de la poziția dată într-o variabilă, dacă există (O(1)), altfel întoarce o excepție. La final, se fac doua join-uri (2O(log N)) pentru a reface treap-ul. Astfel, complexitatea acestei funcții este O(log N) .

5. Pentru funcția de minim se apelează funcția k_element pentru prima poziție. Cum complexitatea funcției apelate este O(log N) atunci și funcția de minim are aceeași complexitate.

6. Pentru funcția de maxim se apelează funcția k_element pentru ultima poziție. Cum complexitatea funcției apelate este O(log N) atunci și funcția de maxim are aceeași complexitate.

7. Pentru funcția succesor se apelează funcția PositionOfElement care returnează care este poziția la care se află valoarea introdusă. Funcția auxiliară folosită apelează și ea o altă funcție, GetOrderOfElement, care face un split (O(log N)), după care face o operație în O(1), iar la final un join O(log N) pentru a reface treap-ul. Astfel, funcțiile GetOrderOfElement și PositionOfElement au fiecare complexitatea O(log N). După ce iese din funcția apelată ,PositionOfElement , succesor apelează funcția k_element care are la rândul ei complexitatea O(log N). Astfel, funcția succesor are complexitatea O(log N).

8. Pentru funcția predecesor se apelează funcția PositionOfElement care returnează care este poziția la care se află valoarea introdusă. Funcția auxiliară folosită apelează și ea o altă funcție GetOrderOfElement care face un split(O(log N)), după care face o operație în O(1), iar la final un join O(log N) pentru a reface treap-ul. Astfel, complexitatea funcțiilor GetOrderOfElement și PositionOfElement este O(log N) . După ce a ieșit din funcția apelată, PositionOfElement, predecesor apelează funcția k_element care are la rândul ei complexitatea O(log N). Astfel, funcția predecesor are complexitatea O(log N).

9. Pentru funcția cardinal se testează dacă există rădăcina, iar dacă există atunci întoarce greutatea (O(1)), altfel întoarce 0. Astfel, funcția cardinal are complexitatea O(1).

5 Descrierea modului de testare

Pentru a testa programul, am creat o clasa nouă pentru a verifica dacă valorile obținute în urma rulării funcțiilor din clasa de bază corespund cu cele obținute în urma rulării lor în clasa creată (FakeArbore - care este ca o copie pentru clasa de bază și ale cărei rezultate sunt sigure). De asemenea, clasa nou creată are implementată o mulțime de numere de tipul cerut (aceasta fiind doar pentru verificare am putut folosi stl-ul set).

Am creat o funcție care primește ca parametri numărul de operații pe care dorim să le facă, cât și probabilitățile pentru a-i permite să modifice și să insereze. Această funcție generează random testele pe care le face și pentru funcții generează numere random (pentru ștergere, maxim, minim și pentru funcția de căutare ne-am folosit de o funcție scrisă în clasa FakeArbore, care generează un număr random din valorile pe care le avem în treap; pentru al k-lea element a generat o poziție random din treap; pentru funcția de căutare a unui element poate genera un număr random din valorile pe care le avem în treap sau un număr random de tipul cerut, dar care are probabilitatea foarte mare să nu se găsească în TREAP-UL construit). Pentru fiecare funcție în parte se testează dacă rezultatele obținute în clasa FakeArbore corespund cu cele obținute în clasa de bază. Dacă s-a obținut același lucru, atunci apare imediat pe ecran mesajul corespunzător funcției care a fost apelată și timpul de execuție în secunde. În caz contrar, se afișează atât ceea ce se aștepta să primească și ce a primit, cât și operația unde a întâmpinat problema și afișează structura FakeArborelui.

Totodată, am creat și o funcție care citește din fișier și afișează în fișier. În fișierul de intrare se scrie manual numărul de operații care se dorește a fi făcut și pe următoarele

rânduri se scriu manual caracterele corespunzătoare funcțiilor cerute (+ pentru inserare, - pentru ștergere, min pentru minim, max pentru maxim, > pentru succesor, < pentru predecesor, ! pentru al k_element, Card pentru cardinal, ? pentru este_in), acestea fiind urmate sau nu de o valoare (la funcțiile cardinal, minim, maxim nu se trece nicio valoare în dreapta caracterului). Pe consolă se va returna timpul în care s-au făcut acele teste. În fișierul de ieșire se vor afișa operațiile făcute, împreună cu mesaje corespunzătoare.

În plus, s-au generat teste care să afișeze în cât timp se execută o operație (Inserare, Stergere, Min, Max, Succesor, Predecesor, k_element, Cardinal și Este_în). Testele sunt: InserareStergere, InserareMin, InserareMax, InserareSuccesor, InserarePredecesor, InserareK_element, InserareCardinal, InserareCautare. Pentru fiecare se generează un număr random de operații de acel tip (numele funcției indică tipurile de operații efectuate). Corpul operațiilor din funcție este preluat din funcția GenerateRandomTest. Pentru a face operațiile a trebuit să se folosească de fiecare dată funcția de inserare. Mai mult, s-a creat o funcție InserareRandom care poate doar să insereze valori și pentru fiecare astfel de test afișează timpul în care a fost executat. În urma testelor efectuate, s-a putut remarca faptul că timpul pentru a insera un element este de aproximativ 0.001 s, iar pentru a șterge între 0.0005 s - 0.03 s. Pentru cazul în care se generează 10 teste pentru inserări de 200 de elemente, timpul mediu pentru cele 10 teste este de 0.1 s. De asemenea, am testat și pentru testele generate pentru un număr dat de operații și în care programul își alege random operațiile pe care le face. Pentru cazul în care sunt generate 110 operații pentru un test, timpul de execuție este cuprins aproximativ între 0.04 s - 0.088 s.

În cazul în care sunt:

- 110 teste, fiecare a câte 110 operații, timpul de rulare pentru 6 rulări este de : 8.536 s, 8.644 s, 9.086 s, 9.144 s, 9.325 s, 9.857 s. Astfel, timpul mediu de rulare este de 9.098 s (s-a calculat manual media aritmetică a valorilor obținute mai sus).

-210 teste a câte 110 operații, timpul de rulare pentru 6 rulări este de :13.175 s,13.151 s, 13.820 s, 13.767 s, 13.634 s, 13.595 s. Astfel timpul mediu aproximativ de rulare este de 13.524 s (s-a calculat manual media aritmetică a valorilor obținute mai sus). - 310 teste a câte 110 operații timpul de rulare pentru 6 rulări este de: 18.422 s 18.507 s, 18.469 s ,18.670 s, 18.285 s, 20.319 s. Astfel timpul mediu aproximativ de rulare este de 18.779 s (s-a calculat manual media aritmetică a valorilor obținute mai sus).

- 410 teste a câte 110 operații timpul de rulare pentru 6 rulări este de: 22.416 s 22.795 s, 22.640 s, 23.149 s, 22.429 s, 22.927 s. Astfel timpul mediu aproximativ de rulare este de 22.726 s (s-a calculat manual media aritmetică a valorilor obținute mai sus).

Mai mult, am testat și pentru testele generate pentru un număr dat de operații și în care programul își alege random operațiile pe care le execută. Pentru cazul în care sunt generate 210 operații pentru un test, timpul de execuție este cuprins aproximativ între 0.093 s - 0.17 s. Pentru cazul în care sunt generate 510 operații pentru un test, timpul de execuție este cuprins aproximativ între 0.43 s - 0.59 s. Pentru cazul în care sunt generate 1100 de operații pentru un test, timpul de execuție este cuprins aproximativ între 0.8 s -1 s. Pentru cazul în care sunt generate 20000 de operații pentru un test, timpul de execuție este cuprins aproximativ între 8 s -10 s.

Astfel, putem remarca faptul că deși structura primește un număr foarte mare de funcții pe care trebuie să le execute, aceasta reușește să le facă într-un timp foarte scurt. Pentru un număr mai mic sau egal cu 1100 de operații pentru un test, programul se execută în mai puțin de o secundă. Acesta ajunge să facă 20000 de operații pentru un test într-un timp foarte scurt, de tipul secundelor (8 s -10 s). De asemenea, toate funcțiile implementate sunt foarte rapide, timpul de execuție pentru fiecare în parte fiind mai mic decât 0.1 s.

6 Sales pitch

Doriți să aveți o structură de date care să vă permită să inserați, ștergeți elemente, să găsiți valoarea minimă, maximă, succesorul, predecesorul, să găsiți elementul de pe o poziție dată, cardinalul sau să verificați dacă un element este în mulțime? Al cărei timp de rulare pentru fiecare funcție să fie cât mai mic? Atunci, TREAP este soluția!

Poate vă întrebați de ce e așa... Ei bine, această structură de date este una rapidă, reușind să facă fiecare dintre operațiile mai sus menționate într-un timp foarte scurt (mai mic decât 0.1 s).

Aceasta poate executa mai multe operații foarte rapid: de exemplu, pentru 1100 de operații programul se execută în mai puțin de 1 secundă. Eficiența acestei structuri se datorează faptului că nodurile din TREAP au și o prioritate, care este dată random, dar și faptului că în implementare s-au utilizat operațiile specifice TREAP-ului, anume: SPLIT și JOIN care au complexitatea O(log N).

Acestea două stau la baza construirii operațiilor mai sus menționate, ceea ce garantează că și ele au complexitatea minim O(log N).

Nu uitați, dacă doriți un cod eficient și rapid, programul nostru este cel mai avantajos, fiind atât un arbore binar, cât și un heap!