Tema - Analiza Algoritmilor

Doboș Claudiu-Florin 323CD

Universitatea Politehnică București

joi, 13 decembrie 2018

Cuprins

1 Introducere

1.1 Descrierea problemei rezolvate

Am ales să mă ocup de problema cozilor de prioritate, si anume de Treap si Max-Heap.

1.2 Exemple de aplicații practice pentru problema aleasa

Cozile de prioritate sunt folosite in sistemele de operare, pentru ordonarea operațiilor de IO, ordinea in care un server procesează comenzile primite, in jocuri pentru afișarea mesajelor pe UI care trebuie sa fie afișate după ce userul a închis mesajul anterior.

1.3 Specificarea soluțiilor alese

Am ales sa mă ocup de Treap si de MaxHeap, voi implementa pentru fiecare operațiile de adăugare, obținere si ștergere a elementului maxim, MaxHeap a fost implementat cu ajutorul unui vector alocat dinamic, iar Treap a fost implementat ca un arbore, folosind pointeri.

Am implementat operațiile specifice unei structuri de date abstracta coada de prioritate (insert, getMax, deleteMax).

Treap-ul este un arbore binar de căutare echilibrat, el respecta atât proprietatea de arbore binar de căutare (pentru chei) cat si pe cea de heap (pentru priorități). Prioritățile sunt alocate aleatoriu, prin aceasta se dorește echilibrarea arborelui, deoarece este o probabilitate mica ca prin alocarea aleatorie a priorităților arborele sa fie dezechilibrat (numărul arborilor echilibrați este mai mare decât numărul arborilor rău echilibrați).

1.4 Criteriile de evaluare pentru validarea soluțiilor

Am generat 10 teste pentru a evalua corectitudinea soluțiilor. Testele au fost generate cu ajutorul programului *generator.c.* Acest program primește ca argumente numărul maxim de operații de insert, ordinea de inserarea a acestora(crescătoare, descrescătoare sau aleatorie), ordinea in care se vor pune operațiile de input si de get/delete, la alegere dintre sortare/random/get. La sortare, se pun mai întâi toate inserturile, după care urmează o secvența de get si delete. In cazul random, se pun de 2 ori mai multe inserturi decât operații de get sau delete iar ordinea acestora este aleatorie. La cazul de get, se inserează toate elementele, iar apoi urmează operațiile de getMax.

Pentru a valida corectitudinea acestor algoritmi am comparat rezultatele prezentate de aceștia, atât intre ei, cat si cu rezultatele produse de algoritmul quicksort implementat deja in C (pentru problema de sortare).

2 Prezentarea soluțiilor

2.1 Descrierea modului in care funcționează algoritmii aleși.

Pentru a implementa ADT-ul coada de priorități folosind Treap, am folosit o structura de date Node care retine informația utila (in cadrul acestei implementări un int), o prioritate aleasa aleatoriu, pointeri la fii săi (drept si stâng) si un pointer la părintele lui, in cazul in care acesta exista.

Operațiile implementate sunt insert, getMax si deleteMax.

La operația de insert creez un nod nou, ii aloc o prioritate aleatorie, găsesc poziția pe care ar trebui sa se afle conform proprietății de arbore binar de căutare (fiul stâng este mai mic decât părintele, fiul drept este mai mare ca părintele), îl inserez pe acea poziție după care verific si modific arborele prin rotiri stânga/dreapta astfel încât sa fie respectata si proprietatea de heap (valorile priorităților sa fie mai mici sau egale cu cea a părintelui.). Am implementat funcția recursiv, iar aceasta întoarce noua rădăcina a arborelui, in cazul in care ea se va modifica.

Operația getMax returnează elementul din nodul cel mai din dreapta al arborelui, deoarece acolo se afla elementul cu valoarea maxima.

Operația deleteMax se poziționează pe elementul cel mai din dreapta, îl șterge si reface legăturile, daca nodul avea fiu stâng.

Pentru implementarea ADT-ului MaxHeap binar m-am folosit de reprezentarea acestuia cu ajutorul unui vector, țin minte cate elemente am in vector si capacitatea acestuia. In reprezentarea cu ajutorul vectorului, copilul stâng se afla la poziția (2*i+1), copilul drept la poziția (2*i+2), iar părintele la ((i-1)/2). In implementarea mea, elementele pot fi doar int-uri.

Operațiile implementate sunt insert, getMax si deleteMax.

La operația de insert pun elementul pe ultima poziție din vector, după care fac operația de SiftUp (mutarea elementului din părinte in părinte pana când proprietatea de heap este respectata: părintele elementului curent este mai mare decât elementul curent).

La operația de getMax pur si simplu întorc primul element din vector (elementul de pe poziția 0), deoarece acesta este maximul din heap.

La operația de deleteMax mut ultimul element din vector in locul elementului maxim, decrementez mărimea vectorului si fac operația de SiftDown (daca unul dintre copii este mai mare ca elementul curent, interschimb elementul curent cu elementul maxim dintre copii) pentru a respecta din nou proprietatea de MaxHeap.

2.2 Analiza complexității soluțiilor

1) Treap

Insert: Aceasta operație presupune parcurgerea arborelui in întregime, pana la poziția unde va fi inserat elementul, ca fiu al unei frunze al arborelui. Știind ca arborele este cel mai probabil echilibrat, înălțimea acestuia este aproximativ log(N). Adăugarea efectiva a nodului in arbore se desfășoară in timp constant, iar pentru refacerea proprietății de heap e posibil sa trebuiască sa aplicam rotații asupra arborelui, in cel

mai rău caz, in care nodul nou inserat ar avea prioritatea maxima, ar trebui sa efectuam un număr de rotații egal cu înălțimea arborelui, adică tot log(N). In concluzie, putem spune ca operația de inserare in Treap are o complexitate ce aparține clasei O(log(N)).

GetMax: Cum am spus mai sus, arborele este probabil echilibrat, trebuie sa ajungem la frunza cea mai din dreapta, deci vom avea de parcurs de fiecare data toata înălțimea arborelui, deci complexitatea acestei operații aparține de θ(log(N)).

DeleteMax: Pentru a șterge maximul, va trebui mai întâi sa îl găsim, cu ajutorul operației de mai sus, deci parcurgem toata înălțimea arborelui, care se realizează in θ(log(N)), după ce găsim maximul mai avem nevoie doar sa refacem niște legături, operații ce se desfășoară in timp constant. In concluzie, complexitatea acestei operații aparține de θ(log(N)).

2) MaxHeap

Insert: Aceasta operație presupune adăugarea elementului la final si realizarea operațiunii de SiftUp. Adăugarea la final se realizează in timp constant, astfel complexitatea operației de insert e data de complexitatea operației de SiftUp. In cazul cel mai defavorabil elementul o sa ajungă pe prima poziție, executând-se aproximativ log(N) operații. Cel mai defavorabil caz e acela al elementelor ordonate crescător, iar cel mai favorabil este când elementele sunt ordonate descrescător, nefiind astfel necesara realizarea operației de SiftUp. Astfel complexitatea operației de insert este de O(log(N)), dar pe un caz normal vor fi efectuate puține operații de SiftUp, astfel încât complexitatea operației de insert se va apropia de a fi constanta. Când se ajunge la capacitatea vectorului, acesta este realocat cu dublul capacității anterioare. Astfel nu este afectata complexitatea algoritmului, deoarece costul amortizat al operației de insert va rămâne același.

GetMax: Aceasta operație se realizează in timp constant, deoarece se returnează pur si simplu elementul aflat pe poziția 0. Complexitatea acestei operații este de θ(1).

DeleteMax: Aceasta operație presupune înlocuirea elementului de pe prima poziție cu elementul de pe ultima poziție din vector, scăderea dimensiunii si realizarea operației de SiftDown. Având in vedere ca primele doua operații se executa in timp constant, complexitatea operației de deleteMax este data de operația de SiftDown. Având in vedere ca înălțimea arborelui este de aproximativ log(N), complexitatea acestei operații va fi O(log(N)).

2.3 Principalele avantaje si dezavantaje ale soluțiilor alese

1) Treap:

Avantaje:

- anumite operații pe care nu le-am implementat se efectuează mai rapid decât cu un heap (căutarea unui element, îmbinarea a 2 treap-uri).
- se poate folosi atât ca o coada pentru Max cat si pentru Min, daca se implementează ambele seturi de operații.
- operația de deleteMax se executa mai repede decât la Heap.

Dezavantaje:
- Consuma de doua ori mai mult spațiu decât un MaxHeap (trebuie sa retina si o prioritate).
- Operațiile de GetMax si Insert sunt mai lente fata de Heap.
- Implementare mai dificila (lucru cu pointeri)

2) MaxHeap:
Avantaje:
- Insert si GetMax rapide.
- Implementare rapida
- Ideala pentru reprezentarea unei cozi de prioritate

Dezavantaje:
- Operația de deleteMax nu e la fel de eficienta ca la Treap.
- Performante slabe ale operației de Insert in cazul cel mai defavorabil.

3 Evaluare

3.1 Descrierea modului de realizare a setului de teste

Testele au fost generate cu ajutorul programului aflat in generator.c, argumentele trimise programului se afla in documentul in/descriereteste. O descriere mai detaliata a acestui program a fost făcută in secțiunea 1.4. Am construit 10 teste cu dimensiunea input-ului cuprinsa intre 20000 si 200000, cu ordinea de input variind intre aleatorie, crescătoare si descrescătoare. Am construit un test suplimentar, test10, in care dimensiunea input-ului este de 10^7, iar timpii de rulare au fost prea mari pentru a justifica crearea mai multor teste cu dimensiuni comparabile.

3.2 Specificațiile sistemului de calcul pe care s-a realizat testarea

Procesor: Intel Core i7 7700HQ
Memorie: 16 GB RAM DDR4, 2400 MHz

3.3 Rezultatele evaluării soluțiilor

Rezultatele pentru operația insert:

Număr operați	Ordinea datelor	Timp Treap (s)	Timp Heap (s)
20000	Aleatorie	0.011236	0.008798
100000	Aleatorie	0.069206	0.019180
200000	Aleatorie	0.166806	0.040994
10000000	Aleatorie	19.806792	1.872983
20000	Crescătoare	0.009608	0.008514

100000	Crescătoare	0.074456	0.030776
200000	Crescătoare	0.25630	0.067614
20000	Descrescătoare	0.009046	0.004663
100000	Descrescătoare	0.068225	0.016330
200000	Descrescătoare	0.236370	0.032770

Rezultatele pentru operația deleteMax:

Număr operații	Ordinea datelor	Timp Treap (s)	Timp Heap (s)
20000	Aleatorie	0.002275	0.007071
100000	Aleatorie	0.016474	0.036535
200000	Aleatorie	0.045121	0.079546
10000000	Aleatorie	2.343536	6.559156
20000	Crescătoare	0.002810	0.005181
100000	Crescătoare	0.035744	0.028313
200000	Crescătoare	0.142449	0.065338
20000	Descrescătoare	0.002185	0.005852
100000	Descrescătoare	0.010373	0.032151
200000	Descrescătoare	0.025612	0.065822

Rezultate pentru operația getMax:

Număr operații	Ordinea datelor	Timp Treap (s)	Timp Heap (s)
20000	Aleatorie	0.003631	0.003381
100000	Aleatorie	0.019546	0.014880
200000	Aleatorie	0.032657	0.027622

3.4 Prezentarea valorilor obținute

Se poate observa ca pentru operația de Insert, MaxHeap-ul este mult mai eficient decât Treap (chiar de 10 ori mai rapid pentru un input suficient de mare), chiar si in cazul cel mai defavorabil pentru MaxHeap, când inputul este ordonat crescător. Se poate observa ca performanta de operației de insert pe Treap nu este afectata de ordinea datelor de input, spre deosebire de MaxHeap, unde cazul defavorabil este de aproximativ doua ori mai lent fata de cel mai favorabil si de 1.5 ori mai lent fata de cazul cu input in ordine aleatorie.

La operația deleteMax se observa faptul ca Treap-ul este, in general, mai rapid ca MaxHeap-ul. La aceasta operație apare o valoare neașteptata, acea de la testul cu 200000 operații de delete cu input ordonat crescător, unde timpul de execuție este mai mare decât la MaxHeap. Probabil aceasta valoare apare din cauza ca treap-ul nu a fost echilibrat bine.

La operația getMax, se observa cum pe Heap se executa intr-un timp mai scurt decât pe Treap, ceea ce era de așteptat, având in vedere complexitatile celor doua operații. Aici timpul este mai mare decat ne-am fi asteptat, comparativ cu operatiile

de delete si insert, deoarece in cadrul operației de get afișam numărul, ceea ce necesita timp.

4 Concluzii

In concluzie, in practica, strict pentru problema cozilor de prioritate, as implementa un MaxHeap, pentru ca are un timp de insert si getMax mai scurt fata de treap, chiar daca deleteMax durează mai mult pe Heap.

As folosi treap când nu mă interesează timpul de insert, deoarece estimez ca voi avea mai multa nevoie de o operație de delete mai rapida sau daca as mai avea nevoie si de alte operații, cum ar fi căutarea unui element, merge intre 2 treap-uri, etc.

5 Referințe

1. Drozdek, A. (2012). *Data Structures and algorithms in C++*. Cengage Learning.
2. LNCS Homepage, http://www.springer.com/lncs, accesat ultima data la 2018/11/19.
3. Laborator curs SD, Treap, https://ocw.cs.pub.ro/courses/sd-ca/laboratoare/lab-10, accesat ultima data la 2018/12/3.
4. Laborator curs SD, Heap, https://ocw.cs.pub.ro/courses/sd-ca/laboratoare/lab-09, accesat ultima data la 2018/12/3.