Tema
-Analiza algoritmilor-

Buliga Theodor Ioan
323 CA
Facultatea de Automatica si Calculatoare
Universitatea Politehnica Bucuresti

9 ianuarie 2023

Rezumat

Lucrarea studiaza comparatia dintre doua structuri de date folosite pentru a reprezenta API-ul specific unei cozi de prioritate. Structurile alese sunt maxheapul (ca heap binar) si treapul (ca arbore binar de cautare echilibrat).

1 Introducere

Coada de prioritate reprezinta extensia unei cozi normale, avand in plus urmatoarele elemente:

-Fiecare element din coada de prioritate este asociat cu o prioritate.

-Elementul cu cea mai mare prioritate este primul element care trebuie eliminat din coadă.

-Dacă mai multe elemente au aceeași prioritate, atunci se ia în considerare ordinea lor în coadă.

Coada de prioritate este o versiune modificata a cozii, modificarea fiind ca atunci cand un element urmeaza sa fie scos din coada, este preluat in functie de prioritatea sa.

Coada de prioritate se poate folosi in multe probleme, cum ar fi:

-Cel mai dens interval

-Radix Sort

-Heap sort

-Algoritmul lui Prim

-Algoritmul lui Dijkstra

-Compresia datelor

-In cadrul sistemelor de operare, la planificarea firelor de executie, in cadrul algoritmului Round-Robin de planificare de exemplu

De asemenea, coada de prioritate se poate folosi si pentru situatii din viata reala in inteligenta artificiala sau situatii precum prioritizarea intrarii pacientilor la o consultatie.

Problema pe care am ales sa o rezolv este prioritizarea intrarii pacientilor la o consultatie. Am presupus ca un spital are nevoie de un sistem de gestiune in cazul cozilor de la urgente. In coada de

pacienti, vor avea prioritate cei care sunt raniti mai grav. In cazul meu, gravitatea accidentul va fi reprezentata de un numar. Cu cat este mai mare numarul, cu atat este mai mare si urgenta. Astfel, pacientii nu vor intra neaparat conform primului venit primului servit, ci in functie de cat de rau sunt afectati.

Lucrarea va compara, deci, doua structuri de date diferite folosite pentru a reprezenta coada de prioritate.

Am ales pentru reprezentarea API-ului specific cozii de prioritare maxheapul(heap binar) si treapul(arbore de cautare echilibrat).

Cele doua structuri vor fi evaluate dupa timpul de executie, memoria utilizata(implicit si complexitate) si dupa gradul de dificultate al implementarii pentru programator.

2 Prezentarea solutiilor

Pentru a putea intelege mai bine cele doua structuri de date folosite, voi defini intai notiunea de arbore binar. Din punct de vedere matematic, un arbore este un graf neorientat, conex si aciclic. In stiinta calculatoarelor, arborele este o structura ce respecta definitia de mai sus, dar are asociate un nod radacina si o orientare inspre sau opusa radacinii. Arborele binar reprezinta un caz paricular de arbore, in care fiecare nod poate avea maxim doi descendenti, respectiv un nod stang si un nod drept.

2.1 MaxHeap

Un max-heap binar este un arbore binar in care fiecare nod are proprietatea ca valoarea sa este mai mica sau egala cu cea a parintelui sau.
Am ales sa implementez maxheapul sub forma unui vector declarat global. Astfel, elementul cu prioritatea maxima va coincide cu radacina maxheapului, respectiv H[0]. In continuare, pentru a accesa, respectiv stoca elementele, se vor folosi urmatoarele reguli:

- H[(i - 1) / 2] intoarce nodul parinte
- H[i * 2 + 1] intoarce nodul copil stang
- H[i * 2 + 2] intoarce nodul copil drept

unde i reprezinta indicele nodului curent, indexarea fiind facuta de la 0, unde H[0] reprezinta radacina maxheapului.

Pentru a modulariza codul si a lucra cat mai eficient, am folosit functii ajutatoare. Trei dintre ele intorc indicele unui nod cautat, dupa algoritmul indicat mai sus, iar urmatoarele doua sunt folosite pentru crearea maxheapului.

Pentru a adauga un element nou la maxheap, el se adauga la orice nod frunza, iar apoi va urma operatia de shiftUp in cazul in care nodul adaugat este mai mare decat parintele, iar invariantul de maxheap nu se mai respecta. Astfel, se vor face interschimbari intre elemente, "in sus", pana cand elementul adaugat va ajunge la pozitia corecta.

Pentru a ajunge la elementul maxim, doar apelam H[0], insa pentru a il elimina, am folosit functia de shiftDown ce va interschimba nodurile cu copiii cu valori mai mari, pentru a mentine proprietatile maxheapului.

Avand in vedere implementarea folosita, au rezultat urmatoarele complexitati:

-pentru adaugarea unui element se va obtine O(Log n) pe cazul general, deoarece urmeaza operatiile necesare mentinerii proprietatilor maxheapului, insa daca elementul adaugat este mai mic decat nodul frunza la care a fost atasat, se va obtine O(1), avand la polul opus O(n) pe cazul cel mai nefavorabil

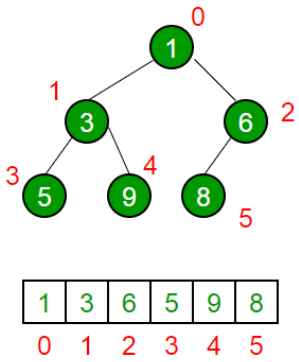

Figura 1: O reprezentare vizuala pentru a intelege mai bine stocarea.

-pentru stergerea unui element se va obtine tot O(Log n) pe cazul general, din aceleasi motive ca mai sus, dar daca se doreste stergerea unui nod frunza al subarborelui drept, se poate obtine tot O(1)

-pentru obtinerea elementului maxim se va obtine O(1)

-pentru eliminarea elementului maxim se va obtine pe cazul general tot O(log n)

unde n reprezinta numarul de elemente.

Utilizarea maxheapului in implementarea unei coze de prioritate este avantajoasa deoarece aflarea maximului se face foarte rapid (O(1)), se pot adauga elemente noi sau sterge elemente existente relativ usor dupa crearea maxheapului, O(log n) fiind destul de rapid, memoria utilizata nu este foarte mare (n * dimensiunea datelor stocate, in cazul de fata intregi), iar implementarea cu un vector este relativ usor de inteles, vizualizat, scris si per total mai putin de munca pentru cel care programeaza. In implementarea mea, memoria alocata este ceva mai mare, deoarece am decis sa lucrez cu un set de date mai numeros, dar in principiu, memoria alocata pentru maxheap este 50 000 * sizeof (long long) in cazul de fata. Stiu ca puteau exista optiuni mult mai blande cu memoria, respectiv alocare dinamica, insa mi s-a parut mult mai usor sa lucrez asa.

2.2 Treap

Treapurile sunt arbori binari de cautare echilibrati, cel mai des folositi datorita implementarii relativ usoare (prin comparatie cu alte structuri similare cum ar fi AVL-uri sau B-trees), dar si a modului de operare destul de intuitiv.
Fiecare nod din treap retine doua campuri:

-cheia - informația care se reține în arbore și pe care se fac operațiile de inserare, căutare și ștergere

-prioritatea - un număr pe baza căruia se face echilibrarea arborelui

Structura trebuie sa respecte doua proprietati:
-Proprietatea de arbore binar de căutare → binary search tree (tr): cheia unui nod va fi mai mare sau egală decât cheia fiului stânga, dacă există şi mai mică sau egală decât cheia fiului dreapta, dacă există. Cu alte cuvinte o parcurgere inordine a arborelui va genera șirul sortat de chei.

-Proprietatea de heap (eap): prioritatea unui nod este mai mare sau egală decât prioritățile fiilor.

Inserarea intr-un treap este asemanatoare cu cea pentru un arbore binar de cautare. Nodul se adauga

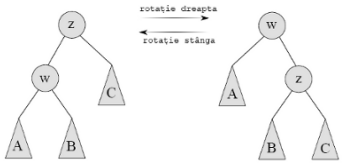

Figura 2: O reprezentare vizuala pentru a intelege mai bine cele doua tipuri de rotatii.

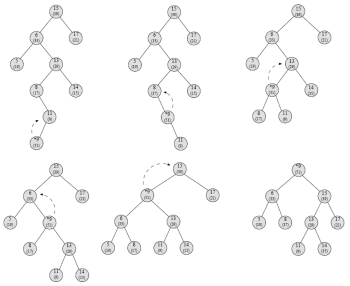

Figura 3:

la radacina, urmand sa ajunga apoi la baza printr-o procedura recursiva, in functie de prioritatea sa.

Deși inserarea menține invariantul arborelui de căutare, invariantul de heap poate să nu se mai respecte. De aceea, trebuie definite operații de rotire (stânga sau dreapta), care să fie aplicate unui nod în cazul în care prioritatea sa este mai mare decât ce a părintelui său.

Spre exemplu, dacă am dori să inserăm nodul cu cheia 9 şi prioritatea 51, pașii vor arata ca in figura 3.

Operația de ștergere este inversul operației de inserare și se aseamăna foarte mult cu ștergerea unui nod în cadrul unui heap. Nodul pe care îl dorim a fi șters este rotit până când ajunge la baza arborelui, iar atunci este șters. Pentru a menține invariantul de heap, vom face o rotire stânga dacă fiul drept are o prioritate mai mare decât fiul stâng și o rotire drepta în caz contrar.

Din nou, elementul cu prioritatea cea mai mare se va afla la radacina.

In implementare, am definit o structura de nod ce contine un camp pentru cheie si unul pentru prioritate, respectiv doi descendenti. Am folosit apoi doua functii ajutatoare pentru rotatii, o functie de inserare si una de stergere.
In urma implementarii folosite, s-au obtinut urmatoarele complexitati:

-pentru adaugarea unui element se va obtine O(Log n) pe cazul general

-pentru stergerea unui element se va obtine tot O(Log n) pe cazuk general

-aflarea prioritatii maxime se va obtine in O(Log n), fiind totusi nevoie de o parcurgere a treapului pana la cel mai din dreapta element

Implementarea folosind treap este avantajoasa deoarece aflarea maximului se face rapid (O(Log n)), se pot adauga si sterge elemente rapid (O(Log n)) dupa ce a fost creat treapul (durand O(n log n), fiind vorba de n elemente), in schimb memoria utilizata este ceva mai mare, deoare e nevoie de pointeri catre nodurile copii, fiind mai costisitor si din punct de vedere al timpului (rotatiile dureaza mai mult). La nivel conceptual e mai dificil de inteles, nemaifiind vorba de un simplu vector, fiind si ceva mai dificil de scris si vizualizat.

Totusi, in implementarea aleasa ma astept sa se obtina o complexitate mai mare, avand in vedere

Treap	Maxheap
0.000175	0.000285
0.000186	0.000283
0.000114	0.000262
0.000135	0.000218

Tabela 1: Testele 1-4.

faptul ca sunt mai multe date ce tin de un treap decat de un maxheap. Inclusiv, in cadrul rotatiilor sunt mai multe operatii, deci automat complexitatea va creste. Rotatiile vor fi mai costisitoare decat interschimbarile (2 operatii in plus), adaugarea unui nod va fi si ea mai costisitoare, va trebui creat nodul (inca patru operatii + memorie suplimentara alocata). Insertia nu consider ca va fi cu mult diferita, la fel si stergerea, insa vor exista eliberari de memorie (+ 1 operatie), precum si alte operatii mentionate mai sus care vor duce la un timp de executie mai crescut.

3 Evaluarea solutiilor

Am gandit testele astfel incat sa verifice treptat diferite aspecte.

Primele teste vor verifica viteza de inserare, in trei cazuri: datele de intrare sunt ordonate crescator, descrescator, iar in cele din urma aleatoriu. Urmatoarele vor verifica aceleasi lucruri, dar pentru date multe mai mari si mai numeroase.

Urmatoarele teste vor verifica viteza de stergere a elementului maxim, de data asta pe date generate complet aleatoriu, aici fiind vorba doar de eficienta structurii in sine, fiindca in ambele cazuri maximul va fi radacina.

Am adaugat si niste de dimensiuni foarte mari pentru a vedea concret diferentele si cat de mult variaza si acestea.
O parte din teste vor testa si viteza de aflare a maximului, dar nu consider ca este foarte relevanta si ca ar trebui sa ma axez pe asta.

In cadrul fiecarui test se va analiza si memoria folosita, pe langa timpul de executie.
Testele au fost rulate pe un laptop HP:

-procesor: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz

-RAM instalat: 16,0 GB (15,9 GB folosibl)

-tipul sistemului: 64-bit operating system, x64-based processor

Testele au fost rulate manual, au fost facute si afisarile acolo unde a fost nevoie, iar pe langa datele de iesire cerute am mai adaugat si timpii de rulare ai algoritmirilor implementati.

Main-urile celor doua structuri de date sunt identice, difera doar functiile si cateva mici alocari.

Pentru primele patru teste, unde am lucrat cu numere mici, doar cu operatii de adaugare au reiesit urmatorii timpi:

Pentru urmatoarele trei teste care au urmat aceeasi regula, dar au avut seturi mai lungi de date au reiesit urmatoarele rezultate:

Urmatorul set de teste cauta maximul din numerele date, pentru un set relativ lung, urmarind aceeaasi regula de mai sus pentru introducere: descrescator, crescator, aleatoriu. In continuare, am testat

Treap	Maxheap
0.001691	0.001155
0.001690	0.000900
0.002459	0.000990

Tabela 2: Testele 5-7.

Treap	Maxheap
0.001496	0.001007
0.002537	0.001509
0.002632	0.001081

Tabela 3: Testele 8-10.

stergerea elementului maxim, iar apoi afisarea imediata pentru seturi relativ mari de date(2000).
Aici au aparut niste probleme la ultimele teste, respectiv 16 si 17, cred ca din cauza memoriei si posibil a implementarii.

Am test, apoi, introducerea unui numar foarte mare de date (10 000+) in ordine descrescatoare, aleatoare si crescatoare.
Dupa cum se observa, in testele 1-4, unde am folosit un set mic de date(20 de numere), treapul a fost mult mai rapid, deoarece rotatiile au costat mai putin timp decat interschimbarile in cazul shifUp-ului.

In schimb, la un set mai mare de date, operatiile in cadrul treapului s-au dovedit mai costisitoare decat in cadrul maxheapului, cred ca datorita modului in care se face stocarea.

Pentru gasirea elementului maxim, in mod evident maxheapul s-a dovedit mult mai rapid, elementul maximul gasindu-se in O(1), iar la seturi de date lungi insertia in treap dureaza mai mult.

In testele 11-15, maxheapul s-a dovedit mult mai eficient, operatia de stergere durand ceva mai mult in cadrul treapului, avand in vedere felul in care sunt stocate datele.

In testele 18-25 se observa clar ca maxheapul este mult mai eficient, maximul fiind din nou mult mai usor de gasit, insertia realizandu-se mult mai usor.

In testele 16 si 17 au aparut niste erori de memorie, pe care din nefericire nu am reusit sa le rezolv, se poate observa si in rezultate, cred ca am gresit in cadrul implementarii.

Am mai adaugat 2 teste pentru a verifica performanta pentru seturi de date scurte, iar rezultatele se pot observa in tabel.

Treap	Maxheap
0.002628	0.001034
0.002715	0.001145
0.001867	0.001202
0.001795	0.000631
0.001833	0.002939
-	0.002671
-	0.002510

Tabela 4: Testele 11-17

Treap	Maxheap
0.013610	0.003591
0.160798	0.012353
0.035257	0.016363
0.163030	0.018310
0.170468	0.012446
0.059127	0.015084
0.019182	0.009591
0.015899	0.005864

Tabela 5: Testele 18-25.

Treap	Maxheap
0.000250	0.000218
0.000226	0.000336

Tabela 6: Testele 26 si 27.

4 Concluzie

In concluzie, implementarea cu maxheap este mult mai rapida pentru numere mai mari si mai multe. Statistic, luand in considerare felul in care am lucrat, exista diferente foarte mari la nivel de timp. Maxheapul s-a dovedit chiar si de 2 ori mai rapid decat treapul. Din punctul meu de vedere, maxheapul este si mai usor de implementat, fiindca am lucrat mai mult cu vectori decat cu arbori si e o structura de date pe care o inteleg si o stapanesc mult mai bine. Din punct de vedere al gestiunii memoriei mi s-a parut mult mai simplu, nefiind nevoie sa mai eliberez memorie sau sa ma chinui cu alocarea si am gestionat mult mai usor erorile aparute.

Cu siguranta, daca ar fi sa lucrez cu o coada de prioritate intr-un mediu in care se proceseaza multe date sau in cadrul unor algoritmi complecsi, as alege sa lucrez cu un maxheap.

Totusi, fiind vorba de sistemul de gestiune al unui spital, tinand cont ca vorbim de primiri urgente, iar in general numarul de urgente este mic, as alege sa lucrez cu un treap in cadrul problemei pe care o rezolv. In cadrul unor teste de lungime redusa, generate aleator, treapul a iesit mai rapid, nu cu foarte mult ce e drept, dar a fost mai eficient. Clar, a fost mai dificil de implementat si de inteles la nivel conceptual, ba chiar au aparut niste erori pe care nu am reusit sa le rezolv, insa pentru o probleme care tina de viata reala aleg sa lucrez cu un treap. Pe langa avantajele mentionate anterior, se pot adauga si multe alte functionalitati care ar avea o complexitate mai mica decat in cazul maxheapului.

As vrea sa adaug ca memoria alocata pentru maxheap nu este folosita in toate cazurile, dar mi s-a parut cea mai rapida solutie, avand in vedere ca am fost putin pe graba. La treap, am lucrat ceva mai eficient, insa dupa cum am spus deja, codul mai trebuie imbunatatit.

5 Bibliografie

Laboratoare de SD:
https://ocw.cs.pub.ro/courses/sd-ca/laboratoare/lab-08
https://ocw.cs.pub.ro/courses/sd-ca/laboratoare/lab-09
https://ocw.cs.pub.ro/courses/sd-ca/laboratoare/lab-10

Alte siteuri:
https://www.educative.io/answers/min-heap-vs-max-heap
https://alexdremov.me/treap-algorithm-explained/
https://ro.myservername.com/priority-queue-data-structure-c-with-illustration
https://andrei.clubcisco.ro/1sd/curs/curs08.pdf
https://www.geeksforgeeks.org/binary-heap/

https://www.educative.io/answers/min-heap-vs-max-heap
https://stackoverflow.com/questions/21219691/advantages-of-a-binary-heap-for-a-priority-queue
https://www.geeksforgeeks.org/priority-queue-using-binary-heap/
https://www.quora.com/Why-does-implementing-a-priority-queue-using-a-heap-lead-to-better-performance-than-both-an-ordered-and-unordered-array-despite-the-fact-that-a-heap-uses-an-array-itself-Explain-with-justification
https://www.techiedelight.com/implementation-treap-data-structure-cpp-java-insert-search-delete/
https://cp-algorithms.com/data$_s tructures/treap.htmlimplementation$
$https://www.techiedelight.com/implementation-treap-data-structure-cpp-java-insert-search-delete/$
$http://www.cs.ubbcluj.ro/\ gabitr/Cursul10.pdf$
$https://www.quora.com/What-are-the-applications-of-the-priority-queue$
$geeksforgeeks.org/applications-priority-queue/$
$https://www.techcrashcourse.com/2016/02/c-program-to-find-execute-time-of-program.html$
$https://www.geeksforgeeks.org/implementation-of-search-insert-and-delete-in-treap/$
$https://alexdremov.me/treap-algorithm-explained/$