Analiza Algoritmilor - Structuri de Date - Cozi de Prioritate

Pescaru Tudor-Mihai 321CA

[1] Universitatea Politehnica Bucureşti, Bucureşti, RO
[2] Facultatea de Automatică şi Calculatoare
tudor_mihai.pescaru@stud.acs.upb.ro

Keywords: Heap · AVL · Coadă de Prioritate.

1 Introducere

1.1 Descrierea problemei rezolvate

Problema rezolvată in cadrul acestui studiu de caz este aceea a implementarii unei cozi de prioritate. Coada de prioritate este un **tip de date abstract**, similar cu coada sau stiva clasica, in care, fiecarui element îi este asociată o **prioritate**, iar elementele cu prioritatea cea mai mare, respectiv cea mai mică, după implementare, sunt procesate primele. În cazul unei priorităţi egale, elementele se procesează în ordinea în care au fost introduse.

1.2 Exemple de aplicaţii practice pentru problema aleasă

În cazuri reale, cozile de prioritate sunt folosite în diverse aplicaţii. Algoritmi celebrii precum **Algoritmul lui Dijkstra** pentru găsirea drumului minim, **Algoritmul lui Prim**, **Algoritmul A* Search** sau Algoritmul de sortare **Heap Sort**. De asemenea, cozile de prioritate sunt folosite în compresia datelor, fiind implementate în **codurile Huffman** dar şi în cadrul **Sistemelor de Operare** pentru gestionarea proceselor, distribuirea sarcinilor sau gestionarea interrupturilor.

1.3 Specificarea soluţiilor alese

Pentru implementarea unei cozi de prioritate, am ales să studiez două moduri diferite de implementare, prin două tipuri de structuri de date, **Heap** şi **Arbori Binari Echilibraţi**. Pentru Heap, am ales implementarea de **Binary Heap**, implementata prin intermediul unui vector, iar pentru Arborele Binar Echilibrat am ales implementarea unui **AVL**. În cadrul acestor implementari, prioritatea va fi dată de valoarea elementului din coadă, iar prioritatea/valoarea mai mică va fi cea favorizată.

1.4 Specificarea criteriilor de evaluare alese pentru validarea soluţiilor

Pentru validarea soluţiilor, am realizat seturi de teste de dimensiuni diferite, compuse din înlanţuiri de comenzi relizate asupra implementarii de coada de prioritate. Comenzile vor fi de tip **inserare**, **ştergerea elementului minim**, si **interogarea pentru obţinerea elementului minim**. Testele vor conţine seturi de instrucţiuni, ordonate aleator si un numar aleator pentru a obţine un mediu de testare apropiat de utilizarea în context real. De asemenea vor exista şi teste ce vor acoperii *"edge-cases"* pentru ambele implementări, pentru a determina comportamentul acestora în scenarii de tip *"best-case"* si *"worst-case"*.

2 Prezentarea soluţilor

2.1 Descrierea modului in care funcţioneaza algoritmii aleşi.

AVL: Conform menţiunii anterioare, pentru implementarea de coadă de prioritate prin intermediul unui arebore binar echilibrat, am ales utilizarea unui AVL. La bază, aceasta structura de date este un arbore binar obişnuit, căruia i se adauga proprietatea definitiorie, prin care diferenta de înalţime dintre cei doi subarbori ai oricărui nod este cel mult 1. Pentru a realiza această operaţie de rebalansare, se definesc două tipuri de operaţii, numite rotaţii. Aceste rotaţii se pot realiza spre dreapta sau spre stânga si afecteaza un nod si cei doi subarbori ai săi. Pentru a realiza inserarile se va face o inserare ca într-un Binary Search Tree, după care se realizează rotaţiile pentru a asigura menţinerea proprietăţii. Similar, ştergerea este şi ea similară celei pentru Binary Search Tree, la care se adaugă rotaţiile pentru rebalansare. Pentru a obţine valoarea minimă se va accesa nodul cel mai îndepărtat de pe subarborele cel mai din stânga.

Heap: Pentru implementarea bazată pe Heap, am ales folosirea unui Binary Heap implementat printr-un vector, folosind implementarea de vector din C++ STL, deoarece acesta permite redimensionare dinamică. Acest tip de implementare este cel mai comun in practică fiind chiar utilizat pentru implementările "built-in" din limbaje precum C++ sau Java. In cadrul acestei implementări, deorece folosim un prioritatea cea mai mică, heap-ul va fi de fapt un Min Heap. Acest lucru înseamnă ca proprietatea de heap va asigura faptul că pentru orice element, părintele său va fi mai mic. Astfel, putem deduce faptul că elementul minim va fi în vârful heap-ului, adică pe prima poziţie din vector. Pentru păstrarea acestei proprietăţi de heap se vor defini două operaţii, numite operaţii de "sifting" sau "cernere". Aceste operaţii sunt de doua tipuri, de sus în jos şi de jos în sus, şi vor fi folosite pentru ştergere şi respectiv inserare. Operaţia de inserare constă în adaugarea unui element la final si de a-l interschimba repetat cu părintele acestuia, cât timp acesta este mai mare decât părinte, interschimbări ce sunt realizate de operaţia de cernere. Pentru ştergerea elementului minim, se interschimbă primul element cu ultimul si se realizează cernerea de sus în jos,

interschimbând primul element cu cel mai mic copil al său până când nu mai are copii mai mici.

2.2 Analiza complexităţii soluţiilor.

Deoarece asceste structuri au la baza modele arborescente, binare, operaţiile se vor reliza într-un timp logaritmic. Pentru **AVL**, complexitatea temporala este aceeaşi pentru toate operaţiile, atât în cazul "average" dar şi în "worst case" este de O($logn$). Din punct de vedere al spaţiului ocupat, complexitatea va fi de O(n) deoarece se vor stoca cel mult n noduri de dimensiune constantă. În cazul **Heap-ului**, datorită faptului că este implementat printr-un vector, iar elementul minim este pe poziţia 0, obţinerea acestuia are o complexitate temporală de $\Theta(1)$. Operaţia de inserare are o complexitate temporala de O($logn$) iar operaţia de stergere are o complexitate de $\Theta(logn)$. Din punct de vedere al spaţiului, heap-ul are şi el o complexitate de O(n). O variantă mai eficientă dar cu o implementare seminificativ mai complexă, este Fibonacci Heap ce beneficiază de complexitate $\Theta(1)$ pentru inserare si obţinerea minimului şi de o complexitate amortizată de O($logn$) pentru ştergere.

2.3 Prezentarea prinicpalelor avantaje şi dezavantaje pentru soluţiile luate în considerare.

Principalul avantaj al implementarii pe bază de **Heap**, faţa de cel pe bază de **AVL** este timpul constant de accesare faţă de cel logaritmic. Un alt avantaj al implementării pe bază de Heap este implementarea mai uşoară de realizat. De asemenea, fără modificări considerabile asupra implementării, AVL-ul nu suportă introducerea de valori duplicate. Un dezavantaj al implementării pe bază de Heap este faptul că, în cazul unei implementări prin "array" clasic, dimensiunea este setată la început iar redimensionarea este costisitoare din punct de vedere al timpului de execuţie.

3 Evaluare

3.1 Descrierea modalitaţii de construire a setului de teste folosite pentru validare.

Pentru a genera testele am creeat un script în **Python**, ce va creea **10 teste**. Testele au fost concepute urmărind formatul de testare descris în documentaţia online de pe **site-ul GCC**. Acest format de testare este utilizat pentru testarea performanţei structurii de date priority queue din STL. Testele sunt de diferite dimensiuni, aceste dimensiuni reprezentând numărul de valori ce vor fi adăugate în priority queue. Aceste dimensiuni sunt de **10k, 100k şi 1m**. La acest număr presatbilit, se adaugă şi un număr aleator de operaţii de obţinere a minimului şi de ştergere a minimului. Acest număr aleator este dat de faptul că la fiecare operaţie de inserare există o şansa de 50% să apară şi o operaţie de alt

tip. Testele au fost grupate în perechi, fiecare pereche având un test în care se află doar operaţii de inserare si un numar aleator de operaţii de obţinere a minimului şi un test în care apar în plus şi un număr aleator de operaţii de ştergere a minimului. Primele 4 teste sunt realizate folosind o însiruire de valori predictibile, primele 2 având valori consecutive în oridne crescătoare, iar următoarele 2 având valori consecutive în ordine descrescătoare. Următoarele 6 teste au valori aleatoare generate în intervalul [0, dimensiunea testului]. Primele 2 teste sunt menite să ofere un scenariu în care inserările şi ştergerile nu necesită operaţii extra de corectare a aşezării elementelor, în timp ce următoarele două vor necesita o operaţie de corectare după fiecare inserare sau ştergere. Restul testelor oferă o simulare cât mai apropiată de o utilizare in viaţa reală, comportamentul fiind impredictibil. Testele au fost generate o singură dată şi au fost apoi rulate folosind ambele implementări. În rezultatele testelor pot apărea mici discrepanţe între cele două moduri de implementare deoarece AVL nu suportă duplicate în timp ce Heap suportă.

3.2 Specificaţiile sistemului de calcul pe care au fost rulate testele.

Testele au fost rulate pe un sistem de calcul cu următoarele specificaţii:

- **Host OS:** Linux Mint Ulyana 20.04 x86_64 cu Linux Kernel 5.4.0-58-generic
- **Procesor:** Intel(R) Core(TM) i7-8565U 4 core, 8 thread @ 1.80 GHz base frequency - 4.60 GHz turbo frequency
- **Cache size:** 8MB
- **RAM:** 15827MB

3.3 Ilustrarea rezultatelor evaluării soluţiilor pe setul de teste.

Pentru măsurarea timpului de rulare a testelor am utilizat biblioteca **chrono**, iar pentru măsurarea memoriei utilizate am folosit biblioteca **sys/resource.h**. Timpul de rulare a fost măsurat în **ms** iar memoria utilizată în **kB**. Folosind **Octave**, au fost generate următoarele grafice care conturează timpul de rulare al celor două metode de implementare pentru fiecare test (Fig. 1) şi spaţiul ocupat de cele două metode pentru fiecare test (Fig. 2). Aceste măsurători au fost obţinute în urma realizării unei medii a rezulatelor obţinute pe mai multe iţeratii de testare.

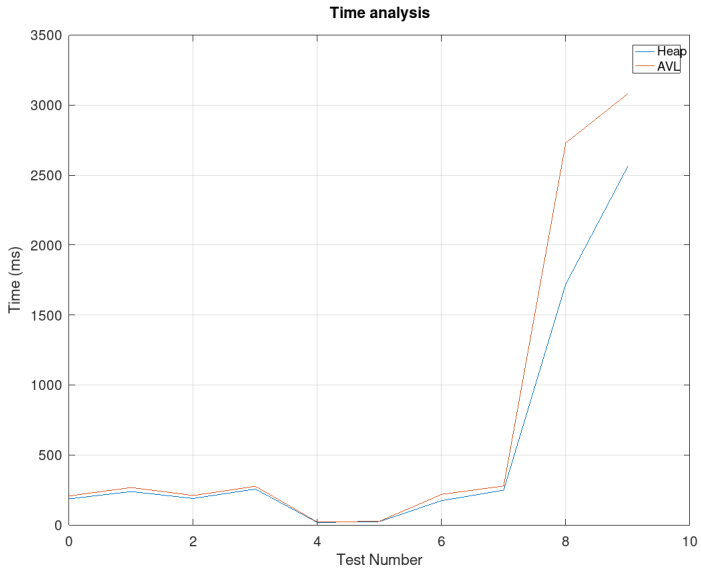

Fig. 1. Analiză pe baza timpului de rulare al testelor.

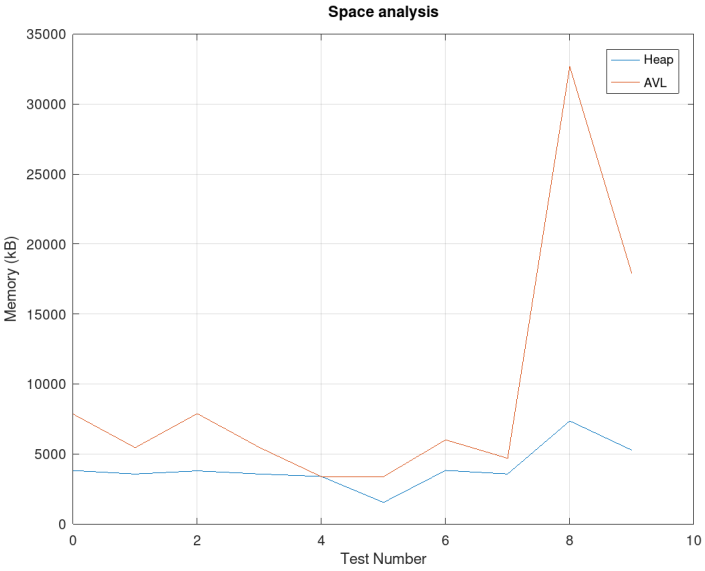

Fig. 2. Analiză pe baza spaţiului ocupat pe parcursul testelor.

3.4 Prezentarea valorilor obţinute pe teste.

După cum se poate observa din graficele ataşate, atât din punctul de vedere al timpului dar şi al spaţiului ocupat, implementarea ce se bazează pe Heap este mai eficientă decât cea care se bazează pe AVL. De asemenea implementarea este şi ea mai uşoară în cazul Heap-ului decât în cazul AVL-ului. Un mod de a îmbunătăţii parţial performanţele implementării pe bază de arbore echilibrat ar fi utilizarea unui Red-Black Tree în locul AVL-ului. De asemenea, implementarea pe bază de heap ar putea să fie şi ea îmbunătăţită prin utilizarea unui Fibonacci Heap în locul unui Binary Heap.

4 Concluzii

În concluzie, având în vedere performanţele superioare dar şi simplitatea de implementare, utilizarea unui Binary Heap pentru un priority queue este cea mai bună variantă. Acestă afirmaţie este confirmată şi de utilizarea acestei metode de implementare de priority queue în bibliotecile native ale mai multor limbaje de programare populare.

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: Introduction to Algorithms 3rd edn. The MIT Press, Cambridge, Massachusetts (2009)
2. Wikipedia Priority Queue, https://en.wikipedia.org/wiki/Priority_queue
3. GeeksforGeeks Priority Queue, https://www.geeksforgeeks.org/priority-queue-set-1-introduction/
4. GeeksforGeeks AVL, Set 1 - Insertion, https://www.geeksforgeeks.org/avl-tree-set-1-insertion/?ref=lbp
5. GeeksforGeeks AVL, Set 2 - Deletion, https://www.geeksforgeeks.org/avl-tree-set-2-deletion/?ref=lbp
6. GeeksforGeeks Fibonacci Heap, https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/?ref=lbp
7. GeeksForGeeks Binary Heap, https://www.geeksforgeeks.org/binary-heap/
8. Testing performance of a priority queue, https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/pq_performance_tests.html