Temă Analiza Algoritmilor

Dinu Ion-Irinel

Universitatea Politehnică din Bucuresti
Facultatea de Automatică Şi Calculatoare, 325CA
ion_irinel.dinu@stud.acs.upb.ro

Abstract. Comparaţia între două structuri de date care pot fi utilizate pentru implementarea unei cozi de prioritate (AVL si Heap) .

Keywords: AVL · Heap · Structuri de date

1 Introducere

1.1 Descrierea problemei rezolvate

O coadă de prioritate reprezintă o colecţie în care fiecare element are asociată o prioritate care determină ordinea în care elementul urmează să fie inserat. Elementul prioritar va fi plasat in varful cozii si tot acesta va fi primul extras din coadă. In cadrul problemei alese, este necesară implementarea unor operaţii specifice unei astfel de cozi de prioritate precum adăugarea unui nou element, ştergerea şi obţinerea elementului cu valoare maximă.

1.2 Aplicaţii practice pentru problema aleasă

O aplicaţie practică care se bazează pe o coadă de prioritate este reprezentată de CPU Scheduling, un algoritm care este utilizat în cadrul sistemelor de operare. Acesta se ocupă cu programarea proceselor în funcţie de prioritate. Dacă două procese cu aceeaşi prioritate sunt gata pentru lansare, algoritmul se foloseşte de conceput *FIFO* (First in First Out) şi îl execută pe primul venit. În acest tip de algoritm de planificare, dacă sosește un proces mai nou, care are o prioritate mai mare decât procesul în curs de execuție, actualul proces în curs de execuție este interzis. De asemeea, cozile de prioritate sunt folosite şi in realizarea unor algoritmi precum:

- Algoritmi specifici grafurilor(Dijkstra's Shortest Path)
- Compresia datelor(Huffman Codes)
- Inteligenţa Artificială(A* Search Algorithm)
- Gestionarea laţimii de bandă a traficului de reţea(Bandwidth management)

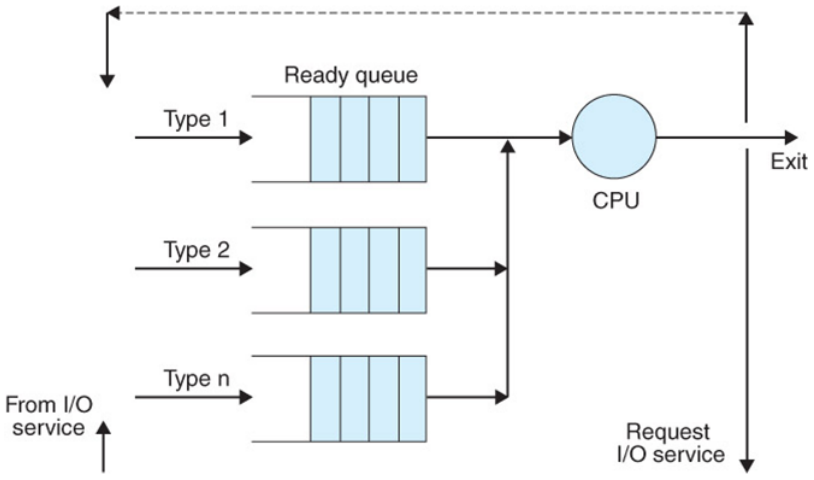

Fig. 1. Un exemplu de Multi-level Priority Queue CPU Scheduling [2]

1.3 Specificarea soluţiilor alese

In cadrul rezolvării problemei, am decis să analizez două structuri de date prin care poate fi implementată o coadă de prioritate: AVL-ul si un Max Heap.
Cele doua structuri de date prezinta următoarele proprietăţi:

- Ambele reprezintă tipuri de arbori binari
- Operaţiile de inserare şi ştergere au complexitate logaritmică
- Ambele pot realizeaza o cautare eficientă a elementului cu valoare maximă. In cadrul unui Max Heap elementul maxim se va afla in rădăcina heap-ului, iar intr-un AVL elementul cu valoare maximă se va afla in ultimul nod din subarborele drept.

Deoarece timpul pentru o căutare reprezintă un factor foarte important in cadrul problemei abordate, probabilitatea să găsim elementul dorit cu valoarea maximă va creşte cu timpul de execuţie al programului realizat.

1.4 Criteriile de evaluare pentru soluţia propusă

Criterii de evaluare pentru soluţia propusă vor fi reprezentate atat de eficienţa fiecaruia algoritm din punct de vedere al timpului de execuţie, cât si din punct de vedere al complexităţii specifice. Pentru a evalua corectitudinea algoritmilor implementaţi mi-am propus sa realizez o serie de teste, după cum urmează:

- Testele[1-7] - vor verifica un numar redus de operaţii de bază:
  - inserare / obţinere element maxim
  - inserare / ştergere / obţinere element maxim
- Testele[8-15] - vor introduce un set mai mare de operaţii si se vor concentra mai mult pe operaţia de obţinere a elementului maxim:
  - inserare / obţinere element maxim
  - inserare / ştergere / obţinere element maxim
  - se vor introduce in coadă intre 0 - 10000 elemente
  - elementele inserare vor avea valori între 0 - 10000
- Testele[16-23] - vor fi generate automat si vor testa intr-un mod complex operaţiile implementate:
  - se vor introduce in coadă între 0 - 10000000 elemente
  - elemente inserate vor avea valori intre 0 - 1000000

Primele teste de bază vor fi realizate manual, iar pentru testele mai complexe cu un numar mare de operaţii voi realiza un program care va genera testele automat. De asemenea pentru a compara rezultatele obţinute cu cele care sunt corecte voi folosi un checker care verifică dacă cele două output-uri sunt identice.

Pentru a testa eficienţa fiecărui algoritm voi analiza pentru fiecare test numărul de operaţii implicate si timpul de execuţie in care se va întoarce rezultatul dorit.

2 Prezentarea soluţiilor alese

2.1 Descrierea modului in care functioneaza algoritmii alesi

Heap

Un Heap reprezinta o structura de date, un arbore binar in care valoarea prezenta in radacina este mai mare decat valorea tuturor copiilor. Aceasta proprietate se respecta in mod recursiv pentru fiecare subarbore din heap, practic fiecare nod are valoarea mai mica sau egala decat valoarea actuala a parintelui.

Din punct de vedere al implementarii un heap poate fi realizat atat folosind inlantuiri de structuri, legate intre ele prin pointeri left si right, acesta modalitatea avand avantajul flexibilitatii redimensionarii arborelui, cat si prin utilizarea unui vector si accesarea nodurilor prin intermediul unui index. Implementarea unui heap folosind un astfel de vector este mai populara si eficienta, avand avantajul eliminarii informatiei legate de pointeri, folosind o formula simpla pentru a determina fiecare nod din heap.

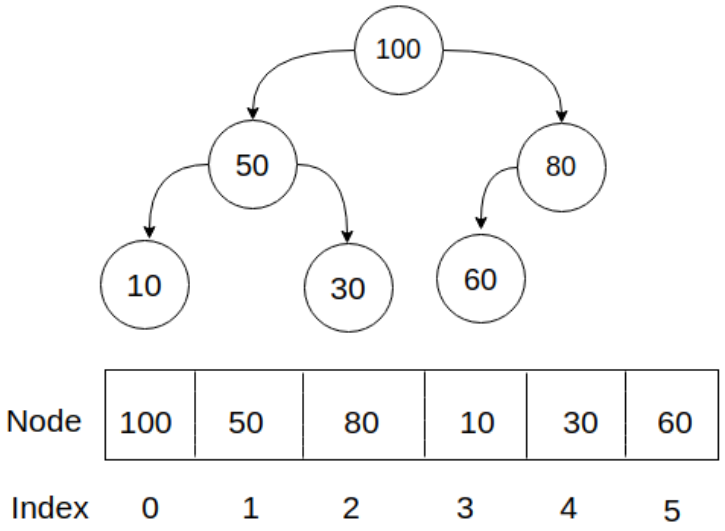

Fig. 2. Un exemplu de implementarea al unui Max Heap folosind vector

Implementarea liniara a heap-ului folosind un vector devine:

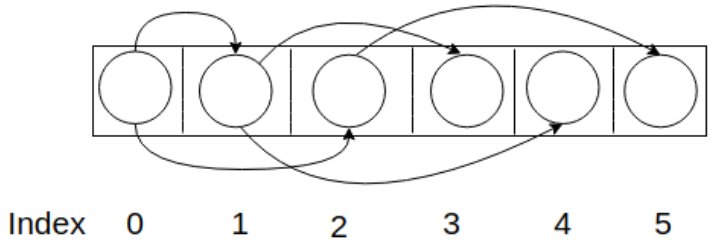

Fig. 3. Determinare nodurilor pe baza formulei de calcul a index-ului

Indexarea in cadrul vectorului se realizeaza incepand cu 0, care reprezinta pozitia radacinii iar apoi pentru fiecare nod se calculeaza pozitia descendentiilor si a parintelui dupa umatoarea formula:

- **Nodul Stang** = 2 * i + 1, unde i este index-ul nodului curent
- **Nodul Drept** = 2 * i + 2, unde i este index-ul nodului curent
- **Parinte** = (i - 1) / 2, unde i este index-ul nodului curent

Operatii specifice unui Max Heap
In cadrul implementarii operatiilor de inserare, stergere, accesare a unui element din Max Heap sunt necesare doua operatii prin intermediul carora sa se mentina propritatea de Max Heap:
Heapify Up
Operatia realizeaza interschimbarea valorii adaugate la sfarsitul vectorului sau modificata, cu valorile mai mari, indreptandu-se spre radacina, pentru a mentine propritatea de heap dorita.

```
heapify_up
   determining the parent of the node
   parent_node = parent(index)
   if array[parent_node] < array[index]
   exchange the array[parent_node], array[index]
   call heapify_up with parent_node
```

Heapify Down

Operatia presupune interschimbarea elementelor in urma modificarii unui nod din heap, noua valoare fiind mai mica decat valoarea descendentiilor, interschimbarea realizandu-se catre nivele de jos ale heap-ului pentru a mentine proprietatea de heap dorita.

```
heapify_down
   access the left and right children of the current node
   index of great_node is index
   if right < heap_size and array[right] > array[index]
       great_node is right

   if left < heap_size and array[left] > array[great_node]
       great_node is left

   exchange array[great_node] with array[index]
   call heapify_down with great_node
```

Add

Prin intermediul acestei operatii se adauga in nou element in cadrul heap-ului.

```
add(new value)
   add the new item to the last position
   array.push_back(value);
   index_size is array.size - 1
   call heapify_up with index_size
```

Remove Max Element

Prin intermediul acestei operatii se elimina elementul cu valoarea maxima din cadrul heap-ului.

```
remove_max()
   if heap is not empty
       exchange array[0] with array[size - 1]
       remove last element of the vector
       call heapify_down with index 0 for first element
```

Max Element
Prin intermediul acestei operatii se returneaza elementul cu valoarea maxima.

```
max_element()
   if the heap is not empty
   return array[0]
   return -1
```

AVL

AVL-ul este o structura de date, un arbore binar de cautarea, care se echilibreaza dupa fieacare operatie de stergere sau de inserare a unui nod. Acesta functioneaza dupa proprietatea ca dieferenta dintre doi subarbori ai unui nod are mereu valoarea maxim 1, iar reechilibrarea se realizeaza folosind rotatii duble.

Din punct de vederea al implementarii, pentru un AVL se folosesc structuri sau clase de pointeri care contin: un pointer catre nodul stang, un pointer catre nodul drept, valoare propriu zisa a nodului si inaltimea acestuia.

Operatii specifice unui AVL
Pentru a realiza operatii de stergere, inserare sau extragere a unui element sunt necesare o serie de operatii de rotatie prin intermediul carora se reechilibreaza arborele.

Rotate Right - daca factorul de echilibru este pozitiv si daca inaltimea subarborelui stang este mai mare decat cea a subarborelui drept se realizeaza o rotatie la dereapta.
Left Right Rotate - daca factorul de echilibru este pozitiv si daca inaltimea subarborelui stang este mai mica decat cea a subarborelui drept se realizeaza o rotatie la stanga si apoi o rotatie la dreapta.
Rotate Left - daca factorul de echilibru este negativ si daca inaltimea subarborelui stang este mai mica decat cea a subarborelui drept se realizeaza o rotatie la stanga.
Right Left Rotate - daca factorul de echilibru este negativ si daca inaltimea subarborelui stang este mai mare decat cea a subarborelui drept se realizeaza o rotatie la dreapta si apoi o rotatie stanga.

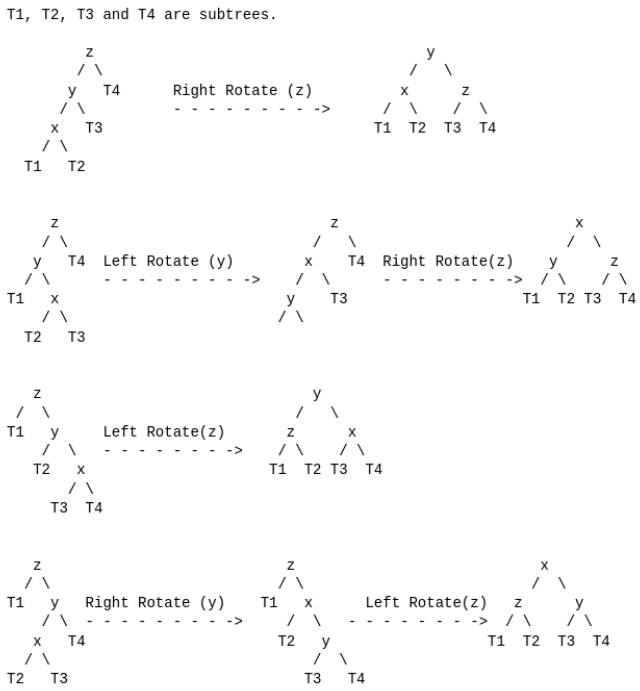

Fig. 4. Exemplu de tipuri de rotatie folosite pentru reechilibrarea AVL-ului [8]

Add

Inserare unui nod in cadrul AVL-ului se realizeaza in mod similar inserarii intr-un arbore binar de cautare. Se cauta pozitia inserarii comparand cheia nodului curent cu cheia nodului care urmeaza sa fie inserat, apoi se alege subarborele stang sau drept. Dupa ce am adaugat elementul se realizeaza echilibrarea avl-ului prin actualizarea inatlimiilor si aplicarea rotatiilor.

```
add(nod , key)

   if node == NULL
       node = create_node(key)

   else if node.key > key
       insert (node.left, key)

   else
       insert (node.right, key)

   nod.height = 1 + maximum(height(node.left), height(node.right))
   apply rotations if necessary
```

Delete
Operatia de stergere a unui element este asemanatoare cu stergerea unui element dintr-un arbore binar de cautare. Se cauta nodul cu cheia pe care dorim sa o stergem. Daca nodul nu are doi succesori, daca este frunza, se inlocuieste cu NULL iar daca nu este frunza se inlocuieste cu unul din descendentii sai nenuli. Altfel, daca nodul are doi succesori, il vom inlocui cu cel mai mic nod din subarborele drept. In final se reechilibreaza AVL-ul.

```
remove (node, key)
  if node == NULL
      return NULL

  if node.key < key
       remode (node.right, key)
  else if node.key > key
      remove (node.left, key)
  else
      if node.right == NULL
          node = node.left
      if node.left == NULL
          node = node.right
      else
          min_node_right = min_node(node->right)
          node.key = min_node_right.key;
          remove (node.right, min_node_right.key)

  nod.height = 1 + maximum(height(node.left), height(node.right))
  apply rotations if necessary
```

Delete Max Element
In cadrul acestei operatii se stergere elementul cu valoare maxima. Acesta se afla in ultimul nod din subarborele drept al AVL-ului.

```
remove_max()
  obtaining and deleting the maximum element
  node = max_node(root);
  remove(root, node.data);
```

Max Element
In cadrul acestei operatii se returneaza nodul care contine elementul maxim, aceasta fiind ultimul nod din subarborele drept.

```
max_node(root)
  current = root;
  while current.right != NULL
      current = current.right;
  return current;
```

2.2 Analiza Complexitatii solutiilor

Analiza complexitatii Heap-ului
1. Analiza Complexitatii de inserare in Max Heap
- complexitatea adaugarii unui nou nod este : O(1)
- complexitatea interschimbarilor (operatia heapify up) : O(h)
- complexitatea totala: O(1) + O(h) = O(h)
- in cadrul unui arbore binar complet, inaltimea este egala cu log N asadar complexitatea generala de inserare a unui nod intr-un Max Heap este **O(log N)**.

2. Analiza Complexitatii de stergere in Max Heap
- complexitata interschimbarii nodului frunza cu nodul parinte : O(1)
- complexitatea stergerii elementului de la final : O(1)
- complxitatea interschimbarilor (operatia heapify down) : O(h)
- complexitatea generala de stergere a unui nod din Max Heap este **O(log N).**

3. Analiza Complexitatii de accesare a elementului cu valoare maxima
- pentru a obtine valoare maxima se returneaza elementul din radacina heapului, iar acesta operatia are complexitate generala **O(1).**

Analiza complexitatii AVL-ului
1. Analiza Complexitatii de inserare in AVL
- inserarea unui nou element in cadrul arborelui necesita rotatii, calcularea factorului de echilibru si actualizarea inaltimii
- complexitate opearatiilor de rotatie : O(1)
- complexitatea actualizarii inaltimii : O(1)
- complexitatea parcurgerii inaltimii arborelui : O(log N)
- complexitatea generala de adaugare a unui nod este **O(log N).**

2. Analiza Complexitatii de stergere din AVL
- un mod similar operatia de stergere a unui nod necesita rotatii, calcularea factorului de echilibru si actualizarea inaltimii dupa stergere
- atat in cel mai defavorabil caz cat si in cel mai favorabil complexitatea generala de stergere a unui nod este **O(log N).**

3. Analiza Complexitatii de accesare a elementului cu valoare maxima
- pentru a determina elementul cu valoare maxima trebuia parcurs subarborele drept al AVL-ului
- in cazul cel mai defavorabil, atunci cand nodul cu elementul maxim este frunza complexitatea de accesaree a elementului maxim este **O(log N).**
- intr-un caz favorabil, in care elementul cu valoarea maxima se afla chiar in radacina, complexitatea este **O(1).**
- pe caz mediu complexitatea generala de accesare a elementului cu valoare maxima este **O(log N).**
In cel mai defavorabil caz, inaltimea unui AVL este 1.44 log N.

Aceasta formula se deduce astfel:
Fie Fh un arbore AVL de înălțime h, având numărul minim de noduri. Fie Fl și Fr arbori AVL care sunt subarborele din stânga și, respectiv, subarborele din dreapta ale lui Fh. Atunci Fl sau Fr trebuie să aibă înălțimea h-2. Să presupunem că Fl are înălțimea h-1, astfel încât Fr are înălțimea h-2. Reținem că Fr trebuie să fie un arbore AVL având numărul minim de noduri dintre toți arborii AVL cu înălțimea h-1. În mod similar, Fr va avea numărul minim de noduri dintre toți arborii AVL de înălțime h–2. Acesti arbori se numesc arbori Fibonacci.
$| F |=| Fh-1 | + | Fh-2 | +1$
$| F0 |= 1 | F1 |= 2.$
$|Fh| + 1 = (|Fh-1| + 1) + (|Fh-2| + 1)$
$| Fh | +1 \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3}$
h ≈ 1,44log Fn
Cel mai rar arbore AVL posibil cu n noduri are înălțime h 1.44log N
Cel mai rău caz de înălțime a unui arbore AVL cu n noduri este 1.44log N

MAX HEAP	**Worst Case**	**Average Case**
Space	O(N)	O(N)
Search Max Element	O(1)	O(1)
Insert	O(log N)	O(log N)
Delete	O(log N)	O(log N)

AVL	**Worst Case**	**Average Case**
Space	O(N)	O(N)
Search Max Element	O(log N)	O(log N)
Insert	O(log N)	O(log N)
Delete	O(log N)	O(log N)

Fig. 5. Tabel cu complexitatile celor doi algoritmi

2.3 Prezentarea principalelor avantaje si dezavantaje pentru solutiile luate in considerare

Avantajele Heap-ului
- Implementeaza operatiile de inserare si stergere in complexitate logaritmica
- Se poate folosi pentru găsirea eficientă a celui mai mic/mare element din heap
- Heap-ul este mereu un arbore echilibrat (complet)
- Datorita rapiditatii operatiilor, Heap-ul sta la baza implementarii unor algoritmi de sortare precum HeapSort care are o complexitate **O(N * log N)**

Dezavantajele Heap-ului
Un dezavantaj al Heap-ului este reprezentat de realizarea operatiei de cautare a unui element, care are o complexitate liniara O(n).

Avantajele AVL-ului
- principalul avantaj al AVL-ului este legat de faptul ca produce cel mai echilibrat arbore in cel mai defavorabil caz
- fiind perfect echilibrat, arborele AVL va scoate un timp foarte rapid pentru cautarea unui element

Dezavantajele AVL-ului
- un mare dezavantaj al AVL-ului este reprezentat de numarul mare de rotatii pe care il efectueaza pentru reechilibrare
- implementarea operatiilor si rotatiilor specifice unui AVL este complicata

La modul general cei doi algoritmi prezinta avantajele si dezavantajele enuntate mai sus, dar restragand studiul de caz la problema abordata, reprezentarea Heap-ului sub forma de Max Heap are marele avantaj al implementarii operatiei de accesare a elementului cu valoare maxima in complexitate constanta O(1) fata de AVL, pentru care acceasi operatie are complexitate O(log N). Din punct de vedere al implementarii, AVL prezinta un dezavantaj, deoarece implementarea acestuia este mai dificila decat cea a unui Max Heap folosind un vector STL.
Ca o consecinta, PriorityQueue in Java este implementat folosind un Max Heap.

3 Evaluare

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

Pentru fiecare test am creat un fisier de input care contine un numar de operatii specifice unei cozi de prioritate ce urmeaza sa fie testat si un fisier de output cu ajutorul caruia voi testa corectitudinea rezultatelor.
Testele de input sunt folosite pentru ambii algoritmi si sunt in numar de 23.

Pentru a genera testelor am realizat un script in python, care genereaza o serie de teste random pe baza unor critetii folosind modulul random.

Pentru realizare unui test sunt preluate de la tastatura o serie de date:
- numarul testului pe care dorim sa il cream
- numarul de elemente al liste de numere
- range-ul maxim al elementelor
- yes / no daca dorim o lista de numere sortata
- lista sortata - crescator / descrescator
- upper limit pentru numarul de operatii de push
- upper limit pentru numarul de operatii random

In cadrul generatorului, am creat o lista de cu cele trei comenzi. Apoi am creat pe baza datelor introduse de la tastatura o lista de numere, care poate fi sortata sau nu. Am creat un fisier pentru testul dorit, si am generat random un numar de operatii de push pe care le-am afisat in fisier. Apoi am generat random un numar de operatii dintre cele trei posibile. Valorile care sunt adaugate in coada sunt generate random din lista de numere creata initial.

Structura unui Test
- in cadrul fiecarui teste se afla o serie de comenzi
- daca numele comenzii este push se va citi si o valoare si se va realiza adaugarea valorii respective in coada (operatia push)
- daca numele comenzii este pop se va sterge elementul cu prioritatea maxima din coada (operatia pop)
- daca numele comenzii este top se va returna elementul cu prioritatea maxima din coada (operatia top)

Folder-ul in :
- contine 23 de teste de intrare care vor fi aplicate fiecarui algoritm
- fiecare test este numit "testX.in", unde X reprezintă numărul testului

Testele [1-8]
- verifica un numar redus de operatii
- testeaza corectitudinea generala a implementarii
- elementele au valori intre [0 - 1000]
- se introduc in coada intre [0 - 100] elemente

Testul [9]
- contine 1000 de operatii de push
- elementele sunt adaugate in ordine crescatoare
- elementele au valori intre [0 -1000]

Testul [10]
- contine 1000 de operatii de push
- elementele sunt adaugate in ordine descrescatoare
- elementele au valori intre [0 - 10000]

Testele [11-15]
contin intre 0-10000 de operatii
valorile elementelor sunt cuprine intre [0-10000]

Testele [16-18]

- contin intre [10000-99999] de operatii
- valorile elementelor sunt cuprinse intre [0-90000]

Testele [19-20]
- contin peste 100000 de operatii
- valorile elementelor sunt cuprinse intre [0-100000]

Testele [21-22]
- contin peste 1000000 de operatii
- valorile elementelor sunt cuprinse intre [0-1000000]
- testeaza intr-un mod complex algoritmii implementati

Testul[23]
- contine peste 10000000 de operatii
- valorile elementelor sunt cuprinse intre [0-1000000]

Pentru testele [11-23] mai intai se genereaza un numar random de operatii de push iar apoi se genereaza un numar random de operatii diferite.

Folder-ul out:
- contine trei foldere: best, p1 si p2 deoarece in cadrul testelor exista elemente cu aceeasi prioritate (duplicate), iar output-ul generat este diferit pentru cei doi algoritmi
- fiecare subfolder contine 23 de teste de iesire care reprezinta testele corecte ce ar trebui obtinute si care vor fi utilizate pentru testarea corectitudinii algoritmilor
- fiecare test este numit "testX.out", unde X reprezintă numărul testului

3.2 Specificatiile sistemului de calcul

- Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
- GeForce GTX 1650 TI
- 16 GB RAM
- Memory Available: 13.808488 GB
- Sistem de operare : Linux/Ubuntu

3.3 Ilustrarea rezultatelor evaluării solutiilor pe setul de teste

Pentru a determina timpul de efecutie al fiecarui algoritm, am creat doua programe folosind limbajul C++ care calculeaza durata specifica pentru fiecare test. Folosind biblioteca chrono, am setat un clock la inceputul rularii algoritmului si un clock la final, dupa care am afisat diferenta dintre cele doua durate.
Rezulatatele obtinute sunt ilustrate folosind un tabel si o serie de grafice reprezentative.

Numar Test	Timp Heap (microsecunde)	Timp AVL (microsecunde)	Diferenta de timp	Numar de operatii	Limita maxima a valorilor
1	92	94	2	4	100
2	51	48	3	7	100
3	57	64	7	9	100
4	59	76	17	5	100
5	56	81	25	10	100
6	87	92	5	16	100
7	77	63	7	17	100
8	78	90	12	89	1000
9	360	411	51	1001	1000
10	151	521	370	1001	10000
11	155	287	132	450	10000
12	902	2041	1139	4165	10000
13	1159	3254	2095	6236	10000
14	1570	3471	1901	7024	10000
15	2744	4866	2122	9997	10000
16	14200	24123	9923	48827	100000
17	16006	36567	20561	62462	100000
18	15076	47413	32337	70975	100000
19	39087	72247	33160	112233	100000
20	156184	572894	416710	903391	100000
21	459848	1287224	827376	122040	1000000
22	978517	8190510	7211993	6541213	1000000
23	4311754	10223953	5912199	12438494	1000000

Fig. 6. Tabelul de analiza al timpilor de executie

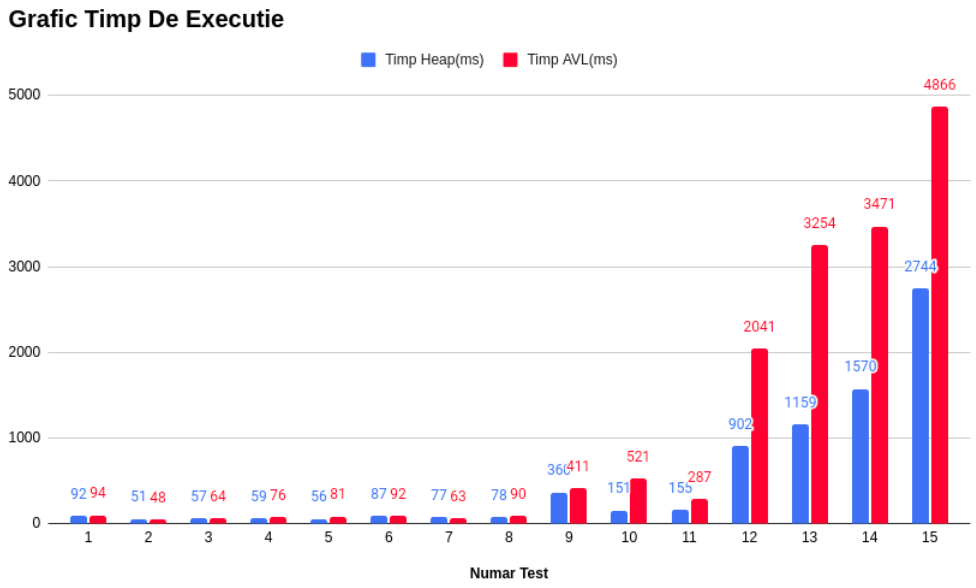

Fig. 7. Graficul timpului de executie pentru testele 1-15

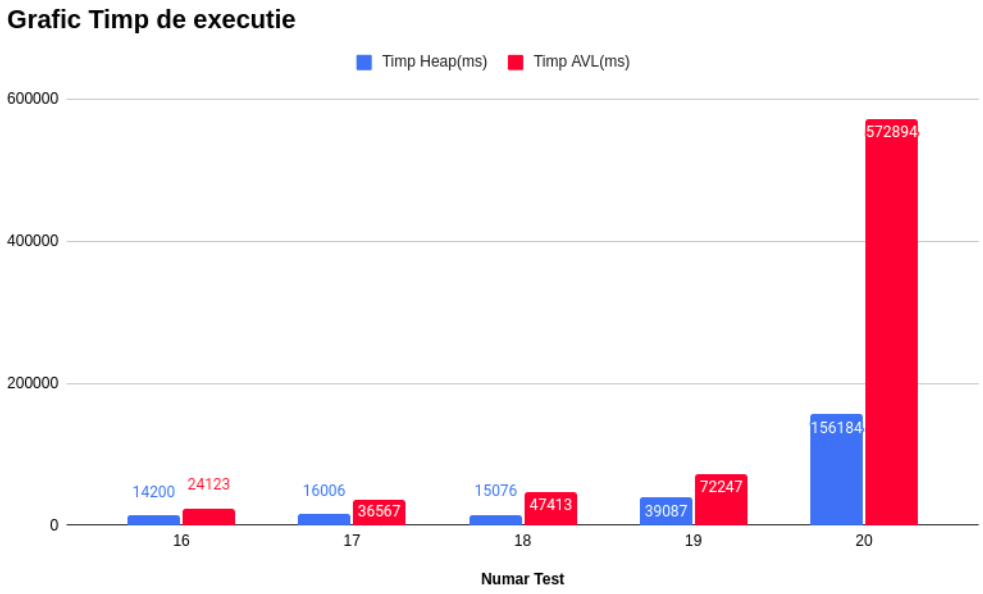

Fig. 8. Graficul timpului de executie pentru testele 16-20

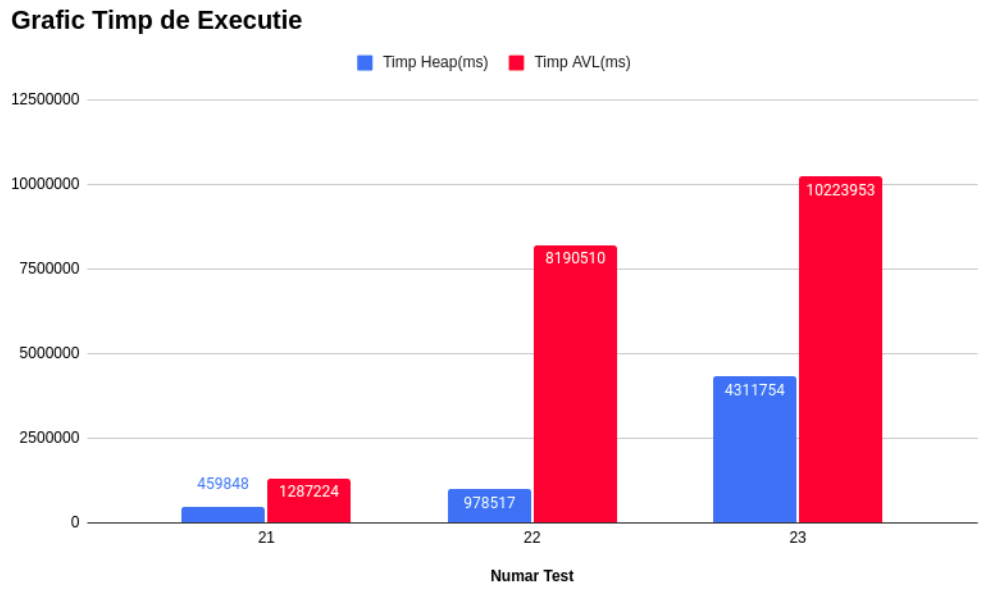

Fig. 9. Graficul timpului de executie pentru testele 21-23

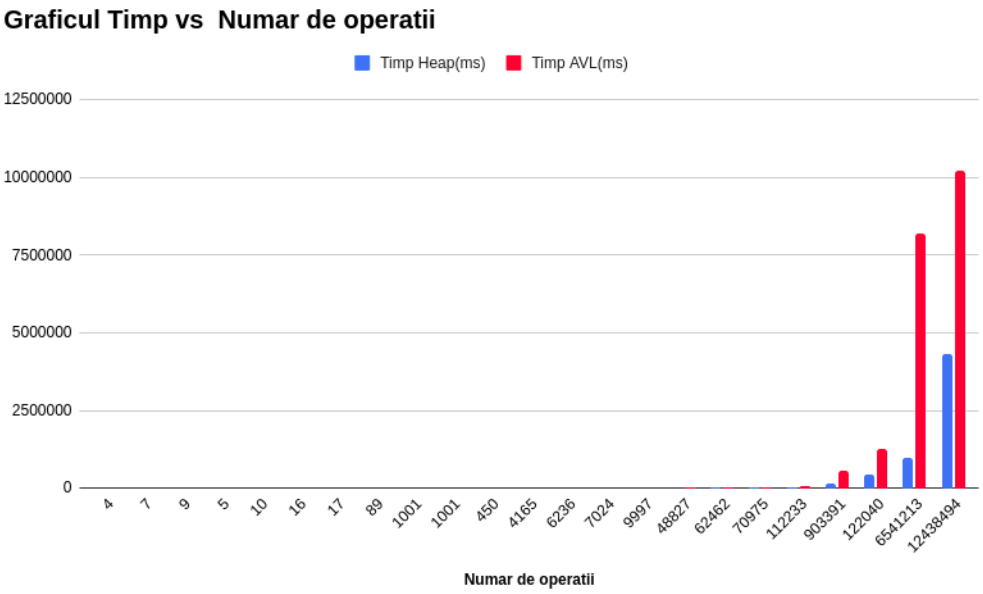

Fig. 10. Graficul timpului de executie raportat la numarul de operatii

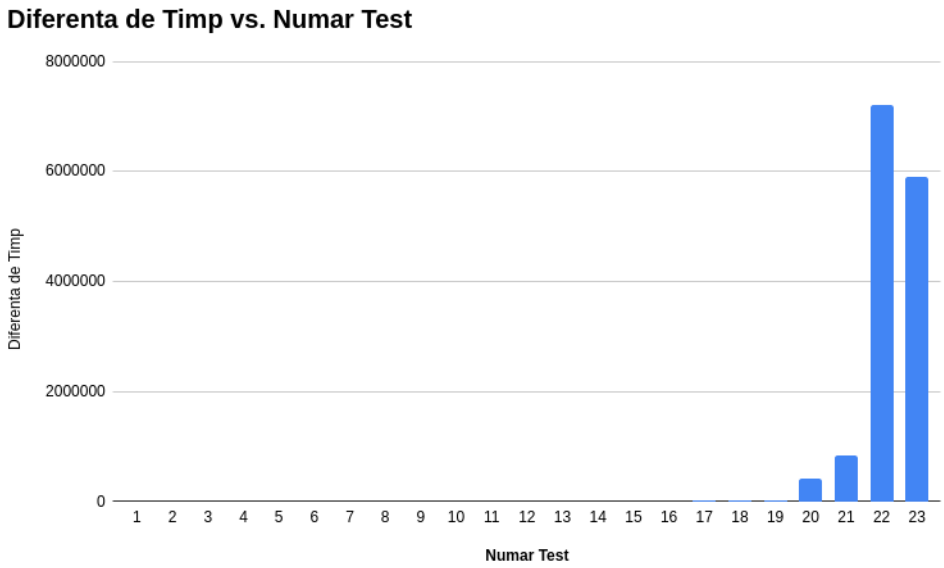

Fig. 11. Graficul diferentei de timp dintre cei doi algoritmi

3.4 Prezentarea, succintă, a valorilor obtinute pe teste

Prin intermediul testelor realizate se poate observa atat corectitudinea implementarilor celor doi algoritmi cat si timpul de executie specific al fiecaruia. Deoarece valorile difera de la o rulare la alta, am incercat sa realizeaza o medie a acestora. Pentru primele teste, unde numarul de operatii nu este mare, timpul de executie este aproximativ egal, ceea ce inseamna ca algoritmi se comporta similar pentru un numar redus de operatii. Insa incepand cu testul 7 se observa o diferenta din ce in ce mai mare a timpului de executie, in favorea algoritmului de Heap. Acest lucru se datoreaza atat operatiile de rotatie necesare AVL-ului pentru reechilibrare cat si reactualizarii inaltimilor. In cadrul testelor 9 si 10 am realizat operatia de push folosind o serie de valori sortate crescator sau descrescator. Analizand rezultate, acest lucru nu influenteaza timpul de executie. In cadrul testului 23, unde numarul de operatii este foarte complex, algoritmul de Heap ruleaza in 4311754 microsecounde, in vreme ce algoritmul de AVL ruleaza in 10223953 microsecunde, diferenta de timp fiind de 5912199 de microsecunde.

4 Concluzii

In urma analizei realizate asupra celor doi algoritmi, se observa ca algoritmul de Heap este superior celui de AVL, atat din punct de vedere al complexitatii operatiilor implementate, cat si din punct de vedere al timpului de executie. AVL-ul este de asemenea o Structura de Date complexa, mai dificil de implementat.
In practica, deoarece timpul pentru o cautare reprezinta un factor foarte important in cadrul unei cozi de prioritate, se alege implementarea acesteia folosind un Heap. Asadar pentru problema propusa si eu as aborda o implementare a cozii de prioritate bazata pe un algoritm de Max Heap, pentru a obtine elementul cu prioritatea maxima in cel mai scurt timp posibil.

References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms
2. http://faculty.salina.k-state.edu/tim/ossg/_images/priority_queues.png
3. https://www.geeksforgeeks.org/priority-queue-set-1-introduction/
4. https://www.geeksforgeeks.org/applications-priority-queue/
5. https://www.geeksforgeeks.org/heap-data-structure/
6. https://www.w3schools.in/data-structures-tutorial/avl-trees/
7. https://www.tutorialspoint.com/data_structures_algorithms/
8. https://www.geeksforgeeks.org/avl-tree-set-1-insertion/
9. https://ocw.cs.pub.ro/courses/sd-ca/laboratoare/lab-09
10. https://ocw.cs.pub.ro/courses/sd-ca/laboratoare/lab-11

Ultima dată în care am accesat referinţele a fost 17 Decembrie 2021.