Temă

- la disciplina ”Analiza Algoritmilor” -
- Structuri de date -
- AVL şi Binary Heap -

- Cozi de prioritate -

Ionaşcu Andrei

Anul II

325CD

Facultatea de Automatică şi Calculatoare

Universitatea Politehnică Bucureşti

14 decembrie 2017

Cuprins

1 Introducere

1.1 Prezentarea algoritmilor

Pentru această temă am ales să studiez structurile de date, mai exact, cozi de prioritate. Voi aduce spre implementare la categoria arbori binari echilibraţi,algoritmul AVL, iar la categoria heap, algoritmul Binary heap.

1.1.1 AVL tree

AVL tree-ul este un arbore binar echilibrat, care urmăreşte aceleaşi proprietăţi ca şi un arbore binar de căutare: fiecare nod din subarborele stâng prezintă o valoare mai mică decât nodul rădăcină, iar fiecare rădăcină din subarborele drept prezintă o valoare mai mare decât nodul rădăcină.

Avantajul AVL-ului vine prin abilitatea sa de a se echilibra, având în fiecare moment, un arbore echilibrat, cu ajutorul factorilor de echilibrare (între -1 şi 1).

Datorită acestei abilităţi, AVL-ul prezintă o complexitate de $\mathcal{O}(logN)$ atât în momentul inserării, cât şi la ştergerea şi căutarea unui număr.

1.1.2 Binary Heap

Binary Heap-ul este o structură de tip heap ce ia forma unui binary tree. Pe lăngă atributele unui binary tree, această structură prezintă încă două proprietăţi: aceasta este un arbore binar complet (toate nivele sunt complete, iar ultimul este completat de la stănga la dreapta) şi cheia stocată în fiecare nod este mai mare sau egală (mai mică sau egală) cu cheia din nodul copil, în funcţie de implementare.

Datorită acestor proprietăţi, Binary Heap-ul prezintă o complexitate la inserare de $\mathcal{O}(logN)$ pe cazul cel mai defavorabil, şi $\mathcal{O}(1)$ pe cazul mediu, la ştergerea unui element $\mathcal{O}(logN)$ în ambele cazuri şi la căutare $\mathcal{O}(N)$.

1.2 Compararea algoritmilor

Implementarea acestor două structuri prezintă atăt avantaje, căt şi dezavantaje pe anumite operaţii, neputănd să spunem că unul dintre aceşti algoritmi este cu mult mai bun, pe toate planurile, faţă de celălalt.

Astfel, algoritmul AVL este bun pe operaţii de căutare, datorită abilităţii acestuia de a se echilibra. Algoritmul Binary Heap este implementat adesea pe baza unui array. Stiind că numărul maxim(minim) din arbore se va afla in nodul root, în funcţie de tipul de Binary heap (minHeap sau maxHeap), acesta poate fi returnat imediat, cu o complexitate de $\mathcal{O}(1)$, ştiind că acesta se află pe poziţia 0 în array. De asemenea, dacă în această structură sunt inserate elemente deja sortate (aproape sortate), complexitatea inserări elementelor tinde spre $\mathcal{O}(1)$, pe când la AVL este nevoie ca după fiecare inserare să se echilibreze arborele, indiferent de oridinea inserării, dacă este nevoie.

1.3 Situaţii practice

Pentru a pune în evidenţă atuurile fiecărui algoritm, putem considera următoarele exemple practice generice:

- AVL va fi implementat atunci cǎnd ştim cǎ vom avea nevoie de interogări multiple asupra tuturor nodurilor, căutarea acestora fiind de complexitate $\mathcal{O}(logN)$, deoarece este echilibrat.

- Binary Heap va fi implementat atunci cǎnd ştim cǎ vom avea nevoie de interogări asupra minimului sau maximului (minHeap sau maxHeap), deoarece aceştia pot fi apelaţi în timp $\mathcal{O}(1)$, deoarece aceştia se află pe poziţia 0 în structură.

Pentru a vă oferi un exemplu complex, pot folosi aceste structuri în cadrul unui campionat de Karate, unde, doresc să menţin ierarhi cu concurenţi şi să realizez statistici.

Cu ajutorul AVL-ului voi putea menţine ierarhiile cu sportivi sortaţi după un anumit criteriu, putând să accesez rapid datele acestor sportivi.

Cu ajutorul Binary heap-ului voi putea realiza multiple statistici pe parcursul competiţiei, în care voi urmări cei mai buni/cei mai slabi concurenţi, în funcţie de lupte căştigate, puncte (tot ce ţine de interogarea minimului/maximului).

2 Complexitate

2.1 AVL tree

2.1.1 Temporală

Temporal, AVL prezintă pe insert, find, cât şi delete, o complexitate $\mathcal{O}(logN)$. Stiind că la fiecare inserare, arborele se va verifica dacă mai este echilibrat şi va realiza operaţii de echilibrare, acestea nu prezintă un impact asupra timpului de execuţie, fiind realizate puţine mutări în structură. Complexitatea acestuia este dată de nevoie de inserare a celor N elemente într-un arbore binar de căutare, reprezentată de înălţimea acestui arbore. Înălţimea unui arbore binar echilibrat este logN, rezultând o complexitate totală de $\mathcal{O}(logN)$.

Aceeaşi logică este prezentă şi în cadrul căutării şi al ştergerii. Pentru a căuta un element (şi a-l şterge) în arbore, programul va trebui să parcurgă aproape întreg arborele, sau chiar în totalitate, fiind de înălţime logN, iar când s-a ajuns la nodul dorit, în cazul în care dorim eliminarea acestuia, programul va realiza mici echilibrări nesemnificative în comparaţie cu logN.

2.1.2 Spaţială

Spaţial, AVL prezintă o complexitate $\mathcal{O}(1)$, deoarece acesta va aloca memorie o singură dată pentru fiecare nod din interiorul arborelui, fără a avea nevoie de spaţiu suplimentar.

2.2 Binary Heap

2.2.1 Temporală

Pe partea de insert, Binary Heap prezintă pe worst case, o complexitate $\mathcal{O}(logN)$, deoarece, la inserarea unui nou element în arbore, va trebui pus pe poziţia potrivită, prin interschimbări ale valorilor, prin array. Dacă introducem un element, acesta va fi plasat prin algoritm, la finalul array-ului, si va trebui să sa fie urcat în arbore până la root, arborele având orst case inălţimea logN, de aici venind şi complexitatea. Pe cazul mediu, inserarea poate fi chiar $\mathcal{O}(1)$, acesta fiind poziţionat chiar unde trebuie, sau în vecinătatea potrivită.

Algoritmul de ştergere prezintă o complexitate de $\mathcal{O}(logN)$, deoarece, după ştergerea unui element din arbore, acesta va fi înlocuit cu ultimul element din array şi pornind de la acea poziţie, trebuie să se readucă arborele la normal, pentru a-şi păstra proriетăţile. Astfel, elementul de la finalul array-ului, care se afla acum pe poziţia elementului eliminat, va trebui să parcurgă arborele păna va fi plasat la poziţia potrivită, arborele având înalţimea logN.

Algoritmul de cătuare este de complexitate $\mathcal{O}(N)$, deoarece acesta trebuie trebuie parcurs întreg array-ul pentru a găsi elementul dorit. Acest lucru se datorează faptului că elementele vor fi adăugate în arbore, de la stănga la dreapta, singura restrictie fiind ca elementul inserat să fie mai mic sau egal cu elementul părinte. Căutarea elementului maxim (minim) dintr-un MaxHeap (MinHeap) are o complexitate de $\mathcal{O}(1)$, deoarece acesta se va afla pe poziţia 0 în array, prezentănd avantajul acestei structuri.

2.2.2 Spaţială

Spaţial, Binary Heap prezintă complexitatea $\mathcal{O}(1)$, deoarece la iniţializarea acesteia, se va aloca memorie pentru un array cu N elemente, care va conţine datele pentru fiecare nod din structură, fără a fi nevoie de memorie auxiliară.

3 Implementare

Codul pentru AVL este implementat în java şi a fost luat de pe site-ul următor[1].

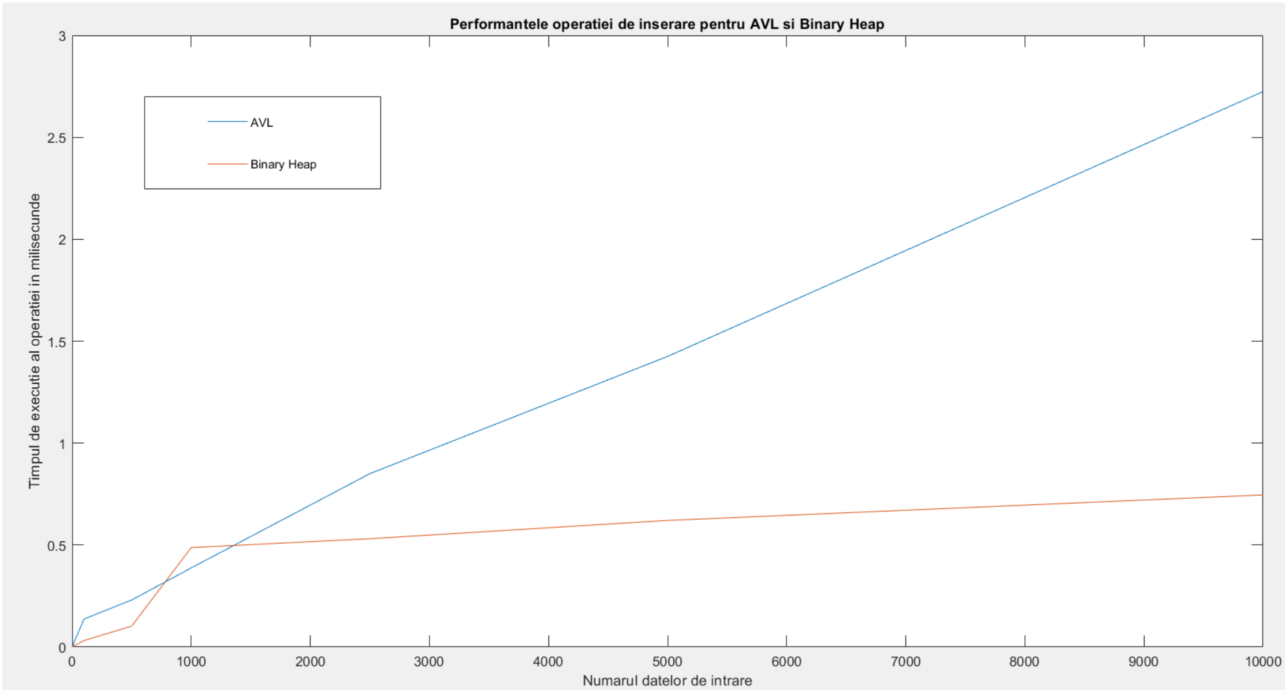
Codul pentru Binary Heap este impementat în java şi a fost luat de pe site-ul următor[2].

Implementările şi clasele de testare se pot găsi în interiorul arhivei. Implenetările algoritmilor se găsesc la finalul documentului în anexa de implementare, deoarece acestea ocupă o dimensiune foarte mare.

4 Măsurarea performanţei

Toate testele au fost realizate pe un set de date pe cazul worst case (datele au fost generate cu ajutorul site-ului Random.org[3] şi sunt ordonate descrescător). Datele introduse sunt de următoarele dimensiuni: 100, 500, 1000, 2500, 5000, 10000.

4.1 Performanţă Insert

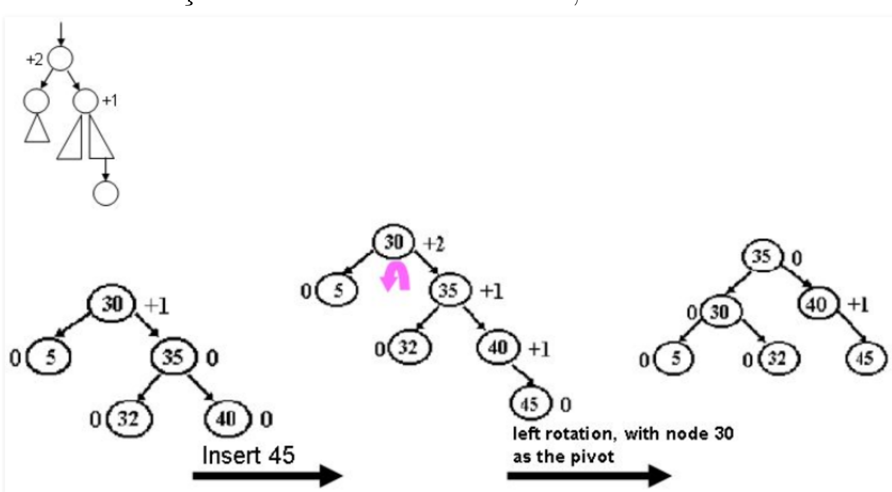


În urma testării performanţei pentru operaţia de inserare, am obţinut graficul de mai sus, în care durata de execuţie a progrmului este în milisecunde.

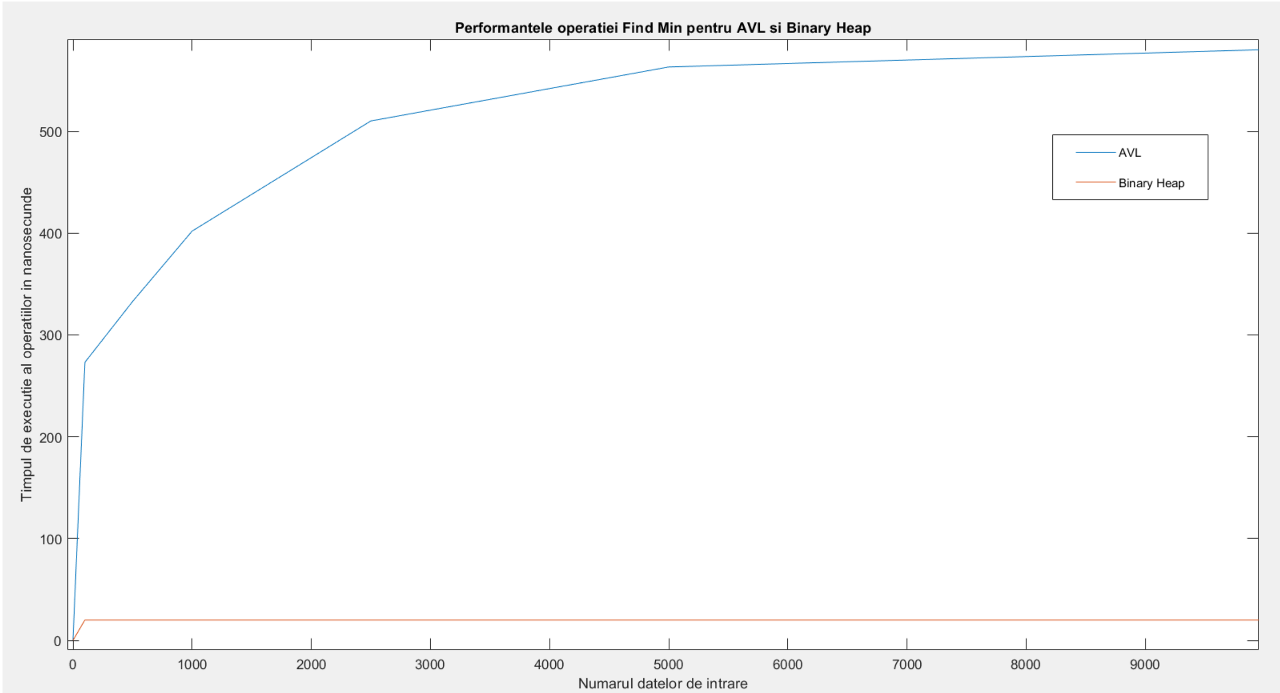
Cu ajutorul acestui grafic putem observa că timpul de execuţie al structurii AVL este cu mult mai înceată faţă de Binary Heap.

Fiecare element nou introdus într-un Binary Heap va fi stocat într-un array, fiind pus la coada array-ului, ca mai apoi să se parcurgă prin părinţi pentru a i se oferi poziţia corectă în arbore. Din acest motiv, Binary Heap este mai rapid decât operaţia de insert din AVL, care, va parcurge de la root fiecare nod, în căutarea locului potrivit, ca mai apoi să asigure echilibrarea arborelui, prin rotaţiile necesare.

Puteţi observa în imaginea următoare, în ce constă inserarea unui element nou în AVL, ce realizează şi o dezechilibrare a acestuia, fiind nevoie să se re-echilibreze.



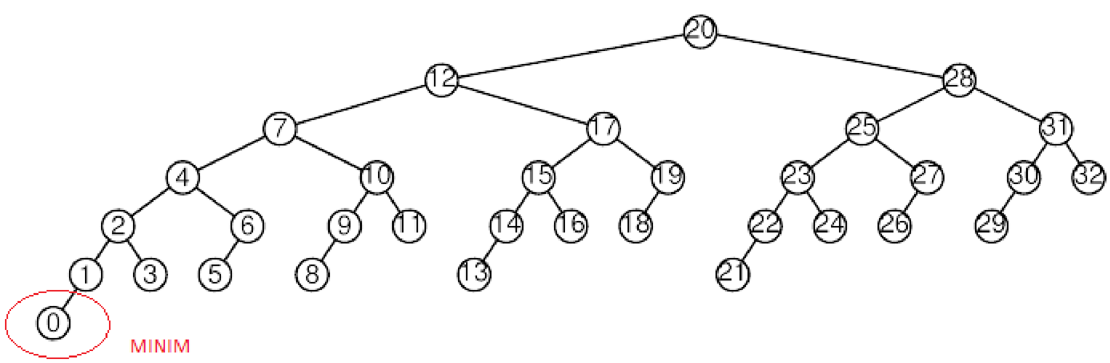
4.2 Performanţă Find Minimum



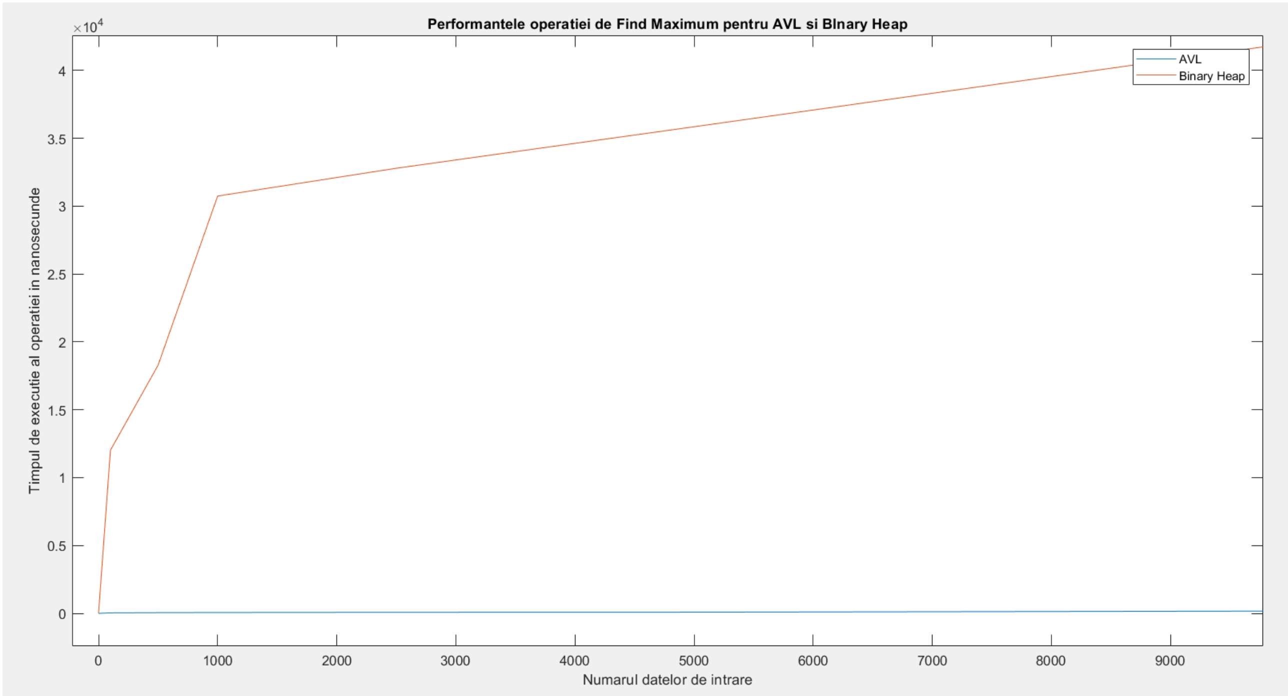
În urma testării performanţei pentru operaţia de căutare a elementului minim, am obţinut graficul de mai sus, în care durata de execuţie a progrmului este în nanosecunde.

Pe acest grafic se poate observa atuu-ul structurii Binary Heap (MinHeap), care oferă termenul minim instant, deoarece acesta se află în nodul root, pe poziţia 0 din array-ul arborelui. Pentru a elimina elementul minim dintr-un AVL, va trebui să se meargă pe subarborele stâng până se ajunge în nodul frunză, acela fiind minimul din arbore.

În următoare imagine se poate observa poziţia elementului minim din interiorul unui AVL, precum am explicat anterior.



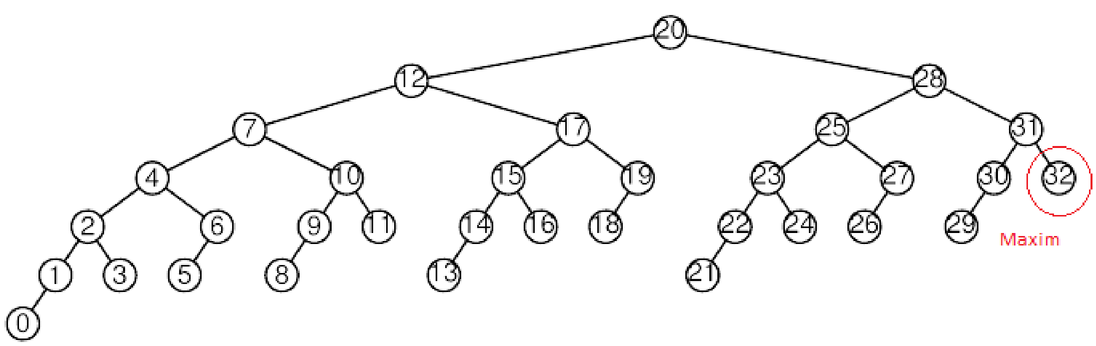
4.3 Performanţă Find Maximum



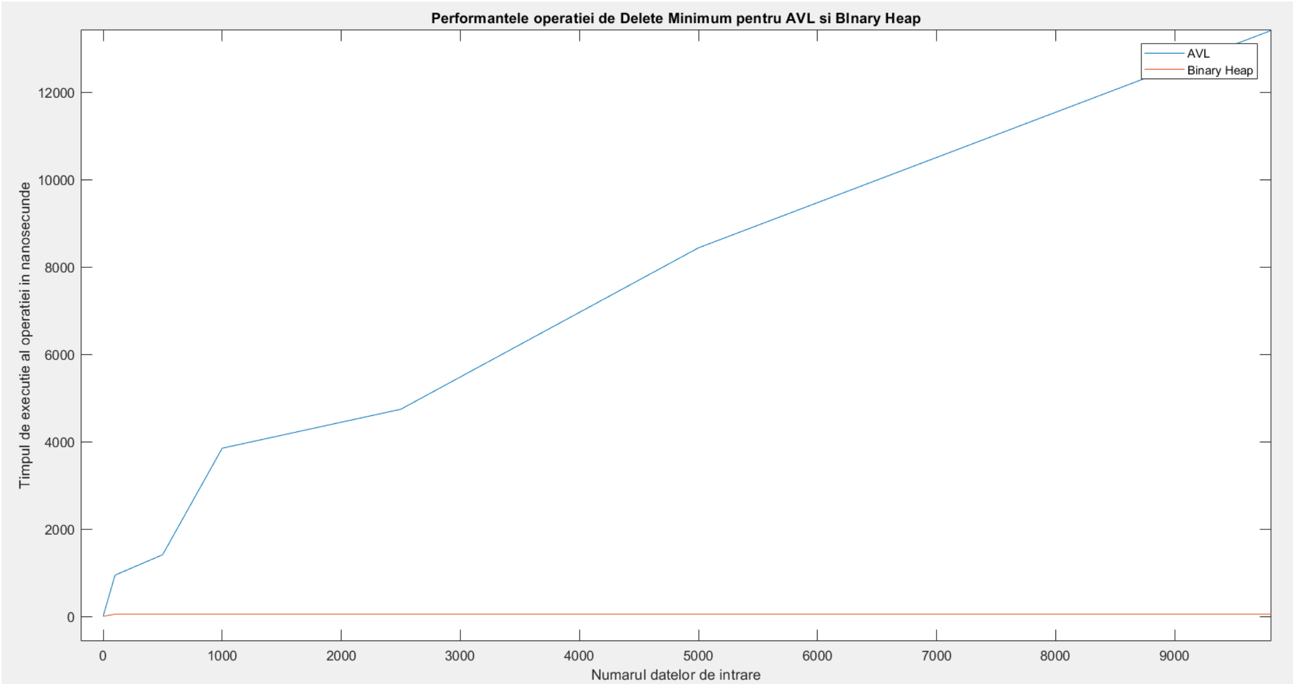
În urma testării performanţei pentru operaţia de căutare a elementului maxim, am obţinut graficul de mai sus, în care durata de execuţie a progrmului este în nanosecunde.

În acest grafic se observă timpul foarte mare de care are nevoie un Binary Heap (MinHeap) pentru a găsi un element din interiorul arborelui. Acesta trebuie să caute în intreg array-ul pentru a găsi elementul dorit, pe când, la AVL, acesta va merge pe subarborele drept până va găsi ultimul element frunză, care va fi maximul din acel arbore.

În următoare imagine se poate observa poziţia elementului maxim din interiorul unui AVL, precum am explicat anterior.



4.4 Performanţă Delete Minimum

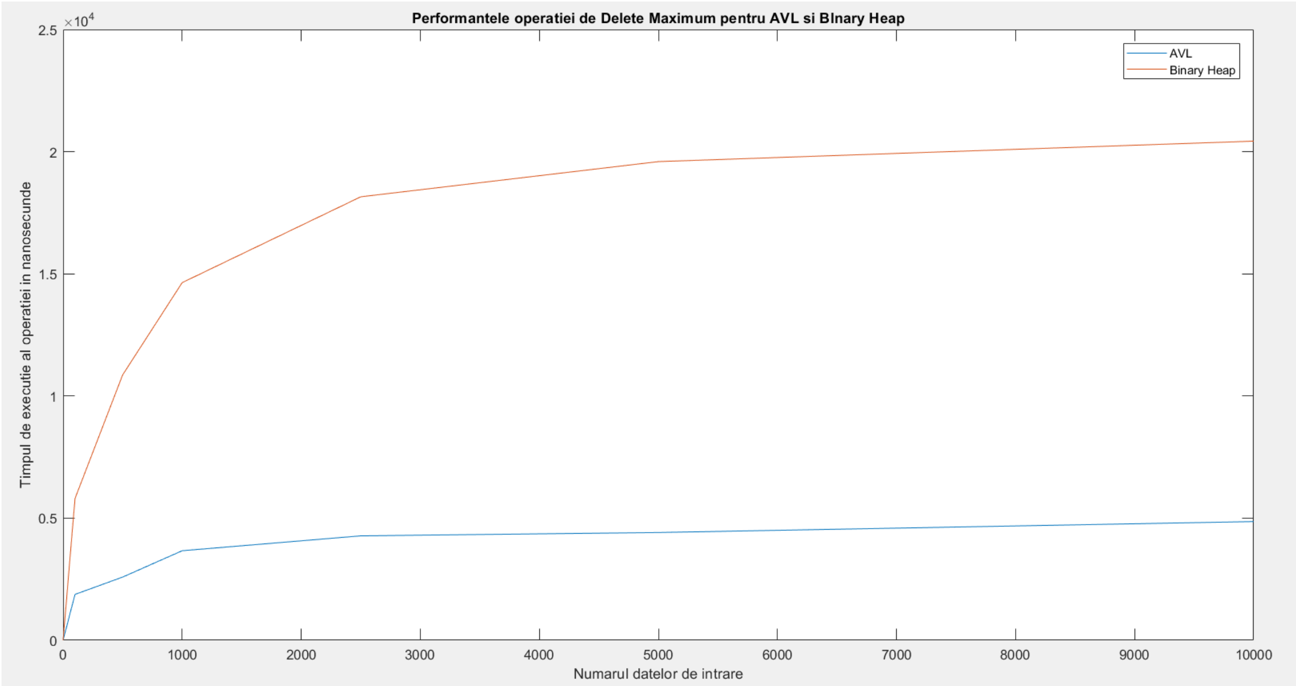


În urma testării performanţei pentru operaţia de ştergere a elementului minim, am obţinut graficul de mai sus, în care durata de execuţie a progrmului este în nanosecunde.

Pe acest grafic se observă eficienţa eliminării elementului minim dintr-un Binary Heap, aflat pe poziţia 0 din array, precum am menţionat anterior, la Performanţa Find Minimum (pe cazul implementat MinHeap). Timpul de execuţie în Binary Heap este dat de necesitatea înlocuiri root-ului cu succesorul acestuia.

Pentru a elimina elementul minim dintr-un AVL, acesta va parcurge arborele stâng până se va ajunge la acesta, se va elimina şi va trebui să se realizeze echilibrare, dacă este nevoie.

4.5 Performanţă Delete Maximum



În urma testării performanţei pentru operaţia de ştergere a elementului maxim, am obţinut graficul de mai sus, în care durata de execuţie a progrmului este în nanosecunde.

În acest grafic se observă cât de mult durează iterarea pentru un BInary Heap să itereze prin întreg array-ul pentru a găsi elementul dorit şi înlocuirea acestuia cu elementul corespunzător pentru a păstra proprietatea arborelui.

AVL-ul va parcurge subarborele drept până va ajunge la nodul frunză şi va elimina elementul, realizând mai apoi, echilibrarea, dacaă este nevoie.

5 Concluzie

În urma analizei prezentate în paginile anterioare, putem concluziona că nu putem spune despre unul din cei doi algoritmi este superior faţă de celălalt pe toate planurile.

AVL este o structură foarte bună în cazul în care nu avem nevoie să adagăm de prea multe ori elemente pe parcursul folosirii acesteia şi cel mai important, dorim să facem foarte multe interogări, în căutarea elementelor din interiorul structurii. Fiind un arbore binar echilibrat, înălţimea arborelui este logN, iar căutarea elementelor se face destul de rapid, deoarece cunoaştem că fiul drept al unui nod este reprezentat de o valoare mai mare decât nodul, iar fiul stâng al unui nod, este reprezentat de o valoare mai mică decât acesta.

Binary Heap este o structură ce adaugă elemente destul de rapid. Dacă la finalul array-ului, unde a fost introdus noul element, se va afla conform proprietăţilor Binary Heap-ului, această inserare este chiar de complexitate $\mathcal{O}(1)$. De asemenea, atuul acestei structuri este dat de accesul în $\mathcal{O}(1)$ asupra elementului maxim (MaxHeap) şi minim (MinHeap). Acest lucru face ca structura să fie ideală pentru cazuri practice în care ştii că este nevoie de interogări repete asupra minimului/maximului, precum statistici.

6 Anexă implementări

6.1 AVL

```
package AVLNou;
class Node
{
    int key, height;
    Node left, right;

    Node(int d)
    {
        key = d;
        height = 1;
    }
}

class AVLTree
{
    Node root;
    int height(Node N)
    {
        if (N == null)
             return 0;
         return N.height;
    }

    int max(int a, int b)
    {
        return (a > b) ? a : b;
    }

    Node rightRotate(Node y)
    {
        Node x = y.left;
        Node T2 = x.right;

        x.right = y;
        y.left = T2;

        y.height = max(height(y.left), height(y.right)) + 1;
        x.height = max(height(x.left), height(x.right)) + 1;

        return x;
    }

    Node leftRotate(Node x)
    {
        Node y = x.right;
        Node T2 = y.left;

        y.left = x;
        x.right = T2;

```

```
        x.height = max(height(x.left), height(x.right)) + 1;
        y.height = max(height(y.left), height(y.right)) + 1;

        return y;
    }

    int getBalance(Node N)
    {
        if (N == null)
            return 0;
        return height(N.left) - height(N.right);
    }

    Node insert(Node node, int key)
    {

        if (node == null)
            return (new Node(key));

        if (key < node.key)
            node.left = insert(node.left, key);
        else if (key > node.key)
            node.right = insert(node.right, key);
        else
            return node;

        node.height = 1 + max(height(node.left),
                              height(node.right));

        int balance = getBalance(node);

        if (balance > 1 && key < node.left.key)
            return rightRotate(node);

        if (balance < -1 && key > node.right.key)
            return leftRotate(node);

        if (balance > 1 && key > node.left.key)
        {
            node.left = leftRotate(node.left);
            return rightRotate(node);
        }

        if (balance < -1 && key < node.right.key)
        {
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }

        return node;
    }

    Node minValueNode(Node node)
    {
```

```
        Node current = node;

        while (current.left != null)
            current = current.left;

        return current;
    }

    Node maxValueNode(Node node)
    {
        Node current = node;

        while (current.right != null)
            current = current.right;

        return current;
    }

    Node deleteNode(Node root, int key)
    {
        if (root == null)
            return root;

        if (key < root.key)
            root.left = deleteNode(root.left, key);

        else if (key > root.key)
            root.right = deleteNode(root.right, key);

        else
        {

            if ((root.left == null) || (root.right == null))
            {
                Node temp = null;
                if (temp == root.left)
                    temp = root.right;
                else
                    temp = root.left;

                if (temp == null)
                {
                    temp = root;
                    root = null;
                }
                else
                    root = temp;
            }
            else
            {
                Node temp = minValueNode(root.right);
                root.key = temp.key;
                root.right = deleteNode(root.right, temp.key);
            }
```

```
        }
        if (root == null)
            return root;
        root.height = max(height(root.left), height(root.right)) + 1;
        int balance = getBalance(root);

        if (balance > 1 && getBalance(root.left) >= 0)
            return rightRotate(root);

        if (balance > 1 && getBalance(root.left) < 0)
        {
            root.left = leftRotate(root.left);
            return rightRotate(root);
        }

        if (balance < -1 && getBalance(root.right) <= 0)
            return leftRotate(root);

        if (balance < -1 && getBalance(root.right) > 0)
        {
            root.right = rightRotate(root.right);
            return leftRotate(root);
        }

        return root;
    }

    void preOrder(Node node)
    {
        if (node != null)
        {
            System.out.print(node.key + " ");
            preOrder(node.left);
            preOrder(node.right);
        }
    }

    Node findNode(Node node, int value) {

        if (node == null) {
            return null;
        } else if (value < node.key && node.left != null) {
            return findNode (node.left, value);
        } else if (value > node.key && node.right != null) {
            return findNode(node.right, value);
        } else {

          return node;
        }


    }
}
```

6.2 Binary Heap

```java
package BinaryHeap;
import java.util.Arrays;
import java.util.NoSuchElementException;

class BinaryHeap
{
    private static final int d = 2;
    private int heapSize;
    private int[] heap;

    public BinaryHeap(int capacity)
    {
        heapSize = 0;
        heap = new int[capacity + 1];
        Arrays.fill(heap, -1);
    }

    public boolean isEmpty( )
    {
        return heapSize == 0;
    }

    public boolean isFull( )
    {
        return heapSize == heap.length;
    }

    public void makeEmpty( )
    {
        heapSize = 0;
    }

    private int parent(int i)
    {
        return (i - 1)/d;
    }

    private int kthChild(int i, int k)
    {
        return d * i + k;
    }

    public void insert(int x)
    {
        if (isFull( ) )
            throw new NoSuchElementException("Overflow Exception");
        heap[heapSize++] = x;
        heapifyUp(heapSize - 1);
    }

    public int findMin( )
    {
```

```
        if (isEmpty() )
            throw new NoSuchElementException("Underflow Exception");
        return heap[0];
    }

    public int findMax( )
    {
        if (isEmpty() )
            throw new NoSuchElementException("Underflow Exception");
        int max =0;
        int pos =-1;
        for(int i=0; i<heapSize;i++) {
            if(max< heap[i] ) {
                max = heap[i];
                pos =i;
            }
        }
        return pos;
    }

    public int deleteMin()
    {
        int keyItem = heap[0];
        delete(0);
        return keyItem;
    }

    public int delete(int ind)
    {
        if (isEmpty() )
            throw new NoSuchElementException("Underflow Exception");
        int keyItem = heap[ind];
        heap[ind] = heap[heapSize - 1];
        heapSize--;
        heapifyDown(ind);
        return keyItem;
    }

    private void heapifyUp(int childInd)
    {
        int tmp = heap[childInd];
        while (childInd > 0 && tmp < heap[parent(childInd)])
        {
            heap[childInd] = heap[ parent(childInd) ];
            childInd = parent(childInd);
        }
        heap[childInd] = tmp;
    }

    private void heapifyDown(int ind)
    {
        int child;
        int tmp = heap[ ind ];
        while (kthChild(ind, 1) < heapSize)
```

```java
        {
            child = minChild(ind);
            if (heap[child] < tmp)
                heap[ind] = heap[child];
            else
                break;
            ind = child;
        }
        heap[ind] = tmp;
    }

    private int minChild(int ind)
    {
        int bestChild = kthChild(ind, 1);
        int k = 2;
        int pos = kthChild(ind, k);
        while ((k <= d) && (pos < heapSize))
        {
            if (heap[pos] < heap[bestChild])
                bestChild = pos;
            pos = kthChild(ind, k++);
        }
        return bestChild;
    }

    public void printHeap()
    {
        System.out.print("\nHeap = ");
        for (int i = 0; i < heapSize; i++)
            System.out.print(heap[i] +" ");
        System.out.println();
    }
}
```

Bibliografie

[1] AVL tree
http://www.geeksforgeeks.org/avl-tree-set-1-insertion/

[2] Binary Heap
http://www.sanfoundry.com/java-program-implement-binary-heap/

[3] Random generator
https://www.random.org/strings/