**Tutoriat SD 2: Complexitati, Heap-uri**

## **1. Introducere în Analiza Complexității**

### **De ce este importantă complexitatea?**

Analiza complexității ne ajută să înțelegem cât de eficient este un algoritm în ceea ce privește timpul și memoria. Aceasta ne permite să comparăm diferite abordări și să alegem cea mai bună soluție pentru o problemă.

### **Tipuri de notații asimptotice**

Există trei tipuri principale de notații asimptotice folosite pentru a analiza algoritmii:

#### **1. Notația Big O (O)**

* Reprezintă **timpul de execuție în cel mai rău caz**.
* Exemplu: Căutarea unui element într-un array nesortat durează **O(n)** în cel mai rău caz.

#### **2. Notația Omega (Ω)**

* Reprezintă **timpul de execuție în cel mai bun caz**.
* Exemplu: Căutarea într-un array sortat folosind Căutare Binare durează **Ω(1)** dacă elementul este găsit în prima poziție.

#### **3. Notația Theta (Θ)**

* Reprezintă **timpul mediu de execuție**.
* Exemplu: Căutarea într-un array nesortat are complexitatea **Θ(n)**, deoarece, în medie, trebuie verificate jumătate dintre elemente.

### **Complexități comune și semnificația acestora**

* **O(1) - Timp Constant**
  + Exemplu: Accesarea unui element dintr-un array folosind un index.

Exemplu de cod:  
  
 int arr[] = {10, 20, 30};

int x = arr[1]; // O(1)

* **O(n) - Timp Liniar**
  + Exemplu: Parcurgerea unui array.

Exemplu de cod:  
  
 for(int i = 0; i < n; i++) {

cout << arr[i] << " ";

}

* **O(log n) - Timp Logaritmic**
  + Exemplu: Căutare Binare.

Exemplu de cod:  
  
 int binarySearch(int arr[], int left, int right, int key) {

while(left <= right) {

int mid = left + (right - left) / 2;

if(arr[mid] == key) return mid;

else if(arr[mid] < key) left = mid + 1;

else right = mid - 1;

}

return -1;

}

* **O(n log n) - Performanță eficientă în anumite algoritmi**
  + Exemplu: Anumiți algoritmi de sortare au complexitatea **O(n log n)**.
* **O(n²) - Timp Quadratic**
  + Exemplu: Parcurgerea unei matrici pătrate de dimensiune **n × n** necesită **O(n²)** operații.

## **2. Înțelegerea Arborilor**

### **Ce este un arbore?**

Un arbore este o structură de date ierarhică formată din noduri. Fiecare nod conține o valoare și legături (muchii) către nodurile copil. Nodul din vârful ierarhiei se numește **rădăcină**.

* **Nod rădăcină**: Nodul de început al arborelui.
* **Nod părinte**: Un nod care are copii.
* **Nod copil**: Un nod care este descendent al altui nod.
* **Nod frunză**: Un nod care nu are copii.

### **Arbori Binari**

Un **Arbore Binar** este un tip special de arbore în care fiecare nod are cel mult **doi copii** (stânga și dreapta).

* **Arbore Binar Complet**: Toate nivelurile sunt complet umplute, cu excepția ultimului nivel.
* **Arbore Binar Perfect**: Toate nodurile interne au exact doi copii, iar toate frunzele sunt la același nivel.

### **Implementare Arbore Binar în C++**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

};

Node\* createNode(int value) {

Node\* newNode = new Node();

newNode->data = value;

newNode->left = newNode->right = nullptr;

return newNode;

}

int main() {

Node\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

cout << "Rădăcină: " << root->data << " Stânga: " << root->left->data << " Dreapta: " << root->right->data << endl;

return 0;

}

## **3. Înțelegerea Heap-urilor**

### **Ce este un Heap?**

Un heap este un **arbore binar complet** în care:

* Într-un **Max Heap**, cel mai mare element este în rădăcină.
* Într-un **Min Heap**, cel mai mic element este în rădăcină.

### **Operații pe Heap**

1. **Inserare (O(log n))**
2. **Ștergere (O(log n))**
3. **Obținerea valorii maxime/minime (O(1))**

### **Implementare Max Heap în C++**

#include <iostream>

using namespace std;

#include <queue>

int main() {

priority\_queue<int> pq; // Max Heap

pq.push(10);

pq.push(20);

pq.push(15);

cout << "Element Max: " << pq.top() << endl;

pq.pop();

cout << "Element Max după ștergere: " << pq.top() << endl;

return 0;

}

### **Aplicații ale Heap-urilor**

* **Cozi de Prioritate** (ex: Algoritmul Dijkstra)
* **Heap Sort** (O(n log n))
* **Probleme de Planificare** (ex: Planificarea sarcinilor în CPU)