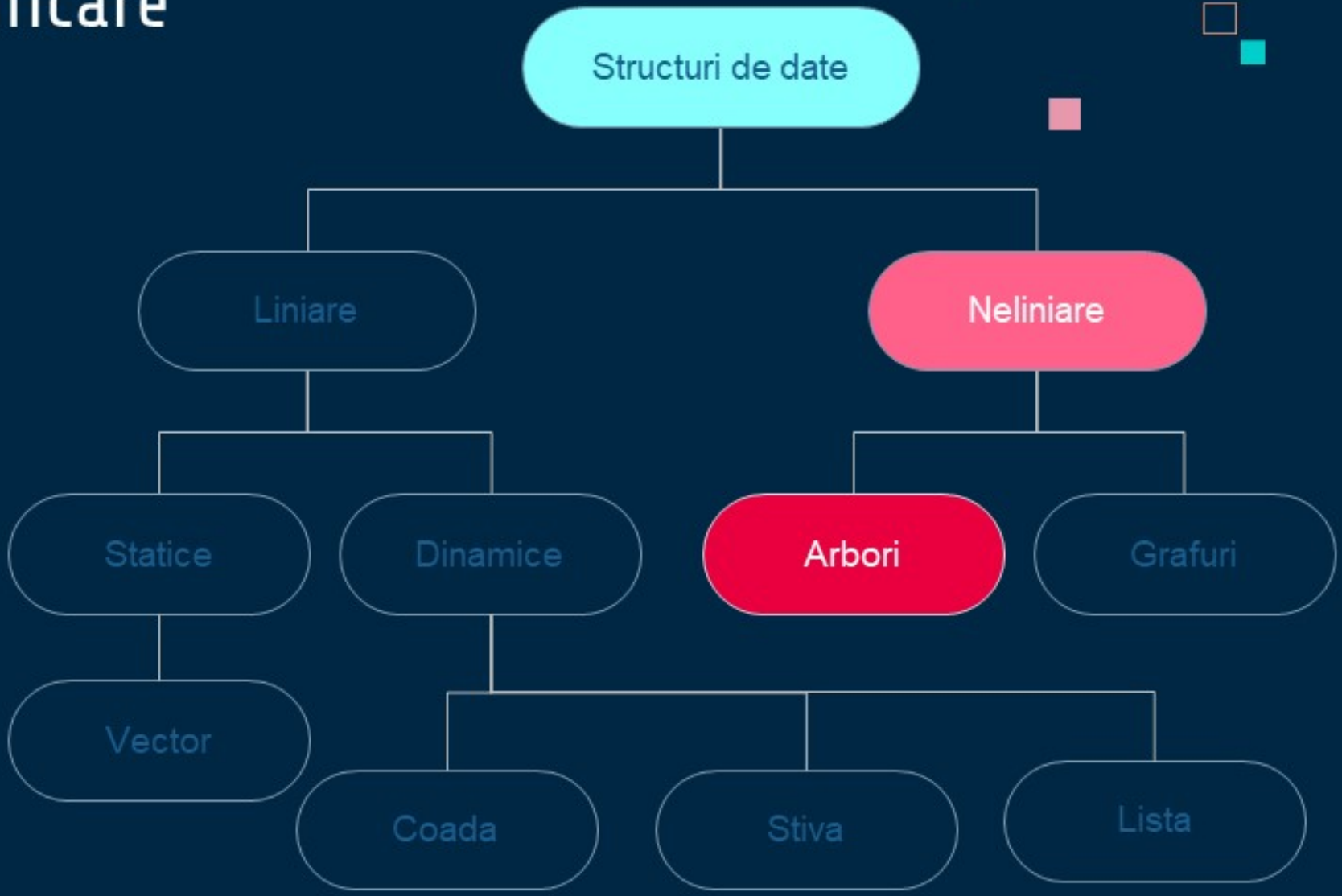
Cursul nr. 12

STRUCTURI DE DATE
neliniare
Aplicații Arbori Binari

Lector dr. Dorin IORDACHE

Clasificare



Agenda

01

Heap

02

Operații heap

03

Heapsort

04

Cod Huffman

Heap

01

Heap – definire

Heap este o structură specială de date bazată pe arbore, în care arborele este un arbore binar complet

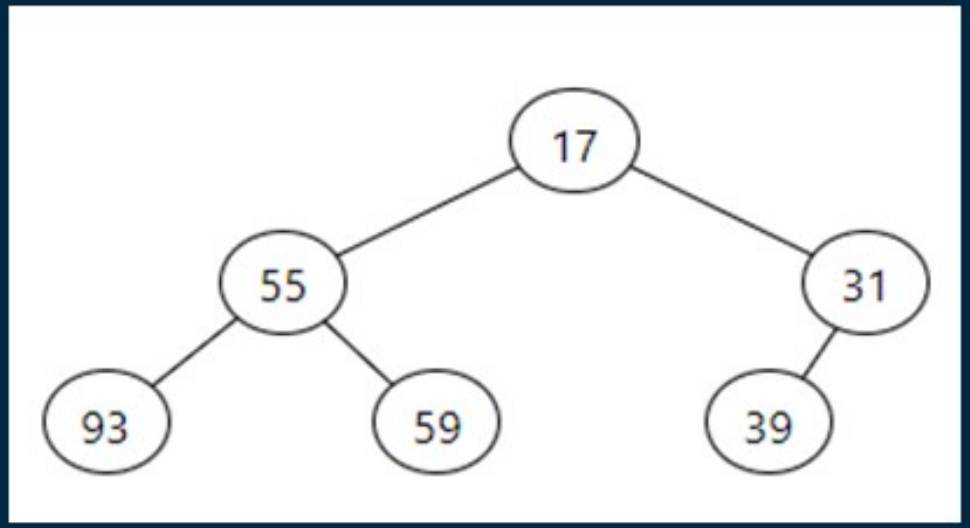
care satisface proprietatea heap, unde orice nod al arborelui este:

- întotdeauna mai mare decât nodul/rile copil și cheia nodului rădăcină este cea mai mare dintre toate celelalte noduri. Această proprietate este numită și proprietatea **maxim heap**.

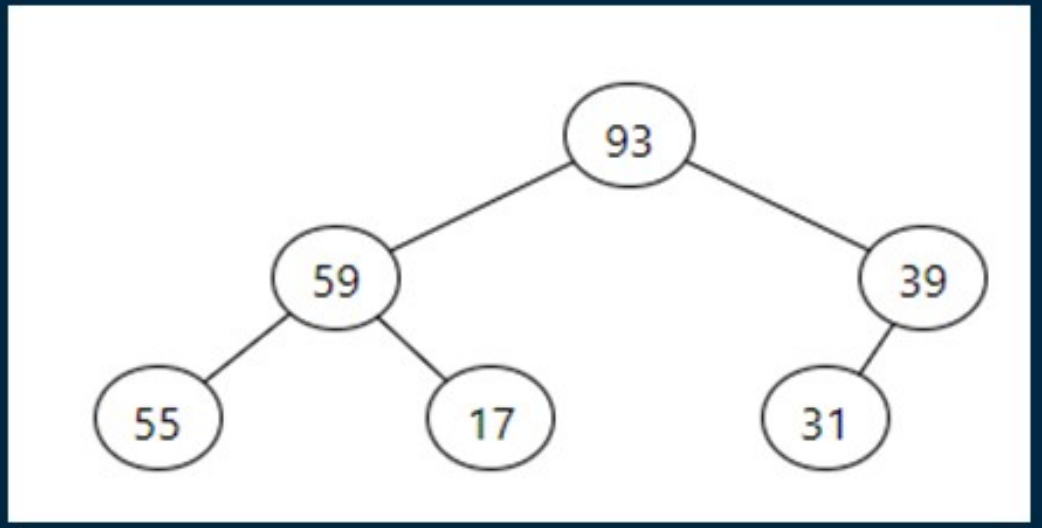
- întotdeauna mai mic decât nodul/rile copil și cheia nodului rădăcină este cea mai mică dintre toate celelalte noduri. Această proprietate se mai numește și proprietate **min heap**.

Heap

Min Heap



Max Heap



Operații heap

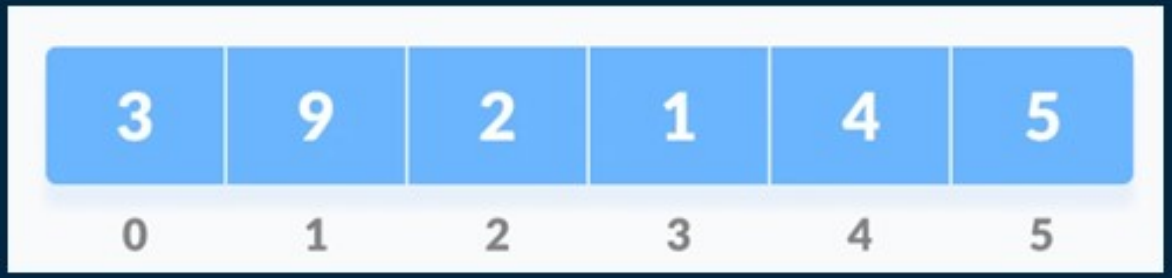
02

Operații heap - Heapify

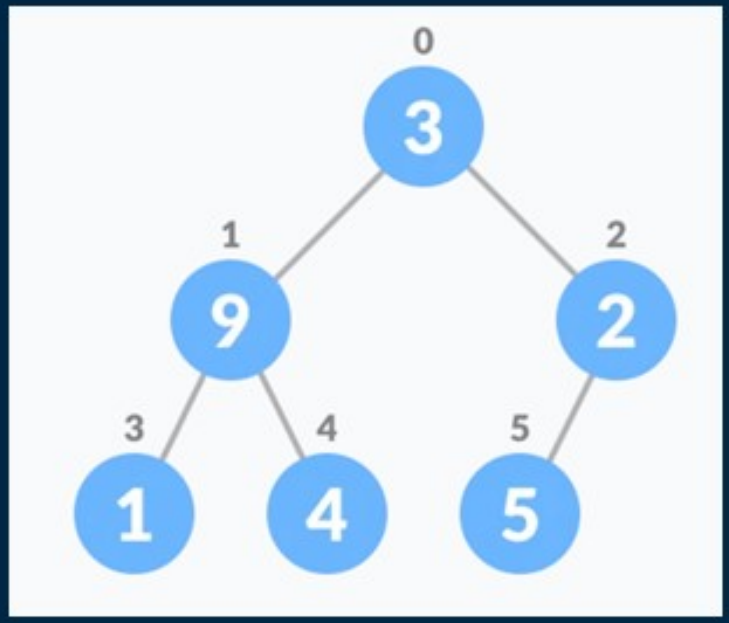
Heapify este procesul de creare a unei structuri de date heap dintr-un arbore binar.
Este folosit pentru a crea un Min-Heap sau un Max-Heap.

Heapify

1. Fie vectorul de mai jos.



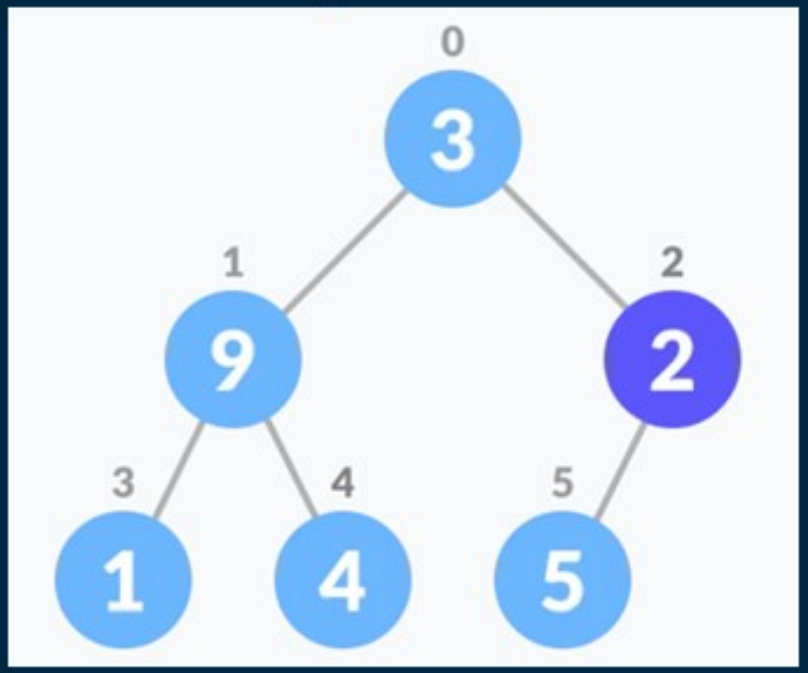
2. Generăm arborele binar complet



https://www.programiz.com/dsa/heap-data-structure

Heapify

3. Start de la primul index al nodului non-frunză al cărui indice este dat de n/2 - 1



4. Setați elementul curent i ca fiind cel mai mare

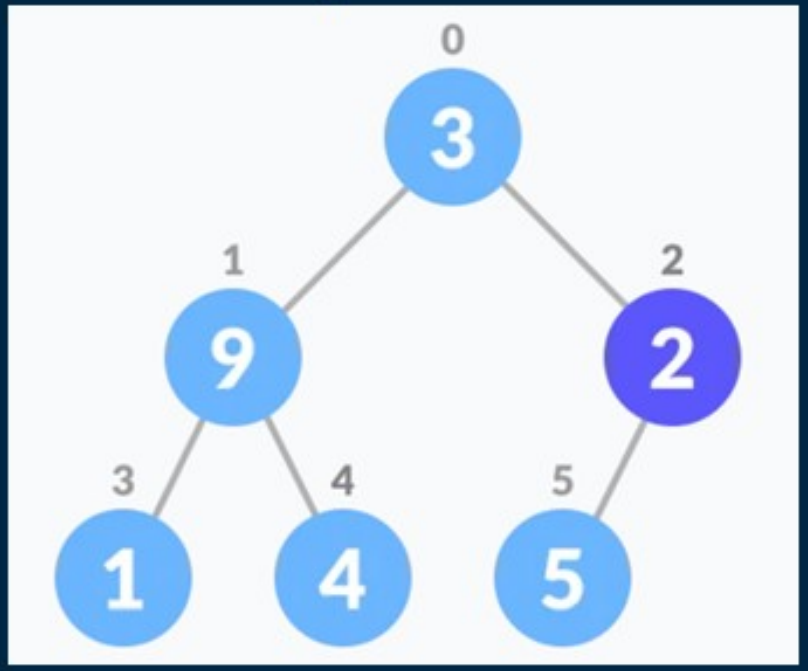
5. Indexul copilului stâng este dat de 2i + 1, iar copilul din dreapta este dat de 2i + 2.
dacă *stangaCopil* este mai mare decât *elementCurent*, set *stangaCopilIndex* ca *celMaiMare*.
dacă *dreaptaCopil* este mai mare decât elementul *celMaiMare*, set *dreaptaCopilIndex* ca *celMaiMare*.

6. Interschimbare *celMaiMare* cu *elementCurent*

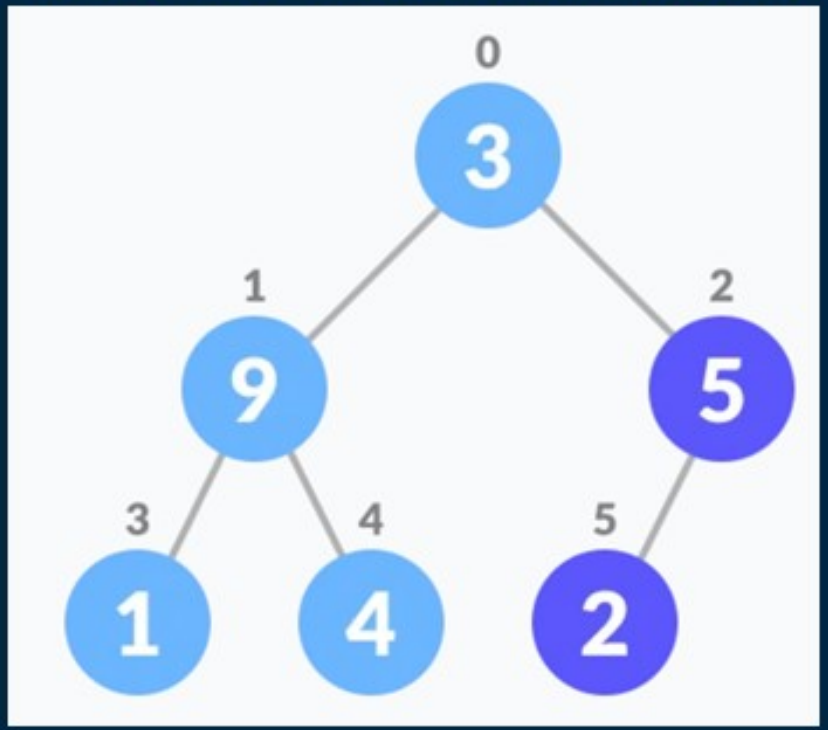
https://www.programiz.com/dsa/heap-data-structure

Heapify

3. Start de la primul index al nodului non-frunză al cărui indice este dat de n/2 - 1



6. Interschimbare *celMaiMare* cu *elementCurent*

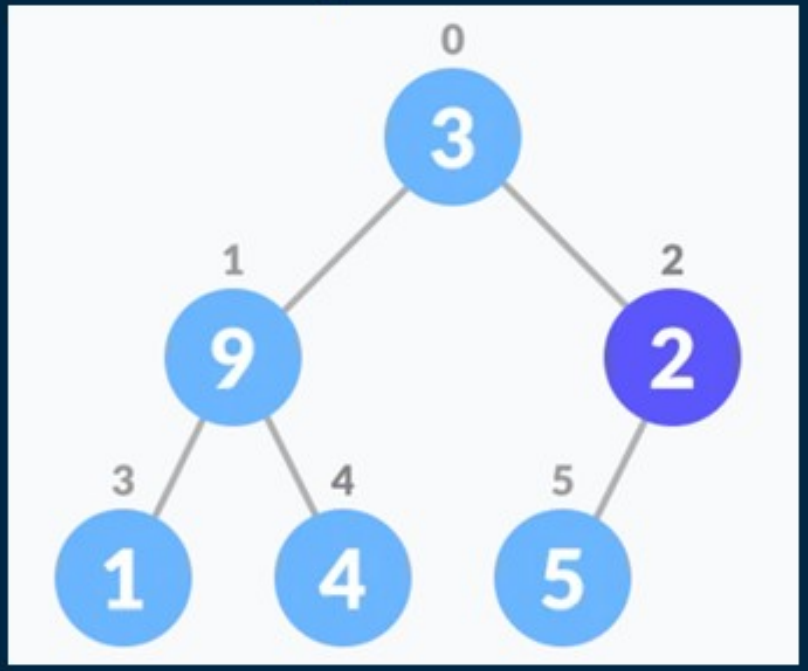


https://www.programiz.com/dsa/heap-data-structure

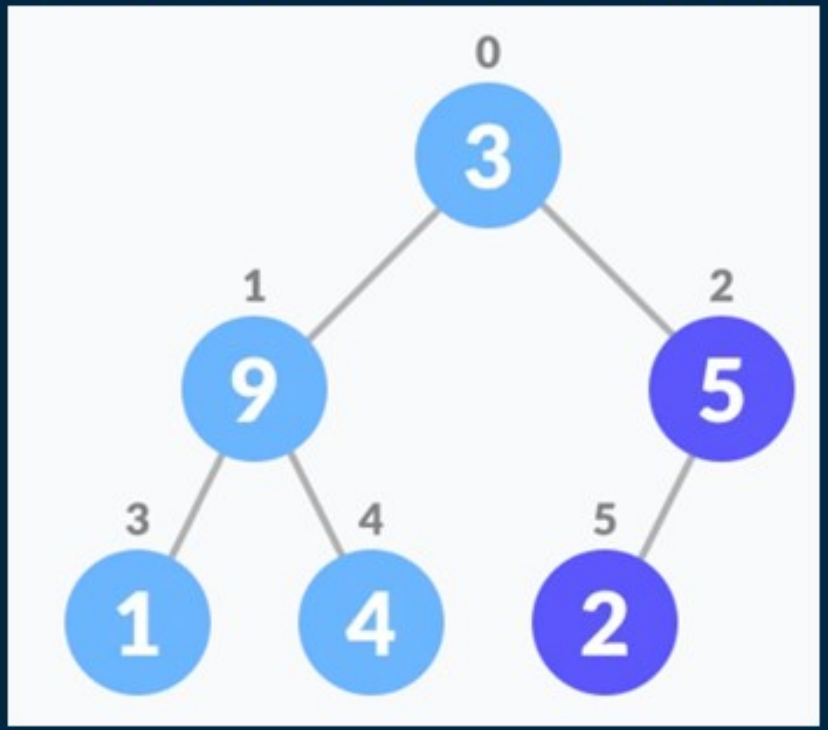
7. Repatare pașii 3-7 până se va obține arborele heap

Heapify

3. Start de la primul index al nodului non-frunză al cărui indice este dat de n/2 - 1



6. Interschimbare *celMaiMare* cu *elementCurent*

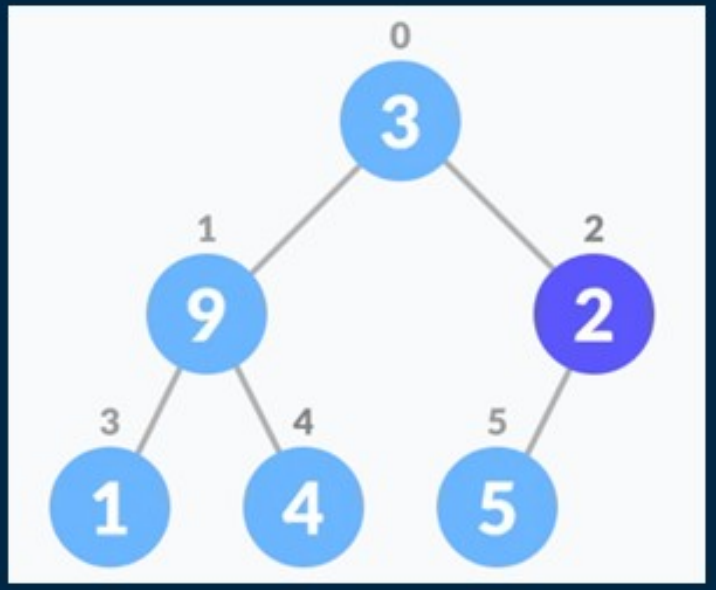


https://www.programiz.com/dsa/heap-data-structure

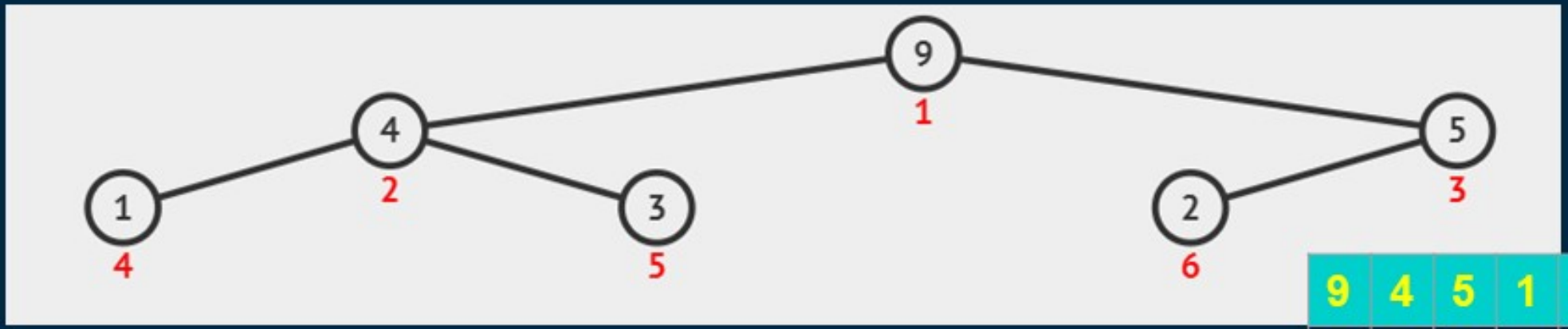
7. Repatare pașii 3-7 până se va obține arborele heap

Heapify

3	9	2	1	4	5
0	1	2	3	4	5



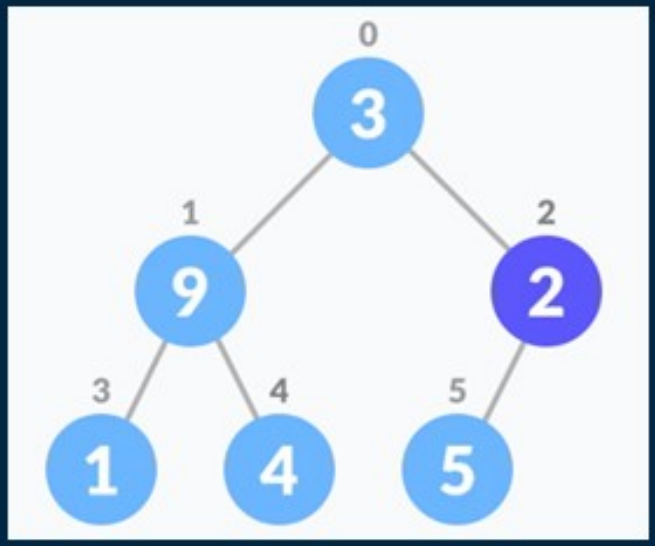
final – Max Heap



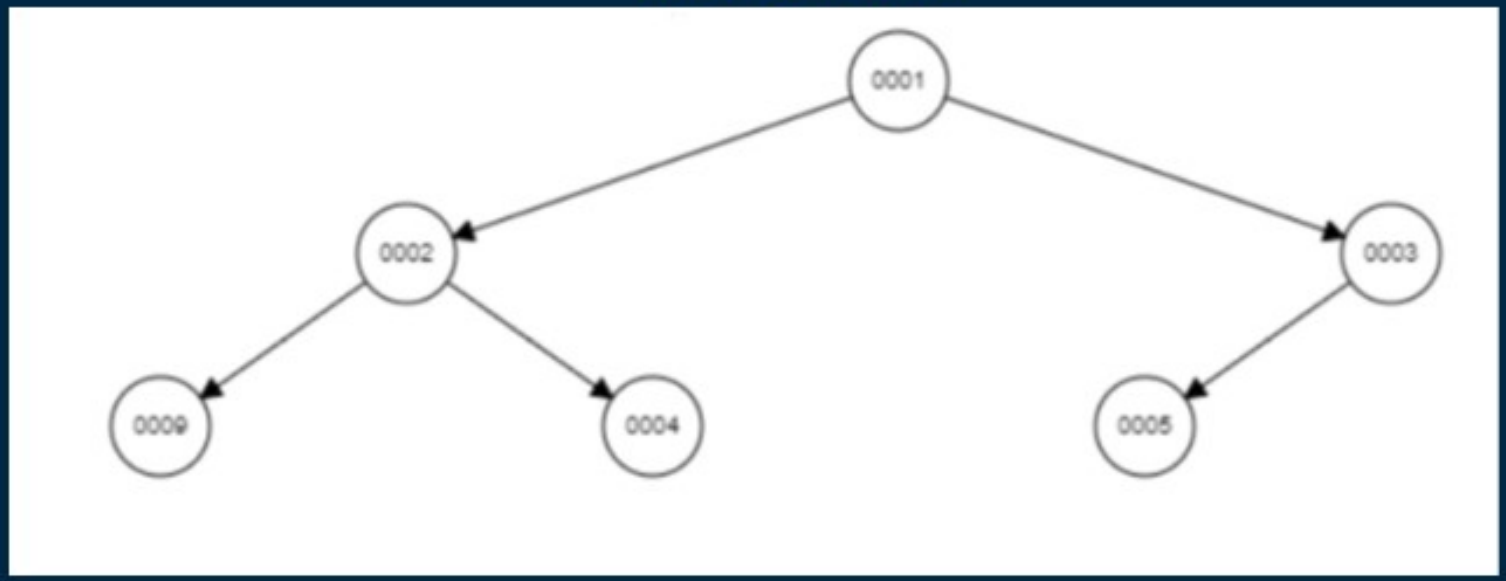
9	4	5	1	3	2
0	1	2	3	4	5

Heapify

3	9	2	1	4	5
0	1	2	3	4	5



final – Min Heap



1	2	3	9	4	5
0	1	2	3	4	5

Heapify - algoritm

```
Heapify(array, size, i)
        set i as celMaiMare
        stangaCopil = 2i + 1
        dreaptaCopil = 2i + 2
        if stangaCopil > array[celMaiMare] then
                set stangaCopilIndex as celMaiMare
        endif
        if dreaptaCopil > array[celMaiMare] then
                set dreaptaCopilIndex as celMaiMare
         endif
 swap array[i] and array[celMaiMare]
```

```
MaxHeap(array, size)
        loop from primul index al nodului care nu e frunza pana la  root
call Heapify
```

Aplicații structuri de date - Heap

- Implementare cozi cu prioritate
- Algoritm Dijkstra's
- Sortare Heap

Heapsort

03

Heapsort

Sortarea heap = o tehnică de sortare prin comparație bazată pe structura de date Binary Heap.

- similară cu sortarea prin selecție în care găsim mai întâi elementul minim și plasăm elementul minim la început.

Algoritm Heapsort

Considerăm arborele Max-Heap (cel mai mare element este root).

1. **Interschimbare**: Interschimbare ultimul element din arbore cu nodul root. Stocare la sfârșitul vectorului (poziția a n-a).

2. **Eliminați**: decrementăm dimensiunea stivei cu 1.

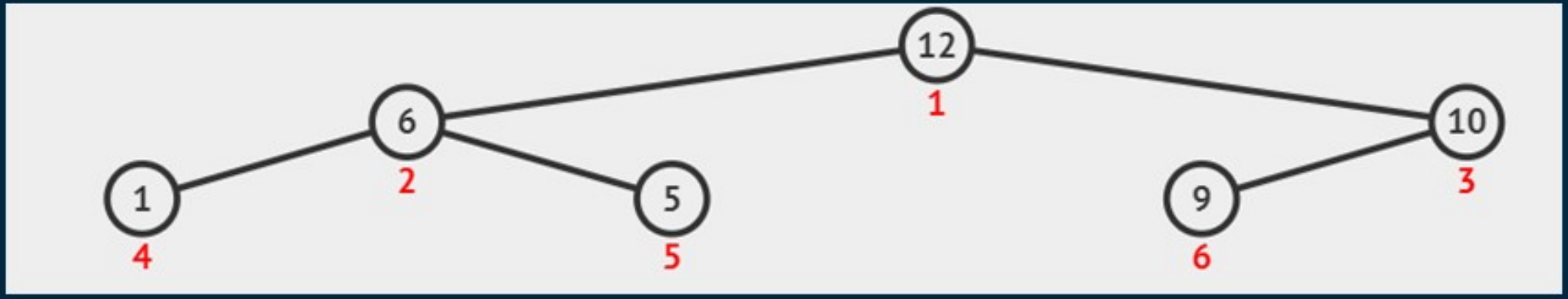
3. **Heapify**: Heapify elementul rădăcină din nou, astfel încât să avem cel mai înalt element la rădăcină.

4. Repetăm pașii 1-3 până toate elementele din listă sunt sortate.

Heapsort

1	12	9	5	6	10
0	1	2	3	4	5

Max Heap

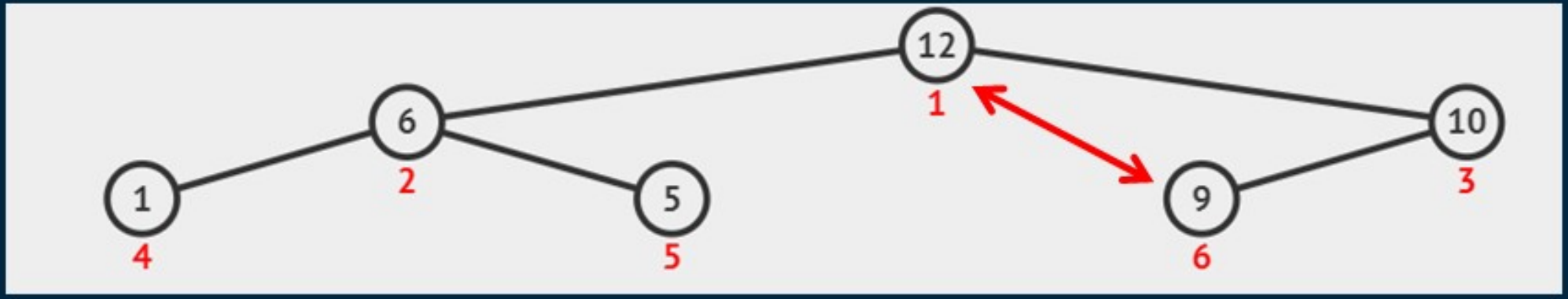


12	6	10	1	5	9
0	1	2	3	4	5

Heapsort

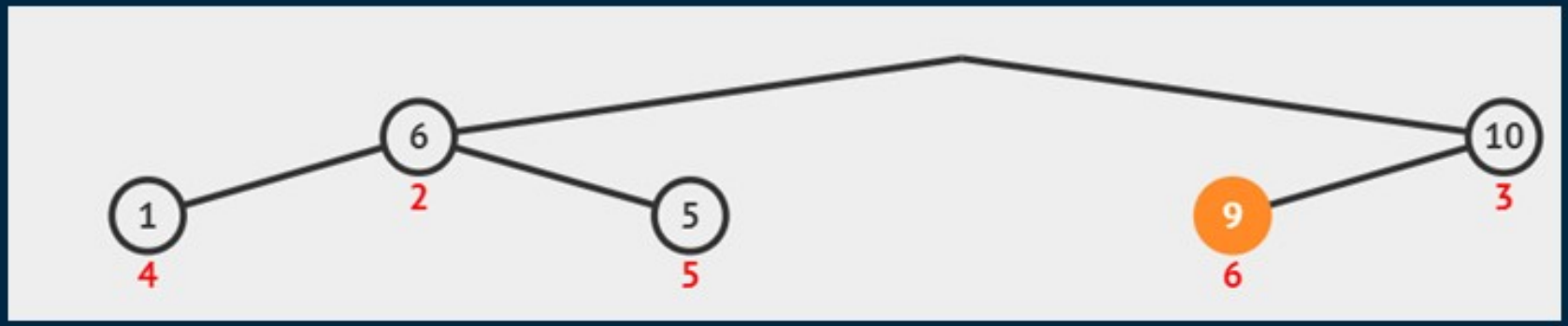
12	6	10	1	5	9
0	1	2	3	4	5

Max Heap



9	6	10	1	5	12
0	1	2	3	4	5

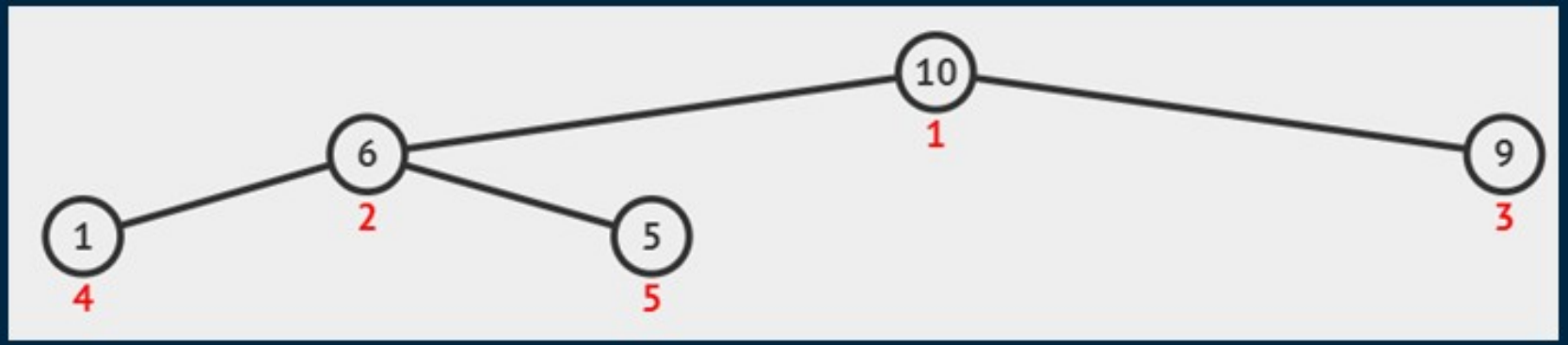
Heapsort



9	6	10	1	5	12
0	1	2	3	4	5

Heapsort

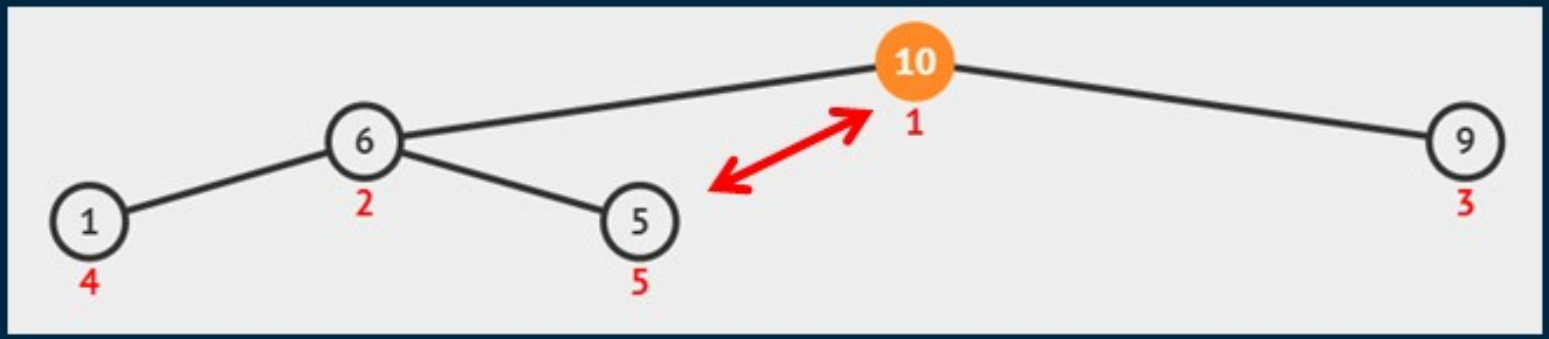
Heapify



9	6	10	1	5	12
0	1	2	3	4	5

Heapsort

9	6	10	1	5	12
0	1	2	3	4	5



9	6	10	1	5	12
0	1	2	3	4	5

Heapsort

9	6	5	1	10	12
0	1	2	3	4	5

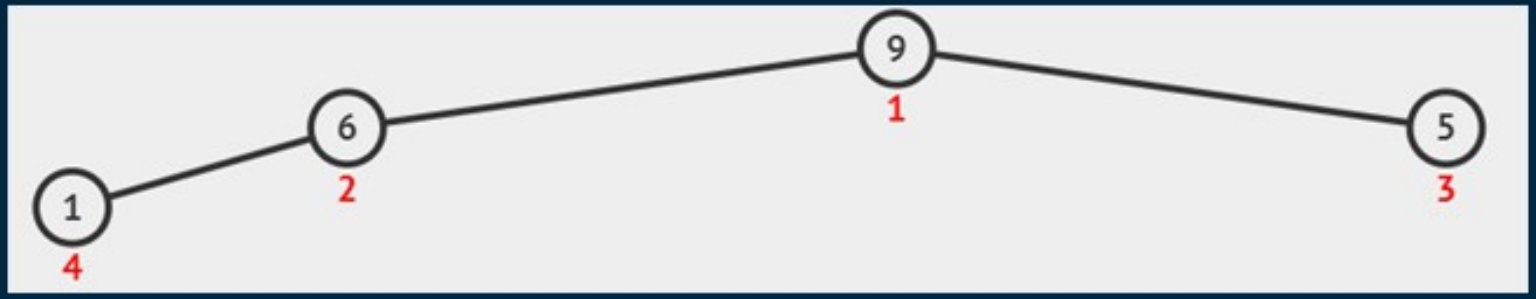
Heapsort

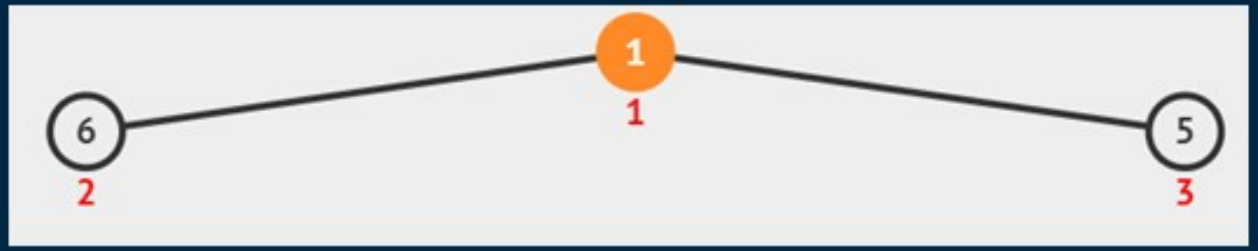
Heapify

9	6	5	1	10	12
0	1	2	3	4	5

Heapsort

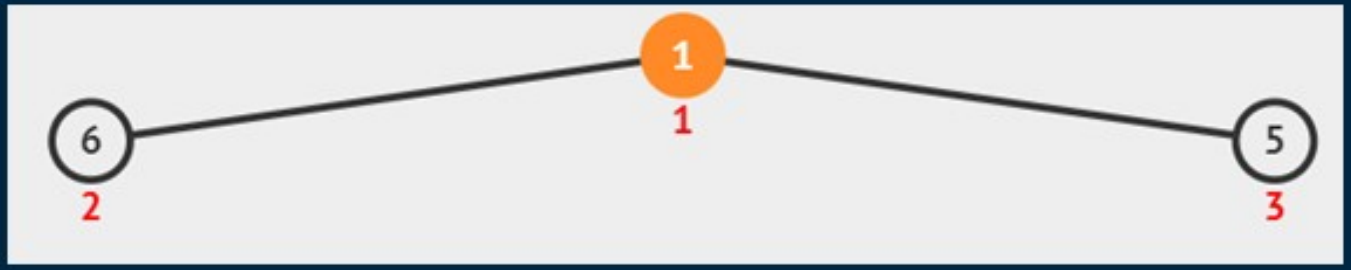
9	6	5	1	10	12
0	1	2	3	4	5



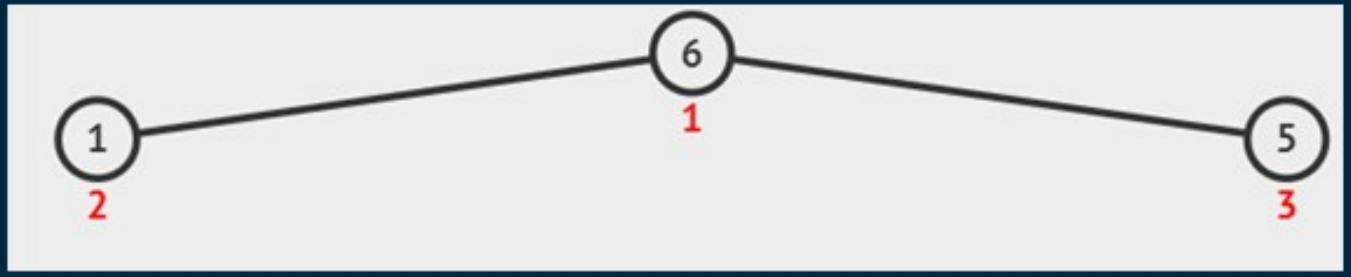


1	6	5	9	10	12
0	1	2	3	4	5

Heapsort



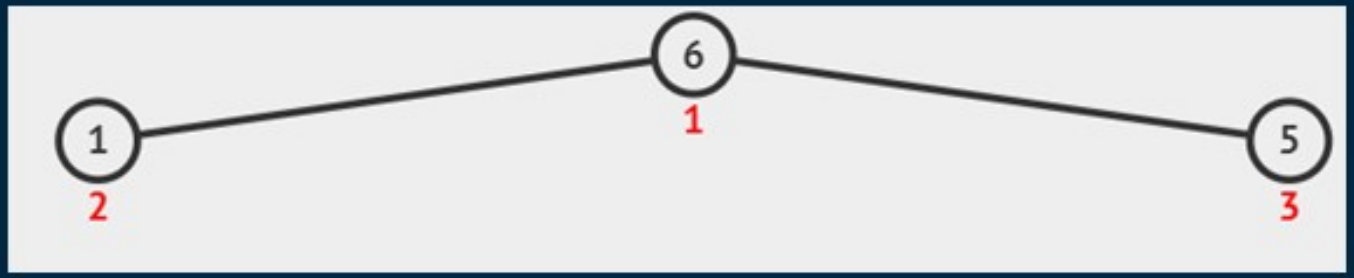
Heapify



1	6	5	9	10	12
0	1	2	3	4	5

Heapsort

1	6	5	9	10	12
0	1	2	3	4	5



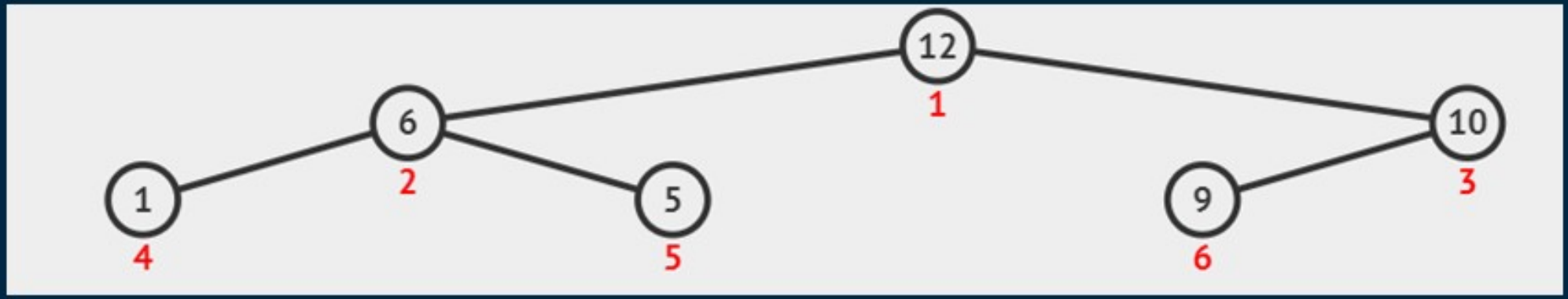
1	5	6	9	10	12
0	1	2	3	4	5

Heapsort

1	5	6	9	10	12
0	1	2	3	4	5

Heapsort

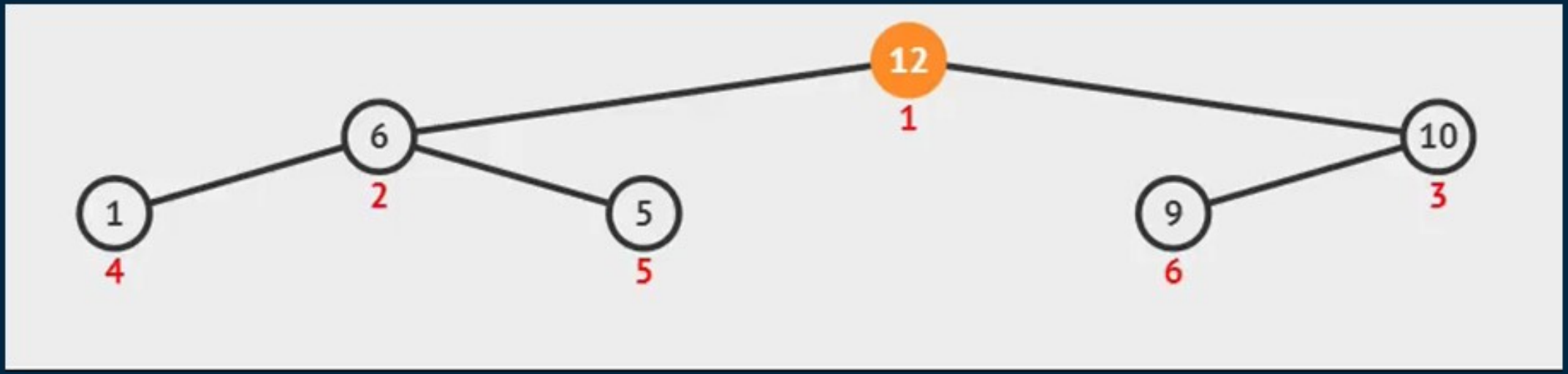
1	12	9	5	6	10
0	1	2	3	4	5



Vector sortat crescător

1	5	6	9	10	12
0	1	2	3	4	5

Heapsort



Vector sortat crescător

1	5	6	9	10	12
0	1	2	3	4	5

Avantaje Heapsort

- **Eficiență** – Timpul necesar pentru a efectua sortarea Heap crește logaritmic, pe măsură ce crește numărul de elemente de sortat.
- **Utilizarea memoriei** –este minimă, nu are nevoie de spațiu de memorie suplimentar pentru a funcționa
- **Simplitate** – nu utilizează concepte avansate de informatică, cum ar fi recursivitatea

Aplicații - Heapsort

- algoritmi hibrizi precum IntroSort.
- Sortarea sau aproape sortarea (K sorted) a vectorilor.
- k cele mai mari (sau cele mai mici) elemente dintr-un vector

Complexitate - Heapsort

Complexitate timp execuție	
Cazul cel mai favorabil	O(nlog n)
Cazul cel mai nefavorabil	O(nlog n)
Medie	O(nlog n)

Algoritmi similari

- Sortarea prin interclasare (Mergesort)
- Sortarea prin pivotare (Quicksort)

Cod Huffman

04

Codificare mesaj

- Codificarea unui mesaj compus dintr-un sir de caractere
- Codul utilizat
  - ASCII
    - 8 biti per caracter
    - se pot codifica 256 caractere
  - Unicode
    - 16 biti per caracter
    - Se pot codifica 65536 caractere
    - Codul ASCII este inclus
- ASCII si Unicode sunt *fixed-length codes*
  - toate caracterele sunt reprezentate de același număr de biți

Problema

- Presupunem ca avem de codificat un mesaj format din simbolurile **A**, **B**, **C**, **D si E** utilizand fixed-length
  - Cati biti sunt necesari pentru codificarea fiecarui caracter?
    - Cel putin 3 biti
    - 2 biti nu sunt suficienti (cu 2 se pot codifica maxim 4 cractere)
  - Câți biți sunt necesari pentru a codifica mesajul **AAACDBEEEDAABEE**?
    - 15 caractere a cate 3 biti fiecare
    - 15*3 = 45 biti

Dezavantaje cod fixed-length

- Risipa spatiului de memorie
  - Unicode utilizeaza spatiu dublu fata de ASCII
    - ineficient pentru mesajele text simplu care conțin doar caractere ASCII
- Pentru reprezentarea caracterelor se utilizeaza acelasi numar de biti
  - 'a' si 'i' apar mai frecvent decat 'x' si 'j' (in limba romana)
- **Solutia**: utilizare cod variable-length
  - număr variabil de biți pentru a reprezenta caracterele atunci când frecvența de apariție este cunoscută
  - coduri scurte pentru caractere care apar frecvent

Advantaje ale codului variable-length

- codurile scurte pot fi date caracterelor care apar frecvent
  - în medie, lungimea mesajului codificat este mai mică decât codarea cu lungime fixă

- **Problema potentiala:** de unde știm unde se termină un caracter și unde începe altul?
  - nu este o problemă dacă numărul de biți este fix!

A = 00
B = 01
C = 10
D = 11

1100101101110011011100110011

D A C D B A D B D A D A D

Prefix

- Un cod poate avea **prefix** dacă niciun cod de caracter nu este prefixul (începutul codului) pentru alt caracter
- Exemple:

Simbol	Cod
P	000
Q	11
R	01
S	001
T	10

01001101100010

R S T Q P T

- 000 nu este prefix pentru 11, 01, 001 sau 10
- 11 nu este prefix pentru 000, 01, 001 sau 10 ...

Codificare fara Prefix

- Urmatoarea codificare nu are proprietatea prefix
- Exemple:

Simbol	Cod
P	0
Q	1
R	01
S	10
T	11

- Modelul 1110 poate fi decodat ca QQQP, QTP, QQS sau TS
- Rezulta o problema!!!!

Solutia – Arbori Huffman

- Arbore binar
  - Fiecare frunza contine un simbol (caracter)
  - Muchia stanga are "costul" 0
  - Muchia stanga are "costul" 1
- Codul oricarui simbol se obtine urmând calea de la rădăcină la frunza care conține simbolul
- Codul are proprietatea prefix
  - Nodul frunză nu poate apărea pe calea către o altă frunză
  - *notă:* codurile cu lungime fixă sunt reprezentate de un arbore Huffman complet și au în mod clar proprietatea prefixului

Constructia Arborelui Huffman

- Deterinarea frecventei de aparitie a fiecarui character din mesaj
- Se construieste arborele binary considerand ca acesta este initial vid
  - Fiecare nod va contine frecventa de aparitie a unui caracter
- Recursiv
  - selectare 2 caractere cu frecventa de aparitie cea mai mica
  - generam un nou arbore binar cu elementele generate anterior și stocam suma frecvențelor lor în rădăcină
- Recursivitatea de finalizeaza cand exista un singur arbore generat
  - Arborele rezultat este arborele Hufman

Exemplu

- Construire arbore binary Huffman pentru textul:

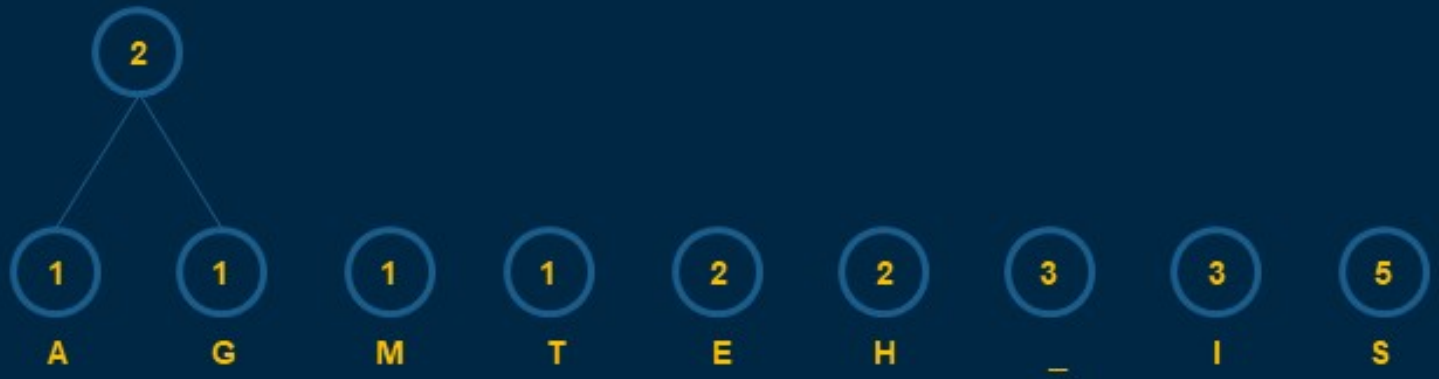
  This is his message

- Frecventa de aparitie

A	G	M	T	E	H	_	I	S
1	1	1	1	2	2	3	3	5

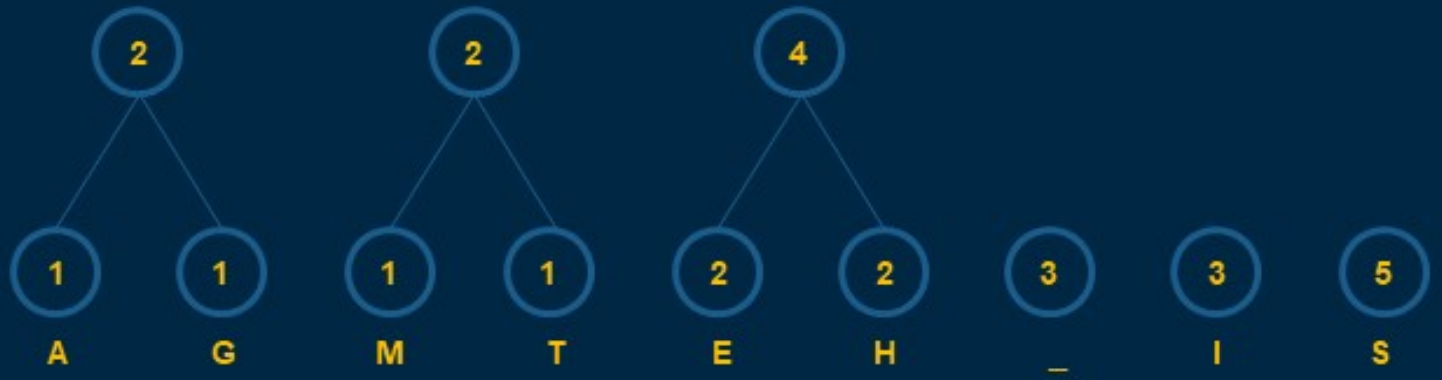
- Generam arborele (pentru inceput avem mai multi arbori cu un singur nod)

Pas 1

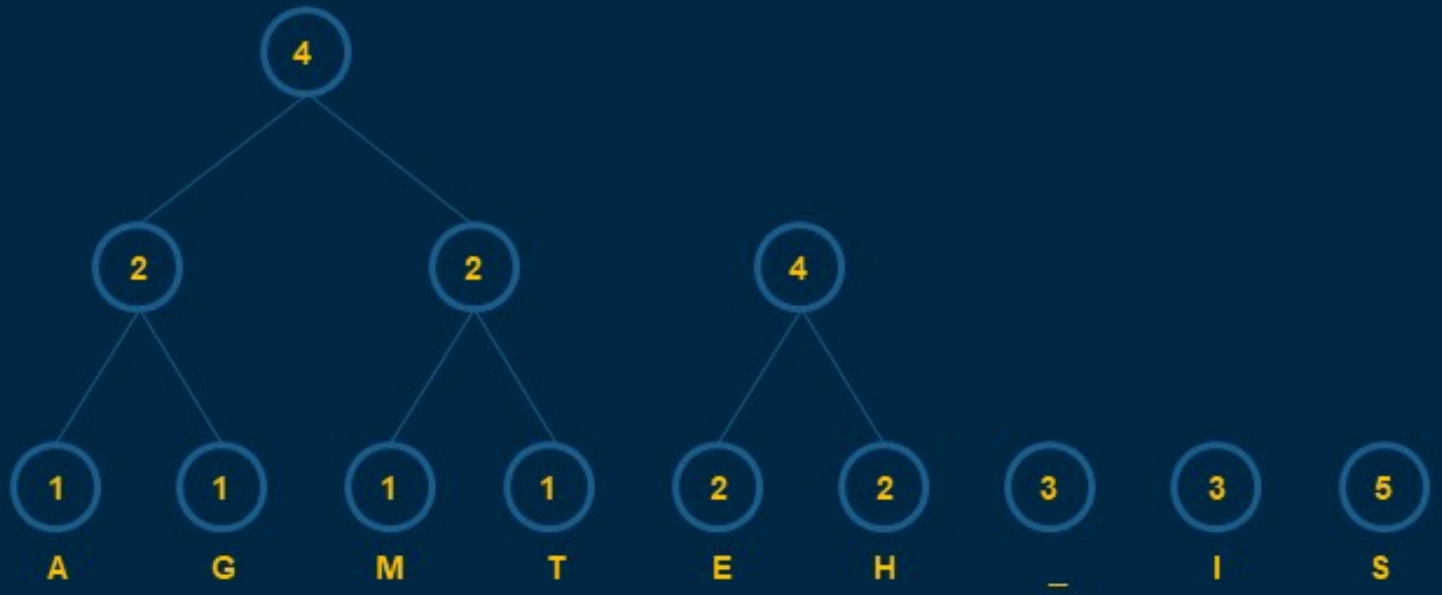


Pas 2

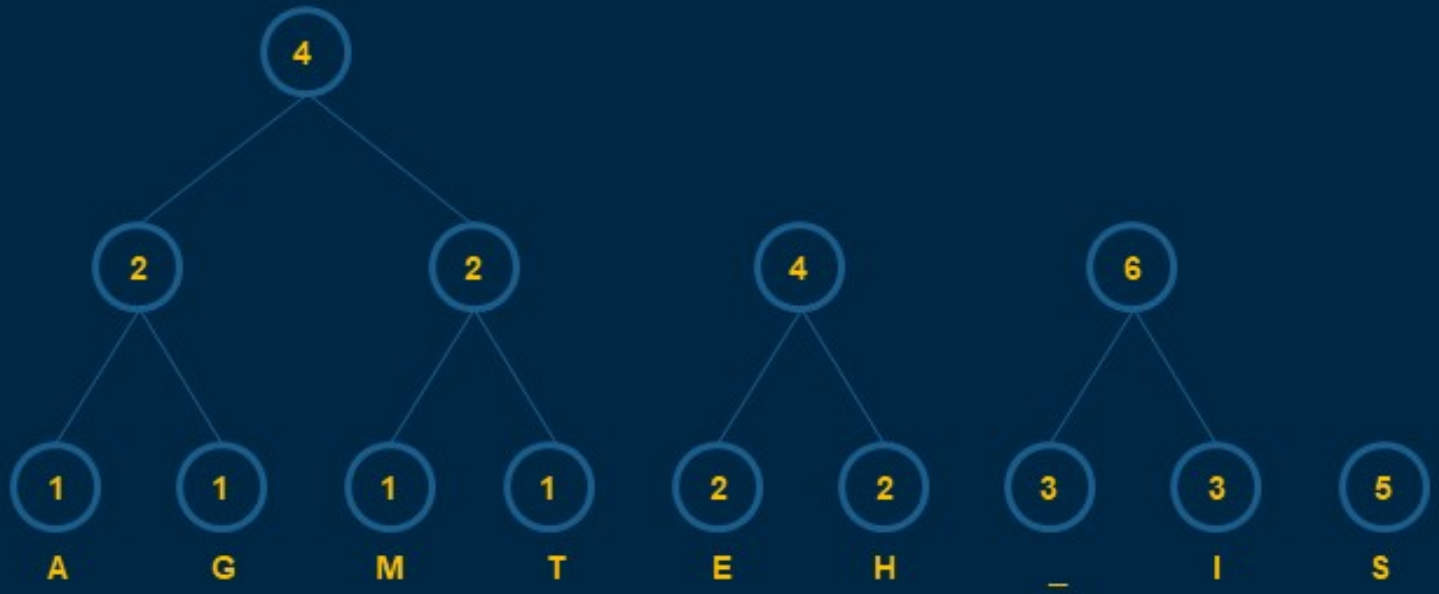
Pas 3



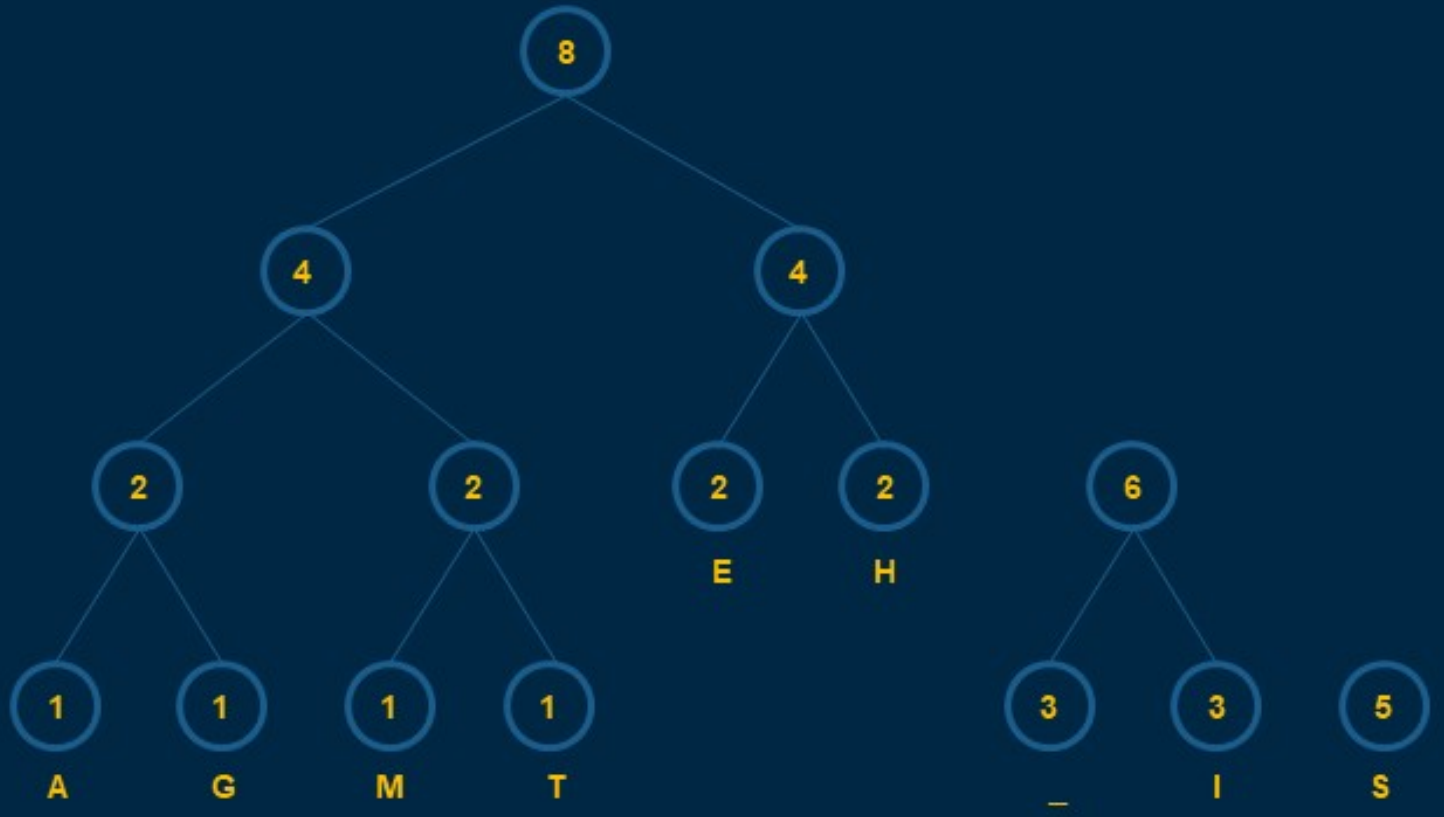
Pas 4



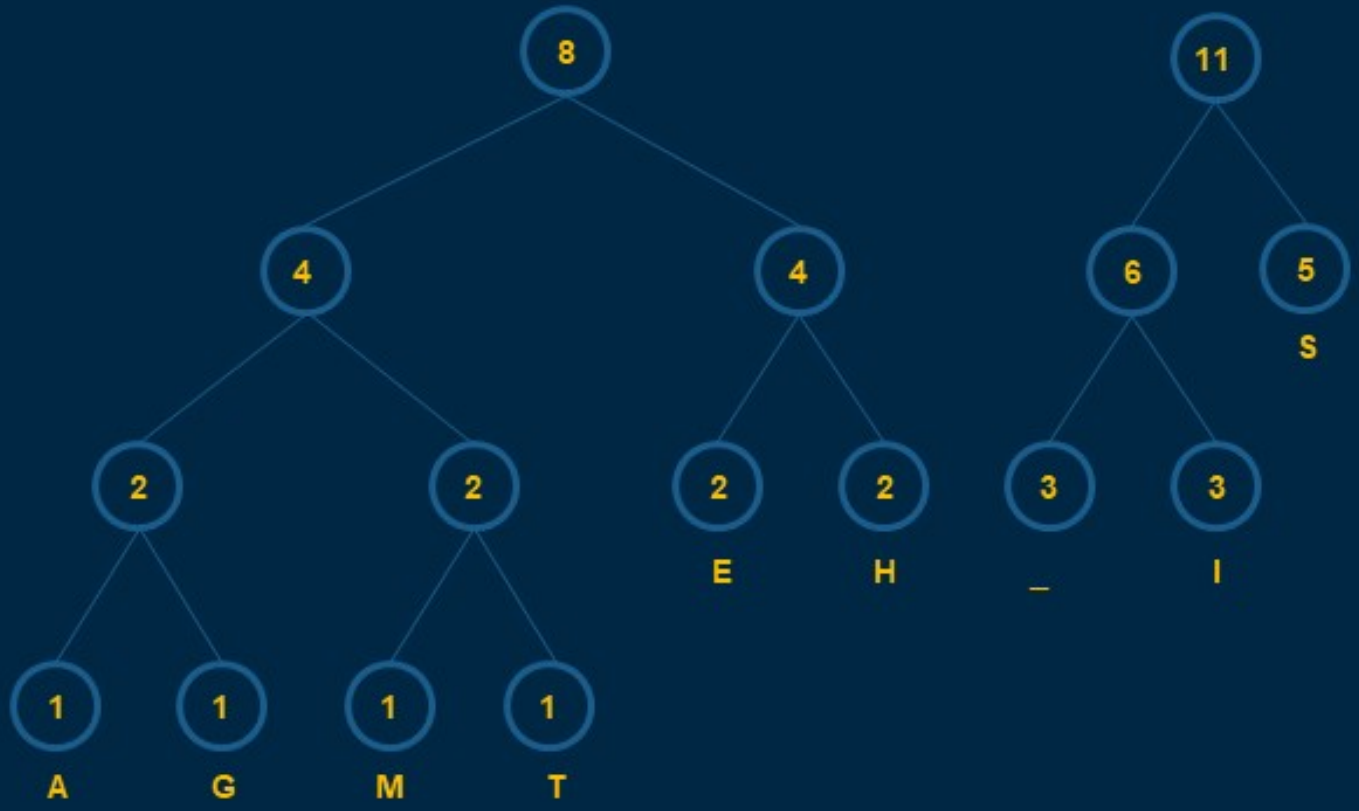
Pas 5



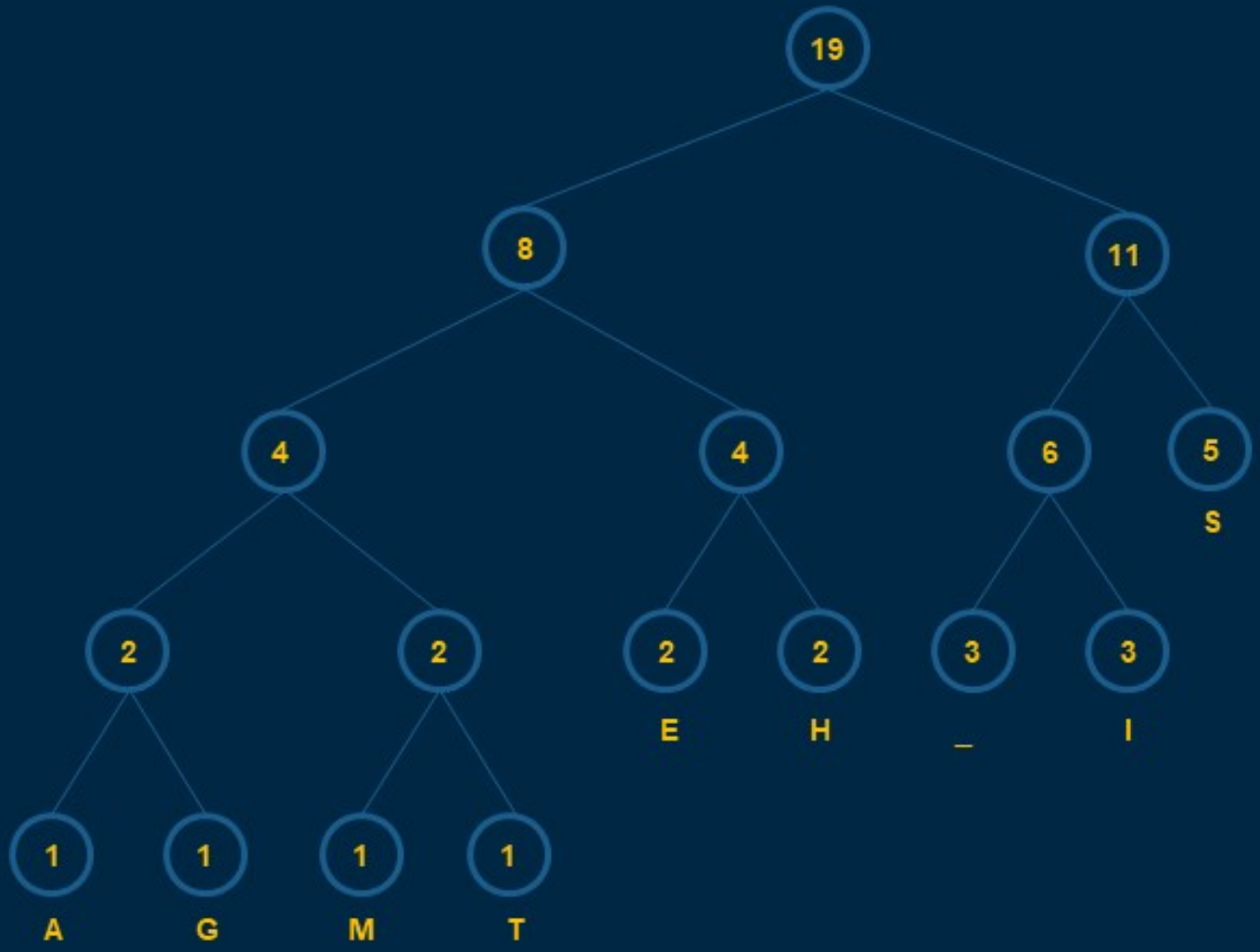
Pas 6



Pas 7

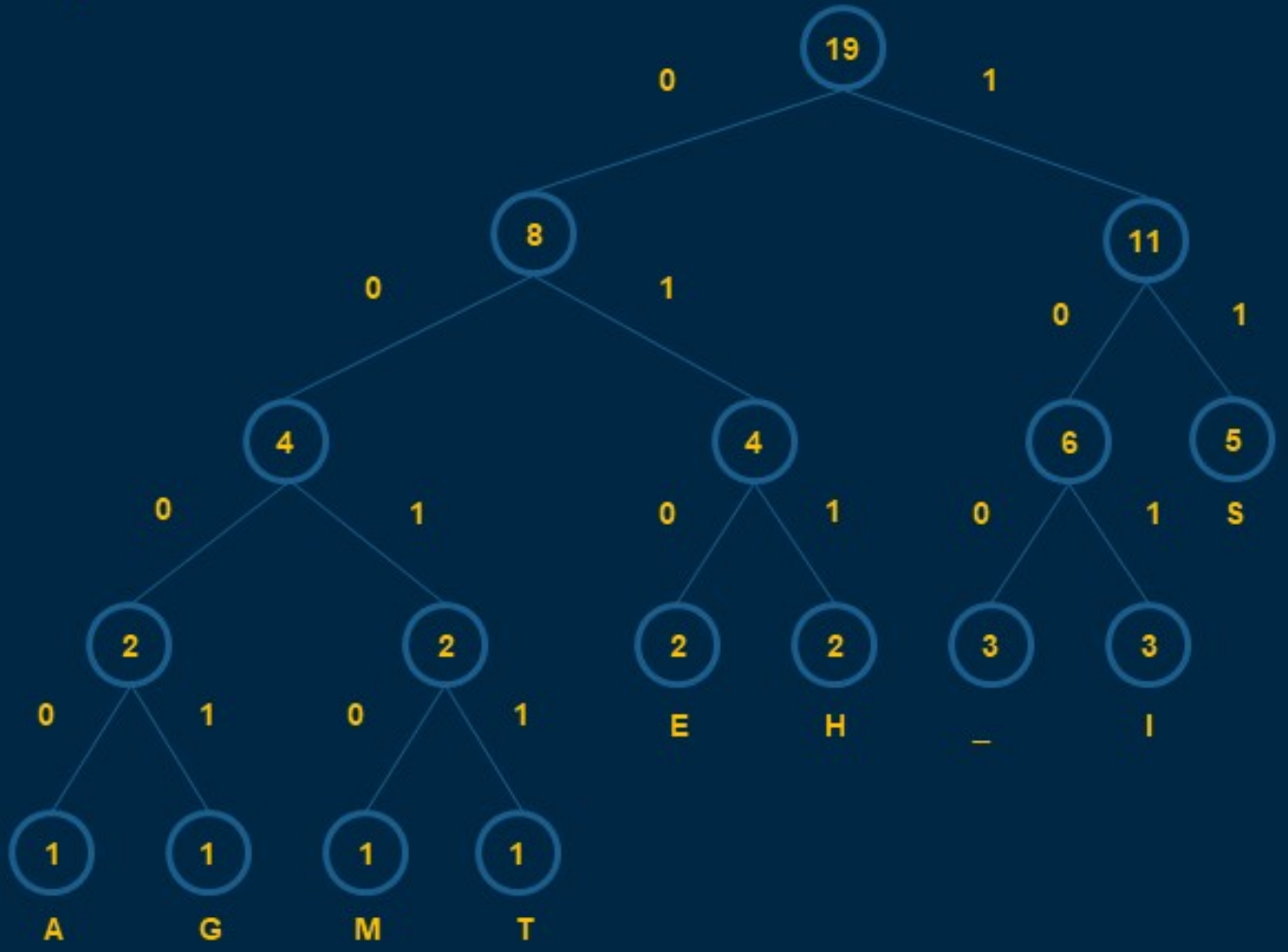


Pas 8



Cost muchii (etichete)

S	11
E	010
H	011
_	100
I	101
A	0000
G	0001
M	0010
T	0011



Huffman code & encoded message

This is his message

S	11
E	010
H	011
_	100
I	101
A	0000
G	0001
M	0010
T	0011

0011011101111001011110001110111100001001011110000001010

Intrebari?

dorin.lordache@365.univ-ovidius.ro

Mulțumesc