# **SDA** (PC2)
# **Curs 10**
# **Arbori**

Iulian Năstac

# Arbori

## Recapitulare

- Definiția 1:

  Arborele este un graf orientat, aciclic și simplu conex.

- Definiția 2:

  Un arbore este un ansamblu de structuri de date de natură recursivă şi dinamică.

- Definiția 3:

  Prin **arbore** înţelegem o mulţime finită şi nevidă de elemente numite noduri:

  **ARB = { A1, A2, A3, ..., An }** , unde n > 0 ,

care are următoarele proprietăţi:

- Există un nod şi numai unul care se numeşte **rădăcina** arborelui.
- Celelalte noduri formează submulţimi disjuncte ale lui **ARB**, care formează fiecare câte un arbore. Arborii respectivi se numesc **subarbori** ai rădăcinii.

# Nodurile unui arbore

## Recapitulare

- Într-un arbore există noduri cărora nu le mai corespund subarbori. Un astfel de nod se numeşte **terminal** sau **frunză**.
- În legătură cu arborii s-a stabilit un limbaj conform căruia un ***nod rădacină*** se spune că este un ***nod tată***, iar subarborii rădăcinii sunt ***descendenţii*** acestuia.
- ***Rădăcinile descendenţilor*** unui ***nod tată*** sunt ***fiii*** lui.

# Relația de ordine
## Recapitulare

- Dacă există o relație de ordine între subarborii oricărui nod al arborelui atunci arborele este **ordonat**.

- Pentru un nod al unui arbore ordonat se notează rădăcina primului subarbore ca fiind fiul cel mai în vârstă, iar rădăcina ultimului subarbore drept fiul cel mai tânăr.

# Arbori binari

## Recapitulare

Un **arbore binar** este o mulţime finită de elemente care sau este **vidă**, sau conţine un element numit **rădăcina**, iar celelalte elemente (dacă există) se împart în două submulţimi disjuncte, care fiecare la rândul ei, este un arbore binar.

# Particularități

## Recapitulare

- Un arbore binar nu se defineşte ca un caz particular de arbore ordonat. Astfel, un arbore nu este niciodată vid, spre deosebire de un arbore binar care poate fi şi vid.

- Orice arbore ordonat poate fi întotdeauna reprezentat printr-un arbore binar.

# Modalitatea de transformare a unui arbore ordonat într-un arbore binar

(Recapitulare)

1. Se leagă între ei frații descendenți ai aceluiași nod tată și se suprimă legăturile lor cu nodul tată, cu excepția primului fiu.
2. Fostul prim fiu devine fiul stâng al nodului tată, iar ceilalți foști frați formează, în mod consecutiv, subarbori drepți. Fiecare dintre foștii frați devine descendent drept (fiu drept) al fostului frate mai mare.

# Utilizarea structurilor pentru realizarea unui arbore binar

Nodul unui arbore binar poate fi reprezentat ca o dată structurală de tipul NOD care se defineşte în felul următor:

```
typedef struct nod
    {
        declaratii ;
        struct nod *st;
        struct nod *dr;
    } NOD;
```

unde:

- st - este pointerul spre fiul stâng al nodului curent;
- dr - este pointerul spre fiul drept al aceluiaşi nod.

Asupra arborilor binari se pot defini mai multe operaţii:

1. inserarea unui nod frunză într-un arbore binar;
2. accesul la un nod al unui arbore;
3. parcurgerea unui arbore;
4. ştergerea unui arbore.

- Operaţiile de **inserare** şi **acces** la un nod, au la bază un **criteriu** care să definească locul în arbore al nodului în cauză.

- Acest **criteriu** este dependent de problema concretă, la care se aplică arborii binari, pentru a fi rezolvată.

# Funcția criteriu

Definim o funcţie pe care o vom denumi **criteriu**. Aceasta are doi parametri care sunt pointeri spre tipul NOD.

Fie p1 primul parametru al funcţiei **criteriu** şi p2 cel de-al doilea parametru al ei. Atunci, funcţia **criteriu** returnează:

- **-1** - dacă p2 pointează spre o dată de tip NOD care poate fi un nod al subarborelui stâng al nodului spre care pointează p1;
- **1** - dacă p2 pointează spre o dată de tip NOD care poate fi un nod al subarborelui drept al nodului spre care pointează p1;
- **0** - dacă p2 pointează spre o dată de tip NOD care nu poate fi nod al subarborilor nodului spre care pointează p1.

# Exemplu:

Considerăm următorul șir de numere:

**20, 30, 5, 20, 4, 30, 7, 40, 25, 28, ...**

Să se creeze un arbore în nodurile căruia vor fi trecute numerele respective împreună cu frecvența lor de apariție.

Practic nodurile acestui arbore vor avea două câmpuri utile:

- primul care va conține numărul;
- celălalt care va conține frecvența de apariție a numărului.

# Abordarea problemei:

a. **p1** - este pointer spre un nod al arborelui în care se face inserarea (iniţial p1 pointează spre rădacina arborelui).

b. **p2** - este un pointer spre nodul curent (nodul de inserat).

c. dacă **p2->val < p1->val**, atunci se va încerca inserarea nodului curent în subarborele stâng al nodului spre care pointează p1.

d. dacă **p2->val > p1->val**, atunci se va încerca inserarea nodului curent în subarborele drept al nodului spre care pointează p1.

e. dacă **p2->val = p1->val**, atunci nodul curent nu se mai inserează în arbore deoarece există deja un nod corespunzător valorii citite curent.

Nodul curent nu se mai inserează în arbore în cazul descris la punctul **e** (situaţie care în general corespunde cazului când funcţia **criteriu** returnează valoarea zero). În acest caz, nodurile spre care pointează p1 şi p2 le vom considera **echivalente**.

În cazul exemplului precedent, funcția criteriu este:

```
int criteriu(NOD *p1, NOD *p2)
{
        if(p2->nr < p1->nr) return(-1);
         if(p2->nr > p1->nr) return(1);
        return(0);
}
```

# Funcția pentru tratarea echivalenței

- De obicei, la întâlnirea unei perechi de noduri echivalente, nodul din arbore (spre care pointează p1) este supus unei prelucrări, iar nodul curent (spre care pointează p2) este eliminat.
- Pentru a realiza o astfel de prelucrare este necesar să se apeleze o funcţie care are ca parametri pointerii p1 şi p2 şi care returnează un pointer spre tipul NOD (de obicei se returnează valoarea lui p1).
- Vom numi această funcţie: **echivalenta**. Ea este dependentă de problema concretă, ca şi funcţia **criteriu**.

**Exemplu:**

```
NOD *echivalenta(NOD *q, NOD *p)
{
        q -> frecventa ++;
        elibnod(p);
        return(q);
}
```

Observație:
Funcția de mai sus realizează următoarele:
- eliberează zona de memorie ocupată de nodul spre care pointează **p**;
- incrementează valoarea componentei: **q -> frecventa**;
- returnează valoarea lui **q**.

# Alte funcții

- În afară de funcţiile enumerate anterior, vom folosi nişte funcţii specifice pentru operaţii asupra arborilor.

- Exemple tipice de funcții: **elibnod** și **incnod**

- Unele funcții utilizează o variabilă globală care este un pointer spre rădăcina arborelui.

Exemplu de funcție folosită la laborator:

```
void elibnod(NOD *p)
/* elibereaza zonele din memoria heap ocupate de
nodul spre care pointează p */
{
 free(p -> cuvant);
 free(p);
}
```

# Intrarea în arbore
## (Recapitulare)

- Numim **prad** o variabilă globală către rădăcina arborelui binar. Ea se defineşte astfel:

  **NOD *prad;**

- În cazul în care într-un program se prelucrează simultan mai mulţi arbori, se vor utiliza mai multe variabile globale de tip **prad**; interfaţa dintre funcţii realizându-se cu ajutorul unui parametru care este pointer spre tipul NOD şi căruia i se atribuie, la apel, adresa nodului rădăcină al arborelui prelucrat prin funcţia apelată.

# 1. Inserarea unui nod frunză într-un arbore binar

Funcţia **insnod** ( declarată NOD *insnod() ) inserează un nod nou **p** în arbore, conform următorilor paşi:

1. Se alocă zona de memorie pentru nodul care urmează să se insereze în arbore. Fie **p** pointerul care are ca valoare adresa de început a zonei respective.

2. Se apelează funcţia **incnod**, cu parametrul **p**, pentru a încărca datele curente în zona spre care pointează p. Dacă **incnod** returnează valoarea 1, se trece la pasul 3. Altfel se revine din funcţie cu valoarea zero.

3. Se fac atribuirile:

**p->st = p->dr = 0**

deoarece nodul de inserat este nod frunză.

4. **q = prad**

5. Se determină poziţia, în arbore, în care sa se faca inserarea. În acest scop se caută nodul care poate fi nod tată pentru nodul curent:

**i = criteriu(q,p)**

6. Dacă **i<0**, se trece la pasul 7; altfel se sare la pasul 8.

7. Se încearcă inserarea nodului spre care pointează **p** (nodul curent) în subarborele stâng al arborelui spre care pointează **q**.
- Dacă **q -> st** are valoarea zero, atunci nodul spre care pointează **q** nu are subarbore stâng şi nodul curent devine fiu stâng al celui spre care pointează **q** (**q->st = p**). Se revine din funcţie returnându-se valoarea lui p.
- Altfel se face atribuirea **q = q->st** (se trece la fiul stâng al nodului spre care pointează **q**) şi se sare la pasul 5.

8. Dacă **i>0**, se trece la pasul 9; altfel se sare la pasul 10.

9. Se încearcă inserarea nodului spre care pointează **p** (nodul curent) în subarborele drept al arborelui spre care pointează **q**.
- Dacă **q -> dr** are valoarea zero, atunci nodul spre care pointează **q** nu are subarbore drept şi nodul curent devine fiu drept al celui spre care pointează **q** (**q->dr=p**). Se revine din funcţie returnându-se valoarea lui **p**.
- Altfel se face atribuirea **q = q->dr** (se trece la fiul drept al nodului spre care pointează **q**) şi se sare la pasul 5.

10. Nodul curent nu poate fi inserat în arbore. Se apelează funcţia numită **echivalenta** şi se revine din funcţie cu valoarea returnată de funcţia **echivalenta**.

# 2. Accesul la un nod al unui arbore binar

- Accesul la un nod presupune existența unui criteriu care să permită localizarea în arbore a nodului respectiv.

- Se va utiliza funcția **criteriu**.

- Funcția care realizează căutarea va identifica în arbore un nod echivalent cu cel spre care pointează un pointer **p** (folosit ca argument al funcției de căutare).

Funcția de căutare (numită **cauta**) va returna:

- pointerul spre nodul determinat;

- 0 dacă nu există un nod echivalent cu **p**.

```
NOD *cauta(NOD *p)
{
        extern NOD *prad;
        NOD *q;
        int i;
        if (prad = = 0) return 0;  /*arborele este vid*/
        for (q = prad; q;  )
          {
                if ( (i = criteriu(q, p)) = = 0)    return q;
                else   if (i < 0)       q = q -> st;
                       else             q = q -> dr;
          }
        return 0;
}
```

# 3. Parcurgerea unui arbore binar

- Prelucrarea informaţiei păstrată în nodurile unui arbore binar se realizează parcurgând nodurile arborelui respectiv.

- Parcurgerea nodurilor unui arbore binar se poate face în mai multe moduri, dintre care se remarcă:
  - **parcurgerea în preordine;**
  - **parcurgerea în inordine;**
  - **parcurgerea în postordine.**

# Parcurgerea în preordine

**Parcurgerea în preordine** înseamnă accesul la rădăcină şi apoi parcurgerea celor doi subarbori ai ei, întâi subarborele stâng, apoi cel drept. Subarborii, fiind ei înşişi arbori binari, se parcurg în acelaşi mod.

# Parcurgerea în inordine

**Parcurgerea în inordine** înseamnă parcurgerea mai întâi a subarborelui stâng, apoi accesul la rădăcină şi în continuare parcurgerea subarborelui drept. Cei doi subarbori se parcurg în acelaşi mod.

# Parcurgerea în postordine

**Parcurgerea în postordine** înseamnă parcurgerea mai întâi a subarborelui stâng, apoi a subarborelui drept şi în final accesul la rădăcina arborelui. Cei doi subarbori se parcurg în acelaşi mod.

- Accesul la un nod permite prelucrarea informaţiei conţinute în nodul respectiv.

- În acest scop se poate apela o funcţie care este dependentă de problema concretă care se rezolvă cu ajutorul parcurgerii arborelui (de exemplu funcţia **prelucrare**).

**void prelucrare( NOD *p)**

# Parcurgerea unui arbore binar in preordine

```
void preord(NOD *p)
{
 if(p != 0)
        {
         prelucrare(p);
         preord(p -> st);
         preord(p -> dr);
        }
}
```

# Parcurgerea unui arbore binar in inordine

```
void inord(NOD *p)
{
 if(p != 0)
        {
         inord(p -> st);
         prelucrare(p);
         inord(p -> dr);
        }
}
```

# Parcurgerea unui arbore binar in postordine

```
void postord (NOD *p)
{
 if(p != 0)
        {
         postord (p -> st);
         postord (p -> dr);
         prelucrare(p);
        }
}
```

# Exemplu:

Reluăm problema cu șirul de numere:

**20, 30, 5, 20, 4, 30, 7, 40, 25, 28, ...**

# Observații:

- Programul de la laborator (cu noduri care conțin cuvinte împreună cu frecvența lor de apariție într-un text) se rezolvă mai ușor prin intermediul arborilor (deoarece operația de căutare într-o listă este ineficientă).

- Procesul de căutare într-un arbore necesită mai puțini pași decât procesul de căutare într-o listă.

# 4. Ștergerea unui arbore binar

- pentru ștergerea unui arbore binar este necesară parcurgerea lui și ștergerea fiecărui nod al arborelui respectiv.

- se apelează funcția elibnod.

- arborele se parcurge în postordine

# Ștergerea unui arbore binar in postordine

```
void sterge_arb(NOD *p)
{
 if(p != 0)
        {
         sterge_arb (p -> st);
         sterge_arb (p -> dr);
         elibnod(p);
        }
}
```

# Observații:

- Funcția **sterge_arb** nu atribuie valoarea zero variabilei globale prad.

- Această atribuire se va face obligatoriu imediat după ce se apelează funcția **sterge_arb**

- **Arbore binar degenerat** – dacă și numai dacă toți subarborii lui sunt de același fel (adică sunt numai stângi, sau numai drepți).

# 5. Ștergerea unui nod precizat printr-o cheie

- Cheia după care se face căutarea este unică și se găsește în fiecare nod:

```
typedef struct nod
        {
                declaratii ;
                tip cheie;
                struct nod *st;
                struct nod *dr;
        }  NOD;
```

unde tipul cheii poate fi char, int, float sau double.

- **Se pot șterge doar noduri care sunt frunză**!

- Ștergerea unui nod care nu este frunză implică operații suplimentare complicate pentru refacerea arborelui.

```
void cauta_sterge(NOD *p, int c)    /*cheia este de tip int*/
{
     if (p != 0)
       {
          if (( p -> st = = p -> dr ) && ( p -> st = = 0 ) && ( p -> cheie = = c))
                        {
                            elibnod (p);
                            return;
                        }
           cauta_sterge(p -> st, c);
           cauta_sterge(p -> dr, c);
       }
}
```

**Observație**: această operație de căutare și ștergere se face în **preordine**.

# **Observație**:

- La o analiză atenta, funcției anterioare îi lipsește ceva: după ce o un nod frunză identificat a fost șters, tatăl său ar trebui să aibă pointerul recursiv zero catre direcția proaspăt ștearsă!

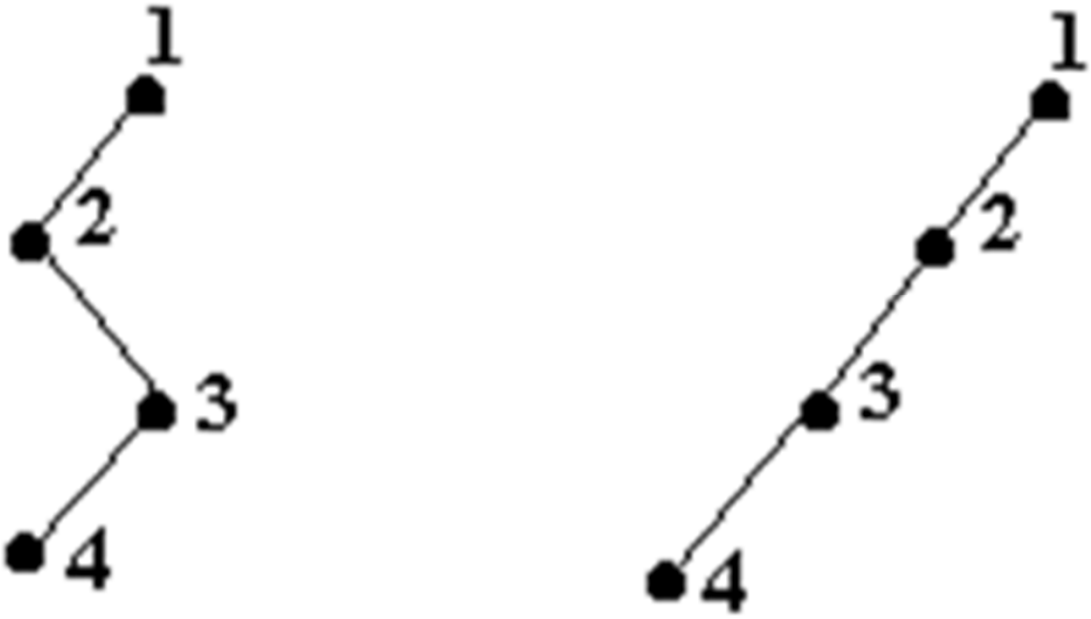
- Cum rezolvați această problemă?

# Precizare

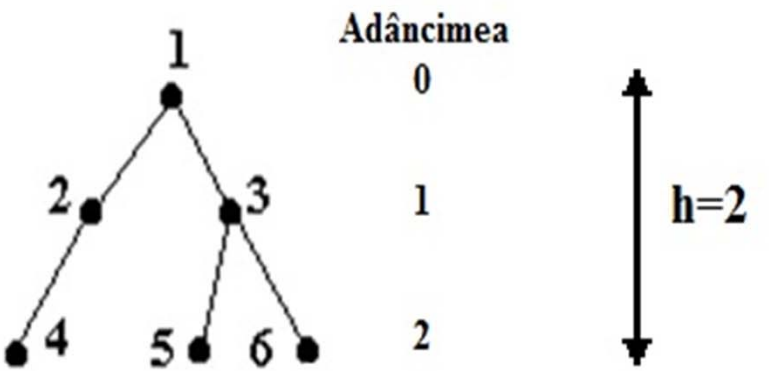
- În anumite aplicații sunt necesari arbori mai aplatizați, de înălțime mică, dar cu număr mare de noduri.
- În aceste cazuri se pot înlocui arborii binari cu unii care să permită fiecărui nod un număr mai mare de descendenți direcți.
- Procedura este relativ simplă, iar în loc de cei doi pointeri recursivi (către subarborele stâng și către cel drept), se utilizează un vector de pointeri recursivi (cu o anumită lungime maximă) care permite tipului de structură, utilizată pentru definirea nodurilor, un număr arbitrar de mare de descendenți direcți.

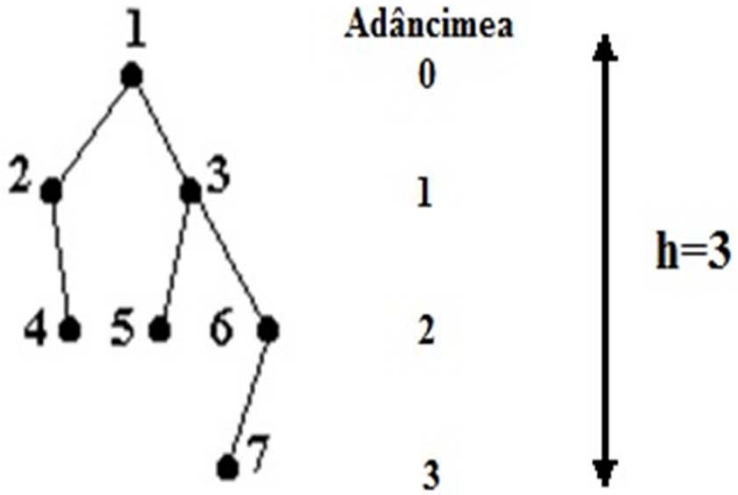
# Arbori binari degenerați

- sunt arbori binari cu n vârfuri dispuse pe n niveluri.



# Adâncimea și înălțimea unui arbore




- Vom nota înălțimea unui arbore binar cu **h** sau cu **i**

# Adâncimea și înălțimea

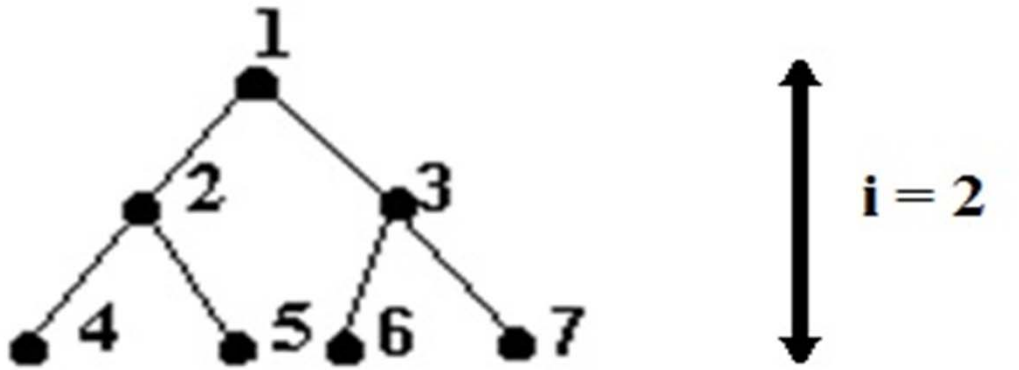
- Adâncimea unui nod este numărul de muchii de la nod la nodul rădăcină al arborelui. Un nod rădăcină va avea adâncimea 0.

- Înălțimea unui nod este numărul de margini pe cea mai lungă cale de la nod la frunză. Cel mai jos nod frunză va avea înălțimea 0.

- Pentru un arbore binar, înălțimea și adâncimea sa (din punct de vedere global) măsoară același lucru.

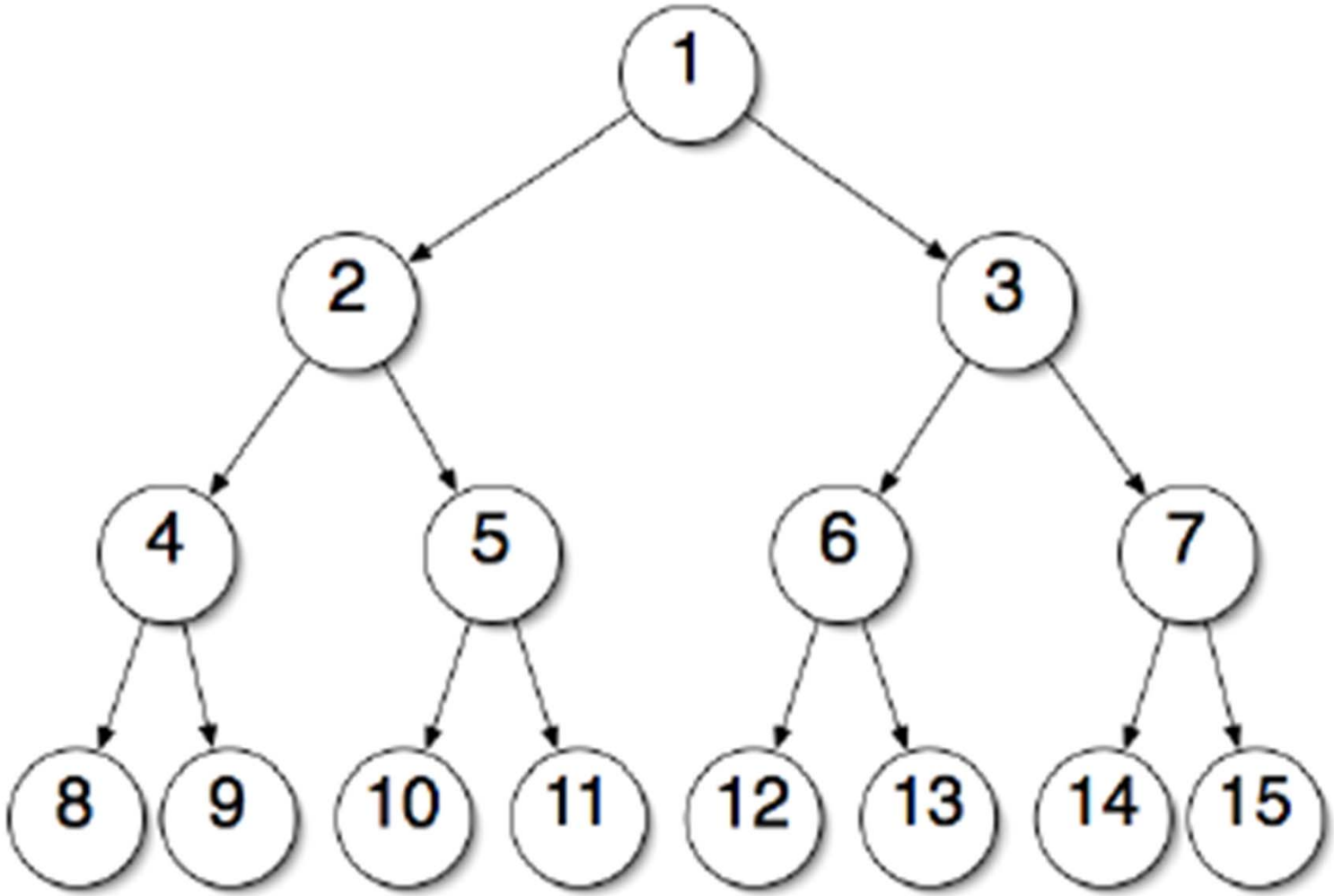
# Arbori binari speciali

**P**: Un arbore binar de înălțime **i** are maximum $2^{i+1}-1$ varfuri (sau noduri)




Observație:

Orice nivel de adâncime **k** are maximum $2^k$ noduri


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

# Demonstrație prin inducție:

- Se observă foarte ușor că propoziția anterioară este valabilă pentru i = 0, 1, 2 ...
- Dacă pentru înălțimea **i** avem $\mathbf{2^{i+1}-1}$ noduri atunci pentru înălțimea **i+1** avem $\mathbf{2^{i+2}-1}$ noduri

  Ne bazăm pe faptul că orice nivel de adâncime **k** are maximum $\mathbf{2^k}$ noduri.

Se observă ușor că pentru înălțimea **i+1** avem:

**Nr_de_noduri** = $\mathbf{2^{i+1} - 1 + 2^{i+1} = 2^{i+1}(1 + 1) - 1 =}$

$\mathbf{2^{i+1} \cdot 2 - 1 = 2^{i+2} - 1}$ noduri

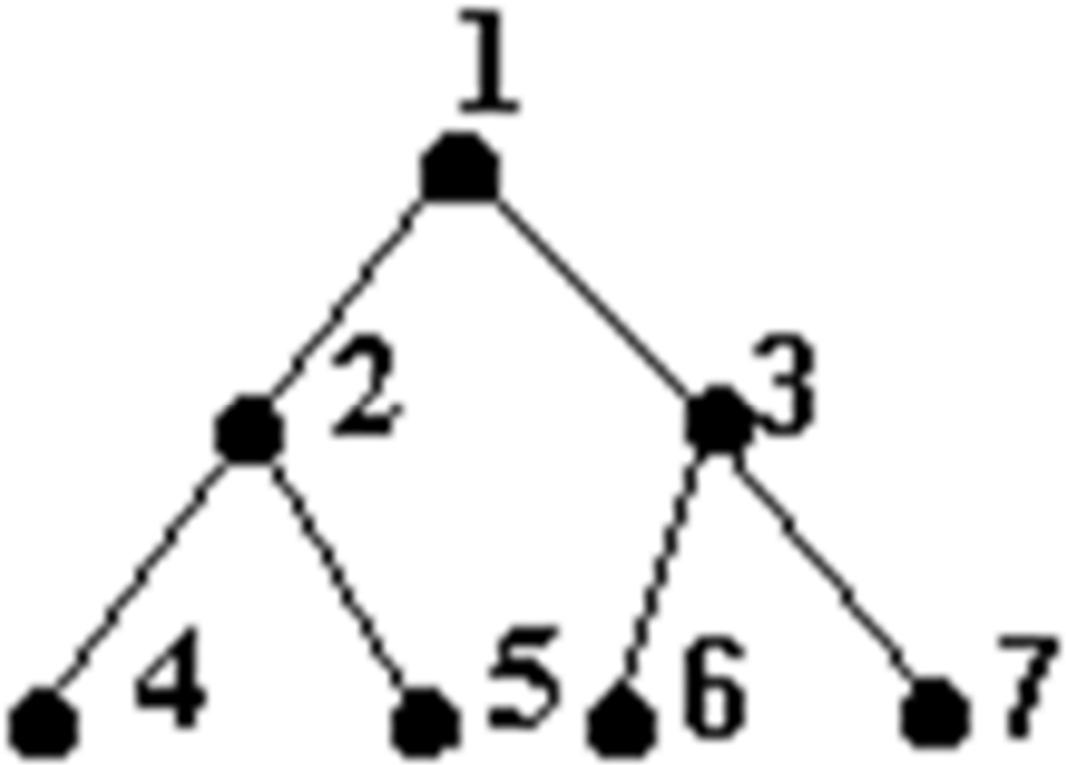
# Arborele binar plin

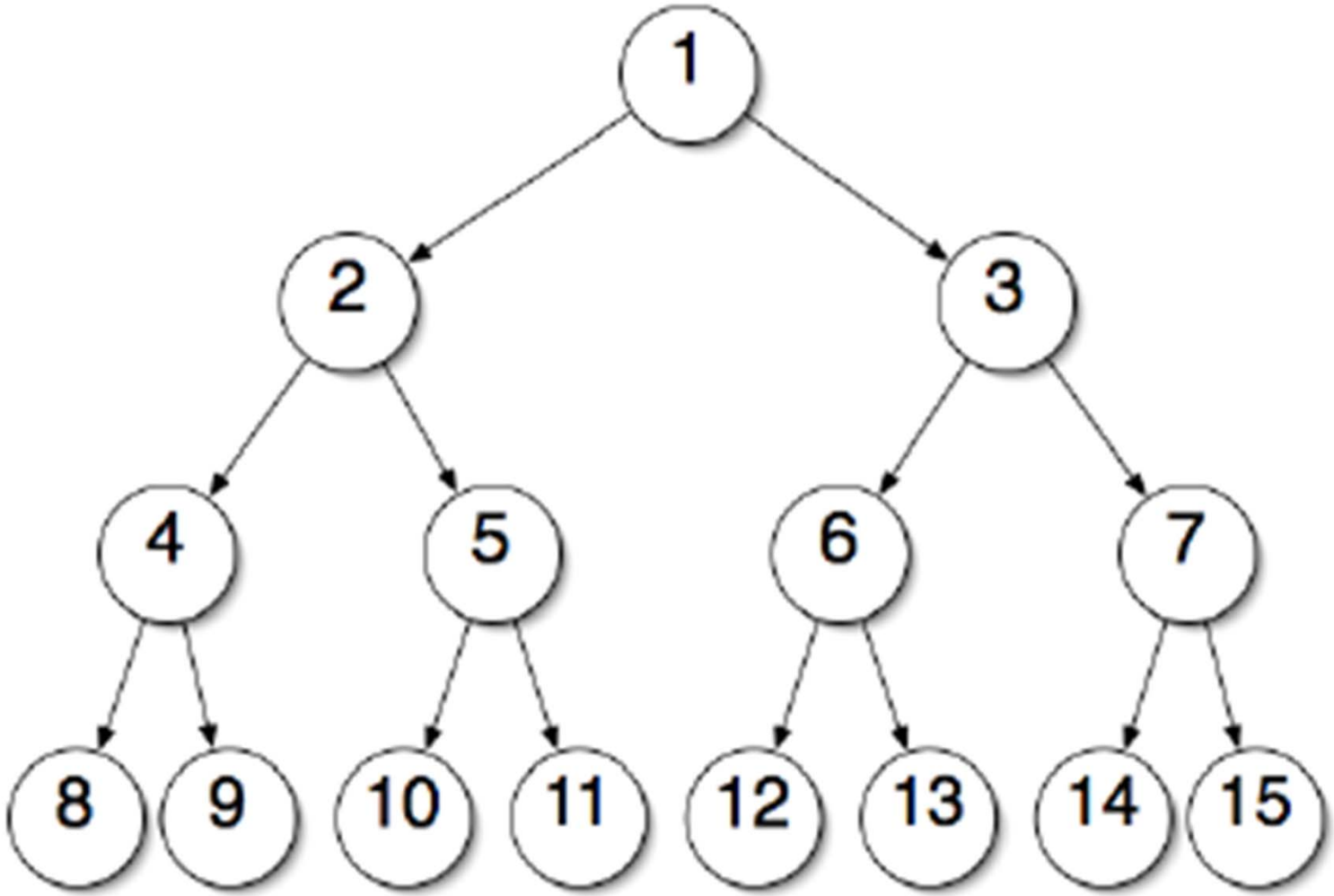
Arborele binar plin este arborele care are numărul maxim de vârfuri ($2^{i+1}-1$) pentru o înălțime **i** dată.

De exemplu, arborele binar plin cu înălţimea 2 se prezintă astfel:

# **Observații**:

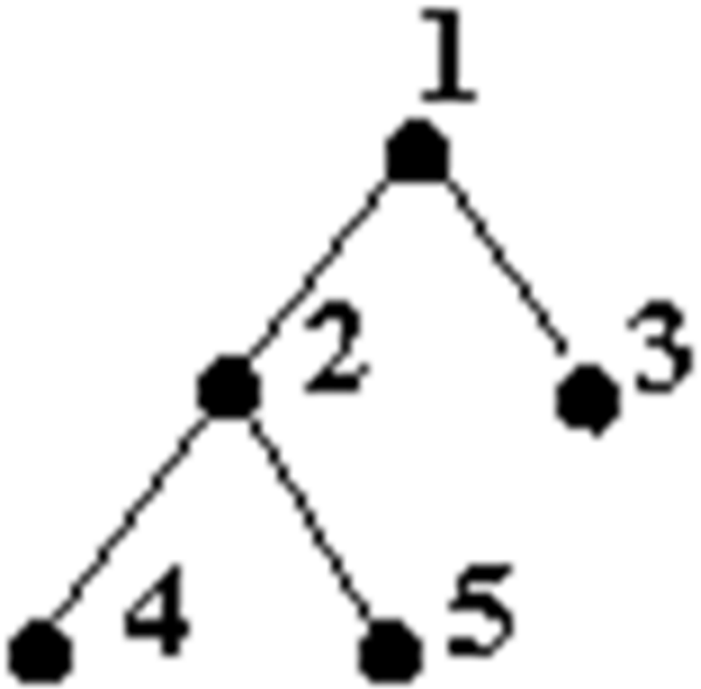
- Vârfurile unui arbore binar plin se numerotează în ordinea adâncimii.
- Pentru aceeași adâncime, numerotarea se face în arbore de la stânga la dreapta.




1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

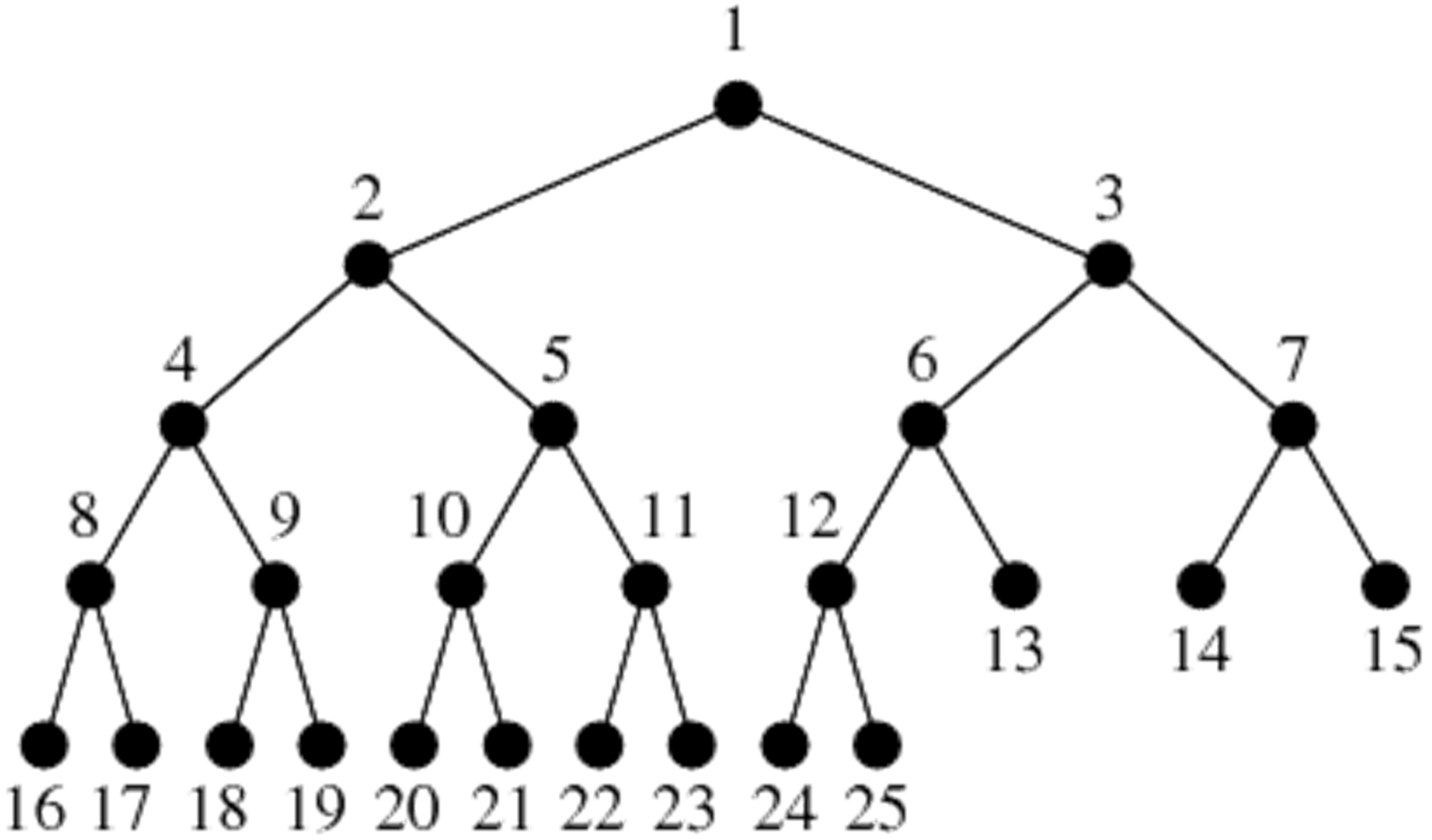
# Arborele binar complet

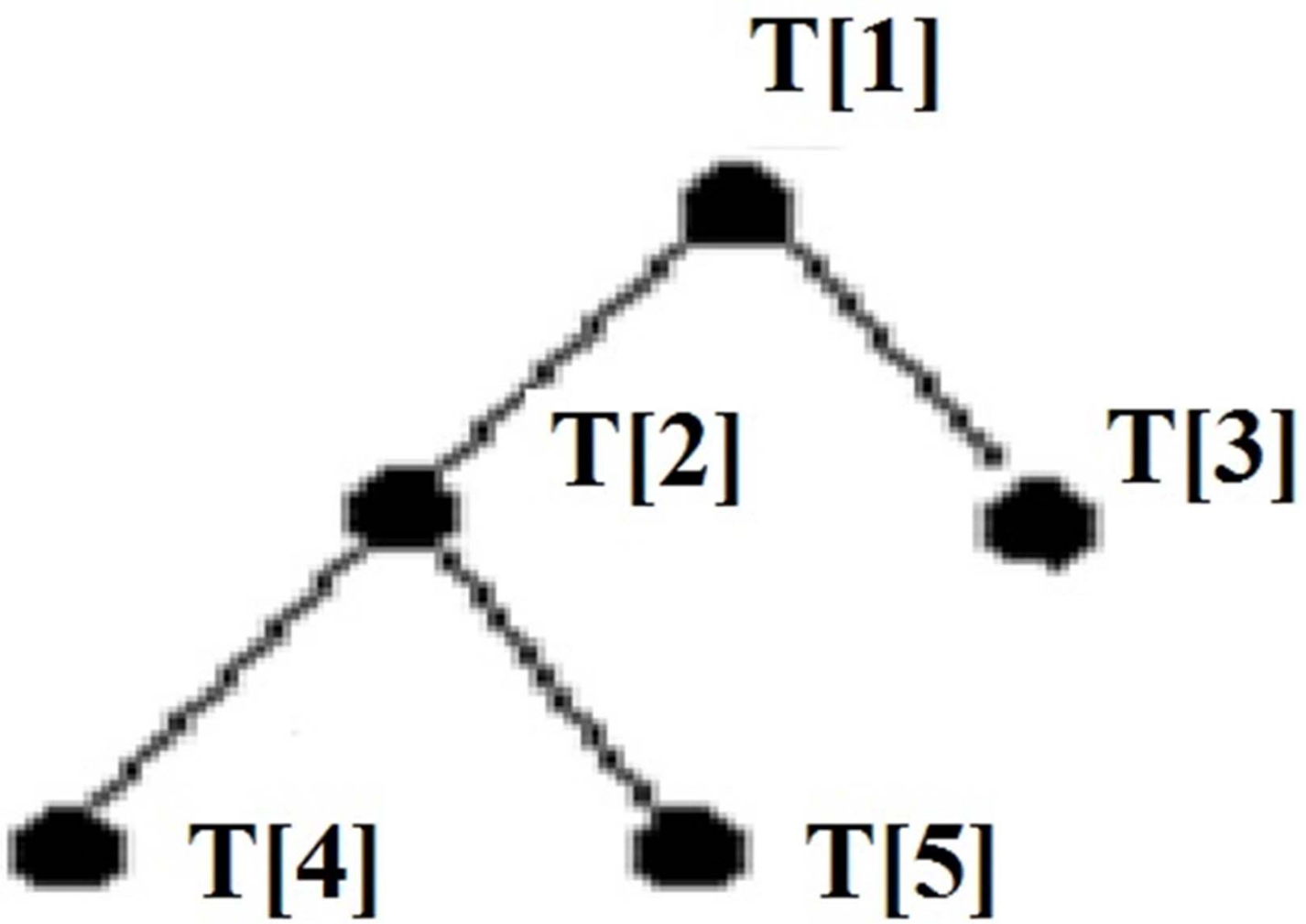
Un arbore binar cu ***n*** vârfuri și de înălțime ***i*** este **complet** dacă se obține din arborele binar plin de înălțime ***i***, prin eliminarea vârfurilor numerotate cu ***n+1***, ***n+2***, ... până la $2^{i+1}-1$.



# **Observație** :

- Un arbore binar complet se poate reprezenta secvențial folosind un tablou **T**, punând vârfurile de adâncime ***k***, de la stânga la dreapta, în pozițiile: $T[2^k]$, $T[2^k+1]$, …, $T[2^{k+1}-1]$, cu excepția nivelului final care poate fi incomplet.


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25



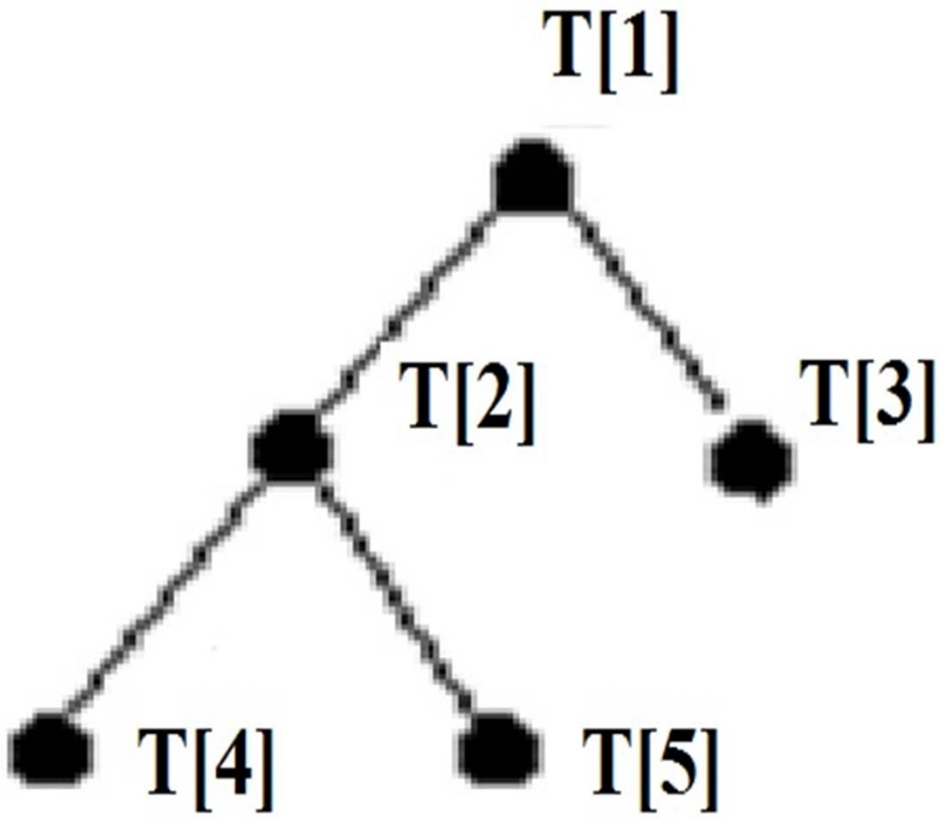

• **Atenție**: acesta este un tablou generic care începe cu T [1] și nu cu T [0].

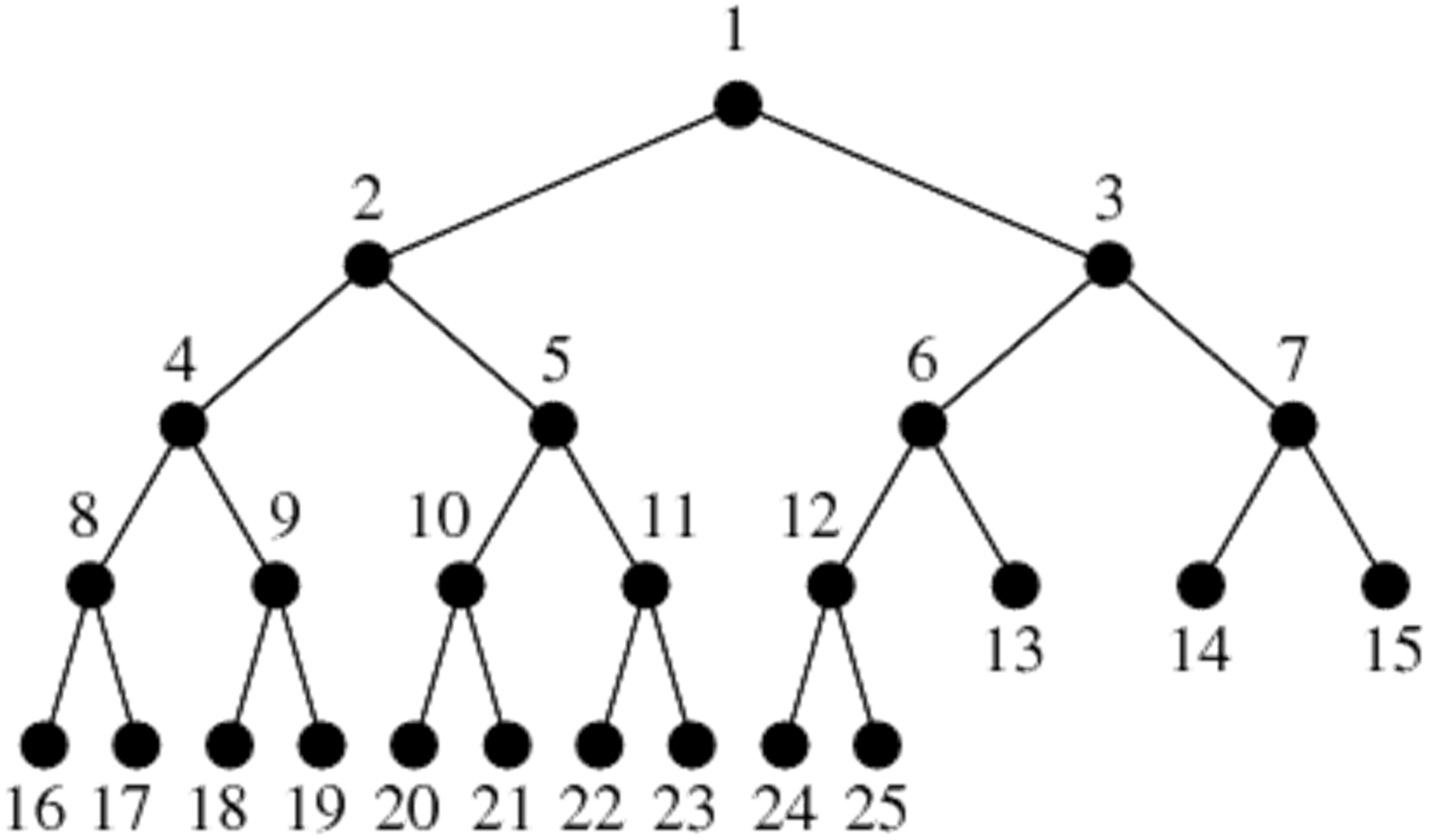
• Se pot opera modificările necesare atunci când se scrie codul în C.

# **Observații**:

- Tatăl unui vârf reprezentat în **T[i]**, i>1, se află în **T[i div 2]** .

- Fiii unui vârf reprezentat în **T[i]**, se află (dacă există) în **T[2·i]** și **T[2·i + 1]** .




1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

**Definim:**

$$\lfloor x \rfloor = \max\{n \mid n \leq x, n \in Z\}$$

$$\lceil x \rceil = \min\{n \mid n \geq x, n \in Z\}$$

# **Proprietăți**:

1. $x-1<\lfloor x\rfloor\le x\le\lceil x\rceil<x+1$ $\forall\ x\in R;$

2. $\left\lfloor\frac{n}{2}\right\rfloor+\left\lceil\frac{n}{2}\right\rceil=n$ $\forall\ n\in Z;$

3. $\left\lceil\left\lceil\frac{n}{a}\right\rceil\Big/b\right\rceil=\left\lceil\frac{n}{ab}\right\rceil$ și $\left\lfloor\left\lfloor\frac{n}{a}\right\rfloor\Big/b\right\rfloor=\left\lfloor\frac{n}{ab}\right\rfloor$ $\forall a, b, n\in Z;$ $a, b\neq 0$

4. $\left\lfloor\frac{n}{m}\right\rfloor=\left\lceil\frac{n-m+1}{m}\right\rceil$ și $\left\lceil\frac{n}{m}\right\rceil=\left\lfloor\frac{n+m-1}{m}\right\rfloor$ $\forall\ n, m\in N^{*};$

# Înălțimea unui arbore binar complet

- Vom arăta că înălțimea unui arbore complet cu ***n*** vârfuri este:

$$i = \lfloor \log_2 n \rfloor$$

# Demonstrație:

- Fie:
  - **n** – numărul de vârfuri;
  - **i** – înălțimea arborelui.
- Știm că:
  - $n_{max} = 2^{i+1}-1$ (când arborele binar este plin)
  - $n_{min} = 2^{i}$ (când pe ultimul nivel avem un singur nod)

- Rezultă că:

$$i = \log_2 n_{\min} \quad => \quad i = \lfloor \log_2 n_{\min} \rfloor \quad \textbf{(1)}$$

Deoarece funcția logaritmică este crescătoare avem:

$$\lfloor \log_2 n_{\max} \rfloor = \lfloor \log_2 (2^{i+1} - 1) \rfloor < \lfloor \log_2 2^{i+1} \rfloor = i + 1$$

Așadar:

$$\lfloor \log_2 n_{\max} \rfloor < i + 1 \qquad \textbf{(2)}$$

Din (1) și (2) rezultă că:

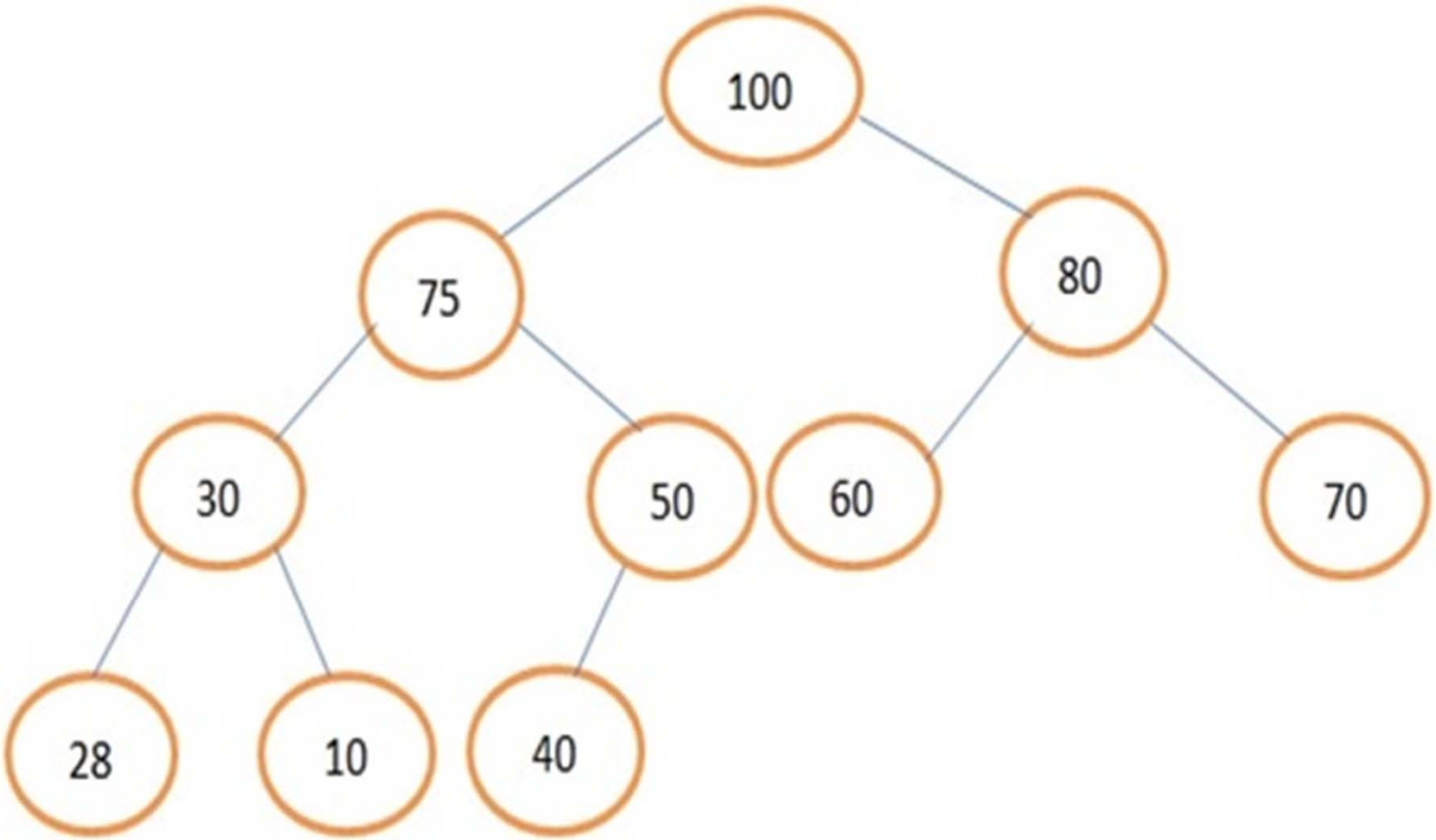
$$i = \lfloor \log_2 n_{\min} \rfloor \leq \lfloor \log_2 n_{\max} \rfloor < i + 1$$

$$\Rightarrow \qquad i = \lfloor \log_2 n \rfloor$$

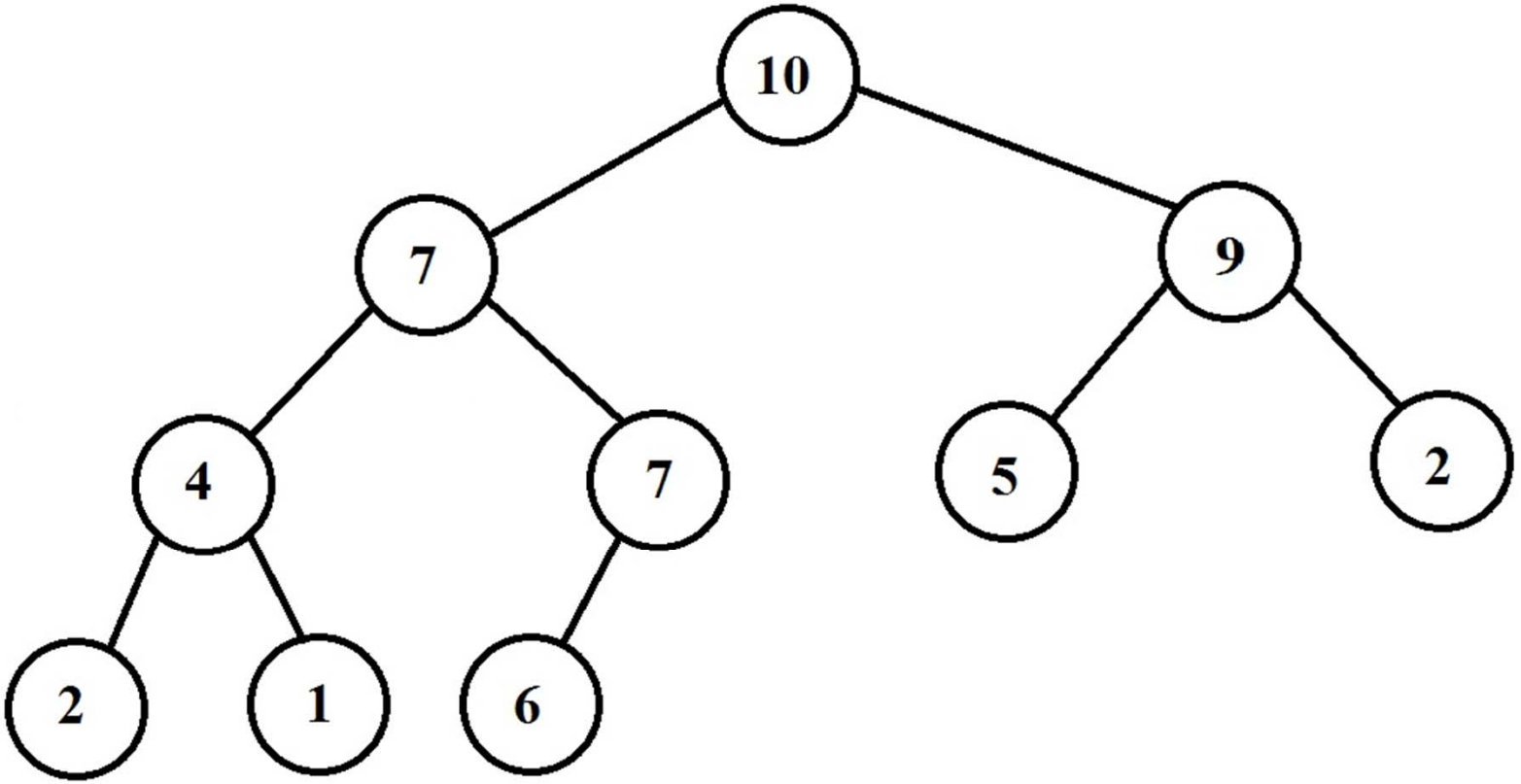
# Arborele HEAP

**Un heap este un arbore "binar" complet care are proprietarea că valoarea fiecărui vârf este mai mare sau egală cu valoarea fiecărui fiu al său.**

Observație: orice heap poate fi reprezentat printr-un vector (tablou unidimensional)


100
75
80
30
50
60
70
28
10
40

# Exemplu:




**Acest heap poate fi reprezentat prin următorul tablou:**

| 10 | 7 | 9 | 4 | 7 | 5 | 2 | 2 | 1 | 6 |
|---|---|---|---|---|---|---|---|---|---|
| T[1] | T[2] | T[3] | T[4] | T[5] | T[6] | T[7] | T[8] | T[9] | T[10] |

# Observații:

- Într-un heap se pot opera modificări la nivel de nod (schimbarea valorii nodului curent).
- Astfel valoarea unui nod poate crește sau poate fi micșorată anulând ordinea inițială de heap.
- Ordinea de heap poate fi restabilită simplu prin intermediul a două operațiuni numite de cernere și de filtrare.

# Filtrarea în heap

Dacă valoarea unui vârf crește astfel încât depășește valoarea tatălui, este suficient să se schimbe între ele aceste două valori și să se continue procesul în mod ascendent, până când proprietatea de heap va fi restabilită.

Se spune că valoarea modificată a fost **filtrată** (***percolated***) către noua sa poziție.

# Cernerea în heap

Dacă valoarea unui vârf scade astfel încât devine mai mică decât valoarea fiului mai mare, este suficient să schimbăm între ele aceste două valori, procesul continuând în mod descendent, până când proprietatea de heap va fi restabilită.

Se spune că valoarea modificată a fost cernută (**sift down**) către noua sa poziție.

# Notă:

- În continuare, marea majoritate a funcțiilor vor fi scrise în varianta **pseudocod**

# Pseudocodul funcției de cernere

```
void cerne (T[1…n], i)
{ int k, x, j;
  k ← i ;
  do      {
            j ← k ;
            if ((2j ≤ n) ∧ (T[2j] > T[k]))  then k ← 2j;
            if ((2j+1 ≤ n) ∧ (T[2j+1] > T[k]))  then k ← 2j+1;
            x ← T[j];
            T[j] ← T[k];
            T[k] ← x
          } while (j ≠ k)
}
```

# Pseudocodul funcției de filtrare

```
void filtreaza (T[1…n], i)
{ int k, j, x;
  k ← i ;
  do      {
           j ← k ;
           if ((j > 1) ∧ (T[ j div 2] < T[k]))  then k ← j div 2;
           x ← T[ j];
           T[ j] ← T[k];
           T[k] ← x
          } while (j ≠ k)
}
```