SDA (PC2)
Curs 11
Arbori
(continuare)

Iulian Năstac

4. Ștergerea unui arbore binar

Recapitulare

- 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?

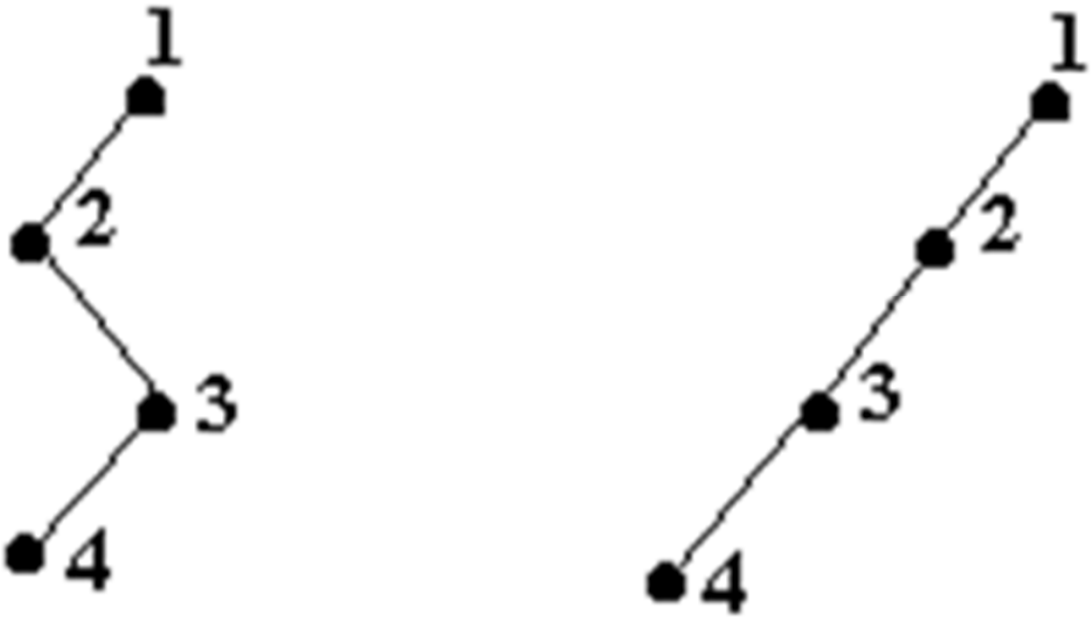
Precizare

- 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.

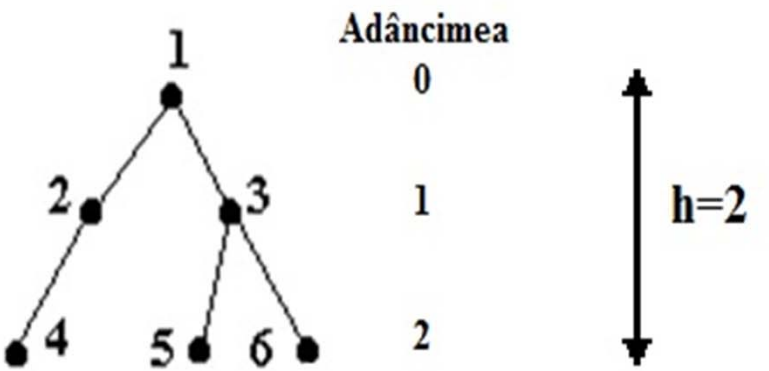
Arbori binari degenerați

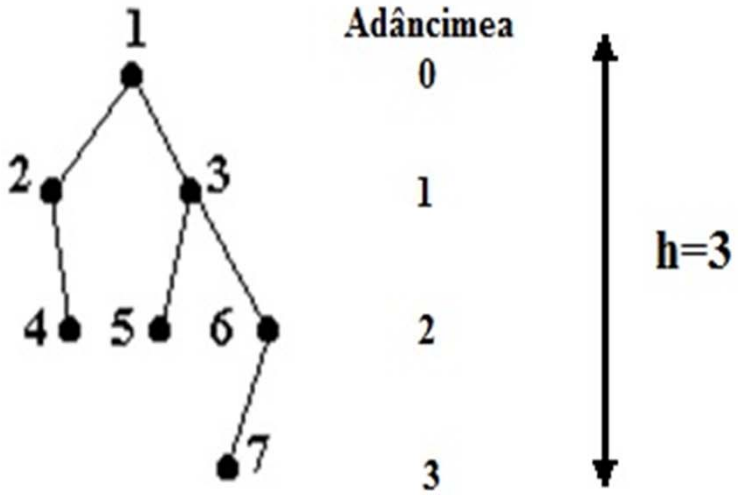
Recapitulare

- 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
Recapitulare

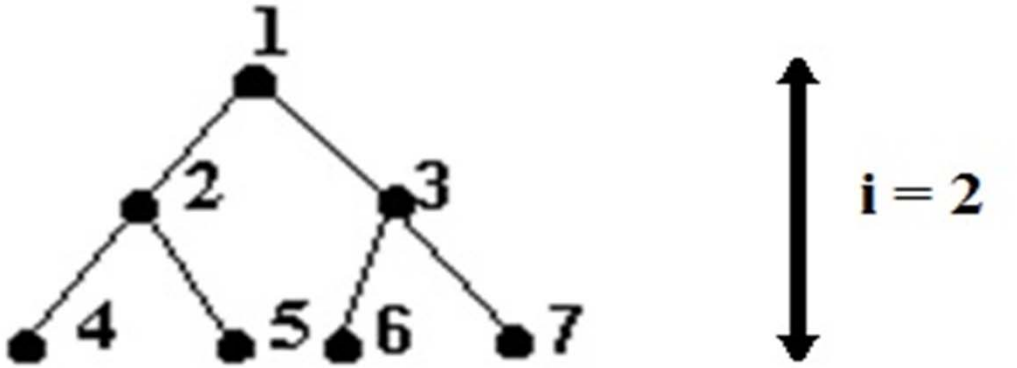
- Adâncimea unui nod este numărul de muchii de la nod la nodul rădăcină al arborelui. Un nod rădăcină va avea adâncimea 0.

- 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.

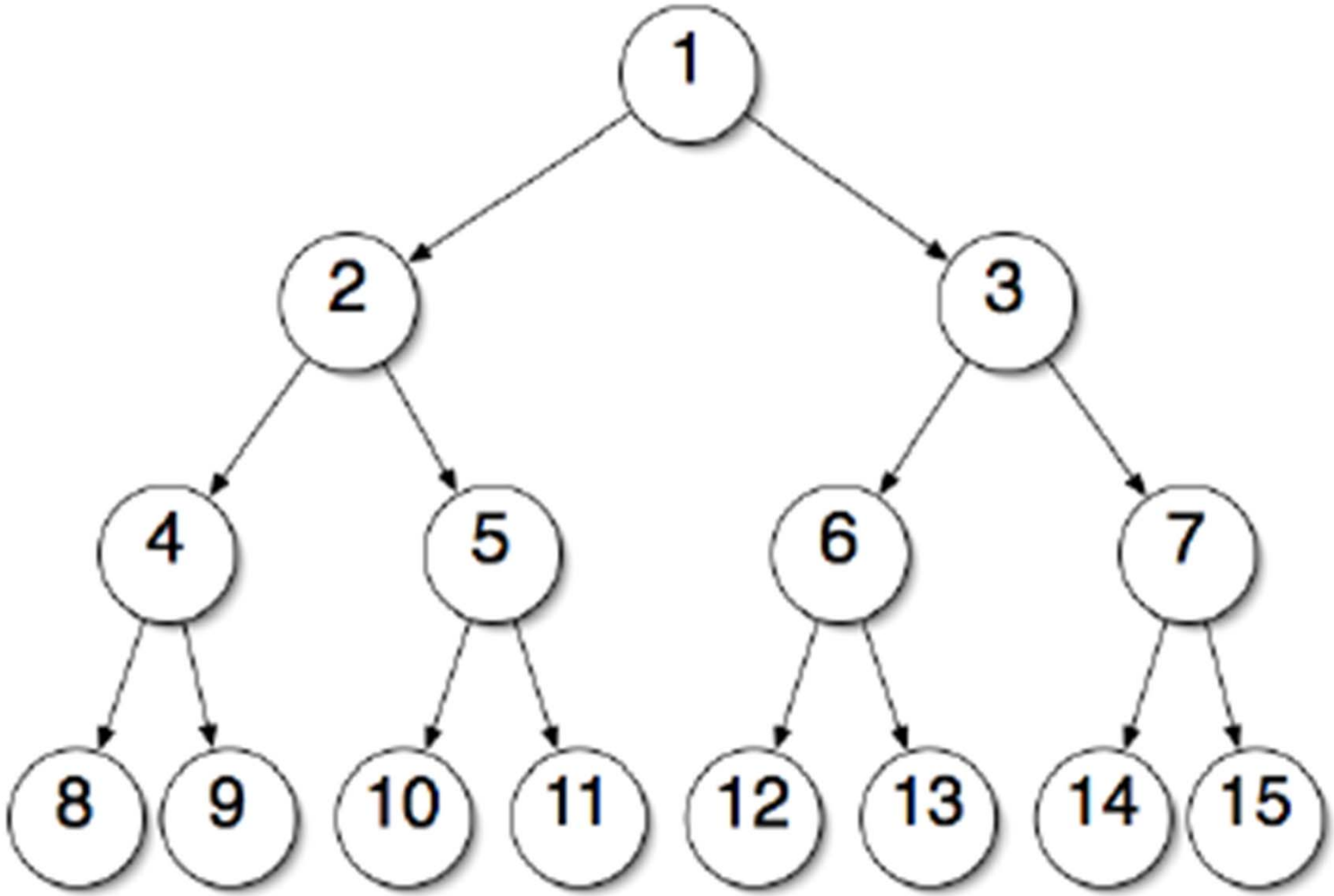
Arbori binari speciali

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

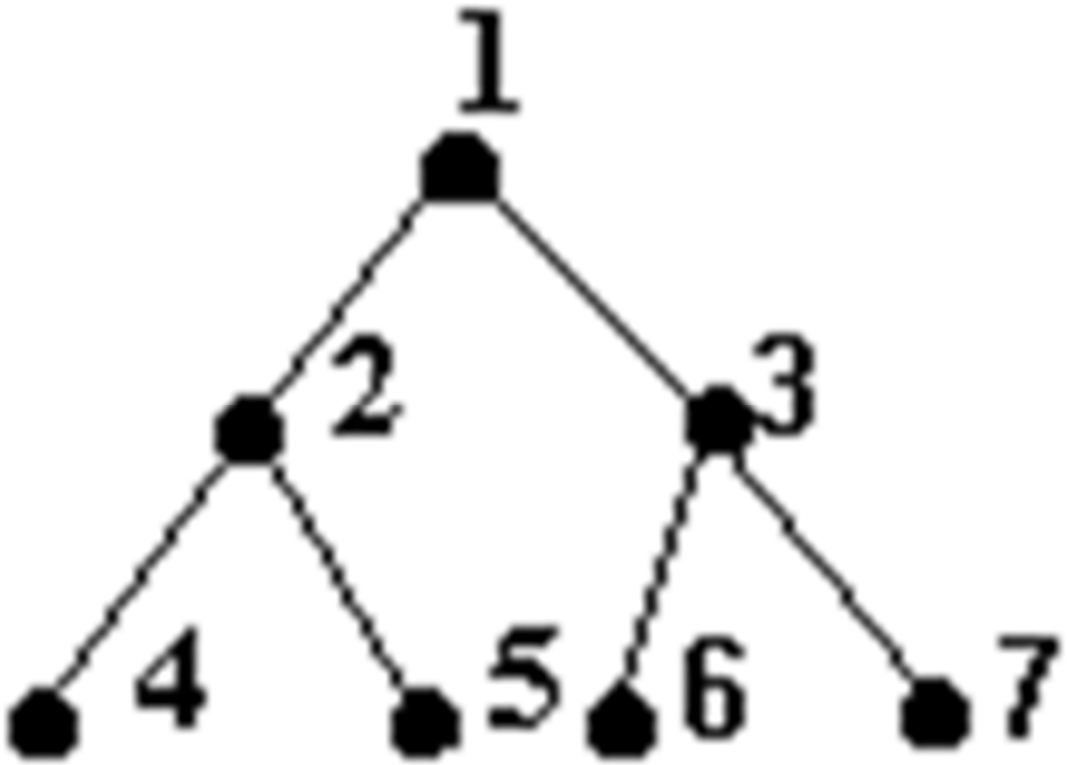
Recapitulare

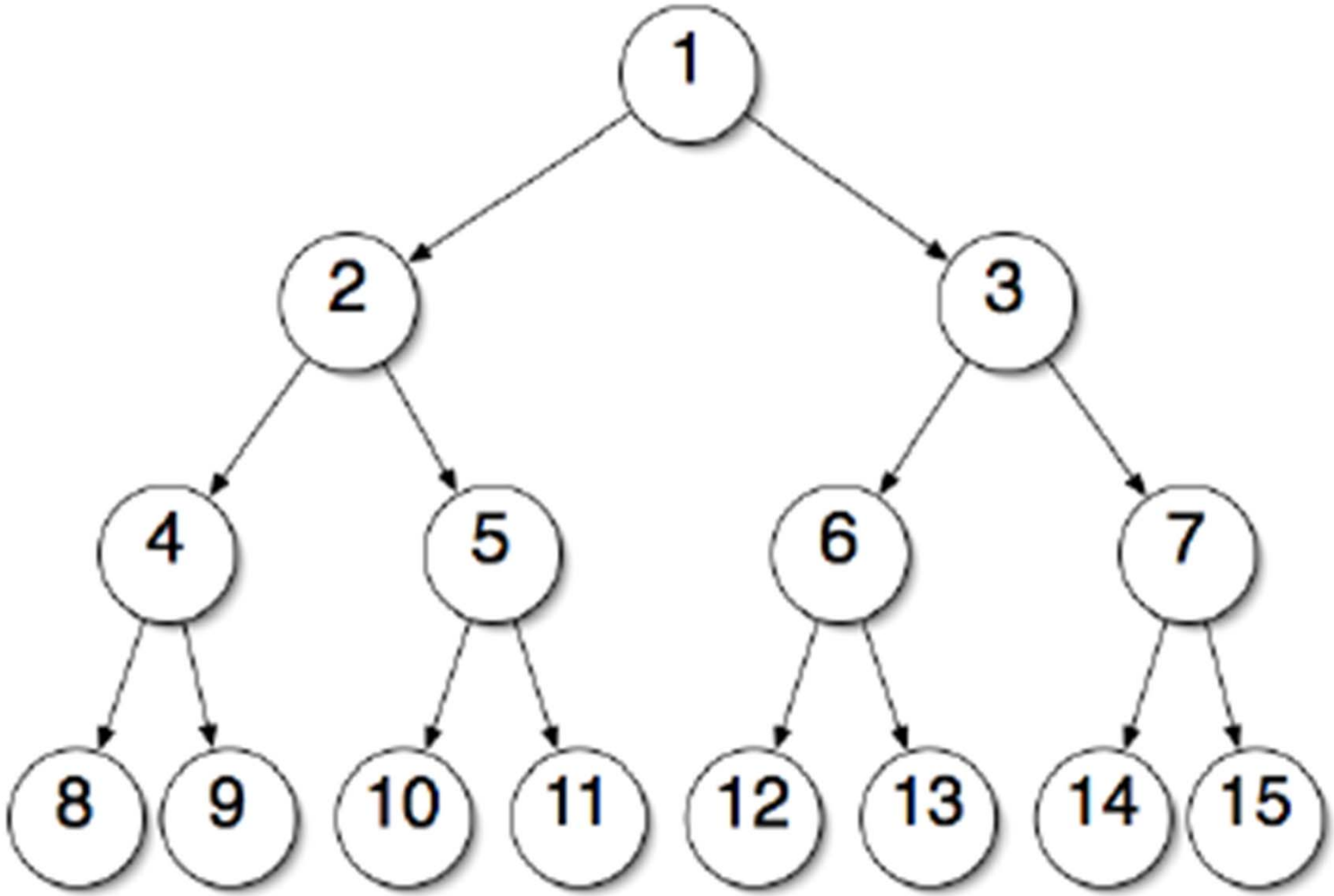
Arborele binar plin este arborele care are numărul maxim de vârfuri ($2^{i+1}-1$) pentru o înălțime **i** dată.

De exemplu, arborele binar plin cu înălţimea 2 se prezint

Observații:

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

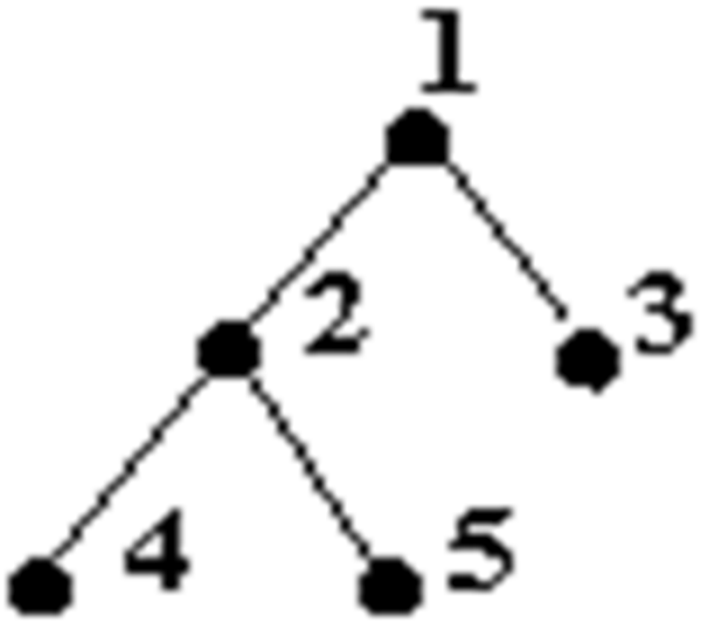



1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Arborele binar complet

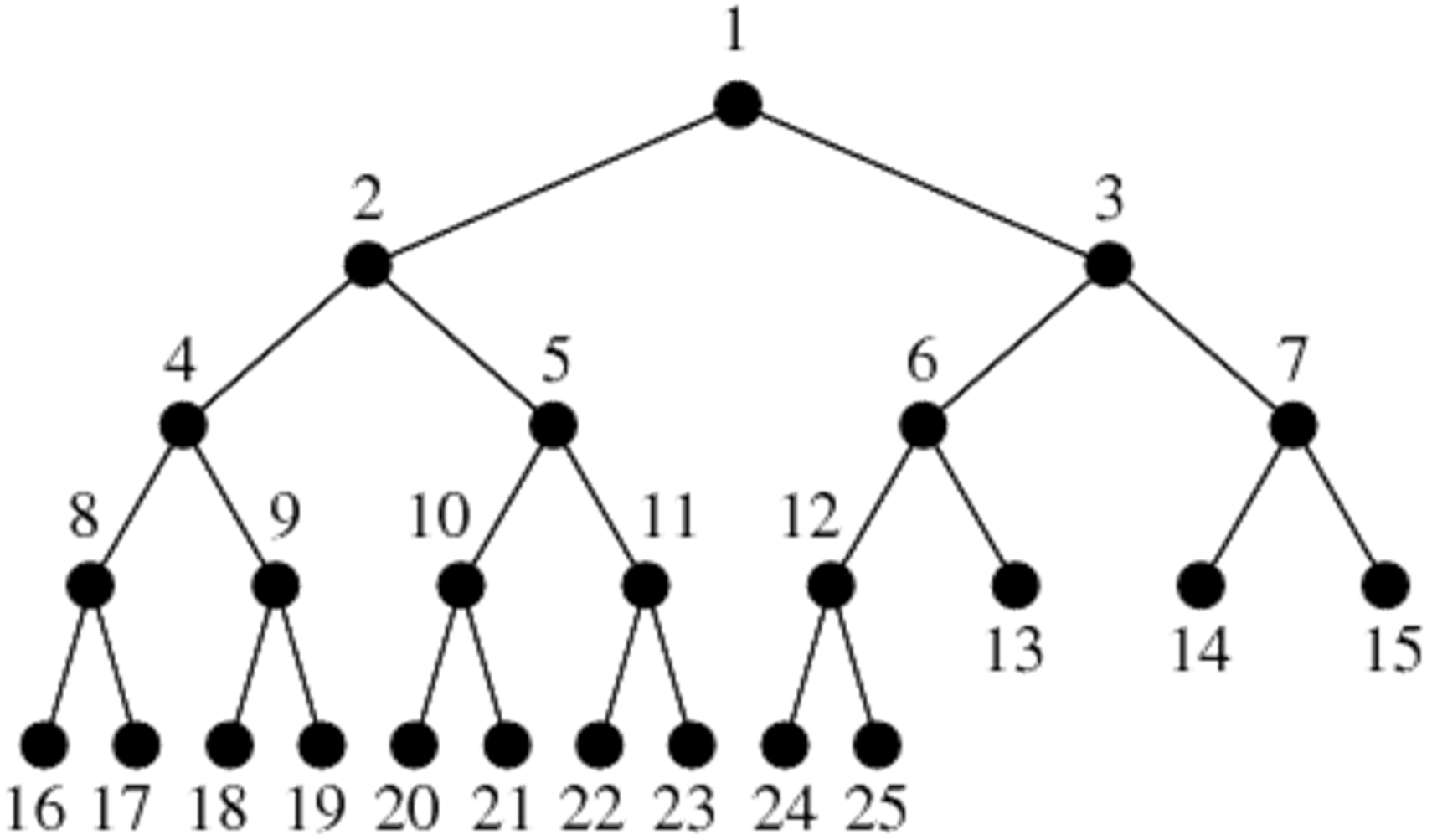
Recapitulare

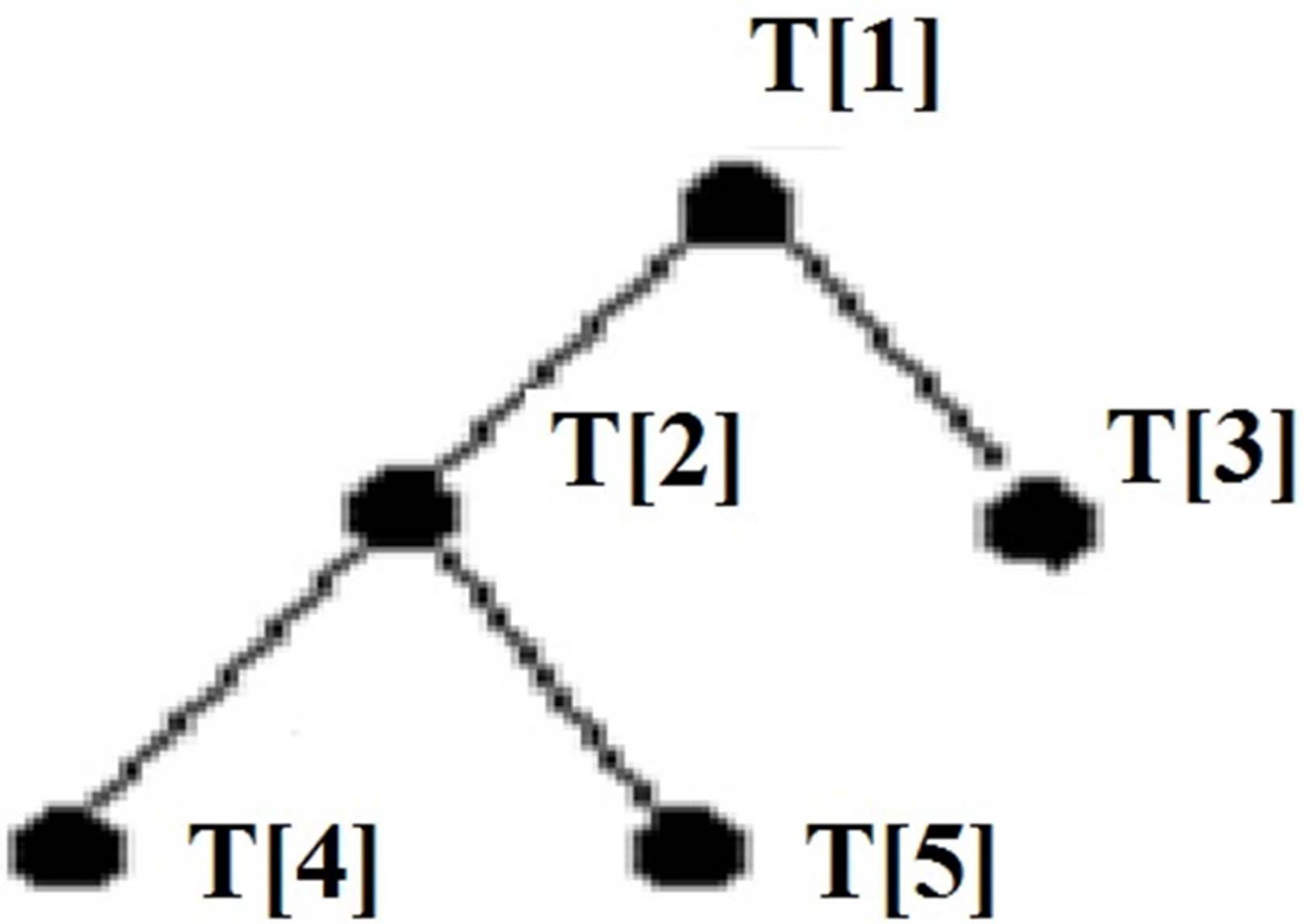
Un arbore binar cu ***n*** vârfuri și de înălțime ***i*** este **complet** dacă se obține din arborele binar plin de înălțime ***i***, prin eliminarea vârfurilor numerotate cu ***n+1***, ***n+2***, ... până la $2^{i+1}-1$.



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



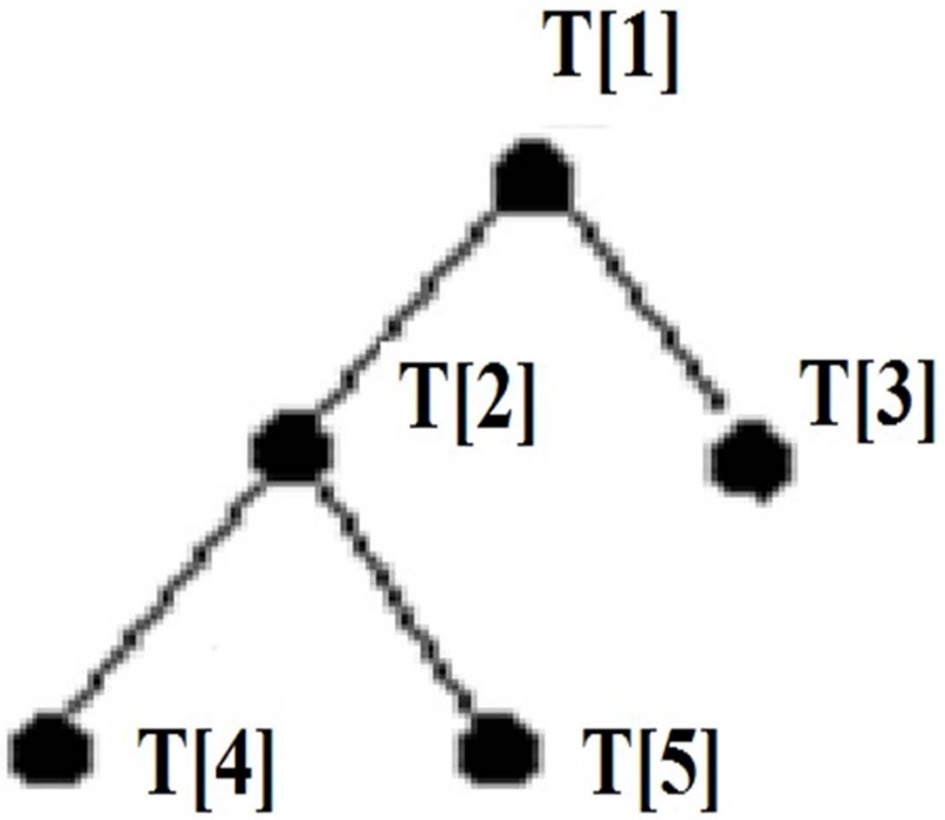
• **Atenție**: acesta este un tablou generic care începe cu T [1] și nu cu T [0].

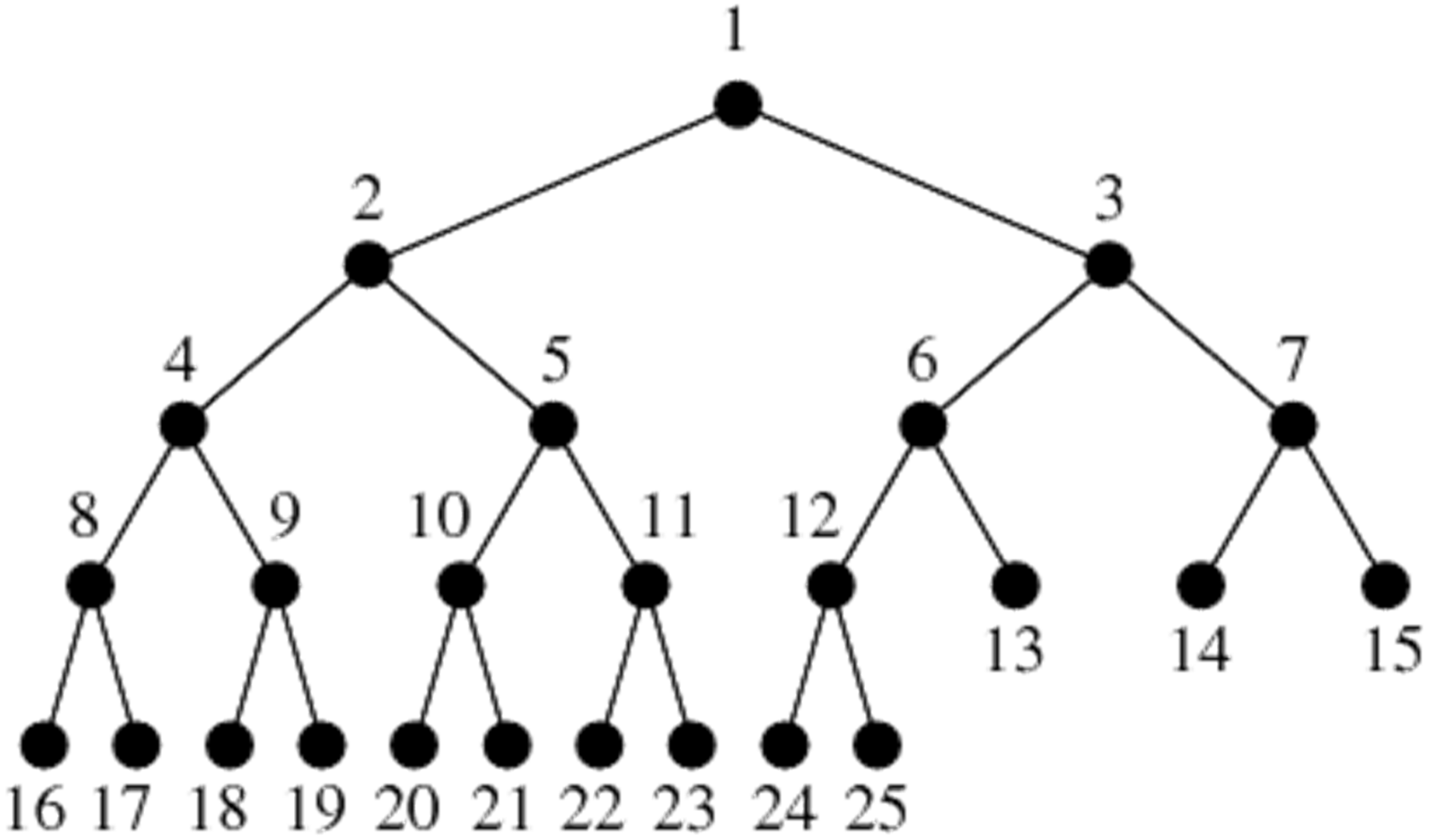
• Se pot opera modificările necesare atunci când se scrie codul în C.

Observaţii:

- 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:

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

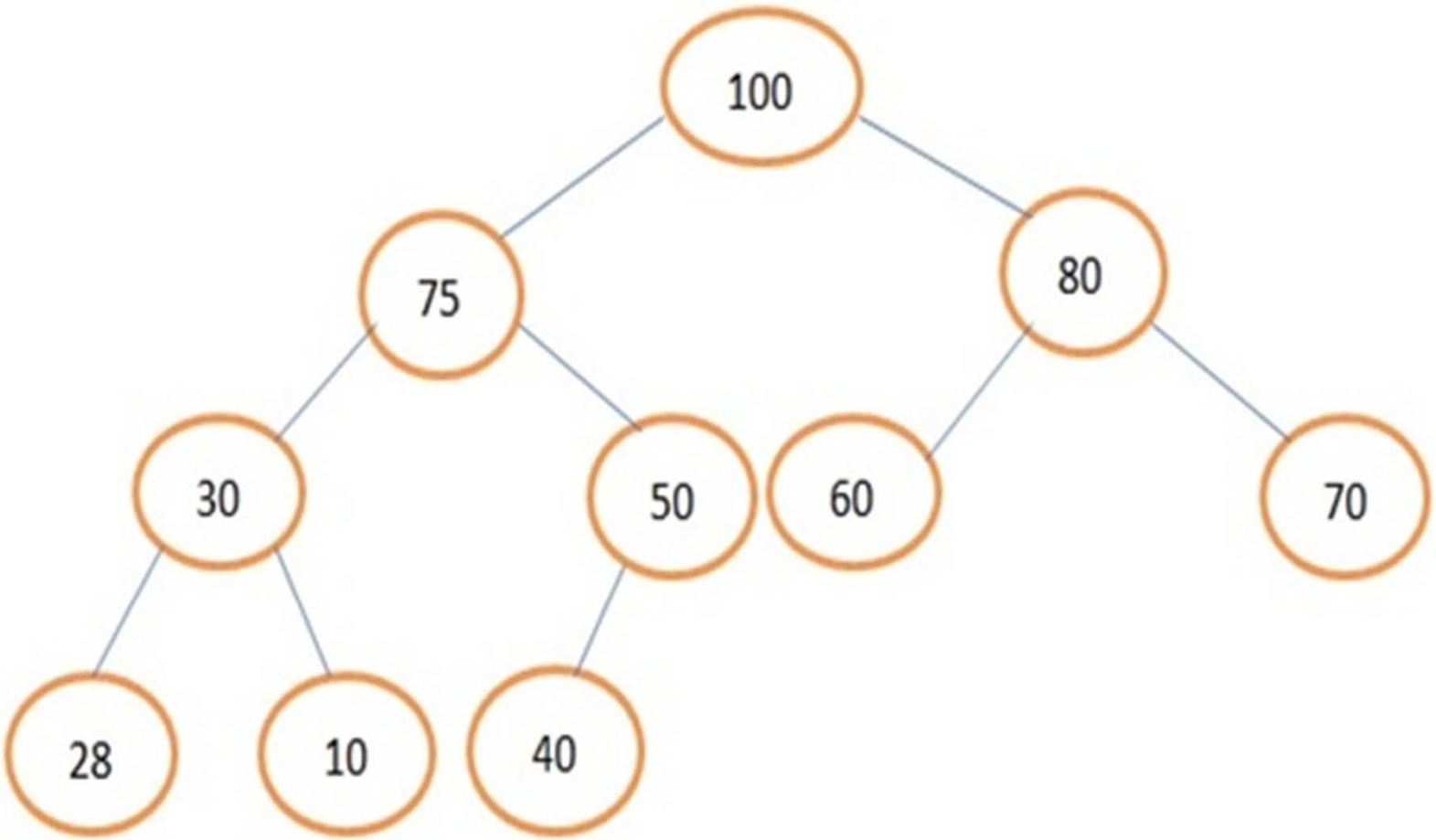
$$\Rightarrow \qquad i = \lfloor \log_2 n \rfloor$$

Arborele HEAP

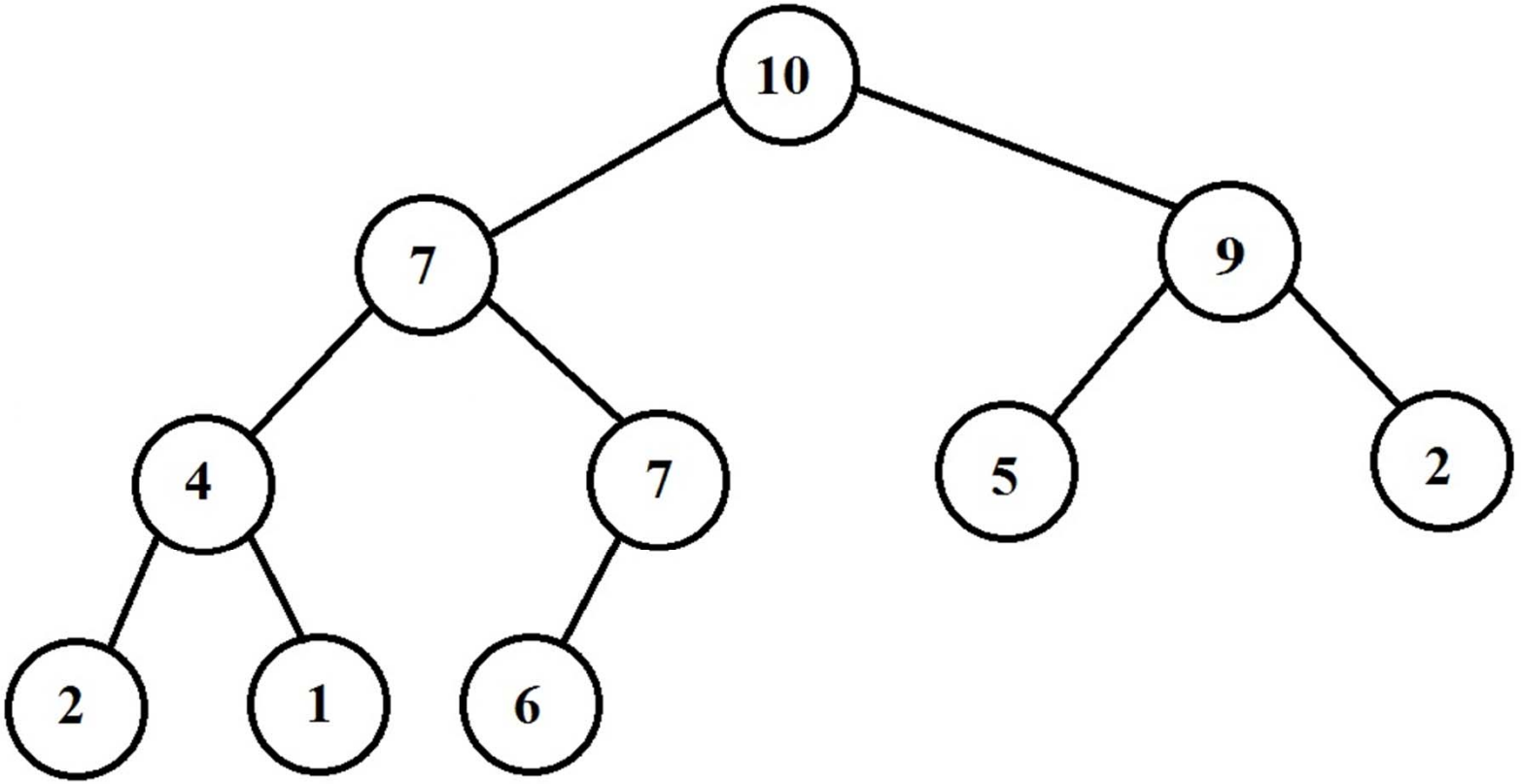
Recapitulare

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)
}
```

Restabilirea proprietății de heap

Considerăm T[1..n] ca fiind un heap. Fie **i**, 1≤i≤n. Lui T[i] i se atribuie valoarea ν și apoi restabilim proprietatea de heap.

```
void restab_heap (T[1..n], i, ν)
{ variab. loc. x;
  x ← T[i];
  T[i] ← ν;
  if  ν < x        then   cerne(T, i);
                   else filtreaza(T,i);
}
```

Heap-ul este un modelul util pentru:

- **Determinarea și extragerea maximului dintr-o mulțime.**
- **Inserarea unui nou vârf.**
- **Modificarea valorii unui vârf** (cu restab_heap).

Operațiile anterioare pot fi folosite pentru a implementa o listă dinamică de priorități:

- Valoarea unui vârf va da prioritatea elementului corespunzător.
- Evenimentul cu probabilitatea cea mai mare se va afla mereu la rădăcina ***heap***-ului.
- Prioritatea unui eveniment poate fi modificată în mod dinamic.

Acestea sunt câteva principii care stau la baza programelor de baze de date.

Exemple de funcții utile:

1) **Funcția pentru găsirea unui maxim**:

```
gaseste_maxim (T[1..n])
{  return T[1];
}
```

2) **Funcția pentru extragerea unui maxim (și ștergerea sa):**

```
extr_max (T[1..n])
{ var loc x;
  x ← T[1];
  T[1] ← T[n];
  cerne (T[1..n-1], 1);
  return x;
}
```

3) **Funcția de inserare a unui element nou în heap**:

```
insert (T[1..n], ν)
{
    T[n+1] ← ν;
    filtreaza (T[1..n+1], n+1);
}
```

Observație: în limbajul C nu este luată în considerație valoarea maximă a numărului de elemente ale unui tablou, folosit ca parametru de funcție, astfel încât, de exemplu, pentru funcția de cernere, filtrare, etc., va trebui să ținem seama de un parametru suplimentar.

Ex.: **cerne (T[ ], *n*, i)**

unde ***n*** indică indicele ultimului element folosit (!de fapt numerotarea ar trebui să înceapă de la 0).

Cum putem forma un heap pornind de la un vector neordonat T[1..n] ?

- O soluție mai puțin eficientă este aceea de a porni de la un heap virtual vid și adăugăm elementele unul câte unul.

```
slow_make_heap(T[1..n])
{
   for (i=2; i ≤ n; i++)
        filtreaza (T[1..i], i);
}
```

Există un algoritm mai eficient care lucrează liniar (din punct de vedere al ordinului/eficienței):

```
make_heap(T[1..n])
{
        for (i = n div 2; i ≥ 1; i - -)
                cerne (T[ ], i);
}
```

Exemplu:

- Pornind de la vectorul următor (care nu este un heap):

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

Printr-o succesiune de câțiva pași se ajunge la:

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]

Există variante

- Def.: Un **min-heap** este un heap inversat (valoarea fiecărui vârf este mai mică sau egală cu valoarea fiecărui fiu al său).

Observații:

- Rădăcina min-heap-ului va conține cel mai mic element al vectorului.
- Se modifică în mod corespunzător și celelalte proceduri de manipulare a heap-ului.

Atenție!

- Heap-ul este o structură de date foarte atractivă, dar are și limitări.

- Există operații care nu pot fi efectuate eficient într-un heap.

Dezavantajele unui heap:

- Necesitatea ca arborele să fie complet.

- Găsirea unui vârf, cu o anumită valoare dată, este ineficientă (pentru că pot exista mai multe vârfuri cu aceeași valoare plasate pe diferite ramuri și nivele din heap).

• O extensie a conceptului de heap este posibilă pentru arbori compleți și ordonați a căror noduri neterminale au mai mult de doi fii.

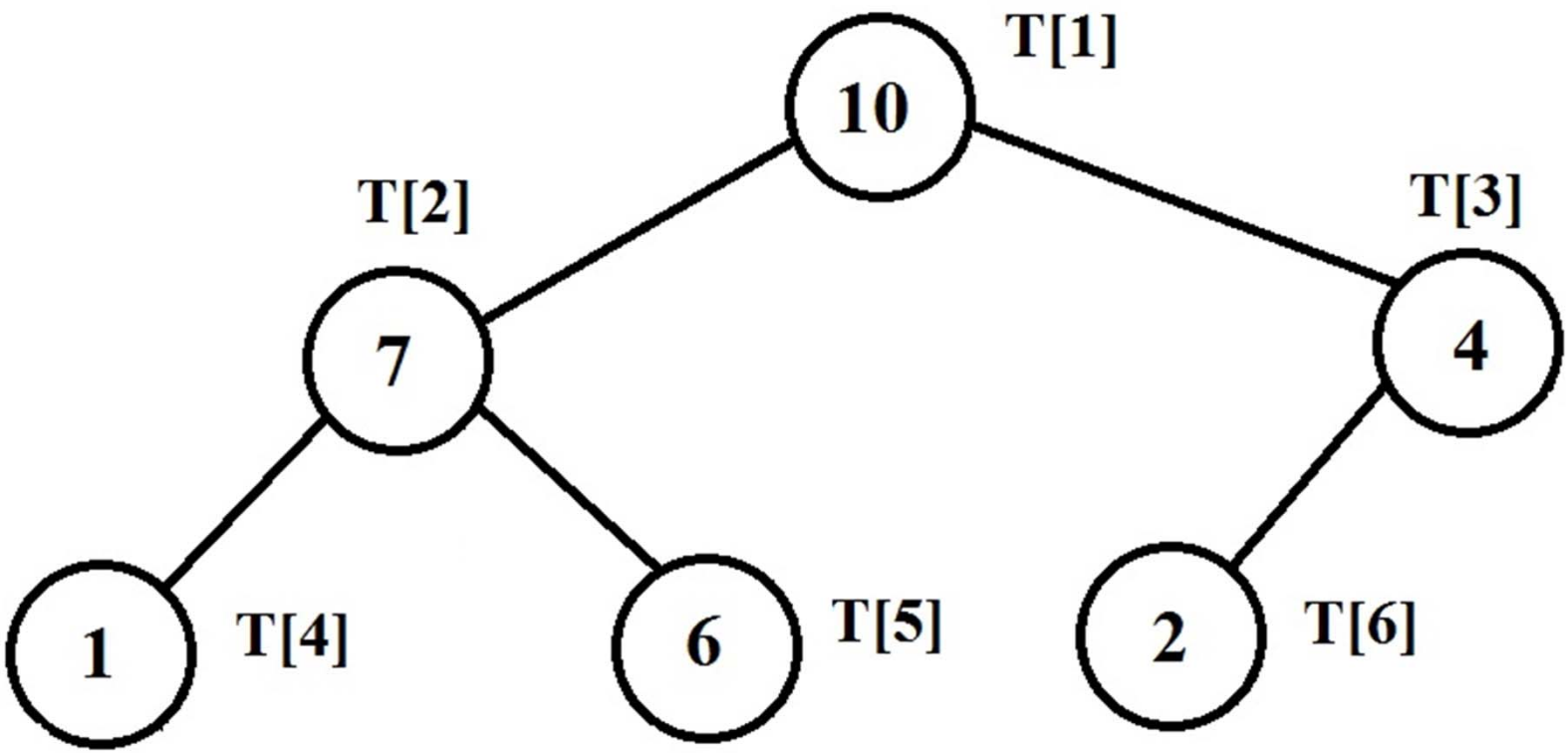
• O astfel de structură accelerează procedura de filtrare.

Principala aplicație:
O nouă tehnică de sortare
(**heapsort**)

- Noțiunea de heap a fost introdusă și folosită în anii 60 de Robert W. Floyd și J. W. J. Williams pentru crearea unui algoritm de sortare numit **heapsort**.

```
heapsort (T[1..n])
{
        make_heap(T);
        for( i = n; i ≥ 2; i - -)
                {
                 T[1] ↔ T[i];
                 cerne (T[1..i-1], 1)
                }
}
```

Exemplu:



Alți algoritmi de sortare

- Inserție
- Selecție
- Merge sort
- Quicksort
- etc.

Metode de sortare

- Prin **metode de sortare** se înţelege o varietate de tehnici în urma cărora sunt **ordonate**, după anumite criterii specificate, diferite *secvenţe de obiecte* (date) care prezintă o trăsătură comună. În acest scop considerăm că datele sunt o *colecţie de elemente* de un anumit tip şi fiecare element conţine o dată sau chiar mai multe, în raport cu care se realizează ordonarea. O astfel de dată se numeşte **cheie**.

Pentru limbajul de programare C, un algoritm de sortare se poate realiza prin una din următoarele metode:

1. Aranjând datele care se sortează în aşa fel încât cheile lor să corespundă ordinii dorite.

2. Ordonând un tablou de pointeri spre datele care trebuiesc sortate în aşa fel încât considerându-i pe aceştia în ordinea crescătoare a indicilor tabloului, datele spre care pointează să formeze o mulţime ordonată în acord cu ordinea dorită.

Observație:

- În continuare ne vom restrânge aria de lucru la sortarea tablourilor unidimensionale (vectori) cu date numerice.