- **Arbori**

Numim *arbore* un graf neorientat conex şi fără cicluri.

Aceasta nu este singurul mod în care putem defini arborii. Câteva definiţii echivalente apar în următoarea teoremă, expusă fără demonstraţie.

Teoremă. Fie `G` un graf cu `n≥1` vârfuri. Următoarele afirmaţii sunt echivalente:

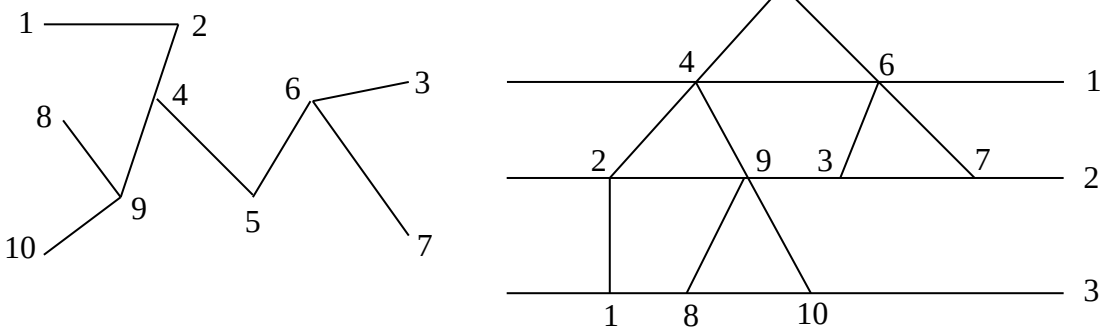
1) `G` este un arbore;
2) `G` are `n-1` muchii şi nu conţine cicluri;
3) `G` are `n-1` muchii şi este conex;
4) oricare două vârfuri din `G` sunt unite printr-un unic drum;
5) `G` nu conţine cicluri şi adăugarea unei noi muchii produce un unic ciclu elementar;
6) `G` este conex, dar devine neconex prin ştergerea oricărei muchii.

În foarte multe probleme referitoare la arbori este pus în evidenţă un vârf al său, numit *rădăcină*. Alegerea unui vârf drept rădăcină are două consecinţe:

- *Arborele poate fi aşezat pe niveluri* astfel:
- rădăcina este aşezată pe nivelul 0;
- pe fiecare nivel `i` sunt plasate vârfurile pentru care lungimea drumurilor care le leagă de rădăcină este `i`;
- se trasează muchiile arborelui.

Această aşezare pe niveluri face mai intuitivă noţiunea de arbore, cu precizarea că în informatică "arborii cresc în jos".

Exemplul 3. Considerăm următorul arbore şi modul în care el este aşezat pe niveluri prin alegerea vârfului 5 drept rădăcină.



- *Arborele poate fi considerat un graf orientat*, stabilind pe fiecare muchie sensul de la nivelul superior către nivelul inferior.

Reprezentarea pe niveluri a arborilor face ca noţiunile de *fii* (*descendenţi*) ai unui vârf, precum şi de *tată* al unui vârf să aibă semnificaţii evidente. Un vârf fără descendenţi se numeşte *frunză*.

- **Arbori binari**

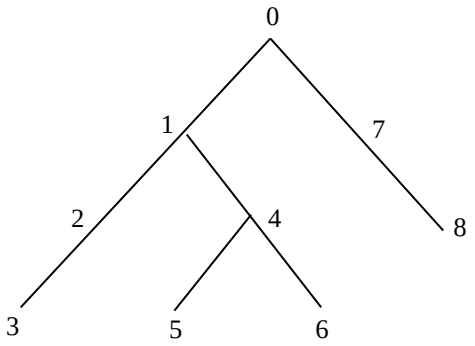
Un *arbore binar* este un arbore în care orice vârf are cel mult doi descendenţi, cu precizarea că se face distincţie între descendentul stâng şi cel drept. Din acestă definiţie rezultă că un arbore binar nu este propriu-zis un caz particular de arbore.

Primele probleme care se pun pentru arborii binari (ca şi pentru arborii oarecare şi pentru grafuri, aşa cum vom vedea ulterior) sunt:
- modul de reprezentare;
- parcurgerea lor.

Forma standard de reprezentare a unui arbore binar constă în:
- a preciza rădăcina `rad` a arborelui;
- a preciza pentru fiecare vârf `i` tripletul `st(i)`, `dr(i)` şi `info(i)`, unde acestea sunt respectiv descendentul stâng, descendentul drept şi informaţia ataşată vârfului.

Trebuie stabilită o convenţie pentru lipsa unuia sau a ambilor descendenţi, ca de exemplu specificarea lor prin simbolul λ.

Exemplul 4. Considerăm de exemplu următorul arbore binar:



Presupunând că informaţia ataşată fiecărui vârf este chiar numărul său de ordine, avem:
- `rad = 0;`
- `st = (1,2,3,λ,5,λ,λ,λ,λ);`
- `dr = (7,4,λ,λ,6,λ,λ,8,λ);`
- `info= (0,1,2,3,4,5,6,7,8).`

Dintre diferitele alte reprezentări posibile, mai menţionăm doar pe cea care se reduce la vectorul său `tata` şi la vectorul `info`. Pentru exemplul de mai sus:
`tata=(λ,0,1,2,1,4,4,0,7).`

Problema *parcurgerii unui arbore binar* constă în identificarea unei modalităţi prin care, plecând din rădăcină şi mergând pe muchii, să ajungem în toate vârfurile; în plus, atingerea fiecărui vârf este pusă în evidenţă *o singură dată*: spunem că *vizităm* vârful respectiv. Acţiunea întreprinsă la vizitarea unui vârf depinde de problema concretă şi poate fi de exemplu tipărirea informaţiei ataşate vârfului.

Distingem trei modalităţi standard de parcurgere a unui arbore binar:

- **Parcurgerea în preordine**

Se parcurg recursiv în ordine: rădăcina, subarborele stâng, subarborele drept.
Concret, se execută apelul `preord(rad)` pentru procedura:

```
procedure preord(x)
   if x=λ
   then
   else vizit(x); preord(st(x)); preord(dr(x))
end
```

Ilustrăm acest mod de parcurgere pentru exemplul de mai sus, figurând îngroşat rădăcinile subarborilor ce trebuie dezvoltaţi:
0
0, **1**, **7**
0, 1, **2**, **4**, 7, 8
0,1, 2, 3, 4, 5, 6, 7, 8

- **Parcurgerea în inordine**

Se parcurg recursiv în ordine: subarborele stâng, rădăcina, subarborele drept.
Ilustrăm acest mod de parcurgere pentru *Exemplul 4*:
0
1, 0, **7**
2, 1, **4**, 0, 7, 8
3, 2, 1, 5, 4, 6, 0, 7, 8
Concret, se execută apelul `inord(rad)` pentru procedura:

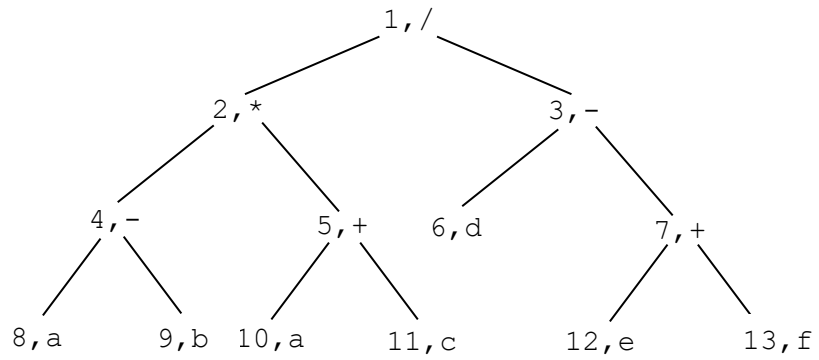
```
procedure inord(x)
   if x=λ
   then
   else inord(st(x)); vizit(x); inord(dr(x))
end
```

- **Parcurgerea în postordine**

Se parcurg recursiv în ordine; subarborele stâng, subarborele drept, rădăcina.
Ilustrăm parcurgerea în postordine pentru *Exemplul 4*:
0
1, **7**, 0
2, **4**, 1, 8, 7, 0
3, 2, 5, 6, 4, 1, 8, 7, 0
Concret, se execută apelul `postord(rad)` pentru procedura:

```
procedure postord(x)
   if x=λ
   then
   else postord(st(x)); postord(dr(x)); vizit(x)
end
```

Exemplu. Expresiei aritmetice `((a-b)*(a+c))/(d-(e+f))` îi corespunde următorul arbore, în care pentru fiecare vârf apare atât numărul său de ordine, cât şi eticheta sa.



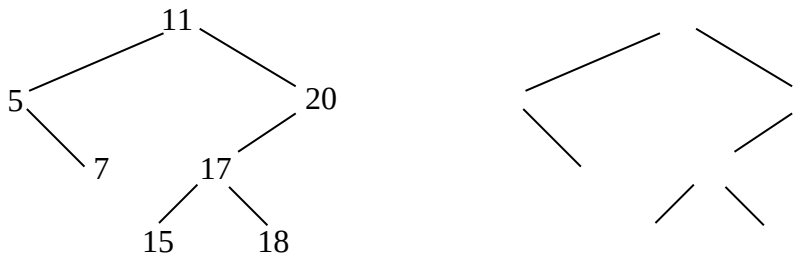
Preordine: `1,2,4,9,5,10,11,3,6,7,12,13`
Forma poloneză directă: `/*-ab+ac-d+ef`

Postordine: `8,9,4,10,11,5,2,6,12,13,7,3,1`
Forma poloneză inversă: `ab-ac+*def+-/`

Cele două forme sunt echivalente cu cea parantezată (deci şi echivalente între ele).

- **Arbori de sortare**

Un *arbore de sortare* este un arbore binar în care pentru orice vârf informaţia ataşată vârfului este mai mare decât informaţiile vârfurilor din subarborele stâng şi mai mică decât informaţiile vârfurilor din subarborele drept.



obţinut de exemplu dacă la intrare avem 11,20,5,17,18,7,15.

Observaţie. Prin parcurgerea în inordine a unui arbore de sortare, informaţiile ataşate vârfurilor vor apărea în ordine crescătoare:

11
5,11,**20**
5,7,11,**17**,20
5,7,11,15,17,18,20

Fie $a=(a_0,\ldots,a_{n-1})$ un vector ce trebuie ordonat crescător. Conform observaţiei de mai sus, este suficient să creăm un arbore de sortare în care informaţiile vârfului să fie tocmai elementele vectorului. Pentru aceasta este suficient să precizăm modul în care prin adăugarea unei noi valori, obţinem tot un arbore de sortare.

Pentru exemplul considerat:
- adăugarea valorii 6 trebuie să conducă la crearea unui nou vârf, cu informaţia 6 şi care este descendent stâng al vârfului cu informaţia 7;
- adăugarea valorii 16 trebuie să conducă la crearea unui nou vârf, cu informaţia 16 şi care este descendent drept al vârfului cu informaţia 15.

Presupunem că un vârf din arborele de sortare este un obiect de tipul `varf`, ce conţine câmpurile:
- informaţia `info` ataşată vârfului;
- descendentul stâng `st` şi descendentul drept `dr` (lipsa acestora este marcată, ca de obicei, prin λ).

Crearea unui nou vârf se face prin funcţia `varf_nou` ce întoarce un nou vârf:

```
function varf_nou(info)
    creez un nou obiect x în care informaţia este info, iar descendentul stâng şi cel
      drept sunt λ;
   return x
end
```

Inserarea unei noi valori `val` (în arborele de rădăcină `rad`) se face prin apelul `adaug(rad,val)`, unde procedura `adaug` are forma:

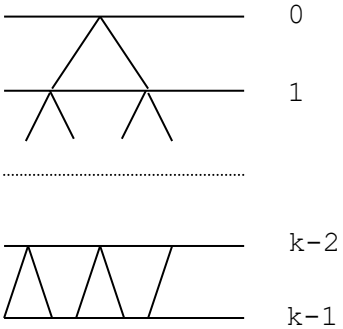
```
procedure adaug(x,val)    { se inserează val în subarborele de rădăcină x}
  if val < x.info
  then if x.st ≠ λ
       then adaug(x.st,val)
       else x.st ← varf_nou(val)
  else if x.dr ≠ λ
       then adaug(x.dr,val)
       else x.dr ← varf_nou(val)
end
```

- **Sortarea cu ansamble**

Metoda sortării de ansamble apelează la o *reprezentare implicită a unui vector* $a=(a_1,...a_n)$ *ca arbore binar* (deci a_0 nu este folosit!). Acest arbore este construit succesiv astfel:
- rădăcina este 1;
- pentru orice vârf `i`, descendenţii săi stâng şi drept sunt `2i` şi `2i+1` (cu condiţia ca fiecare dintre aceste valori să nu depăşească pe `n`). Rezultă că tatăl oricărui vârf `i` este $tata(i)=\lfloor i/2 \rfloor$.

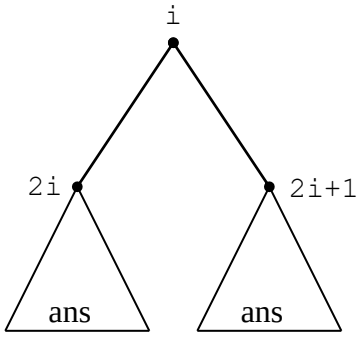
Evident, fiecărui vârf `i` îi vom ataşa eticheta a_i.

Pentru $2^{k-1} \le n < 2^k$ arborele va avea `k` niveluri, dintre care numai ultimul poate fi incomplet (pe fiecare nivel $i<k-1$ se află exact 2^i vârfuri).



Vectorul a se numeşte *ansamblu* dacă pentru orice i avem $a_i \geq a_{2i}$ şi $a_i \geq a_{2i+1}$ (dacă fiii există).

Să presupunem că subarborii de rădăcini 2i şi 2i+1 sunt ansamble. Ne propunem să transformăm arborele de rădăcină i într-un ansamblu. Ideea este de a retrograda valoarea a_i până ajunge într-un vârf ai cărui descendenţi au valorile mai mici decât a_i. Acest lucru este realizat de procedura combin.




```
procedure combin(i,n)
   j ← 2i; b ← a_i
   while j≤n
      if j<n & a_j<a_{j+1} then j←j+1
      if b>a_j
      then a_⌊j/2⌋ ← b; exit
      else a_⌊j/2⌋ ← a_j; j ←2j
   a_⌊j/2⌋ ← b
end
```

Timpul de executare pentru procedura combin este O(k)=O(log n).

Sortarea vectorului a se va face prin apelul succesiv al procedurilor creare şi sortare prezentate în continuare.

Procedura creare transformă vectorul într-un ansamblu; în particular în a_1 se obţine cel mai mare element al vectorului.

Procedura sortare lucrează astfel:
- pune pe a_1 pe poziţia n şi reface ansamblul format din primele n-1 elemente;
- pune pe a_1 pe poziţia n-1 şi reface ansamblul format din primele n-2 elemente;
- etc.

```
procedure creare
   for i=⌊n/2⌋,1,-1
      combin(i,n)
end
```

```
procedure sortare
   for i=n,2,-1
      a_1 ↔ a_i; combin(1,i-1)
end
```

Timpul total de lucru este de ordinul O(n·log n).

Aşa cum am menţionat chiar de la început, structura de arbore este implicită şi este menită doar să clarifice modul de lucru al algoritmului: calculele se referă doar la componentele vectorului.