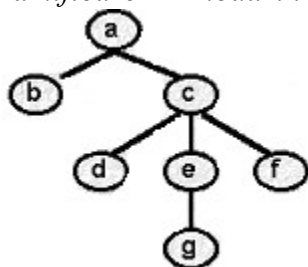


STRUCTURI ARBORESCENTE

1. Arbori și operații cu arbori

Cele mai importante **structuri neliniare** care apar în algoritmi sunt structurile arborescente. Prin ele se pot descrie diferite ramificații. Un **arbore** este o mulțime finită A de unul sau mai multe noduri din care se distinge un nod R numit **rădăcină** toate celelalte noduri din A fiind partiționate în $n \geq 0$ mulțimi disjuncte A_1, A_2, \dots, A_n , fiecare dintre aceste mulțimi fiind la rândul ei un arbore numit **subarbore** al rădăcinii R . În teoria grafurilor, un arbore este un graf neorientat, conex și fără cicluri.

Numărul de subarbori ai unui nod se numește *gradul* acelui nod. Nodurile cu gradul zero se numesc *noduri terminale*, *frunze* sau *noduri externe*. Celelalte noduri se numesc *noduri de ramificare* sau *noduri interne*. O mulțime de arbori se numește *pădure*.



De exemplu, pentru arborele din stânga:

$$\text{grad}(a) = 2$$

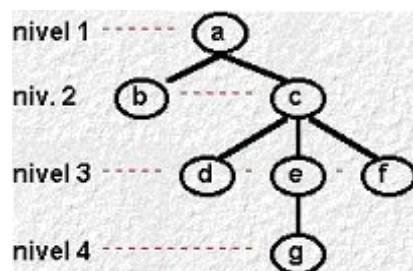
$$\text{grad}(c) = 3$$

$$\text{grad}(e) = 1$$

$$\text{grad}(b) = \text{grad}(d) = \text{grad}(g) = \text{grad}(f) = 0.$$

Astfel: a, c și e sunt noduri interne, iar b, d, g și f sunt frunze.

Pentru un arbore se definește *nivelul unui nod* în felul următor: rădăcina arborelui are nivelul 1 și rădăcina unui subarbore al unui nod are nivelul cu 1 mai mare decât nivelul nodului respectiv.



Se definește *înălțimea (h) a unui nod* ca fiind numărul de muchii din cel mai scurt drum de la rădăcină la nodul respectiv (înălțimea rădăcinii este 0). *Înălțimea unui arbore* se definește ca fiind valoarea maximă dintre înălțimile nodurilor arborelui. De exemplu, pentru arborele de mai sus: $h(a) = 0$, $h(b) = h(c) = 1$, $h(d) = h(e) = h(f) = 2$, $h(g) = 3$, iar înălțimea arborelui este $\max\{0, 1, 2, 3\} = 3$.

Dacă **ordinea relativă a subarborilor** fiecărui nod în parte **este semnificativă** spunem că arborele este un **arbore ordonat**, altfel se numește **neordonat**. Pentru precizarea pozițiilor unor noduri ale arborilor în raport cu alte noduri se folosesc noțiuni utilizate în arborii genealogici. Rădăcinile subarborilor unui nod se numesc **fii** acestuia, nodul fiind **tatăl** acestor rădăcini. **Doi fii ai unui nod sunt frați** între ei. Rădăcina arborelui nu are tată.

O clasă particulară de arbori o formează arborii binari. Se numește **arbore binar** un arbore în care **fiecare nod are cel mult doi subarbori, unul stâng și celălalt drept**, și când are un singur subarbore se face precizarea dacă este subarbore stâng sau subarbore drept. Un arbore binar în care orice vârf are **0 sau 2 descendenți** se numește **arbore binar strict**. Un arbore binar strict în

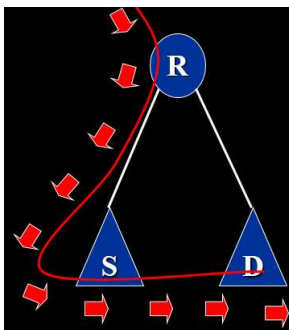
care vârfurile terminale apar pe cel mult două nivele consecutive se numește *arbore echilibrat*. Un arbore binar în care orice vârf are 2 descendenți se numește *arbore binar complet (plin)*.

Dintre operațiile posibile cu arbori un rol important îl deține parcurgerea arborilor. Prin *parcurgerea arborilor* se înțelege considerarea unei ordini totale asupra nodurilor din arbore. Astfel se poate spune pentru orice nod care nod îl precede și care îl succede logic. Orice permutare de noduri este o parcurgere, dar numai unele dintre ele sunt semnificative pentru o structură arborescentă.

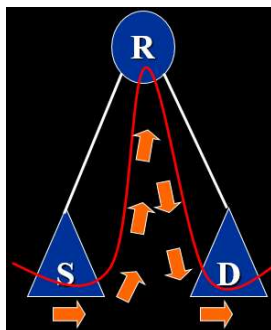
Astfel, pentru **arborii binari** se folosesc de obicei trei tipuri de parcurgere:

- *parcurgerea în preordine*, în care se prelucrează rădăcina, apoi se parcurge în preordine subarboarele stâng și se parcurge în preordine subarboarele drept (RSD);
- *parcurgerea în inordine*, în care se parcurge în inordine subarboarele stâng, apoi se prelucrează rădăcina și se parcurge în inordine subarboarele drept (SRD);
- *parcurgerea în postordine*, în care se parcurge în postordine subarboarele stâng apoi se parcurge în postordine subarboarele drept și se prelucrează rădăcina (SDR).

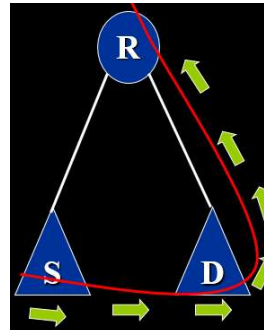
Preordine (RSD)



Inordine (SRD)



Postordine (SDR)



Se consideră că parcurgerea unui arbore vid nu produce nici o acțiune, arborele respectiv fiind considerat parcurs.

De exemplu, pentru arborele din figura 1, parcurgerea în preordine este: 1, 2, 3, 4, 5, 6, 7, 8, 9, parcurgerea în inordine este: 4, 3, 2, 6, 5, 7, 8, 9, iar parcurgerea în postordine este: 4, 3, 6, 7, 5, 2, 9, 8, 1.

Alt exemplu:

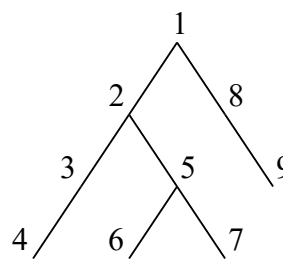
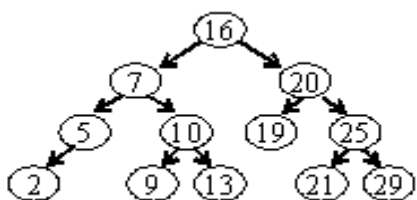


figura 1



- a) în *preordine* : 16,7,5,2,10,9,13,20,19,25,21,29;
 b) în *inordine* : 2,5,7,9,10,13,16,19,20,21,25,29;
 c) în *postordine* : 2,5,9,13,10,7,19,21,29,25,20,16.

Alt exemplu: să considerăm arborele binar de mai jos.

Parcurea în preordine:

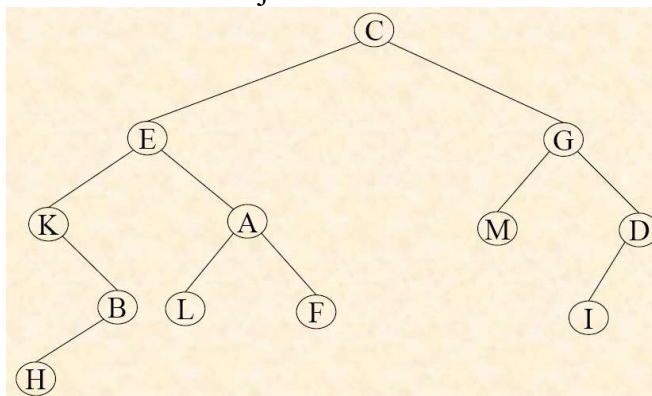
C, E, K, B, H, A, L, F, G, M, D, I.




Parcurea în inordine:

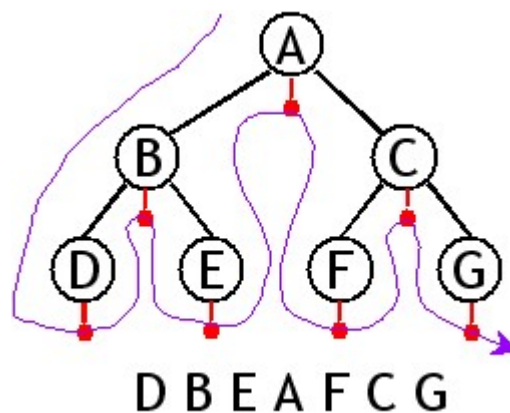
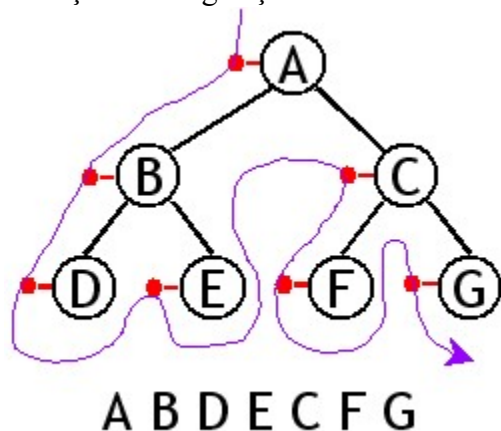
K, H, B, E, L, A, F, C, M, G, I, D.

Parcurea în preordine:

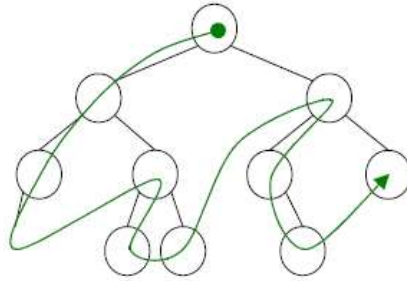
C, E, K, B, H, A, L, F, G, M, D, I.



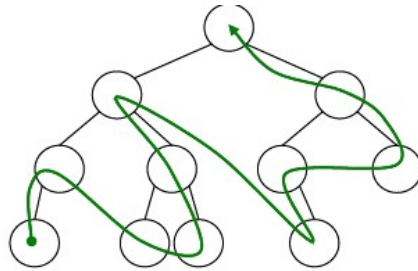
Pentru a determina mai ușor aceste parcureri se pot folosi stegulețe atașate fiecărui nod,  pentru preordine,  pentru inordine și  pentru postordine și trebuie colecționate stegulețele:



În general, o reprezentare schematică a ordinii în care se parcurg în preordine nodurile din arbore binar:



respectiv în postordine:



Pentru **arbori oarecare** se folosesc următoarele parcurgeri:

- *parcurea în **preordine***, în care se prelucrează rădăcina, apoi se parcurg în preordine subarborii acestei rădăcini;
- *parcurea în **postordine***, în care se parcurg în postordine subarborii corespunzători rădăcinii apoi se prelucrează rădăcina.

De exemplu, pentru arborele

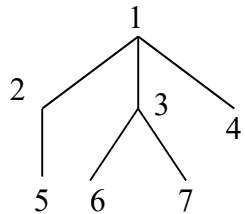
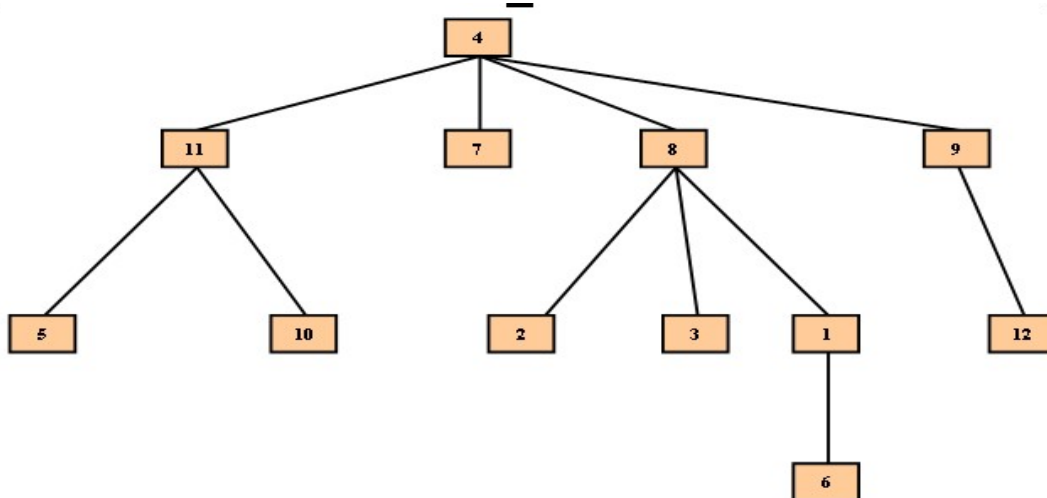


figura 2

parcurea în preordine este: 1, 2, 5, 3, 6, 7, 4, iar parcurea în postordine este: 5, 2, 6, 7, 3, 4, 1.

Alt exemplu: pentru arborele



- parcurea în preordine: 4, 11, 5, 10, 7, 8, 2, 3, 1, 6, 9, 12;
- parcurea în postordine: 5, 10, 11, 7, 2, 3, 6, 1, 8, 12, 9, 4.

Pot fi efectuate și alte tipuri de operații cu arbori cum ar fi adăugarea unor noi noduri, eliminarea unor noduri, combinarea a doi sau mai mulți rbori, copierea unui arbore și altele asemenea.

2. Reprezentarea arborilor oarecare

Există mai multe modalități de reprezentare a arborilor oarecare:

1. Reprezentarea secvențială cu ajutorul informațiilor **fiu-frate**:

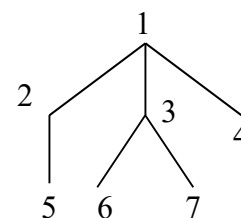
O primă modalitate de reprezentare a unui arbore ordonat oarecare este aceea de a memora pentru fiecare vârf i următoarele informații reținute în vectori:

- $fiu[i]$ reprezentând primul dintre descendenții vârfului i (sau **fiul cel mai din stânga**);
- $frate[i]$ reprezentând descendentul tatălui lui i , descendent care urmează imediat lui i (**fratele imediat din dreapta**).

Lipsa fiului, respectiv a fratelui, este marcată prin valoarea 0.

De exemplu, pentru arborele din figura 2 avem

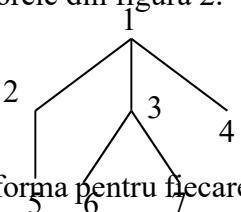
nodul i	1	2	3	4	5	6	7
info[i]	1	2	3	4	5	6	7
fiu[i]	2	5	6	0	0	0	0
frate[i]	0	3	4	0	0	7	0



figura

2. O altă modalitate de reprezentare secvențială este aceea de a într-un vector **tata** nodului i (lipsa tatălui unui nod este marcată prin valoarea 0). De exemplu, pentru arborele din figura 2.

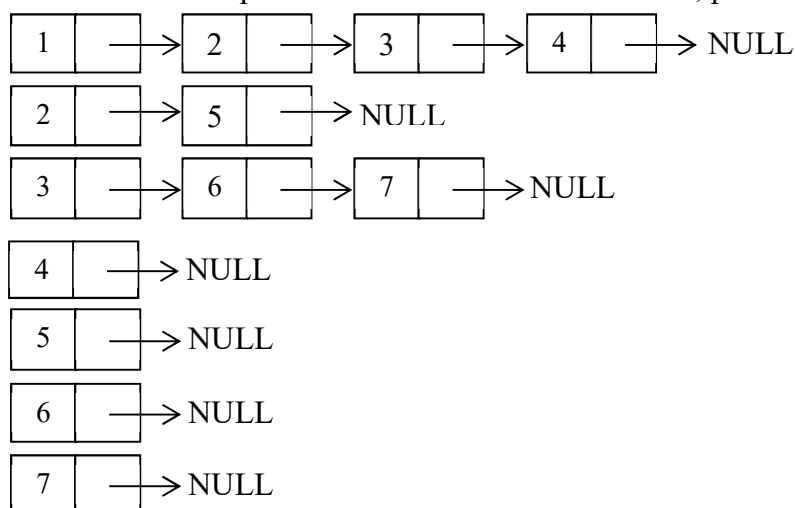
nodul i	1	2	3	4	5	6	7
info[i]	1	2	3	4	5	6	7
tata[i]	0	1	1	1	2	3	3



figura

3. O a treia modalitate de reprezentare a unui arbore oarecare este de a forma pentru fiecare vârf o *listă a descendenților* (fiilor) săi.

În cazul reprezentării dinamice a listelor de fii, pentru exemplul din figura 1 avem



În cazul reprezentării secvențiale a listelor de fii: considerând N numărul de vârfuri ale arborelui, se poate folosi un vector *info* cu N componente pentru a memora informația conținută în vârfuri și un vector *cap* cu N componente întregi cu semnificația

$$cap(i) = \begin{cases} 0 & , \text{ dacă } i \text{ este un vârf terminal} \\ c > 0, & \text{ dacă lista descendenților lui } i \text{ începe de la adresa } c \end{cases}$$

Pentru a memora listele descendenților vârfurilor se poate folosi, de exemplu o matrice M cu două linii și $N-1$ coloane în care memorăm aceste liste, folosind pentru ele alocarea în lanțuită. Pentru fiecare $i \in \{1, \dots, N-1\}$, $M(1, i)$ reprezintă un descendent al vârfului pentru care lista atașată conține coloana i din M , iar $M(2, i)$ reprezintă numărul coloanei corespunzătoare următorului descendent sau 0 (dacă nu există vreun alt descendent). Pentru exemplul considerat mai sus, avem:

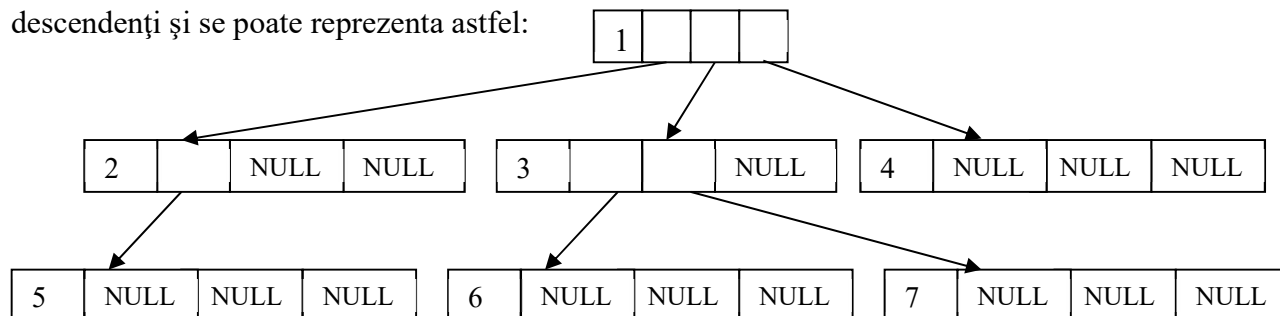
$$cap = (5, 3, 6, 0, 0, 0, 0) \quad M = \begin{pmatrix} 3 & 7 & 5 & 4 & 2 & 6 \\ 4 & 0 & 0 & 0 & 1 & 2 \end{pmatrix}$$

4. Utilizând *structurile de date* dinamice obținem o reprezentare arborescentă pentru arborii orientați în felul următor: presupunând că fiecare vârf al arborelui are cel mult n descendenți, fiecărui vârf îi este atașată structura:

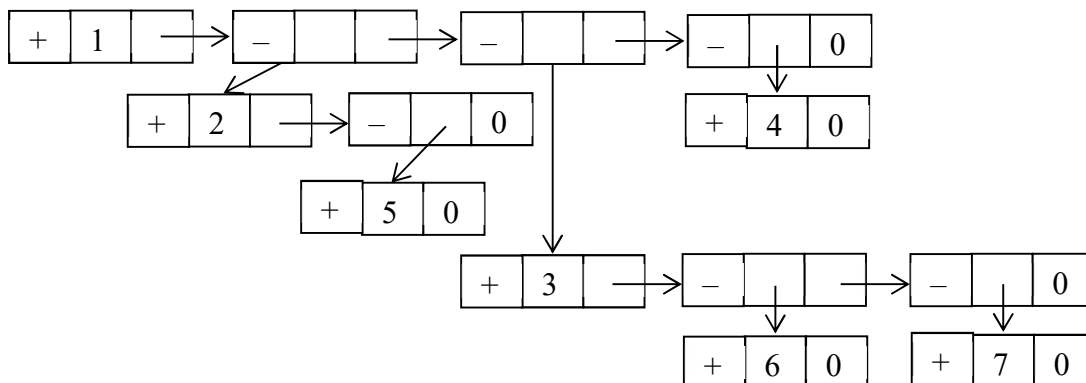
Identificatorul vârfului	Vector de legături către descendenții vârfului		
	Legătura către fiu 1	...	Legătura către fiu n

Dacă un vârf are $p < n$ descendenți, atunci primele p legături indică legăturile către descendenți, ultimele $n-p$ legături fiind NULL.

De exemplu, arborele din figura 2 are proprietatea că fiecare vârf are maxim trei descendenți și se poate reprezenta astfel:



5. Altă modalitate de reprezentare a arborilor oarecare este cu *structuri de liste*, care folosește liste dinamice ale descendenților unui nod. Pentru exemplul din figura 1, reprezentarea este următoarea (marcajul ‘-’ specifică faptul că celula conține numai legături și ‘+’ că celula conține și date):



3. Reprezentarea arborilor binari

Modul de reprezentare a arborilor binari este de obicei prin folosirea unor celule ce conțin câmpurile: *INFO* în care se pun datele asociate nodului respectiv, *LS* care conține legătura la celula în care se află nodul fiu din stânga și *LD* care conține legătura la celula în care se află nodul fiu din dreapta al nodului respectiv.

În unele aplicații este util și un al treilea câmp de legătură numit *TATA* care conține legătura la celula în care se află tatăl nodului respectiv.

De exemplu, pentru arborele din figura 1, în cazul reprezentării secvențiale avem

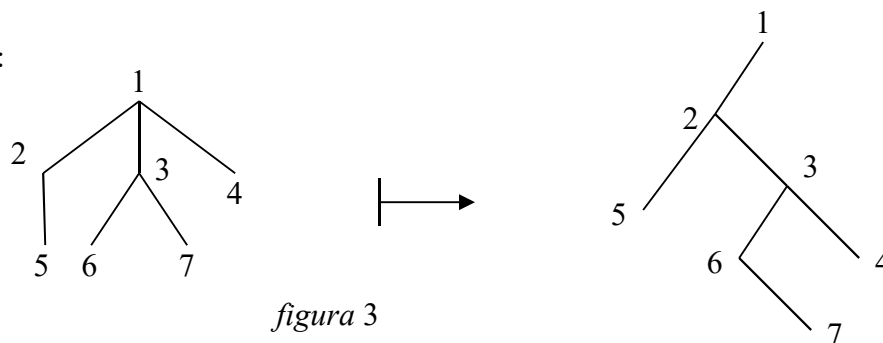
nodul <i>i</i>	1	2	3	4	5	6	7	8	9
INFO[<i>i</i>]	1	2	3	4	5	6	7	8	9
LS[<i>i</i>]	2	3	4	0	6	0	0	0	0
LD[<i>i</i>]	8	5	0	0	7	0	0	9	0

iar în cazul reprezentării înlănțuite, fiecare nod al arborelui poate avea următoarea structură

```
struct nod
{
    int info;
    nod *ls, *ld;
} *rad;
```

Dacă se folosește pentru un arbore oarecare reprezentarea *fiu-frate*, **fiecărui arbore oarecare i se poate atașa un arbore binar**, identificând *fiu* cu *st* și *frate* cu *dr*. Această corespondență este reciprocă.

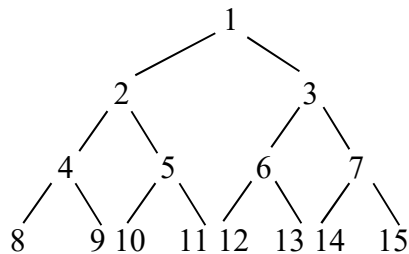
De exemplu:



Corespondența descrisă mai sus reduce **parcurea arborilor oarecare în preordine la parcurea în preordine a arborilor binari atașați**.

Aplicația 1. Să se creeze un arbore binar complet (plin) și să se parcurgă arborele în preordine, inordine și postordine, folosind pentru parcurgeri funcții recursive.

Soluție: Deoarece într-un arbore binar complet (plin) fiecare nod are exact doi descendenți, arborele are $2^n - 1$ noduri, unde n reprezintă numărul de nivele al acestuia (nodul rădăcină al arborelui considerându-se pe nivelul 1). De exemplu, un arbore binar complet cu 4 nivele poate arăta astfel:



În acest caz, parcurgerea în preordine (rădăcină, descendent stâng, descendent drept) înseamnă vizitarea nodurilor 1, 2, 4, 8, 9, 5, 10, 11, 3, 6, 12, 13, 7, 14, 15 parcurgerea în inordine (descendent stâng, rădăcină, descendent drept) înseamnă 8, 4, 9, 2, 10, 5, 11, 1, 12, 6, 13, 3, 14, 7, 15 și parcurgerea în postordine (descendent stâng, descendent drept, rădăcină) înseamnă 8, 9, 4, 10, 11, 5, 2, 12, 13, 6, 14, 15, 7, 3, 1.

Se pot folosi următoarele declarații

```

struct nod
{
    int info; //datele asociate nodului respectiv
    nod *ls, *ld; //legături către fiul din stânga/dreapta
} *rad; //arborele este dat de adresa celulei radacina

void CreareArb(nod *&rad, int n)
//crearea se face pe nivele 1, 2,...n
{
    nod *p, *fr[100];
    int i=1, nv; //nv=numarul de nivele create deja
    rad = new(nod); rad->info=1;
    if (n==1) {rad->ls = rad->ld = NULL;}
    else
    {
        fr[1] = rad; nv = 1;
        while (nv<n)
        {
            //pt. fiecare nod de pe nivelul anterior
            for (int j=pow(2,nv-1); j<pow(2,nv); j++)
            {
                p = fr[j];
                p->ls = new nod;
                p->ls->info = ++i;
                fr[i] = p->ls;
                p->ld = new nod;
                p->ld->info = ++i;
                fr[i] = p->ld;
            }
            nv++;
        }
    }
    for (int j=pow(2,n-1); j<pow(2,n); j++) //ultimul nivel
    {
        p = fr[j]; p->ls = p->ld = NULL;
    }
}
  
```

```

void Preordine(nod *p)
{if (p!=NULL)
  {cout<<p->info<<"  "; Preordine(p->ls); Preordine(p->ld);}
}
void Inordine(nod *p)
{if (p!=NULL)
  {Inordine(p->ls); cout<<p->info<<"  "; Inordine(p->ld);}
}
void Postordine(nod *p)
{if (p!=NULL)
  {Postordine(p->ls);Postordine(p->ld);cout<<p->info<<"  ";}
}
void StergeArb(nod *&rad)//recursiv, in preordine
{nod *ls, *ld;
  if (rad!=NULL)
  {ls = rad->ls;
   ld = rad->ld;//se retin adresele descendentialor
   delete(rad);
   StergeArb(ls); StergeArb(ld); //se sterg descendentiali
  }
}

```

Aplicația 2. Să se implementeze crearea recursivă a unui arbore binar.

Soluție: Crearea recursivă a arborelui se poate face în preordine (întâi se creează nodul, apoi recursiv descendenții), informația nodurilor dându-se în preordine. Pentru nodurile finale ale arborelui se poate da ca informație un element din afara domeniului informației din nod (de exemplu, când nodurile arborelui sunt de tip întreg se poate da un caracter, constanta NULL, etc.). În acest caz, funcția C++ poate arăta astfel:

```

void CreareArb_Preordine(nod *&p)
{int x;
  cin>>x;
  if (x!=0)
  //0 repr. informatia din “descendenții” frunzelor și care nu se găsește în alte noduri
  {
    p = new nod;
    p->info = x;
    CreareArb_Preordine(p->arb_st);
    CreareArb_Preordine(p->arb_dr);
  }
  else p = NULL;
}

```

Aplicatia 3 (suplimentar). Să se implementeze variantele nerecursive ale parcurgerii unui arbore binar în preordine, inordine și postordine.

Soluție: Variantele nerecursive folosesc o stivă și funcțiile pentru prelucrarea acesteia:

```
int ns; //numarul de elemente efective din stiva
nod *stiva[100];
void InitStiva() {ns=0;}
int StivaVida() {return ns==0;}
void InserareStiva(nod* elem) {stiva[ns++]=elem;}
nod* VarfStiva() {ns--; return stiva[ns];}
```

În acest caz, funcțiile nerecursive pentru parcurgerea în preordine și inordine pot fi:

```
void PreordineNerecursiv(nod *p)
{InitStiva();
 while (1)
 {
  while (p!=NULL)
  {cout<<p->info<<" "; //afisez informatia din nod
   InserareStiva(p); //retin in stiva nodul
   p = p->ls; //se trece la descendentul stang
  }
  if (StivaVida()) break;
  else
  {p = VarfStiva(); //se scoate varful stivei
   p = p->ld; //se trece la descendentul drept al sau
  }
 }
}

void InordineNerecursiv(nod *p)
{InitStiva();
 while (1)
 {
  while (p!=NULL) {InserareStiva(p); p = p->ls;}
  if (StivaVida()) break;
  else {p=VarfStiva();cout<<p->info<<" ";p=p->ld;}
 }
}
```

La parcurgerea în postordine, pentru că trebuie parcurși atât subarborele drept cât și cel drept înainte de a vizita nodul, trebuie reținut fiecare nod de două ori în stivă: o dată pentru a putea avea acces la legătura din stânga și a doua oară pentru legătura din dreapta. De aceea fiecărui nod introdus în stivă i se poate asocia o componentă a unui vector cu valoarea 0 dacă a fost introdus în stivă o dată sau 1 când este introdus în stivă a doua oară:

```
int nr_retineri[100];
```

În acest caz, funcția nerecursivă pentru parcurgerea arborelui în postordine poate fi:

```
void PostordineNerecursiv(nod *p)
{InitStiva()
 while (1)
 {
  while (p!=NULL)
```

```

    {InserareStiva(p); nr_retineri[ns-1]=0;
      p=p->ls;
    }
    while (!StivaVida())
    {p=VarfStiva();
      if (nr_retineri[ns]==0)
      {InserareStiva(p); nr_retineri[ns-1]=1;
        p=p->ld; break;
      }
      else cout<<p->info<<" ";
    }
    if (StivaVida()) break;
  }
}



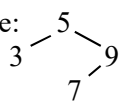
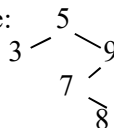
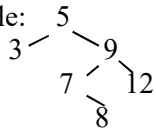
```

4. **Arbori binar de căutare/sortare**

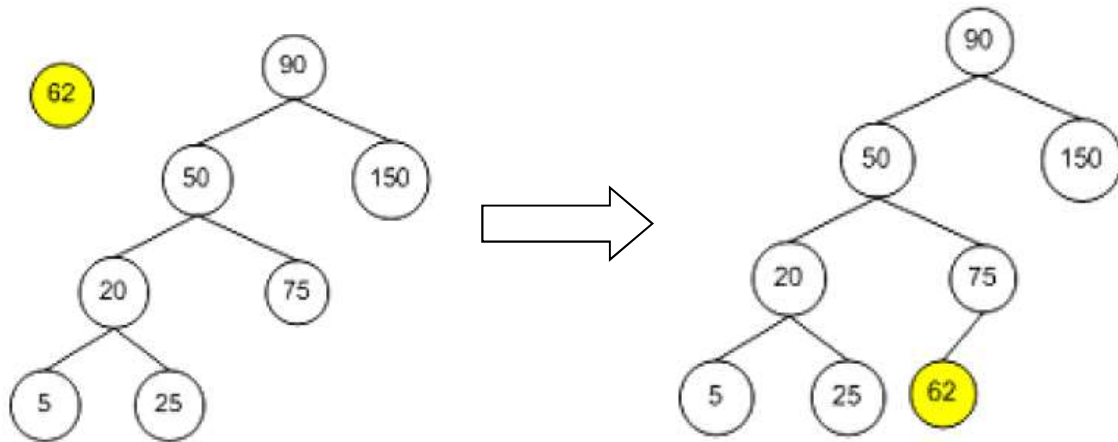
Arborele binar de căutare (sortare) are proprietatea că pentru fiecare nod neterminal al arborelui valoarea sa este mai mare decât valorile nodurilor din subarborele stâng și mai mică sau egală decât valorile nodurilor din subarborele drept.

Un nou nod se inserează în arbore căutând poziția corespunzătoare și se inserează întotdeauna ca *nod frunză*.

De exemplu:

- Adauga(rad , 5) va conduce la arborele: 5
- Adauga(rad , 9) va conduce la arborele: 
- Adauga(rad , 7) va conduce la arborele: 
- Adauga(rad , 3) va conduce la arborele: 
- Adauga(rad , 8) va conduce la arborele: 
- Adauga(rad , 12) va conduce la arborele: 

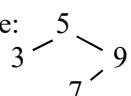
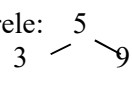
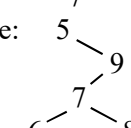
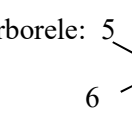
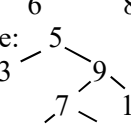
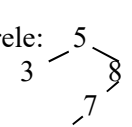
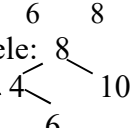
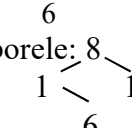
Alt exemplu:



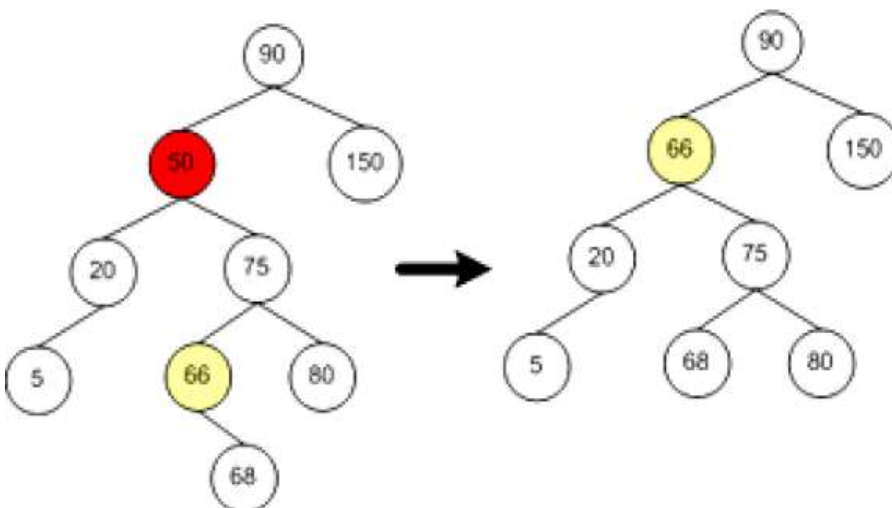
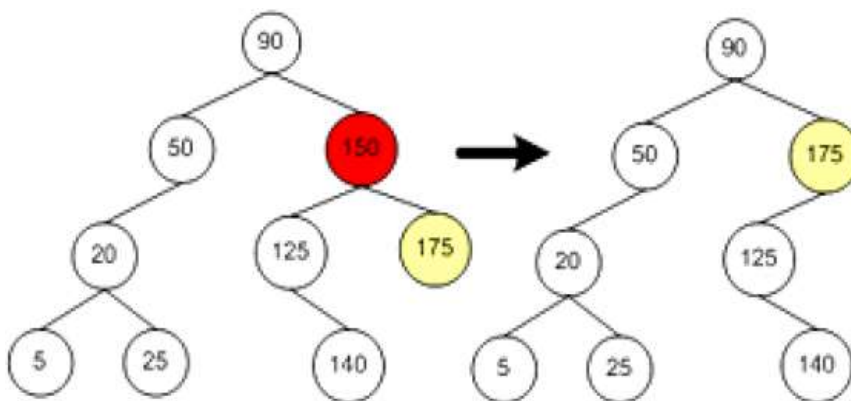
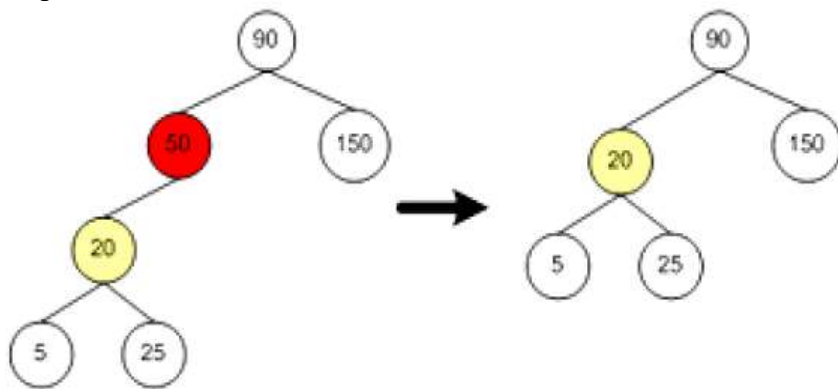
Pentru **ștergerea unui nod dintr-un arbore binar de căutare/sortare**, există 3 cazuri posibile:

- nodul ce trebuie eliminat este nod terminal** (frunză); în acest caz se șterge pur și simplu acel nod din arbore;
- nodul are un singur descendent direct**; în acest caz se face legătura părintelui său la acel descendent;
- nodul are doi descendenți direcți**; în acest caz acestui **nod i se atribuie o valoare** adecvată luată dintr-un nod terminal, apoi se șterge nodul terminal din care s-a copiat informația. Sunt posibile două variante de alegere a nodului terminal: **cel mai mare element mai mic** decât nodul care se dorește a fi șters, respectiv cel mai mic element mai mare decât nodul care se dorește a fi șters. Aceste două variante se reduc la *alegerea celui mai din dreapta nod al subarboarelui stâng al nodului care se dorește a fi șters, respectiv a celui mai din stânga nod al subarboarelui drept*. Aici am ales prima variantă.

De exemplu:

- pentru arborele:  Sterge(rad, 7) conduce la arborele: 
- pentru arborele:  Sterge(rad, 9) va conduce la arborele: 
- pentru arborele:  Sterge(rad, 9) conduce la arborele: 
- pentru arborele:  Sterge(rad, 4) conduce la arborele: 

Alte exemple:



Aplicația 4. Să se implementeze operația de adăugare a unui nod și de căutare a unui element într-un arbore binar de căutare/sortare.

Soluție: Cum arborele binar de căutare (sortare) are proprietatea că pentru fiecare nod neterminal al arborelui valoarea sa este mai mare decât valorile nodurilor din subarborele stâng și mai mică sau egală decât valorile nodurilor din subarborele drept, atunci căutarea unui element se poate face recursiv (apelând “Exista(rad,elem)”) astfel:

```
int Exista(nod *p, int elem) //verifica existenta elementului in arbore
{
    if (p)
        if (p->info==elem) return 1;
        else
            if (elem < p->info) return Exista(p->ls,elem);
            else return Exista(p->ld,elem);
    else
        return 0;
}
```

Un nou nod se inserează în arbore căutând poziția corespunzătoare și se *inserează* întotdeauna ca *nod frunză*. Astfel parcurgerea arborelui în inordine va duce la afișarea valorilor nodurilor în ordine crescătoare. Apel: “Adauga(rad,elem)”.

```
void Adauga(nod *&p, int elem) //recursiv
{
    if (p==NULL) //am ajuns dincolo de nodurile din arbore
    {
        p = new nod;
        p->info = elem;
        p->ls = p->ld=NULL;
    }
    else
        if (elem < p->info) Adauga(p->ls,elem);
        else Adauga(p->ld,elem);
}
```

Aplicația 5. Să se implementeze operația de ștergere a unui nod dintr-un arbore binar de căutare/sortare.

Soluție: Există 3 cazuri posibile:

- d) nodul ce trebuie eliminat este nod terminal (frunză); în acest caz se face legătura părintelui său la NULL și se șterge nodul din memorie;
- e) nodul are un singur descendent direct; în acest caz se face legătura părintelui său la acel descendent;
- f) nodul are doi descendenți direcți; în acest caz acestui nod i se atribuie o valoare adecvată luată dintr-un nod terminal, apoi de șters se șterge nodul terminal din care s-a copiat informația. Sunt posibile două variante de alegere a nodului terminal: *cel mai mare element mai mic* decât nodul care se dorește a fi șters, respectiv *cel mai mic element mai mare* decât nodul care se

dorește a fi șters. Aceste două variante se reduc la *alegerea celui mai din dreapta nod al subarborelui stâng al nodului care se dorește a fi șters*, respectiv a celui mai din stânga nod al subarborelui drept. Aici am ales prima variantă.

În acest caz, funcția C++ ce rezolvă problema poate fi:

```
void Sterge(nod *&rad, int elem)
{
    nod *p=rad, *tata=NULL;
    //caut nodul cu info = elem. de sters
    while (p!=NULL && p->info!=elem)
    {
        tata=p;
        if (elem < p->info) p =p->ls;
        else p=p->ld;
    } //p este nodul de sters
    if (p == NULL)
    {
        cout<<endl<<"Elementul nu exista in arbore";
        return; }
    //cazul 1: nodul de sters este frunza
    if (p->ls == NULL && p->ld == NULL) //p este frunza
        if (tata->ls == p) tata->ls = NULL;
        else tata->ld = NULL;
    else
    //cazul 2: nodul de sters are un singur descendent direct
    if (p->ls==NULL || p->ld==NULL)
        if (p->ls != NULL)
            if (tata->ls == p) tata->ls = p->ls;
            else tata->ld = p->ls;
        else
            if (tata->ls == p) tata->ls = p->ld;
            else tata->ld = p->ld;
    else
    //cazul 3: nodul de sters are 2 descendenti directi
    //se cauta frunza cu cea mai apropiata valoare mai mica
    {
        tata=p;
        nod *temp = p->ls;
        //caut in subarborele stang al lui p
        while (temp->ld != NULL)
            {tata = temp; temp = temp->ld; }
        //copiez valoarea din temp in p
        p->info = temp->info;
        //sterg nodul cu care am inlocuit valoarea
        if (tata == p) //nu am mai mers in dreapta lui temp
            tata->ls = temp->ls;
        else tata->ld = temp->ls;
        p = temp; //nodul de sters va fi temp
    }
    delete p;
}
```

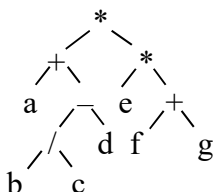

5. Arbori de structură

Fiecărei expresii aritmetice în care apar numai operatori binari i se poate atașa în mod natural un arbore **binar**.

Un arbore de structură are prin definiție vârfurile etichetate astfel:

- fiecare **nod neterminal** este etichetat cu unul dintre operatorii: **+, −, * sau /;**
- fiecare **nod terminal** este etichetat cu un **operand** (**variabilă sau constantă**).

De exemplu, pentru arborele binar:



- **forma infixată** este $((a + ((b/c) - d)) * (e * (f + g)))$
- **forma prefixată** este: $* + a - / b c d * e + f g$
- **forma postfixată** este: $abc/d - + efg + **$

și se pot obține parcurgând arborele în inordine, preordine și postordine.

Pentru crearea arborelui binar corespunzător se poate citi expresia aritmetică infixată într-un șir de caractere, apoi în funcție de prioritatea operatorilor se inserează operatorii și operanzii în arbore. Se folosesc următoarele date:

```

struct nod
{
    char info;
    nod *ls, *ld;
} *rad;          //arborele binar in care se retine expresia
char exp[30], efp[30]; //expresia citita si cea fara paranteze
int p[30], pfp[30];  //vectorul prioritatilor si cel fara paranteze
  
```

Citirea expresiei aritmetice și construirea vectorului priorităților se poate face astfel:

```

cout<<"Introduceti expresia aritmetica : ";
gets(exp);
int j=0, n=strlen(exp);
for (int i=0; i<n; i++)
    switch (exp[i])
    {
        case ')': j -= 10; break;
        case '(': j += 10; break;
        case '+': p[i] = j+1; break;
        case '-': p[i] = j+1; break;
        case '*': p[i] = j+10; break;
        case '/': p[i] = j+10; break;
        default : p[i] = 1000; //operanzii au prioritatea 1000
    }
  
```

Apoi se vor construi expresia fără paranteze și vectorul priorităților expresiei fără paranteze:

```

j = 0;
for (i=0; i<n; i++)
    if ( (exp[i]!=' ') && (exp[i]!='(') )
    {
        efp[j] = exp[i];
        pfp[j++] = p[i];
    } //j = dim. expresiei fara paranteze

```

În final, arborele binar se poate construi apelând instrucțiunea:

```
rad = Arb(0, j-1, efp, pfp);
```

unde funcția recursivă *Arb* determină caracterul din expresia *exp* între limita inferioară *li* și limita superioară *ls*, cu prioritatea cea mai mică și îl inserează în arbore, apoi se reapelează funcția pentru restul expresiei folosind metoda Divide et Impera.

```

nod *Arb(int li, int ls, char efp[30], int pfp[30])
{
    nod *p;
    //se determina caracterul cu prioritatea cea mai mica
    int min = pfp[li], i=li;
    for (int j=li; j<=ls; j++)
        if (pfp[j]<min) { min=pfp[j]; i=j; }
    p = new nod;
    p->info = efp[i];
    if (li==ls) p->ls = p->ld = NULL;
    else {p->ls = Arb(li, i-1, efp, pfp);
        p->ld = Arb(i+1, ls, efp, pfp);
    }
    return p;
}

```