12. ARBORI BINARI ŞI ARBORI DE CĂUTARE

12.1 Structura de date de tip arborescent

Construirea unei structurii de date de tip arborescent porneşte de la problema pe care o avem de rezolvat.

Pentru găsirea soluţiei ecuaţiei $f(x)=0$ pentru $x \in [a,b]$ se efectuează cu metode de calcul numeric, dintre care cea mai simplă este aceea a înjumătăţirii intervalului.

Se calculează:

$$c=(a+b)/2 \qquad (12.1)$$

Dacă:

$$f(a)*f(c)<0 \qquad (12.2)$$

înseamnă că rădăcina $x \in [a,c)$, în caz contrar $x \in (c,b]$.

Noul subinterval este şi el divizat. Astfel, avem imaginea unei modalităţi de lucru ierarhizată pe atâtea niveluri câte sunt necesare obţinerii unei precizii pentru soluţia ecuaţiei $f(x)=0$. Se asociază acestor niveluri reprezentarea grafică din figura 12.1.

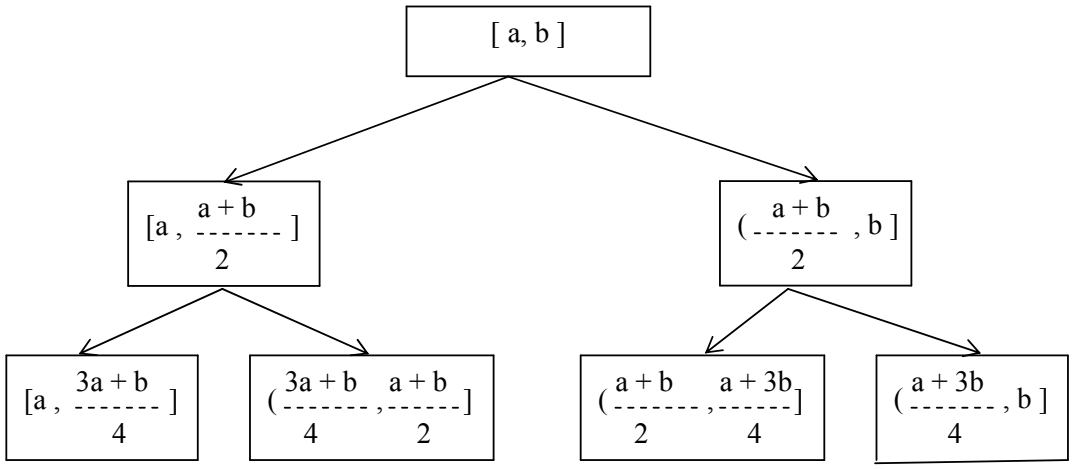

Figura 12.1 *Reprezentarea grafică asociată divizării intervalelor*

Pentru evaluarea expresiei aritmetice:

$$e=\frac{(a+b+c+d)\cdot(x-y)}{c-d} \qquad (12.3)$$

prin modul de efectuare a calculelor conduce la reprezentarea din figura 12.2.

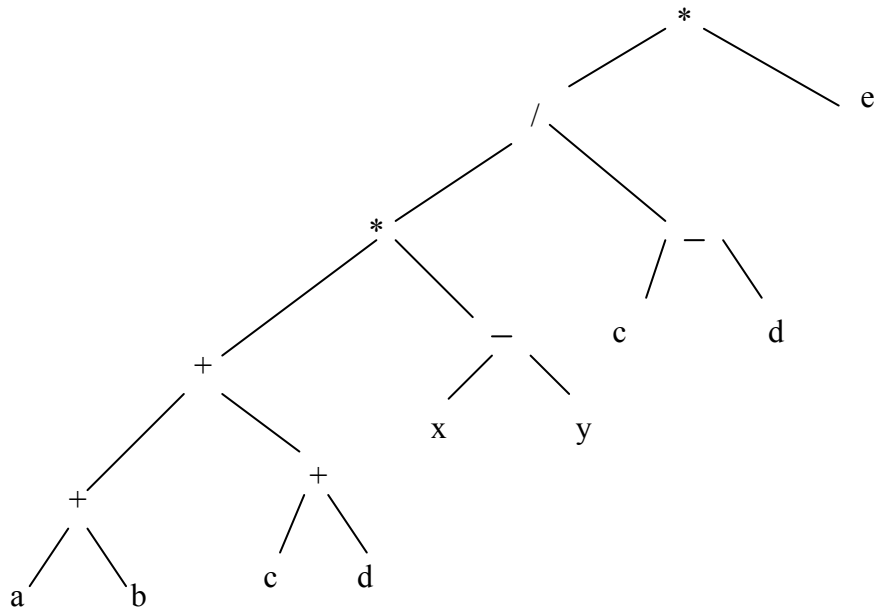

Figura 12.2 *Reprezentarea arborescentă asociată evaluării expresiei*

Pentru regăsirea cărţilor într-o bibliotecă, în fişierul tematic cotele se structurează pe nivele, ca în exemplul din figura 12.3.

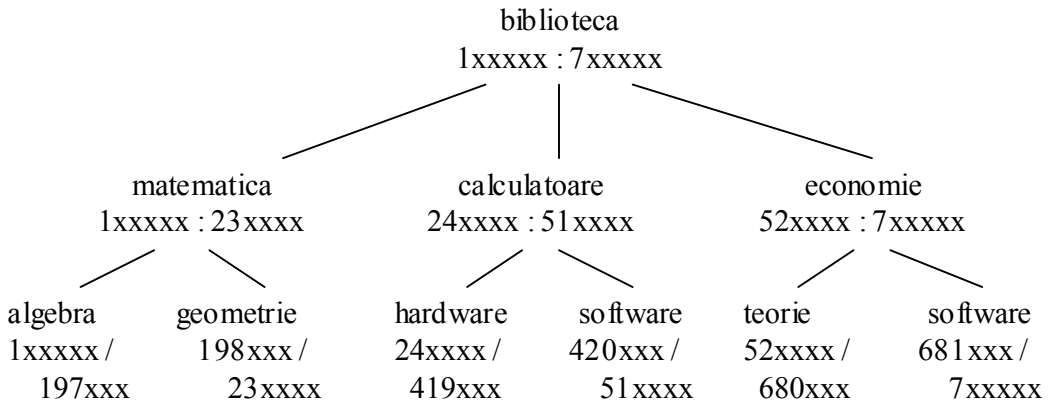

Figura 12.3 *Structura asociată fişierului de cote ale cărţilor*

Reprezentarea în memorie a informaţiilor care au multiple legături între ele, trebuie să fie astfel făcută, încât să poată realiza parcurgerea completă a zonelor sau localizarea unui element din structură.

Se observă că în arborescenţă există un nod numit rădăcină sau părinte. Acesta are descendenţi. Fiecare descendent poate fi la rândul său părinte şi în acest caz are descendenţi.

Arborele binar este caracterizat prin aceea că, orice nod al său are un singur părinte şi maxim doi descendenţi. Fiecare nod este reprezentat prin trei informaţii şi anume:

- Z – informaţia utilă care face obiectul prelucrării, ea descriind elementul sau fenomenul asociat nodului;
- γ – informaţia care localizează nodul părinte;
- θ – informaţia (informaţiile) care localizează nodul (nodurile) descendent (descendente).

Alocarea dinamică determină ca γ şi θ să fie variabile aleatoare, ale căror legi de repartiţie sunt necunoscute. Valorile lor sunt stocate în zona de memorie, formând împreună cu variabila *Z* structura de date asociată unui nod, structură ce corespunde tripletului (Z_j, γ_j, θ_j) asociat nodului *j* dintr-un graf de formă arborescentă binară.

Volumul informaţiilor destinate localizării nodurilor, depinde de obiectivele urmărite şi de rapiditatea cu care se doreşte localizarea unui anumit nod. Cu cât diversitatea prelucrărilor este mai mare, cu atât se impune stocarea mai multor informaţii de localizare.

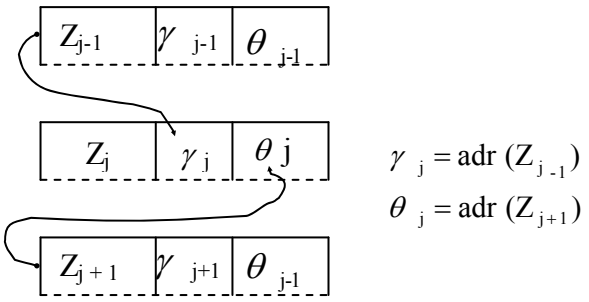

Figura 12.4 *Modelul grafic asociat tripletului (Z_j, γ_j, θ_j)*

Dacă tripletul (Z_j, γ_j, θ_j) ce corespunde modelului grafic din figura 12.4 permite baleierea de sus în jos, sau de jos în sus, pe o ramură a arborelui. Pentru a parcurge pe orizontală arborele, este necesară atât o informaţie suplimentară de localizare δ_j, care corespunde nodului vecin din dreapta de pe acelaşi nivel, cât şi informaţia Ω_j pentru localizarea nodului din stânga de pe acelaşi nivel.

Un astfel de arbore se descrie prin:

$$(Z_j, \gamma_j, \theta_j, \delta_j, \Omega_j) \qquad (12.4)$$

Descrierea completă a unui arbore, presupune crearea mulţimii tripletelor sau cvintuplelor asociate nodurilor.

De exemplu, arborescenţei asociate evaluării expresiei:

$$e = a + b + c; \qquad (12.5)$$

prezentate în figura 12.5 îi corespunde alocarea şi iniţializarea zonelor de memorie din figura 12.6.

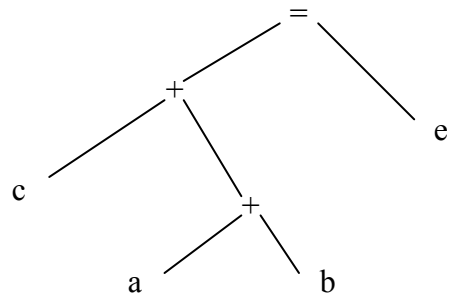

Figura 12.5 *Structura arborescentă asociată evaluării expresiei e*

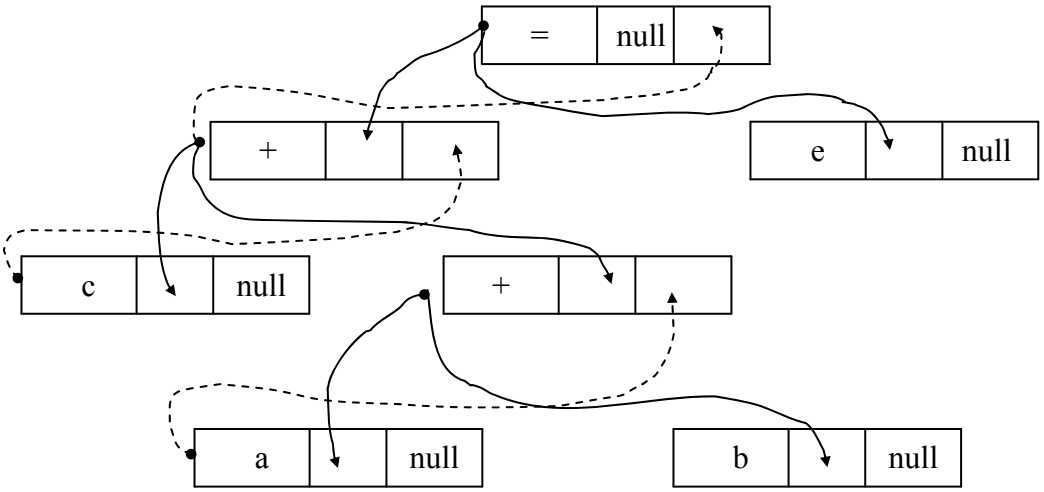

Figura 12.6 *Alocarea şi iniţializarea zonelor de memorie pentru implementarea structurii arborescente*

Acestor reprezentări grafice, trebuie să le corespundă algoritmi pentru încărcarea adreselor, întrucât a construi forma liniarizată a structurii arborescente, înseamnă în primul rând, a aloca zone de memorie şi în al doilea rând, a iniţializa membrii structurii cu adrese care să permită reconstruirea corectă a înlănţuirii.

Se observă că pentru iniţializarea corectă, este necesară existenţa la un moment dat a pointerilor spre trei zone de memorie alocate dinamic şi anume:

- *pp* – pointerul spre zona alocată nodului precedent (părinte);
- *pc* – pointerul spre zona asociată nodului curent;
- *pd* – pointerul spre zona asociată primului descendent;

Construirea nodului $(\mathrm{Zj}, \gamma\,\mathrm{j}, \theta\,\mathrm{j})$ revine la efectuarea iniţializărilor:

$$\begin{aligned}
&\mathrm{ref(pc).Z_j = valoare_citita;}\\
&\mathrm{ref(pc).}\,\gamma_{\mathrm{j}} = \mathrm{pp;}\\
&\mathrm{ref(pc).}\,\theta_{\mathrm{j}} = \mathrm{pd;}\\
&\mathrm{pp = pc;}\\
&\mathrm{pc = pd;}\\
&\mathrm{new(pd);}
\end{aligned} \qquad (12.6)$$

După efectuare secvenţei se face trecerea la pasul următor. Folosind simbolurile, orice structură arborescentă se reprezintă prin:

- *N* mulţimea nodurilor
- *A* mulţimea arcelor

De exemplu, arborescenţa dată pentru calculul expresiei 12.5 se reprezintă prin:

$$N = \{\, e\,,\ =\ ,\ a,\ +,\, b,\, +\,,\, c\,\} \qquad (12.7)$$

şi

$$A = \{ (a+),\ (b+),\ (c+),\ (+, =),\ (=, e) \} \quad (12.8)$$

Se observă că pentru fiecare element al mulţimii N, se alocă dinamic zona de memorie care conţine şi informaţiile γ_j, θ_j . La alocare este posibilă numai iniţializarea zonei Z_j.

Odată cu alocarea se iniţializează şi un vector de pointeri, cu adresele corespunzătoare zonelor de memorie puse în corespondenţă cu nodurile. În continuare, preluând elementele mulţimii A, are loc completarea câmpurilor γ_j şi θ_j. De exemplu, pentru arborele binar din figura 12.7 construirea se efectuează prin alocarea a 7 zone de memorie cu structura $(Z_j, \gamma_j, \theta_j)$ pentru nodul din mulţimea $N = \{A, B, C, D, E, F, G\}$ şi se iniţializează vectorul de pointeri pp[i] după cum urmează:

$$\begin{aligned} pp[1] &= adr(ZA) \\ pp[2] &= adr(ZB) \\ pp[3] &= adr(ZC) \\ pp[4] &= adr(ZD) \\ pp[5] &= adr(ZE) \\ pp[6] &= adr(ZF) \\ pp[7] &= adr(ZG) \end{aligned} \quad (12.9)$$

Baleierea mulţimii:

$$A = \{(AB), (AC), (BD), (BE), (CF), (CG)\} \quad (12.10)$$

revine la efectuarea atribuirilor:

$$\begin{aligned} &ref(pp[1]).\theta = pp[2] \\ &ref(pp[2]).\theta = pp[4] \\ &ref(pp[3]).\theta = pp[6] \\ &ref(pp[4]).\theta = ref(pp[5]).\theta = ref(pp[6]).\theta = ref(pp[7]).\theta = NULL \\ &ref(pp[2]).\gamma = pp[1] \\ &ref(pp[4]).\gamma = pp[2] \\ &ref(pp[3]).\gamma = pp[1] \\ &ref(pp[6]).\gamma = pp[3] \\ &ref(pp[7]).\gamma = pp[3] \\ &ref(pp[1]).\theta = ref(pp[1]).\gamma = NULL \end{aligned} \quad (12.11)$$

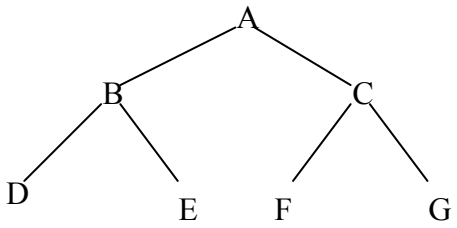

Figura 12.7 *Arbore binar*

Pentru arborii binari echilibraţi, există posibilitatea ca după alocarea memoriei să se descrie elementele mulţimii A, după care iniţializările câmpurilor θ_j şi γ_j să se facă prin apelarea unei funcţii. Problematica devine mai simplă, dacă arborele binar este complet, adică are *n* niveluri la bază $2^{(n-1)}$ elemente descendente, fără a fi părinte, noduri terminale.

În programele C/C++, pentru implementarea structurilor de date necontigue, arborele binar se defineşte prin următorul tip de bază derivat.

```
struct arbore
      {
            int valoare;
            arbore *precedent;
            arbore *urmator;
      };
class arb
      {
            arbore *rad;
      };

struct arbore_oarecare
      {
            int valoare;
            arbore_oarecare **fii; //lista fiilor unui nod
            int nrfii;        //nr de fii ai nodului
      };
class arb_oarecare
      {
            arbore_oarecare *rad;
      };

struct arbore_binar
      {
            int valoare;
            arbore_binar *stanga;
            arbore_binar *dreapta;
      };
class arb_binar
      {
            arbore_binar *rad;
      };
```

12.2 Transformarea arborilor oarecare în arbori binari

Aplicaţiile concrete, asociază unor obiecte, subansamble sau procese, structuri de tip arborescent care nu sunt binare, astfel încât se conturează ideea ca arborii binar sunt cazuri particulare de arbori, mai ales prin frecvenţa cu care sunt identificaţi în practică. Mecanismele de realizare şi de utilizare a arborilor binar îi fac practice, deşi în realitate au frecvenţa de apariţie scăzută.

Apare problema transformării unei structuri arborescente oarecare într-o structură arborescentă binară, problema rezolvabilă prin introducerea de noduri fictive. Astfel, fiind dată arborescenţa din figura 12.8, transformarea ei în structură arborescentă binară, revine la introducerea nodurilor fictive, x, y, u, v, w, rezultând arborele din figura 12.9.

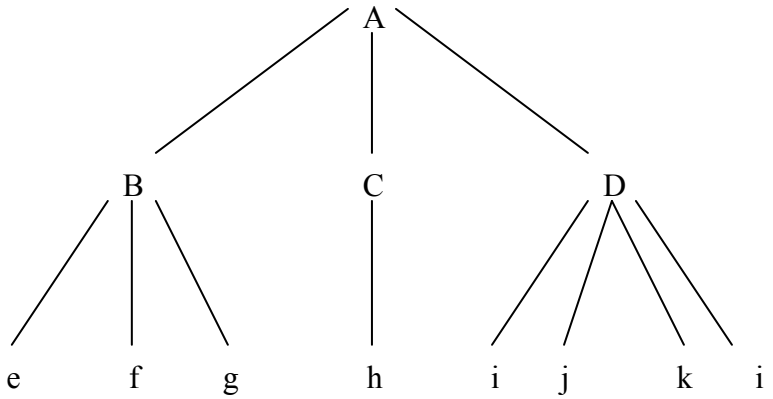

Figura 12.8 *Structură arborescentă oarecare*

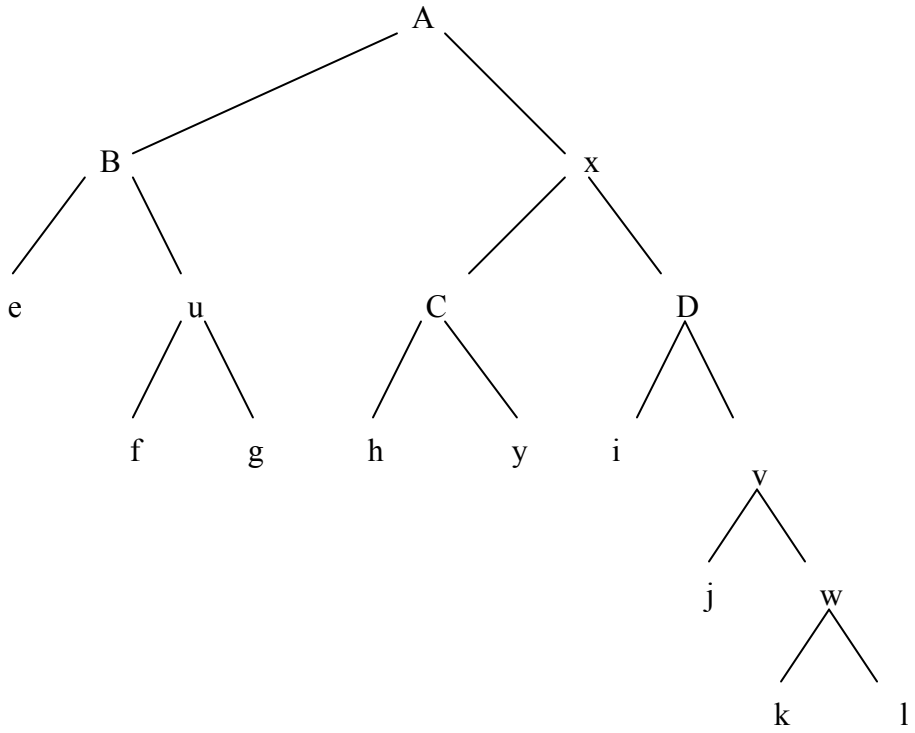

Figura 12.9 *Structură arborescentă cu noduri fictive*

Arborele oarecare, are un număr de noduri mai mic decât arborele binar, nodurilor fictive corespunzându-le zone de memorie structurate (NULL, γ j, θj).

Alocarea dinamică presupune, ca în zona $[D_i, D_f]$ prin dealocare să apară goluri, adică zone libere ce sunt realocate altor variabile. Este necesară o gestionare a acestor zone şi periodic trebuie să se efectueze o realocare prin reorganizare, aşa fel încât să dispară golurile rezultate în procesul de alocare-dealocare multiplă. De exemplu, pentru un program P, este necesară alocarea a 3 zone de memorie de 1500, 2000, 4000 baiţi, ce corespund arborilor binar A, B şi C.

Alocarea este efectuată iniţializând variabilele pointer p_A, p_B şi p_C prin apelul succesiv al funcţiei *alocare( )*, (pas 1).

Dealocarea după un timp a arborelui binar B, determină apariţia unui gol între zonele ocupate de variabilele A şi C, (pas 2).

Alocarea unei zone de memorie pentru arborii binari D (3000 baiţi) şi E (1000 baiţi), revin la a dispune pe D în continuarea lui C şi a intercala arborele E între A şi C, în "golul" rezultat din dealocarea lui E, rămânând între E şi C un "gol" de 300 baiţi, (pas 3).

Dacă se păstrează informaţiile privind modul de iniţializare a variabilelor pointer care stochează adresele nodurilor rădăcină a arborilor A, E, C şi D, este posibilă glisarea informaţiilor în aşa fel încât să dispară "golul" dintre E şi C. Nu s-a luat în considerare însăşi necontiguitatea din interiorul fiecărui arbore.

În practică, apare problema optimizării dispunerii variabilelor dinamice, dar şi cea a alegerii momentului în care dispersia elementelor atinge un astfel de nivel, încât zona pentru alocare dinamică este practic inutilizabilă şi trebuie reorganizată.

12.3 Arbori binari de căutare

Un arbore binar de căutare este un arbore binar care are proprietatea că prin parcurgerea în inordine a nodurilor se obţine o secvenţă monoton crescătoare a cheilor (cheia este un câmp ce serveşte la identificarea nodurilor în cadrul arborelui). Câmpul cheie este singurul care prezintă interes din punct de vedere al operaţiilor care se pot efectua asupra arborilor de căutare.

Principala utilizare a arborilor binari de căutare este regăsirea rapidă a unor informaţii memorate în cheile nodurilor. Pentru orice nod al unui arbore de căutare, cheia acestuia are o valoare mai mare decât cheile tuturor nodurilor din subarborele stâng şi mai mică decât cheile nodurilor ce compun subarborele drept.

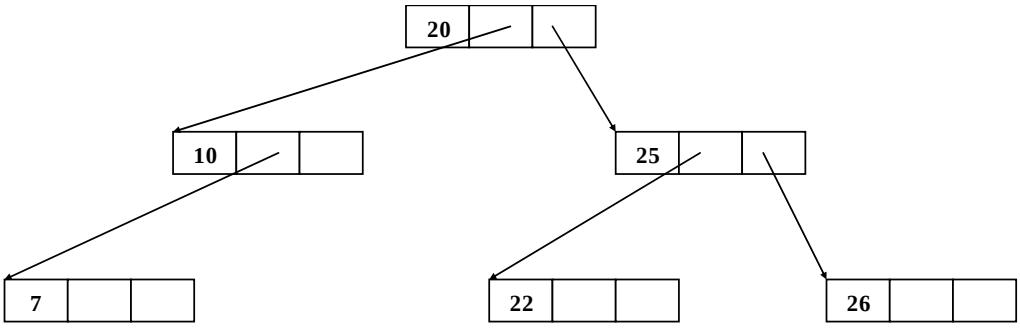

Figura 12.10 *Arbore binar de căutare*

Structura de date folosită pentru descrierea unui nod al unui arbore binar de căutare va fi următoarea:

```
struct arbore_binar
      {
            int cheie;
            arbore_binar *stanga;
            arbore_binar *dreapta;
      };
class arb_binar
      {
            arbore_binar *rad;
      };
```

Rădăcina arborelui binar de căutare va fi definită în felul următor:

*arb *Radacina = NULL; (12.12)*

Se observă ca fiecare nod este compus din cheia asociată şi din informaţiile de legătură care se referă eventualii fii.

Aşa cum le spune şi numele, arborii binari de căutare sunt folosiţi pentru regăsirea rapidă a informaţiilor memorate în cheile nodurilor. De aceea căutarea unui nod cu o anumită valoarea a cheii este o operaţie deosebit de importantă.

Căutarea începe cu nodul rădăcină al arborelui prin compararea valorii cheii căutate cu valoarea cheii nodului curent. Dacă cele două valori coincide, căutarea s-a încheiat cu succes. Dacă informaţia căutată este mai mică decât cheia nodului, căutarea se continuă în subarborele stâng. Dacă cheia căutată este mai mare decât valoarea cheii nodului, căutarea se reia pentru subarborele drept.

Crearea unui arbore binar de căutare presupune adăugarea câte unui nod la un arbore iniţial vid. După inserarea unui nod, arborele trebuie să rămână în continuare ordonat. Din acest motiv, pentru adăugarea unui nod se parcurge arborele începând cu rădăcina şi continuând cu subarborele stâng sau drept în funcţie de relaţia de ordine dintre cheia nodului şi cheia de inserat. Astfel, dacă cheia de inserat este mai mica decât cheia nodului, următorul nod vizitat va fi rădăcina subarborelui stâng. În mod similar, dacă cheia de inserat este mai mare decât cheia nodului, traversarea se va continua cu nodul rădăcină al subarborelui drept. Această modalitate de traversare se continuă până în momentul în care se ajunge la un nod fără

descendent. Acestui nod îi va fi adăugat un nod fiu cu valoarea dorită a cheii.

Aplicaţiile care utilizează arbori binari de căutare pot permite sau pot interzice, în funcţie de filozofia proprie, inserarea în cadrul arborelui a unei chei care există deja.

Inserarea în cadrul arborelui anterior a unui nou nod cu valoarea cheii egală cu 12 conduce către următorul arbore, figura 12.11.

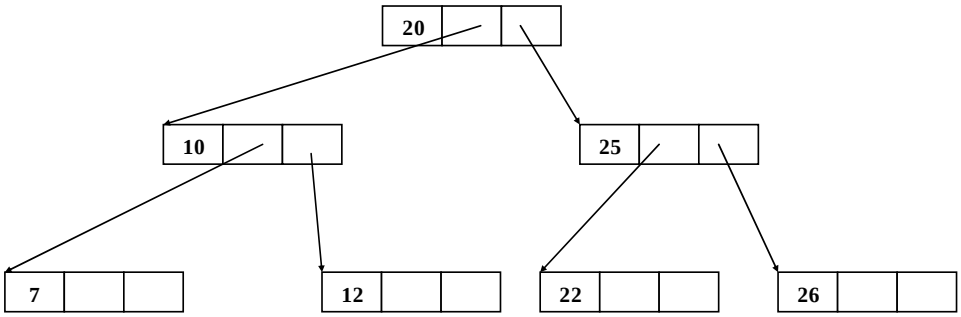

Figura 12.11 *Arbore binar de căutare după inserarea unui nod*

Maniera uzuală de inserare a unui nod într-un arbore binar de căutare este cea recursivă.

În practică, de cele mai multe ori căutarea şi inserarea se folosesc împreună. Astfel, în cazul în care căutarea unei chei s-a efectuat fără succes, aceasta este adăugată la arborele binar de căutare.

O altă operaţie care se poate efectua asupra unui arbore binar de căutare este ştergerea unui nod care are o anumită valoare a cheii. Dacă valoarea cheii este găsită în cadrul arborelui, nodul corespunzător este şters. Arborele trebuie să rămân arbore de căutare şi după ştergerea nodului.

În ceea ce priveşte nodul care va fi şters, acesta se va încadra într-una din variantele următoare:

a) nodul nu are subarbori (fii);
b) nodul are doar subarbore stâng;
c) nodul are doar subarbore drept;
d) nodul are atât subarbore stâng cât şi subarbore drept.

În cazul în care nodul nu are nici subarbore stâng dar nici subarbore drept (varianta a) este necesară doar ştergerea nodului. Nu sunt necesare alte operaţiuni de actualizare a arborelui.

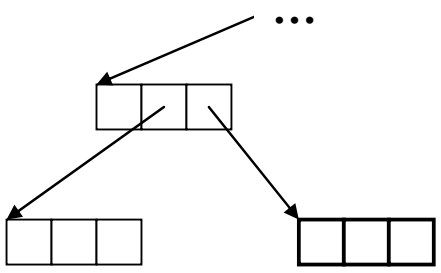

Figura 12.12 *Arbore binar de căutare înainte de ştergerea unui nod*

În figura 12.12 este prezentat un fragment al unui arbore binar de căutare din care dorim să ştergem nodul din dreapta, iar în figura 12.13 se prezintă acelaşi fragment de arbore dar după ştergerea nodului dorit.

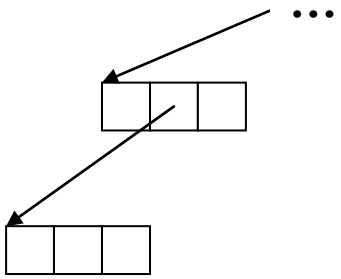

Figura 12.13 *Arbore binar de căutare după ştergerea unui nod*

Pentru cazurile b şi c (nodul pe e dorim să-l ştergem are subarbore stâng sau drept), pe lângă ştergerea nodului este necesară şi actualizarea legăturilor dintre nodurile arborelui. În figurile 12.14 şi 12.16 se prezintă câte un fragment dintr-un arbore binar de căutare din care dorim să ştergem nodul evidenţiat. În figura 12.14, nodul pe care dorim să-l ştergem are numai subarbore stâng iar cel din figura 12.16 are doar subarbore drept. În figurile 12.15 şi 12.17 se pot observa efectele operaţiei de ştergere.

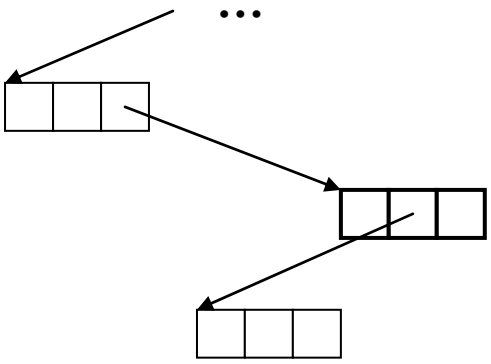

Figura 12.14 *Subarbore numai cu descendent stâng*

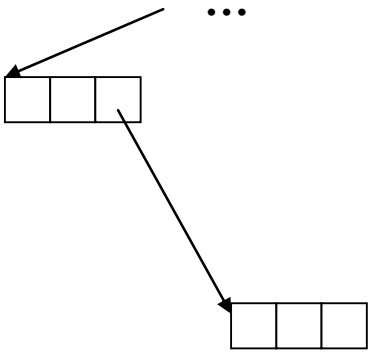

Figura 12.15 *Subarbore după efectuarea ştergerii*

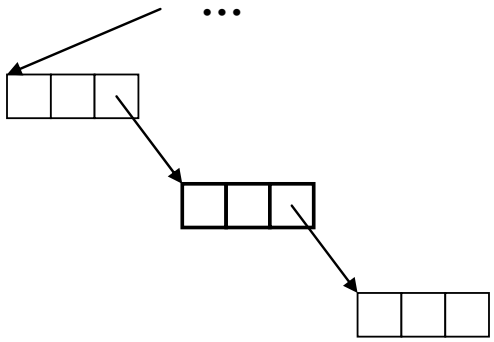

Figura 12.16 *Subarbore numai cu descendent drept*

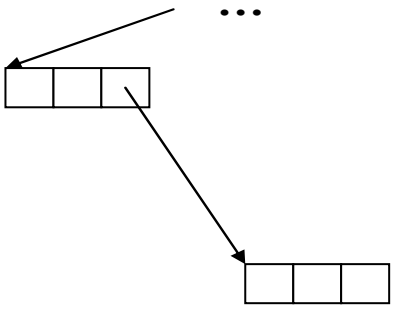

Figura 12.17 *Subarbore după efectuarea ştergerii*

Pentru aceste prime trei cazuri, actualizarea arborelui se face în felul următor: fiul nodului care va fi şters, dacă există, va deveni fiul tatălui acestuia. Actualizarea arborelui este urmată de ştergerea nodului din memorie.

Cazul în care nodul ce se doreşte a fi şters are atât subarbore stâng cât şi subarbore drept necesită o tratare specială. Astfel, mai întâi se localizează fie cel mai din stânga fiu al subarborelui drept fie cel mai din dreapta fiu al subarborelui drept. Cheile acestor noduri sunt valoarea imediat următoare cheii nodului ce se doreşte a fi şters şi valoarea precedentă.

După suprimarea nodului dorit, arborele va trebui să rămână în continuare arbore de căutare ceea ce înseamnă că relaţia de ordine dintre nodurile arborelui va trebui să se păstreze. Pentru aceasta, unul din nodurile prezentate anterior va trebui adus în locul nodului care se doreşte a fi şters după care are loc ştergerea efectivă a nodului dorit.

Determinarea celui mai din dreapta fiu al subarborelui stâng se face parcurgând subarborelui stâng prin vizitarea numai a fiilor din dreapta. Primul nod care nu are subarbore drept este considerat ca fiind cel mai din dreapta nod al subarborelui stâng.

În figura 12.18 se prezintă un fragment de arbore binar de căutare din care dorim să suprimăm nodul evidenţiat care are doi descendenţi. Presupunem ca nodul haşurat este cel mai din dreapta fiu al subarborelui stâng. Acest nod va lua locul nodului care se va şterge. În figura 12.19 este reprezentat fragmentul de arbore după efectuarea operaţiei de ştergere a nodului dorit.

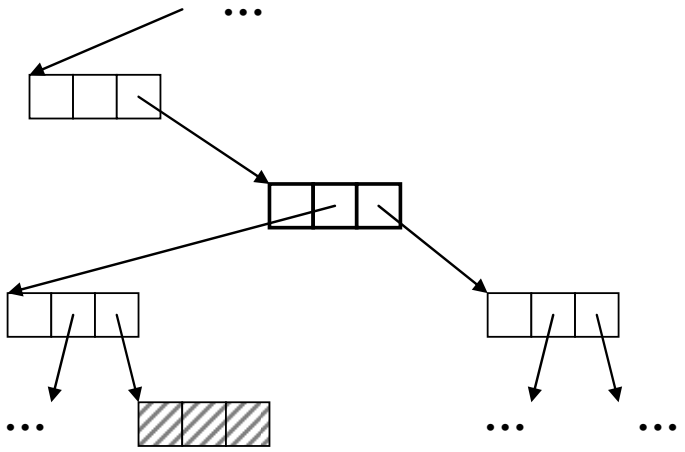

Figura 12.18 *Subarbore cu doi descendenţi*

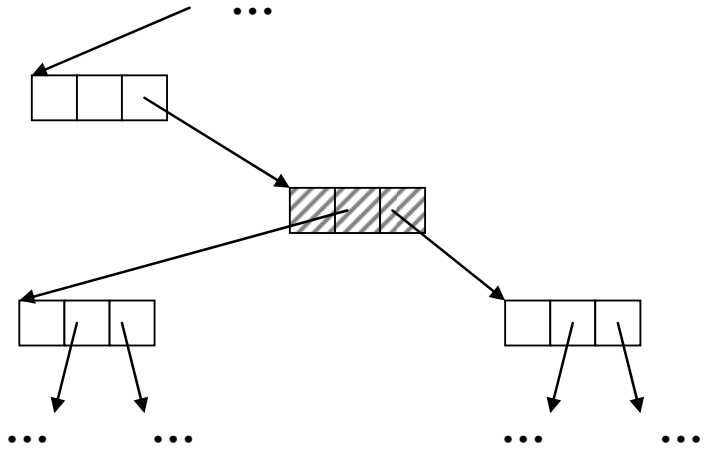

Figura 12.19 *Subarbore după efectuarea ştergerii*

12.4 Aplicaţii care utilizează structura de date de tip arbore binar de căutare

Asupra arborilor binari de căutare pot fi efectuate o serie de operaţii, dintre care o parte sunt specifice tuturor structurilor de date compuse (adăugare element, ştergere elemente) iar altele sunt specifice acestui tip de structură de date. De asemenea se remarcă operaţii la nivel de element (nod), precum şi operaţii care implică întregul arbore.

Programul următor, exemplifică o modalitate de creare a unui arbore binar de căutare, stergere noduri, traversare, numarare noduri şi de tipărire a acestuia:

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <malloc.h>
#include <string.h>
```

```
struct nod
      {
            int info;
            nod *stg,*drt;
      };

class arbbin
      {
                  nod *rad;
                  int stergere(nod *&);
                  void stergere_nod(nod*&,int);
            public:
                  arbbin()
                        {rad=NULL;};
                  ~arbbin()
                        {rad=NULL;};
                  void traversare_srd(nod*);
                  void srd();
                  void traversare_rsd(nod *);
                  void rsd();
                  void traversare_sdr(nod *);
                  void sdr();
                  int sumaFrunze(nod*);
                  int sFrunza();
                  int numara(nod *);
                  int numara_nod();
                  void print(nod *);
                  void tiparire();
                  void salvare();
                  nod *inserare_nod(nod *,int );
                  void operator + (int);
                  void operator - (int);
                  arbbin &operator >(FILE *);
                  arbbin &operator <(FILE *);
                  void inserare_cuv(nod *& ,char*);
                  void insert(char *);
                  nod *operator [] (int);
};

nod *arbbin::inserare_nod(nod *rad,int k)
      {
            if (rad)
                  {
                  if (k<rad->info) rad->stg=inserare_nod(rad->stg,k);
                  else
                  if (k>rad->info) rad->drt=inserare_nod(rad->drt,k);
                        else  printf("\nNodul exista in arbore!");
                  return rad;
                  }
            else
                  {
                        nod *p=new nod;
                        p->stg=NULL;
                        p->drt=NULL;
                        p->info=k;
                        return p;
                  }
      }

void arbbin::operator +(int k)
      {
```

```
            rad=inserare_nod(rad,k);
      }

void arbbin::traversare_srd(nod *rad)
      {
            if (rad)
                  {
                        traversare_srd(rad->stg);
                        printf("  %d",rad->info);
                        traversare_srd(rad->drt);
                  }
      }

void arbbin::srd()
      {
            nod *p;
            p=rad;
            traversare_srd(p);
      }

void arbbin::traversare_rsd(nod *rad)
      {
            if (rad)
                  {
                        printf("  %d",rad->info);
                        traversare_rsd(rad->stg);
                        traversare_rsd(rad->drt);
                  }
      }

void arbbin::rsd()
      {
            nod *p;
            p=rad;
            traversare_rsd(p);
      }

void arbbin::traversare_sdr(nod *rad)
      {
            if (rad)
                  {
                        traversare_sdr(rad->stg);
                        traversare_sdr(rad->drt);
                        printf("  %d",rad->info);
                  }
      }

void arbbin::sdr()
      {
            nod *p;
            p=rad;
            traversare_sdr(p);
      }

void arbbin::print(nod *rad)
      {
            if (rad)
                  {
                        printf("%d",rad->info);
                        if((rad->stg)||(rad->drt))
                              {
```

```
                                    printf("(");
                                    print(rad->stg);
                                    printf(",");
                                    print(rad->drt);
                                    printf(")");
                              }
                  }
            else
                  printf("-");
      }

void arbbin::tiparire()
      {
            nod *p;
            p=rad;
            print(p);
      }

int arbbin::sumaFrunze(nod *rad)
      {
            if (rad)
                  if(!rad->stg&&!rad->drt)
                  return rad->info;
                  else
                  return sumaFrunze(rad->stg)+sumaFrunze(rad->drt);
            else
                  return 0;
      }

int arbbin::sFrunza()
      {
            nod *p;
            p=rad;
            return sumaFrunze(p);
      }

int arbbin::numara(nod *rad)
      {
            if (rad)
                  return 1+numara(rad->stg)+numara(rad->drt);
            else return 0;
      }

int arbbin::numara_nod()
      {
            nod *p;
            int nr;
            p=rad;
            nr=numara(p);
            return nr;
      }

void arbbin::stergere_nod(nod *&rad,int inf)
      {
            nod *aux;
            if (!rad) printf("\nNodul nu exista in arbore!");
            else
                  if (inf<rad->info) stergere_nod(rad->stg,inf);
                  else
                        if (inf>rad->info) stergere_nod(rad->drt,inf);
                        else
```

```
                        {
                              aux=rad;
                              if (!aux->stg)
                              {
                                    rad=aux->stg;
                                    delete(aux);
                              }
                              else
                                    if(!aux->drt)
                                    {
                                          rad=aux->drt;
                                          delete(aux);
                                    }
                                    else
                                        rad->info=stergere(rad->stg);
            }
      }

int arbbin::stergere(nod *&p)
      {
            if (p->stg)
                  return stergere(p->stg);
            else
                  {
                        nod *q=p;
                        int inf=q->info;
                        p=p->stg;
                        delete(q);
                        return inf;
                  }
      }

void arbbin::operator -(int inform)
      {
            nod *nou;
            nou=rad;
            stergere_nod(nou,inform);
      }

nod *arbbin::operator [] (int inf)
      {
            nod *aux;
            aux=rad;
            while(aux&&aux->info!=inf)
                  {
                        if (inf<aux->info)
                              aux=aux->stg;
                        else
                              if (inf>aux->info)
                                    aux=aux->drt;
                  };
            if (aux&&aux->info==inf)
                  cout<<"\nNodul cautat exista in arbore!";
            else
                cout<<"\nNodul cautat nu exista in arbore!";
            return aux;
      }

void meniu()
      {
```

```
            cout<<"\n     1.ADAUGARE NOD DE LA TASTATURA";
            cout<<"\n     2.STERGERE NOD DE LA TASTATURA";
            cout<<"\n     3.CAUTARE NOD IN ARBORE";
            cout<<"\n     4.TRAVERSARI ARBORE-SRD(),RSD(),SDR()";
            cout<<"\n     5.SUMA INFORMATIILOR DIN FRUNZE ";
            cout<<"\n     6.NUMARUL NODURILOR DIN ARBORE";
            cout<<"\n     7.AFISARE ARBORE";
            cout<<"\n     0.TERMINARE";
      }

void main()
{
      arbbin arb;
      int informatie;
      char optiune;
      arb+10;arb+7;arb+15;arb+9;arb+3;arb+8;arb+25;
      meniu();
      optiune='1';
      while (optiune!='0')
            {
                cout<<"\nOptiunea dorita este:";
                  cin>>optiune;
                  if(((optiune>='0')&&(optiune<='9'))||(optiune=='n'))
                  {
                  switch(optiune)
                  {
                  case '1':
                  char op;
                  cout<<"\nNoul nod de introdus:";
                  cin>>informatie;
                  arb+informatie;
                  cout<<"\nMai doriti adaugarea unui alt nod(d/n)?";
                  cin>>op;
                  while (op=='d')
                        {
                        cout<<"\nNoul nod de introdus:";
                        cin>>informatie;
                        arb+informatie;
                        cout<<"\nSe va mai adauga alt nod(d/n)?";
                        cin>>op;
                        }
                  getch();
                  meniu();
                  break;
                  case '2':
                  cout<<"\nNodul care se va sterge:";
                  cin>>informatie;
                  arb-informatie;
                  getch();
                  meniu();
                  break;
                  case '3':
                  int elem;
                  cout<<"\nNodul care va fi cautat:";
                  cin>>elem;
                  arb[elem];
                  getch();
                  meniu();
                  break;
                  case '4':
                  printf("\nArborele traversat in SRD:");
```

```
                  arb.srd();
                  printf("\nArborele traversat in RSD:");
                  arb.rsd();
                  printf("\nArborele traversat in SDR:");
                  arb.sdr();
                  getch();
                  meniu();
                  break;
                  case '5':
                  printf("\nSuma inf frunze:%d", arb.sFrunza());
                  getch();
                  meniu();
                  break;
                  case '6':
                  cout<<"\nNumarul de noduri:"<<arb.numara_nod();
                  getch();
                  meniu();
                  break;
            case '7':
                  printf("\nArborele tiparit in RSD:");
                  arb.tiparire();
                  getch();
                  meniu();
                  break;
            case '0':
                  break;
            }
      }
       else cout<<"\nOptiunea nu exista in meniu!";
      }
}
```