STRUCTURI DE DATE

Adrian CARABINEANU

0

Cuprins

Lista figurilor

Capitolul 1

Algoritmi. Noţiuni generale

Definiţie preliminară. Un *algoritm* este o *procedură de calcul bine definită* care primeşte o mulţime de valori ca *date de intrare* şi produce o mulţime de valori ca *date de ieşire.*

1.1 Exemplu de algoritm. Sortarea prin inserţie

Vom considera mai întâi problema sortării (ordonării) unui şir de n numere întregi $a[0], a[1], ..., a[n-1]$ (ce reprezintă datele de intrare). Şirul ordonat (de exemplu crescător) reprezintă datele de ieşire. Ca procedură de calcul vom considera procedura *sortării prin inserţie* pe care o prezentăm în cele ce urmează: Începând cu al doilea număr din şir, repetăm următorul procedeu : inserăm numărul de pe poziţia j ,reţinut într-o cheie, în subşirul deja ordonat $a[0], ..., a[j-1]$ astfel încât să obţinem subşirul ordonat $a[0], ..., a[j]$. Ne oprim când obţinem subşirul ordonat de n elemente. De exemplu, pornind de la şirul de numere întregi $7, 1, 3, 2, 5$, folosind sortarea prin inserţie obţinem succesiv

$$\begin{array}{c} 7 \mid 1\ 3\ 2\ 5 \\ 1\ 7 \mid 3\ 2\ 5 \\ 1\ 3\ 7 \mid 2\ 5 \\ 1\ 2\ 3\ 7 \mid 5 \\ 1\ 2\ 3\ 5\ 7 \end{array}$$

Linia verticală | separă subşirul ordonat de restul şirului. Prezentăm mai jos programul scris în $C++$ pentru sortarea elementelor unui şir de 10 numere

întregi:

```
# include <iostream.h>
 void main(void)
{// datele de intrare
int a[10];
int i=0;
while(i<n)
{cout<<"a["<<i<<"]="; cin>>*(a+i); i=i+1;}
//procedura de calcul
for(int j=1;j<10;j++)
{int key=a[j];
//insereaza a[j] în sirul sortat a[0,...,j-1]
i=j-1;
while((i>=0)*(a[i]>key))
{a[i+1]=a[i];
i=i-1;}
a[i+1]=key;}
//datele de iesire
for(j=0;j<n;j++)
cout<<"a["<<j<<"]="<<*(a+j)<<";"; }
```

Putem modifica programul $C++$ de sortare a n numere întregi impunând să se citească numărul n, alocând dinamic un spaţiu de n obiecte de tip **int** şi ţinând cont că $*(a+i) = a[i]$:

```
# include <iostream.h>
 void main(void)
{
// datele de intrare
int n;
int i=0;
cout<<"n=";
cin>>n;
 int* a;
a = new int [n];
while(i<n)
{cout<<"a["<<i<<"]="; cin>>*(a+i); i=i+1;}
//procedura de calcul
for(int j=1;j<n;j++)
{int key=*(a+j);
```

```
//insereaza a[j] in sirul sortat a[0,...,j-1]
i=j-1;
while((i>=0)*(*(a+i)>key))
{*(a+i+1)=*(a+i);
i=i-1;}
*(a+i+1)=key; }
//datele de iesire
for(j=0;j<n;j++)
cout<<"a["<<j<<"]="<<*(a+j)<<";";}
```

1.2 Aspecte care apar la rezolvarea unei probleme

Când se cere să se elaboreze un algoritm pentru o problemă dată, care să furnizeze o soluţie (fie şi aproximativă, cu condiţia menţionării acestui lucru), cel puţin teoretic trebuie să fie parcurse următoarele etape:

1) Demonstrarea faptului că este posibilă elaborarea unui algoritm pentru determinarea unei soluţii.

2) Elaborarea unui algoritm (în acest caz etapa anterioară poate deveni inutilă).

3) Demonstrarea corectitudinii.

4) Determinarea timpului de execuţie al algoritmului.

5) Investigarea optimalităţii algoritmului.

Vom prezenta în cele ce urmează, nu neapărat în ordinea indicată mai sus, aceste aspecte.

1.3 Timpul de execuţie a algoritmilor

Un algoritm este elaborat nu doar pentru un set de date de intrare, ci pentru o mulţime de astfel de seturi. De aceea trebuie bine precizată mulţimea (seturilor de date) de intrare. Timpul de execuţie se măsoară în funcţie de lungimea n a setului de date de intrare.

Ideal este să determinăm o formulă matematică pentru $T(n)$ = timpul de executare pentru orice set de date de intrare de lungime n. Din păcate, acest lucru nu este în general posibil. Din această cauză ne vom limita la a evalua *ordinul de mărime* al timpului de execuţie.

Să reluăm procedura de calcul pentr algoritmul de sortare prin inserţie:
//procedura de calcul .....*cost*.....*timp*
for(int j=1;j<n;j++) .....c_1.....n
{**int key=*(a+j);** .....c_2.....$n-1$
//insereaza a[j] in sirul sortat a[1,...,j-1]
i=j-1; .....c_3.....$n-1$
while((i>=0)*(*(a+i)>key)).....c_4..... $\sum_{j=2}^{n} t_j$
{***(a+i+1)=*(a+i);**.....c_5..... $\sum_{j=2}^{n} (t_j - 1)$
i=i-1;}.....c_6..... $\sum_{j=2}^{n} (t_j - 1)$
***(a+i+1)=key; }**.....c_7.....$n-1$

$c_1, ..., c_7$ sunt costurile de timp pentru fiecare instrucţiune. În coloana *timp* punem de câte ori este repetată instrucţiunea; t_j reprezintă numărul de execuţii ale testului **while** (comparaţia) pentru valoarea fixată j. Timpul de execuţie este

$$T(n) = nc_1 + (n-1)(c_2 + c_3 + c_7 - c_5 - c_6) + (c_4 + c_5 + c_6) \sum_{j=2}^{n} t_j. \quad (1.1)$$

Timpul de execuţie poate să depindă de natura datelor de intrare. În cazul în care vectorul de intrare este deja sortat crescător, $t_j = 1$ (deoarece pentru fiecare j, $a[0, ..., j-1]$ este sortat). Timpul de execuţie în acest caz (cel mai favorabil) este

$$T(n) = n(c_1 + c_2 + c_3 + c_4 + c_7) - (c_2 + c_3 + c_4 + c_7). \quad (1.2)$$

Dacă vectorul este sortat în sens invers (în ordine descrescătoare) avem cazul cel mai defavorabil. Fiecare element $a[j]$ este comparat cu fiecare element din $a[0, ..., j-1]$ şi astfel $t_j = j$ pentru $j = 2, 3, ..., n$. Cum

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1, \ \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2},$$

deducem că

$$T(n) =$$

$$\left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right) n - (c_2 + c_3 + c_4 + c_7). \quad (1.3)$$

Timpul de execuţie este are ordinul de mărime $O(n^2)$. În general spunem că timpul de execuţie este de ordinul $O(f(n))$ dacă

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = l,\ l \text{ finită}.$$

Când $f(n) = n^k, k \in \mathbf{N}^*$ spunem că algoritmul este *polinomial*. Specificăm faptul că un algoritm este considerat *acceptabil* dacă este polinomial.

1.4 Corectitudinea algoritmilor

În demonstrarea corectitudinii algoritmilor există două aspecte importante

- *Corectitudinea parţială*: presupunând că algoritmul se termină într-un număr finit de paşi, trebuie demonstrat că rezultatul este corect.

- *Terminarea programului*: trebuie demonstrat că algoritmul se încheie în timp finit.

Evident, condiţiile enumerate mai sus trebuie îndeplinite pentru orice set de date de intrare admis de problemă.

Modul tipic de lucru constă în introducerea în anumite locuri din program a unor *invarianţi*, care reprezintă relaţii ce sunt îndeplinite la orice trecere a programului prin acele locuri. De exemplu în cazul sortării prin inserţie, invarianţii sunt următorii:

- *după fiecare executare a ciclului* **while** *(corespunzătoare lui $i = j - 1$) elementele cu indici mai mici sau egali cu j au fost sortate parţial.*

Ciclul **for** se termină odată cu ultima executare a ciclului **while**, când $j = n$ şi când toate elementele sunt sortate.

1.5 Optimalitatea algoritmilor

Să presupunem că pentru o anumită problemă am elaborat un algoritm şi am putut calcula timpul său de execuţie $T(n)$. Este natural să ne punem problema dacă algoritmul este executat în timpul *cel mai scurt* posibil sau există un alt algoritm cu timpul de execuţie mai mic. Spunem că un algoritm este *optim* dacă raportul dintre timpul său de execuţie şi timpul de execuţie al oricărui alt algoritm care rezolvă aceeaşi problema are ordinul de mărime $O(1)$. Problema demonstrării optimalităţii este deosebit de dificilă, mai ales

datorită faptului că trebuie să considerăm toţi algoritmii posibili şi să arătăm că ei au un timp de execuţie superior celui al algoritmului optim.

În cazul algoritmilor de sortare ne propunem să găsim o margine inferioară a timpilor de execuţie. Vom face mai întâi observaţia că numărul total de instrucţiuni executate şi numărul de comparaţii au acelaşi ordin de mărime. Mulţimea comparaţiilor poate fi vizualizată cu ajutorul *arborilor de decizie*. Într-un arbore de decizie fiecare nod este etichetat prin $a_i : a_j$ pentru i şi j din intervalul $1 \leq i, j \leq n$ unde n este să numărul de elemente din secvenţa de intrare. Fiecare frunză este etichetată cu o permutare $(\sigma(1), ..., \sigma(n))$. Execuţia algoritmului de sortare corespunde trasării unui drum elementar de la rădăcina arborelui de sortare la o frunză. La fiecare nod intern este făcută o comparaţie între a_i şi a_j. Subarborele stâng dictează comparaţiile următoare pentru $a_i \leq a_j$ iar subarborele drept dictează comparaţiile următoare pentru $a_i > a_j$. Când ajungem la o frunză algoritmul a stabilit ordonarea $a_{\sigma(1)} \leq a_{\sigma(2)} \leq ... \leq a_{\sigma(n)}$. Pentru ca algoritmul să ordoneze adecvat, fiecare din cele $n!$ permutări de n elemente trebuie să apară într-o frunză a arborelui de decizie. În figura (1.1) prezentăm arborele de decizie corespunzător sortării unei mulţimi $\{a_1, a_2, a_3\} = \{1, 2, 3\}$. În funcţie de datele de intrare, comparaţiile efectuate de program reprezintă un drum elementar în arborele de decizie ce uneşte rădăcina arborelui cu o frunză. Numărul de noduri (comparaţii) dintr-un astfel de drum elementar este egal cu cel mult h, înălţimea arborelui.

Teorema 1. *Orice arbore de decizie care sortează n elemente are înălţimea de ordinul $O(n \ \ln n)$.*

Demonstrţie. Întrucât există $n!$ permutări ale celor n elemente, arborele trebuie să aibă $n!$ frunze. Un arbore binar de înălţime h are cel mult 2^h frunze. Deci

$$n \leq 2^h,$$

$$h \geq \log_2 n! = \ln n! \ \log_2 e.$$

Plecând de la inegalitatea

$$n! > \left(\frac{n}{e}\right)^n,$$

obţinem

$$h \geq n(\ln n - 1)\log_2 e,$$

adică

$$h = O(n \ln n).$$

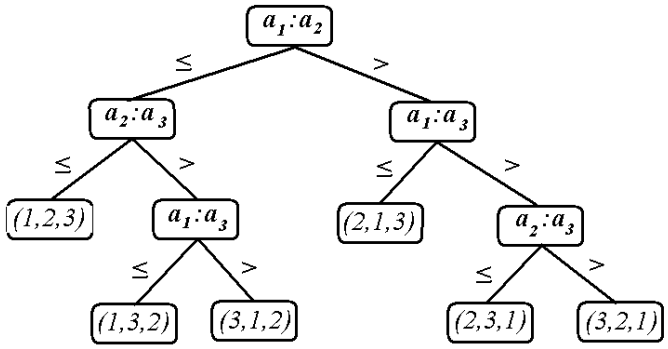


Figura 1.1: Arbore de decizie

Ţinând cont de teorema mai sus enunţată, este de presupus că un algoritm de sortare optim are timpul de execuţie de ordinul $O(n \ \ln n)$. Algoritmul de sortare prin inserţie, având timpul de execuţie de ordinul $O(n^2)$, are toate şansele să nu fie optimal. Vom da în cele ce urmează un exemplu de algoritm de sortare optim şi anume algoritmul de *sortare prin interclasare.*

Pentru a avea o imagine intuitivă a procedurii de interclasare, să considerăm că un pachet cu n cărţi de joc este împărţit în alte 2 pachete aşezate pe masă cu faţa în sus. Fiecare din cele 2 pachete este sortat, cartea cu valoarea cea mai mică fiind deasupra. Dorim să amestecăm cele două subpachete într-un singur pachet sortat, care să rămână pe masă cu faţa în jos. Pasul principal este acela de a selecta cartea cu valoarea cea mai mică dintre cele 2 aflate deasupra pachetelor (fapt care va face ca o nouă carte să fie deasupra pachetului respectiv) şi de a o pune cu faţa în jos pe locul în care se va forma pachetul sortat final. Repetăm procedeul până când unul din pachete este epuizat. În această fază este suficient să luăm pachetul rămas şi să-l punem peste pachetul deja sortat întorcând toate cărţile cu faţa în jos. Din punct de vedere al timpului de execuţie, deoarece avem de făcut cel mult $n/2$ comparaţii, timpul de execuţie pentru procedura de interclasare este de ordinul $O(n)$.

Procedura de interclasare este utilizată ca subrutină pentru algoritmul de

sortare prin interclasare care face apel la tehnica *divide şi stăpâneşte,* după cum urmează:

Divide: Împarte şirul de n elemente ce urmează a fi sortat în două subşiruri de câte $n/2$ elemente.

Stăpâneşte: Sortează recursiv cele două subşiruri utilizând sortarea prin interclasare.

Combină: Interclasează cele două subşiruri sortate pentru a produce rezultatul final.

Descompunerea şirului în alte două şiruri ce urmează a fi sortate are loc până când avem de sortat şiruri cu unul sau două componente. Prezentăm mai jos programul scris în $C++$. În program, funcţia **sort** sortează un vector cu maximum 2 elemente, funcţia **intecl** interclasează 2 şiruri sortate iar **dist** implementează strategia *divide şi stăpâneşte* a metodei studiate.

```
#include<iostream.h>
/**************************/
void sort(int p,int q, int n, int *a) {
int m;
 if(*(a+p)>*(a+q))
 {m=*(a+p); *(a+p)=*(a+q); *(a+q)=m;}}
/*********************************/
void intecl(int p, int q, int m, int n, int *a){
int *b,i,j,k;
i=p;j=m+1;k=1;
b=new int[n];
while(i<=m && j<=q)
if(*(a+i)<=*(a+j))
{*(b+k)=*(a+i);i=i+1;k=k+1;}
else
{*(b+k)=*(a+j);j=j+1;k=k+1;}
if(i<=m)
 for (j=i;j<=m;j++)
 {*(b+k)= *(a+j); k=k+1;}
else for(i=j;i<=q;i++)
 {*(b+k)=*(a+i); k=k+1;}
k=1;
 for(i=p;i<=q;i++) {*(a+i)=*(b+k); k=k+1;}}
/************************************/
void dist(int p, int q, int n, int *a){
```

```
 int m;
 if((q-p)<=1) sort(p,q,n,a);
else
{m=(p+q)/2; dist(p,m,n,a); dist(m+1,q,n,a);intecl(p,q,m,n,a);
}}
/************************************/
void main(void){
 int n; *a,i;
 cout<<"n="; cin>>n;
 a=new int[n];
 for(i=1;i<=n;i++)
{cout<<"a["<<i<<"]="; cin>>*(a+i-1);}
dist(0,n-1,n,a);
for(i=1;i<=n;i++)
cout<<"a["<<i-1<<"]="<<a[i-1];}
```

În continuare să calculăm $T(n)$, numărul aproximativ de comparaţii efectuat de algoritm. Succesiv, problema se descompune în alte două probleme, fiecare referindu-se la $n/2$ elemente. Urmează interclasarea elementelor, care necesită un număr de $n/2$ comparaţii. Avem deci

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{2},\ T(0) = 0.$$

Pentru început să considerăm $n = 2^k,\ k \in \mathbf{N}$. Rezultă (efectuând implicit un raţionament prin inducţie):

$$T\left(2^k\right) = 2T\left(2^{k-1}\right) + 2^{k-1} = 2\left(2T\left(2^{k-2}\right) + 2^{k-2}\right) + 2^{k-1} = ...$$

$$= 2^p T\left(2^{k-p}\right) + p2^{k-1} = 2^p\left(2T\left(2^{k-p-1}\right) + 2^{k-p-1}\right) + p2^{k-1} =$$

$$= 2^{p+1}T\left(2^{k-p-1}\right) + (p+1)\,2^{k-1} = T(0) + k2^{k-1} = k2^{k-1}.$$

Pentru un număr natural oarecare n, fie k astfel încât să avem $2^k \leq n < 2^{k+1}$. Rezultă

$$k2^{k-1} = T\left(2^k\right) \leq T(n) < T\left(2^{k+1}\right) = (k+1)\,2^k. \tag{1.4}$$

Cum $k = O(\ln n)$, din (1.4) rezultă că

$$T = O(n \ln n),$$

deci algoritmul de sortare prin interclasare este optim.

1.6 Existenţa algoritmilor

Problema existenţei algoritmilor a stat în atenţia matematicienilor încă înainte de apariţia calculatoarelor. Un rol deosebit în această teorie l-a jucat matematicianul englez Alan Turing (1912-1954), considerat părintele inteligenţei artificiale.

Numim *problemă nedecidabilă* o problemă pentru care nu poate fi elaborat un algoritm. Definirea matematică a noţiunii de algoritm a permis detectarea de probleme nedecidabile. Câteva aspecte legate de decidabilitate sunt următoarele:

- *Problema opririi programelor:* pentru orice program şi orice valori de intrare să se decidă dacă programul se termină.

- *Problema echivalenţelor programelor*: să se decidă pentru oricare două programe dacă sunt echivalente (produc aceeaşi ieşire pentru aceleaşi date de intrare).

Capitolul 2

Tipuri de structuri de date

2.1 Generalităţi

Structurile de date reprezintă *modalităţi în care datele sunt dispuse în memoria calculatorului sau sunt păstrate pe discul magnetic*. Structurilor de date sunt utilizate în diferite circumstanţe ca de exemplu:

- Memorarea unor date din realitate;
- Instrumente ale programatorilor;
- Modelarea unor situaţii din lumea reală.

  Cele mai des utilizate structuri de date sunt tablourile, listele, stivele, cozile, arborii, tabelele de dispersie si grafurile.

2.2 Liste

O *listă liniară* este o structură de date omogenă, secvenţială formată din *elemente* ale listei. Un element (numit uneori şi *nod*) din listă conţine o *informaţie specifică* şi eventual o *informaţie de legătură*. De exemplu, în lista echipelor de fotbal înscrise într-un campionat, un element (ce reprezintă o echipă) poate conţine următoarele informaţiie specifice: nume, număr de puncte, număr de goluri înscrise şi număr de goluri primite. Fiecare din aceste informaţii poate reprezenta o *cheie* care poate fi utilizată pentru comparaţii şi inspecţii. De exemplu luând drept cheie numărul de puncte şi golaverajul se poate face clasificarea echipelor.

Poziţia elementelor din listă defineşte ordinea elementelor (primul element, al doilea, etc). Conţinutul listei se poate schimba prin:

- *adăugarea* de noi elemente la sfârşitul listei;
- *inserarea* de noi elemente în orice loc din listă;
- *ştergerea* de elemente din orice poziţie a listei;
- *modificarea* unui element dintr-o poziţie dată.

Printre operaţiile care schimbă structura listei vom considera şi *iniţializarea* unei liste ca o *listă vidă*.

Alte operaţii sunt operaţiile de *caracterizare*. Ele nu modifică structura listelor dar furnizează informaţii despre ele. Dintre operaţiile de caracterizare vom menţiona în cele ce urmează:

- *localizarea* elementului din listă care satisface un anumit criteriu;
- *determinarea lungimii listei.*

Pot fi luate în considerare şi operaţii mai complexe, ca de exemplu:

- *separarea* unei liste în două sau mai multe subliste în funcţie de îndeplinirea unor condiţii;
- *selecţia* elementelor dintr-o listă, care îndeplinesc unul sau mai multe criterii, *într-o listă nouă;*
- *crearea unei liste ordonate* după valorile creascătoare sau descrescătoare ale unei chei;
- *combinarea* a două sau mai multor liste prin *concatenare (alipire)* sau *interclasare*.

Spaţiul din memorie ocupat de listă poate fi alocat în două moduri: prin *alocare secvenţială* sau prin *alocare înlănţuită*.

2.2.1 Liste alocate secvenţial

În acest caz nodurile ocupă poziţii succesive în memorie. Acest tip de alocare este întâlnit când se lucrează cu tablouri (vectori). Avantajul alocării secvenţiale este dat de faptul că accesul la oricare din nodurile listei este direct. Dezavantajul constă în faptul că operaţiile de adăugare, eliminare sau schimbare de poziţie a unui nod necesită un efort mare de calcul după cum s-a văzut în algoritmii de sortare prezentaţi mai înainte.

2.2.2 Liste alocate înlănţuit

Există două feluri de alocare înlănţuită: *alocare simplu înlănţuită* şi *alocare dublu înlănţuită* (figura 2.1). Alocarea înlănţuită poate fi efectuată static

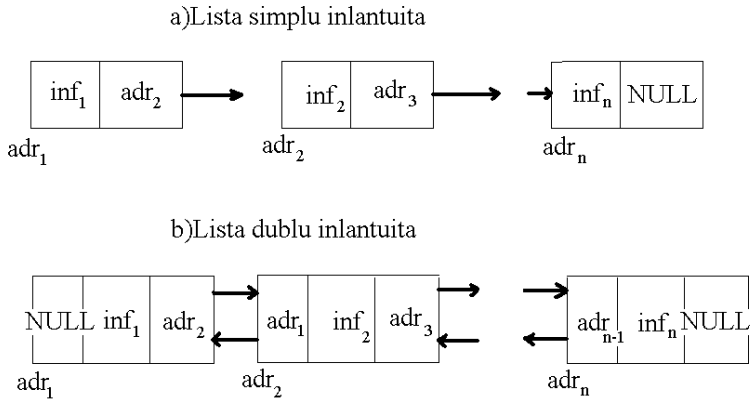


Figura 2.1: Liste simplu şi dublu înlănţuite

(utilizând vectori) sau dinamic. În acest din urmă caz (de care ne vom ocupa în cele ce urmează), se utilizează o zonă de memorie numită *HEAP* (movilă, grămadă). În $C++$ variabilele păstrate în HEAP (cum ar fi de exemplu nodurile listei), se alocă atunci când doreşte programatorul, cu ajutorul operatorului **new**, iar zona se eliberează, tot la dorinţa acestuia prin operatorul **delete**. În cazul alocării dinamice, nodul unei liste este o structură care conţine alături de informaţie şi adresa nodului următor (pentru liste simplu înlănţuite) sau adresele nodului următor şi a celui precedent în cazul listelor dublu înlănţuite. După cum se va vedea din exemplele ce urmează, principala problemă în cazul operaţiilor cu liste o reprezintă redefinirea legăturilor (adreselor) când se şterge sau se inserează un element.

Liste simplu înlănţuite

Mai jos prezentăm proceduri (funcţii) de **creare** (prin inserţie repetată) şi **parcurgere** a listelor precum şi procedurile de **ştergere** (eliminare) a unui nod şi de **inserare** a unui nod imediat după alt nod ce conţine o anumită informaţie.

```
#include<iostream.h>
//introducem structura de nod al unei liste
```

```
struct nod {int inf; nod *adr ;};
//declararea funcţiilor de creare, parcurgere, eliminare şi inserare
nod *creare(void);
void parcurge(nod *prim);
 nod *elimina(nod *prim, int info);
nod *inserare_dupa(nod* prim, int info, int info1);
//functia principală
/*****************************************/
void main(void) {
int info, info1; nod *cap;
 cap=creare();
 parcurge(cap);
 cout<<"Spuneti ce nod se elimina";
 cin>>info;
cap= elimina(cap, info);
 parcurge(cap);
cout<<"Spuneti ce valoare se insereaza si dupa cine"<<endl;
cin>>info1;
cin>>info;
inserare_dupa(cap,info,info1);
parcurge(cap);}
//functia de creare a listei
/*****************************************/
nod *creare(void)
{ nod *prim,*p,*q; int inf; char a;
//crearea primului element al listei
 prim=new nod;
cout<<" Introduceti prim->inf"<<endl;
 cin>>inf;
 prim->inf=inf;
prim->adr=NULL;
q=prim;
/*pana cand se decide ca operatia este gata, se creaza elementele urmatoare*/
cout<<"Gata?[d/n]"<<endl;
cin>>a;
while (a!='d') {
p=new nod;
```

```
 cout<<"p->inf"<<endl;
cin>>inf;
/*se creaza un nou nod*/
 p->inf=inf ;
p->adr=NULL;
/*se stabileste legatura dintre nodul anterior si nodul nou creat*/
 q->adr=p;
 q=p;
 cout<<"Gata?[d/n]"<<endl;
  cin>>a;}
return prim;}
/*****************************************/
/*urmeaza procedura de parcurgere a listei*/
void parcurge(nod *cap)
{nod *q;
 if(!cap) {cout<<"Lista vida"; return;}
 cout<<"Urmeaza lista"<<endl;
 for(q=cap;q;q=q->adr)
 cout<<q->inf<<" ";}
/***************************/
/*urmeaza procedura de eliminare a nodului ce contine o anumita informatie*/
nod *elimina(nod* prim, int info)
{ nod *q,*r;
/*se studiaza mai intai cazul cand trebuie eliminat primul nod*/
 while((*prim).inf==info)
{q=(*prim).adr; delete prim; prim=q;}
/*se studiaza cazul cand informatia cautata nu este in primul nod*/
q=prim;
 while(q->adr)
{ if(q->adr->inf==info)
/*in cazul in care a fost gasit un nod cu informatia ceruta, acesta se elimina iar nodul anterior este legat de nodul ce urmeaza*/
{ r=q->adr; q->adr=r->adr; delete r;}
 else q=q->adr;}
 return prim;}
/***********************************/
```

```
/*functia de inserare a unui nod ce contine o anumita informatie dupa
un nod ce contine o informatie data*/
nod *inserare_dupa(nod*prim, int info, int info1)
{nod *c, *d;
c=prim;
while(c->adr&&(c->inf!=info)) c=c->adr;
d=new nod; d->inf=info1;
if(!c->adr)
/*daca nici-un nod nu contine informatia data, noul nod se insereaza
dupa ultimul element al listei*/
{d->adr=NULL; c->adr=d;}
/* daca au fost depistate noduri ce contin informatia data noul element
se insereaza dupa primul astfel de nod*/
else {d->adr=c->adr; c->adr=d;}
return prim;}
```

Ca o ilustrare a utilităţii listelor liniare simplu inlănţuite vom prezenta în cele ce urmează o procedură de adunare şi înmulţire a două polinoame de o variabilă. Un polinom va fi reprezentat printr-o listă, un nod al listei corespunzând unui monom. Informaţia din nodul listei (monom) va conţine gradul monomului şi coeficientul. Pentru a efectua produsul polinoamelor vom crea cu funcţia **prod** o nouă lista, în fiecare nod al noii liste fiind produsul a două monoame (fiecare dintre aceste monoame aparţinând altui polinom). Cu o funcţie numită **canonic,** dacă două noduri (monoame) din lista nou creata (lista produs) au acelaşi grad, coeficientul unuia din monoame este înlocuit cu suma coeficienţilor iar coeficientul celuilalt cu 0. Cu funcţia **elimina** ştergem apoi nodurile (monoamele) care conţin coeficientul 0.

Pentru a aduna două polinoame, vom efectual mai întâi concatenarea lor (ultimul element din lista ce reprezintă primul polinom va fi legată de primul element din lista ce reprezintă al doilea element). Aplicând apoi funcţiile **canonic** şi **elimina** obţinem polinomul sumă. Prezentăm programul mai jos:

```
#include<iostream.h>

struct nod{int grad; int coef; nod *adr ;};
/**********************/
//declararea functiilor utilizate
nod *creare(void);
void parcurge(nod *prim);
void canonic (nod *prim);
```

```
nod* concatenare(nod *prim1, nod*prim2);
nod *elimina(nod* prim, int info);
nod* prod(nod *prim1, nod* prim2);
/**********************/
//functia principala
void main(void){
nod *cap, *cap1,*suma,*produs,*prim;
cap=creare();
cout<<"Listeaza primul polinom"<<endl;
parcurge(cap);
cap1=creare();
cout<<"Listeaza al doilea polinom"<<endl;
parcurge(cap1);
cout<<"Produsul polinoamelor"<<endl;
cout<<"*****************";
produs=prod(cap,cap1);
canonic(produs);
produs=elimina(produs,0);
parcurge(produs);
prim=concatenare(cap,cap1);
cout<<"Polinoamele concatenate"<<endl;
parcurge(prim);
canonic(prim);
cout<<"Suma polinoamelor"<<endl;
suma=elimina(prim,0);
parcurge(suma);}
/**********************/
//functia de creare a unui polinom
nod *creare(void){
nod *prim,*p,*q;int coef,grad;char a;
prim=new nod;
cout<<" Introduceti prim->grad"<<endl;
cin>>grad;
prim->grad=grad;
cout<<" Introduceti prim->coef"<<endl;
cin>>coef;
prim->coef=coef;
prim->adr=NULL;
```

```
q=prim;
cout<<"Gata?[d/n]"<<endl;
cin>>a;
while (a!='d'){
p=new nod;
cout<<"p->grad"<<endl;
cin>>grad;
p->grad=grad;
cout<<"p->coef<<endl";
cin>>coef;
p->coef=coef;
p->adr=NULL;
q->adr=p;
q=p;
cout<<"Gata?[d/n]"<<endl;
cin>>a;}
return prim;}
/*************************/
//functia de parcurgere a unui polinom
void parcurge(nod *cap){
nod *q;
if(!cap)cout<<"Polinom vid";
return;
cout<<"Urmeaza polinomul"<<endl;
for(q=cap;q;q=q->adr)
cout<<"+("<<(*q).coef<<")"<<"X"<<(*q).grad;}
/*******************************/
//functia care efectueaza produsul a doua polinoame
nod* prod(nod* prim1, nod* prim2)
{nod *p,*q,*r,*cap,*s;
cap=new nod;
cap->grad=1;cap->coef=0;cap->adr=NULL;
s=cap;
for(p=prim1;p;p=p->adr)
for(q=prim2;q;q=q->adr)
{r=new nod;r->coef=p->coef*q->coef;
r->grad=p->grad+q->grad;r->adr=NULL;
s->adr=r;s=r;}
```

```
return cap;}
/************************/
/*functia care aduna coeficientii a doua monoame de acelasi grad si atribuie valoarea 0 coeficientului unuia din monoamele adunate*/
void canonic(nod *prim){
nod *p,*q;
for (p=prim;p;p=p->adr)
{q=p->adr;
while(q) if(p->grad==q->grad)
{p->coef+=q->coef;q->coef=0;q=q->adr;}}return;}
/*****************************/
/*functia care elimina monoamele ale caror coeficienti au o valoare data*/
nod *elimina(nod* prim, int info)
{nod *q,*r;
if((*prim).coef==info)
q=(*prim).adr; delete prim; prim=q;
q=prim;
while(q->adr)
if(q->adr->coef==info)
{r=q->adr; q->adr=r->adr; delete r;}
else q=q->adr;
return prim;}
/*****************************/
//functia de concatenare a doua monoame
nod* concatenare(nod *prim1, nod*prim2)
nod *q,*r;
for(q=prim1;q;q=q->adr) r=q;
r->adr=prim2;
return prim1;
```

Liste dublu înlănţuite

În continuare vom prezenta un program în cadrul căruia se creează o listă dublu înlănţuită, se parcurge direct şi invers şi se elimină nodurile ce conţin o informaţie dată. Structura **nod** va conţie trei câmpuri: unul (**inf**) este informaţia nodului, al doilea (**urm**) este pointerul care indică adresa nodului următor iar al treilea (**ant**) este pointerul care indica adresa nodului anterior.

Vom introduce un nou tip de variabila, numit **lista** constând dintr-o structură formată din două variabile de tip **nod** (cele două noduri reprezentând primul şi ultimul element al listei dublu înlănţuite). Funcţia **creare** permite crearea unei liste dublu înlănţuite. Funcţia **parcurg_d**, al cărui argument este primul element al listei, permite parcurgerea directă (plecând de la primul element şi ajungând la ultimul) a listei. Funcţia **parcurg_i** al cărei argument este ultimul nod al listei realizează parcurgerea listei în sens invers. Funcţia **elimin_d** ale cărei argumente sunt o variabilă de tip **lista** şi o variabilă de tip **int**, parcurge direct lista dublă şi elimină nodurile ce conţin argumentul de tip întreg de câte ori le întâlneşte. Proprietatea pe care o au listele duble de a putea fi parcurse în ambele sensuri este folosită de funcţia **sortin** pentru a sorta prin inserţie, după valorile din câmpul **inf**, elementele listei, valoarea întoarsă de funcţie fiind lista sortată (mai precis primul şi ultimul element al listei sortate).

```
#include<iostream.h>
struct nod{int inf; nod *urm; nod *ant;};
typedef struct{nod *p;nod *u;} lista;
//declararea functiilor utilizate
lista creare(void);
void parcurg_i(nod*prim);
void parcurg_d(nod*prim);
lista elimin_d(lista list,int info);
nod*sortin(lista list);
/*******************************/
//functia principala
void main(void)
{int info;
nod *prim;lista list;
list= creare();
parcurg_d(list.p);
parcurg_i(list.u);
cout<<"Spuneti ce nod se elimina"<<endl;
cout<<"list.p->inf";
cin>>info;
list=elimin_d(list,info);
parcurg_d(list.p);
parcurg_i(list.u);
prim=sortin(list);
```

```
cout<<"Urmeaza lista sortata"<<endl;
parcurg_d(prim);}
/*********************************/
//functia de creare
lista creare(void)
{nod *prim,*p,*q;int inf;char a; lista val;
prim=new nod;
cout<<" Introduceti prim->inf"<<endl;
cin>>inf;
prim->inf=inf;prim->urm=NULL;prim->ant=NULL;
q=prim;cout<<"Gata?[d/n]"<<endl;
cin>>a;
while (a!='d')
{p=new nod;
cout<<"p->inf"<<endl;
cin>>inf;
p->inf=inf ;p->urm=NULL;p->ant=q;
q->urm=p;
q=p;
cout<<"Gata?[d/n]"<<endl;
cin>>a;}
val.p=prim;
if(!prim->urm) val.u=prim;
else val.u=p;
return val;}
/*****************************/
//functia de parcurgere directa
void parcurg_d(nod *prim)
 {nod *q;
 if(!prim){cout<<"Lista vida";return;}
 cout<<"Urmeaza lista directa"<<endl;
 for(q=prim;q;q=q->urm)
 cout<<q->inf<<" ";}
/*****************************/
//functia de parcurgere inversa
void parcurg_i(nod *ultim)
 {nod *q;
 if(!ultim){cout<<"Lista vida";return;}
```

```
 cout<<"Urmeaza lista inversa<<endl";
 for(q=ultim;q;q=q->ant)
 cout<<q->inf<<" ";}
/*********************************/
/*functia de eliminare a nodurilor ce contin o informatie data*/
lista elimin_d(lista list, int info){
 nod *q,*r,*p;
 while(list.p->inf==info)
 if (list.p==list.u) {list.p=NULL;list.u=NULL;return list;}
 else
 {q=list.p->urm;q->ant=NULL; delete list.p; list.p=q;}
 q=list.p;
 while(q->urm)
 {if(q->urm==list.u)
 {if (q->urm->inf==info)
 {r=q->urm;
 list.u=q;q->urm=NULL;delete r;}
 return list;}
 if(q->urm->inf==info)
 {p=q;r=q->urm; q->urm=r->urm;
 q->urm->ant=p; delete r;}
 else q=q->urm;}
 list.u=q;return list;}
/*****************************/
//functia de sortare prin insertie
nod* sortin(lista list){
nod*s,*r,*p=list.p,*q;int m;
if(!list.p) cout<<"Lista vida"<<endl;
else
for(q=p->urm;q;q=q->urm)
{for(s=q->ant;s;s=s->ant)
{if (!s->ant) break;
if(q->inf>=s->ant->inf) break;}
if(q->inf<s->inf)
{m=q->inf;
for(r=q;!(r==s);r=r->ant)
 r->inf=r->ant->inf;s->inf=m;}}
return list.p;}
```

2.3 Stive

Stiva este o listă pentru care singurele operaţii permise sunt:

- adăugarea unui element în stivă;

- eliminarea, vizitarea sau modificarea ultimului element introdus în stivă.

Stiva funcţionează după principiul *ultimul intrat primul ieşit - Last In First Out (LIFO).*

În cazul alocării dinamice (de care ne vom ocupa în cele ce urmează), elementele (nodurile) stivei sunt structuri în componenţa cărora intră şi adresa nodului anterior.

În programul de mai jos dăm funcţiile **push** de adăugare în stivă a unei înregistrări şi **pop** de extragere. Cu ajutorul lor construim o stivă folosind funcţia **creare**.

```
#include<iostream.h>
struct nod{int inf; nod*ant;};
nod*stiva;
/*********************************/
//functia de adaugare in stiva a unei noi inregistrari
nod*push(nod *stiva, int info)
{nod *r;
r=new nod;
r->inf=info;
if(!stiva)r->ant=NULL;
else r->ant=stiva;
stiva=r;return stiva;}
/*********************************/
//functia de extragere din stiva a ultimului nod
nod *pop(nod*stiva)
{nod *r;
if(!stiva)
cout<<"Stiva vida "<<endl;
else
{r=stiva;stiva=stiva->ant;delete r;}
return stiva;}
/*********************************/
/*functia de parcurgere a stivei in ordine inversa (de la ultimul nod intrat
la primul)*/
void parcurge(nod*stiva)
```

```
{nod*r=stiva;
if(!stiva) cout<<"Stiva vida"<<endl;
else {cout<<"Urmeaza stiva"<<endl;
while(r){cout<<r->inf<<" ";r=r->ant;}}
return;}
/**********************************/
/*functia de creare a stivei prin adaugari si eliminari*/
nod* creare()
{char a;int info;nod*stiva;
cout<<"Primul nod"<<endl;
cout<<"stiva->inf";
cin >>info;
stiva=new nod;
stiva->inf=info;
stiva->ant=NULL;
a='a';
while(!(a=='t')){
cout<<"Urmatoarea operatie[a/e/t]"<<endl;
/* t=termina; a=adauga nod; e=elimina nod;*/
cin>>a;
switch(a)
{case 't': break;
case 'a':cout<<"Stiva->inf"<<endl;
cin>>info;
stiva=push(stiva,info);break;
default:stiva=pop(stiva);break;}}
return stiva;}
/*****************************/
//functia principala
void main()
{stiva=creare();
parcurge(stiva);}
```

2.4 Liste de tip "coadă"

O *coadă* este o listă pentru care toate inserările sunt făcute la unul din capete iar toate ştergerile (consultările, modificările) sunt efectuate la celălalt capăt.

Coada funcţionează pe principiul *primul intrat primul iesit - First In First Out (FIFO).*

În cazul alocării dinamice, elementele (nodurile) cozii sunt structuri în componenţa cărora intră şi adresa nodului următor.

În programul de mai jos dăm funcţiile **pune** de adăugare în coadă a unei înregistrări şi **scoate** de extragere. Cu ajutorul lor construim o coadă folosind funcţia **creare**. Vom reprezenta coada printr-o structură **coada** care conţine primul şi ultimul nod al cozii.

```
#include<iostream.h>
//urmeaza structura de tip nod
struct nod{int inf;nod*urm;};
//urmeaza structura de tip coada
typedef struct{nod *p;nod *u;} coada;
coada c;
/*********************************/
//functia de adaugare in coada
coada pune(coada c,int n)
{nod *r;
r=new nod;r->inf=n;r->urm=NULL;
if(!c.p)
{c.p=r;c.u=r;}
else{c.u->urm=r;c.u=r;} return c;}
/*********************************/
//functia de extragere din coada
coada scoate(coada c)
{nod *r;
if(!c.p) cout<<"Coada vida"<<endl;
else
{cout <<"Am scos "<<c.p->inf<<endl;
r=c.p;c.p=c.p->urm;delete r;return c;
/*********************************/
//functia de listare (parcurgere) a cozii
void parcurge(coada c)
{nod *r=c.p;
cout<<"Urmeaza coada"<<endl;
while(r)
{cout<<r->inf<<" ";
r=r->urm;}
```

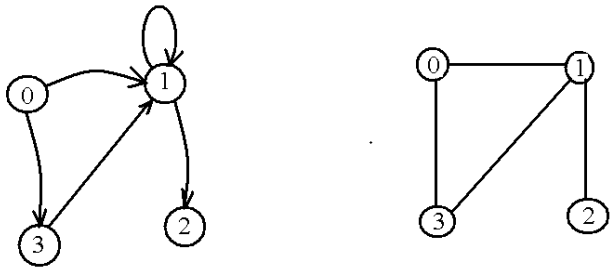
```
cout <<endl;}
/*********************************/
//functia de creare a cozii
coada creare()
{char a;int info;coada c;
cout<<"Primul nod"<<endl;
cout<<"c.p->inf"<<endl;
cin >>info;
c.p=new nod;
c.p->inf=info;
c.p->urm=NULL;
c.u=c.p;
a='a';
while((a=='s')+(a=='a')){
cout<<"Urmatoarea operatie";
cout<<"[a=pune nod/s=scoate/t=termina]"<<endl;
cin>>a;
switch(a)
{case 's': c=scoate(c);break;
case 'a':cout<<"c.p->inf"<<endl;cin>>info;
c=pune(c,info);break;
default:break;}}
return c;}
/*********************************/
//functia principala
void main()
{c=creare();
parcurge(c);}
```

2.5 Grafuri

Un *graf orientat* G este o pereche (V, E) unde V este o mulţime finită iar E este o relaţie binară pe V. Mulţimea V se numeşte mulţimea *vârfurilor* iar mulţimea E se numeşte mulţimea *arcelor* lui G $((u, v) \in E, u \in V, v \in V)$.

Într-un *graf neorientat* $G = (V, E)$ mulţimea muchiilor este constituită din perechi de vârfuri neordonate $(\{u, v\} \in E, u \in V, v \in V)$. Dacă (u, v) este un arc dintr-un graf orientat $G = (V, E)$ spunem că (u, v) este *incident din*

sau *pleacă din* vârful u şi este *incident în* sau *intră în* vârful v. Despre (u, v) sau $\{u, v\}$ spunem că sunt incidente vârfurilor u şi v. Dacă (u, v) sau $\{u, v\}$ reprezintă un arc (muchie) într-un graf spunem că vârful v este *adiacent* vârfului u. (Pentru simplitate folosim notaţia (u, v) şi pentru muchiile grafurilor neorientate.)



a) Graf orientat b) Graf neorientat

Figura 2.2: Exemple de grafuri

În figura (2.2) prezentăm un graf orientat şi un graf neorientat. Adiacenţa grafurilor se pune în evidenţă cu ajutorul unei *matrice de adiacenţă*. De exemplu matricea de adiacenţă a grafului orientat din figură este

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ 0 & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ 1 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 2 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 3 & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \end{array}.$$

$0, 1, 2, 3$ sunt etichetele vârfurilor grafului considerat. Dacă $(u, v) \in \{0, 1, 2, 3\} \times \{0, 1, 2, 3\}$ este un arc al grafului punem valoarea 1 pentru elementul aflat pe linia u şi coloana v a matricei. În caz contrar punem 0.

Matricea de adiacenţă a grafului neorientat este

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ 0 & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ 1 & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ 2 & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ 3 & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \end{array}.$$

Dacă $\{u, v\}$ este o muchie a grafului punem valoarea 1 pentru elementul aflat pe linia u şi coloana v a matricei. În caz contrar punem 0. Observăm că matricea de adiacenţă a unui graf neorientat este simetrică.

Gradul unui vârf al unui graf neorientat este numărul muchiilor incidente acestuia. Într-un graf orientat, *gradul exterior* al unui vârf este numărul arcelor care pleacă din el iar *gradul interior* al unui vârf este numărul arcelor ce intră în el. Suma gradului exterior şi a gradului interior este *gradul* vârfului.

Un *drum de lungime* k de la un vârf u la un vârf u' într-un graf $G = (V, E)$ este un şir de vârfuri $(v_0, v_1, v_2, ..., v_{k-1}, v_k)$ astfel încât $u = v_0, u' = v_k$ şi $(v_{i-1}, v_i) \in E$ pentru $i = 1, 2, ..., k$. Drumul *conţine* vârfurile $v_0, v_1, ..., v_{k-1}, v_k$ şi muchiile(arcele) $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$.

Lungimea unui drum este numărul de muchii (arce) din acel drum. Un drum este *elementar* dacă toate vârfurile din el sunt distincte.

Prezentăm în continuare un program scris în C care stabileşte, pe baza matricei de adiacenţă dacă între două vârfuri ale grafului există un drum. Ideea pe care se bazează programul este următoarea: dacă între două vârfuri i şi j există un drum, atunci oricare ar fi vârful k, între i şi k există un drum dacă şi numai dacă (i, k) este arc al grafului sau între j şi k există un drum. În final programul afişează o matrice în care elementul de pe linia i şi coloana j ia valoarea 1 dacă există un drum de la i la j şi valoarea 0 dacă nu există. Urmează programul:

```
# include <stdio.h>
# define maxcol 20
# define maxlin 20
unsigned char i,j,k,n,x[maxcol][maxlin],y[maxcol][maxlin];
void main(void)
{printf("Dimensiunea matricii n=");scanf("%d",&n);
for(i=0;i<n;i++)
for(j=0;j<n;j++)
```

```
{printf(" x[%d][%d]=",i,j);
scanf("%d",&x[i][j]);y[i][j]=x[i][j];}
j=0;while(j<n)
{
i=0;
while(i<n)
{if(y[i][j]!=0)
{k=0;
while(k<n)
{y[i][k]=y[i][k]||y[j][k];k++;};};
i++;};
j++;};
for(i=0;i<n;i++)
for(j=0;j<n;j++)
printf("y[%d][%d]=%d ",i,j,y[i][j]);}
```

Pentru graful orientat din figura (2.2) se obţine matricea

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ 0 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ 1 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 2 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 3 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \end{array}.$$

În cazul grafurilor neorientate, dacă $x_1 = x_r$ şi nici-o muchie nu este parcursă de două ori (adică nu apar perechi de muchii alăturate de forma $(x_i x_{i+1}), (x_{i+1}, x_i)$, drumul $(x_1, x_2, ..., x_r = x_1)$ se numeşte *ciclu*. Dacă muchiile ciclului sunt arce orientate avem un ciclu într-un graf orientat.

Un graf neorientat este *conex* dacă pentru fiecare două vârfuri u, v există un drum $(u, ..., v)$ de la u la v. Dacă un graf nu este conex, atunci el are $r \geq 2$ componente conexe care sunt subgrafuri conexe maximale ale lui G în raport cu relaţia de incluziune a mulţimilor. Numărul de vârfuri $|V(G)|$ al unui graf se numeşte *ordinul* grafului iar numărul muchiilor $|E(G)|$ reprezintă *mărimea* grafului.

Un graf (neorientat) conex care nu conţine cicluri se numeşte *arbore*. Dăm în continuare câteva teoreme de caracterizare a arborilor:

Teorema 2. *Fie $G = (V, E)$ un graf neorientat de ordin $n \geq 3$. Următoarele condiţii sunt echivalente:*

a) G este un graf conex fără cicluri;

b) G este un graf fără cicluri maximal (dacă i se adaugă lui G o muchie, atunci apare un ciclu);

c) G este un graf conex minimal (dacă se sterge o muchie a lui G, graful rezultat nu mai este conex).

Demonstraţie.a) $\Rightarrow$ *b).* Fie G conex şi fără cicluri. Fie u, v din V astfel că $(u, v) \in E$. Cum graful este conex, există un drum $(v, ..., u)$ de la v la u. Adăugând şi muchia (u, v), graful $G' = (V, E \cup (u, v))$ va conţine ciclul $(v, ..., u, v)$.

b) $\Rightarrow$ *c)* Dacă G nu ar fi conex, în virtutea proprietăţii de maximalitate, adăugând o nouă muchie ce uneşte două vârfuri din două componente conexe diferite nu obţinem un ciclu, ceea ce contrazice *b)*. Aşadar G este conex. Să presupunem mai departe prin absurd că $e \in E$ şi $G'' = (V, E - \{e\})$ este conex. Atunci există un drum ce uneşte extremităţile lui e în G, deci G conţine un ciclu, ceea ce contrazice din nou *b)*.

c) $\Rightarrow$ *a).* Dacă G ar conţine un ciclu şi e ar fi o muchie a acestui ciclu, atunci $G'' = (V, E - \{e\})$ rămâne conex, ceea ce contrazice *c)*.

Teorema 3. *Orice arbore $G = (V, E)$ de ordinul n, are $n - 1$ muchii.*

Demonstraţie. Mai întâi vom demonstra că G conţine cel puţin un vârf de gradul 1 (vârfuri terminale, frunze). Să presupunem prin absurd că gradul $d(v) \geq 2$ pentru orice $v \in V$. În acest caz să considerăm în G un drum D de lungime maximă şi să notăm cu x o extremitate a acestui drum. Cu presupunerea făcută, vârful x ar avea gradul cel puţin 2, deci în virtutea maximalităţii lui D trebuie să fie adiacent cel puţin altui vârf din D, formându-se astfel un ciclu în G. Apare o contradicţie.

Acum proprietatea că G are $n - 1$ muchii rezultă uşor prin inducţie. Ea este adevărată pentru $n = 1$. Presupunem că este adevărată pentru toţi arborii de ordin cel mult $n - 1$ şi fie G un arbore de ordin n. Dacă x este un nod terminal al lui G, atunci G' cu $V(G') = V - \{x\}$ este şi el un arbore de ordinul $n - 1$ şi conform ipotezei de inducţie va avea $n - 2$ muchii. Rezultă că G are $n - 1$ muchii.

Teorema 4. *Fie G un graf de ordinul $n \geq 3$. Următoarele condiţii sunt echivalente:*

a) G este un graf conex fără cicluri;

d) G este un graf fără cicluri cu $n - 1$ muchii;

e) G este conex şi are $n - 1$ muchii;

f) Există un unic drum între orice două vârfuri distincte ale lui G.

Demonstraţie.a) $\Rightarrow$ *d).* Avem *a)* $\Rightarrow$ (*b* şi teorema 2) $\Rightarrow$ *d).*

$d) \Rightarrow a)$. Presupunem prin absurd că G nu este conex. Adăugând un număr de muchii care să unească elemente din diverse componente conexe ale grafului putem obţine un graf conex fără cicluri cu ordinul mai mare ca $n-1$. Se ajunge la o contradicţie (în virtutea teoremei 2), deci G este conex.

$a) \Rightarrow e)$. Avem $a) \Rightarrow (c$ şi teorema 2$) \Rightarrow e)$.

$e) \Rightarrow a)$. Presupunem prin absurd că G are cicluri. Extrăgând în mod convenabil unele muchii, se ajunge astfel la un graf conex fără cicluri, de ordin n cu mai puţin de $n-1$ muchii. Se obţie astfel o contradicţie în virtutea teoremei 3.

$a) \Leftrightarrow f)$. Rezultă imediat în virtutea definiţiilor.

Din teoremele 2 şi 4 obţinem şase caracterizări diferite ale arborilor : $(a) - (f)$.

2.6 Arbori binari

Ne vom ocupa în continuare de studiul *arborilor binari* deoarece aceştia constituie una din cele mai importante structuri neliniare întâlnite în teoria algoritmilor. În general, structura de arbore se referă la o relaţie de ramnificare la nivelul nodurilor, asemănătoare aceleia din cazul arborilor din natură. În cele ce urmează vom introduce într-o altă manieră noţiunea de arbore.

Arborii sunt constituiţi din *noduri interne* (puncte de ramnificare) şi *noduri terminale (frunze).* Fie $V = \{v_1, v_2, ...\}$ o mulţime de noduri interne şi $B = \{b_1, b_2, ...\}$ o mulţime de frunze. Definim inductiv mulţimea arborilor peste V şi B :

Definiţie.

a) Orice element $b_i \in B$ este un arbore. b_i este de asemenea rădăcina unui arbore.

b) Dacă $T_1, ..., T_m, m \geq 1$ sunt arbori cu mulţimile de noduri interne şi frunze disjuncte două câte două iar $v \in V$ este un nou nod, atunci $(m+1)$ - tuplul $T = \langle v, T_1, ..., T_m \rangle$ este un arbore. Nodul v este rădăcina arborelui, $\rho(v) = m$ este gradul arborelui iar $T_i, i = 1, ..., m$ sunt subarbori ai lui T.

Când reprezentăm grafic un arbore rădăcina este deasupra iar frunzele sunt dedesupt (figura (2.3)).

Vom preciza termenii folosiţi atunci când ne referim în general la arbori. Fie T un arbore cu rădăcina v şi având subarborii $T_i, 1 \leq i \leq m$. Fie $w_i = root(T_i)$ rădăcina subarborelui T_i. Atunci w_i este *fiul* numărul i al lui v iar v este *tatăl* lui w_i. De asemenea w_i este fratele lui $w_j, i, j = 1, ..., m$. Noţiunea

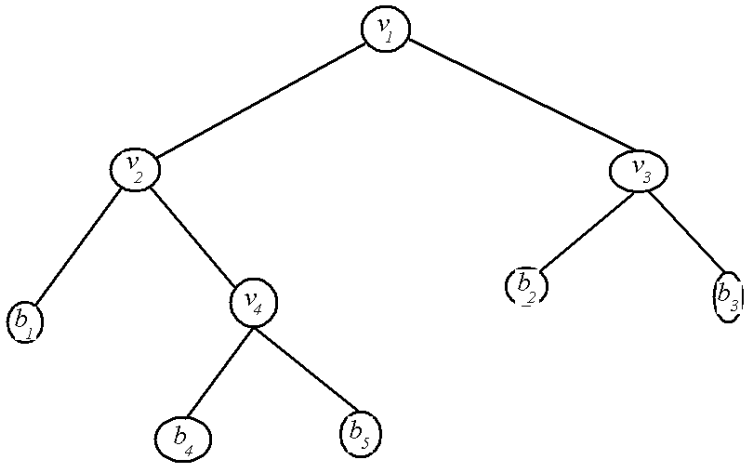


Figura 2.3: Exemplu de arbore binar

de *descendent (ascendent)* indică închiderea tranzitivă şi reflexivă a relaţiei de fiu (tată).

Nivelul (sau *adâncimea*) unui nod v al unui arbore T este definit astfel: dacă v este rădăcina lui T atunci $nivel\,(v,T)=0$. Dacă v nu este rădăcina lui T atunci pentru un anumit i, v aparţine subarborelui T_i. Vom pune $nivel\,(v,T)=1+nivel\,(v,T_i)$. Vom omite al doilea argument din sumă când contextul o va cere $(T_i=\emptyset)$.

Înălţimea $h\,(T)$ a unui arbore T este definită după cum urmează: $h\,(T)=\max\{nivel\,(b,T)\,;b\ este\ frunza\ lui\ T\}$. În exemplul din figură $nivel\,(v_3)=1$, $nivel\,(v_4)=2$, $nivel\,(b_5)=3$ şi $h\,(T)=3$.

Un arbore T este un *arbore binar* dacă toate nodurile interne ale lui T sunt rădăcini ale unor subarbori de gradul 1 sau 2. Un arbore T este *complet* dacă toate nodurile interne ale sale sunt rădăcini ale unor subarbori de gradul 2. Arborele binar din figură este un arbore complet. Un arbore complet binar cu n noduri interne are $n+1$ frunze. Primul respectiv al doilea subarbore este numit *subarborele stâng* respectiv *drept*

2.6.1 Parcurgerea arborilor binari

În cazul arborilor binari, informaţiile pot fi stocate în frunze sau în nodurile

interne. Fiecare nod al arborelui binar este o structură care conţine alături de informaţia specifică şi adresele nodurilor fiu stâng respectiv drept (figura (2.4)).

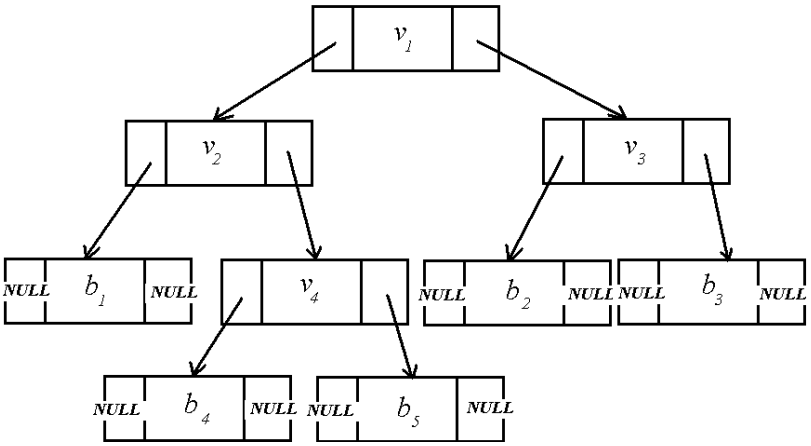


Figura 2.4: Exemplu de arbore binar cu precizarea legaturilor

Pentru a accesa informaţiile păstrate de un arbore, trebuie să-l explorăm (parcurgem). Cum orice arbore binar are trei componente (o rădăcină, un subarbore stâng şi un subarbore drept), în mod natural apar trei metode de parcurgere a arborelui:

- Parcurgere în **preordine (rsd)**: se vizitează rădăcina, se parcurge subarborele stâng, se parcurge sub arborele drept.

- Parcurgere în **postordine (sdr)**: se parcurge subarborele stâng, se parcurge subarborele drept, se vizitează rădăcina.

- Parcurgere **simetrică** sau în **inordine(srd)**: se parcurge subarborele stâng, se vizitează rădăcina, se parcurge subarborele drept. Aplicând aceste definiţii arborelui din figură obţinem $v_1v_2b_1v_4b_4b_5v_3b_2b_3$, pentru parcurgerea în preordine, $b_1b_4b_5v_4v_2b_2b_3v_3v_1$ pentru parcurgerea în postordine iar pentru parcurgerea în inordine $b_1v_2b_4v_4b_5v_1b_2v_3b_3$.

Vom prezenta în cele ce urmează un program $C++$ cu funcţiile de creare, parcurgere şi stergere a unui arbore. Pentru utilizarea nodurilor unui arbore binar a fost definită o structură cu următoarele câmpuri: **info** - câmpul

informaţie, în acest program de tip întreg, **st** - adresa nodului descendent stâng, **dr** - adresa nodului descendent drept. Programul realizează prelucrarea arborelui prin următoarele funcţii:

a) Funcţia **creare** pentru crearea arborelui efectuează operaţiile:

- alocă memoria necesară unui nod;

- citeşte informaţia nodului şi afişează mesaje de invitaţie pentru crearea nodurilor descendente (stâng şi drept);

- dacă informaţia introdusă este un număr întreg diferit de 0, se apelează recursiv funcţia pentru crearea subarborelui stâng stocându-sa adresa de legătură pentru accesul la descendenţi. Urmează apoi apelul recursiv pentru crearea subarborelui drept;

- dacă informaţia introdusă este 0, atunci nodului i se atribuie valoarea *NULL*.

Funcţia întoarce adresa nodului creat.

b) Funcţia **rsd** pentru parcurgerea în preordine efectuează operaţiile:

- prelucrează (afişează) rădăcina;

- trece la descendentul stâng apelându-se recursiv pentru parcurgerea subarborelui stâng;

- trece la descendentul drept apelându-se recursiv pentru parcurgerea subarborelui drept.

c) Funcţia **srd** pentru parcurgerea în inordine efectuează operaţiile:

- trece la descendentul stâng apelându-se recursiv pentru parcurgerea subarborelui stâng;

- prelucrează (afişează) rădăcina;

- trece la descendentul drept apelându-se recursiv pentru parcurgerea subarborelui drept.

d) Funcţia **sdr** pentru parcurgerea în postordine efectuează operaţiile:

- trece la descendentul stâng apelându-se recursiv pentru parcurgerea subarborelui stâng;

- trece la descendentul drept apelându-se recursiv pentru parcurgerea subarborelui drept;

- prelucrează (afişează) rădăcina.

e) Funcţia **sterge** pentru ştergerea arborelui efectuează operaţiile:

- se şterge subarborele stâng: se apelează recursiv ştergerea pentru descendentul stâng;

- se şterge subarborele drept: se apelează recursiv ştergerea pentru descendentul drept;

- se şterge nodul curent;

Urmează programul.

```
#include <iostream.h>
typedef struct nod {int info; struct nod *st; struct nod *dr;}arbore;
arbore *rad; int n;
// Se declara functiile
/*********************************/
arbore *creare();
void srd(arbore *rad);
void rsd(arbore *rad);
void sdr(arbore *rad);
void sterge(arbore *rad);
/*********************************/
// Functia principala
void main() {
cout<<"Radacina=";
rad=creare();
cout<<"Urmeaza arborele parcurs in inordine"<<endl;
srd(rad);
cout<<"Urmeaza arborele parcurs in preordine"<<endl;
rsd(rad);
cout<<"Urmeaza arborele parcurs in postordine"<<endl;
sdr(rad);
sterge(rad);}
/*********************************/
// Functia de creare a arborelui
arbore *creare( )
{arbore *p; cin>>n;
if (n!=0)
{p=new arbore;
p->info=n;cout<<p->info<<"->st ";p->st=creare( );
cout<<p->info<<"->dr ";p->dr=creare( );return p;}
 else
p=NULL;return p;}
/*********************************/
// Functia de parcurgere in inordine
void srd(arbore *rad)
{if (rad!=NULL)
{srd(rad->st);
```

```
cout<<rad->info<<" ";
srd(rad->dr);}}
/*******************************/
// Functia de parcurgere in postordine
void sdr(arbore *rad)
{if (rad!=NULL)
{sdr(rad->st);
sdr(rad->dr);cout<<rad->info<<" ";}}
/*********************************/
// Functia de parcurgere in preordine
void rsd(arbore *rad)
{if(rad!=NULL)
{cout<<rad->info<<" ";
rsd(rad->st);
rsd(rad->dr);}}
/*********************************/
// Functia de stergere
void sterge(arbore *rad)
{ if (rad!=NULL){sterge(rad->st);sterge(rad->dr);
cout<<"S-a sters"<<rad->info<<endl;
delete rad;}};
```

Vom prezenta mai departe o variantă nerecursivă de traversare a unui arbore în inordine folosind o stivă auxiliară **a**. Schematic, algoritmul se prezintă astfel:

```
p=radacina; a=NULL;
do {
while(p) { a⇐p /*pune nodul p al arborelui in stiva*/;
p=p->st;}
if(!a) break;
else p⇐a/*scoate p din stiva*/;
viziteaza p; p=p->adr;
 } while(p||a);
```

Ideea algoritmului este sa salvăm nodul curent **p** în stivă şi apoi să traversăm subarborele stâng; după aceea scoatem nodul **p** din stivă, îl vizităm şi apoi traversăm subarborele drept.

Să demonstrăm că acest algoritm parcurge un arbore binar de n noduri în ordine simetrică folosind inducţia după n. Pentru aceasta vom demonstra

următorul rezultat: după o intrare în ciclul **do**, cu $\mathbf{p}_0$ rădăcina unui subarbore cu n noduri şi stiva **a** conţinând nodurile **a[1],...,a[m]**, procedura va traversa subarborele în chestiune în ordine simetrică iar după parcurgerea lui stiva va avea în final aceleaşi noduri **a[1],...,a[m]**. Afirmaţia este evidentă pentru $n = 0$. Dacă $n > 0$, fie $\mathbf{p}=\mathbf{p}_0$ nodul arborelui binar la intrarea în setul de instrucţiuni al ciclului **do**. Punând $\mathbf{p}_0$ în stivă, aceasta devine **a[1],...,a[m]**,$\mathbf{p}_0$ iar noua valoare a lui **p** este $\mathbf{p}=\mathbf{p}_0 \rightarrow$**st**. Acum subarborele stâng are mai puţin de n noduri şi conform ipotezei de inducţie subarborele stâng va fi traversat în inordine, după parcurgere stiva fiind **a[1],...,a[m]**,$\mathbf{p}_0$. Se scoate $\mathbf{p}_0$ din stivă (aceasta devenind **a[1],...,a[m]),** se vizitează $\mathbf{p}=\mathbf{p}_0$ şi se atribuie o nouă valoare lui **p** adică $\mathbf{p}=\mathbf{p}_0 \rightarrow$**dr**. Acum subarborele drept are mai puţin de n noduri şi conform ipotezei de inducţie se va traversa arborele drept având în final în stivă nodurile **a[1],...,a[m].**

Aplicând propoziţia de mai sus, se intră în ciclul **do** cu rădăcina arborelui şi stiva vidă, şi se iese din ciclu cu arborele traversat şi stiva vidă.

Urmează programul.

```
#include <iostream.h>
typedef struct nod {int inf; nod *st; nod *dr;}arbore;
arbore *rad;int n;
typedef struct nods{arbore *inf; nods*ant;}stiva;
typedef struct{ arbore *arb; stiva *stiv;} arbstiv;
/*****************************/
//Functia de punere in stiva
stiva *push(stiva*a, arbore *info)
{stiva *r;
r=new stiva;
r->inf=info;
if(!a)r->ant=NULL;
else r->ant=a;
a=r;return a;}
/*****************************/
//Functia de scoatere din stiva si de vizitare a nodului
arbstiv pop(stiva *a)
{arbstiv q;stiva *r;
if(!a)
cout<<"Stiva vida "<<endl;
else
{q.arb=a->inf;r=a;
```

```
cout<<(a->inf)->inf<<" ";
a=a->ant;delete r;q.stiv=a;}
return q;}
/*********************************/
//Functia de creare a arborelui
arbore *creare()
{arbore *p; ;cin>>n;
if (n!=0)
{p=new arbore;
p->inf=n;cout<<p->inf<<"->st ";p->st=creare( );
cout<<p->inf<<"->dr ";p->dr=creare( );return p;}
else
p=NULL;return p;}
/*********************************/
//Functia principala care realizeaza traversarea in inordine
void main()
{stiva *a; arbore *p;arbstiv q;
cout<<"Radacina"<<endl;
p=creare();
a=NULL;
do{
while (p) {a=push(a,p);p=p->st;}
if (!a) break;
else {q=pop(a);p=q.arb;a=q.stiv;}
p=p->dr;}
while(p||a); }
```

2.7 Algoritmul lui Huffman

Algoritmul de codificare (compresie, compactare) *Huffman* poartă numele inventatorului său, David Huffman, profesor la MIT. Acest algoritm este ideal pentru compresarea (compactarea, arhivarea) textelor şi este utilizat în multe programe de compresie.

2.7.1 Prezentare preliminară

Algoritmul lui Huffman aparţine familiei de algoritmi ce realizează **codificări cu lungime variabilă**. Aceasta înseamnă că simbolurile individuale (ca de exemplu caracterele într-un text) sunt înlocuite de secvenţe de biţi **(cuvinte de cod)** care pot avea lungimi diferite. Astfel, simbolurilor care se întâlnesc de mai multe ori în text (fişier) li se atribuie o secvenţă mai scurtă de biţi în timp ce altor simboluri care se întâlnesc mai rar li se atribuie o secvenţă mai mare. Pentru a ilustra principiul, să presupunem că vrem să compactăm următoarea secvenţă :$AEEEEBEEDECDD$. Cum avem 13 caractere (simboluri), acestea vor ocupa în memorie $13 \times 8 = 104$ biţi. Cu algoritmul lui Huffman, fişierul este examinat pentru a vedea care simboluri apar cel mai frecvent (în cazul nostru E apare de şapte ori, D apare de trei ori iar A, B şi C apar câte o dată). Apoi se construieşte (vom vedea în cele ce urmează cum) un arbore care înlocuieşte simbolurile cu secvenţe (mai scurte) de biţi. Pentru secvenţa propusă, algoritmul va utiliza substituţiile **(cuvintele de cod)** $A = 111,\ B = 1101,\ C = 1100,\ D = 10,$ $E = 0$ iar secvenţa compactată (codificată) obţinută prin concatenare va fi 111000011010010011001010. Aceasta înseamnă că s-au utilizat 24 biţi în loc de 104, raportul de compresie fiind $24/104 \approx 1/4$.. Algoritmul de compresie al lui Huffman este utilizat în programe de compresie ca **pkZIP, lha, gz, zoo, arj**.

2.7.2 Coduri prefix. Arbore de codificare

Vom considera în continuare doar codificările în care nici-un cuvânt nu este prefixul altui cuvânt. Aceste codificări se numesc **codificări prefix**. Codificarea prefix este utilă deoarece simplifică atât codificarea (deci compactarea) cât şi decodificarea. Pentru orice codificare binară a caracterelor; se concatenează cuvintele de cod reprezentând fiecare caracter al fişierului ca în exemplul din subsecţiunea precedentă.

Decodificarea este de asemenea simplă pentru codurile prefix. Cum nici un cuvânt de cod nu este prefixul altuia, începutul oricărui fişier codificat nu este ambiguu. Putem să identificăm cuvântul de cod de la început, să îl convertim în caracterul original, să-l îndepărtăm din fişierul codificat şi să repetăm procesul pentru fişierul codificat rămas. În cazul exemplului prezentat mai înainte şirul se desparte în

$$111 - 0 - 0 - 0 - 0 - 1101 - 0 - 0 - 10 - 0 - 1100 - 10 - 10,$$

secvenţă care se decodifică în $AEEEBEEDECDD$. Procesul de decodificare necesită o reprezentare convenabilă a codificării prefix astfel încât cuvântul iniţial de cod să poată fi uşor identificat. O astfel de reprezentare poate fi dată de un **arbore binar de codificare**, care este un arbore complet (adică un arbore în care fiecare nod are 0 sau 2 fii), ale cărui frunze sunt caracterele date. Interpretăm un cuvânt de cod binar ca fiind o secvenţă de biţi obţinută etichetând cu 0 sau 1 muchiile drumului de la rădăcină până la frunza ce conţine caracterul respectiv: cu 0 se etichetează muchia ce uneşte un nod cu fiul stâng iar cu 1 se etichetează muchia ce uneşte un nod cu fiul drept. În figura (2.5) este prezentat arborele de codificare al lui Huffman corespunzător exemplului nostru. Notând cu $\mathcal{C}$ alfabetul din care fac parte simbolurile (caracterele), un arbore de codificare are exact $|\mathcal{C}|$ frunze, una pentru fiecare literă (simbol) din alfabet şi aşa cum se ştie din teoria grafurilor, exact $|\mathcal{C}|-1$ noduri interne (notăm cu $|\mathcal{C}|$, cardinalul mulţimii $\mathcal{C}$).

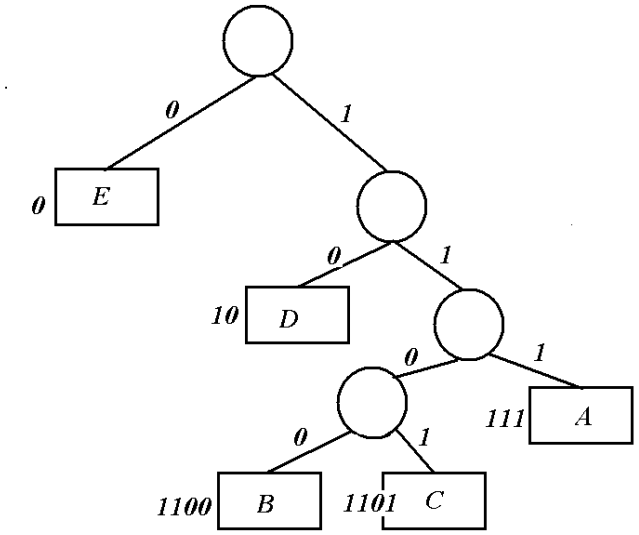


Figura 2.5: Exemplu de arbore Huffman

Dându-se un arbore T, corespunzător unei codificări prefix, este foarte simplu să calculăm numărul de biţi necesari pentru a codifica un fişier.

Pentru fiecare simbol $c \in \mathcal{C}$, fie $f(c)$ frecvenţa (numărul de apariţii) lui c în fişier şi să notăm cu $d_T(c)$ adâncimea frunzei în arbore. Numărul de biţi necesar pentru a codifica fişierul este numit **costul arborelui** T şi se

calculează cu formula

$$COST\,(T) = \sum_{c \in \mathcal{C}} f\,(c)\, d_T\,(c)\,.$$

2.7.3 Construcţia codificării prefix a lui Huffman

Huffman a inventat un algoritm greedy care construieşte o codificare prefix optimă numită **codul Huffman**. Algoritmul construieşte arborele corespunzător codificării optime (numit **arborele lui Huffman**) pornind *de jos în sus*. Se începe cu o mulţime de $|\mathcal{C}|$ frunze şi se realizează o secvenţă de $|\mathcal{C}|-1$ operaţii de *fuzionări* pentru a crea arborele final. În algoritmul scris în pseudocod care urmează, vom presupune că $\mathcal{C}$ este o mulţime de n caractere şi fiecare caracter $c \in \mathcal{C}$ are frecvenţa $f\,(c)$. Asimilăm $\mathcal{C}$ cu o *pădure* constituită din arbori formaţi dintr-o singură frunză.Vom folosi o stivă $\mathcal{S}$ formată din noduri cu mai multe câmpuri; un câmp pastrează ca informaţie pe $f\,(c)$, alt câmp păstrează rădăcina c iar un câmp suplimentar păstrează adresa nodului anterior (care indică un nod ce are ca informaţie pe $f\,(c')$ cu proprietatea că $f\,(c) \leq f\,(c')$). Extragem din stivă vârful şi nodul anterior (adică obiectele cu frecvenţa cea mai redusă) pentru a le face să fuzioneze. Rezultatul fuzionării celor două noduri este un nou nod care în câmpul informaţiei are $f\,(c)+f\,(c')$ adică suma frecvenţelor celor două obiecte care au fuzionat. De asemenea în al doilea câmp apare rădăcina unui arbore nou format ce are ca subarbore stâng, respectiv drept, arborii de rădăcini c şi c'. Acest nou nod este introdus în stivă dar nu pe poziţia vârfului ci pe poziţia corespunzătoare frecvenţei sale. Se repetă operaţia până când în stivă rămâne un singur element. Acesta va avea în câmpul rădăcinilor chiar rădăcina arborelui Huffman.

Urmează programul în pseudocod. Ţinând cont de descrierea de mai sus a algoritmului numele instrucţiunilor programului sunt suficient de sugestive.

n$\leftarrow |\mathcal{C}|$
$S \leftarrow C$
cat timp (S **are mai mult decât un nod)**
{ $z \leftarrow$ **ALOCA-NOD( )**
$x \leftarrow$**stanga**$[z] \leftarrow$**EXTRAGE-MIN**(S)
$y \leftarrow$**dreapta**$[z] \leftarrow$**EXTRAGE-MIN**(S)
f$(z) \leftarrow$ **f**$(x)+$**f**(y)
INSEREAZA(S, z) **}**
returnează EXTRAGE-MIN(S).

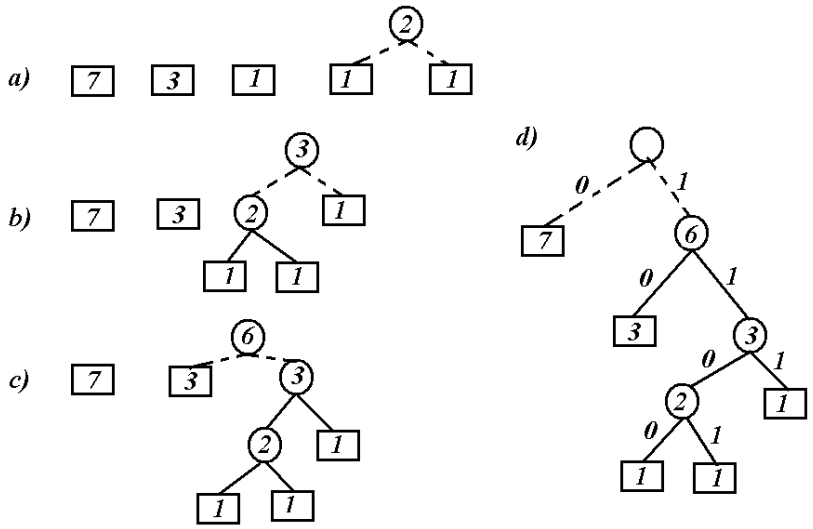


Figura 2.6: Construirea arborelui Huffman

În cazul exemplului deja considerat, avem $f(E) = 7,\ f(D) = 3,\ f(A) = f(B) = f(C) = 1$. Putem, pentru comoditate să etichetăm frunzele nu cu simbolurile corespunzătoare, ci cu frecvenţele lor. În figura (2.6) prezentăm arborii aflaţi în nodurile stivei, după fiecare repetare a instrucţiunilor din ciclul **while**. Se pleacă cu o pădure de frunze, ordonată descrescător după frecvenţele simbolurilor şi se ajunge la arborele de codificare.

Prezentăm mai jos şi programul în $C++$ pentru algoritmul Huffman de codificare prefixată.

```
#include<iostream.h>
#include<string.h>
typedef struct nod{char *symb; nod*st;nod*dr;} arbore;
typedef struct nods{ arbore*rad; nods*ant;int frecv;} stiva;
/*********************************************/
/*Functia de punere a unui nou nod in stiva ordonata dupa frecventa*/
stiva *push(stiva*varf, arbore *a, int f)
{stiva *v,*p,*q;
v=new stiva; v->rad=a;v->frecv=f;v->ant=NULL;
if(!varf)
{varf=v; return varf;}
for(p=varf,q=NULL;(p!=NULL)* (p->frecv<f;q=p,p=p->ant
```

```
v->ant=p;
if(q!=NULL) q->ant=v;
else {varf=v;}
return varf;}
/*********************************************/
//Functia de stergere a varfului stivei
void pop(stiva *
{stiva*r;
r=varf;
varf=varf->ant;
delete r;
return;}
/*********************************************/
//Functia de fuzionare a doi arbori
arbore*fuziune(arbore *p, arbore*q)
{arbore*r;
r=new arbore;
r->symb='\0';
r->dr=p;r->st=q;
return r;}
/*********************************************/
//Functia de parcurgere in preordine a arborelui lui Huffman
//si de codificare a frunzelor
void rsd(arbore*rad, char*shot)
{char frunza[60];
for (int i=0;i<60;i++)
frunza[i]='\0';
if (rad==NULL)return;
if(rad->symb!=NULL)
cout<<rad->symb<<":"<<shot<<endl;
if(shot!=NULL)
strcpy(frunza,shot);
strcat(frunza,"0");
rsd(rad->st,frunza);
if(shot!=NULL)
strcpy(frunza,shot);
strcat(frunza,"1");
rsd(rad->dr,frunza);}
```

```
/*****************************************/
//Functia de creare a unui arbore constand dintr-o singura
//frunza (radacina) care contine simbolul ce trebuie codificat
arbore *frunza(char *simb)
{arbore*r;r=new arbore;
r->symb=simb;r->st=NULL;r->dr=NULL; return r;}
/*****************************************/
//Functia de parcurgere a stivei
void parcurge(stiva*s)
{stiva*rr;rr=s;
if(!rr) cout<<"Stiva vida"<<endl;
else {cout<<"Urmeaza stiva"<<endl;
while(rr){
cout<<(rr->rad)->symb<<" ";
rr=rr->ant;}}
cout<<endl;return;}
/*****************************************/
//Functia principala
void main()
{arbore *r1,*r2;int f1,f2; stiva *vf;
vf=new stiva; vf=NULL;
vf=push(vf,frunza("D"),3);
vf=push(vf,frunza("A"),1);
vf=push(vf,frunza("B"),1);
vf=push(vf,frunza("C"),1);
vf=push(vf,frunza("E"),7);
parcurge(vf);
cout<<endl;
do
{r1=vf->rad;f1=vf->frecv;pop(vf);
r2=vf->rad;f2=vf->frecv;pop(vf);
vf=push(vf,fuziune(r1,r2),f1+f2);
} while(vf->ant);
cout<<"Urmeaza codificarea"<<endl;
rsd(vf->rad->st,"0");
rsd(vf->rad->dr,"1");}
```

2.7.4 Optimalitatea algoritmului Huffman

Definiţie. Un arbore de codificare T pentru alfabetul $\mathcal{C}$ este *optim* dacă pentru orice alt arbore de codificare T' al aceluiaşi alfabet avem

$$COST(T) = \sum_{c \in \mathcal{C}} f(c)\, d_T(c) \leq COST(T') = \sum_{c \in \mathcal{C}} f(c)\, d_{T'}(c).$$

Vom demonstra în cele ce urmează că algoritmul Huffman construieşte un arbore de codificare optim.

Lema 1. *Fie T un arbore de codificare optim pentru alfabetul $\mathcal{C}$. Dacă $f(c) < f(c')$ atunci $d_T(c) \geq d_T(c')$.*

Demonstraţie. Să presupunem prin absurd că avem $f(c) < f(c')$ şi $d_T(c) < d_T(c')$. Schimbând între ele frunzele care conţin pe c şi c' obţinem un nou arbore de codificare cu costul

$$COST(T) - f(c)\, d_T(c) - f(c')\, d_T(c') + f(c)\, d_T(c') + f(c')\, d_T(c) =$$

$$= COST(T) - (f(c) - f(c'))\,(d_T(c) - d_T(c')) < COST(T),$$

ceea ce contrazice ipoteza de optimalitate.

Lema 2. *Fie frecvenţele minime f_1 şi f_2 corespunzătoare simbolurilor c_1 şi c_2 din alfabetul $\mathcal{C}$. Atunci există un arbore de codificare optim în care frunzele c_1 şi c_2 sunt fraţi.*

Demonstrţie. Fie h înălţimea unui arbore de codificare optim T. Fie γ un nod de adâncime $h-1$ şi c_i şi c_j fii săi, care evident sunt frunze. Să presupunem că $f(c_i) \leq f(c_j)$. Conform lemei precedente sau $f(c_i) = f_1$ sau $d_T(c_i) \leq d_T(c_1)$ de unde $d_T(c_i) = d_T(c_1)$ din alegerea lui γ. În ambele cazuri putem schimba între ele frunzele c_i şi c_1 fără a afecta costul arborelui. La fel procedăm cu c_2 şi c_j şi obţinem un arbore de codificare optim în care c_1 şi c_2 sunt fraţi.

Teorema 5. *Algoritmul Huffman construieşte un arbore de codificare optim.*

Demonstraţie (prin inducţie după $n = |\mathcal{C}|$). Teorema este evidentă pentru $n \leq 2$. Să presupunem că $n \geq 3$ şi fie T_{Huff} arborele construit cu algoritmul Huffman pentru frecvenţele $f_1 \leq f_2 \leq ... \leq f_n$. Algoritmul adună frecvenţele f_1 şi f_2 şi construieşte nodul corespunzător frecvenţei $f_1 + f_2$. Fie

T'_{Huff} arborele construit cu algoritmul Huffman pentru mulţimea de frecvenţe $\{f_1 + f_2, f_3, ..., f_n\}$. Avem

$$COST\left(T_{Huff}\right) = COST\left(T'_{Huff}\right) + f_1 + f_2,$$

deoarece T_{Huff} poate fi obţinut din T'_{Huff} înlocuind frunza corespunzătoare frecvenţei $f_1 + f_2$ cu un nod intern având ca fii frunzele de frecvenţe f_1 şi f_2. De asemenea, conform ipotezei de inducţie T'_{Huff} este un arbore de codificare optim pentru un alfabet de $n-1$ simboluri cu frecvenţele $f_1 + f_2, f_3, ..., f_n$. Fie T_{opt} un arbore de codificare optim satisfăcând lema anterioară, adică frunzele de frecvenţe f_1 şi f_2 sunt fraţi în T_{opt}. Fie T' arborele obţinut din T_{opt} prin înlocuirea frunzelor de frecvenţe f_1 şi f_2 şi a tatălui lor cu o singură frunză de frecvenţă $f_1 + f_2$. Atunci

$$COST\left(T_{opt}\right) = COST\left(T'\right) + f_1 + f_2 \geq$$

$$\geq COST\left(T'_{Huff}\right) + f_1 + f_2 = COST\left(T_{Huff}\right),$$

deoarece $COST\left(T'\right) \geq COST\left(T'_{Huff}\right)$ conform ipotezei de inducţie. Rezultă de aici

$$COST\left(T_{opt}\right) = COST\left(T_{Huff}\right).$$

Capitolul 3

Tehnici de sortare

3.1 Heapsort

Definiţie. Un vector $A[1..n]$ este un **heap (ansamblu)** dacă satisface **proprietatea de heap** :

$$A[\lfloor k/2 \rfloor] \geq A[k], \ 2 \leq k \leq n.$$

Folosim notaţia $\lfloor \cdot \rfloor$ pentru a desemna partea întreagă a unui număr real.

Un vector A care reprezintă un heap are două atribute: *lungime[A]* este numărul elementelor din vector şi *dimensiune-heap[A]* reprezintă numărul elementelor heap-ului memorat în vectorul A. Astfel, chiar dacă $A[1..lungime[A]]$ conţine în fiecare element al său date valide, este posibil ca elementele următoare elementului *A[dimensiune-heap[A]]*, unde *dimensiune-heap[A]* $\leq$ *lungime[A]*, să nu aparţină heap-ului.

Structurii de **heap** i se ataşează în mod natural un arbore binar aproape complet (figura (3.1)).

Fiecare nod al arborelui corespunde unui element al vectorului care conţine valorile ataşate nodurilor. Rădăcina arborelui este $A[1]$. Dat fiind un indice i, corespunzător unui nod, se poate determina uşor indicii nodului tată, $TATA(i)$ şi ai fiilor $STANG(i)$ şi $DREPT(i)$:

indicele $TATA(i)$
returnează $\lfloor i/2 \rfloor$ (partea întreagă a lui $i/2$),
indicele $STANG(i)$
returnează $2i$,
indicele $DREPT(i)$

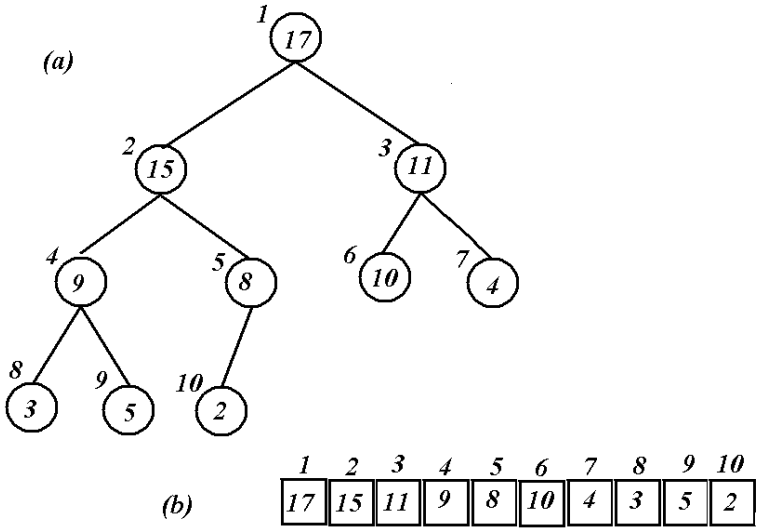


Figura 3.1: Exemplu de heap reprezentat sub forma unui arbore binar şi sub forma unui vector

returnează $2i + 1$.

Pentru orice nod i diferit de rădăcină este, în virtutea definiţiei heap-ului, adevărată **proprietatea de heap**:

$$A[TATA(i)] \geq A[i].$$

Definim *înălţimea* unui nod al arborelui ca fiind numărul muchiilor celui mai lung drum ce nu vizitează tatăl nodului şi leagă nodul respectiv cu o frunză. Evident, înalţimea arborelui este înălţimea rădăcinii.

3.1.1 Reconstituirea proprietăţii de heap

Funcţia **ReconstituieHeap** este un subprogram important în prelucrarea heap-urilor. Datele de intrare sunt un vector A şi un indice i. Atunci când se apelează **ReconstituieHeap** se presupune că subarborii având ca rădăcini nodurile $STANG(i)$ şi $DREPT(i)$ sunt heap-uri. Dar cum elementul $A[i]$ poate fi mai mic decât descendenţii săi, este posibil ca acesta să nu respecte **proprietatea de heap**. Sarcina funcţiei **ReconstituieHeap** este să *scufunde* în heap valoarea $A[i]$, astfel încât subarborele care are în rădăcină valoarea elementului de indice i, să devină un heap.

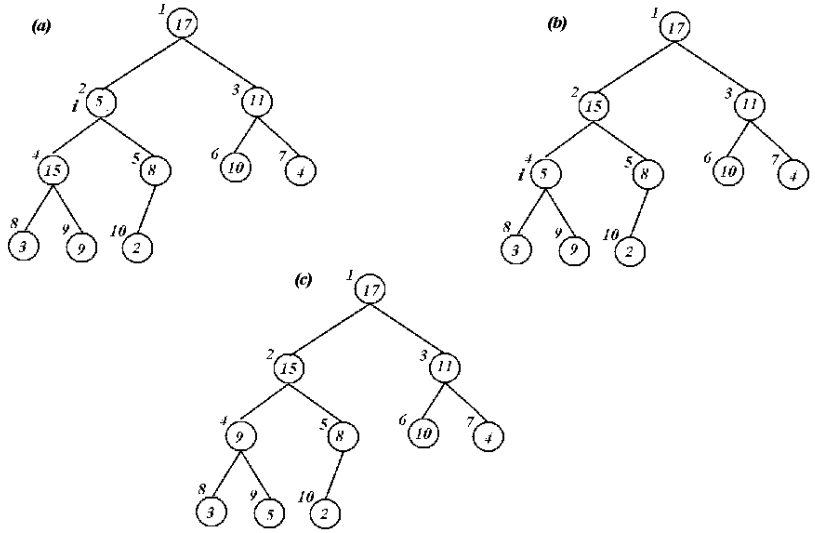


Figura 3.2: Efectul funcţiei ReconstituieHeap

Urmează funcţia scrisă în pseudocod.
ReconstituieHeap(A,i)
st←STANG(i)
dr←DREPT(i)
dacă st≤ dimensiune-heap[A] şi A[st]>A[i] atunci
maxim←st
altfel
maxim←i
dacă dr≤ dimensiune-heap[A] şi A[dr]>A[i] atunci
maxim←dr
dacă maxim≠i atunci
schimbă A[i]↔A[maxim]
ReconstituieHeap(A,maxim)
În figura (3.2) este ilustrat efectul funcţiei **ReconstituieHeap.**

În figura (3.2, a) este desenată configuraţia iniţială a heap-ului unde $A[2]$ nu respectă proprietatea de heap deoarece nu este mai mare decât descendenţii săi. Proprietatea de heap este restabilită pentru nodul 2 în figura (3.2, b) prin interschimbarea lui $A[2]$ cu $A[4]$, ceea ce anulează proprietatea de heap pentru nodul 4. Apelând recursiv **ReconstituieHeap**$(A, 4)$ se poziţionează valoarea lui i pe 4. După interschimbarea lui $A[4]$ cu $A[9]$

aşa cum se vede în figura (3.2, c), nodul 4 ajunge la locul său şi apelul recursiv **ReconstituieHeap**$(A, 9)$ nu mai găseşte elemente care nu îndeplinesc proprietatea de heap.

3.1.2 Construcţia unui heap

Funcţia **ReconstituieHeap** poate fi utilizată de *jos în sus* pentru transformarea vectorului $A[1..n]$ în heap.

Cum toate elementele subşirului $A\,[\lfloor n/2 \rfloor + 1..n]$ sunt frunze, ele pot fi considerate heap-uri formate din câte un element. Funcţia **ConstruiesteHeap** pe care o prezentăm în continuare traversează restul elementelor şi execută funcţia **ReconstituieHeap** pentru fiecare nod întâlnit. Ordinea de prelucrare a nodurilor satisface cerinţa ca subarborii, având ca rădăcină descendenţi ai nodului i să formeze heap-uri înainte ca **ReconstituieHeap** să fie executat pentru aceste noduri.

Urmează, în pseudocod funcţia **ConstruiesteHeap:**
dimensiune-heap[A]←**lungime[A]**
pentru i← $\lfloor$**lungime**[**A**]/**2**$\rfloor$**,1 executa**
ReconstituieHeap(A,i).

Figura (3.2) ilustrează modul de aplicare a funcţiei **ConstruiesteHeap.** În figură se vizualizează structurile de date (heap-urile) în starea lor anterioară apelării funcţiei **ReconstituieHeap**. (a) Se consideră vectorul A cu 10 elemente şi arborele binar corespunzător. Se observă că variabila de control i a ciclului în momentul apelului funcţiei **ReconstituieHeap(A,i)** indică nodul 5. (b) reprezintă rezultatul; variabila de control i a ciclului indică acum nodul 4. (c)-(e) vizualizează iteraţiile succesive ale ciclului **pentru** din **ConstruiesteHeap**. Se observă că atunci când se apelează funcţia **ReconstituieHeap** pentru un nod dat, subarborii acestui nod sunt deja heap-uri. (f) prezintă heap-ul final.

3.1.3 Algoritmul heapsort

Algoritmul de sortare **heapsort** începe cu apelul funcţiei **ConstruiesteHeap** în scopul transformării vectorului de intrare $A[1..n]$ în heap. Deoarece cel mai mare element al vectorului este ataşat nodului rădăcină $A[1]$, acesta va ocupa locul definitiv în vectorul ordonat prin interschimbarea sa cu $A[n]$. Mai departe, excluzând din heap elementul de pe locul n, şi micşorând cu 1 *dimensiune-heap[A]*, restul de $A[1..(n-1)]$ elemente se pot transforma uşor

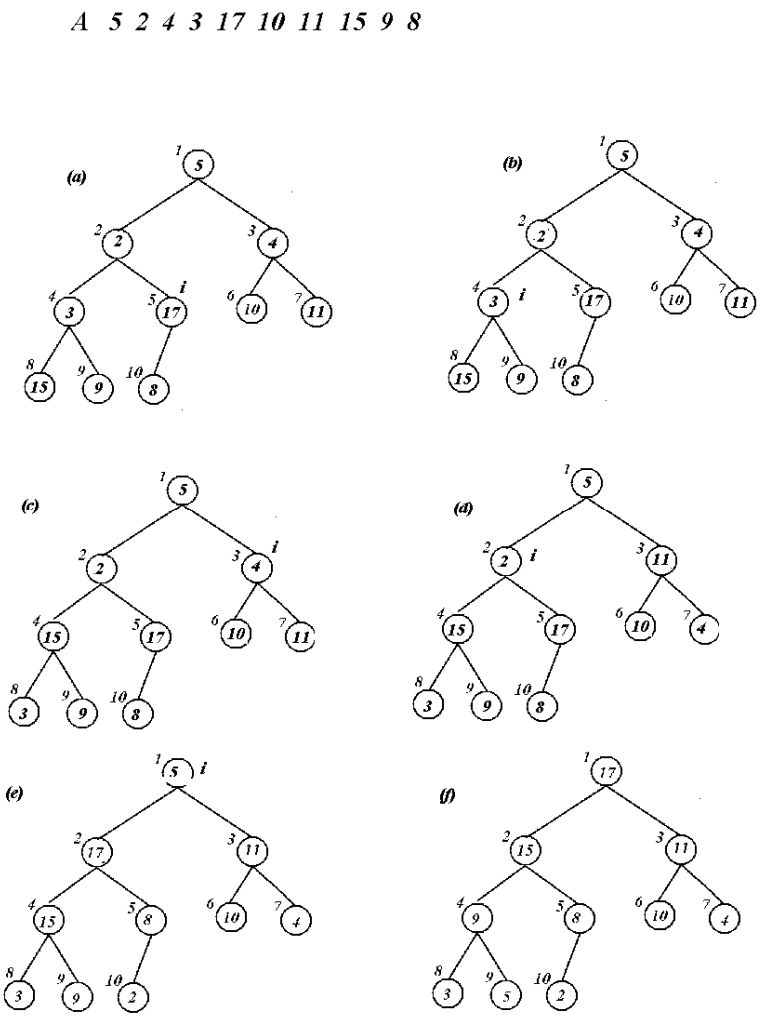


Figura 3.3: Model de execuţie a funcţiei ConstruiesteHeap

în heap, deoarece subarborii nodului rădăcină au proprietatea de heap, cu eventuala excepţie a elementului ajuns în nodul rădăcină. Pentru aceasta se apelează funcţia **ReconstituieHeap(A,1).** Procedeul se repetă micşorând dimensiunea heap-ului de la $n-1$ la 2.

Urmează, scris în pseudocod, algoritmul descris de funcţia **Heapsort(A):**

```
ConstruiesteHeap(A),
pentru i←lungime[A],2 executa
schimba A[1]↔A[i]
dimensiune-heap[A]←dimensiune-heap[A]-1
ReconstituieHeap(A,i)
```

Vom scrie programul **heapsort** şi în $C++$. Faţa de descrierea de mai înainte a algoritmului, vor interveni câteva mici modificări datorate faptului că în $C++$ indexarea elementelor unui vector începe de la 0 şi nu de la 1.

```
/************************************/
#include<iostream.h>
 void ReconstituieHeap(int* A, int i, int dimensiune )
{int a,maxim, stang=2*i+1,drept=stang+1;
if (stang<dimensiune&&A[stang]>A[i]) maxim=stang;
 else maxim=i;
if (drept<dimensiune&&A[drept]>A[maxim]) maxim=drept;
if(maxim!=i){a=A[i];A[i]=A[maxim];A[maxim]=a;
ReconstituieHeap(A,maxim,dimensiune); }}
/************************************/
void ConstruiesteHeap( int* A, int dimensiune )
{ int i;
for (i = (dimensiune - 2)/2; i >= 0; i–)
ReconstituieHeap(A, i, dimensiune);}
/************************************/
void heapsort( int* A, int dimensiune )
 { int i, temp;
ConstruiesteHeap( A, dimensiune );
for (i = dimensiune-1; i >=1; i–) {
temp = A[i]; A[i] = A[0]; A[0]=temp;
dimensiune=dimensiune-1;
 ReconstituieHeap( A,0,dimensiune );}}
/************************************/
void main()
{int N=10;
```

```
int A[10];
A[0]=10;A[1]=8;A[2]=6;A[3]=5;
A[4]=11;A[5]=5;A[6]=17;A[7]=9;A[8]=3;A[9]=21;
heapsort(A,N);
cout<<" Sirul sortat (metoda heapsort)"<<endl;
for(int i=0;i<N;i++) cout<<A[i]<<" "; }
```

Timpul de execuţie

Funcţia **ReconstituieHeap** constă din comparaţii şi interschimbări ale de elemente aflate pe nivele consecutive. Timpul de execuţie al funcţiei va fi deci de ordinul înălţimii arborelui binar asociat care este de ordinul $O(\ln n)$ unde n este numărul de elemente din heap.

Funcţia **ConstruiesteHeap** apelează funcţia **ReconstituieHeap** de n ori. Deducem de aici că timpul de execuţie al funcţiei **ConstruiesteHeap** este de ordinul $O(n \ln n)$. Funcţia **heapsort** apelează o dată funcţia **ConstruiesteHeap** şi de $n-1$ ori funcţia **ReconstituieHeap**. Rezultă de aici că timpul de execuţie al funcţiei **heapsort** este $O(n \ln n) + (n-1)O(\ln n) = O(n \ln n)$.

3.2 Cozi de priorităţi

Vom prezenta în cele ce urmează încă o aplicaţie a noţiunii de heap: utilizarea lui sub forma de coadă cu priorităţi.

Coada cu priorităţi este o structură de date care conţine o mulţime S de elemente, fiecare având asociată o valoare numită *cheie*. Asupra unei cozi cu priorităţi se pot efectua următoarele operaţii:

Insereaza(S, x) insereaza elementul x în mulţimea S.Această operaţie este scrisă în felul următor $S \leftarrow S \cup \{x\}$.

ExtrageMax(S) elimină şi întoarce elementul din S având cheia cea mai mare.

O aplicaţie a cozilor de priorităţi o constituie planificarea lucrărilor pe calculatoare partajate. Lucrările care trebuie efectuate şi priorităţile relative se memorează într-o coadă de priorităţi. Când o lucrare este terminată sau întreruptă, funcţia **ExtrageMax** va selecta lucrarea având prioritatea cea mai mare dintre lucrările în aşteptare. Cu ajutorul funcţiei **Insereaza** oricând se introduce în coadă o sarcină nouă.

În mod natural o coadă cu priorităţi poate fi implementată utilizând un heap. Funcţia **Insereaza(**S, x**)** inserează un element în heap-ul S. La prima expandare a heap-ului se adaugă o frunză arborelui. Apoi se traversează un drum pornind de la această frunză către rădăcină în scopul găsirii locului definitiv al noului element.

Urmează instrucţiunile funcţiei **Insereaza(**S,x**)** în pseudocod.

dimensiune-heap[S**]←dimensiune-heap[**S**]+1**
i ←dimensiune-heap[S**]**
cat timp i>1 si S**[TATA(i)]<cheie executa**
S**[i]←** S**[TATA(i)]**
i←TATA[i]
S**[i]←cheie.**

ExtrageMax(S**)** este asemănătoare funcţiei **Heapsort**, instrucţiunile sale în pseudocod fiind:

daca dimensiune-heap[S**]<1 atunci**
eroare "depasire inferioara heap";
max← S**[1]**
S**[1]←** S**[dimensiune-heap[S]]**
dimensiune-heap[S**]←dimensiune-heap[**S**]-1**
ReconstituieHeap(S**,1)**
returneaza max.

Urmează scris in $C++$ un program care implementează o coadă cu priorităţi.

```
#include<iostream.h>
/*****************************************/
void ReconstituieHeap(int* A, int i, int dimensiune )
{ int a,maxim, stang=2*i+1,drept=stang+1;
if (stang<dimensiune&&A[stang]>A[i]) maxim=stang;
 else maxim=i;
if (drept<dimensiune&&A[drept]>A[maxim])
 maxim=drept;
if(maxim!=i){a=A[i];A[i]=A[maxim];A[maxim]=a;
ReconstituieHeap(A,maxim,dimensiune); }}
/*****************************************/
int ExtrageMax( int* A, int dim)
{int max;
if(dim<0) cout<<"Depasire inferioara heap"<<endl;
max=A[0];
```

```
A[0]=A[dim];
ReconstituieHeap( A,0,dim- - );
return max;}
/*****************************************/
void Insereaza(int* A, int cheie, int dimensiune )
{int i,tata;
i=dimensiune+1;
A[i]=cheie;tata=(i-1)/2;
while(i>0&&A[tata]<cheie)
{A[i]=A[tata];i=tata;A[i]=cheie;tata=(i-1)/2;} }
/*********************************************/
void main() {
char x,y;int cheie,dimensiune,max, S[100];
cout<<"Indicati primul element"<<endl;
cin>>max;
S[0]=max;
dimensiune=0;
cout<<"Intrerupem?[d/n]"<<endl;
cin>>x;
while(x!='d'){
cout<<"Extragem sau inseram[e/i]"<<endl;
cin>>y;
switch (y)
{case 'e':
max=ExtrageMax( S,dimensiune );
dimensiune=dimensiune-1;
if (dimensiune>=0) {cout<<"heap-ul ramas este:"<<endl;
for (int i=0;i<=dimensiune;i++) cout<<S[i]<<" ";
cout<<endl;
cout<<"Elementul maxim este = "<<max<<endl;
cout<<"dimensiunea"<<dimensiune<<endl;}
break;
default:cout<<"Introduceti cheia"<<endl;
cin>>cheie; Insereaza(S,cheie,dimensiune);
dimensiune=dimensiune+1;
cout<<"dimensiunea"<<dimensiune<<endl;
cout<<"heap-ul este:"<<endl;
for (int i=0;i<=dimensiune;i++) cout<<S[i]<<" ";
```

cout<<endl;}
cout<<"Intrerupem?[d/n]"<<endl;
cin>>x;}}

3.3 Sortarea rapidă

Sortarea rapidă este un algoritm care sortează pe loc (în spaţiul alocat şirului de intrare). Cu acest algoritm, un şir de n elemente este sortat într-un timp de ordinul $O(n^2)$, în cazul cel mai defavorabil. Algoritmul de sortare rapidă este deseori cea mai bună soluţie practică deoarece are o comportare medie remarcabilă: timpul său mediu de execuţie este $O(n \ln n)$ şi constanta ascunsă în formula $O(n \ln n)$ este destul de mică.

3.3.1 Descrierea algoritmului

Ca şi algoritmul de sortare prin interclasare, algoritmul de sortare rapidă ordonează un şir $A[p..r]$ folosind tehnica *divide şi stăpâneşte:*

Divide : Şirul $A[p..r]$ este rearanjat în două subşiruri nevide $A[p..q]$ şi $A[q+1..r]$ astfel încât fiecare element al subşirului $A[p..q]$ să fie mai mic sau egal cu fiecare element al subşirului $A[q+1..r]$. Indicele q este calculat de procedura de partiţionare.

Stăpâneşte: Cele două subşiruri $A[p..q]$ şi $A[q+1..r]$ sunt sortate prin apeluri recursive ale aalgoritmului de sortare rapidă.

Combină: Deoarece cele două subşiruri sunt sortate pe loc, şirul $A[p..r] = A[p..q] \cup A[q+1..r]$ este ordonat.

Algoritmul (intitulat **QUICKSORT**(A, p, r)) în pseudocod este următorul:

QUICKSORT(A, p, r)**:**
//urmează algoritmul
daca $p < r$ **atunci**
$q \leftarrow$**PARTITIE**(A, p, r)
QUICKSORT(A, p, q)
QUICKSORT$(A, q+1, r)$**.**

Pentru ordonarea şirului A se apelează **QUICKSORT**$(A, 1$**,lungime**$[A])$**.**

O altă funcţie a algoritmului este funcţia **PARTITIE**(A, p, r) care rearanjează pe loc şirul $A[p..r]$, returnând şi indicele q menţionat mai sus:

PARTITIE(A, p, r)
//urmează funcţia

x← $A[p]$
i← p
j← r
cat timp i<j executa
{repeta
j←j-1
pana cand A**[j]≤x**
repeta
i←i+1
pana cand A**[i]≥x**
daca i<j atunci
interschimba A**[i]↔** A**[j]; j←j-1}**
returneaza j.
Urmează programul scris în $C++$:

```
#include <iostream.h>
/************************/
int Partitie(int*A, int p, int r) {int x,y,i,j;
x=A[p];
i=p;
j=r;
while(i<j)
{ while(A[j]>x)j- - ;
while (A[i]<x) i+ +;
if(i<j){y=A[i];A[i]=A[j];A[j]=y;j- -;}}
return j;}
/*****************************/
void Quicksort(int *A, int p, int r)
{int q;
if (p<r)
{q= Partitie(A,p,r);
Quicksort(A,p,q);
Quicksort(A,q+1,r); }}
/*****************************/
void main()
{int *A,n;
cout<<"Introduceti numarul de elemente"<<endl;
cin>>n;
A=new int[n];
```

```
for(int i=0;i<n;i++)
{cout<<"A["<<i<<"]=";cin>>*(A+i);}
Quicksort(A,0,n-1);
for(i=0;i<n;i++)
cout<<" A["<<i<<"]="<<A[i];}
```

3.3.2 Performanţa algoritmului de sortare rapidă

Timpul de execuţie al algoritmului depinde de faptul că partiţionarea este echilibrată sau nu.

Cazul cel mai defavorabil

Vom demonstra că cea mai defavorabilă comportare a algoritmului de sortare rapidă apare în situaţia în care procedura de partiţionare produce un vector de $n-1$ elemente şi altul de 1 element. Mai întâi observăm că timpul de execuţie al funcţiei **Partitie** este $O(n)$. Fie $T(n)$ timpul de execuţie al algoritmului de sortare rapidă. Avem pentru partiţionarea amintită mai înainte, formula de recurenţă

$$T(n) = T(n-1) + O(n),$$

de unde rezultă

$$T(n) = \sum_{k=1}^{n} O(k) = O\left(\sum_{k=1}^{n} k\right) = O\left(n^2\right),$$

adică, pentru un $c > 0$ suficient de mare,

$$T(n) \leq cn^2. \tag{3.1}$$

Vom arăta prin inducţie că estimarea (3.1) este valabilă pentru orice partiţionare.

Să presupunem că partiţionare produce subvectori de dimensiuni q şi $n-q$. Cu ipoteza de inducţie avem

$$T(n) = T(q) + T(n-q) + O(n) \leq$$

$$\leq c \max_{1 \leq q \leq n-1} \left(q^2 + (n-q)^2\right) + O(n) = c\left(n^2 - 2n + 2\right) + O(n) \leq cn^2.$$

Cazul cel mai favorabil

Dacă funcţia de partiţionare produce doi vectori de $n/2$ elemente, algoritmul de sortare rapidă lucrează mult mai repede. Formula de recurenţă în acest caz

$$T(n) = 2T(n/2) + O(n),$$

conduce, după cum s-a arătat în cazul algoritmului de inserţie prin interclasare la un timp de execuţie de ordinul $O(n \ln n)$.

Estimarea timpului de execuţie mediu

Timpul de execuţie mediu se defineţe prin inducţie cu formula

$$T(n) = \frac{1}{n} \sum_{q=1}^{n-1} (T(q) + T(n-q)) + O(n).$$

Presupunem că

$$T(n) \leq an \ln n + b,$$

pentru un $a > 0$ şi $b > T(1)$.

Pentru $n > 1$ avem

$$T(n) = \frac{2}{n} \sum_{k=1}^{n-1} T(k) + O(n) \leq \frac{2}{n} \sum_{k=1}^{n-1} (ak \ln k + b) + O(n) =$$

$$= \frac{2a}{n} \sum_{k=1}^{n-1} k \ln k + \frac{2b}{n}(n-1) + O(n).$$

Tinând cont de inegalitatea

$$\sum_{k=1}^{n-1} k \ln k \leq \frac{1}{2} n^2 \ln n - \frac{1}{8} n^2,$$

obţinem

$$T(n) \leq \frac{2a}{n} \left(\frac{1}{2} n^2 \ln n - \frac{1}{8} n^2 \right) + \frac{2b}{n}(n-1) + O(n) \leq$$

$$\leq an \ln n - \frac{a}{4} n + 2b + O(n) = an \ln n + b + \left(O(n) + b - \frac{a}{4} n \right) \leq an \ln n + b,$$

deoarece valoarea lui a poate fi aleasă astfel încât $\frac{an}{4}$ să domine expresia $O(n) + b$. Tragem deci concluzia că timpul mediu de execuţie a algoritmului de sortare rapidă este $O(n \ln n)$.

3.4 Metoda bulelor (bubble method)

Principiul acestei metode de sortare este următorul: pornind de la ultimul element al şirului către primul, se schimba între ele fiecare element cu cel anterior, dacă elementul anterior (de indice mai mic) este mai mare. În felul acesta primul element al şirului este cel mai mic element. Se repetă procedura pentru şirul format din ultimele $n-1$ elemente şi aşa mai departe, obţinându-se în final şirul de n elemente ordonat. Numărul de comparaţii şi deci timpul de execuţie este de ordinul $O(n^2)$.

Urmează programul scris în $C++$.

```
#include <iostream.h>
//returneaza p,q in ordine crescătoare
/*****************************/
void Order(int *p,int *q) {
int temp;
if(*p>*q) {temp=*p; *p=*q; *q=temp; } }
//Bubble sorting
void Bubble(int *a,int n)
{int i,j;
for (i=0; i<n; i++)
for (j=n-1; i<j; j–)
Order(&a[j-1],&a[j]);}
//functia principala
void main()
int i,n=10; static int a[] = {7,3,66,3,-5,22,-77,2,36,-12};
cout<<"Sirul initial"<<endl;
for(i=0; i<n; i++)
cout<<a[i]<<" ";cout<<endl;
Bubble(a,n);
cout<<"Sirul sortat"<<endl;
for(i=0; i<n; i++)
cout<<a[i]<<" ";cout<<endl;
//Rezultatele obtinute
* Sirul initial * 7 3 66 3 -5 22 -77 2 36 -12 *
* Sirul sortat * -77 -12 -5 2 3 3 7 22 36 66 *
```

Capitolul 4

Tehnici de căutare

4.1 Algoritmi de căutare

Vom presupune că în memoria calculatorului au fost stocate un număr de n înregistrări şi că dorim să localizăm o anumită *înregistrare*. Ca şi în cazul sortării , presupunem că fiecare înregistrare conţine un câmp special, numit *cheie*. Colecţia tuturor înregistrărilor se numeşte *tabel* sau *fişier*. Un grup mai mare de fişiere poartă numele de *bază de date*.

În cazul unui *algoritm de căutare* se consideră un anumit argument K, iar problema este de a căuta acea înregistrare a cărei cheie este tocmai K. Sunt posibile două cazuri: *căutarea cu succes (reuşită)*, când înregistrarea cu cheia avută în vedere este depistată, sau *căutarea fără succes (nereuşită)*, când cheia niciunei înregistrări nu coincide cu cheia după care se face căutarea. După o căutare fără succes, uneori este de dorit să inserăm o nouă înregistrare, conţinând cheia K în tabel; metoda care realizează acest lucru se numeşte algoritmul de *căutare şi inserţie*.

Deşi scopul căutării este de a afla informaţia din înregistrarea asociată cheii K, algoritmii pe care îi vom prezenta în continuare, ţin în general cont numai de cheie, ignorând celelalte câmpuri.

În multe programe căutarea este partea care consumă cel mai mult timp, de aceea folosirea unui bun algoritm de căutare se reflectă în sporirea vitezei de rulare a programului.

Există de asemenea o importantă interacţiune între sortare şi căutare, după cum vom vedea în cele ce urmează.

4.1.1 Algoritmi de căutare secvenţială (pas cu pas)

Algoritmul S (căutare secvenţială). Fiind dat un tabel de înregistrări $R_1, R_2, ..., R_n$, $n \geq 1$, având cheile corespunzătoare $K_1, K_2, ..., K_n$, este căutată înregistrarea corespunzătoare cheii K. Vom mai introduce o înregistrare fictivă R_{n+1} cu proprietatea că valoarea cheii K_{n+1} este atât de mare încât K nu va căpăta nici-o dată această valoare (punem formal $K_{n+1} = \infty$). Urmează algoritmul scris în pseudocod

i←0
Executa
i←i+1
Cat timp i≤n si $K \neq K_i$
Daca $K = K_i$ **cautarea a avut succes**
Altfel, cautarea nu a avut succes.

În cazul în care cheile sunt ordonate crescător, o variantă a algoritmului de căutare secvenţială este

Algoritmul T (căutare secvenţială într-un tabel ordonat):
i←0
Executa
i←i+1
Cat timp i≤n si $K \leq K_i$
Daca $K = K_i$ **cautarea a avut succes**
Altfel, cautarea nu a avut succes.

Să notăm cu p_i probabilitatea ca să avem $K = K_i$, unde $p_1+p_2+...+p_n = 1$. Dacă probabilităţile sunt egale, în medie, algoritmul **S** consumă acelaşi timp ca şi algoritmul **T** pentru o căutare cu succes. Algoritmul **T** efectuează însă în medie de două ori mai repede căutările fără succes.

Să presupunem mai departe că probabilităţile p_i nu sunt egale. Timpul necesar unei căutări cu succes este proporţional cu numărul de comparaţii, care are valoarea medie

$$\overline{C}_n = p_1 + 2p_2 + ... + 3p_n.$$

Evident, $\overline{C}_n$ ia cea mai mică valoare atunci când $p_1 \geq p_2 \geq ... \geq p_n$, adică atunci când cele mai utilizate înregistrări apar la început. Dacă $p_1 = p_2 = ... = p_n = 1/n$, atunci

$$\overline{C}_n = \frac{n+1}{2}.$$

O repartiţie interesantă a probabilităţilor este *legea lui Zipf* care a observat că în limbajele naturale, cuvântul aflat pe locul n în ierarhia celor mai utilizate cuvinte apare cu o frecvenţă invers proporţională cu n:

$$p_1 = \frac{c}{1},\ p_2 = \frac{c}{2}, ..., p_n = \frac{c}{n},\ c = \frac{1}{H_n},\ H_n = 1 + \frac{1}{2} + ... + \frac{1}{n}.$$

Dacă legea lui Zipf guvernează frecvenţa cheilor într-un tabel, atunci

$$\overline{C}_n = \frac{n}{H_n}$$

iar căutarea într-o astfel de circumstanţă, este de circa $\frac{1}{2}\ln n$ ori mai rapidă decât căutarea în cazul general.

4.1.2 Căutarea în tabele sortate (ordonate)

În cele ce urmează vom discuta algoritmi de căutare pentru tabele ale căror chei sunt ordonate. După compararea cheii date K cu o cheie K_i a tabelului, căutarea continuă în trei moduri diferite după cum $K < K_i$, $K = K_i$ sau $K > K_i$. Sortarea tabelelor (listelor) este recomandabilă în cazul căutărilor repetate. De aceea în această subsecţiune vom studia metode de căutare în tabele ale căror chei sunt ordonate $K_1 < K_2 < ... < K_n$. După compararea cheilor K şi K_i într-o tabelă ordonată putem avea $K < K_i$ (caz în care $R_i, R_{i+1}, ..., R_n$ nu vor mai fi luate în consideraţie), $K = K_i$ (în acest caz căutarea se termină cu succes) sau $K > K_i$ (caz în care $R_1, R_2, ..., R_i$ nu vor mai fi luate în consideraţie).

Faptul că o căutare, chiar fără succes duce la eliminarea unora din cheile cu care trebuie comparată K, duce la o eficientizare a căutării.

Vom prezenta mai departe un algoritm general de căutare într-un tabel sortat. Fie $S = \{K_1 < K_2 < ... < K_n\}$ stocată într-un vector $K\,[1..n]$, adică $K\,[i] = K_i$ şi fie o cheie a. Pentru a decide dacă $a \in S$, comparăm a cu un element al tabelului şi apoi continuăm cu partea superioară sau cea inferioară a tabelului. Algoritmul (numit în cele ce urmează **algoritmul B**) scris în pseudocod este:

prim←1
ultim←n
urmator←un întreg în intervalul [prim,ultim]
executa

{daca a<K[urmator] atunci ultim←urmator-1
altfel prim←prim+1
urmator←un întreg în intervalul [prim,ultim]}
cat timp a≠K[urmator] si ultim >prim
daca a=K[urmator] atunci avem *cautare cu succes*
altfel avem *cautare fara succes.*

În cazul în care **un întreg în intervalul [prim,ultim]=prim** spunem că avem o *căutare liniară*.

Dacă **un întreg în intervalul [prim,ultim]=⌈(prim + ultim)/2⌉** spunem că avem o *căutare binară*.

Amintim că folosim notaţia $\lfloor\cdot\rfloor$ pentru partea întreagă a unui număr, în timp ce $\lceil\cdot\rceil$ are următoarea semnificaţie

$$\lceil a\rceil = \begin{cases} a \text{ dacă } a \text{ este } \hat{i}ntreg, \\ \lfloor a\rfloor + 1 \text{ dacă } a \text{ nu este } \hat{i}ntreg. \end{cases}$$

Prezentăm în continuare un proiect pentru căutarea într-o listă ordonată (cu relaţia de ordine lexicografică) de nume, pentru a afla pe ce poziţie se află un nume dat. Proiectul conţine programele **cautare.cpp**, **fcautare.cpp** (în care sunt descrise funcţiile folosite de cautare.cpp) şi fişierul de nume scrise în ordine lexicografică **search.dat**.

```
/**********cautare.cpp**********/
#include <stdio.h>
#include <string.h>
#include <io.h>
typedef char Str20[21]; //Nume de lungime 20
extern Str20 a[], cheie; //vezi fcautare.cpp
char DaNu[2], alegere[2];
int marime, unde, cate;
void GetList() {
FILE *fp;
fp=fopen("search.dat","r");
marime=0;
while (!feof(fp)) {
fscanf(fp, "s", a[marime]);
marime++;
}
fclose(fp);
```

```
printf("numarul de nume in lista = %d\n", marime}
//functii implementate in fcautare.cpp
void GasestePrimul(int, int *);
void GasesteToate(int, int *);
void Binar(int, int, int *);
void main() {
GetList(); // citeste numele din fisier
strcpy(DaNu,"d");
while (DaNu[0]=='d')
printf(" Ce nume cautati? "); scanf("%s", cheie);
//Se cere tipul cautarii
printf(" Secventiala pentru (p)rimul, (t)oate, ori (b)inara? ");
scanf("%s",alegere);
switch(alegere[0]) {
case 't':
GasesteToate(marime,
if (cate>0)
printf("%d aparitii gasite.\n", cate);
else
printf(" %s nu a fost gasit.\n", cheie);
break;
case 'b':
Binar(0,marime-1,
if (unde>0)
printf(" %s gasit la pozitia %d.\n", cheie, unde);
else
printf(" %s nu a fost gasit.\n", cheie);
break;
case 'p':
GasestePrimul(marime,
if (unde>0)
printf(" %s gasit la pozitia %d.\n", cheie, unde);
else
printf(" %s nu a fost gasit.\n", cheie);}
printf(" Inca o incercare (d/n)? "); scanf("%s, DaNu);
}
}
/******fcautare.cpp**************************/
```

```
/*************************************
!* Functii de cautare intr-o lista de nume *
!* ce urmeaza a fi folosite de programul Cautare.cpp. *
/*************************************/
#include <string.h>
#include <stdio.h>
typedef char Str20[21]; //Nume de lungimea 20
Str20 a[100], cheie;
void GasestePrimul(int marime, int *unde) {
// Cautare secventiala intr-o lista de nume pentru
// a afla prima aparitie a cheii.
int iun;
iun=0;
while (strcmp(a[iun],cheie)!=0
if (strcmp(a[iun],cheie)!=0) iun=-1;
*unde = iun+1;
}
void GasesteToate(int marime, int *cate) {
// Cautare secventiala intr-o lista de nume pentru
// a afla toate aparitiile cheii.
int cat, i;
cat=0;
for (i=0; i<marime; i++)
if (strcmp(a[i],cheie)==0) {
printf(" %s pe pozitia %d.\n", cheie, i + 1);
cat++;
}
*cate = cat;
}
void Binar(int prim, int ultim, int *unde) {
// Cautare binara intr-o lista ordonata pentru o aparitie
// a cheii specificate. Se presupune ca prim<ultim.
int urmator, pr, ul, iun;
pr=prim; ul=ultim; iun=-1;
while (pr <= ul
urmator=(pr+ul) / 2;
if (strcmp(a[urmator],cheie)==0)
iun=urmator;
```

```
else if (strcmp(a[urmator],cheie) > 0)
ul=urmator-1;
else
pr=urmator+1; }
*unde = iun+1;
}
```

4.1.3 Arbori de decizie asociaţi căutării binare

Pentru a înţelege mai bine ce se întâmplă în cazul algoritmului de căutare binară, vom construi *arborele de decizie asociat căutării binare*.

Arborele binar de decizie corespunzător unei căutări binare cu n înregistrări poate fi construit după cum urmează:

Dacă $n = 0$, arborele este format din frunza [0]. Altfel nodul rădăcină este $\lceil n/2 \rceil$, subarborele stâng este arborele binar construit asemănător cu $\lceil n/2 \rceil - 1$ noduri iar subarborele drept este arborele binar construit asemănător cu $\lfloor n/2 \rfloor$ noduri şi cu indicii nodurilor incrementaţi cu $\lceil n/2 \rceil$. Am avut în vedere numai nodurile interne corespunzătoare unei căutări cu succes.

Prezentăm mai jos un arbore de căutare binară pentru $n = 16$ (figura 4.1).

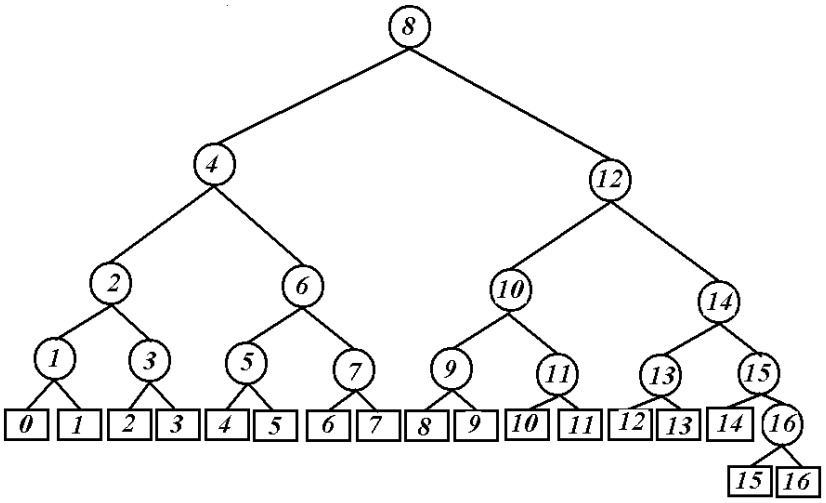


Figura 4.1: Arbore de cautare binară

Prima comparaţie efectuată este $K : K_8$ care este reprezentată de nodul (8) din figură. Dacă $K < K_8$, algoritmul urmează subarborele stâng iar dacă $K > K_8$, este folosit subarborele drept. O căutare fără succes va conduce la una din frunzele numerotate de la 0 la n; de exemplu ajungem la frunza [6] dacă şi numai dacă $K_6 < K < K_7$.

4.1.4 Optimalitatea căutării binare

Vom face observaţia că orice arbore binar cu n noduri interne (etichetate cu (1), (2), (3), ... (n))şi deci $n+1$ frunze (etichetate cu $[0]$, $[1]$, $[2]$, ... $[n-1]$, $[n]$), corespunde unei metode valide de căutare într-un tabel ordonat dacă parcurs în ordine simetrică obţinem $[0]$ (1) $[1]$ (2) $[2]$ (3) ... $[n-1]$ (n) $[n]$. Nodul intern care corespunde în **Algoritmul B** lui **urmator[prim,ultim]** va fi rădăcina subarborelui care are pe **[ultim]** drept cea mai din dreapta frunză iar pe **[prim]** drept cea mai din stânga frunză. De exemplu în figura 4.2, a) **urmator[0,4]=2, urmator[3,4]=4**, pe când în figura 4.2, b) **urmator[0,4]=1, urmator[3,4]=4**.

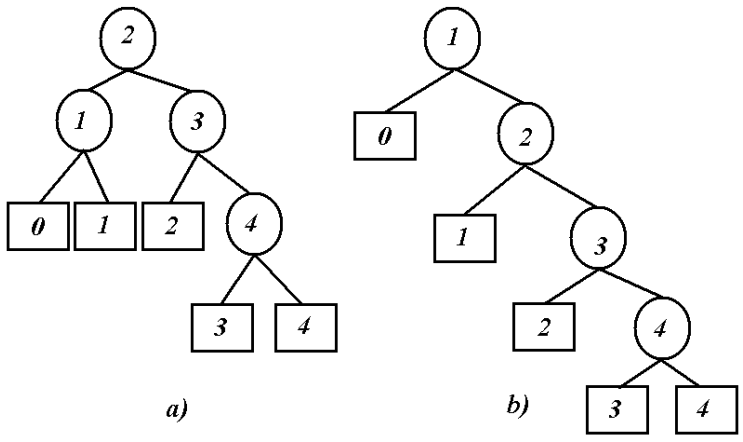


Figura 4.2: Arbori de cautare

Vom demonstra în cele ce urmează ca între arborii de decizie asociaţi algoritmului **B** de căutare, cei optimi sunt arborii corespunzători căutării binară. Vom introduce mai întâi două numere ce caracterizează arborele T:

$E(T)$, *lungimea drumurilor externe* ale arborelui reprezintă suma lungimilor drumurilor (numărul de muchii) care unesc frunzele arborelui cu rădăcina iar $I(T)$, *lungimea drumurilor interne* ale arborelui reprezintă suma lungimilor drumurilor care unesc nodurile interne cu rădăcina. De exemplu în figura 4.2, a) $E(T) = 2+2+2+3+3 = 12,\ I(T) = 1+1+2 = 4$ iar în figura 4.2, b) $E(T) = 1+2+3+4+4 = 14,\ I(T) = 1+2+3 = 6$. În continuare ne va fi necesară

Lema 3. *Dacă T este un arbore binar complet având N frunze, atunci $E(T)$ este minim dacă şi numai dacă toate frunzele lui T se află cel mult pe două nivele consecutive (cu $2^q - N$ frunze pe nivelul $q-1$ şi $2N - 2^q$ frunze pe nivelul q, unde $q = \lceil \log_2 N \rceil$, nivelul rădăcinii fiind 0).*

Demonstraţie. Să presupunem că arborele binar T are frunzele u şi v (fie x tatăl lor), pe nivelul L iar frunzele y şi z pe nivelul l astfel ca $L - l \geq 2$. Vom construi un alt arbore binar T_1 (vezi figura 4.3) transferând pe u şi v

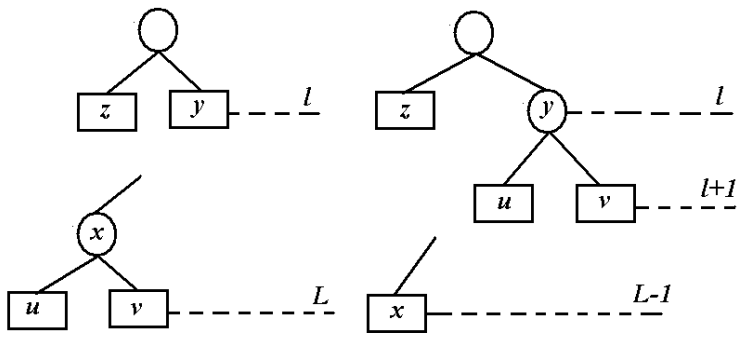


Figura 4.3: Optimizarea lungimii drumurilor externe

pe nivelul $l+1$ ca fii ai lui y; x devine frunză iar y nod intern. Rezultă că

$$E(T) - E(T_1) = 2L - (L-1) + l - 2(l+1) = L - l - 1 \geq 1,$$

şi deci T nu poate avea o lungime a drumurilor externe minimă. Deci T are toate frunzele pe un singur nivel dacă N este o putere a lui 2 sau, altfel, pe două nivele consecutive $q-1$ şi q, unde $q = \lceil \log_2 N \rceil$.

Medoda de construire a arborilor binari de decizie asociat algoritmului **B** conduce la

Lema 4. *Dacă* $2^{k-1} \leq n < 2^k$, *o căutare cu succes folosind algoritmul* **B** *necesită cel mult* k *comparaţii. Dacă* $n = 2^k - 1$, *o căutare fără succes necesită* k *comparaţii iar dacă* $2^{k-1} \leq n < 2^k - 1$, *o căutare fără succes necesită sau* $k-1$ *sau* k *comparaţii. Aceasta înseamnă că pentru* $n = 2^k - 1$ *toate frunzele arborelui binar asociat algoritmului* **B** *sunt pe nivelul* k *iar pentru* $2^{k-1} \leq n < 2^k - 1$ *frunzele se găsesc pe nivelele* $k-1$ *şi* k.

Demonstraţie. Lema este adevărată pentru $k = 1$. Fie $k \geq 2$ şi să presupunem (ipoteza de inducţie) că teorema este adevărată pentru orice $2^{k-1} \leq n < 2^k$. Dacă $2^k \leq n < 2^{k+1}$ vom avea două cazuri: (I) n este impar, adică $n = 2p + 1$; (II) n este par adică $n = 2p$, unde $p \in \mathbf{N}, p \geq 1$.

Cazul (I): Rădăcina arborelui binar T, corespunzător algoritmului **B** are eticheta $p + 1$ iar subarborele stâng T_l şi subarborele drept T_r conţin câte p noduri interne. Din relaţia

$$2^k \leq 2p + 1 < 2^{k+1},$$

rezultă că

$$2^{k-1} \leq p < 2^k.$$

Aplicând ipoteza de inducţie ambilor subarbori T_l şi T_r avem $h(T_l) = h(T_r) = k$ deci $h(T) = k + 1$, înălţimea lui T fiind numărul maxim de comparaţii ale cheilor efectuate de algoritmul **B** în cazul unei căutări cu succes. Dacă $p = 2^k - 1$ toate frunzele lui T_l şi T_r se află pe nivelul k, deci dacă $n = 2p + 1 = 2^{k+1} - 1$ toate frunzele lui T se vor afla pe nivelul $k + 1$. Dacă $2^{k-1} \leq p < 2^k - 1$, ceea ce implică $2^k \leq n < 2^{k+1} - 1$, cum (cu ipoteza de inducţie) atât T_l cât şi T_r au frunzele pe nivelele $k - 1$ sau k, deducem că T va avea frunzele pe nivelele k sau $k + 1$.

Cazul (II): În acest caz rădăcina arborelui binar are eticheta p, T_l are $p - 1$ noduri interne iar T_r are p noduri interne. Cum $2^k \leq 2p < 2^{k+1}$ rezultă că $2^{k-1} \leq p < 2^k$. Avem de asemenea $2^{k-1} \leq p - 1 < 2^k$ în afara cazului când $p = 2^{k-1}$ şi deci $n = 2^k$. În acest ultim caz arborele T este asemănător cu arborele din figura 4.1: el are $h(T) = k + 1$, $N - 1$ frunze pe nivelul k şi două frunze pe nivelul $k + 1$; teorema a fost demonstrată direct în această circumstanţă ($n = 2^k$). Ne mai rămâne să considerăm cazul $2^k < n < 2^{k+1}$. Cu ipoteza de inducţie deducem că $h(T_l) = h(T_r) = k$ deci $h(T) = k + 1$ iar algoritmul **B** necesită cel mult $k + 1$ comparaţii pentru o căutare cu succes.

Cum n este par, $n \neq 2^s - 1, s \geq 1$, avem de arătat că frunzele lui T se află pe nivelele k sau $k+1$. Aceasta rezultă din aplicarea ipotezei de inducţie la subarborii T_l şi T_r care au $p-1$ respectiv p noduri interne. Într-adevăr, cum $p-1 \neq 2^k - 1$ rezultă că frunzele lui T_l se află pe nivelele $k-1$ sau k. Pentru T_r toate frunzele se află fie pe nivelul k (dacă $p = 2^k - 1$) fie pe nivelele $k-1$ sau k. Rezultă că frunzele lui T se află pe nivelele k sau $k+1$.

Deducem că numărul de comparaţii în cazul unei căutări (cu sau fără succes) este de cel mult $\lfloor \log_2 N \rfloor + 1$.

Vom demonstra în continuare

Lemă 5. *Pentru orice arbore binar complet T este satisfăcută relaţia*

$$E(T) = I(T) + 2N,$$

unde N reprezintă numărul de noduri interne al lui T.

Demonstraţie. Să presupunem că arborele binar T are α_j noduri interne şi β_j noduri externe (frunze) la nivelul j; $j = 0, 1, ...$ (rădăcina este la nivelul 0). De exemplu în figura 4.2, a) avem $(\alpha_0, \alpha_1, \alpha_2, ...) = (1, 2, 1, 0, 0, ...)$, $(\beta_0, \beta_1, \beta_2, ...) = (0, 0, 3, 2, 0, 0, ...)$ iar în figura 4.2, b) avem $(\alpha_0, \alpha_1, \alpha_2, ...) = (1, 1, 1, 1, 0, 0, ...)$, $(\beta_0, \beta_1, \beta_2, ...) = (0, 1, 1, 1, 2, 0, 0, ...)$.

Considerăm funcţiile generatoare asociate acestor şiruri

$$A(x) = \sum_{j=0}^{\infty} \alpha_j x^j, \ B(x) = \sum_{j=0}^{\infty} \beta_j x^j,$$

unde numai un număr finit de termeni sunt diferiţi de 0. Este valabilă relaţia

$$2\alpha_{j-1} = \alpha_j + \beta_j, \ j = 0, 1, ...,$$

deoarece toate cele α_{j-1} noduri interne de la nivelul $j-1$ au fiecare în parte câte 2 fii pe nivelul k şi numărul total al acestor fii este $\alpha_j + \beta_j$. Rezultă de aici că

$$A(x) + B(x) = \sum_{j=0}^{\infty} (\alpha_j + \beta_j) x^j = \alpha_0 + \beta_0 + \sum_{j=1}^{\infty} (\alpha_j + \beta_j) x^j =$$

$$= 1 + 2\sum_{j=1}^{\infty} \alpha_{j-1} x^j = 1 + 2x \sum_{j=1}^{\infty} \alpha_{j-1} x^{j-1} = 1 + 2x \sum_{j=0}^{\infty} \alpha_j x^j,$$

adică

$$A(x) + B(x) = 1 + 2xA(x). \tag{4.1}$$

Pentru $x = 1$ se obţine $B(1) = 1 + A(1)$, dar $B(1) = \sum_{j=0}^{\infty} \beta_j$ este numărul de frunze ale lui T iar $A(1) = \sum_{j=0}^{\infty} \alpha_j$ este numărul de noduri interne, deci numărul de noduri interne este cu 1 mai mic decât numărul de nodduri externe. Derivând relaţia (4.1) obţinem

$$A'(x) + B'(x) = 2A(x) + 2xA'(x),$$
$$B'(1) = 2A(1) + A'(1).$$

Cum $A(1) = N$, $A'(1) = \sum_{j=0}^{\infty} j\alpha_j = I(T)$, $B'(1) = \sum_{j=0}^{\infty} j\beta_j = E(T)$, deducem relaţia

$$E(T) = I(T) + 2N. \tag{4.2}$$

Reprezentarea sub formă de arbore binar a algoritmului binar de căutare **B**, ne sugerează cum să calculăm într-un mod simplu numărul mediu de comparaţii. Fie C_N numărul mediu de comparaţii în cazul unei căutări reuşite şi fie C'_N numărul mediu de căutări în cazul unei încercări nereuşite. Avem

$$C_N = 1 + \frac{I(T)}{N},\ C'_N = \frac{E(T)}{N+1}. \tag{4.3}$$

Din (4.2) şi (4.3) rezultă

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1. \tag{4.4}$$

Rezultă că C_N este minim dacă şi numai dacă C'_N este minim, ori după cum am arătat mai înainte acest lucru se întâmplă atunci şi numai atunci când frunzele lui T se află pe cel mult două nivele consecutive. Cum lemei 2 arborele asociat căutării binare satisface această ipoteză, am demonstrat:

Teorema 6. *Căutarea binară este optimă în sensul că minimizează numărul mediu de comparaţii indiferent de reuşita căutării.*

4.2 Arbori binari de căutare

Am demonstrat în secţiunea precedentă că pentru o valoare dată n, arborele de decizie asociat căutării binare realizează numărul minim de comparaţii necesare căutării într-un tabel prin compararea cheilor. Metodele prezentate în secţiunea precedentă sunt potrivite numai pentru tabele de mărime fixă deoarece alocarea secvenţială a înregistrărilor face operaţiile de inserţie şi

ştergere foarte costisitoare. În schimb, folosirea unei structuri de arbore binar facilitează inserţia şi ştergerea înregistrărilor, făcând căutarea în tabel eficientă.

Definiţie: Un arbore binar de căutare pentru mulţimea

$$S = \{x_1 < x_2 < ... < x_n\}$$

este un arbore binar cu n noduri $\{v_1, v_2, ..., v_n\}$ *. Aceste noduri sunt etichetate cu elemente ale lui S, adică există o funcţie injectivă*

$$CONTINUT : \{v_1, v_2, ..., v_n\} \to S.$$

Etichetarea păstrează ordinea, adică în cazul în care v_i *este un nod al subarborelui stâng aparţinând arborelui cu rădăcina* v_k*,atunci*

$$CONTINUT\,(v_i) < CONTINUT\,(v_k)$$

iar în cazul în care v_j *este un nod al subarborelui drept aparţinând arborelui cu rădăcina* v_k*,atunci*

$$CONTINUT\,(v_k) < CONTINUT\,(v_j)\,.$$

O definiţie echivalentă este următoarea : *o traversare în ordine simetrică a unui arbore binar de căutare pentru mulţimea S reproduce ordinea pe S.*

Prezentăm mai jos un program de inserţie şi ştergere a nodurilor într-un arbore binar de căutare.

```
# include<iostream.h>
# include<stdlib.h>
int cheie;
struct nod{int inf; struct nod *st, *dr;} *rad;
/*****************************************/
void inserare(struct nod**rad)
{if(*rad==NULL)
{*rad=new nod;(*rad)->inf=cheie;
(*rad)->st=(*rad)->dr=NULL;return;}
if(cheie<(*rad)->inf) inserare(&(*rad)->st);
else if(cheie>(*rad)->inf) inserare(&(*rad)->dr);
else cout<<cheie<<" exista deja in arbore.";}
/*********************************************/
void listare(struct nod *rad,int indent)
```

```
{if(rad){listare(rad->st, indent+1);
cheie=indent;
while(cheie–) cout<<" ";
cout<<rad->inf;
listare(rad->dr,indent+1);}}
/*********************************************/
void stergere(struct nod**rad)
{struct nod*p,*q;
if(*rad==NULL)
{cout<<"Arborele nu contine "<<cheie<<endl;return;}
if(cheie<(*rad)->inf) stergere(&(*rad)->st);
if(cheie>(*rad)->inf) stergere(&(*rad)->dr);
if(cheie==(*rad)->inf)
{if((*rad)->dr==NULL)
{q=*rad;*rad=q->st; delete q;}
else
if((*rad)->st==NULL)
{q=*rad;*rad=q->dr; delete q;}
else
{for(q=(*rad),p=(*rad)->st;p->dr;q=p,p=p->dr);
(*rad)->inf=p->inf;
if((*rad)->st==p)(*rad)->st=p->st;
else q->dr=p->st; delete p;}}}
/*********************************************/
void main()
{rad=new nod;
cout<<"Valoarea radacinii este:";cin>>rad->inf;
rad->st=rad->dr=NULL;
do{
cout<<"Operatia:Listare(1)/Inserare(2)/Stergere(3)/Iesire(0)";
cout<<endl;
cin>>cheie;if(!cheie) return;
switch(cheie)
{case 1:listare(rad,1);cout<<endl; break;
case 2: cout<<"inf=";cin>>cheie;inserare(&rad);break;
case 3: cout<<"inf=";cin>>cheie;stergere(&rad);break;}
}while(rad);
cout<<"Ati sters radacina"<<endl;}
```

După cum se observă din program, ideea inserţiei în arborele binar de căutare este următoarea: dacă arborele este vid, se crează un arbore având drept unic nod şi rădăcină nodul inserat; altfel se compară cheia nodului inserat cu cheia rădăcinii. Dacă avem cheia nodului inserat mai mică decât cheia rădăcinii, se trece la subarborele stâng şi se apelează recursiv procedura de inserare, altfel se trece la subarborele drept şi se apelează recursiv procedura.

Procedura prezentată în program de stergere a unui nod din arborele binar de căutare este de asemenea recursivă. În cazul în care cheia nodului ce urmează a fi şters este mai mică decât cheia rădăcinii, se trece la subarborele stâng şi se apelează recursiv procedura de ştergere; altfel se trece la subarborele drept şi se apelează recursiv procedura de ştergere. În cazul în care nodul ce urmează a fi şters este chiar rădăcina vom avea mai multe posibilităti: a) subarborele drept este vid: se şterge rădăcină iar fiul stâng al rădăcinii devine noua rădăcină; b) subarborele stâng este vid: se şterge rădăcină iar fiul drept al rădăcinii devine noua rădăcină; c) rădăcina are ambii fii; în acest se şterge cel mai din dreapta descendent al fiului stâng al rădăcinii iar informaţia (cheia) acestuia înlocuieşte informaţia (cheia) rădăcinii, fiul stâng al nodului eliminat devine totodată fiul drept al tatălui nodului eliminat. Dacă fiul stâng al rădăcinii nu are fiu drept, atunci el este cel eliminat, informaţia lui înlocuieşte informaţia rădăcinii iar fiul lui stâng devine fiul stâng al rădăcinii (figura 4.4).

Algoritmul de căutare, după o anumită cheie, într-un arbore de căutare este în esenţă următorul: comparăm cheia înregistrării căutate cu cheia rădăcinii. Dacă cele două chei coincid căutarea este reuşită. Dacă cheia înregistrării este mai mică decât cheia rădăcinii continuăm căutarea în subarborele stâng iar dacă este mai mare în subarborele drept.

De foarte multe ori este util să prezentăm arborii binari de căutare ca în figura 4.5.

Aici arborele binar de căutare este prezentat ca un arbore complet în care informaţiile (cheile) sunt stocate în cele N noduri interne iar informaţia fiecărui nod extern (frunză) este intervalul deschis dintre două chei consecutive, astfel încât, dacă x_i, x_{i+1} sunt două chei consecutive şi vrem să căutăm nodul ce conţine cheia a cu $a \in (x_i, x_{i+1})$ să fim conduşi aplicând algoritmul de căutare (într-un arbore binar de căutare) la frunza etichetată cu (x_i, x_{i+1}).

Vom arăta că înălţimea medie a unui arbore binar de căutare este de ordinul $O(\ln\ N)$ (N este numărul nodurilor interne) şi căutarea necesită în medie circa $2 \ln\ N \approx 1,386 \log_2 N$ comparaţii în cazul în care cheile sunt

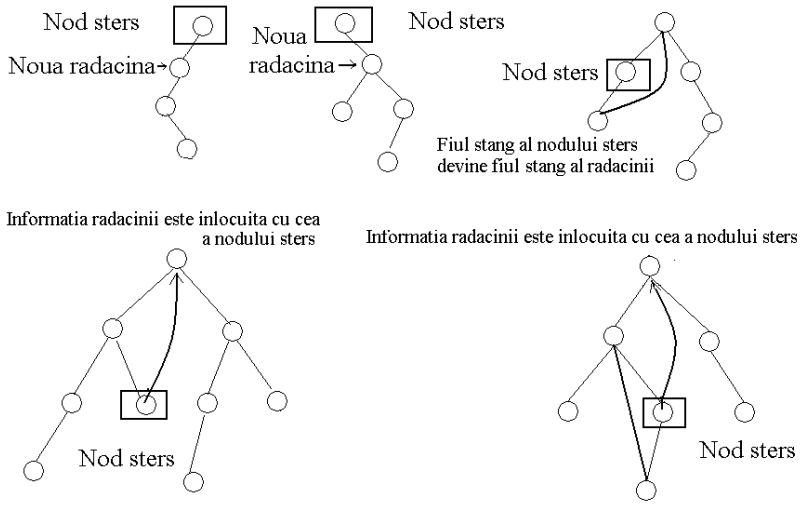


Figura 4.4: Ştergerea rădăcinii unui arbore binar de căutare

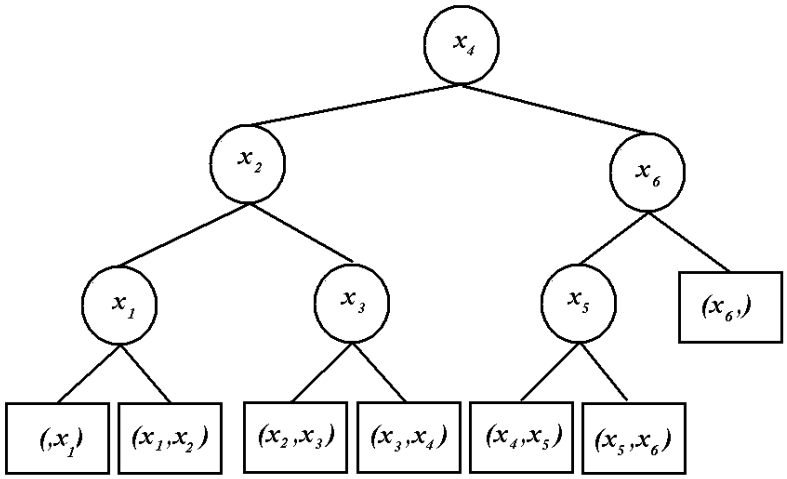


Figura 4.5: Arbore binar de căutare

inserate în arbore în mod aleator. Să presupunem că cele $N!$ ordonări posibile ale celor N chei corespund la $N!$ modalităţi de inserţie. Numărul de comparaţii necesare pentru a găsi o cheie este exact cu 1 mai mare decât numărul de comparaţii efectuate atunci când cheia a fost inserată în arbore. Notând cu C_N numărul mediu de comparaţii pentru o căutare reuşită şi cu C'_N numărul mediu de comparaţii pentru o căutare nereuşită avem

$$C_N = 1 + \frac{C'_0 + C'_1 + ... + C'_{N-1}}{N},$$

pentru că înainte de reuşita căutării vom avea căutări nereuşite în mulţimi de $0, 1, ..., n-2$ sau $n-1$ elemente. Tinând cont şi de relaţia

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1,$$

deducem

$$(N+1)\, C'_N = 2N + C'_0 + C'_1 + ... + C'_{N-1}.$$

Scăzând din această ecuaţie următoarea ecuaţie

$$NC'_{N-1} = 2\,(N-1) + C'_0 + C'_1 + ... + C'_{N-2}$$

obţinem

$$(N+1)\, C'_N - NC'_{N-1} = 2 + C'_{N-1} \Rightarrow C'_N = C'_{N-1} + \frac{2}{N+1}.$$

Cum $C'_0 = 0$ deducem că

$$C'_N = 2H_{N+1} - 2,$$

de unde

$$C_N = 2\left(1 + \frac{1}{N}\right) H_{N+1} - 3 - \frac{2}{N} \sim 2 \ln N.$$

4.3 Arbori de căutare ponderaţi (optimali)

În cele ce urmează vom asocia fiecărui element din mulţimea ordonată S câte o pondere (probabilitate). Ponderile mari indică faptul că înregistrările corespunzătoare sunt importante şi frecvent accesate; este preferabil de aceea

ca aceste elemente să fie cât mai aproape de rădăcina arborelui de căutare pentru ca accesul la ele să fie cât mai rapid.

Să analizăm în continuare problema găsirii unui arbore optimal. De exemplu, Fie $N = 3$ şi să presupunem că următoarele chei $K_1 < K_2 < K_3$ au probabilităţie p, q respectiv r. Există 5 posibili arbori binari de căutare având aceste chei drept noduri interne (figura 4.6).

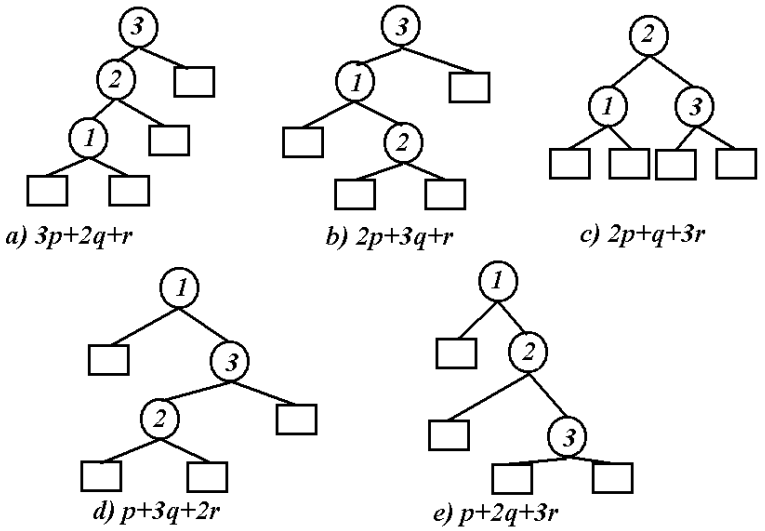


Figura 4.6: Arbori posibili de cautare şi numărul mediu de comparaţii pentru o căutare reuşită

Obţinem astfel 5 expresii algebrice pentru numărul mediu de comparaţii într-o căutare. Când N este mare, este foarte costisitor să construim toţi arborii de căutare pentru a vedea care din ei este cel optim. Vom pune de aceea în evidenţă un algoritm de găsire al acestuia.

Fie $S = \{K_1 < K_2 < ... < K_N\}$. Fie $p_i \geq 0, i = 1, ..., N$ probabilitatea de căutare a cheii $a = K_i$ şi $q_j \geq 0, j = 0, 1, ...N$ probabilitatea de căutare a cheii $a \in (K_j, K_{j+1})$ (punem $K_0 = -\infty$ şi $K_{N+1} = \infty$). Avem deci

$$\sum_{i=1}^{N} p_i + \sum_{j=0}^{N} q_j = 1.$$

$(2N + 1)$ - tuplul $(q_0, p_1, q_1, ..., p_N, q_N)$ se numeşte *distribuţia probabilităţilor (ponderilor) de căutare (acces). Fie* T un arbore de căutare pentru S, fie α_i^T

adâncimea (nivelul) nodului intern i (al i - lea nod intern în ordine simetrică) şi fie β_j^T adâncimea frunzei j (al $(j+1)-$ lea nod extern sau frunza (K_j, K_{j+1})).

Să considerăm o căutare a cheii a. Dacă $a = K_i$, vom compara a cu $\alpha_i^T + 1$ elemente din arbore; dacă $K_j < a < K_{j+1}$, atunci vom compara a cu β_j^T elemente din arbore. Aşadar

$$POND(T) = \sum_{i=1}^{N} p_i \left(\alpha_i^T + 1\right) + \sum_{j=0}^{N} q_j \beta_j^T,$$

este *numărul mediu de comparaţii pentru o căutare*. $POND(T)$ este *lungimea ponderată a drumurilor arborelui* T (sau *costul lui* T *relativ la o distribuţie dată a probabilităţilor de căutare*).

Vom considera $POND\,(T)$ drept indicator de bază pentru eficienţa operaţiei de căutare (acces) deoarece numărul aşteptat de comparaţii într-o căutare va fi proporţional cu $POND(T)$. De exemplu, în cazul arborelui din figura 4.6 b) (unde în loc de p, q, r punem p_1, p_2, p_3) avem

$$\alpha_1 = 1, \alpha_2 = 2\alpha_3 = 0, \beta_0 = 2, \beta_1 = 3, \beta_2 = 3, \beta_3 = 1$$

şi deci

$$POND(T) = 2q_0 + 2p_1 + 3q_1 + 3p_2 + 3q_2 + p_3 + q_3.$$

(Vom omite indicele T când este clar din context la ce arbore ne referim.)

Definiţie. Arborele de căuare T *peste mulţimea ordonată* S *cu distribuţia ponderilor de căutare* $(q_0, p_1, q_1, ..., p_N, q_N)$, *este* **optimal** *dacă* $POND\,(T)$ *(costul arborelui sau lungimea ponderată a drumurilor arborelui) este minim în raport cu costurile celorlalţi arbori de căutare peste* S.

Vom prezenta mai departe un algoritm de construire a unui arbore de căutare optimal. Fie un arbore de căutare peste S având nodurile interne etichetate cu $1, 2, ..., N$ (corespunzător cheilor $K_1, ..., K_N$) şi frunzele etichetate cu $0, 1, ..., N$ (corespunzând lui $(, K_1), (K_1, K_2), ..., (K_{N-1}, K_N), (K_N,)$). Un subarbore al acestuia ar putea avea nodurile interne $i+1, ..., j$ şi frunzele $i, ..., j$ pentru $0 \leq i, j \leq n,\ i < j$. Acest subarbore este la rândul său arbore de căutare pentru mulţimea cheilor $\{K_{i+1} < ... < K_j\}$. Fie k eticheta rădăcinii subarborelui.

Fie costul acestui subarbore$POND\,(i, j)$şi *greutatea sa*:

$$GREUT\,(i, j) = p_{i+1} + ... + p_j + q_i + ... + q_j,$$

de unde rezultă imediat că

$$GREUT\,(i,j) = GREUT\,(i,j-1) + p_j + q_j,\ GREUT\,(i,i) = q_i.$$

Avem relaţia

$$POND\,(i,j) = GREUT\,(i,j) + POND\,(i,k-1) + POND\,(k,j)\,.$$

Într-adevăr, subarborele stâng al rădăcini k are frunzele $i, i+1, ..., k-1$, iar subarborele drept are frunzele $k, k+1, ..., j$, şi nivelul fiecărui nod din subarborele drept sau stâng este cu 1 mai mic decât nivelul aceluiaşi nod în arborele de rădăcină k. Fie $C\,(i,j) = \min POND\,(i,j)$ costul unui subarbore optimal cu ponderile $\{p_{i+1}, ..., p_j; q_i, ..., q_j\}$. Rezultă atunci pentru $i < j$:

$$C\,(i,j) = GREUT\,(i,j) + \min_{i<k\leq j}\left(C\,(i,k-1) + C\,(k,j)\right),$$

$$C\,(i,i) = 0.$$

Pentru $i = j+1$ rezultă imediat că

$$C\,(i,i+1) = GREUT\,(i,i+1)\,,\ k = i+1.$$

Plecând de la aceste relaţii prezentăm mai jos un program, scris în limbajul C de construire a unui arbore optimal. Câmpul informaţiei fiecărui nod conţine un caracter (literă) iar acestea se consideră ordonate după ordinea citirii.

```
/*******************************************/
#include<stdio.h>
#include<stdlib.h>
#include <io.h>
# define N 25
struct nod {char ch; struct nod *st,*dr;} *rd;
char chei[N];
//cheile de cautare se considera ordonate dupa ordinea citirii
int i,nr;
int p[N-1];/*ponderile cheilor*/
int q[N];
/*ponderile informaţiilor aflate intre 2 chei consecutive*/
int c[N][N], greut[N][N], rad[N][N];
FILE*f;
```

```
/****Functia de calcul a greutatii si costului**********/
void calcul()
{int x,min,i,j,k,h,m;
for(i=0;i<=nr;i++)
{greut[i][i]=q[i];
for(j=i+1;j<=nr;j++)
greut[i][j]=greut[i][j-1]+p[j]+q[j];}
for(i=0;i<=nr;i++) c[i][i]=q[i];
for(i=0;i<nr;i++)
{j=i+1;
c[i][j]=greut[i][j];
rad[i][j]=j;}
for(h=2;h<=nr;h++)
for(i=0;i<=nr-h;i++)
{j=i+h;m=rad[i][j-1];
min=c[i][m-1]+c[m][j];
for(k=m+1;k<=rad[i+1][j];k++)
{x=c[i][k-1]+c[k][j];
if(x<min)m=k;min=x;}}
c[i][j]=min+greut[i][j];
rad[i][j]=m;}}
/****Functia de generare a arborelui optimal****/
struct nod *arbore(int i, int j)
{struct nod *s;
if(i==j) s=NULL;
else{s=new nod;
s->st=arbore(i,rad[i][j]-1);
s->ch=chei[rad[i][j]];
s->dr=arbore(rad[i][j],j);}
return s;}
/****** Functia de listare indentata a nodurilor arborelui*****/
void listare(struct nod*r, int nivel)
{int i;
if(r){ listare(r->dr,nivel+1);i=nivel;
while(i–) printf(" ");
printf("%c\n", r->ch);
listare(r->st, nivel+1);}}
\ ******* Functiaprincipala ********************** \
```

```
void main() {f=fopen("arboptim.dat","r");
fscanf(f,"%d\n",&nr);
if(nr>0)
{fscanf(f,"%d\n",&q[0]);
for(i=1;i<=nr;i++)
fscanf(f,"%c %d\n%d\n",&chei[i],&p[i],&q[i]);
calcul();
 printf("Lungimea medie a unei cautari: %f\n",
(float)c[0][nr]/greut[0][nr]);
struct nod*radacina=arbore(0,nr);
listare(radacina,0);}}
/*************************************************/
```

Fişierul **arboptim.dat** conţine pe prima linie numărul de noduri interne ale arborelui, pe a doua linie valoarea ponderii q_0 iar pe celelalte linii cheile K_i cu ponderile p_i şi q_i. Un exemplu de astfel de fişier este următorul:

```
/*****************arboptim.dat**********************/
5
1
a 0 2
b 1 1
c 1 0
f 2 2
e 3 0
d 1 2
/*************************************************/
```

4.4 Arbori echilibraţi

Inserţia de noi noduri într-un arbore binar de căutare poate conduce la arbori dezechilibraţi în care între înălţimea subarborelui drept al unui nod şi înălţimea subarborelui stâng să fie o mare diferenţă. Într-un astfel de arbore acţiunea de căutare va consuma mai mult timp.

O soluţie la problema menţinerii unui bun arbore de căutare a fost descoperită de G. M. Adelson-Velskii şi E. M. Landis în 1962 care au pus în evidenţă aşa numiţii *arbori echilibraţi* (sau *arbori AVL*).

Definiţie. Un arbore binar este **echilibrat (AVL)** *dacă înălţimea subarborelui stâng al oricărui nod nu diferă mai mult decât* ± 1 *de înălţimea subarborelui său drept.*

Definiţie. Diferenţa dintre înălţimea subarborelui drept şi înălţimea subarborelui stâng poartă numele de **factor de echilibru** *al unui nod.*

Aşadar, dacă un arbore binar este echilibrat, atunci factorul de echilibru al fiecărui nod este $1, 0$ sau -1.

4.4.1 Arbori Fibonacci

O clasă importantă de arbori echilibraţi este clasa de *arbori Fibonacci* de care ne vom ocupa în continuare.

Să considerăm mai întâi sirul $(F_n)_{n\geq 1}$ de numere Fibonacci, definite prin următoarea formulă de recurenţă:

$$\begin{gathered} F_1 = F_2 = 1, \\ F_{n+2} = F_{n+1} + F_n,\ n \geq 1. \end{gathered} \tag{4.5}$$

Primii termeni din şirul lui Fibonacci sunt $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$.

Pentru a găsi o formulă explicită pentru numerele Fibonacci, vom căuta o soluţie a recurenţei (4.5) de forma $F_n = r^n$. Rezultă că r satisface ecuaţia algebrică de gradul 2 :

$$r^2 - r - 1 = 0,$$

cu soluţia

$$r_{1,2} = \frac{1 \pm \sqrt{5}}{2}.$$

Soluţia generală va fi

$$F_n = Ar_1^n + Br_2^n.$$

Constantele A şi B vor fi determinate din condiţiile $F_1 = F_2 = 1$ care conduc la sistemul algebric

$$A\frac{1+\sqrt{5}}{2} + B\frac{1-\sqrt{5}}{2} = 1,$$

$$A\frac{3+\sqrt{5}}{2} + B\frac{3-\sqrt{5}}{2} = 1,$$

cu soluţia

$$A = \frac{\sqrt{5}}{5},\ B = -\frac{\sqrt{5}}{5}.$$

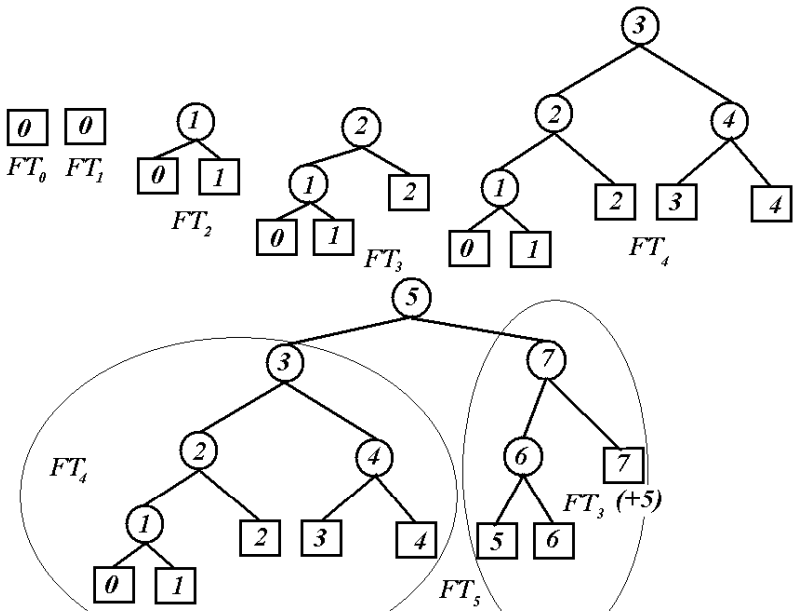


Figura 4.7: Arbori Fibonacci

Obţinem astfel *formula lui Binet* pentru numerele Fibonacci:

$$F_n = \frac{\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^n .$$

.

Arborii binari Fibonacci, notaţi cu $FT_k, k = 0, 1, 2, ...$ sunt definiţi prin recurenţă după cum urmează:

- FT_0 şi FT_1 sunt formaţi fiecare dintr-un singur nod extern etichetat cu $[0]$;

- pentru $k \geq 2$, arborele Fibonacci de ordin k, FT_k are rădăcina etichetată cu F_k; subarborele stâng al rădăcinii este FT_{k-1} iar subarborele drept al rădăcinii este arborele FT_{k-2} cu etichetele tuturor nodurilor incrementate cu F_k (eticheta rădăcinii lui FT_k) (vezi figura 4.7).

Vom pune mai departe în evidenţă câteva proprietăţi ale arborilor Fibonacci.

Lemă. *Pentru orice $k \geq 1$, arborele Fibonacci FT_k este un arbore echilibrat având înălţimea $h(FT_k) = k-1$, F_{k+1} frunze şi $F_{k+1}-1$ noduri interne.*

*Demonstraţie.*Pentru $k = 1$ şi $k = 2$ proprietatea este verificată. Să presupunem că proprietatea este adevărată pentru toţi arborii Fibonacci $FT_{k'}$ cu $k' < k$. Fie FT_k un arbore Fibonacci de ordin $k > 3$. Din definiţia recursivă obţinem că $h(FT_k) = h(FT_{k-1}) + 1 = k - 1$, numărul de frunze al lui FT_k este egal cu $F_k + F_{k-1} = F_{k+1}$ iar numărul de noduri interne este $1 + (F_k - 1) + (F_{k-1} - 1) = F_{k+1} - 1$. Din ipoteza de inducţie, proprietatea de echilibrare este satisfăcută pentru toate nodurile cu excepţia rădăcinii. În ceea ce priveşte rădăcina, subarborele stâng are înălţimea $k-2$ iar subarborele drept are înălţimea $k - 3$, deci FT_k este echilibrat.

4.4.2 Proprietăti ale arborilor echilibraţi

Arborii echilibraţi reprezintă o treaptă intermediară între clasa arborilor optimali (cu frunzele aşezate pe două nivele adiacente) şi clasa arborilor binari arbitrari. De aceea este firesc să ne întrebăm cât de mare este diferenţa dintre un arbore optimal şi un arbore echilibrat; prezentăm în acest context următoarea teoremă:

Teoremă. *Înălţimea unui arbore echilibrat T cu N noduri interne se află întotdeauna între* $\log_2(N+1)$ *şi* $1.4404 \log_2(N+2) - 0.328$.

Demonstraţie. Un arbore binar de înălţime h are cel mult $2^h - 1$ noduri interne; deci

$$N \leq 2^{h(T)} - 1 \Rightarrow h\left(T\right) \geq \log_2\left(N+1\right).$$

Pentru a găsi limitarea superioară a lui $h\left(T\right)$, ne vom pune problema aflării numărului minim de noduri interne conţinute într-un arbore echilibrat de înălţime h. Fie deci T_h arborele echilibrat de înălţima h cu cel mai mic număr de noduri posibil; unul din subarborii rădăcinii, de exemplu cel stâng va avea înălţimea $h-1$ iar celălalt subarbore va avea înălţimea $h-1$ sau $h-2$. Cum T_h are numărul minim de noduri, va trebui să considerăm că subarborele stâng al rădăcinii are înălţimea $h-1$ iar subarborele drept are înălţimea $h-2$.Putem aşadar considera că subarborele stâng al rădăcinii este T_{h-1} iar subarborele drept este T_{h-2}. În virtutea acestei consideraţii, se demonstrează prin inducţie că arborele Fibonacci FT_{h+1} este arborele T_h căutat, în sensul că între toţi arborii echilibraţi de înălţime impusă h, acesta are cel mai mic număr de noduri. Conform lemei precedente avem

$$N \geq F_{h+2} - 1 = \frac{\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - \frac{\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^{h+2} - 1.$$

Cum

$$-\frac{\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^{h+2} > -1,$$

rezultă că

$$\log_2\left(N+2\right) > \left(h+2\right)\log_2\left(\frac{1+\sqrt{5}}{2}\right) - \frac{1}{2}\log_2 5 \Rightarrow$$

$$\Rightarrow h < 1.4404\log_2\left(N+2\right) - 0.328.$$

Din consideraţiile făcute pe parcursul demonstraţiei teoremei, rezultă următorul

Corolar. *Între arborii echilibraţi cu un număr dat de noduri, arborii Fibonacci au înăltimea maximă*, deci sunt cei mai puţin performanţi.

4.5 Inserţia unui nod într-un arbore echilibrat

Inserarea unui nou nod se efectuează cu algoritmul cunoscut de inserare a nodurilor într-un arbore binar de căutare. După inserare însă, va trebui să *re-echilibrăm* arborele dacă vom ajunge într-una din următoarele situaţii:

- subarborelui stâng al unui nod cu factorul de echilibru -1 îi creşte înălţimea;

- subarborelui drept al unui nod cu factorul de echilibru 1 îi creşte înălţimea.

Ambele cazuri sunt tratate apelând la *rotaţii* (pe care le vom descrie în cele ce urmează). Rotaţiile implică doar modificări ale legăturilor în cadrul arborelui, nu şi operaţii de căutare, de aceea timpul lor de execuţie este de ordinul $O(1)$.

4.5.1 Rotaţii în arbori echilibraţi

Rotaţiile sunt operaţii de schimbare între ele a unor noduri aflate în relaţia de tată-fiu (şi de refacere a unor legături) astfel încât să fie păstrată structura de arbore de căutare. Astfel, printr-o *rotaţie simplă a unui arbore la stânga*, fiul drept al rădăcinii iniţiale devine noua rădăcină iar rădăcina iniţială devine fiul stâng al noii rădăcini. Printr-o *rotaţie simplă a unui arbore la dreapta*, fiul stâng al rădăcinii iniţiale devine noua rădăcină iar rădăcina iniţială devine fiul drept al noii rădăcini. O *rotaţie dublă la dreapta* afectează două nivele: fiul drept al fiului stâng al rădăcinii iniţiale devine noua rădăcină, rădăcina iniţială devine fiul drept al noii rădăcini iar fiul stâng al rădăcinii iniţiale devine fiul stâng al noii rădăcini. O *rotaţie dublă la stânga* afectează de asemenea două nivele: fiul stâng al fiului drept al rădăcinii iniţiale devine noua rădăcină, rădăcina iniţială devine fiul stâng al noii rădăcini iar fiul drept al rădăcinii iniţiale devine fiul drept al noii rădăcini.

Vom studia cazurile de dezechilibru ce pot apărea şi vom efectua re-echilibrarea prin rotaţii.

Cazul 1. Creşte înălţimea subarborelui stâng al nodului a care are factorul de echilibru iniţial -1.

a) Factorul de echilibru al fiului stâng b al lui a este -1 (figura 4.8). Aceasta înseamnă că noul element a fost inserat în subarborele A. Re-echilibrarea necesită o rotaţie simplă la dreapta a perechii tată-fiu ($a > b$).

b) Factorul de echilibru al fiului stâng b al lui a este 1.

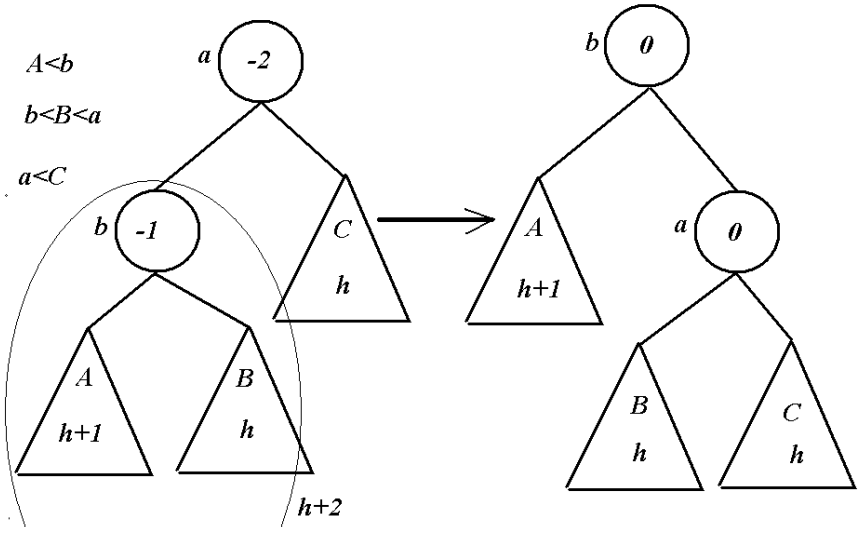


Figura 4.8: Rotaţie simplă la dreapta pentru re-echilibrare

b.1) Factorul de echilibru al fiului drept c al lui b este 1 (figura 4.9). Aceasta înseamnă că noul element a fost inserat în subarborele C. Pentru re-echilibrare vom avea nevoie de o rotaţie dublă la dreapta a nodurilor $b < c < a$.

b.2) Factorul de echilibru al fiului drept c al lui b este -1. Se tratează ca şi cazul b.1) (figura 4.10). .

b.3) Factorul de echilibru al fiului drept c al lui b este 0. Dezechilibrarea este imposibilă.

c) Factorul de echilibru al fiului stâng b al lui a este 0. Dezechilibrarea este imposibilă.

Cazul I1. Creşte înălţimea subarborelui drept al nodului a care are factorul de echilibru iniţial 1. Se tratează asemănător cu cazul I).

a) Factorul de echilibru al fiului stâng b al lui a este 1 (figura 4.11). Aceasta înseamnă că noul element a fost inserat în subarborele C. Re-echilibrarea necesită o rotaţie simplă la stânga a perechii tată-fiu ($a < b$).

b) Factorul de echilibru al fiului drept b al lui a este -1.

b.1) Factorul de echilibru al fiului stâng c al lui b este -1(figura 4.12). Aceasta înseamnă că noul element a fost inserat în subarborele B. Pentru re-echilibrare vom avea nevoie de o rotaţie dublă la stânga a nodurilor $b > c > a$.

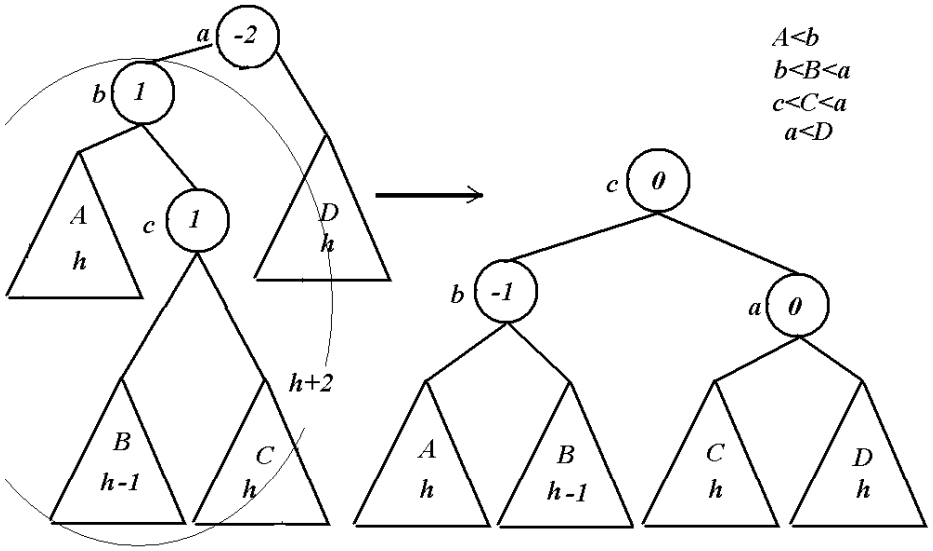


Figura 4.9: Rotaţie dublă la dreapta pentru re-echilibrare

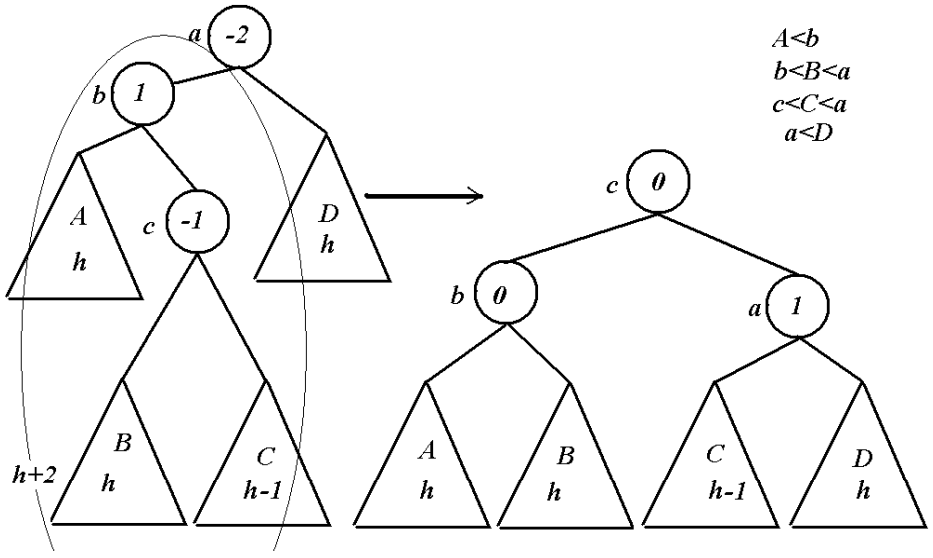


Figura 4.10: Rotaţie dublă la dreapta pentru re-echilibrare

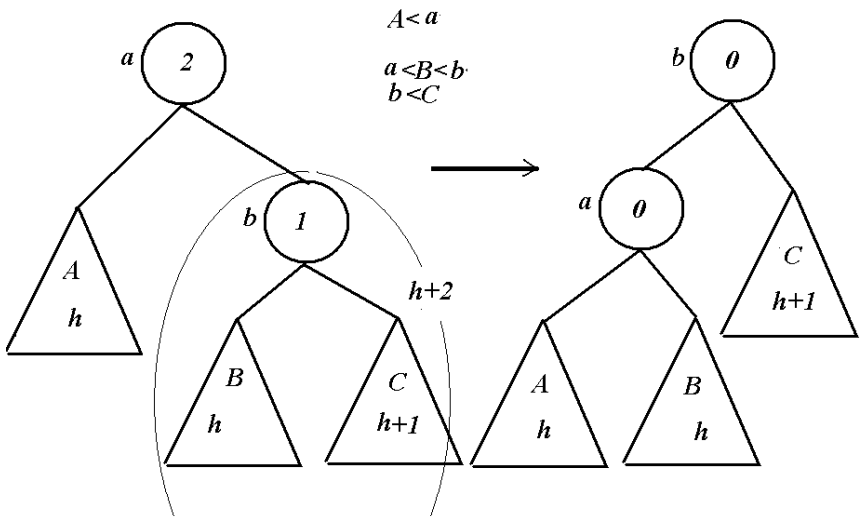


Figura 4.11: Rotaţie simplă la stânga pentru re-echilibrare

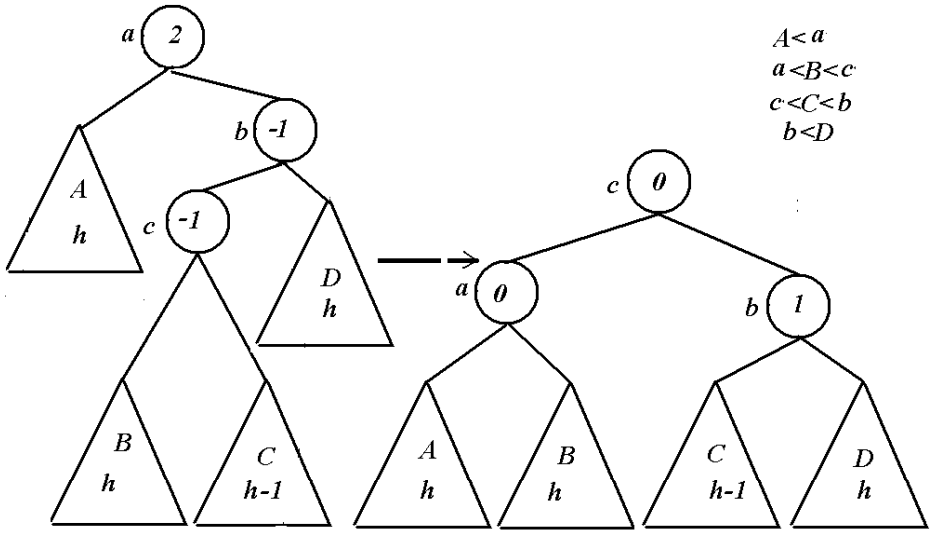


Figura 4.12: Rotaţie dublă la stânga pentru re-echilibrare

b.2) Factorul de echilibru al fiului drept c al lui b este 1. Se tratează ca şi cazul b.1) (figura 4.13). .

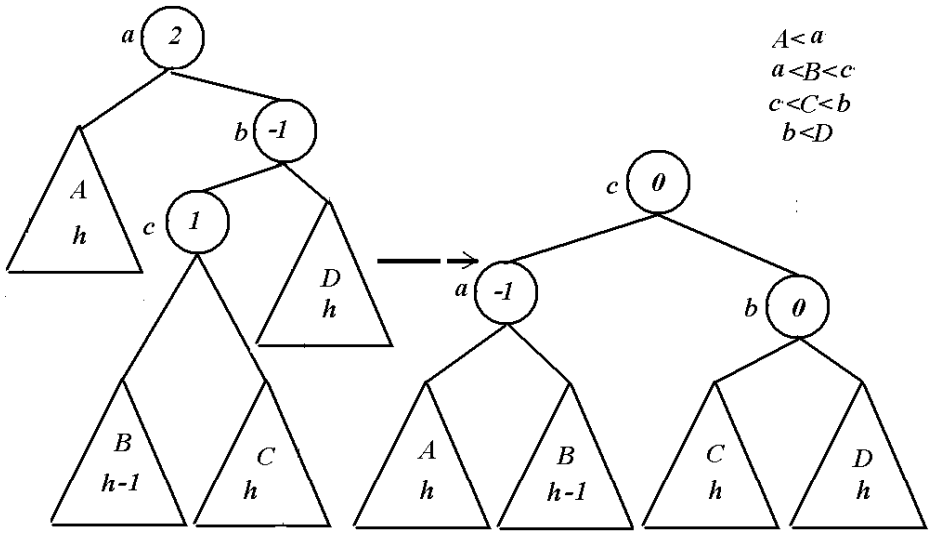


Figura 4.13: Rotaţie dublă la stânga pentru re-echilibrare

b.3) Factorul de echilibru al fiului drept c al lui b este 0. Dezechilibrarea este imposibilă.

c) Factorul de echilibru al fiului stâng b al lui a este 0. Dezechilibrarea este imposibilă.

4.5.2 Exemple

În arborele din figura 4.14 ne propunem să inserăm elementul 58. Suntem în cazul I.a). Subarborii cu rădăcinile 70 şi 80 devin dezechilibraţi. Pentru echilibrare se roteşte perechea $(60, 70)$ la dreapta, obţinându-se arborele din figura 4.15.

În arborele din figura 4.16 ne propunem să inserăm elementul 68. Suntem în cazul I.b.1). Pentru echilibrare se roteşte la stânga perechea $(60, 65)$ şi apoi se roteşte la dreapta perechea $(70, 65)$. Se obţine în final arborele din figura 4.17.

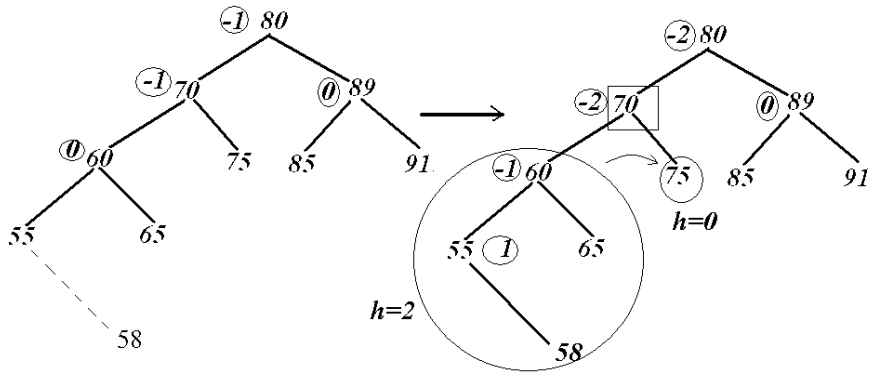


Figura 4.14: Exemplu de inserţie într-un arbore echilibrat

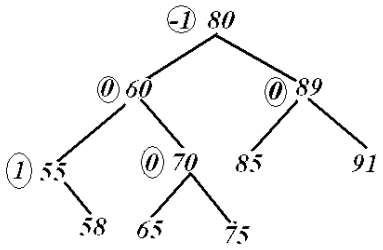


Figura 4.15: Exemplu de inserţie într-un arbore echilibrat

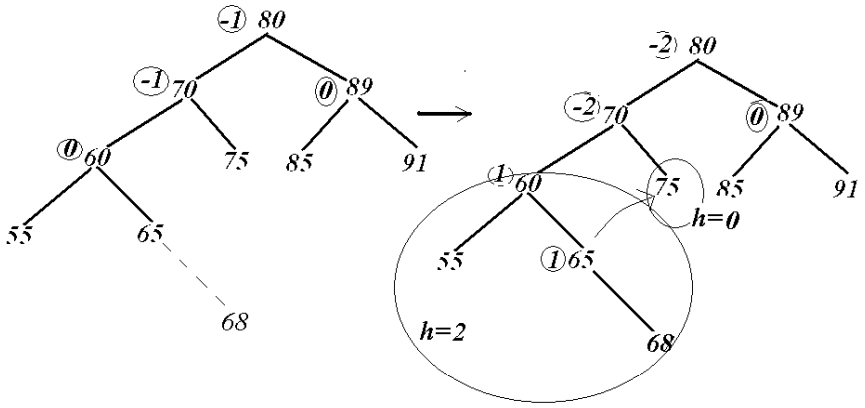


Figura 4.16: Exemplu de inserţie într-un arbore echilibrat

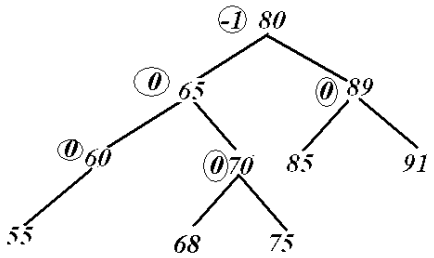


Figura 4.17: Exemplu de inserţie într-un arbore echilibrat

4.5.3 Algoritmul de inserţie în arbori echilibraţi

Pentru a insera un element x într-un arbore binar de căutare echilibrat se parcurg următoarele etape:

- se caută poziţia în care noul element trebuie inserat (ca în orice arbore binar de căutare);

- se inserează elementul x (ca în orice arbore binar de căutare);

- se reactualizează factorii de echilibru ai ascendenţilor lui x până la rădăcină sau până se găseşte cel mai apropiat ascendent p al lui x, dacă există, astfel încât subarborele T cu rădăcina p să fie dezechilibrat;

- dacă p există, se re-echilibrează subarborele T cu ajutorul unei rotaţii (simple sau duble).

4.6 Ştergerea unui nod al unui arbore echilibrat

Ştergerea este similară inserţiei: ştergem nodul aşa cum se procedează în cazul unui arbore binar de căutare şi apoi re-echilibrăm arborele rezultat. Deosebirea este că numărul de rotaţii necesar poate fi tot atât de mare cât nivelul (adâncimea) nodului ce urmează a fi şters. De exemplu să ştergem elementul x din figura 4.18.a).

În urma ştergerii se ajunge la arborele ne-echilibrat din figura 4.18.b). Subarborele cu rădăcina în y este ne-echilibrat. Pentru a-l echilibra este necesară o rotaţie simplă la dreapta a perechii (y, i)obţinându-se arborele din figura 4.19.d); şi acest arbore este ne-echilibrat, rădăcina a având factorul de echilibru -2. O rotaţie dublă la dreapta a lui e, b şi a, re-echilibrează arborele ajungându-se la arborele din figura 4.20.e).

4.6.1 Algoritmul de ştergere a unui nod dintr-un arbore echilibrat

Fie dată înregistrarea având cheia x. Pentru a şterge dintr-un arbore de căutare echilibrat nodul cu cheia x parcurgem următoarele etape

- se localizează nodul r având cheia x;

- dacă r nu există, algoritmul se termină;

- altfel, se şterge nodul r, utilizând algoritmul de ştergere într-un arbore binar de căutare;.

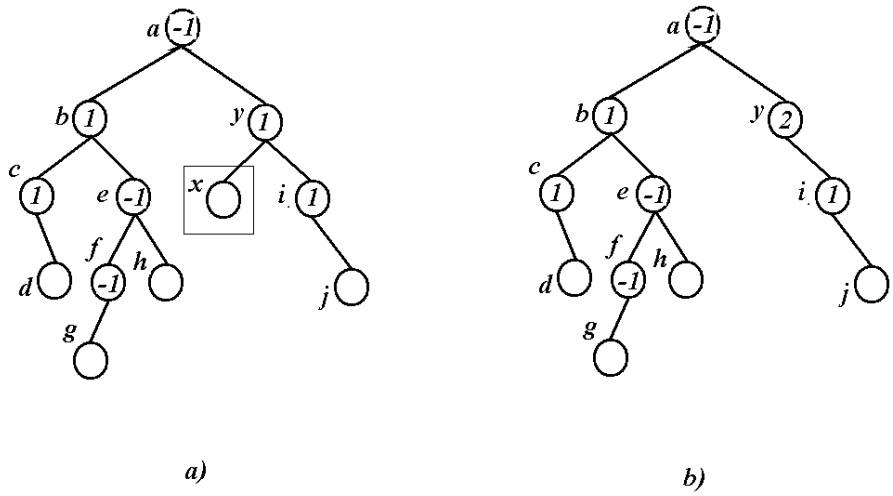


Figura 4.18: Exemplu de ştergere a unui nod dintr-un arbore echilibrat

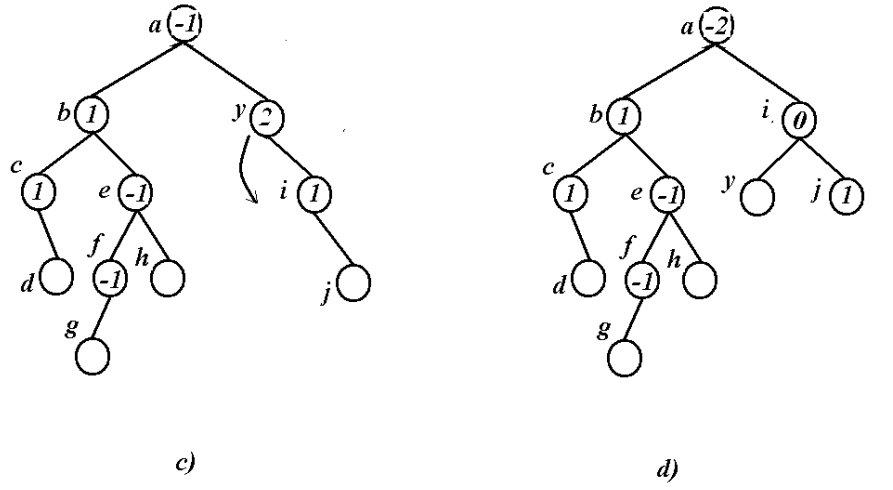


Figura 4.19: Exemplu de ştergere a unui nod dintr-un arbore echilibrat

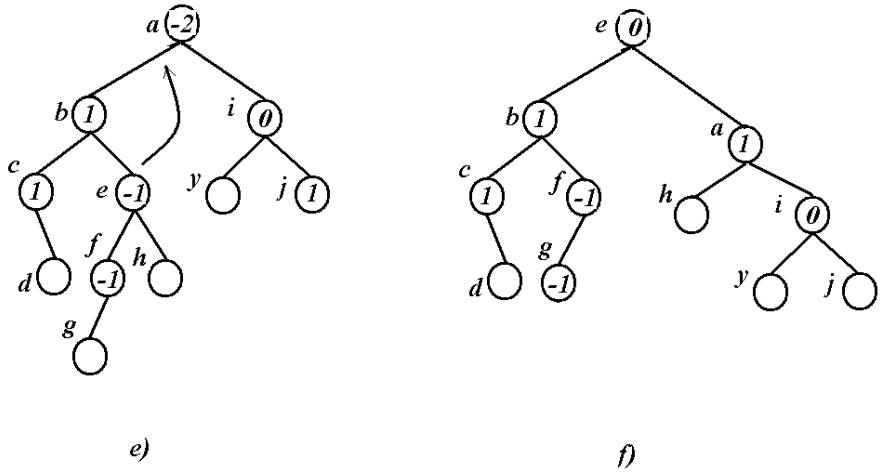


Figura 4.20: Exemplu de ştergere a unui nod dintr-un arbore echilibrat

- fie q nodul extern eliminat în urma aplicării algoritmului de ştergere şi p tatăl său. Se re-echilibrează (dacă este cazul) arborele prin rotaţii implicând nodul p şi eventual ascendenţii acestuia.

Vom arăta că numărul de rotaţii necesar ştergerii unui nod poate ajunge până la numărul ce indică nivelul nodului în arbore. Observăm mai întâi că o rotaţie reduce cu 1 înălţimea arborelui căruia îi este aplicată. Fie a cel mai apropiat predecesor al elementului şters x, astfel ca subarborele T_a cu rădăcina în a să fie neechilibrat. Dacă înainte de rotaţie $h(T_a) = k$, atunci după rotaţie $h(T_a) = k - 1$.

Fie b tatăl lui a (dacă a nu este rădăcina arborelui). Avem următoarele situaţii favorabile:

- factorul de echilibru al lui b este 0: nu este necesară nici-o rotaţie;
- factorul de echilibru al lui b este 1 şi T_a este subarborele drept al lui b: nu este necesară nici-o rotaţie;
- factorul de echilibru al lui b este -1 şi T_a este subarborele stâng al lui b: nu este necesară nici-o rotaţie.

Dificultăţile se ivesc atunci când:

- factorul de echilibru al lui b este -1 şi T_a este subarborele drept al lui b;
- factorul de echilibru al lui b este 1 şi T_a este subarborele stâng al lui b

În ambele cazuri va trebui să re-echilibrăm arborele T cu rădăcina în b.

Să observăm că înainte de echilibrare $h(T) = k+1$ iar după re-echilibrare $h(T) = k$. Astfel, subarborele având drept rădăcină pe tatăl lui b poate deveni ne-echilibrat şi tot aşa până când ajungem la rădăcina arborelui iniţial (în cel mai rău caz).

În continuare prezentăm un program, scris în C de inserare şi ştergere de noduri într-un arbore binar de căutare echilibrat.

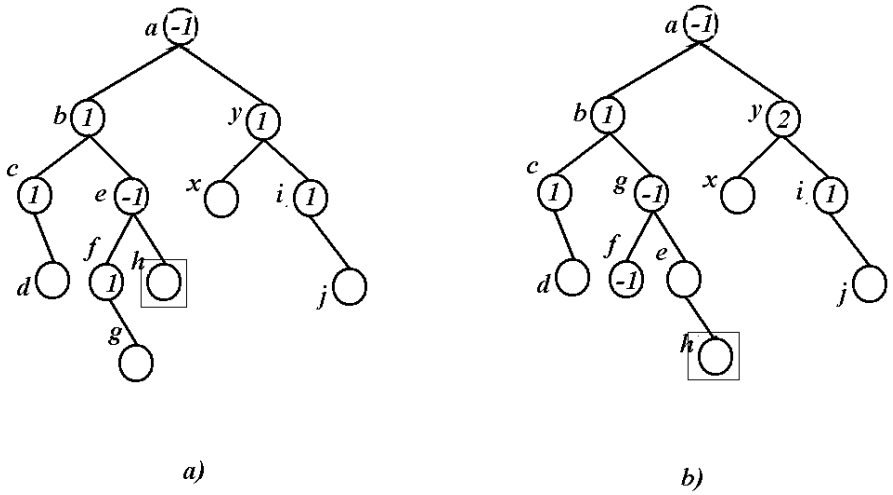


Figura 4.21: Exemplu de ştergere a unui nod dintr-un arbore echilibrat

```
# include<stdio.h>
# include<malloc.h>
# define F 0
# define T 1
struct Nod{char Info;int FactEch;struct Nod *st;struct Nod *dr;};
struct Nod *Inserare (char , struct Nod *, int *);
void Listare(struct Nod *, int );
struct Nod *EchilibrareDr(struct Nod *, int *);
struct Nod *EchilibrareSt(struct Nod *, int *);
struct Nod *Sterge(struct Nod *, struct Nod *, int *);
struct Nod *StergeElement(struct Nod *, char , int *);
/*****************************************/
/* Functia de inserare in arborele de cautare */
struct Nod * Inserare (char Info, struct Nod *tata, int *H)
```

```
{struct Nod *Nod1;
struct Nod *Nod2;
if(!tata)
{tata = (struct Nod *) malloc(sizeof(struct Nod));
tata->Info = Info;
tata->st = NULL;
tata->dr = NULL;
tata->FactEch = 0;
*H = T;
return (tata);}
if(Info < tata->Info)
{tata->st = Inserare(Info, tata->st, H);
if(*H)
/* Creste inaltimea subarborelui stang */
{
switch(tata->FactEch)
{
case 1: /* Subarborele drept mai inalt*/
tata->FactEch = 0;
*H = F;
break;
case 0: /* Arbore echilibrat */
tata->FactEch = -1;
break;
case -1: /* Subarborele stang mai inalt */
Nod1 = tata->st;
if(Nod1->FactEch == -1)
{//Cazul din figura 4.8
printf("Rotatie simpla la dreapta \n");
tata->st= Nod1->dr;
Nod1->dr = tata;
tata->FactEch = 0;
tata = Nod1;
tata->FactEch = 0;}
else
/*cazul Nod1->FactEch == 0 nu este posibil pentru ca
am fi avut *H=F; ramane Nod1->FactEch == 1 ca in
figurile 4.9 si 4.10 */
```

```
{
printf("Rotatie dubla la dreapta \n");
Nod2 = Nod1->dr;
Nod1->dr = Nod2->st;
Nod2->st = Nod1;
tata->st = Nod2->dr;
Nod2->dr = tata;
if(Nod2->FactEch == -1)
tata->FactEch = 1;
else
tata->FactEch = 0;
if(Nod2->FactEch == 1)
Nod1->FactEch = -1;
else
Nod1->FactEch = 0;
tata = Nod2;}
tata->FactEch = 0;
*H = F;}}}
if(Info > tata->Info)
{
tata->dr = Inserare(Info, tata->dr, H);
if(*H)
/* Subarborele drept devine mai inalt */
{
switch(tata->FactEch)
{
case -1: /* Subarborele stang este mai inalt */
tata->FactEch = 0;
*H = F;
break;
case 0: /* Arbore echilibrat */
tata->FactEch = 1;
break;
case 1: /* Subarborele drept este mai inalt */
Nod1 = tata->dr;
if(Nod1->FactEch == 1)
{/*Cazul din figura 4.11 */
printf("Rotatie simpla la stanga \n");
```

```
tata->dr= Nod1->st;
Nod1->st = tata;
tata->FactEch = 0;
tata = Nod1;
tata->FactEch = 0;}
else
/*cazul Nod1->FactEch == 0 nu este posibil pentru ca
am fi avut *H=F; ramane Nod1->FactEch == -1 ca in
figurile 4.12 si 4.136 */
{printf("Rotatie dubla la stanga \n");
Nod2 = Nod1->st;
Nod1->st = Nod2->dr;
Nod2->dr = Nod1;
tata->dr = Nod2->st;
Nod2->st = tata;
if(Nod2->FactEch == 1)
tata->FactEch = -1;
else
tata->FactEch = 0;
if(Nod2->FactEch == -1)
Nod1->FactEch = 1;
else
Nod1->FactEch = 0;
tata = Nod2;
}
tata->FactEch = 0;
*H = F;}}}
return(tata);}
/*********************************************/
/* Functia de listare */
void Listare(struct Nod *Arbore,int Nivel)
{int i;
if (Arbore)
{
Listare(Arbore->dr, Nivel+1);
printf("\n");
for (i = 0; i < Nivel; i++)
printf(" ");
```

```
printf("%c", Arbore->Info);
Listare(Arbore->st, Nivel+1);
}
}
/* Echilibrare in cazul cand subarborele drept
devine mai inalt in comparatie cu cel stang*/
/************************************/
struct Nod * EchilibrareDr(struct Nod *tata, int *H)
{
struct Nod *Nod1, *Nod2;
switch(tata->FactEch)
{
case -1:
tata->FactEch = 0;
break;
case 0:
tata->FactEch = 1;
*H= F;
break;
case 1: /* Re-echilibrare */
Nod1 = tata->dr;
if(Nod1->FactEch >= 0)
/* Cazul din figura 4.18 a) cu tata==y */
{
printf("Rotatie simpla la stanga \n");
tata->dr= Nod1->st;
Nod1->st = tata;
if(Nod1->FactEch == 0)
{
tata->FactEch = 1;
Nod1->FactEch = -1;
*H = F;
}
else
{
tata->FactEch = Nod1->FactEch = 0;
}
tata = Nod1;
```

```
}
else
{
printf("Rotatie dubla la stanga \n");
Nod2 = Nod1->st;
Nod1->st = Nod2->dr;
Nod2->dr = Nod1;
tata->dr = Nod2->st;
Nod2->st = tata;
if(Nod2->FactEch == 1)
tata->FactEch = -1;
else
tata->FactEch = 0;
if(Nod2->FactEch == -1)
Nod1->FactEch = 1;
else
Nod1->FactEch = 0;
tata = Nod2;
Nod2->FactEch = 0;
}
}
return(tata);
}
/* Echilibrare in cazul cand subarborele stang
devine mai inalt in comparatie cu cel drept*/
/*****************************************/
struct Nod * EchilibrareSt(struct Nod *tata, int *H)
{
struct Nod *Nod1, *Nod2;
switch(tata->FactEch)
{
case 1:
tata->FactEch = 0;
break;
case 0:
tata->FactEch = -1;
*H= F;
break;
```

```
case -1: /* Re-echilibrare */
Nod1 = tata->st;
if(Nod1->FactEch <= 0)
/*Cazul figurii 4.18 a) cu tata==e */
{
printf(" Rotatie simpla la dreapta \n");
tata->st= Nod1->dr;
Nod1->dr = tata;
if(Nod1->FactEch == 0)
{
tata->FactEch = -1;
Nod1->FactEch = 1;
*H = F; }
else
{ tata->FactEch = Nod1->FactEch = 0; }
tata = Nod1; }
else
/*cazul din figura 4.21 cu tata==e */
{ printf("Rotatie dubla la dreapta \n");
Nod2 = Nod1->dr;
Nod1->dr = Nod2->st;
Nod2->st = Nod1;
tata->st = Nod2->dr;
Nod2->dr = tata;
if(Nod2->FactEch == -1)
tata->FactEch = 1;
else
tata->FactEch = 0;
if(Nod2->FactEch == 1)
Nod1->FactEch = -1;
else
Nod1->FactEch = 0;
tata = Nod2;
Nod2->FactEch = 0; } }
return(tata); }
/* Inlocuieste informatia nodulului 'Temp' in care a fost gasita cheia
cu informatia celui mai din dreapta descendent al lui 'R' (pe care
apoi il sterge)*/
```

```
/*******************************************/
struct Nod * Sterge(struct Nod *R, struct Nod *Temp, int *H)
{ struct Nod *DNod = R;
if( R->dr != NULL)
{ R->dr = Sterge(R->dr, Temp, H);
if(*H)
R = EchilibrareSt(R, H); }
else
{ DNod = R;
Temp->Info = R->Info;
R = R->st;
free(DNod);
*H = T; }
return(R); }
/* Sterge element cu cheia respectiva din arbore */
/*********************************************/
struct Nod * StergeElement(struct Nod *tata, char Info, int
*H)
{ struct Nod *Temp;
if(!tata) {
printf(" Informatia nu exista \n");
return(tata); }
else { if (Info < tata->Info ) {
tata->st = StergeElement(tata->st, Info, H);
if(*H)
tata = EchilibrareDr(tata, H); }
else
if(Info > tata->Info) { tata->dr = StergeElement(tata->dr,
Info, H);
if(*H)
tata = EchilibrareSt(tata, H); }
else { Temp= tata;
if(Temp->dr == NULL) {
tata = Temp->st;
*H = T;
free(Temp); }
else
if(Temp->st == NULL) { tata = Temp->dr;
```

```
*H = T;
free(Temp); }
else
{ Temp->st = Sterge(Temp->st, Temp, H);
if(*H)
tata = EchilibrareDr(tata, H); } } }
return(tata); }
/* Functia principala*/
/*************************************/
void main()
{ int H;
char Info ;
char choice;
struct Nod *Arbore = (struct Nod *)malloc(sizeof(struct Nod));
Arbore = NULL;
printf(" Tastati 'b' pentru terminare: \n");
choice = getchar();
while(choice != 'b')
{ fflush(stdin);
printf("Informatia nodului (tip caracter: a,b,1,2,etc.): \n");
scanf("%c",&Info);
Arbore = Inserare(Info, Arbore, &H);
printf("Arborele este: \n");
Listare(Arbore, 1); fflush(stdin);
printf("Tastati 'b' pentru terminare: \n");
choice = getchar(); }fflush(stdin);
while(1) {
printf(" Tastati 'b' pentru terminare: \n");
printf(" Introduceti cheia pe care vreti s-o stergeti: \n");
scanf("%c",&Info);
if (Info == 'b') break;
Arbore = StergeElement(Arbore, Info, &H);
printf(" Arborele este: \n");
Listare(Arbore, 1); } }
```

Bibliografie

[1] T. H. CORMEN, C. E. LEISERSON, R. R. RIVEST, *Introducere în algoritmi*, Edit. Computer Libris AGORA, Cluj-Napoca, 2000

[2] H. GEORGESCU, *Tehnici de programare*, Edit. Universităţii Bucureşti, 2005.

[3] D. E. KNUTH, *The Art of Computer Programming*, vol.1, Reading, Massachusets, 1969; vol. 3, Addison-Wesley, 1973.

[4] D. STOILESCU, *Culegere de C/C++*, Edit. Radial, Galaţi, 1998

[5] I. TOMESCU, *Data Structures*, Edit. Universităţii Bucureşti, 1998.