

STRUCTURI DE DATE

Adrian CARABINEANU

0

Cuprins

1	Algoritmi. Notiuni generale	5
1.1	Exemplu de algoritm. Sortarea prin insertie	5
1.2	Aspecte care apar la rezolvarea unei probleme	7
1.3	Timpul de executie a algoritmilor	7
1.4	Corectitudinea algoritmilor	9
1.5	Optimalitatea algoritmilor	9
1.6	Existenta algoritmilor	14
2	Tipuri de structuri de date	15
2.1	Generalitati	15
2.2	Liste	15
2.2.1	Liste alocate secvential	16
2.2.2	Liste alocate nlantuit	16
2.3	Stive	27
2.4	Liste de tip coada	28
2.5	Grafuri	30
2.6	Arbori binari	35
2.6.1	Parcurgerea arborilor binari	36
2.7	Algoritmul lui Hu man	42
2.7.1	Prezentare preliminara	43
2.7.2	Coduri pre x. Arbore de codi care	43
2.7.3	Constructia codi carii pre x a lui Hu man	45
2.7.4	Optimalitatea algoritmului Hu man	49
3	Tehnici de sortare	51
3.1	Heapsort	51

3.1.1	Reconstituirea proprietatii de heap	52
3.1.2	Constructia unui heap	54
3.1.3	Algoritmul heapsort	54
3.2	Cozi de prioritati	57
3.3	Sortarea rapida	60
3.3.1	Descrierea algoritmului	60
3.3.2	Performanta algoritmului de sortare rapida	62
3.4	Metoda bulelor (bubble method)	64
4	Tehnici de cautare	65
4.1	Algoritmi de cautare	65
4.1.1	Algoritmi de cautare secventiala (pas cu pas)	66
4.1.2	Cautarea n tabele sortate (ordonate)	67
4.1.3	Arbori de decizie asociati cautarii binare	71
4.1.4	Optimalitatea cautarii binare	72
4.2	Arbori binari de cautare	76
4.3	Arbori de cautare ponderati (optimali)	81
4.4	Arbori echilibrati	86
4.4.1	Arbori Fibonacci	87
4.4.2	Proprietati ale arborilor echilibrati	89
4.5	Insertia unui nod ntr-un arbore echilibrat	91
4.5.1	Rotatii n arbori echilibrati	91
4.5.2	Exemple	95
4.5.3	Algoritmul de insertie n arbori echilibrati	98
4.6	Stergerea unui nod al unui arbore echilibrat	98
4.6.1	Algoritmul de stergere a unui nod dintr-un arbore echilibrat	98

Lista gurilor

1.1	Arbore de decizie	11
2.1	Liste simplu si dublu nlantuite	17
2.2	Exemple de grafuri	31
2.3	Exemplu de arbore binar	36
2.4	Exemplu de arbore binar cu precizarea legaturilor	37
2.5	Exemplu de arbore Hu man	44
2.6	Construirea arborelui Hu man	46
3.1	Exemplu de heap reprezentat sub forma unui arbore binar si sub forma unui vector	52
3.2	Efectul functiei ReconstituieHeap	53
3.3	Model de executie a functiei ConstruiesteHeap	55
4.1	Arbore de cautare binara	71
4.2	Arbori de cautare	72
4.3	Optimizarea lungimii drumurilor externe	73
4.4	Stergerea radacinii unui arbore binar de cautare	80
4.5	Arbore binar de cautare	80
4.6	Arbori posibili de cautare si numarul mediu de comparatii pentru o cautare reusita	82
4.7	Arbori Fibonacci	88
4.8	Rotatie simpla la dreapta pentru re-echilibrare	92
4.9	Rotatie dubla la dreapta pentru re-echilibrare	93
4.10	Rotatie dubla la dreapta pentru re-echilibrare	93
4.11	Rotatie simpla la stanga pentru re-echilibrare	94
4.12	Rotatie dubla la stanga pentru re-echilibrare	94
4.13	Rotatie dubla la stanga pentru re-echilibrare	95
4.14	Exemplu de insertie ntr-un arbore echilibrat	96

4.15	Exemplu de insertie ntr-un arbore echilibrat	96
4.16	Exemplu de insertie ntr-un arbore echilibrat	97
4.17	Exemplu de insertie ntr-un arbore echilibrat	97
4.18	Exemplu de stergere a unui nod dintr-un arbore echilibrat . .	99
4.19	Exemplu de stergere a unui nod dintr-un arbore echilibrat . .	99
4.20	Exemplu de stergere a unui nod dintr-un arbore echilibrat . .	100
4.21	Exemplu de stergere a unui nod dintr-un arbore echilibrat . .	101

Capitolul 1

Algoritmi. Notiuni generale

Definiție preliminară. Un algoritm este o procedură de calcul bine definită care primește o mulțime de valori ca date de intrare și produce o mulțime de valori ca date de ieșire.

1.1 Exemplu de algoritm. Sortarea prin inserție

Vom considera mai întâi problema sortării (ordonării) unui sir de n numere întregi $a[0]; a[1]; \dots; a[n-1]$ (ce reprezintă datele de intrare). Sirul ordonat (de exemplu crescător) reprezintă datele de ieșire. Ca procedură de calcul vom considera procedura sortării prin inserție pe care o prezentăm în cele ce urmează: Începând cu al doilea număr din sir, repetăm următorul procedeu: inserăm numărul de pe poziția j , reținut într-o cheie, în subsirul deja ordonat $a[0]; \dots; a[j-1]$ astfel încât să obținem subsirul ordonat $a[0]; \dots; a[j]$: Ne oprim când obținem subsirul ordonat de n elemente. De exemplu, pornind de la sirul de numere întregi 7; 1; 3; 2; 5; folosind sortarea prin inserție obținem succesiv

```
7 j 1 3 2 5
1 7 j 3 2 5
1 3 7 j 2 5
1 2 3 7 j 5
1 2 3 5 7
```

Linia verticală j separă subsirul ordonat de restul sirului. Prezentăm mai jos programul scris în C++ pentru sortarea elementelor unui sir de 10 numere

ntregi:

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

Putem modifica programul C++ de sortare a n numere ntregi impunand sa se citeasca numarul n; alocand dinamic un spatiu de n obiecte de tip **int** si tinand cont ca $(a + i) = a[i]$:

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


1.2. ASPECTE CARE APAR LA REZOLVAREA UNEI PROBLEME 7

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

1.2 Aspecte care apar la rezolvarea unei probleme

Cand se cere sa se elaboreze un algoritm pentru o problema data, care sa furnizeze o solutie (e si aproximativa, cu conditia mentionarii acestui lucru), cel putin teoretic trebuie sa se parcurse urmatoarele etape:

1) Demonstrarea faptului ca este posibila elaborarea unui algoritm pentru determinarea unei solutii.

2) Elaborarea unui algoritm (in acest caz etapa anterioara poate deveni inutila).

3) Demonstrarea corectitudinii.

4) Determinarea timpului de executie al algoritmului.

5) Investigarea optimalitatii algoritmului.

Vom prezenta in cele ce urmeaza, nu neaparat in ordinea indicata mai sus, aceste aspecte.

1.3 Timpul de executie a algoritmilor

Un algoritm este elaborat nu doar pentru un set de date de intrare, ci pentru o multime de astfel de seturi. De aceea trebuie bine precizata multimea (seturilor de date) de intrare. Timpul de executie se masoara in functie de lungimea n a setului de date de intrare.

Ideal este sa determinam o formula matematica pentru $T(n)$ = timpul de executare pentru orice set de date de intrare de lungime n : Din pacate, acest lucru nu este in general posibil. Din aceasta cauza ne vom limita la a evalua ordinul de marime al timpului de executie.

Sa reluam procedura de calcul pentru algoritmul de sortare prin insertie:

```
//procedura de calcul .....cost:::.....timp
for(int j= 1;j< n;j++ ) .....c1:::.....n
find key= *(a+j); .....c2:::.....n-1
//insereaza a[j] in sirul sortat a[1,...,j-1]
i=j-1; .....c3:::.....n-1
while((i>= 0)* (*(a+i)> key)) .....c4:::..... $\sum_{j=2}^n t_j$ 
f*(a+i+1)= *(a+i); .....c5:::..... $\sum_{j=2}^n (t_j-1)$ 
i=i-1;g.....c6:::..... $\sum_{j=2}^n (t_j-1)$ 
*(a+i+1)= key; g.....c7:::.....n-1
```

c_1, \dots, c_7 sunt costurile de timp pentru fiecare instructiune. In coloana timp punem de cate ori este repetata instructiunea; t_j reprezinta numarul de executii ale testului **while** (comparatia) pentru valoarea j : Timpul de executie 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 executie poate sa depinda de natura datelor de intrare. In cazul in care vectorul de intrare este deja sortat crescator, $t_j = 1$ (deoarece pentru fiecare j ; $a[0; \dots; j-1]$ este sortat). Timpul de executie in acest caz (cel mai favorabil) este

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

Daca vectorul este sortat in sens invers (in ordine descrescatoare) avem cazul cel mai defavorabil. Fiecare element $a[j]$ este comparat cu fiecare element din $a[0; \dots; j-1]$ si astfel $t_j = j$ pentru $j = 2; 3; \dots; n$: Cum

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

deducem ca

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

Timpul de executie este are ordinul de marime $O(n^2)$: In general spunem ca timpul de executie este de ordinul $O(f(n))$ daca

$$\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} = l; l \neq 0$$

Cand $f(n) = n^k; k \in \mathbf{N}$ spunem ca algoritmul este **polinomial**. Spunem ca un algoritm este considerat acceptabil daca este polinomial.

1.4 Corectitudinea algoritmilor

In demonstrarea corectitudinii algoritmilor exista doua aspecte importante

- **Corectitudinea partiala**: presupunand ca algoritmul se termina intr-un numar finit de pasi, trebuie demonstrat ca rezultatul este corect.
- **Terminarea programului**: trebuie demonstrat ca algoritmul se incheie in timp finit.

Evident, conditiile enumerate mai sus trebuie indeplinite pentru orice set de date de intrare admis de problema.

Modul tipic de lucru consta in introducerea in anumite locuri din program a unor **invarianti**, care reprezinta relatii ce sunt indeplinite la orice trecere a programului prin acele locuri. De exemplu in cazul sortarii prin insertie, invariantii sunt urmatoarii:

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

Ciclul **for** se termina odata cu ultima executare a ciclului **while**, cand $j = n$ si cand toate elementele sunt sortate.

1.5 Optimalitatea algoritmilor

Sa presupunem ca pentru o anumita problema am elaborat un algoritm si am putut calcula timpul sau de executie $T(n)$: Este natural sa ne punem problema daca algoritmul este executat in timpul cel mai scurt posibil sau exista un alt algoritm cu timpul de executie mai mic. Spunem ca un algoritm este **optim** daca raportul dintre timpul sau de executie si timpul de executie al oricarui alt algoritm care rezolva aceeasi problema are ordinul de marime $O(1)$: Problema demonstrarii optimalitatii este deosebit de dificila, mai ales

datorita faptului ca trebuie sa consideram toti algoritmii posibili si sa aratam ca ei au un timp de executie superior celui al algoritmului optim.

In cazul algoritmilor de sortare ne propunem sa gasim o margine inferioara a timpilor de executie. Vom face mai intai observatia ca numarul total de instructiuni executate si numarul de comparatii au acelasi ordin de marime. Multimea comparatiilor poate fi vizualizata cu ajutorul arborilor de decizie. Intr-un arbore de decizie fiecare nod este etichetat prin $a_i : a_j$ pentru i si j din intervalul $1 \leq i, j \leq n$ unde n este sa numarul de elemente din secventa de intrare. Fiecare frunza este etichetata cu o permutare $((1) ; \dots ; (n))$: Executia algoritmului de sortare corespunde trasarii unui drum elementar de la radacina arborelui de sortare la o frunza. La fiecare nod intern este facuta o comparatie intre a_i si a_j : Subarborele stang dicteaza comparatiile urmatoare pentru $a_i \leq a_j$ iar subarborele drept dicteaza comparatiile urmatoare pentru $a_i > a_j$: Cand ajungem la o frunza algoritmul a stabilit ordonarea $a_{(1)} \leq a_{(2)} \leq \dots \leq a_{(n)}$: Pentru ca algoritmul sa ordoneze adecvat, fiecare din cele $n!$ permutari de n elemente trebuie sa apara intr-o frunza a arborelui de decizie. In figura (1.1) prezentam arborele de decizie corespunzator sortarii unei multimi $\{a_1; a_2; a_3\} = \{1; 2; 3\}$: In functie de datele de intrare, comparatiile efectuate de program reprezinta un drum elementar in arborele de decizie ce uneste radacina arborelui cu o frunza. Numarul de noduri (comparatii) dintr-un astfel de drum elementar este egal cu cel mult h , inaltimea arborelui.

Teorema 1. Orice arbore de decizie care sorteaza n elemente are inaltimea de ordinul $O(n \ln n)$:

Demonstratie. Intrucat exista $n!$ permutari ale celor n elemente, arborele trebuie sa aiba $n!$ frunze. Un arbore binar de inaltime h are cel mult 2^h frunze. Deci

$$n! \leq 2^h;$$

$$h \geq \log_2 n! = \ln n! / \log_2 e;$$

Plecand de la inegalitatea

$$n! > \frac{n^n}{e^n};$$

obtinem

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

adica

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

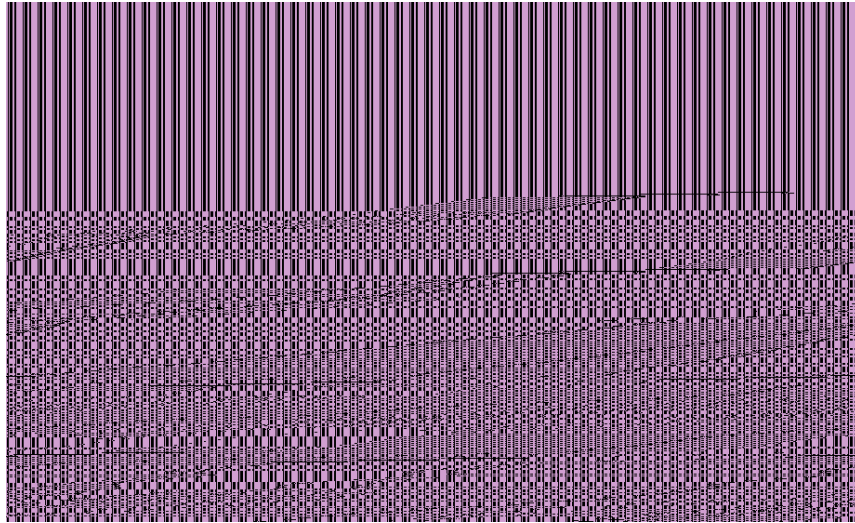


Figura 1.1: Arbore de decizie

Ținând cont de teorema mai sus enunțată, este de presupus că un algoritm de sortare optim are timpul de execuție de ordinul $O(n \ln n)$: Algoritmul de sortare prin inserție, având timpul de execuție de ordinul $O(n^2)$; are toate șansele să nu fie optim. Vom da în cele ce urmează un exemplu de algoritm de sortare optim și anume algoritmul de sortare prin interclasare.

Pentru a avea o imagine intuitivă a procedurii de interclasare, să considerăm că un pachet cu n cărți de joc este împărțit în alte 2 pachete așezate pe masă cu fața în sus. Fiecare din cele 2 pachete este sortat, cartea cu valoarea cea mai mică fiind deasupra. Dorim să amestecăm cele două sub-pachete într-un singur pachet sortat, care să rămână pe masă cu fața 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) și de a o pune cu fața 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 și să-l punem peste pachetul deja sortat întorcând toate cărțile cu fața în jos. Din punct de vedere al timpului de execuție, deoarece avem de făcut cel mult $n=2$ comparații, timpul de execuție 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 si stapaneste**, dupa cum urmeaza:

Divide: Imparte sirul de n elemente ce urmeaza a fi sortat in doua subsiruri de cate $n/2$ elemente.

Stapaneste: Sorteaza recursiv cele doua subsiruri utilizand sortarea prin interclasare.

Combina: Interclaseaza cele doua subsiruri sortate pentru a produce rezultatul final.

Descompunerea sirului in alte doua siruri ce urmeaza a fi sortate are loc pana cand avem de sortat siruri cu unul sau doua componente. Prezentam mai jos programul scris in C++. In program, functia **sort** sorteaza un vector cu maximum 2 elemente, functia **intec1** interclaseaza 2 siruri sortate iar **dist** implementeaza strategia **divide si stapaneste** a metodei studiate.

```
#include<iostream.h>
/*****/
void sort(int p,int q, int n, int *a) {
    int m;
    if(* (a+ p)> * (a+ q))
        f m= * (a+ p); * (a+ p)= * (a+ q); * (a+ q)= m;gg
/*****/
void intec1(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))
            f*(b+ k)= * (a+ i);i= i+ 1;k= k+ 1;g
        else
            f*(b+ k)= * (a+ j);j= j+ 1;k= k+ 1;g
    if(i<= m)
        for (j= i;j<= m;j++)
            f*(b+ k)= * (a+ j); k= k+ 1;g
    else for(i= j;i<= q;i++)
        f*(b+ k)= * (a+ i); k= k+ 1;g
    k= 1;
    for(i= p;i<= q;i++) f*(a+ i)= * (b+ k); k= k+ 1;gg
/*****/
void dist(int p, int q, int n, int *a){f
```

```

int m;
if((q-p)<= 1) sort(p,q,n,a);
else
fm= (p+ q)/2; dist(p,m,n,a); dist(m+ 1,q,n,a);intekl(p,q,m,n,a);
gg
/*****/
void main(void)f
int n; *a,i;
cout<< n= ; cin>> n;
a= new int[n];
for(i= 1;i<= n;i+ +)
fcout<< a[ <<i<< ]= ; cin>> *(a+ i-1);g
dist(0,n-1,n,a);
for(i= 1;i<= n;i+ +)
cout<< a[ <<i-1<< ]= <<a[i-1];g

```

In continuare sa calculam $T(n)$, numarul aproximativ de comparatii efectuat de algoritm. Succesiv, problema se descompune in alte doua probleme, fiecare referindu-se la $n/2$ elemente. Urmeaza interclasarea elementelor, care necesita un numar de $n/2$ comparatii. Avem deci

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{2}; \quad T(0) = 0:$$

Pentru inceput sa consideram $n = 2^k$; $k \in \mathbf{N}$: Rezulta (efectuand implicit un rationament prin inductie):

$$\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + 2^{k-1} = 2[2T(2^{k-2}) + 2^{k-2}] + 2^{k-1} = \dots \\
&= 2^p T(2^{k-p}) + p2^{k-1} = 2^p [2T(2^{k-p-1}) + 2^{k-p-1}] + p2^{k-1} = \\
&= 2^{p+1} T(2^{k-p-1}) + (p+1)2^{k-1} = T(0) + k2^{k-1} = k2^{k-1}.
\end{aligned}$$

Pentru un numar natural oarecare n ; $\exists k$ astfel incat sa avem $2^k \leq n < 2^{k+1}$:
Rezulta

$$k2^{k-1} = T(2^k) \quad T(n) < T(2^{k+1}) = (k+1)2^k: \quad (1.4)$$

Cum $k = O(\ln n)$; din (1.4) rezulta ca

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

deci algoritmul de sortare prin interclasare este optim.

1.6 Existenta algoritmilor

Problema existentei algoritmilor a stat în atenta matematicienilor încă înainte de aparitia calculatoarelor. Un rol deosebit în aceasta teorie l-a jucat matematicianul englez Alan Turing (1912-1954), considerat parintele inteligenței artificiale.

Numim **problema nedecidabilă** o problema pentru care nu poate fi elaborat un algoritm. De nirea matematica a notiunii de algoritm a permis detectarea de probleme nedecidabile. Cateva aspecte legate de decidabilitate sunt urmatoarele:

- **Problema opririi programelor:** pentru orice program și orice valori de intrare să se decida dacă programul se termina.
- **Problema echivalentelor programelor:** să se decida pentru oricare două programe dacă sunt echivalente (produc aceeași ieșire pentru aceleași date de intrare).

Capitolul 2

Tipuri de structuri de date

2.1 Generalitati

Structurile de date reprezinta modalitati in care datele sunt dispuse in memoria calculatorului sau sunt pastrate pe discul magnetic. Structurilor de date sunt utilizate in diferite circumstante ca de exemplu:

- Memorarea unor date din realitate;

- Instrumente ale programatorilor;

- Modelarea unor situatii din lumea reala.

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

2.2 Liste

O lista liniara este o structura de date omogena, secventiala formata din elemente ale listei. Un element (numit uneori si nod) din lista contine o informatie specifica si eventual o informatie de legatura. De exemplu, in lista echipelor de fotbal inscrite intr-un campionat, un element (ce reprezinta o echipa) poate contine urmatoarele informatii specifice: nume, numar de puncte, numar de goluri inscrite si numar de goluri primite. Fiecare din aceste informatii poate reprezenta o cheie care poate fi utilizata pentru comparatii si inspectii. De exemplu luand drept cheie numarul de puncte si golaverajul se poate face clasierea echipelor.

Pozitia elementelor din lista de neste ordinea elementelor (primul element, al doilea, etc). Continutul listei se poate schimba prin:

- adaugarea de noi elemente la sfarsitul listei;
- inserarea de noi elemente n orice loc din lista;
- stergerea de elemente din orice pozitie a listei;
- modi carea unui element dintr-o pozitie data.

Printre operatiile care schimba structura listei vom considera si **initializarea** unei liste ca o lista vida.

Alte operatii sunt operatiile de caracterizare. Ele nu modi ca structura listelor dar furnizeaza informatii despre ele. Dintre operatiile de caracterizare vom mentiona n cele ce urmeaza:

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

Pot luate n considerare si operatii mai complexe, ca de exemplu:

- separarea unei liste n doua sau mai multe subliste n functie de ndeplinirea unor conditii;
- selectia elementelor dintr-o lista, care ndeplinesc unul sau mai multe criterii, ntr-o lista noua;
- crearea unei liste ordonate dupa valorile creascatoare sau descrescatoare ale unei chei;
- combinarea a doua sau mai multor liste prin concatenare (alipire) sau interclasare.

Spatiul din memorie ocupat de lista poate alocat n doua moduri: prin alocare secventiala sau prin alocare nlantuita.

2.2.1 Liste alocate secvential

In acest caz nodurile ocupa pozitii succesive n memorie. Acest tip de alocare este ntalnit cand se lucreaza cu tablouri (vectori). Avantajul alocarii secventiale este dat de faptul ca accesul la oricare din nodurile listei este direct. Dezavantajul consta n faptul ca operatiile de adaugare, eliminare sau schimbare de pozitie a unui nod necesita un efort mare de calcul dupa cum s-a vazut n algoritmii de sortare prezentati mai nainte.

2.2.2 Liste alocate nlantuit

Exista doua feluri de alocare nlantuita: **alocare simplu nlantuita** si **alocare dublu nlantuita** (gura 2.1). Alocarea nlantuita poate efectuata static

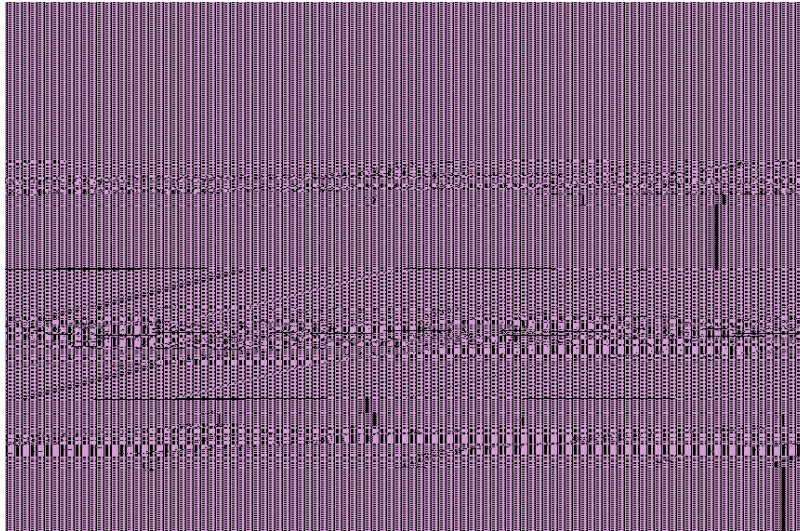


Figura 2.1: Liste simplu si dublu nlantuite

(utilizand vectori) sau dinamic. In acest din urma caz (de care ne vom ocupa in cele ce urmeaza), se utilizeaza o zona de memorie numita **HEAP** (movila, gramada). In C++ variabilele pastrate in HEAP (cum ar fi de exemplu nodurile listei), se aloca atunci cand doreste programatorul, cu ajutorul operatorului **new**, iar zona se elibereaza, tot la dorinta acestuia prin operatorul **delete**. In cazul alocarii dinamice, nodul unei liste este o structura care contine alaturi de informatie si adresa nodului urmator (pentru liste simplu nlantuite) sau adresele nodului urmator si a celui precedent in cazul listelor dublu nlantuite. Dupa cum se va vedea din exemplele ce urmeaza, principala problema in cazul operatiilor cu liste o reprezinta redenumirea legaturilor (adreselor) cand se sterge sau se insereaza un element.

Liste simplu nlantuite

Mai jos prezentam proceduri (functii) de **creare** (prin insertie repetata) si **parcure** a listelor precum si procedurile de **stergere** (eliminarea) a unui nod si de **inserare** a unui nod imediat dupa alt nod ce contine o anumita informatie.

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

```

struct nod fint inf; nod *adr ;g;
//declararea functiilor de creare, parcurgere, eliminare si inserare
nod *creare(void);
void parcurge(nod *prim);
    nod *elimina(nod *prim, int info);
nod *inserare_dupa(nod* prim, int info, int info1);
//functia principala
/*****/

void main(void) f
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);g
//functia de creare a listei
/*****/

nod *creare(void)
f 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 urma-
toare*/
    cout<< "Gata?[d/n] << endl;
    cin>> a;
    while (a!= 'd') f
        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;g
    return prim;g
    /*****/
    /*urmeaza procedura de parcurgere a listei*/
    void parcurge(nod *cap)
    fnod *q;
    if(!cap) fcout<< Lista vida ; return;g
    cout<< Urmeaza lista << endl;
    for(q= cap;q=q->adr)
    cout<< q->inf<< ;g
    /*****/
    /*urmeaza procedura de eliminare a nodului ce contine o anumita infor-
    matie*/
    nod *elimina(nod* prim, int info)
    f nod *q,*r;
    /*se studiaza mai intai cazul cand trebuie eliminat primul nod*/
    while((*prim).inf== info)
    fq= (*prim).adr; delete prim; prim= q;g
    /*se studiaza cazul cand informatia cautata nu este in primul nod*/
    q= prim;
    while(q->adr)
    f 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*/
    f r= q->adr; q->adr= r->adr; delete r;g
    else q= q->adr;g
    return prim;g
    /*****/

```

/*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)
```

```
fnod *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*/

```
fd-> adr= NULL; c-> adr= d;g
```

/* daca au fost depistate noduri ce contin informatia data noul element se insereaza dupa primul astfel de nod*/

```
else fd-> adr= c-> adr; c-> adr= d;g
```

```
return prim;g
```

Ca o ilustrare a utilitatii listelor liniare simplu inlantuite vom prezenta n cele ce urmeaza o procedura de adunare si nmultire a doua polinoame de o variabila. Un polinom va reprezentat printr-o lista, un nod al listei corespunzand unui monom. Informatia din nodul listei (monom) va contine gradul monomului si coe cientul. Pentru a efectua produsul polinoamelor vom crea cu functia **prod** o noua lista, n ecare nod al noii liste ind produsul a doua monoame (ecare dintre aceste monoame apartinand altui polinom). Cu o functie numita **canonic**, daca doua noduri (monoame) din lista nou creata (lista produs) au acelasi grad, coe cientul unuia din monoame este nlocuit cu suma coe cientilor iar coe cientul celuilalt cu 0. Cu functia **elimina** stergem apoi nodurile (monoamele) care contin coe cientul 0.

Pentru a aduna doua polinoame, vom efectua mai ntai concatenarea lor (ultimul element din lista ce reprezinta primul polinom va legata de primul element din lista ce reprezinta al doilea element). Aplicand apoi functiile **canonic** si **elimina** obtinem polinomul suma. Prezentam programul mai jos:

```
#include< iostream.h>
```

```
struct nodfint grad; int coef; nod *adr ;g;
```

```
/******
```

```
//declararea functiilor utilizate
```

```
nod *create(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)f
nod *cap, *cap1,*suma,*produs,*prim;
cap= creare();
cout<<  Listeaza primul polinom << endl;
parcure(cap);
cap1= creare();
cout<<  Listeaza al doilea polinom << endl;
parcure(cap1);
cout<<  Produsul polinoamelor << endl;
cout<<  ***** ;
produs= prod(cap,cap1);
canonic(produs);
produs= elimina(produs,0);
parcure(produs);
prim= concatenare(cap,cap1);
cout<<  Polinoamele concatenate << endl;
parcure(prim);
canonic(prim);
cout<<  Suma polinoamelor << endl;
suma= elimina(prim,0);
parcure(suma);g
/*****/
//functia de creare a unui polinom
nod *creare(void)f
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')f
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;g
return prim;g
/*****/
//functia de parcurgere a unui polinom
void parcurge(nod *cap)f
nod *q;
if(!cap)cout<< "Polinom vid ";
return;
cout<< "Urmeaza polinomul " << endl;
for(q= cap;q=q-> adr)
cout<< " + ( " << (*q).coef<< " ) " << "X " << (*q).grad;g
/*****/
//functia care efectueaza produsul a doua polinoame
nod* prod(nod* prim1, nod* prim2)
fnod *p,*q,*r,*cap,*s;
cap= new nod;
cap-> grad= 1;cap-> coef= 0;cap-> adr= NULL;
s= cap;
for(p= prim1;p=p-> adr)
for(q= prim2;q=q-> adr)
fr= new nod;r-> coef= p-> coef*q-> coef;
r-> grad= p-> grad+ q-> grad;r-> adr= NULL;
s-> adr= r;s= r;g

```



```

return cap;g
/*****/
/*functia care aduna coe cientii a doua monoame de acelasi grad si
atribuie valoarea 0 coe cientului unuia din monoamele adunate*/
void canonic(nod *prim)f
nod *p,*q;
for (p= prim;p;p= p-> adr)
fq= p-> adr;
while(q) if(p-> grad== q-> grad)
fp-> coef+= q-> coef;q-> coef= 0;q= q-> adr;ggreturn;g
/*****/
/*functia care elimina monoamele ale caror coe cienti au o valoare data*/
nod *elimina(nod* prim, int info)
fnod *q,*r;
if((*prim).coef== info)
q= (*prim).adr; delete prim; prim= q;
q= prim;
while(q-> adr)
if(q-> adr-> coef== info)
fr= q-> adr; q-> adr= r-> adr; delete r;g
else q= q-> adr;
return prim;g
/*****/
//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 nlantuite

In continuare vom prezenta un program n cadrul caruia se creeaza o lista dublu nlantuita, se parcurge direct si invers si se elimina nodurile ce contin o informatie data. Structura **nod** va contie trei campuri: unul (**inf**) este informatia nodului, al doilea (**urm**) este pointerul care indica adresa nodului urmator iar al treilea (**ant**) este pointerul care indica adresa nodului anterior.

Vom introduce un nou tip de variabila, numit **lista** constand dintr-o structura formata din doua variabile de tip **nod** (cele doua noduri reprezentand primul si ultimul element al listei dublu nlantuite). Functia **creare** permite crearea unei liste dublu nlantuite. Functia **parcureg_d**, al carui argument este primul element al listei, permite parcuregerea directa (plecand de la primul element si ajungand la ultimul) a listei. Functia **parcureg_i** al carei argument este ultimul nod al listei realizeaza parcuregerea listei n sens invers. Functia **elimin_d** ale carei argumente sunt o variabila de tip **lista** si o variabila de tip **int**, parcureg direct lista dubla si elimina nodurile ce contin argumentul de tip ntreg de cate ori le ntalneste. Proprietatea pe care o au listele duble de a putea parcureg n ambele sensuri este folosita de functia **sortin** pentru a sorta prin insertie, dupa valorile din campul **inf**, elementele listei, valoarea ntoarsa de functie ind lista sortata (mai precis primul si ultimul element al listei sortate).

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

```

cout<< Urmeaza lista sortata << endl;
parcureg_d(prim);g
/*****/
//functia de creare
lista creare(void)
fnod *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')
fp=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;g
val.p=prim;
if(!prim->urm) val.u=prim;
else val.u=p;
return val;g
/*****/
//functia de parcuregere directa
void parcureg_d(nod *prim)
fnod *q;
if(!prim)fcout<< Lista vida ;return;g
cout<< Urmeaza lista directa << endl;
for(q=prim;q=q->urm)
cout<< q->inf<< ;g
/*****/
//functia de parcuregere inversa
void parcureg_i(nod *ultim)
fnod *q;
if(!ultim)fcout<< Lista vida ;return;g

```

```

cout<< Urmeaza lista inversa<< endl ;
for(q= ultim;q;q= q-> ant)
cout<< q-> inf<< ;g
/*****/
/*functia de eliminare a nodurilor ce contin o informatie data*/
lista elimin_d(lista list, int info)f
nod *q,*r,*p;
while(list.p-> inf== info)
if (list.p== list.u) flist.p=NULL;list.u=NULL;return list;g
else
fq= list.p-> urm;q-> ant=NULL; delete list.p; list.p=q;g
q= list.p;
while(q-> urm)
fif(q-> urm== list.u)
fif (q-> urm-> inf== info)
fr= q-> urm;
list.u=q;q-> urm=NULL;delete r;g
return list;g
if(q-> urm-> inf== info)
fp=q;r= q-> urm; q-> urm= r-> urm;
q-> urm-> ant=p; delete r;g
else q= q-> urm;g
list.u=q;return list;g
/*****/
//functia de sortare prin insertie
nod* sortin(lista list)f
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)
ffor(s= q-> ant;s;s= s-> ant)
fif (!s-> ant) break;
if(q-> inf>= s-> ant-> inf) break;g
if(q-> inf< s-> inf)
fm= q-> inf;
for(r= q;!(r== s);r= r-> ant)
r-> inf= r-> ant-> inf;s-> inf= m;gg
return list.p;g

```

2.3 Stive

Stiva este o lista pentru care singurele operatii permise sunt:

- adaugarea unui element n stiva;
- eliminarea, vizitarea sau modi carea ultimului element introdus n stiva.

Stiva functioneaza dupa principiul ultimul intrat primul ieseit - Last In First Out (LIFO).

In cazul alocarii dinamice (de care ne vom ocupa n cele ce urmeaza), elementele (nodurile) stivei sunt structuri n componenta carora intra si adresa nodului anterior.

In programul de mai jos dam functiile **push** de adaugare n stiva a unei nregistrari si **pop** de extragere. Cu ajutorul lor construim o stiva folosind functia **create**.

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

```

fnod*r= stiva;
if(!stiva) cout<< Stiva vida << endl;
else fcout<< Urmeaza stiva << endl;
while(r)fcout<< r->inf<< ;r= r->ant;gg
return;g
/******
/*functia de creare a stivei prin adaugari si eliminari*/
nod* creare()
fchar 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'))f
cout<< Urmatoarea operatie[a/e/t] << endl;
/* t= termina; a= adauga nod; e= elimina nod;*/
cin>> a;
switch(a)
fcase 't': break;
case 'a':cout<< Stiva->inf << endl;
cin>> info;
stiva= push(stiva,info);break;
default:stiva= pop(stiva);break;gg
return stiva;g
/******
//functia principala
void main()
fstiva= creare();
parcurge(stiva);g

```

2.4 Liste de tip coada

O coada este o lista pentru care toate inserarile sunt facute la unul din capete iar toate stergerile (consultarile, modificările) sunt efectuate la celalalt capat.

Coadă funcționează pe principiul primul intrat primul iese - First In First Out (FIFO).

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

În programul de mai jos dam funcțiile **pune** de adăugare în coadă a unei înregistrări și **scoate** de extragere. Cu ajutorul lor construim o coadă folosind funcția **create**. Vom reprezenta coada printr-o structură **coada** care conține primul și ultimul nod al cozii.

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

```

cout << endl;g
/*****/
//functia de creare a cozii
coada creare()
fchar 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'))f
cout<<  Urmatoarea operatie ;
cout<<  [a= pune nod/s= scoate/t= termina] << endl;
cin>> a;
switch(a)
fcase 's': c= scoate(c);break;
case 'a':cout<<  c.p->inf << endl;cin>> info;
c= pune(c,info);break;
default:break;gg
return c;g
/*****/
//functia principala
void main()
fc= creare();
parcurge(c);g

```

2.5 Grafuri

Un graf orientat G este o pereche $(V; E)$ unde V este o multime nita iar E este o relatie binara pe V . Multimea V se numeste multimea varfurilor iar multimea E se numeste multimea arcelor lui G $((u; v) \in E; u \in V; v \in V)$.

Intr-un graf neorientat $G = (V; E)$ multimea muchiilor este constituita din perechi de varfuri neordonate $(fu; vg \in E; u \in V; v \in V)$: Daca $(u; v)$ este un arc dintr-un graf orientat $G = (V; E)$ spunem ca $(u; v)$ este incident din

sau pleaca din varful u si este incident n sau intra n varful v . Despre $(u; v)$ sau $fu;vg$ spunem ca sunt incidente varfurilor u si v : Daca $(u; v)$ sau $fu;vg$ reprezinta un arc (muchie) ntr-un graf spunem ca varful v este **adiacent** varfului u : (Pentru simplitate folosim notatia $(u; v)$ si pentru muchiile grafurilor neorientate.)

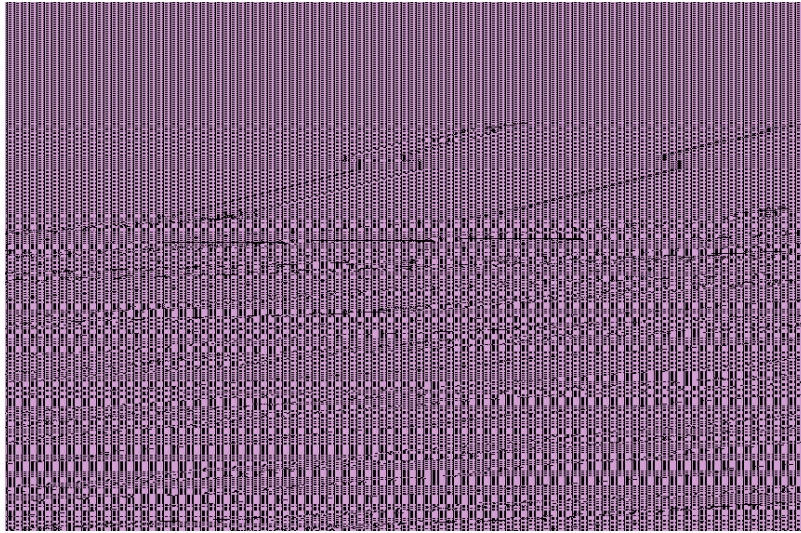


Figura 2.2: Exemple de grafuri

In gura (2.2) prezentam un graf orientat si un graf neorientat. Adiacenta grafurilor se pune n evidenta cu ajutorul unei **matrice de adiacenta**. De exemplu matricea de adiacenta a grafului orientat din gura este

	0	1	2	3
0	0	1	0	1
1	0	1	1	0
2	0	0	0	0
3	0	1	0	0

0; 1; 2; 3 sunt etichetele varfurilor grafului considerat. Daca $(u; v) \in E$ este un arc al grafului punem valoarea 1 pentru elementul aflat pe linia u si coloana v a matricei. In caz contrar punem 0.

Matricea de adiacenta a grafului neorientat este

	0	1	2	3
0	0	1	0	1
1	1	0	1	1
2	0	1	0	0
3	1	1	0	0

Dacă $fu;vg$ este o muchie a grafului punem valoarea 1 pentru elementul a^{at} pe linia u și coloana v a matricei. În caz contrar punem 0. Observăm că matricea de adiacență a unui graf neorientat este simetrică.

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

Un drum de lungime k de la un varf u la un varf u^0 într-un graf $G = (V; E)$ este un sir de varfuri $(v_0; v_1; v_2; \dots; v_{k-1}; v_k)$ astfel încât $u = v_0; u^0 = v_k$ și $(v_{i-1}; v_i) \in E$ pentru $i = 1; 2; \dots; k$: Drumul conține varfurile $v_0; v_1; \dots; v_{k-1}; v_k$ și muchiile(arcele) $(v_0; v_1); (v_1; v_2); \dots; (v_{k-1}; v_k)$:

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

Prezentăm în continuare un program scris în C care stabilește, pe baza matricei de adiacență dacă între două varfuri ale grafului există un drum. Ideea pe care se bazează programul este următoarea: dacă între două varfuri i și j există un drum, atunci oricare ar fi varful k , între i și k există un drum dacă și numai dacă $(i; k)$ este arc al grafului sau între j și k există un drum. În final programul alegea o matrice în care elementul de pe linia i și coloana j ia valoarea 1 dacă există un drum de la i la j și 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)
fprintf( "Dimensiunea matricii n= ");scanf( "%d",&n);
for(i=0;i<n;i++)
for(j=0;j<n;j++)
```

```

fprintf( x[%d][%d]= ,i,j);
scanf( %d ,&x[i][j]);y[i][j]=x[i][j];g
j= 0;while(j< n)
f
i= 0;
while(i< n)
fif(y[i][j]!= 0)
fk= 0;
while(k< n)
fy[i][k]=y[i][k]jy[j][k]; k + + ; g; g;
i+ + ;g;
j+ + ;g;
for(i= 0;i< n;i+ + )
for(j= 0;j< n;j+ + )
printf( y[%d][%d]= %d ,i,j,y[i][j]);g

```

Pentru graful orientat din gura (2.2) se obtine matricea

	0	1	2	3	
0	0	1	1	1	
1	0	1	1	0	:
2	0	0	0	0	
3	0	1	1	0	

În cazul grafurilor neorientate, dacă $x_1 = x_r$ și 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 $(x_1; x_2; \dots; x_r = x_1)$ se numește **ciclu**. Dacă muchiile ciclului sunt arce orientate avem un ciclu într-un graf orientat.

Un graf neorientat este **conex** dacă pentru fiecare două varfuri $u; v$ există un drum $(u; \dots; v)$ 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 relația de incluziune a multimilor. Numărul de varfuri $jV(G)$ al unui graf se numește **ordinul** grafului iar numărul muchiilor $jE(G)$ reprezintă **marimea** grafului.

Un graf (neorientat) conex care nu conține cicluri se numește **arbore**. Dam în continuare câteva teoreme de caracterizare a arborilor:

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

- a) G este un graf conex fără cicluri;

b) G este un graf fara cicluri maximal (daca i se adauga lui G o muchie, atunci apare un ciclu);

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

Demonstratie. a) \Rightarrow b). Fie G conex si fara cicluri. Fie u, v din V astfel ca $(u; v) \notin E$: Cum graful este conex, exista un drum $(v; \dots; u)$ de la v la u : Adaugand si muchia $(u; v)$; graful $G^0 = (V; E \cup (u; v))$ va contine ciclul $(v; \dots; u; v)$:

b) \Rightarrow c) Daca G nu ar fi conex, in virtutea proprietatii de maximalitate, adaugand o noua muchie ce uneste doua varfuri din doua componente conexe diferite nu obtinem un ciclu, ceea ce contrazice b). Asadar G este conex. Sa presupunem mai departe prin absurd ca $e \notin E$ si $G^0 = (V; E \cup e)$ este conex. Atunci exista un drum ce uneste extremitatile lui e in G ; deci G contine un ciclu, ceea ce contrazice din nou b):

c) \Rightarrow a): Daca G ar contine un ciclu si e ar fi o muchie a acestui ciclu, atunci $G^0 = (V; E \cup e)$ ramane conex, ceea ce contrazice c):

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

Demonstratie. Mai intai vom demonstra ca G contine cel putin un varf de gradul 1 (varfuri terminale, frunze). Sa presupunem prin absurd ca gradul $d(v) \geq 2$ pentru orice $v \in V$: In acest caz sa consideram in G un drum D de lungime maxima si sa notam cu x o extremitate a acestui drum. Cu presupunerea facuta, varful x ar avea gradul cel putin 2; deci in virtutea maximalitatii lui D trebuie sa fie adiacent cel putin altui varf din D ; formandu-se astfel un ciclu in G . Apare o contradictie.

Acum proprietatea ca G are $n - 1$ muchii rezulta usor prin inductie. Ea este adevarata pentru $n = 1$. Presupunem ca este adevarata pentru toti arborii de ordin cel mult $n - 1$ si fie G un arbore de ordin n . Daca x este un nod terminal al lui G ; atunci G^0 cu $V(G^0) = V \setminus \{x\}$ este si el un arbore de ordinul $n - 1$ si conform ipotezei de inductie va avea $n - 2$ muchii. Rezulta ca G are $n - 1$ muchii.

Teorema 4. Fie G un graf de ordinul $n \geq 3$: Urmatoarele conditii sunt echivalente:

- a) G este un graf conex fara cicluri;
- d) G este un graf fara cicluri cu $n - 1$ muchii;
- e) G este conex si are $n - 1$ muchii;
- f) Exista un unic drum intre orice doua varfuri distincte ale lui G ;

Demonstratie. a) \Leftrightarrow d): Avem a) \Rightarrow (b si teorema 2) \Rightarrow d):

d) \Rightarrow a): Presupunem prin absurd ca G nu este conex. Adaugand un numar de muchii care sa uneasca elemente din diverse componente conexe ale grafului putem obtine un graf conex fara cicluri cu ordinul mai mare ca $n - 1$: Se ajunge la o contradictie (in virtutea teoremei 2), deci G este conex.

a) \Rightarrow e): Avem a) \Rightarrow (c si teorema 2) \Rightarrow e):

e) \Rightarrow a): Presupunem prin absurd ca G are cicluri. Extragand in mod convenabil unele muchii, se ajunge astfel la un graf conex fara cicluri, de ordin n cu mai putin de $n - 1$ muchii. Se obtine astfel o contradictie in virtutea teoremei 3.

a) , f): Rezulta imediat in virtutea de notiilor.

Din teoremele 2 si 4 obtinem sase caracterizari diferite ale arborilor :
(a) (f) :

2.6 Arbori binari

Ne vom ocupa in continuare de studiul arborilor binari deoarece acestia constituie una din cele mai importante structuri neliniare intalnite in teoria algoritmilor. In general, structura de arbore se refera la o relatie de ramnii - care la nivelul nodurilor, asemanatoare aceleia din cazul arborilor din natura. In cele ce urmeaza vom introduce intr-o alta maniera notiunea de arbore.

Arborii sunt constituiti din **noduri interne** (puncte de ramnii care) si **noduri terminale** (frunze). Fie $V = \{v_1; v_2; \dots; v_n\}$ o multime de noduri interne si $B = \{b_1; b_2; \dots; b_m\}$ o multime de frunze. Definim inductiv multimea arborilor peste V si B :

Definitie.

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

b) Daca $T_1; \dots; T_m; m \geq 1$ sunt arbori cu multimile de noduri interne si frunze disjuncte doua cate doua iar $v \in V$ este un nou nod, atunci $(m + 1)$ -tuplul $T = \{v; T_1; \dots; T_m\}$ este un arbore. Nodul v este radacina arborelui, $\deg(v) = m$ este gradul arborelui iar $T_i; i = 1; \dots; m$ sunt subarbori ai lui T .

Cand reprezentam grafic un arbore radacina este deasupra iar frunzele sunt dedesupt (vezi figura (2.3)).

Vom preciza termenii folositi atunci cand ne referim in general la arbori. Fie T un arbore cu radacina v si avand subarborii $T_i; 1 \leq i \leq m$: Fie $w_i = \text{root}(T_i)$ radacina subarborelui T_i . Atunci w_i este cel de-al i -lea fiu al lui v iar v este tatal lui w_i : De asemenea w_i este fratele lui $w_j; i, j = 1; \dots; m$: Notiunea

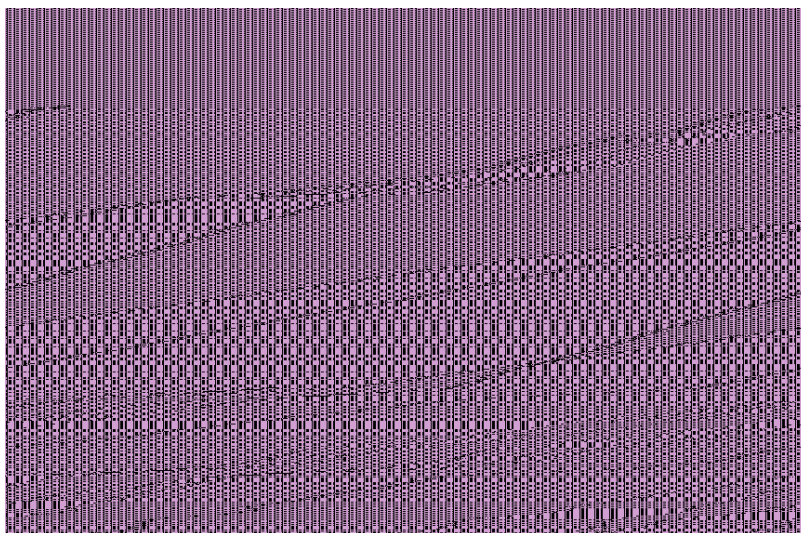


Figura 2.3: Exemplu de arbore binar

de descendent (ascendent) indică închiderea tranzitivă și reflexivă a relației de \leq (tata).

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

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

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 figura 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, informațiile pot fi stocate în frunze sau în nodurile

interne. Fiecare nod al arborelui binar este o structura care contine alaturi de informatia speci ca si adresele nodurilor u stang respectiv drept (gura (2.4)).

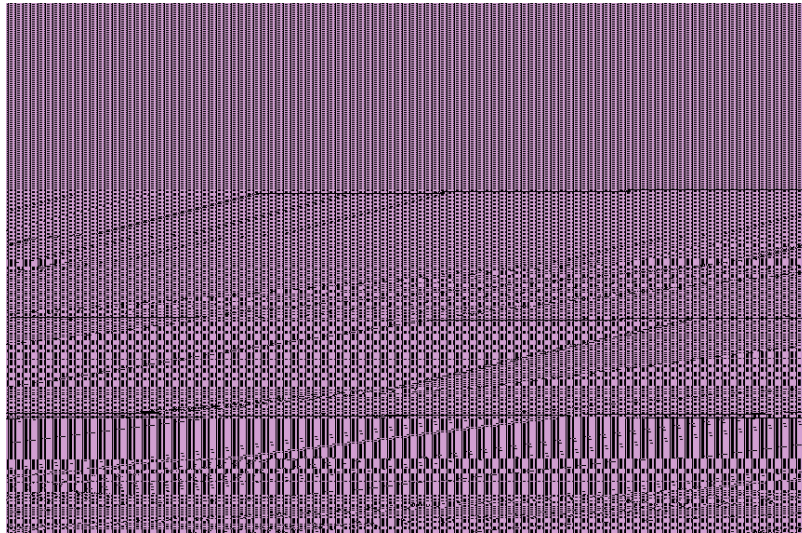


Figura 2.4: Exemplu de arbore binar cu precizarea legaturilor

Pentru a accesa informatiile pastrate de un arbore, trebuie sa-l exploram (parcurgem). Cum orice arbore binar are trei componente (o radacina, un subarbore stang si un subarbore drept), n mod natural apar trei metode de parcurgere a arborelui:

- Parcurgere n **preordine (rsd)**: se viziteaza radacina, se parcurge sub-arborele stang, se parcurge sub arborele drept.
- Parcurgere n **postordine (sdr)**: se parcurge subarborele stang, se parcurge subarborele drept, se viziteaza radacina.
- Parcurgere **simetrica** sau n **inordine(srd)**: se parcurge subarborele stang, se viziteaza radacina, se parcurge subarborele drept. Aplicand aceste de nitii arborelui din gura obtinem $v_1v_2b_1v_4b_4b_5v_3b_2b_3$, pentru parcurgerea n preordine, $b_1b_4b_5v_4v_2b_2b_3v_3v_1$ pentru parcurgerea n postordine iar pentru parcurgerea n inordine $b_1v_2b_4v_4b_5v_1b_2v_3b_3$:

Vom prezenta n cele ce urmeaza un program C++ cu functiile de creare, parcurgere si stergere a unui arbore. Pentru utilizarea nodurilor unui arbore binar a fost de nita o structura cu urmatoarele campuri: **info** - campul

informatie, n acest program de tip ntreg , **st** - adresa nodului descendent stang, **dr** - adresa nodului descendent drept. Programul realizeaza prelucrarea arborelui prin urmatoarele functii:

a) Functia **creare** pentru crearea arborelui efectueaza operatiile:

- alocă memoria necesară unui nod;
- citește informația nodului și a seza mesaje de invitație pentru crearea nodurilor descendente (stang și drept);
- dacă informația introdusă este un număr ntreg diferit de 0, se apelează recursiv funcția pentru crearea subarborelui stang stocându-se adresa de legatură pentru accesul la descendenți. Urmează apoi apelul recursiv pentru crearea subarborelui drept;
- dacă informația introdusă este 0, atunci nodului i se atribuie valoarea NULL:

Funcția întoarce adresa nodului creat.

b) Functia **rsd** pentru parcurgerea n preordine efectuează operațiile:

- prelucrează (a seza) rădăcina;
- trece la descendentul stang apelându-se recursiv pentru parcurgerea subarborelui stang;
- trece la descendentul drept apelându-se recursiv pentru parcurgerea subarborelui drept.

c) Functia **srd** pentru parcurgerea n inordine efectuează operațiile:

- trece la descendentul stang apelându-se recursiv pentru parcurgerea subarborelui stang;
- prelucrează (a seza) rădăcina;
- trece la descendentul drept apelându-se recursiv pentru parcurgerea subarborelui drept.

d) Functia **sdr** pentru parcurgerea n postordine efectuează operațiile:

- trece la descendentul stang apelându-se recursiv pentru parcurgerea subarborelui stang;
- trece la descendentul drept apelându-se recursiv pentru parcurgerea subarborelui drept;
- prelucrează (a seza) rădăcina.

e) Functia **sterge** pentru ștergerea arborelui efectuează operațiile:

- se șterge subarboarele stang: se apelează recursiv ștergerea pentru descendentul stang;
- se șterge subarboarele drept: se apelează recursiv ștergerea pentru descendentul drept;
- se șterge nodul curent;

Urmeaza programul.

```
#include <iostream.h>
typedef struct nod fint info; struct nod *st; struct nod *dr; arbore;
arbore *rad; int n;
// Se declara functiile
/*****/

arbore *create();
void srd(arbore *rad);
void rsd(arbore *rad);
void sdr(arbore *rad);
void sterge(arbore *rad);
/*****/

// Functia principala
void main() f
cout<< Radacina= ;
rad=create();
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);g
/*****/

// Functia de creare a arborelui
arbore *create( )
farbore *p; cin>>n;
if (n!=0)
fp=new arbore;
p->info= n;cout<< p->info<< ->st ;p->st=create( );
cout<< p->info<< ->dr ;p->dr=create( );return p;g
else
p=NULL;return p;g
/*****/

// Functia de parcurgere in inordine
void srd(arbore *rad)
fif (rad!=NULL)
fsrd(rad->st);
```

```

cout<<rad->info<<    ;
srd(rad->dr);gg
/*****/
// Functia de parcurgere in postordine
void sdr(arbore *rad)
fif (rad!= NULL)
fsdr(rad->st);
sdr(rad->dr);cout<<rad->info<<    ;gg
/*****/
// Functia de parcurgere in preordine
void rsd(arbore *rad)
fif(rad!= NULL)
fcout<<rad->info<<    ;
rsd(rad->st);
rsd(rad->dr);gg
/*****/
// Functia de stergere
void sterge(arbore *rad)
f if (rad!= NULL)fsterge(rad->st);sterge(rad->dr);
cout<< S-a sters <<rad->info<<endl;
delete rad;gg;

```

Vom prezenta mai departe o varianta nerecursiva de traversare a unui arbore n in ordine folosind o stiva auxiliara a . Schematic, algoritmul se prezinta astfel:

```

p=radacina; a=NULL;
do f
while(p) f a( p /*pune nodul p al arborelui in stiva*/;
p=p->st;g
if(!a) break;
else p( a/*scoate p din stiva*/;
viziteaza p; p=p->adr;
g while(pjja);

```

Ideea algoritmului este sa salvam nodul curent p in stiva si apoi sa traversam subarborele stang; dupa aceea scoatem nodul p din stiva, l vizitam si apoi traversam subarborele drept.

Sa demonstram ca acest algoritm parcurge un arbore binar de n noduri in ordine simetrica folosind inductia dupa n . Pentru aceasta vom demonstra

urmatorul rezultat: dupa o intrare n in ciclul **do**, cu p_0 radacina unui subarboare cu n noduri si stiva **a** continand nodurile $a[1], \dots, a[m]$, procedura va traversa subarboarele n chestiune n ordine simetrica iar dupa parcurgerea lui stiva va avea n inal aceleasi noduri $a[1], \dots, a[m]$. Afirmatia este evidenta pentru $n = 0$. Daca $n > 0$, este $p = p_0$ nodul arborelui binar la intrarea n in setul de instructiuni al ciclului **do**. Punand p_0 in stiva, aceasta devine $a[1], \dots, a[m], p_0$ iar noua valoare a lui **p** este $p = p_0 \neq st$. Acum subarboarele stang are mai putin de n noduri si conform ipotezei de inductie subarboarele stang va fi traversat in ordine, dupa parcurgere stiva fiind $a[1], \dots, a[m], p_0$. Se scoate p_0 din stiva (aceasta devenind $a[1], \dots, a[m]$), se viziteaza $p = p_0$ si se atribuie o noua valoare lui **p** adica $p = p_0 \neq dr$. Acum subarboarele drept are mai putin de n noduri si conform ipotezei de inductie se va traversa arborele drept avand n inal in stiva nodurile $a[1], \dots, a[m]$.

Aplicand propozitia de mai sus, se intra in ciclul **do** cu radacina arborelui si stiva vida, si se iese din ciclu cu arborele traversat si stiva vida.

Urmeaza programul.

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

```

cout<<(a->inf)->inf<<    ;
a=a->ant;delete r;q.stiv=a;g
return q;g
/*****/
//Functia de creare a arborelui
arbore *creare()
farbore *p; ;cin>>n;
if (n!=0)
fp= new arbore;
p->inf= n;cout<<p->inf<<  ->st   ;p->st= creare( );
cout<<p->inf<<  ->dr   ;p->dr= creare( );return p;g
else
p=NULL;return p;g
/*****/
//Functia principala care realizeaza traversarea in inordine
void main()
fstiva *a; arbore *p;arbstiv q;
cout<<  Radacina  <<endl;
p= creare();
a= NULL;
dof
while (p) fa= push(a,p);p= p->st;g
if (!a) break;
else fq= pop(a);p= q.arb;a= q.stiv;g
p= p->dr;g
while(pjja); g

```

2.7 Algoritmul lui Hu man

Algoritmul de codi care (compresie, compactare) Hu man poarta numele inventatorului sau, David Hu man, profesor la MIT. Acest algoritm este ideal pentru compresarea (compactarea, arhivarea) textelor si este utilizat n multe programe de compresie.

2.7.1 Prezentare preliminară

Algoritmul lui Huffman aparține familiei de algoritmi ce realizează **codi cari cu lungime variabilă**. Aceasta înseamnă că simbolurile individuale (ca de exemplu caracterele într-un text) sunt înlocuite de secvențe de biți (**cuvinte de cod**) care pot avea lungimi diferite. Astfel, simbolurilor care se întâlnesc de mai multe ori în text (și) li se atribuie o secvență mai scurtă de biți în timp ce altor simboluri care se întâlnesc mai rar li se atribuie o secvență mai mare. Pentru a ilustra principiul, să presupunem că vrem să compactăm următoarea secvență: AEEEEBEEDDECDD. Cum avem 13 caractere (simboluri), acestea vor ocupa în memorie $13 \cdot 8 = 104$ biți. Cu algoritmul lui Huffman, șirul este examinat pentru a vedea care simboluri apar cel mai frecvent (în cazul nostru E apare de șapte ori, D apare de trei ori iar A, B și C apar câte o dată). Apoi se construiește (vom vedea în cele ce urmează cum) un arbore care înlocuiește simbolurile cu secvențe (mai scurte) de biți. Pentru secvența propusă, algoritmul va utiliza substituțiile (**cuvintele de cod**) A = 111; B = 1101; C = 1100; D = 10; E = 0 iar secvența compactată (codificată) obținută prin concatenare va fi 111000011010010011001010. Aceasta înseamnă că s-au utilizat 24 biți în loc de 104; raportul de compresie fiind $24=104 \cdot 1=4$. 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 codi cari

Vom considera în continuare doar codi cari în care nici-un cuvânt nu este prefix al altui cuvânt. Aceste codi cari se numesc **codi cari prefix**. Codificarea prefix este utilă deoarece simplă, atât codificarea (deci compactarea) cât și decodificarea. Pentru orice codi cari binară a caracterelor; se concatenează cuvintele de cod reprezentând fiecare caracter al șirului ca în exemplul din subsecțiunea precedentă.

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

111 0 0 0 0 1101 0 0 10 0 1100 10 10;

secvența care se decodifică în AEEEEBEEDCDD: Procesul de decodificare care necesită o reprezentare convenabilă a codificării prex astfel încât cuvântul inițial de cod să poată fi ușor 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 copii), ale cărui frunze sunt caracterele date. Interpretăm un cuvânt de cod binar ca fiind o secvență de biți obținută etichetând cu 0 sau 1 muchiile drumului de la rădăcina până la frunza care conține caracterul respectiv: cu 0 se etichetează muchia care unește un nod cu copilul stâng iar cu 1 se etichetează muchia care unește un nod cu copilul drept. În figura (2.5) este prezentat arborele de codificare al lui Huffman corespunzător exemplului nostru. Notând cu C alfabetul din care fac parte simbolurile (caracterele), un arbore de codificare are exact $|C|$ frunze, una pentru fiecare literă (simbol) din alfabet și așa cum se știe din teoria grafurilor, exact $|C| - 1$ noduri interne (notăm cu $|C|$; cardinalul mulțimii C).

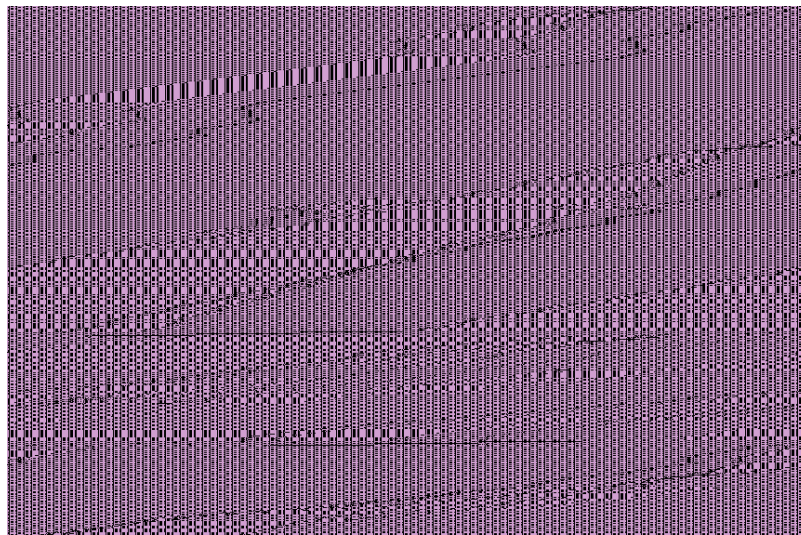


Figura 2.5: Exemplu de arbore Huffman

Dându-se un arbore T , corespunzător unei codificări prex, este foarte simplu să calculăm numărul de biți necesari pentru a codifica un șir.

Pentru fiecare simbol $c \in C$, $f(c)$ frecvența (numărul de apariții) lui c în șir și să notăm cu $d_T(c)$ adâncimea frunzei în arbore. Numărul de biți necesar pentru a codifica șirul este numit **costul arborelui** T și se

calculeaza cu formula

$$\text{COST}(T) = \sum_{c \in C} f(c) d_T(c) :$$

2.7.3 Constructia codii carii pre x a lui Hu man

Hu man a inventat un algoritm greedy care construiesc o codii care pre x optima numita **codul Hu man**. Algoritmul construiesc arborele corespunzator codii carii optime (numit **arborele lui Hu man**) pornind de jos n sus. Se ncepe cu o multime de $|C|$ frunze si se realizeaza o secventa de $|C| - 1$ operatii de fuzionari pentru a crea arborele nal. In algoritmul scris n pseudocod care urmeaza, vom presupune ca C este o multime de n caractere si fiecare caracter $c \in C$ are frecventa $f(c)$. Asimilam C cu o padure constituita din arbori formati dintr-o singura frunza. Vom folosi o stiva S formata din noduri cu mai multe campuri; un camp pastreaza ca informatie pe $f(c)$, alt camp pastreaza radacina c iar un camp suplimentar pastreaza adresa nodului anterior (care indica un nod ce are ca informatie pe $f(c^0)$ cu proprietatea ca $f(c) = f(c^0)$). Extragem din stiva varful si nodul anterior (adica obiectele cu frecventa cea mai redusa) pentru a le face sa fuzioneze. Rezultatul fuzionarii celor doua noduri este un nou nod care n campul informatiei are $f(c) + f(c^0)$ adica suma frecventelor celor doua obiecte care au fuzionat. De asemenea n al doilea camp apare radacina unui arbore nou format ce are ca subarbore stang, respectiv drept, arborii de radacini c si c^0 . Acest nou nod este introdus n stiva dar nu pe pozitia varfului ci pe pozitia corespunzatoare frecventei sale. Se repeta operatia pana cand n stiva ramane un singur element. Acesta va avea n campul radacinilor chiar radacina arborelui Hu man.

Urmeaza programul n pseudocod. Tinand cont de descrierea de mai sus a algoritmului numele instructiunilor programului sunt suficient de sugestive.

```

n  |C|
S  C
cat timp (S are mai mult decat un nod)
  f z  ALOCA-NOD( )
  x  stanga[z]  EXTRAGE-MIN(S)
  y  dreapta[z]  EXTRAGE-MIN(S)
  f(z)  f(x) + f(y)
  INSEREAZA(S; z) g
returneaza EXTRAGE-MIN(S) :
```

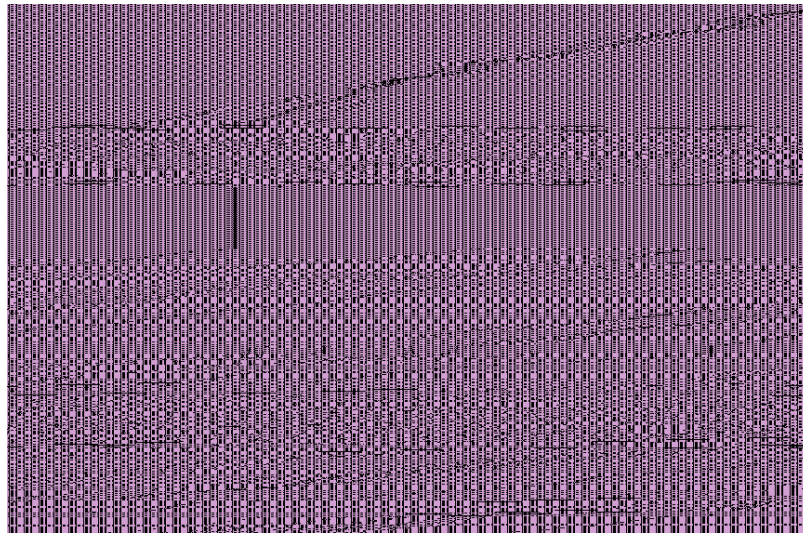


Figura 2.6: Construirea arborelui Hu man

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 frecvențele lor. În figura (2.6) prezentăm arborii a tuturor nodurilor stivei, după fiecare repetare a instrucțiunilor din ciclul **while**. Se pleacă cu o pădure de frunze, ordonată descrescător după frecvențele simbolurilor și se ajunge la arborele de codificare.

Prezentăm mai jos și programul în C++ pentru algoritmul Hu man de codificare.

```
#include<iostream.h>
#include<string.h>
typedef struct nodfchar *symb; nod*st;nod*dr;g arbore;
typedef struct nods f arbore*rad; nods*ant;int freqv;g stiva;
/*****/
/*Funcția de punere a unui nou nod în stiva ordonată după frecvență*/
stiva *push(stiva*varf, arbore *a, int f)
fstiva *v,*p,*q;
v= new stiva; v->rad= a;v->freqv= f;v->ant= NULL;
if(!varf)
fvvarf= v; return varf;g
for(p= varf,q= NULL;(p!= NULL)* (p->freqv< f;q= p,p= p->ant
```



```

v->ant= p;
if(q!= NULL) q->ant= v;
else fvarf= v;g
return varf;g
/*****/
//Functia de stergere a varfului stivei
void pop(stiva *
fstiva*r;
r= varf;
varf= varf->ant;
delete r;
return;g
/*****/
//Functia de fuzionare a doi arbori
arbore*fuziune(arbore *p, arbore*q)
farbore*r;
r= new arbore;
r->symb= 'n00;
r->dr= p;r->st= q;
return r;g
/*****/
//Functia de parcurgere in preordine a arborelui lui Huffman
//si de codificare a frunzelor
void rsd(arbore*rad, char*shot)
fchar frunza[60];
for (int i= 0;i< 60;i++)
frunza[i]= 'n00;
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);g

```

```

/*****/
//Functia de creare a unui arbore constand dintr-o singura
//frunza (radacina) care contine simbolul ce trebuie codi cat
arbore *frunza(char *simb)
farbore*r;r= new arbore;
r-> symb= simb;r-> st= NULL;r-> dr= NULL; return r;g
/*****/
//Functia de parcurgere a stivei
void parcurge(stiva*s)
fstiva*rr;rr= s;
if(!rr) cout<< Stiva vida << endl;
else fcout<< Urmeaza stiva << endl;
while(rr)f
cout<< (rr-> rad)-> symb<< ;
rr= rr-> ant;gg
cout<< endl;return;g
/*****/
//Functia principala
void main()
farbore *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
fr1= vf-> rad;f1= vf-> frecv;pop(vf);
r2= vf-> rad;f2= vf-> frecv;pop(vf);
vf= push(vf,fuziune(r1,r2),f1+ f2);
g while(vf-> ant);
cout<< Urmeaza codi carea << endl;
rsd(vf-> rad-> st, 0 );
rsd(vf-> rad-> dr, 1 );g

```

2.7.4 Optimalitatea algoritmului Huffman

Definiție. Un arbore de codi care T pentru alfabetul C este **optim** dacă pentru orice alt arbore de codi care T^0 al aceluiași alfabet avem

$$\text{COST}(T) = \sum_{c \in C} f(c) d_T(c) \quad \text{COST}(T^0) = \sum_{c \in C} f(c) d_{T^0}(c) :$$

Vom demonstra în cele ce urmează că algoritmul Huffman construiește un arbore de codi care optim.

Lema 1. Fie T un arbore de codi care optim pentru alfabetul C . Dacă $f(c) < f(c^0)$ atunci $d_T(c) > d_T(c^0)$:

Demonstratie. Să presupunem prin absurd că avem $f(c) < f(c^0)$ și $d_T(c) < d_T(c^0)$: Schimbând între ele frunzele care conțin pe c și c^0 obținem un nou arbore de codi care cu costul

$$\begin{aligned} \text{COST}(T) - f(c) d_T(c) - f(c^0) d_T(c^0) + f(c) d_T(c^0) + f(c^0) d_T(c) = \\ = \text{COST}(T) - (f(c) - f(c^0)) (d_T(c) - d_T(c^0)) < \text{COST}(T) ; \end{aligned}$$

ceea ce contrazice ipoteza de optimalitate.

Lema 2. Fie frecvențele minime f_1 și f_2 corespunzătoare simbolurilor c_1 și c_2 din alfabetul C . Atunci există un arbore de codi care optim în care frunzele c_1 și c_2 sunt frați.

Demonstratie. Fie înălțimea unui arbore de codi care optim T . Fie un nod de adâncime $h - 1$ și c_i și c_j îi săi, care evident sunt frunze. Să presupunem că $f(c_i) < f(c_j)$: Conform lemei precedente sau $f(c_i) = f_1$ sau $d_T(c_i) > d_T(c_1)$ de unde $d_T(c_i) = d_T(c_1)$ din alegerea lui i . În ambele cazuri putem schimba între ele frunzele c_i și c_1 fără a afecta costul arborelui. La fel procedăm cu c_2 și c_j și obținem un arbore de codi care optim în care c_1 și c_2 sunt frați.

Teorema 5. Algoritmul Huffman construiește un arbore de codi care optim.

Demonstratie (prin inducție după $n = |C|$). Teorema este evidentă pentru $n = 2$. Să presupunem că $n \geq 3$ și este T_{Huff} arborele construit cu algoritmul Huffman pentru frecvențele f_1, f_2, \dots, f_n . Algoritmul adună frecvențele f_1 și f_2 și construiește nodul corespunzător frecvenței $f_1 + f_2$: Fie

T_{Huff}^0 arborele construit cu algoritmul Huffman pentru multimea de frecvențe $f_1 + f_2; f_3; \dots; f_n$. Avem

$$COST(T_{Huff}) = COST(T_{Huff}^0) + f_1 + f_2;$$

deoarece T_{Huff} poate fi obținut din T_{Huff}^0 înlocuind frunza corespunzătoare frecvenței $f_1 + f_2$ cu un nod intern având ca fiice frunzele de frecvențe f_1 și f_2 . De asemenea, conform ipotezei de inducție T_{Huff}^0 este un arbore de codi care este optim pentru un alfabet de $n - 1$ simboluri cu frecvențele $f_1 + f_2; f_3; \dots; f_n$. Fie T_{opt} un arbore de codi care este optim satisfăcând lema anterioară, adică frunzele de frecvențe f_1 și f_2 sunt frați în T_{opt} . Fie T^0 arborele obținut din T_{opt} prin înlocuirea frunzelor de frecvențe f_1 și f_2 și a tatălui lor cu o singură frunză de frecvență $f_1 + f_2$. Atunci

$$COST(T_{opt}) = COST(T^0) + f_1 + f_2$$

$$COST(T_{Huff}^0) + f_1 + f_2 = COST(T_{Huff});$$

deoarece $COST(T^0) = COST(T_{Huff}^0)$ conform ipotezei de inducție. Rezultă de aici

$$COST(T_{opt}) = COST(T_{Huff});$$

Capitolul 3

Tehnici de sortare

3.1 Heapsort

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

$$A[\lfloor k/2 \rfloor] \leq A[k]; \quad 2 \leq k \leq n:$$

Folosim notația $\lfloor x \rfloor$ 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 și **dimensiune-heap**[A] reprezintă numărul elementelor heap-ului memorat în vectorul A . Astfel, chiar dacă $A[1::\text{lungime}[A]]$ conține numai elemente al sau date valide, este posibil ca elementele următoare elementului $A[\text{dimensiune-heap}[A]]$; unde $\text{dimensiune-heap}[A] \leq \text{lungime}[A]$, să nu aparțină heap-ului.

Structurii de **heap** îi se atașează în mod natural un arbore binar aproape complet (Figura (3.1)).

Fiecare nod al arborelui corespunde unui element al vectorului care conține valorile atașate nodurilor. Radacina arborelui este $A[1]$: Dat fiind un indice i , corespunzător unui nod, se poate determina ușor indicii nodului tată, $\text{TATA}(i)$ și ai fiilor $\text{STANG}(i)$ și $\text{DREPT}(i)$:

indicele $\text{TATA}(i)$

returnează $\lfloor i/2 \rfloor$ (partea întreagă a lui $i/2$);

indicele $\text{STANG}(i)$

returnează $2i$;

indicele $\text{DREPT}(i)$

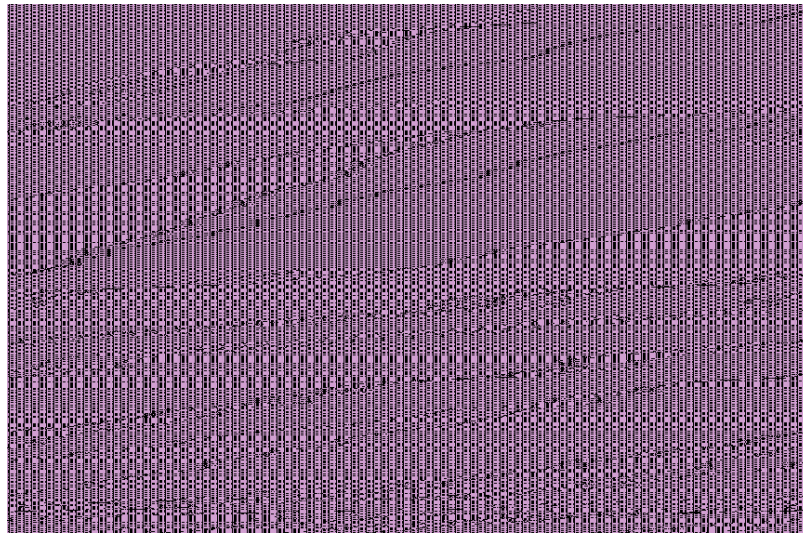


Figura 3.1: Exemplu de heap reprezentat sub forma unui arbore binar si sub forma unui vector

returneaza $2i + 1$:

Pentru orice nod i diferit de radacina este, în virtutea de nitiei heap-ului, adevarata **proprietatea de heap**:

$$A[\text{TATA}(i)] \geq A[i]:$$

De asemenea, înălțimea unui nod al arborelui este numărul muchiilor celui mai lung drum ce nu vizitează tatăl nodului și leagă nodul respectiv cu o frunză. Evident, înălțimea arborelui este înălțimea radacinii.

3.1.1 Reconstituirea proprietatii de heap

Funcția **ReconstituieHeap** este un subprogram important în prelucrarea heap-urilor. Datele de intrare sunt un vector A și un indice i . Atunci când se apelează **ReconstituieHeap** se presupune că subarborii având ca radacini nodurile $\text{STANG}(i)$ și $\text{DREPT}(i)$ sunt heap-uri. Dar cum elementul $A[i]$ poate fi mai mic decât descendenții săi, este posibil ca acesta să nu respecte **proprietatea de heap**. Sarcina funcției **ReconstituieHeap** este să scufunde în heap valoarea $A[i]$; astfel încât subarborile care au în radacina valoarea elementului de indice i , să devină un heap.

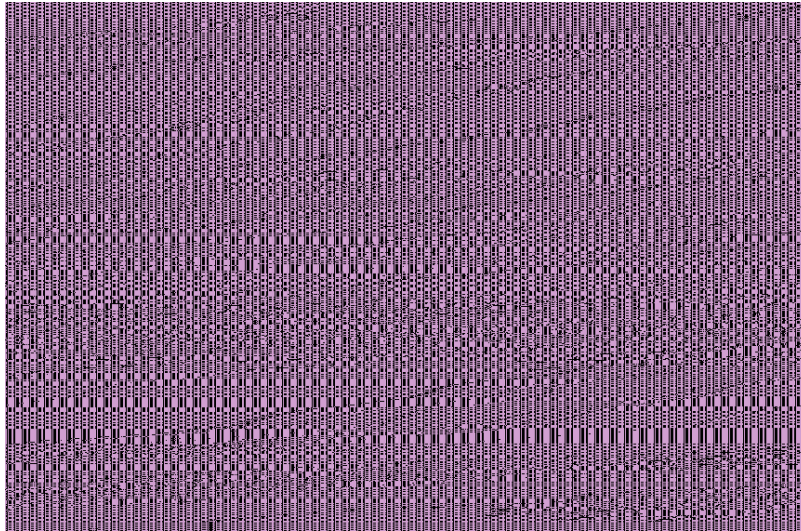


Figura 3.2: Efectul functiei ReconstituieHeap

Urmeaza functia scrisa in pseudocod.

ReconstituieHeap(A,i)

st STANG(i)

dr DREPT(i)

daca st dimensiune-heap[A] si $A[st] > A[i]$ atunci

maxim st

altfel

maxim i

daca dr dimensiune-heap[A] si $A[dr] > A[i]$ atunci

maxim dr

daca maxim $\neq i$ atunci

schimba $A[i]$ si $A[\text{maxim}]$

ReconstituieHeap(A,maxim)

In gura (3.2) este ilustrat efectul functiei **ReconstituieHeap**.

In gura (3.2, a) este desenata configuratia initiala a heap-ului unde $A[2]$ nu respecta proprietatea de heap deoarece nu este mai mare decat descendentii sai. Proprietatea de heap este restabilita pentru nodul 2 in gura (3.2, b) prin interschimbarea lui $A[2]$ cu $A[4]$; ceea ce anuleaza proprietatea de heap pentru nodul 4. Apeland recursiv **ReconstituieHeap(A; 4)** se pozitioneaza valoarea lui i pe 4. Dupa interschimbarea lui $A[4]$ cu $A[9]$

asa cum se vede în figura (3.2, c), nodul 4 ajunge la locul sau si apelul recursiv **ReconstituieHeap**(A; 9) nu mai gaseste elemente care nu îndeplinesc proprietatea de heap.

3.1.2 Constructia unui heap

Funcția **ReconstituieHeap** poate fi utilizata de jos în sus pentru transformarea vectorului $A[1::n]$ în heap.

Cum toate elementele subsirului $A[bn=2c+1::n]$ sunt frunze, ele pot fi considerate heap-uri formate din cate un element. Funcția **ConstruiesteHeap** pe care o prezentam în continuare traverseaza restul elementelor si executa funcția **ReconstituieHeap** pentru fiecare nod întâlnit. Ordinea de prelucrare a nodurilor satisface cerinta ca subarborii, având ca radacina descendenți ai nodului i sa formeze heap-uri înainte ca **ReconstituieHeap** sa fie executat pentru aceste noduri.

Urmeaza, în pseudocod funcția **ConstruiesteHeap**:

```
dimensiune-heap[A] lungime[A]
pentru i ← lungime[A]=2c, 1 executa
    ReconstituieHeap(A, i).
```

Figura (3.2) ilustreaza modul de aplicare a funcției **ConstruiesteHeap**. În figura se vizualizeaza structurile de date (heap-urile) în starea lor anterioara apelarii funcției **ReconstituieHeap**. (a) Se considera vectorul A cu 10 elemente si arborele binar corespunzator. Se observa ca variabila de control i a ciclului în momentul apelului funcției **ReconstituieHeap(A, i)** indica nodul 5. (b) reprezinta rezultatul; variabila de control i a ciclului indica acum nodul 4. (c)-(e) vizualizeaza iteratiile succesive ale ciclului **pentru** din **ConstruiesteHeap**. Se observa ca atunci când se apeleaza funcția **ReconstituieHeap** pentru un nod dat, subarborii acestui nod sunt deja heap-uri. (f) prezinta heap-ul final.

3.1.3 Algoritmul heapsort

Algoritmul de sortare **heapsort** începe cu apelul funcției **ConstruiesteHeap** în scopul transformării vectorului de intrare $A[1::n]$ în heap. Deoarece cel mai mare element al vectorului este atasat nodului radacina $A[1]$; acesta va ocupa locul de nivel în vectorul ordonat prin interschimbarea sa cu $A[n]$. Mai departe, excluzând din heap elementul de pe locul n , si micșorând cu 1 dimensiune-heap[A], restul de $A[1::(n-1)]$ elemente se pot transforma ușor

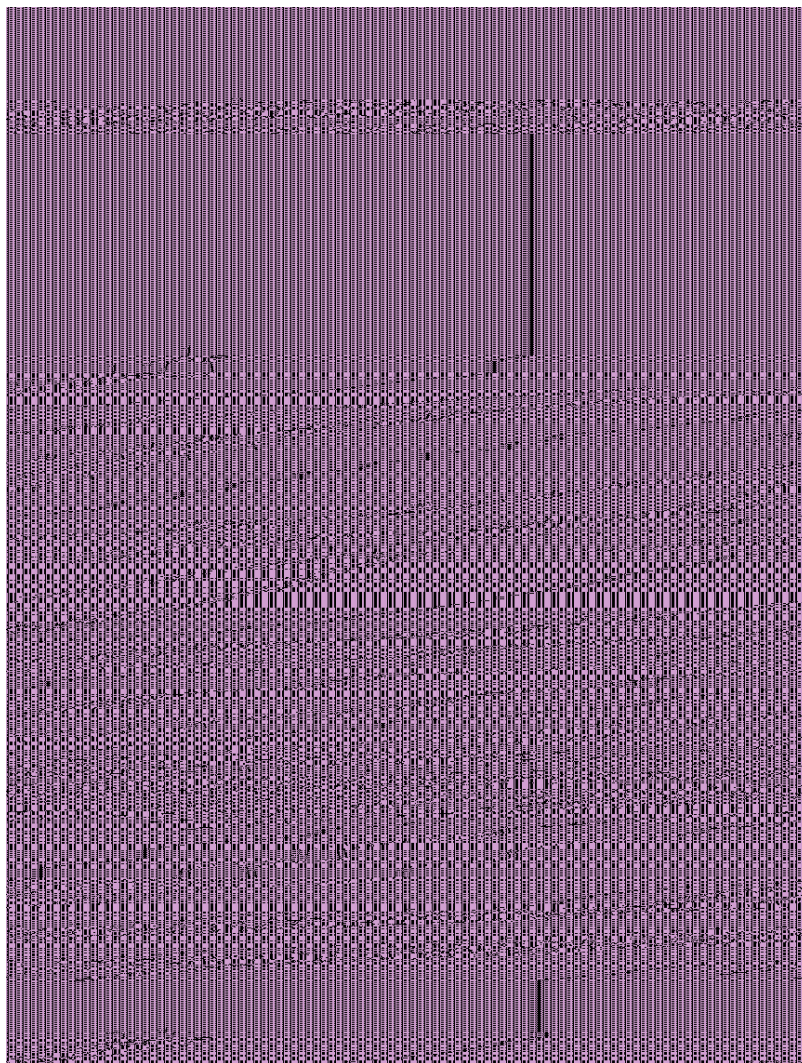


Figura 3.3: Model de executie a functiei ConstruiesteHeap

n heap, deoarece subarborii nodului radacina au proprietatea de heap, cu eventuala exceptie a elementului ajuns n nodul radacina. Pentru aceasta se apeleaza functia **ReconstituieHeap(A,1)**. Procedeu se repeta micorand dimensiunea heap-ului de la n - 1 la 2.

Urmeaza, scris n pseudocod, algoritmul descris de functia **Heapsort(A)**:

```

ConstruiesteHeap(A),
pentru i = lungime[A]-2 executa
schimba A[i] S A[1]
dimensiune-heap[A] = dimensiune-heap[A]-1
ReconstituieHeap(A,i)

```

Vom scrie programul **heapsort** si n C + +. Fata de descrierea de mai nainte a algoritmului, vor interveni cateva mici modi cari datorate faptului ca n C + + indexarea elementelor unui vector incepe de la 0 si 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 )
{
    for (i = (dimensiune - 2)/2; i >= 0; i )
        ReconstituieHeap(A, i, dimensiune);
}
/*****/

void heapsort( int* A, int 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]<< " "; g

```

Timpul de executie

Funcția **ReconstituieHeap** consta din comparatii si interschimbari ale de elemente a câte pe nivele consecutive. Timpul de executie al funcției va deci de ordinul $n \lg n$ al înălțimii arborelui binar asociat care este de ordinul $O(\lg n)$ unde n este numărul de elemente din heap.

Funcția **ConstruiesteHeap** apelează funcția **ReconstituieHeap** de n ori. Deducem de aici ca timpul de executie al funcției **ConstruiesteHeap** este de ordinul $O(n \lg n)$: Funcția **heapsort** apelează o dată funcția **ConstruiesteHeap** si de $n - 1$ ori funcția **ReconstituieHeap**. Rezulta de aici ca timpul de executie al funcției **heapsort** este $O(n \lg n) + (n - 1)O(\lg n) = O(n \lg n)$:

3.2 Cozi de prioritati

Vom prezenta în cele ce urmează încă o aplicație a notiunii de heap: utilizarea lui sub forma de coada cu prioritati.

Coada cu prioritati este o structura de date care contine o multime S de elemente, fiecare având asociată o valoare numită cheie. Asupra unei cozi cu prioritati se pot efectua următoarele operații:

Insereaza($S; x$) inserează elementul x în mulțimea S . Această operație este scrisă în felul următor $S \leftarrow S \cup \{x\}$:

ExtrageMax(S) elimina și întoarce elementul din S având cheia cea mai mare.

O aplicație a cozilor de prioritati o constituie planul de lucru pe calculatoare partajate. Lucrările care trebuie efectuate și prioritățile relative se memorează într-o coadă de prioritati. Când o lucrare este terminată sau întreruptă, funcția **ExtrageMax** va selecta lucrarea având prioritatea cea mai mare dintre lucrările în așteptare. Cu ajutorul funcției **Insereaza** oricând se introduce în coadă o sarcină nouă.

În mod natural o coadă cu priorități poate fi implementată utilizând un heap. Funcția **Insereaza**(S; x) inserează un element în heap-ul S. La prima expansiune a heap-ului se adaugă o frunză arborelui. Apoi se traversează un drum pornind de la această frunză către rădăcina în scopul găsirii locului de nivel al noului element.

Urmărește instrucțiunile funcției **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 funcției **Heapsort**, instrucțiunile sale în pseudocod sunt:

```

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.

```

Urmărește scris în C++ un program care implementează o coadă cu priorități.

```

#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;g
/*****/
void Insereaza(int* A, int cheie, int dimensiune )
fint i,tata;
i= dimensiune+ 1;
A[i]= cheie;tata= (i-1)/2;
while(i> 0&&A[tata]< cheie)
fA[i]= A[tata];i= tata;A[i]= cheie;tata= (i-1)/2;g g
/*****/
void main() f
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')f
cout<< Extragem sau inseram[e/i] << endl;
cin>> y;
switch (y)
fcase 'e':
max= ExtrageMax( S,dimensiune );
dimensiune= dimensiune-1;
if (dimensiune>= 0) fcout<< 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;g
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;g
cout<< Intrerupem?[d/n] <<endl;
cin>>x;gg
```

3.3 Sortarea rapida

Sortarea rapida este un algoritm care sorteaza pe loc (n spatiul alocat sirului de intrare). Cu acest algoritm, un sir de n elemente este sortat intr-un timp de ordinul $O(n^2)$, in cazul cel mai defavorabil. Algoritmul de sortare rapida este deseori cea mai buna solutie practica deoarece are o comportare medie remarcabila: timpul sau mediu de executie este $O(n \ln n)$ si constanta ascunsa in formula $O(n \ln n)$ este destul de mica.

3.3.1 Descrierea algoritmului

Ca si algoritmul de sortare prin interclasare, algoritmul de sortare rapida ordoneaza un sir $A[p:r]$ folosind tehnica **divide si stapaneste**:

Divide : Sirul $A[p:r]$ este rearanjat in doua subsiruri nevide $A[p:q]$ si $A[q+1:r]$ astfel incat fiecare element al subsirului $A[p:q]$ sa fie mai mic sau egal cu fiecare element al subsirului $A[q+1:r]$: Indicele q este calculat de procedura de partitionare.

Stapaneste: Cele doua subsiruri $A[p:q]$ si $A[q+1:r]$ sunt sortate prin apeluri recursive ale algoritmului de sortare rapida.

Combina: Deoarece cele doua subsiruri sunt sortate pe loc, sirul $A[p:r] = A[p:q] \cup A[q+1:r]$ este ordonat.

Algoritmul (intitulat **QUICKSORT**($A; p; r$)) in pseudocod este urmatorul:

```
QUICKSORT( $A; p; r$ ):
//urmeaza algoritmul
daca  $p < r$  atunci
   $q \leftarrow$  PARTITIE( $A; p; r$ )
  QUICKSORT( $A; p; q$ )
  QUICKSORT( $A; q + 1; r$ ).
```

Pentru ordonarea sirului A se apeleaza **QUICKSORT**($A; 1, \text{lungime}[A]$).

O alta functie a algoritmului este functia **PARTITIE**($A; p; r$) care rearanjeaza pe loc sirul $A[p:r]$; returnand si indicele q mentionat mai sus:

```
PARTITIE( $A; p; r$ )
//urmeaza functia
```

```

x  A[p]
i  p
j  r
cat timp i<j executa
frepetă
j j-1
pana cand A[j] < x
repetă
i i+1
pana cand A[i] < x
daca i<j atunci
interschimba A[i] & A[j]; j j-1
returnează 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++)
    fcout<< A[ i] = *(A+i);g
Quicksort(A,0,n-1);
for(i=0;i<n;i++)
    cout<< A[ i] = A[i];g

```

3.3.2 Performanta algoritmului de sortare rapida

Timpul de executie al algoritmului depinde de faptul ca partitionarea este echilibrata sau nu.

Cazul cel mai defavorabil

Vom demonstra ca cea mai defavorabila comportare a algoritmului de sortare rapida apare in situatia in care procedura de partitionare produce un vector de $n-1$ elemente si altul de 1 element. Mai ntai observam ca timpul de executie al functiei **Partitie** este $O(n)$: Fie $T(n)$ timpul de executie al algoritmului de sortare rapida. Avem pentru partitionarea amintita mai nainte, formula de recurenta

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

de unde rezulta

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

adica, pentru un $c > 0$ su cient de mare,

$$T(n) \geq cn^2; \quad (3.1)$$

Vom arata prin inductie ca estimarea (3.1) este valabila pentru orice partitionare.

Sa presupunem ca partitionare produce subvectori de dimensiuni q si $n-q$: Cu ipoteza de inductie avem

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

$$c_1 \max_{1 \leq q \leq n-1} q^2 + (n-q)^2 + O(n) = c_1 n^2 - 2n + 2 + O(n) \geq cn^2;$$

Cazul cel mai favorabil

Daca functia de partitionare produce doi vectori de $n/2$ elemente, algoritmul de sortare rapida lucreaza mult mai repede. Formula de recurenta in acest caz

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

conduce, dupa cum s-a aratat in cazul algoritmului de insertie prin interclasare la un timp de executie de ordinul $O(n \ln n)$:

Estimarea timpului de executie mediu

Timpul de executie mediu se poate determina prin inductie cu formula

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

Presupunem ca

$$T(n) = an \ln n + b;$$

pentru un $a > 0$ si $b > T(1)$:

Pentru $n > 1$ avem

$$\begin{aligned} T(n) &= \frac{2}{n} \sum_{k=1}^{n-1} T(k) + O(n) = \frac{2}{n} \sum_{k=1}^{n-1} (ak \ln k + b) + O(n) = \\ &= \frac{2a}{n} \sum_{k=1}^{n-1} k \ln k + \frac{2b}{n} (n-1) + O(n); \end{aligned}$$

Tinand cont de inegalitatea

$$\sum_{k=1}^{n-1} k \ln k \geq \frac{1}{2} n^2 \ln n - \frac{1}{8} n^2;$$

obtinem

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

$$an \ln n - \frac{a}{4} n + 2b + O(n) = an \ln n + b + O(n) + b - \frac{a}{4} n \geq an \ln n + b;$$

deoarece valoarea lui a poate fi aleasa astfel incat $\frac{an}{4}$ sa domine expresia $O(n) + b$: Tragem deci concluzia ca timpul mediu de executie a algoritmului de sortare rapida este $O(n \ln n)$:

3.4 Metoda bulelor (bubble method)

Principiul acestei metode de sortare este urmatorul: pornind de la ultimul element al sirului catre primul, se schimba ntre ele fiecare element cu cel anterior, daca elementul anterior (de indice mai mic) este mai mare. In felul acesta primul element al sirului este cel mai mic element. Se repeta procedura pentru sirul format din ultimele $n - 1$ elemente si asa mai departe, obtinandu-se n final sirul de n elemente ordonat. Numarul de comparatii si deci timpul de executie este de ordinul $O(n^2)$:

Urmeaza programul scris n C++:

```
#include <iostream.h>
//returneaza p,q in ordine crescatoare
/*****/
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)
    {
        for (i=0; i<n; i++)
            for (j=n-1; j>i; 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 cautare

4.1 Algoritmi de cautare

Vom presupune ca în memoria calculatorului au fost stocate un număr de n înregistrări și ca dorim să localizăm o anumită înregistrare. Ca și în cazul sortării, presupunem că fiecare înregistrare conține un câmp special, numit cheie. Colectia tuturor înregistrărilor se numește tabel sau șir. Un grup mai mare de șiruri poartă numele de bază de date.

În cazul unui algoritm de cautare se consideră un anumit argument K ; iar problema este de a găsi acea înregistrare a cărei cheie este tocmai K . Sunt posibile două cazuri: cautarea cu succes (reusită), când înregistrarea cu cheia avută în vedere este depistată, sau cautarea fără succes (nereusită), când cheia niciunei înregistrări nu coincide cu cheia după care se face cautarea. După o cautare fără succes, uneori este dorit să inserăm o nouă înregistrare, conținând cheia K în tabel; metoda care realizează acest lucru se numește algoritmul de cautare și inserție.

Deși scopul cautării este de a găsi informația din înregistrarea asociată cheii K , algoritmi pe care îi vom prezenta în continuare, țin în general cont numai de cheie, ignorând celelalte câmpuri.

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

Există de asemenea o importantă interacțiune între sortare și cautare, după cum vom vedea în cele ce urmează.

4.1.1 Algoritmi de cautare secventiala (pas cu pas)

Algoritmul S (cautare secventiala). Fiind dat un tabel de nregistrari $R_1; R_2; \dots; R_n$, $n \geq 1$; avand cheile corespunzatoare $K_1; K_2; \dots; K_n$, este cautata nregistrarea corespunzatoare cheii K . Vom mai introduce o nregistrare fictiva R_{n+1} cu proprietatea ca valoarea cheii K_{n+1} este atat de mare incat K nu va capata nici-o data aceasta valoare (punem formal $K_{n+1} = \infty$). Urmeaza algoritmul scris in pseudocod

```

i ← 0
Executa
i ← i + 1
Cat timp i ≤ n si K ≠ Ki
Daca K = Ki cautarea a avut succes
Altfel, cautarea nu a avut succes.

```

In cazul in care cheile sunt ordonate crescator, o varianta a algoritmului de cautare secventiala este

Algoritmul T (cautare secventiala intr-un tabel ordonat):

```

i ← 0
Executa
i ← i + 1
Cat timp i ≤ n si K > Ki
Daca K = Ki cautarea a avut succes
Altfel, cautarea nu a avut succes.

```

Sa notam cu p_i probabilitatea ca sa avem $K = K_i$; unde $p_1 + p_2 + \dots + p_n = 1$: Daca probabilitatile sunt egale, in medie, algoritmul **S** consuma acelasi timp ca si algoritmul **T** pentru o cautare cu succes. Algoritmul **T** efectueaza insa in medie de doua ori mai repede cautarile fara succes.

Sa presupunem mai departe ca probabilitatile p_i nu sunt egale. Timpul necesar unei cautari cu succes este proportional cu numarul de comparatii, care are valoarea medie

$$\bar{C}_n = p_1 + 2p_2 + \dots + np_n$$

Evident, \bar{C}_n ia cea mai mica valoare atunci cand $p_1 = p_2 = \dots = p_n$, adica atunci cand cele mai utilizate nregistrari apar la inceput. Daca $p_1 = p_2 = \dots = p_n = 1/n$; atunci

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

O repartitie interesanta a probabilitatilor este **legea lui Zipf** care a observat ca n limbajele naturale, cuvantul a at pe locul n n ierarhia celor mai utilizate cuvinte apare cu o frecventa invers proportionala cu n:

$$p_1 = \frac{c}{1}; p_2 = \frac{c}{2}; \dots; p_n = \frac{c}{n}; c = \frac{1}{H_n}; H_n = 1 + \frac{1}{2} + \dots + \frac{1}{n}:$$

Daca legea lui Zipf guverneaza frecventa cheilor ntr-un tabel, atunci

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

iar cautarea ntr-o astfel de circumstanta, este de circa $\frac{1}{2} \ln n$ ori mai rapida decat cautarea n cazul general.

4.1.2 Cautarea n tabele sortate (ordonate)

In cele ce urmeaza vom discuta algoritmi de cautare pentru tabele ale caror chei sunt ordonate. Dupa compararea cheii date K cu o cheie K_i a tabelului, cautarea continua n trei moduri diferite dupa cum $K < K_i$; $K = K_i$ sau $K > K_i$: Sortarea tabelelor (listelor) este recomandabila n cazul cautarilor repetate. De aceea n aceasta subsectiune vom studia metode de cautare n tabele ale caror chei sunt ordonate $K_1 < K_2 < \dots < K_n$: Dupa compararea cheilor K si K_i ntr-o tabela ordonata putem avea $K < K_i$ (caz n care $R_i; R_{i+1}; \dots; R_n$ nu vor mai luate n considerare), $K = K_i$ (n acest caz cautarea se termina cu succes) sau $K > K_i$ (caz n care $R_1; R_2; \dots; R_i$ nu vor mai luate n considerare).

Faptul ca o cautare, chiar fara succes duce la eliminarea unora din cheile cu care trebuie comparata K; duce la o eficientizare a cautarii.

Vom prezenta mai departe un algoritm general de cautare ntr-un tabel sortat. Fie $S = \{K_1 < K_2 < \dots < K_n\}$ stocata ntr-un vector K [1::n], adica $K[i] = K_i$ si e o cheie a. Pentru a decide daca $a \in S$; comparam a cu un element al tabelului si apoi continuam cu partea superioara sau cea inferioara a tabelului. Algoritmul (numit n cele ce urmeaza **algoritmul B**) scris n pseudocod este:

```

prim 1
ultim n
urmator un ntreg n intervalul [prim,ultim]
executa
```

fdaca $a < K[\text{urmator}]$ atunci $\text{ultim} = \text{urmator} - 1$
 altfel $\text{prim} = \text{prim} + 1$
 $\text{urmator} = \text{un_ntreg_n_intervalul}[\text{prim}, \text{ultim}]$
 cat timp $a \notin K[\text{urmator}]$ si $\text{ultim} > \text{prim}$
 daca $a = K[\text{urmator}]$ atunci avem cautare cu succes
 altfel avem cautare fara succes.

In cazul $\text{un_ntreg_n_intervalul}[\text{prim}, \text{ultim}] = \text{prim}$ spunem ca avem o cautare liniara.

Daca $\text{un_ntreg_n_intervalul}[\text{prim}, \text{ultim}] = d(\text{prim} + \text{ultim})/2$ spunem ca avem o cautare binara.

Amintim ca folosim notatia $b \mid c$ pentru partea ntreaga a unui numar, nt timp ce $d \mid e$ are urmatoarea semnificatie

$$d \mid e = \begin{cases} a & \text{daca } a \text{ este } \text{ntreg}; \\ bac + 1 & \text{daca } a \text{ nu este } \text{ntreg}; \end{cases}$$

Prezentam in continuare un proiect pentru cautarea intr-o lista ordonata (cu relatia de ordine lexicografica) de nume, pentru a afla pe ce pozitie se afla un nume dat. Proiectul contine programele **cautare.cpp**, **fcautare.cpp** (in care sunt descrise functiile folosite de **cautare.cpp**) si fisierul de nume scris in ordine lexicografica **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", marimeg
//functii implementate in fcautare.cpp
void GasestePrimul(int, int *);
void GasesteToate(int, int *);
void Binar(int, int, int *);
void main() f
GetList(); // citeste numele din  sier
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]) f
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); g
printf( Inca o incercare (d/n)? ); scanf( %s, DaNu);
g
g
/*****fcautare.cpp*****/

```

```

/*****
!* Functii de cautare intr-o lista de nume *
!* ce urmeaza a 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) f
// Cautare secventiala intr-o lista de nume pentru
// a a a 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;
g
void GasesteToate(int marime, int *cate) f
// Cautare secventiala intr-o lista de nume pentru
// a a a toate aparitiile cheii.
int cat, i;
cat= 0;
for (i= 0; i< marime; i+ +)
if (strcmp(a[i],cheie)= = 0) f
printf( %s pe pozitia %d.\n",cheie,i + 1);
cat+ +;
g
*cate = cat;
g
void Binar(int prim, int ultim, int *unde) f
// Cautare binara intr-o lista ordonata pentru o aparitie
// a cheii speci cate. 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; g
*unde = iun+ 1;
g

```

4.1.3 Arbori de decizie asociati cautarii binare

Pentru a ntelege mai bine ce se ntampla n cazul algoritmului de cautare binara, vom construi **arborele de decizie asociat cautarii binare**.

Arborele binar de decizie corespunzator unei cautari binare cu n nregistrari poate construit dupa cum urmeaza:

Daca $n = 0$, arborele este format din frunza [0]. Altfel nodul radacina este $dn=2e$; subarborele stang este arborele binar construit asemanator cu $dn=2e-1$ noduri iar subarborele drept este arborele binar construit asemanator cu $dn=2e$ noduri si cu indicii nodurilor incrementati cu $dn=2e$: Am avut n vedere numai nodurile interne corespunzatoare unei cautari cu succes.

Prezentam mai jos un arbore de cautare binara pentru $n = 16$ (gura 4.1).

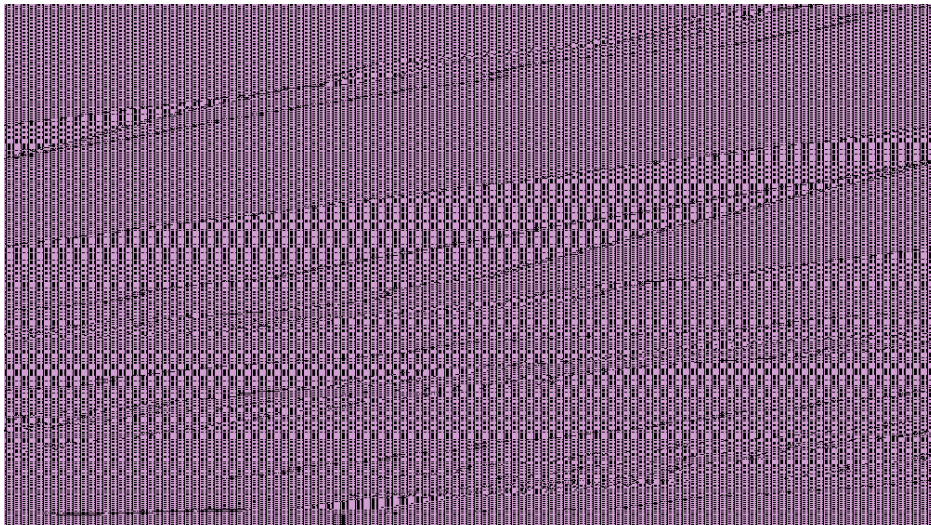


Figura 4.1: Arbore de cautare binara

Prima comparatie efectuata este $K : K_8$ care este reprezentata de nodul (8) din gura. Daca $K < K_8$, algoritmul urmeaza subarborele stang iar daca $K > K_8$, este folosit subarborele drept. O cautare fara succes va conduce la una din frunzele numerotate de la 0 la n ; de exemplu ajungem la frunza [6] daca si numai daca $K_6 < K < K_7$:

4.1.4 Optimalitatea cautarii binare

Vom face observatia ca orice arbore binar cu n noduri interne (etichetate cu (1); (2); (3); ... (n)) si deci $n + 1$ frunze (etichetate cu [0]; [1]; [2]; ... [n - 1]; [n]), corespunde unei metode valide de cautare intr-un tabel ordonat daca parcurs in ordine simetrica obtinem [0] (1) [1] (2) [2] (3) ... [n - 1] (n) [n]. Nodul intern care corespunde n **Algoritmul B** lui urmator[prim,ultim] va radacina subarborelui care are pe [ultim] drept cea mai din dreapta frunza iar pe [prim] drept cea mai din stanga frunza. De exemplu in gura 4.2, a) urmator[0,4]=2, urmator[3,4]=4, pe cand in gura 4.2, b) urmator[0,4]=1, urmator[3,4]=4.

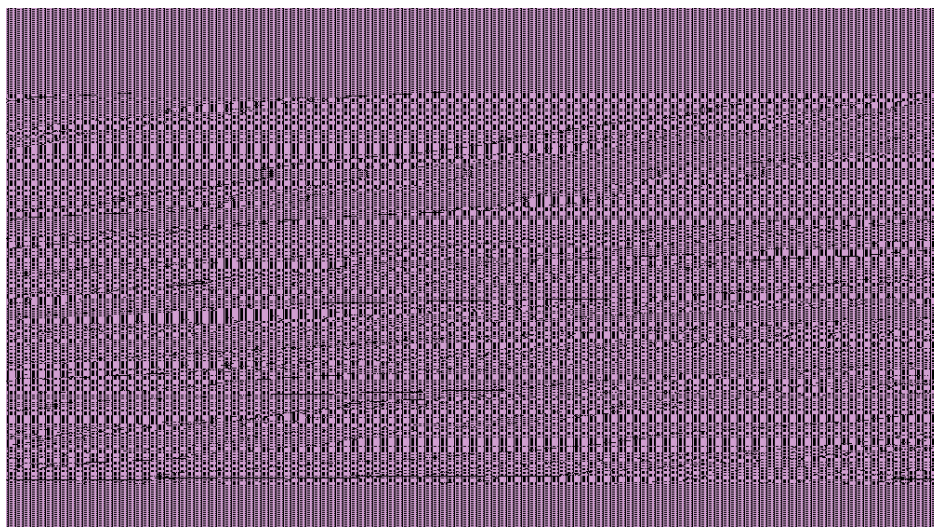


Figura 4.2: Arbori de cautare

Vom demonstra in cele ce urmeaza ca intre arborii de decizie asociati algoritmului **B** de cautare, cei optimi sunt arborii corespunzatori cautarii binare. Vom introduce mai intai doua numere ce caracterizeaza 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 și numai dacă toate frunzele lui T se află pe două nivele consecutive (cu $2^{q-1} \leq N < 2^q$ frunze pe nivelul $q-1$ și $2^q - N + 1$ frunze pe nivelul q , unde $q = \lceil \log_2 N \rceil$; nivelul rădăcinii fiind 0).

Demonstratie. Să presupunem că arborele binar T are frunzele u și v (ce sunt fratele și sora lor), pe nivelul L iar frunzele y și z pe nivelul l astfel ca $L > l + 2$: Vom construi un alt arbore binar T_1 (vezi figura 4.3) transferând pe u și v

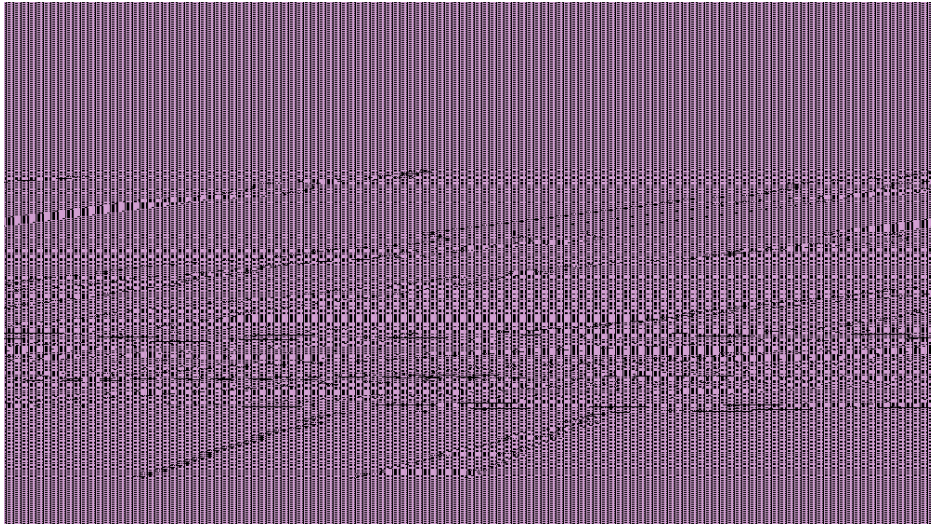


Figura 4.3: Optimizarea lungimii drumurilor externe

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

$$E(T) - E(T_1) = 2L - (L - 1) + 1 - 2(l + 1) = L - l - 1 > 0;$$

și 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 $q - 1$ și q , unde $q = \lceil \log_2 N \rceil$:

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

Lema 4. Daca $2^{k-1} \leq n < 2^k$; o cautare cu succes folosind algoritmul **B** necesita cel mult k comparatii. Daca $n = 2^k - 1$; o cautare fara succes necesita k comparatii iar daca $2^{k-1} \leq n < 2^k - 1$, o cautare fara succes necesita sau $k - 1$ sau k comparatii. Aceasta inseamna ca 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 gasesc pe nivelele $k - 1$ si k .

Demonstratie. Lema este adevarata pentru $k = 1$. Fie $k \geq 2$ si sa presupunem (ipoteza de inductie) ca teorema este adevarata pentru orice $2^{k-1} \leq n < 2^k$. Daca $2^k \leq n < 2^{k+1}$ vom avea doua cazuri: (I) n este impar, adica $n = 2p + 1$; (II) n este par adica $n = 2p$, unde $p \in \mathbb{N}$; $p \geq 1$:

Cazul (I): Radacina arborelui binar T , corespunzator algoritmului **B** are eticheta $p + 1$ iar subarborele stang T_l si subarborele drept T_r contin cate p noduri interne. Din relatia

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

rezulta ca

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

Aplicand ipoteza de inductie ambilor subarbori T_l si T_r avem $h(T_l) = h(T_r) = k$ deci $h(T) = k + 1$; naltimea lui T ind numarul maxim de comparatii ale cheilor efectuate de algoritmul **B** in cazul unei cautari cu succes. Daca $p = 2^k - 1$ toate frunzele lui T_l si T_r se afla pe nivelul k ; deci daca $n = 2p + 1 = 2^{k+1} - 1$ toate frunzele lui T se vor afla pe nivelul $k + 1$. Daca $2^{k-1} \leq p < 2^k - 1$, ceea ce implica $2^k \leq n < 2^{k+1} - 1$, cum (cu ipoteza de inductie) atat T_l cat si T_r au frunzele pe nivelele $k - 1$ sau k ; deducem ca T va avea frunzele pe nivelele k sau $k + 1$:

Cazul (II): In acest caz radacina arborelui binar are eticheta p , T_l are $p - 1$ noduri interne iar T_r are p noduri interne. Cum $2^k \leq 2p < 2^{k+1}$ rezulta ca $2^{k-1} \leq p < 2^k$. Avem de asemenea $2^{k-1} \leq p - 1 < 2^k - 1$ n afara cazului cand $p = 2^{k-1}$ si deci $n = 2^k$. In acest ultim caz arborele T este asemanator cu arborele din figura 4.1: el are $h(T) = k + 1$; $N = 1$ frunze pe nivelul k si doua frunze pe nivelul $k + 1$; teorema a fost demonstrata direct in aceasta circumstanta ($n = 2^k$). Ne mai ramane sa consideram cazul $2^k < n < 2^{k+1}$. Cu ipoteza de inductie deducem ca $h(T_l) = h(T_r) = k$ deci $h(T) = k + 1$ iar algoritmul **B** necesita cel mult $k + 1$ comparatii pentru o cautare cu succes.

Cum n este par, $n \in 2^s - 1$; $s \geq 1$; avem de aratat ca frunzele lui T se afla pe nivelele k sau $k + 1$: Aceasta rezulta din aplicarea ipotezei de inductie la subarborii T_l si T_r care au $p - 1$ respectiv p noduri interne: Intr-adevar, cum $p - 1 \in 2^k - 1$ rezulta ca frunzele lui T_l se afla pe nivelele $k - 1$ sau k : Pentru T_r toate frunzele se afla pe nivelul k (daca $p = 2^k - 1$) sau pe nivelele $k - 1$ sau k . Rezulta ca frunzele lui T se afla pe nivelele k sau $k + 1$.

Deducem ca numarul de comparatii in cazul unei cautari (cu sau fara succes) este de cel mult $\log_2 Nc + 1$:

Vom demonstra in continuare

Lema 5. Pentru orice arbore binar complet T este satisfacuta relatia

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

unde N reprezinta numarul de noduri interne al lui T .

Demonstratie. Sa presupunem ca arborele binar T are i_j noduri interne si e_j noduri externe (frunze) la nivelul j ; $j = 0; 1; \dots$ (radacina este la nivelul 0). De exemplu in figura 4.2, a) avem $(i_0; i_1; i_2; \dots) = (1; 2; 1; 0; 0; \dots)$; $(e_0; e_1; e_2; \dots) = (0; 0; 3; 2; 0; 0; \dots)$ iar in figura 4.2, b) avem $(i_0; i_1; i_2; \dots) = (1; 1; 1; 1; 0; 0; \dots)$; $(e_0; e_1; e_2; \dots) = (0; 1; 1; 1; 2; 0; 0; \dots)$:

Consideram functiile generatoare asociate acestor siruri

$$A(x) = \sum_{j=0}^{\infty} i_j x^j; \quad B(x) = \sum_{j=0}^{\infty} e_j x^j;$$

unde numai un numar finit de termeni sunt diferiti de 0. Este valabila relatia

$$2i_{j-1} = i_j + e_j; \quad j = 0; 1; \dots;$$

deoarece toate cele i_{j-1} noduri interne de la nivelul $j - 1$ au fiecare in parte cate 2 fiice pe nivelul j si numarul total al acestor fiice este $i_j + e_j$. Rezulta de aici ca

$$\begin{aligned} A(x) + B(x) &= \sum_{j=0}^{\infty} (i_j + e_j) x^j = i_0 + e_0 + \sum_{j=1}^{\infty} (i_j + e_j) x^j = \\ &= 1 + 2 \sum_{j=1}^{\infty} i_{j-1} x^j = 1 + 2x \sum_{j=1}^{\infty} i_{j-1} x^{j-1} = 1 + 2x \sum_{j=0}^{\infty} i_j x^j; \end{aligned}$$

adica

$$A(x) + B(x) = 1 + 2xA(x); \quad (4.1)$$

Pentru $x = 1$ se obtine $B(1) = 1 + \sum_{j=0}^1 j$, dar $B(1) = \sum_{j=0}^1 j$ este numarul de frunze ale lui T iar $A(1) = \sum_{j=0}^1 j$ este numarul de noduri interne, deci numarul de noduri interne este cu 1 mai mic decat numarul de noduri externe. Derivand relatia (4.1) obtinem

$$\begin{aligned} A^0(x) + B^0(x) &= 2A(x) + 2xA^0(x); \\ B^0(1) &= 2A(1) + A^0(1); \end{aligned}$$

Cum $A(1) = N$; $A^0(1) = \sum_{j=0}^1 j = I(T)$; $B^0(1) = \sum_{j=0}^1 j = E(T)$; deducem relatia

$$E(T) = I(T) + 2N; \quad (4.2)$$

Reprezentarea sub forma de arbore binar a algoritmului binar de cautare **B**, ne sugereaza cum sa calculam intr-un mod simplu numarul mediu de comparatii. Fie C_N numarul mediu de comparatii in cazul unei cautari reusite si C_N^0 numarul mediu de cautari in cazul unei incercari nereusite. Avem

$$C_N = 1 + \frac{I(T)}{N}; \quad C_N^0 = \frac{E(T)}{N+1}; \quad (4.3)$$

Din (4.2) si (4.3) rezulta

$$C_N = 1 + \frac{1}{N} (C_N^0 - 1); \quad (4.4)$$

Rezulta ca C_N este minim daca si numai daca C_N^0 este minim, ori dupa cum am aratat mai inainte acest lucru se intampla atunci si numai atunci cand frunzele lui T se afla pe cel mult doua nivele consecutive. Cum lemei 2 arborele asociat cautarii binare satisface aceasta ipoteza, am demonstrat:

Teorema 6. Cautarea binara este optima in sensul ca minimizeaza numarul mediu de comparatii indiferent de reusita cautarii.

4.2 Arbori binari de cautare

Am demonstrat in sectiunea precedenta ca pentru o valoare data n , arborele de decizie asociat cautarii binare realizeaza numarul minim de comparatii necesare cautarii intr-un tabel prin compararea cheilor. Metodele prezentate in sectiunea precedenta sunt potrivite numai pentru tabele de marime fixa deoarece alocarea secventiala a inregistrarilor face operatiile de insertie si

stergere foarte costisitoare. In schimb, folosirea unei structuri de arbore binar faciliteaza insertia si stergerea nregistrarilor, facand cautarea n tabel e ciente.

Definitie: Un arbore binar de cautare pentru multimea

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

este un arbore binar cu n noduri $f(v_1; v_2; \dots; v_n)$: Aceste noduri sunt etichetate cu elemente ale lui S, adica exista o functie injectiva

$$\text{CONTINUT} : f(v_1; v_2; \dots; v_n) \rightarrow S:$$

Etichetarea pastreaza ordinea, adica n cazul n care v_i este un nod al subarborelui stang apartinand arborelui cu radacina v_k ; atunci

$$\text{CONTINUT}(v_i) < \text{CONTINUT}(v_k)$$

iar n cazul n care v_j este un nod al subarborelui drept apartinand arborelui cu radacina v_k ; atunci

$$\text{CONTINUT}(v_k) < \text{CONTINUT}(v_j):$$

O definitie echivalenta este urmatoarea : o traversare n ordine simetrica a unui arbore binar de cautare pentru multimea S reproduce ordinea pe S.

Prezentam mai jos un program de insertie si stergere a nodurilor intr-un arbore binar de cautare.

```
# include<iostream.h>
# include<stdlib.h>
int cheie;
struct nod {int inf; struct nod *st, *dr;};
/*****/
void inserare(struct nod**rad)
{if(*rad==NULL)
{f*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)
```



```
fif(rad)flistare(rad->st, indent+ 1);
cheie= indent;
while(cheie ) cout<< " ";
cout<< rad->inf;
listare(rad->dr,indent+ 1);gg
/*****/
void stergere(struct nod**rad)
fstruct nod*p,*q;
if(*rad== NULL)
fcout<< "Arborele nu contine " << cheie<< endl;return;g
if(cheie< (*rad)->inf) stergere(&(*rad)->st);
if(cheie> (*rad)->inf) stergere(&(*rad)->dr);
if(cheie== (*rad)->inf)
fif((*rad)->dr== NULL)
fq= *rad;*rad=q->st; delete q;g
else
if((*rad)->st== NULL)
fq= *rad;*rad=q->dr; delete q;g
else
ffor(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;ggg
/*****/
void main()
frad= new nod;
cout<< "Valoarea radacinii este: ";cin>> rad->inf;
rad->st= rad->dr= NULL;
dof
cout<< "Operatia:Listare(1)/Inserare(2)/Stergere(3)/Iesire(0) ";
cout<< endl;
cin>> cheie;if(!cheie) return;
switch(cheie)
fcase 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;g
while(rad);
cout<< "Ati sters radacina " << endl;g
```


Dupa cum se observa din program, ideea insertiei n arborele binar de cautare este urmatoarea: daca arborele este vid, se creaza un arbore avand drept unic nod si radacina nodul inserat; altfel se compara cheia nodului inserat cu cheia radacinii. Daca avem cheia nodului inserat mai mica decat cheia radacinii, se trece la subarborele stang si se apeleaza recursiv procedura de inserare, altfel se trece la subarborele drept si se apeleaza recursiv procedura.

Procedura prezentata n program de stergere a unui nod din arborele binar de cautare este de asemenea recursiva. In cazul n care cheia nodului ce urmeaza a fi sters este mai mica decat cheia radacinii, se trece la subarborele stang si se apeleaza recursiv procedura de stergere; altfel se trece la subarborele drept si se apeleaza recursiv procedura de stergere. In cazul n care nodul ce urmeaza a fi sters este chiar radacina vom avea mai multe posibilitati: a) subarborele drept este vid: se sterge radacina iar nodul stang al radacinii devine noua radacina; b) subarborele stang este vid: se sterge radacina iar nodul drept al radacinii devine noua radacina; c) radacina are ambii copii; n acest caz se sterge cel mai din dreapta descendent al nodului stang al radacinii iar informatia (cheia) acestuia nlocuieste informatia (cheia) radacinii, nodul stang al nodului eliminat devine totodata nodul drept al noului nod eliminat. Daca nodul stang al radacinii nu are niciun drept, atunci el este cel eliminat, informatia lui nlocuieste informatia radacinii iar nodul stang devine nodul stang al radacinii (figura 4.4).

Algoritmul de cautare, dupa o anumita cheie, intr-un arbore de cautare este n esenta urmatorul: comparam cheia nregistrarii cautate cu cheia radacinii. Daca cele doua chei coincid cautarea este reusita. Daca cheia nregistrarii este mai mica decat cheia radacinii continuam cautarea n subarborele stang iar daca este mai mare n subarborele drept.

De foarte multe ori este util sa prezentam arborii binari de cautare ca n figura 4.5.

Aici arborele binar de cautare este prezentat ca un arbore complet n care informatiile (cheile) sunt stocate n cele N noduri interne iar informatia ecarui nod extern (frunza) este intervalul deschis dintre doua chei consecutive, astfel incat, daca $x_i; x_{i+1}$ sunt doua chei consecutive si vrem sa cautam nodul ce contine cheia a cu $a \in [x_i; x_{i+1})$ sa fim condusi aplicand algoritmul de cautare (intr-un arbore binar de cautare) la frunza etichetata cu $(x_i; x_{i+1})$:

Vom arata ca naltimea medie a unui arbore binar de cautare este de ordinul $O(\ln N)$ (N este numarul nodurilor interne) si cautarea necesita n medie circa $2 \ln N - 1; 386 \log_2 N$ comparatii n cazul n care cheile sunt

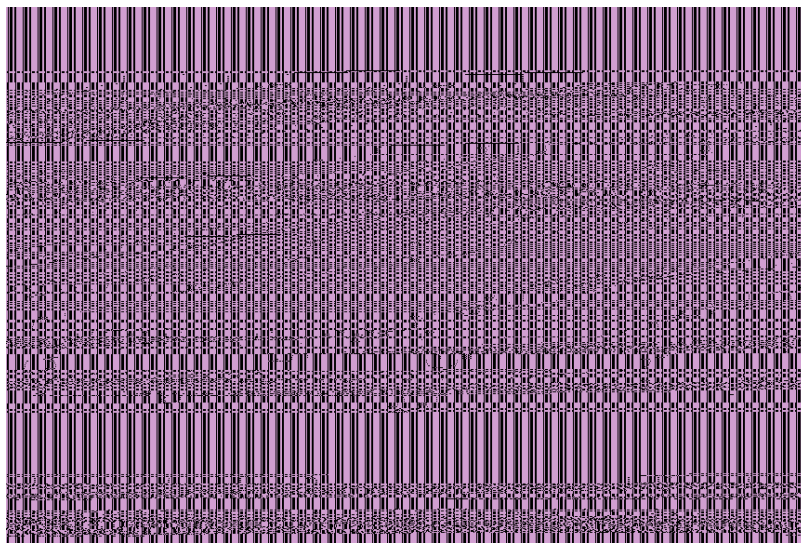


Figura 4.4: Stergerea radacinii unui arbore binar de cautare

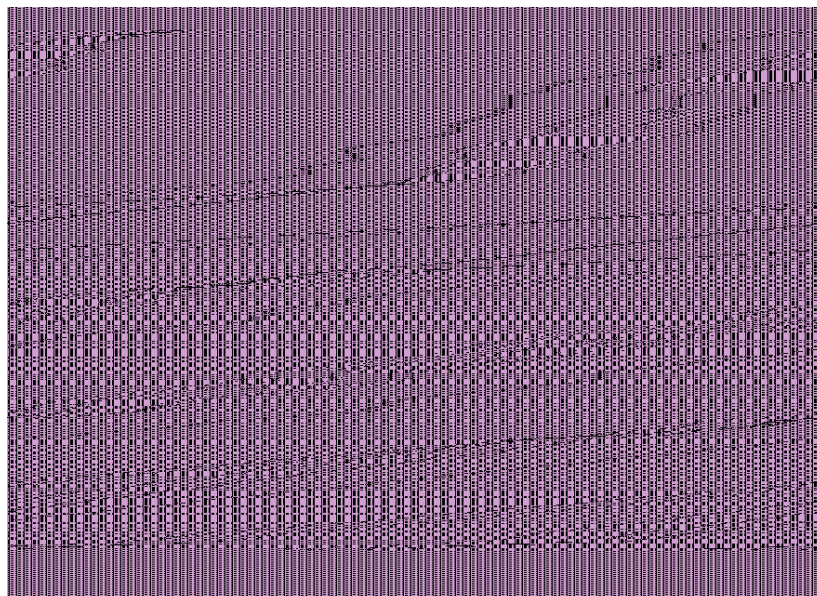


Figura 4.5: Arbore binar de cautare

inserate în arbore în mod aleator. Să presupunem că cele $N!$ ordonări posibile ale celor N chei corespund la $N!$ modalități de inserție. Numărul de comparații necesare pentru a găsi o cheie este exact cu 1 mai mare decât numărul de comparații efectuate atunci când cheia a fost inserată în arbore. Notând cu C_N numărul mediu de comparații pentru o căutare reușită și cu C_N^0 numărul mediu de comparații pentru o căutare nereușită avem

$$C_N = 1 + \frac{C_0^0 + C_1^0 + \dots + C_{N-1}^0}{N};$$

pentru că înainte de reușita căutării vom avea căutări nereușite în mulțimi de $0; 1; \dots; n-2$ sau $n-1$ elemente. Ținând cont și de relația

$$C_N = 1 + \frac{1}{N} C_N^0 - 1;$$

deducem

$$(N+1) C_N^0 = 2N + C_0^0 + C_1^0 + \dots + C_{N-1}^0;$$

Scazând din această ecuație următoarea ecuație

$$N C_{N-1}^0 = 2(N-1) + C_0^0 + C_1^0 + \dots + C_{N-2}^0$$

obținem

$$(N+1) C_N^0 - N C_{N-1}^0 = 2 + C_{N-1}^0 \Rightarrow C_N^0 = C_{N-1}^0 + \frac{2}{N+1};$$

Cum $C_0^0 = 0$ deducem că

$$C_N^0 = 2H_{N+1} - 2;$$

de unde

$$C_N = 2 - 1 + \frac{1}{N} H_{N+1} - 3 + \frac{2}{N} = 2 \ln N;$$

4.3 Arbori de căutare ponderați (optimali)

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

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

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

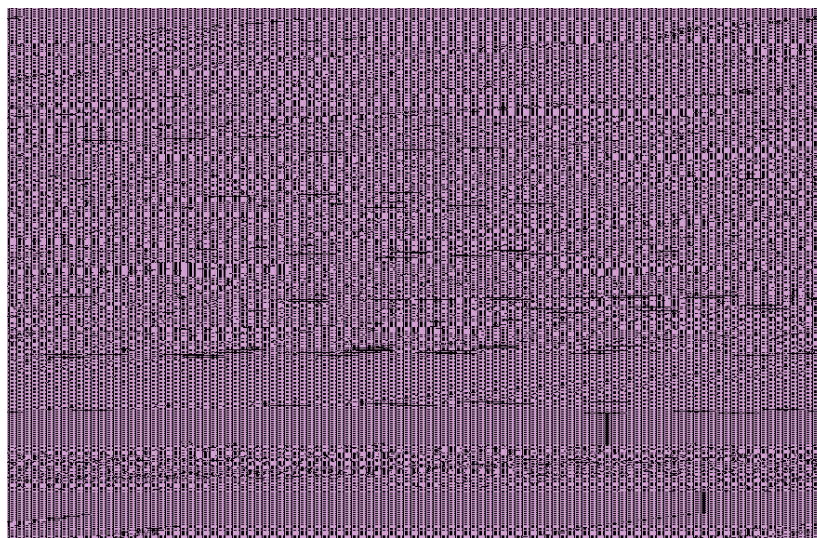


Figura 4.6: Arbori posibili de cautare și numărul mediu de comparații pentru o cautare reușită

Obținem astfel 5 expresii algebrice pentru numărul mediu de comparații într-o cautare. Când N este mare, este foarte costisitor să construim toți arborii de cautare pentru a vedea care din ei este cel optim. Vom pune de aceea în evidență un algoritm de găsire al acestuia.

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

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

$(2N + 1)$ - tuplul $(q_0; p_1; q_1; \dots; p_N; q_N)$ se numește **distributia probabilitatilor (ponderilor) de cautare (acces)**. Fie T un arbore de cautare pentru S , e_i^T

adancimea (nivelul) nodului intern i (al i -lea nod intern n ordine simetrica) si e_j^T adancimea frunzei j (al $(j + 1)$ -lea nod extern sau frunza $(K_j; K_{j+1})$).

Sa consideram o cautare a cheii a . Daca $a = K_i$, vom compara a cu $i + 1$ elemente din arbore; daca $K_j < a < K_{j+1}$, atunci vom compara a cu j elemente din arbore. Asadar

$$POND(T) = \sum_{i=1}^n p_i (i + 1) + \sum_{j=0}^n q_j j;$$

este numarul mediu de comparatii pentru o cautare. $POND(T)$ este lungimea ponderata a drumurilor arborelui T (sau costul lui T relativ la o distributie data a probabilitatilor de cautare).

Vom considera $POND(T)$ drept indicator de baza pentru eficienta operatiei de cautare (acces) deoarece numarul asteptat de comparatii intr-o cautare va fi proportional cu $POND(T)$: De exemplu, n cazul arborelui din figura 4.6 b) (unde n loc de $p; q; r$ punem $p_1; p_2; p_3$) avem

$$p_1 = 1; \quad p_2 = 2; \quad p_3 = 0; \quad q_0 = 2; \quad q_1 = 3; \quad q_2 = 3; \quad q_3 = 1$$

si deci

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

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

Definitie. Arborele de cautare T peste multimea ordonata S cu distributia ponderilor de cautare $(q_0; p_1; q_1; \dots; p_N; q_N)$, este **optimal** daca $POND(T)$ (costul arborelui sau lungimea ponderata a drumurilor arborelui) este minim n raport cu costurile celorlalti arbori de cautare peste S .

Vom prezenta mai departe un algoritm de construire a unui arbore de cautare optimal. Fie un arbore de cautare peste S avand nodurile interne etichetate cu $1; 2; \dots; N$ (corespunzator cheilor $K_1; \dots; K_N$) si frunzele etichetate cu $0; 1; \dots; N$ (corespunzand lui $(; K_1); (K_1; K_2); \dots; (K_{N-1}; K_N); (K_N;)$). Un subarbore al acestuia ar putea avea nodurile interne $i + 1; \dots; j$ si frunzele $i; \dots; j$ pentru $0 \leq i; j \leq n$; $i < j$: Acest subarbore este la randul sau arbore de cautare pentru multimea cheilor $fK_{i+1} < \dots < K_jg$. Fie k eticheta radacinii subarborelui.

Fie costul acestui subarbore $POND(i; j)$ si greutatea sa:

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

de unde rezulta imediat ca

$$\text{GREUT}(i;j) = \text{GREUT}(i;j-1) + p_j + q_j; \text{GREUT}(i;i) = q_i:$$

Avem relatia

$$\text{POND}(i;j) = \text{GREUT}(i;j) + \text{POND}(i;k-1) + \text{POND}(k;j):$$

Intr-adevar, subarborele stang al radacini k are frunzele $i; i+1; \dots; k-1$; iar subarborele drept are frunzele $k; k+1; \dots; j$, si nivelul ecarui nod din subarborele drept sau stang este cu 1 mai mic decat nivelul aceluiasi nod in arborele de radacina k . Fie $C(i;j) = \min \text{POND}(i;j)$ costul unui subarbore optimal cu ponderile $fp_{i+1}; \dots; p_j; q_i; \dots; q_j$: Rezulta atunci pentru $i < j$:

$$C(i;j) = \text{GREUT}(i;j) + \min_{i < k < j} (C(i;k-1) + C(k;j));$$

$$C(i;i) = 0:$$

Pentru $i = j + 1$ rezulta imediat ca

$$C(i;i+1) = \text{GREUT}(i;i+1); k = i+1:$$

Plecand de la aceste relatii prezentam mai jos un program, scris in limbajul C de construire a unui arbore optimal. Campul informatiei ecarui nod contine un caracter (litera) iar acestea se considera ordonate dupa ordinea citirii.

```

/*****
#include<stdio.h>
#include<stdlib.h>
#include <io.h>
# define N 25
struct nod {char ch; struct nod *st,*dr;g *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 informatiilor a ate 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()
fint x,min,i,j,k,h,m;
for(i= 0;i<= nr;i++ )
fgreut[i][i]= q[i];
for(j= i+ 1;j<= nr;j++ )
greut[i][j]= greut[i][j-1]+ p[j]+ q[j];g
for(i= 0;i<= nr;i++ ) c[i][i]= q[i];
for(i= 0;i< nr;i++ )
fj= i+ 1;
c[i][j]= greut[i][j];
rad[i][j]= j;g
for(h= 2;h<= nr;h++ )
for(i= 0;i<= nr-h;i++ )
fj= 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++ )
fx= c[i][k-1]+ c[k][j];
if(x< min)m= k;min= x;gg
c[i][j]= min+ greut[i][j];
rad[i][j]= m;gg
/****Functia de generare a arborelui optimal*****/
struct nod *arbore(int i, int j)
fstruct nod *s;
if(i== j) s= NULL;
elses= new nod;
s->st= arbore(i,rad[i][j]-1);
s->ch= chei[rad[i][j]];
s->dr= arbore(rad[i][j],j);g
return s;g
/***** Functia de listare indentata a nodurilor arborelui*****/
void listare(struct nod*r, int nivel)
fint i;
if(r)f listare(r-> dr,nivel+ 1);i= nivel;
while(i ) printf( );
printf( %cn\n",r->ch);
listare(r-> st, nivel+ 1);gg
n Functiaprincipala

```

```

void main() ff=fopen( arboptim.dat , r );
fscanf(f, %dnn0; &nr);
if(nr > 0)
ffscanf(f, %dnn0; &q[0]);
for(i= 1; i <= nr; i++ )
fscanf(f, %c %dnn%dnn0; &chei[i]; &p[i]; &q[i]);
calcul();
printf( Lungimea medie a unei cautari: %fnn0;
( oat)c[0][nr]/greut[0][nr]);
struct nod*radacina= arbore(0,nr);
listare(radacina,0);gg
/*****

```

Fisierul **arboptim.dat** contine pe prima linie numarul de noduri interne ale arborelui, pe a doua linie valoarea ponderii q_0 iar pe celelalte linii cheile K_i cu ponderile p_i si q_i : Un exemplu de astfel de sir este urmatorul:

```

/*****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 echilibrati

Insertia de noi noduri intr-un arbore binar de cautare poate conduce la arbori dezechilibrati in care ntre naltimea subarborelui drept al unui nod si naltimea subarborelui stang sa e o mare diferenta. Intr-un astfel de arbore actiunea de cautare va consuma mai mult timp.

O solutie la problema mentinerii unui bun arbore de cautare a fost descoperita de G. M. Adelson-Velskii si E. M. Landis in 1962 care au pus in evidenta asa numiti arbori echilibrati (sau arbori AVL).

Definiție. Un arbore binar este **echilibrat (AVL)** dacă înălțimea sub-arborelui stâng al oricărui nod nu diferă mai mult decât 1 de înălțimea sub-arborelui său drept.

Definiție. Diferența dintre înălțimea sub-arborelui drept și înălțimea sub-arborelui stâng poartă numele de **factor de echilibru** al unui nod.

Asadar, 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 echilibrați este clasa de **arbori Fibonacci** de care ne vom ocupa în continuare.

Să considerăm mai întâi șirul $(F_n)_{n=1}$ de numere Fibonacci, definite prin următoarea formulă de recurență:

$$\begin{aligned} F_1 &= F_2 = 1; \\ F_{n+2} &= F_{n+1} + F_n; \quad n \geq 1: \end{aligned} \quad (4.5)$$

Primii termeni din șirul lui Fibonacci sunt 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89; ... :

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

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

cu soluția

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

Soluția generală va

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

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

$$\begin{aligned} A \frac{1 + \sqrt{5}}{2} + B \frac{1 - \sqrt{5}}{2} &= 1; \\ A \frac{3 + \sqrt{5}}{2} + B \frac{3 - \sqrt{5}}{2} &= 1; \end{aligned}$$

cu soluția

$$A = \frac{\sqrt{5}}{5}; \quad B = -\frac{\sqrt{5}}{5};$$

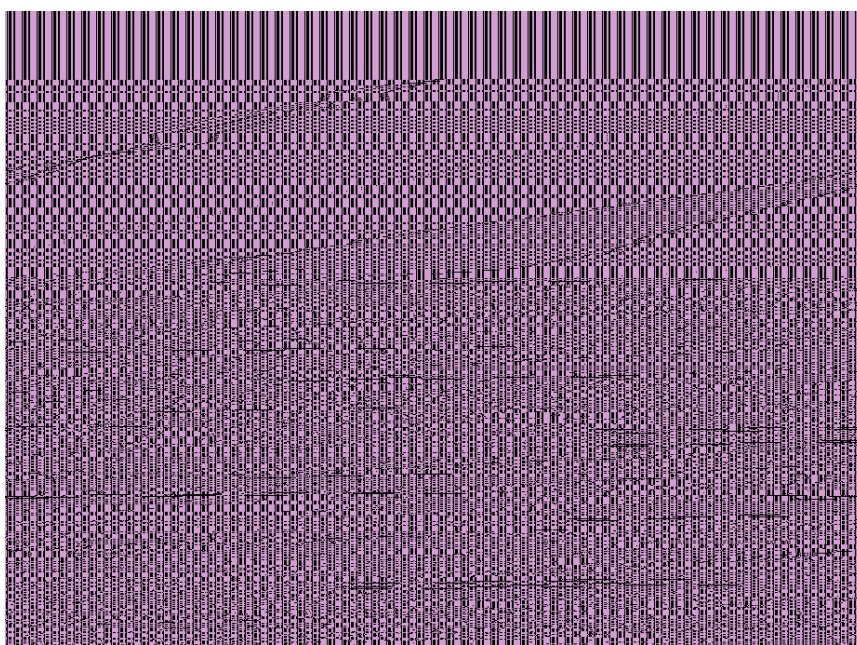


Figura 4.7: Arbori Fibonacci

Obtinem astfel formula lui Binet pentru numerele Fibonacci:

$$F_n = \frac{P_5 - 1}{5} \left(\frac{1 + P_5}{2} \right)^n - \frac{P_5 - 1}{5} \left(\frac{1 - P_5}{2} \right)^n :$$

: Arborii binari Fibonacci, notati cu FT_k ; $k = 0; 1; 2; \dots$ sunt definiti prin recurenta dupa cum urmeaza:

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

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

Vom pune mai departe in evidenta cateva proprietati ale arborilor Fibonacci.

Lema. Pentru orice $k \geq 1$, arborele Fibonacci FT_k este un arbore echilibrat avand inaltimea $h(FT_k) = k - 1$, F_{k+1} frunze si $F_{k+1} - 1$ noduri interne.

Demonstratie. Pentru $k = 1$ si $k = 2$ proprietatea este veri cata. Sa presupunem ca proprietatea este adevarata pentru toti arborii Fibonacci FT_{k_0} cu $k_0 < k$. Fie FT_k un arbore Fibonacci de ordin $k > 3$. Din definitia recursiva obtinem ca $h(FT_k) = h(FT_{k-1}) + 1 = k - 1$, numarul de frunze al lui FT_k este egal cu $F_k + F_{k-1} = F_{k+1}$ iar numarul de noduri interne este $1 + (F_k - 1) + (F_{k-1} - 1) = F_{k+1} - 1$: Din ipoteza de inductie, proprietatea de echilibrare este satisfacuta pentru toate nodurile cu exceptia radacinii. In ceea ce priveste radacina, subarborele stang are inaltimea $k - 2$ iar subarborele drept are inaltimea $k - 3$, deci FT_k este echilibrat.

4.4.2 Proprietati ale arborilor echilibrati

Arborii echilibrati reprezinta o treapta intermediara ntre clasa arborilor optimali (cu frunzele asezate pe doua nivele adiacente) si clasa arborilor binari arbitrari. De aceea este interesant sa ne intrebam cat de mare este diferenta dintre un arbore optimal si un arbore echilibrat; prezentam in acest context urmatoarea teorema:

Teorema. Inaltimea unui arbore echilibrat T cu N noduri interne se afla intotdeauna ntre $\log_2(N + 1)$ si $1.4404 \log_2(N + 2) - 0.328$:

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

$$N \leq 2^{h(T)} - 1 \Rightarrow h(T) \leq \log_2(N + 1):$$

Pentru a gasi limitarea superioara a lui $h(T)$, ne vom pune problema a arii numarului minim de noduri interne continute intr-un arbore echilibrat de naltime h . Fie deci T_h arborele echilibrat de naltima h cu cel mai mic numar de noduri posibil; unul din subarborii radacinii, de exemplu cel stang va avea naltimea $h - 1$ iar celalalt subarbore va avea naltimea $h - 1$ sau $h - 2$: Cum T_h are numarul minim de noduri, va trebui sa consideram ca subarboarele stang al radacinii are naltimea $h - 1$ iar subarboarele drept are naltimea $h - 2$. Putem asadar considera ca subarboarele stang al radacinii este T_{h-1} iar subarboarele drept este T_{h-2} . In virtutea acestei consideratii, se demonstreaza prin inductie ca arborele Fibonacci F_{h+1} este arborele T_h cautat, in sensul ca intre toti arborii echilibrati de naltime impusa h ; acesta are cel mai mic numar de noduri. Conform lemei precedente avem

$$N = F_{h+2} - 1 = \frac{P_5}{5} - \frac{1 + P_5^{h+2}}{2} = \frac{P_5}{5} - \frac{1 + P_5^{h+2}}{2} - 1:$$

Cum

$$\frac{P_5}{5} - \frac{1 + P_5^{h+2}}{2} > -1;$$

rezulta ca

$$\log_2(N + 2) > (h + 2) \log_2 \left(\frac{1 + P_5^{h+2}}{2} \right) = \frac{1}{2} \log_2 5 \cdot (h + 2)$$

$$\Rightarrow h < 1.4404 \log_2(N + 2) - 0.328:$$

Din consideratiile facute pe parcursul demonstratiei teoremei, rezulta urmatorul

Corolar. Intre arborii echilibrati cu un numar dat de noduri, arborii Fibonacci au naltimea maxima, deci sunt cei mai putin performanti.

4.5 Insertia unui nod intr-un arbore echilibrat

Inserarea unui nou nod se efectueaza cu algoritmul cunoscut de inserare a nodurilor intr-un arbore binar de cautare. Dupa inserare nsă, va trebui să re-echilibrăm arborele dacă vom ajunge într-una din următoarele situații:

- subarborelui stang al unui nod cu factorul de echilibru $\neq 1$ și crește înălțimea;
- subarborelui drept al unui nod cu factorul de echilibru $\neq 1$ și crește înălțimea.

Ambele cazuri sunt tratate apelând la rotații (pe care le vom descrie în cele ce urmează). Rotațiile implică doar modificări ale legăturilor în cadrul arborelui, nu și operații de cautare, de aceea timpul lor de execuție este de ordinul $O(1)$:

4.5.1 Rotații în arbori echilibrați

Rotațiile sunt operații de schimbare între ele a unor noduri aflate în relația de tată-fiu (și de refacere a unor legături) astfel încât să se păstreze structura de arbore de cautare. Astfel, printr-o rotație simplă a unui arbore la stanga, fiul drept al rădăcinii inițiale devine noua rădăcină iar rădăcina inițială devine fiul stang al noii rădăcini. Printr-o rotație simplă a unui arbore la dreapta, fiul stang al rădăcinii inițiale devine noua rădăcină iar rădăcina inițială devine fiul drept al noii rădăcini. O rotație dublă la dreapta afectează două nivele: fiul drept al fiului stang al rădăcinii inițiale devine noua rădăcină, rădăcina inițială devine fiul drept al noii rădăcini iar fiul stang al rădăcinii inițiale devine fiul stang al noii rădăcini. O rotație dublă la stanga afectează de asemenea două nivele: fiul stang al fiului drept al rădăcinii inițiale devine noua rădăcină, rădăcina inițială devine fiul stang al noii rădăcini iar fiul drept al rădăcinii inițiale devine fiul drept al noii rădăcini.

Vom studia cazurile de dezechilibru ce pot apărea și vom efectua re-echilibrarea prin rotații.

Cazul 1. Crește înălțimea subarborelui stang al nodului a care are factorul de echilibru inițial $\neq 1$.

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

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

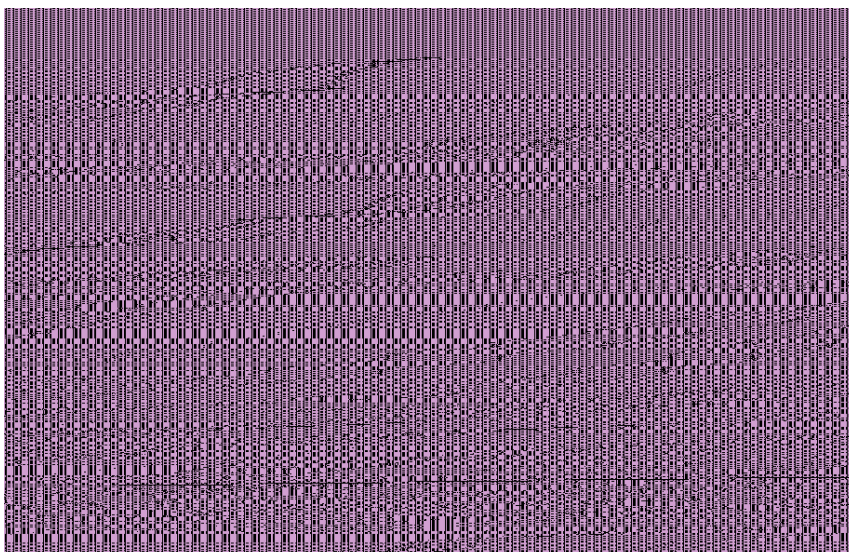


Figura 4.8: Rotatie simpla la dreapta pentru re-echilibrare

b.1) Factorul de echilibru al ului drept c al lui b este 1 (gura 4.9). Aceasta nseamna ca noul element a fost inserat n subarborele C . Pentru re-echilibrare vom avea nevoie de o rotatie dubla la dreapta a nodurilor $b < c < a$:

b.2) Factorul de echilibru al ului drept c al lui b este -1: Se trateaza ca si cazul b.1) (gura 4.10).

b.3) Factorul de echilibru al ului drept c al lui b este 0. Dezechilibrarea este imposibila.

c) Factorul de echilibru al ului stang b al lui a este 0. Dezechilibrarea este imposibila.

Cazul II. Creste naltimea subarborelui drept al nodului a care are factorul de echilibru initial 1. Se trateaza asemanator cu cazul I).

a) Factorul de echilibru al ului stang b al lui a este 1 (gura 4.11). Aceasta nseamna ca noul element a fost inserat n subarborele C . Re-echilibrarea necesita o rotatie simpla la stanga a perechii tata- u ($a < b$).

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

b.1) Factorul de echilibru al ului stang c al lui b este -1(gura 4.12). Aceasta nseamna ca noul element a fost inserat n subarborele B . Pentru re-echilibrare vom avea nevoie de o rotatie dubla la stanga a nodurilor $b > c > a$:

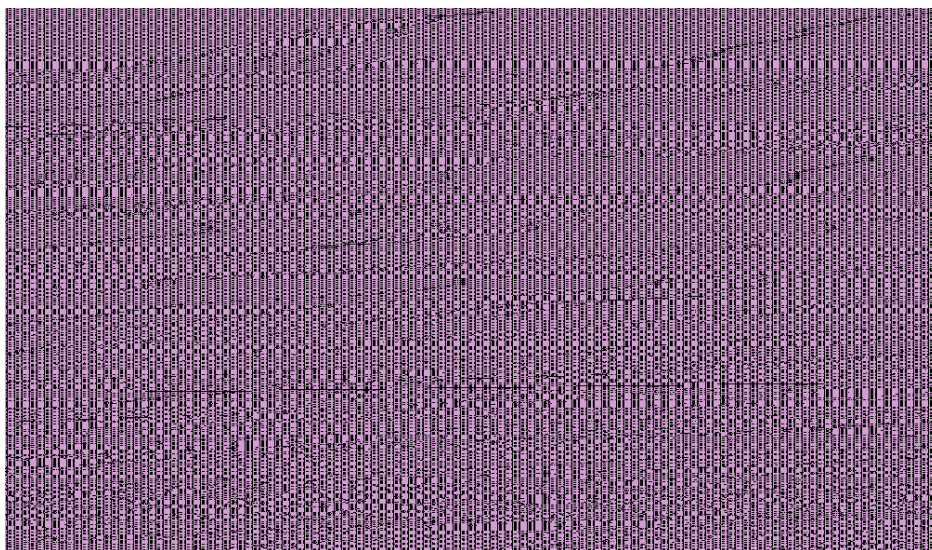


Figura 4.9: Rotatie dubla la dreapta pentru re-echilibrare

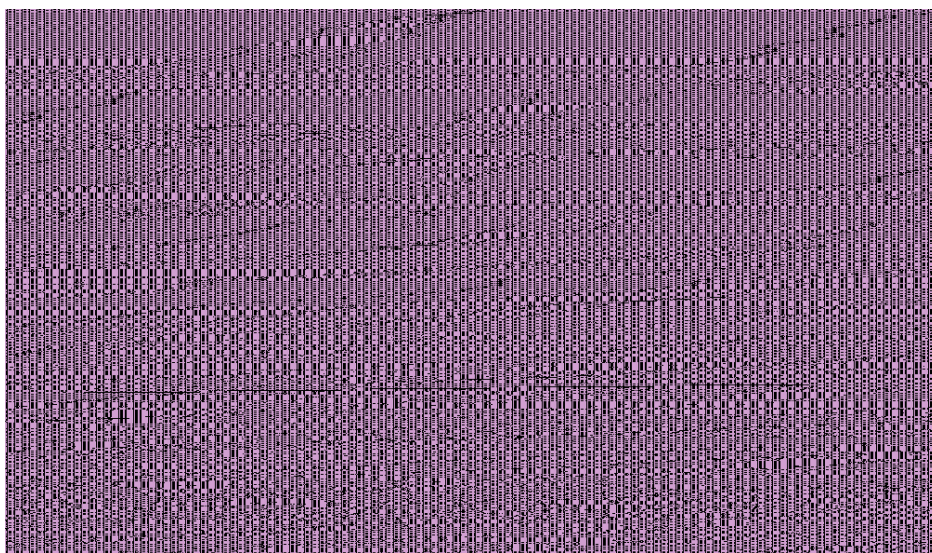


Figura 4.10: Rotatie dubla la dreapta pentru re-echilibrare

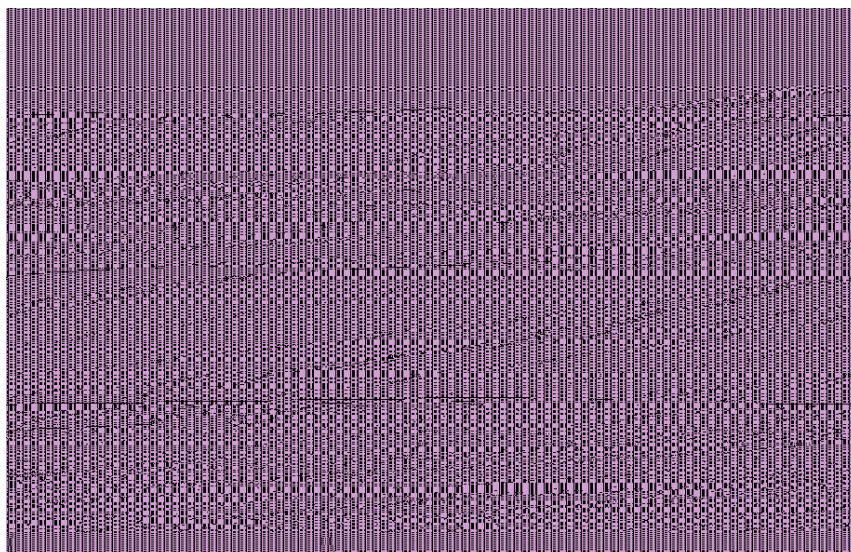


Figura 4.11: Rotatie simpla la stanga pentru re-echilibrare

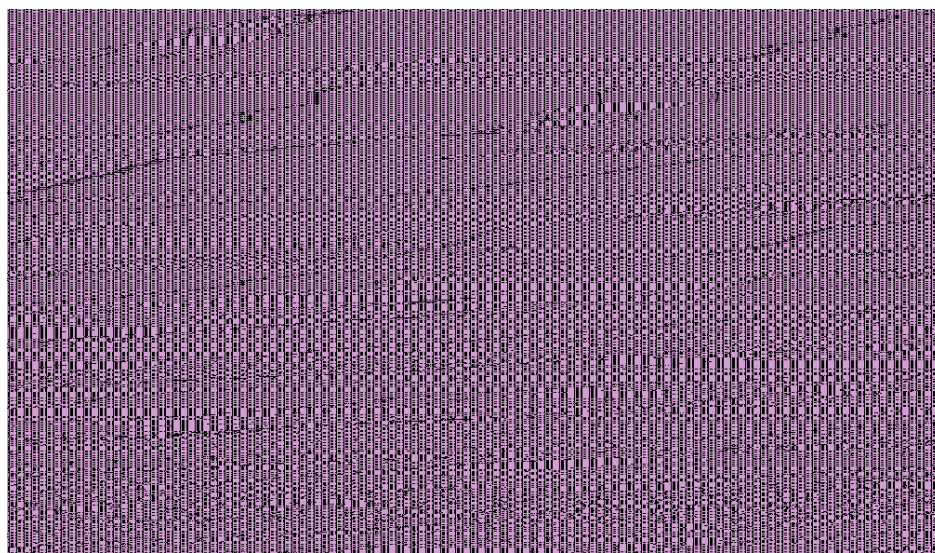


Figura 4.12: Rotatie dubla la stanga pentru re-echilibrare

b.2) Factorul de echilibru al u ului drept c al lui b este 1: Se trateaza ca si cazul b.1) (figura 4.13).

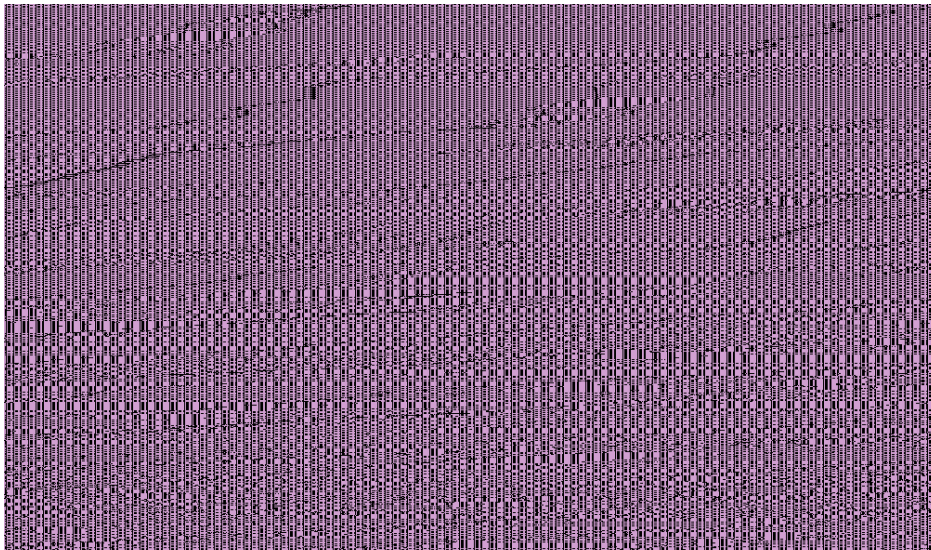


Figura 4.13: Rotatie dubla la stanga pentru re-echilibrare

b.3) Factorul de echilibru al u ului drept c al lui b este 0. Dezechilibrarea este imposibila.

c) Factorul de echilibru al u ului stang b al lui a este 0. Dezechilibrarea este imposibila.

4.5.2 Exemple

In arborele din figura 4.14 ne propunem sa inseram elementul 58. Suntem in cazul I.a). Subarborii cu radacinile 70 si 80 devin dezechilibrati. Pentru echilibrare se roteste perechea (60; 70) la dreapta, obtinandu-se arborele din figura 4.15.

In arborele din figura 4.16 ne propunem sa inseram elementul 68. Suntem in cazul I.b.1). Pentru echilibrare se roteste la stanga perechea (60; 65) si apoi se roteste la dreapta perechea (70; 65). Se obtine noul arborele din figura 4.17.

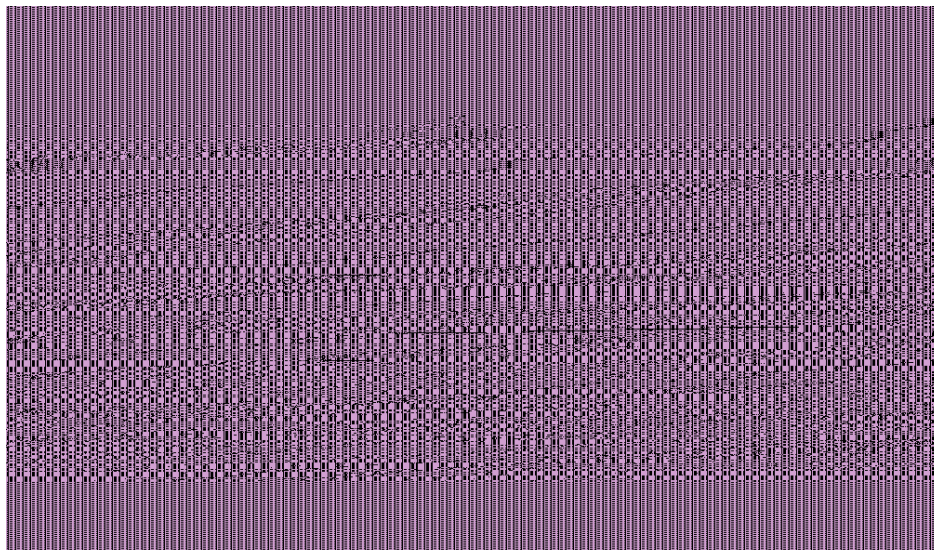


Figura 4.14: Exemplu de insertie într-un arbore echilibrat

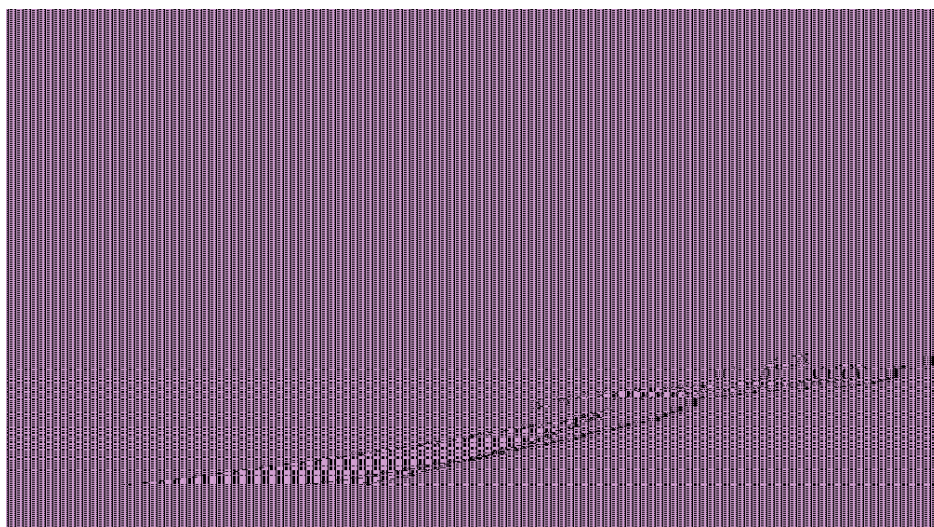


Figura 4.15: Exemplu de insertie într-un arbore echilibrat

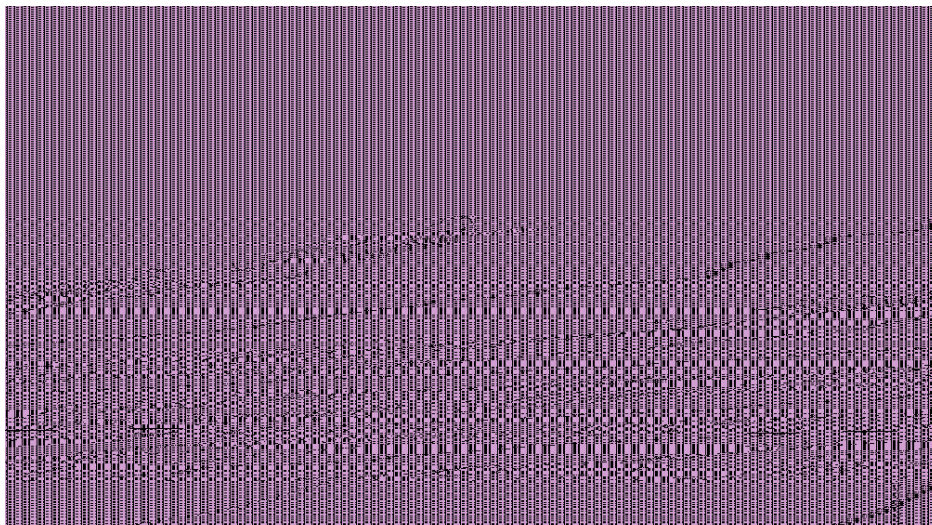


Figura 4.16: Exemplu de insertie ntr-un arbore echilibrat

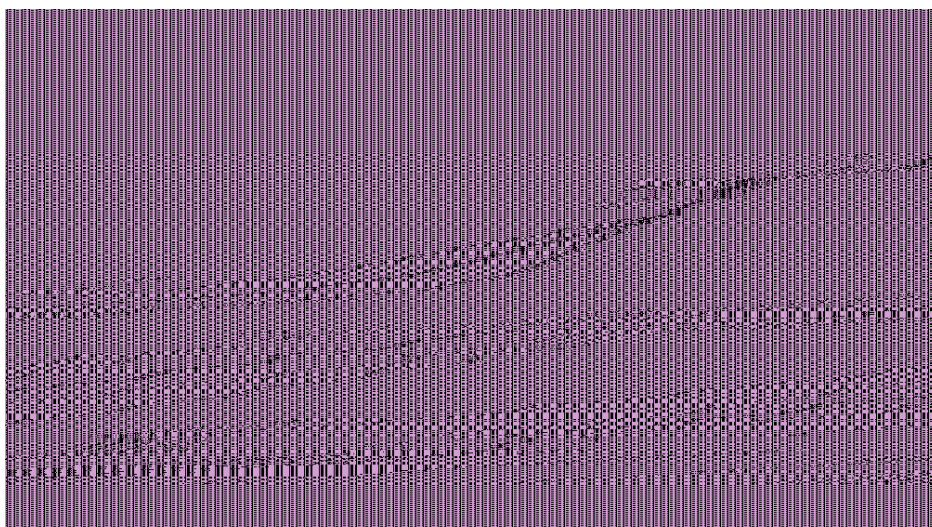


Figura 4.17: Exemplu de insertie ntr-un arbore echilibrat

4.5.3 Algoritmul de insertie n arbori echilibrati

Pentru a insera un element x ntr-un arbore binar de cautare echilibrat se parcurg urmatoarele etape:

- se cauta pozitia n care noul element trebuie inserat (ca n orice arbore binar de cautare);
- se insereaza elementul x (ca n orice arbore binar de cautare);
- se reactualizeaza factorii de echilibru ai ascendentilor lui x pana la radacina sau pana se gaseste cel mai apropiat ascendent p al lui x , daca exista, astfel ncat subarborele T cu radacina p sa e dezechilibrat;
- daca p exista, se re-echilibreaza subarborele T cu ajutorul unei rotatii (simple sau duble).

4.6 Stergerea unui nod al unui arbore echilibrat

Stergerea este similara insertiei: stergem nodul asa cum se procedeaza n cazul unui arbore binar de cautare si apoi re-echilibram arborele rezultat. Deosebirea este ca numarul de rotatii necesar poate tot atat de mare cat nivelul (adancimea) nodului ce urmeaza a sters. De exemplu sa stergem elementul x din gura 4.18.a).

In urma stergerii se ajunge la arborele ne-echilibrat din gura 4.18.b). Subarborele cu radacina n y este ne-echilibrat. Pentru a-l echilibra este necesara o rotatie simpla la dreapta a perechii $(y; i)$ obtinandu-se arborele din gura 4.19.d); si acest arbore este ne-echilibrat, radacina a avand factorul de echilibru -2 . O rotatie dubla la dreapta a lui $e; b$ si a , re-echilibreaza arborele ajungandu-se la arborele din gura 4.20.e).

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

Fie data nregistrarea avand cheia x . Pentru a sterge dintr-un arbore de cautare echilibrat nodul cu cheia x parcurgem urmatoarele etape

- se localizeaza nodul r avand cheia x ;
- daca r nu exista, algoritmul se termina;
- altfel, se sterge nodul r , utilizand algoritmul de stergere ntr-un arbore binar de cautare;.

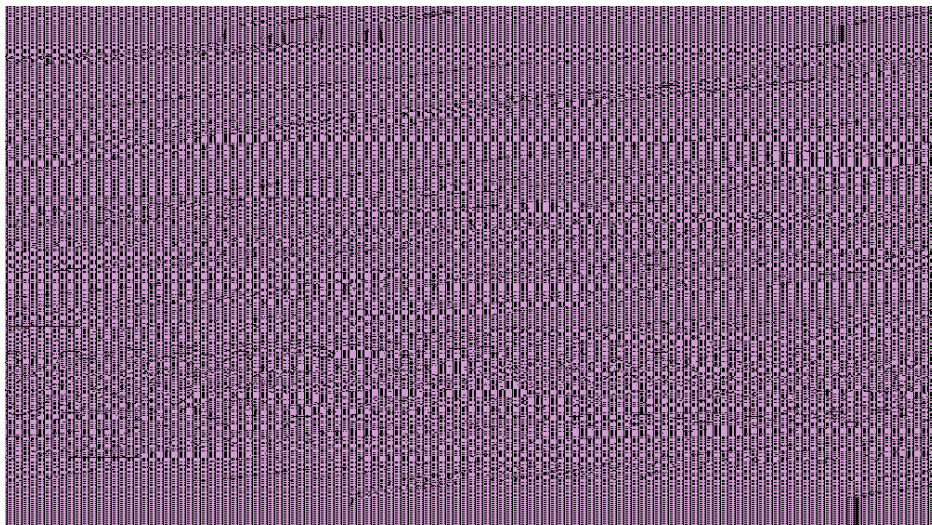


Figura 4.18: Exemplu de stergere a unui nod dintr-un arbore echilibrat

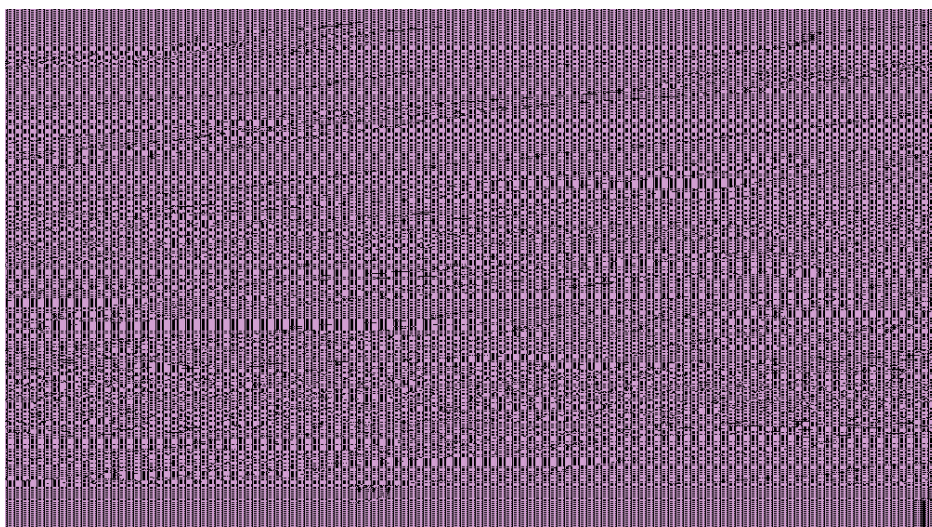


Figura 4.19: Exemplu de stergere a unui nod dintr-un arbore echilibrat

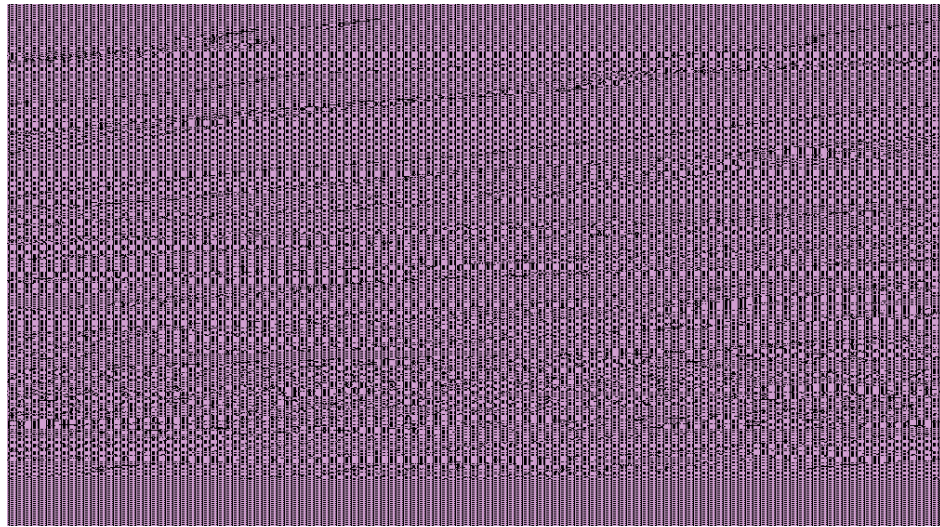


Figura 4.20: Exemplu de stergere a unui nod dintr-un arbore echilibrat

- e q nodul extern eliminat în urma aplicării algoritmului de stergere și p tatal sau. Se re-echilibrează (daca este cazul) arborele prin rotații implicând nodul p și eventual ascendentii acestuia.

Vom arata ca numărul de rotații necesar stingerii unui nod poate ajunge până la numărul ce indică nivelul nodului în arbore. Observăm mai întâi ca o rotație reduce cu 1 înălțimea arborelui caruia i este aplicată. Fie a cel mai apropiat predecesor al elementului sters x, astfel ca subarborele T_a cu rădăcina n a să fie neechilibrat. Dacă înainte de rotație $h(T_a) = k$, atunci după rotație $h(T_a) = k - 1$.

Fie b tatal lui a (daca a nu este rădăcina arborelui). Avem următoarele situații favorabile:

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

Discuțiile se ivesc atunci când:

- factorul de echilibru al lui b este -1 și T_a este subarborele drept al lui b;
 - factorul de echilibru al lui b este 1 și T_a este subarborele stâng al lui b
- În ambele cazuri va trebui să re-echilibram arborele T cu rădăcina n b.

4.6. STERGerea UNUI NOD AL UNUI ARBORE ECHILIBRAT 101

Sa observam ca nainte de echilibrare $h(T) = k + 1$ iar dupa re-echilibrare $h(T) = k$. Astfel, subarboarele avand drept radacina pe tatal lui b poate deveni ne-echilibrat si tot asa pana cand ajungem la radacina arborelui initial (n cel mai rau caz).

In continuare prezentam un program, scris n C de inserare si stergere de noduri ntr-un arbore binar de cautare echilibrat.

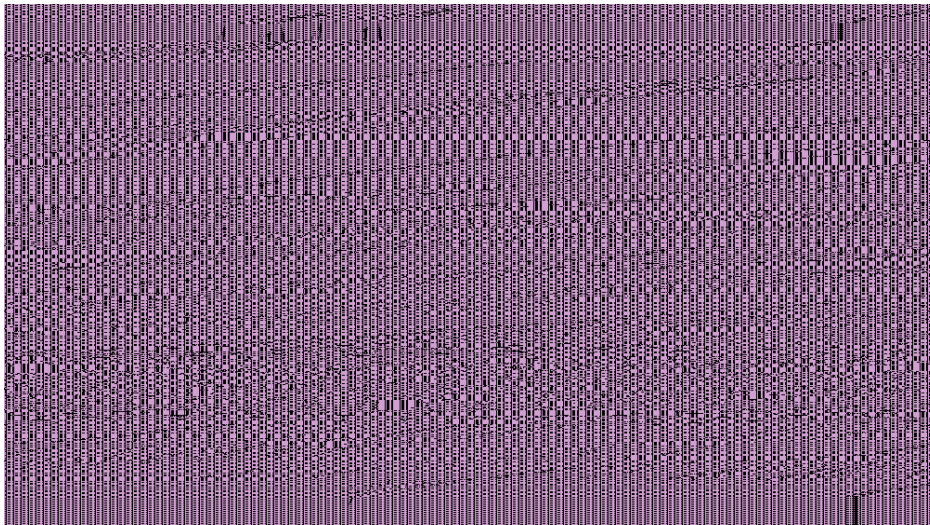


Figura 4.21: Exemplu de stergere a unui nod dintr-un arbore echilibrat

```
# include<stdio.h>
# include<malloc.h>
# de ne F 0
# de ne T 1
struct Nod{char Info;int FactEch;struct Nod *st;struct Nod *dr;g;
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)
```

```

fstruct Nod *Nod1;
struct Nod *Nod2;
if(!tata)
ftata = (struct Nod *) malloc(sizeof(struct Nod));
tata->Info = Info;
tata->st = NULL;
tata->dr = NULL;
tata->FactEch = 0;
*H = T;
return (tata);g
if(Info < tata->Info)
ftata->st = Inserare(Info, tata->st, H);
if(*H)
  /* Creste inaltimea subarborelui stang */
  f
  switch(tata->FactEch)
  f
    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)
      f//Cazul din gura 4.8
      printf( Rotatie simpla la dreapta nn0);
      tata->st= Nod1->dr;
      Nod1->dr = tata;
      tata->FactEch = 0;
      tata = Nod1;
      tata->FactEch = 0;g
    else
      /*cazul Nod1->FactEch == 0 nu este posibil pentru ca
      am avut *H=F; ramane Nod1->FactEch == 1 ca in
      gurile 4.9 si 4.10 */

```



```

f
printf( Rotatie dubla la dreapta nn0);
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;g
tata->FactEch = 0;
*H = F;ggg
if(Info > tata->Info)
f
tata->dr = Inserare(Info, tata->dr, H);
if(*H)
/* Subarborele drept devine mai inalt */
f
switch(tata->FactEch)
f
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)
f/*Cazul din gura 4.11 */
printf( Rotatie simpla la stanga nn0);

```

```

tata->dr= Nod1->st;
Nod1->st = tata;
tata->FactEch = 0;
tata = Nod1;
tata->FactEch = 0;g
else
/*cazul Nod1->FactEch == 0 nu este posibil pentru ca
am avut *H=F; ramane Nod1->FactEch == -1 ca in
gurile 4.12 si 4.136 */
fprintf( Rotatie dubla la stanga nn0);
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;
g
tata->FactEch = 0;
*H = F;ggg
return(tata);g
/*****
/* Functia de listare */
void Listare(struct Nod *Arbore,int Nivel)
fint i;
if (Arbore)
f
Listare(Arbore->dr, Nivel+ 1);
printf( nn0);
for (i = 0; i < Nivel; i+ +)
printf(   );

```

```

printf( %c , Arbore->Info);
Listare(Arbore->st, Nivel+ 1);
g
g
/* Echilibrare in cazul cand subarborele drept
devine mai inalt in comparatie cu cel stang*/
/*****/
struct Nod * EchilibrareDr(struct Nod *tata, int *H)
f
struct Nod *Nod1, *Nod2;
switch(tata->FactEch)
f
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 gura 4.18 a) cu tata==y */
f
printf( Rotatie simpla la stanga mn0);
tata->dr= Nod1->st;
Nod1->st = tata;
if(Nod1->FactEch == 0)
f
tata->FactEch = 1;
Nod1->FactEch = -1;
*H = F;
g
else
f
tata->FactEch = Nod1->FactEch = 0;
g
tata = Nod1;

```

```

g
else
f
printf( Rotatie dubla la stanga nm0);
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;
g
g
return(tata);
g
/* Echilibrare in cazul cand subarborele stang
devine mai inalt in comparatie cu cel drept*/
/*****/
struct Nod * EchilibrareSt(struct Nod *tata, int *H)
f
struct Nod *Nod1, *Nod2;
switch(tata->FactEch)
f
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 gurii 4.18 a) cu tata==e */
f
printf( Rotatie simpla la dreapta nn00);
tata->st= Nod1->dr;
Nod1->dr = tata;
if(Nod1->FactEch == 0)
f
tata->FactEch = -1;
Nod1->FactEch = 1;
*H = F; g
else
f tata->FactEch = Nod1->FactEch = 0; g
tata = Nod1; g
else
/*cazul din gura 4.21 cu tata==e */
f printf( Rotatie dubla la dreapta nn00);
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; g g
return(tata); g
/* 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)
f struct Nod *DNod = R;
if( R->dr != NULL)
f R->dr = Sterge(R->dr, Temp, H);
if(*H)
R = EchilibrareSt(R, H); g
else
f DNod = R;
Temp->Info = R->Info;
R = R->st;
free(DNod);
*H = T; g
return(R); g
/* Sterge element cu cheia respectiva din arbore */
/*****/
struct Nod * StergeElement(struct Nod *tata, char Info, int
*H)
f struct Nod *Temp;
if(!tata) f
printf( Informatia nu exista nn0);
return(tata); g
else f if (Info < tata->Info ) f
tata->st = StergeElement(tata->st, Info, H);
if(*H)
tata = EchilibrareDr(tata, H); g
else
if(Info > tata->Info) f tata->dr = StergeElement(tata->dr,
Info, H);
if(*H)
tata = EchilibrareSt(tata, H); g
else f Temp= tata;
if(Temp->dr == NULL) f
tata = Temp->st;
*H = T;
free(Temp); g
else
if(Temp->st == NULL) f tata = Temp->dr;

```

```

    *H = T;
    free(Temp); g
else
    f Temp->st = Sterge(Temp->st, Temp, H);
    if(*H)
        tata = EchilibrareDr(tata, H); g g g
    return(tata); g
/* Functia principala*/
/*****/

void main()
f int H;
char Info ;
char choice;
struct Nod *Arbore = (struct Nod *)malloc(sizeof(struct Nod));
Arbore = NULL;
printf( 'Tastati 'b' pentru terminare: nn0);
choice = getchar();
while(choice != 'b')
f    ush(stdin);
printf( Informatia nodului (tip caracter: a,b,1,2,etc.): nn0);
scanf( %c ,&Info);
Arbore = Inserare(Info, Arbore, &H);
printf( Arborele este: nn0);
Listare(Arbore, 1);    ush(stdin);
printf( Tastati 'b' pentru terminare: nn0);
choice = getchar(); g    ush(stdin);
while(1) f
printf( 'Tastati 'b' pentru terminare: nn0);
printf( Introduceti cheia pe care vreti s-o stergeti: nn0);
scanf( %c ,&Info);
if (Info == 'b') break;
Arbore = StergeElement(Arbore, Info, &H);
printf( Arborele este: nn0);
Listare(Arbore, 1); g g

```


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. Universitatii Bucuresti, 2005.
- [3] D. E. KNUTH, *The Art of Computer Programming*, vol.1, Reading, Massachusetts, 1969; vol. 3, Addison-Wesley, 1973.
- [4] D. STOILESCU, *Culegere de C/C++*, Edit. Radial, Galati, 1998
- [5] I. TOMESCU, *Data Structures*, Edit. Universitatii Bucuresti, 1998.